# An Advanced Introduction to Reinforcement Learning

C. S. Schroeder

March 10, 2012

**Abstract**

In the first half of this paper I intend to provide an introduction to the topic of Reinforcement Learning within Machine Learning. Only assuming a Bachelor's Level understanding of mathematics and recursion. It is an "advanced" introduction for the second half, which discusses two limitations of the basic reinforcement learning approach - handling large state spaces and handling complex actions - and approaches to solving these problems. This is an edited and reformatted version of an earlier paper, originally written in 2009.

# 1 Introduction

The fundamental problem in reinforcement learning is to understand how agents can learn from punishments and rewards, without given explicit explanation of what it was that they did right or wrong in a given case. It is the task of the agent to infer what was done right or wrong in a given situation by learning from the feedback which is associated with their actions. Reinforcement learning is a natural phenomenon. Understanding how reinforcement learning takes place, and may be replicated, does not always present a serious problem. For instance, it may not be terribly difficult to model the behavior of a rat at the feeder-bar being fed sugar pellets. The more difficult problem is to devise an algorithm by which the rat could learn from its experience that hitting the feeder bar and eating the pellet is detrimental if done too often. That is, the rat must learn the long-term benefit/detriment of hitting the feeder bar at different frequencies. In order for the rat to learn such a thing, it must take into account negative reinforcement far distant from single instances of pellet eating. The problem of accounting for such distant debt (or payoff in the opposite case) is what reinforcement learning promises to solve.

This essay is a brief tour of classical reinforcement learning, in the rigorous form given to it over the past three decades, followed by an analysis of the limitations and attempts to overcome those limitations over the past couple of years. I will not be able to account for every subtlety in the various approaches within classical RL or within the attempts to overcome the limitations, but I hope to present the basic lay of the land and suggest a direction in which research may proceed.

# 2 Classic Reinforcement Learning

Reinforcement Learning finds its roots in a mix of Dynamic Programming and Monte Carlo Simulation. Effectively, it can be viewed as an attempt to overcome the limitations of Dynamic Programming; an attempt that has similarities to both the methods of Monte Carlo Simulation and Dynamic Programming itself. This understanding of the relationship between the three fields is the running theme of Sutton and Barto's book Reinforcement Learning [12]. I will only hope to summarize this understanding here. For the more thorough treatment, the reader is encouraged to study this book by

two leaders in the field. Needless to say, the following presentation of what I call Classic Reinforcement Learning borrows much from this work.

## 2.1 Markov Decision Processes

Dynamic Programming (DP) and Classic Reinforcement Learning alike, address a very specific case of a decision problem. In this decision problem, the agent's goal is to maximize reward. In order to do this, the agent has the following capabilities and limitations:

1. The agent is completely aware of what state they are in;
2. The agent knows what actions they have to choose from in that state;
3. The agent knows an action will always take them to some state at the next time period, i.e. the action will not span multiple time periods;
4. The agent does not know a priori what next state an action will take them to (though a more restricted form of DP assumes they do); and
5. The agent receives rewards (costs) when they enter the next state.

Furthermore the world is characterized by the fact that states, actions, and rewards all interact according to the following equivalence:

$$P(s|s', a) = P(s|s', a, s_{-1}, a_{-1}, s_{-2}, a_{-2}, s_{-3}, a_{-3}, s_{-4}, a_{-4}, ...) \quad (1)$$

That is, the probability of entering a given state s (with its associated distribution of rewards) given the previous state and action is the same as the probability of entering that state given the agents entire history of states and actions. This is the Markov Property which makes this process a Markov Decision Process (MDP). What makes this important is that in order to decide what is best for the agent to do, we do not have to keep track of where it has been; we can simply decide based on where it is at present and where it might go thereafter. Moreover, as indicated, the agent does not have any limitations on knowing what state they are in. Their current state is fully observable, so there will be no uncertainty as to where they are at any given time. What MDPs do not assume, however, is that our agent or ourselves understand what actions will take them to which next states, what the rewards are in those next states, or that they have a clue as to how to act rationally in this process.

## 2.2 Dynamic Programming

So far we have not made any assumptions that are unique to dynamic programming (note that we will assume by DP that we mean a probabilistic DP). MDPs are the context in which RL and DP are assumed to both be applicable. But DP is unique in that it assumes in advance that we have a model of our environment. That is, that we do in fact know $P(s|s', a)$ for every s, s', and a; and moreover, we know R(s) for all s, i.e. the payout of rewards in state s. Given this understanding, Dynamic Programming seeks to figure out the best policy for our agent to take.

What is important to our agent is not simply what next state will provide them with the most reward. What we are interested in is what next state will garner us the most long term reward - or in other words, the next state with the most value. The DP method iteratively improves upon your current policy by making alterations to choose the states with more value according to your current policy valuations. As such, it is a bootstrapping method, and that it works by such a simple method is somewhat amazing on first glance.

The heart of Dynamic Programming is the Bellman Equation for the value of state s under policy $\pi$:

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} P_{s,a,s'}[R(s, a, s') + dV^{\pi}(s')] \tag{2}$$

Where $\pi(s, a)$ is the probability of taking action a, when in state s, according to policy $\pi$; $P_{s,a,s'}$ is the probability of ending up in state s' after taking action a in state s; $R(s, a, s')$ is the reward you would receive in state (after taking action a in s); and $dV^{\pi}(s')$ is the value of s', discounted according to discount rate d. That this understanding of the value of a state is appropriate can be seen most clearly by stripping away the probabilistic factors. If we are considering a deterministic policy with a deterministic environment, then the above equation reduces to:

$$V^{\pi}(s) = R(s, a, s') + dV^{\pi}(s') \tag{3}$$

Where a is the action that you will in fact choose in state s, according to policy $\pi$; and s' is the state that you will in fact end up in when you make this decision. If we expand this equation, we then get the following:

$$V^{\pi}(s) = R(s, a, s') + d[R(s', a', s'') + dV^{\pi}(s'')] \tag{4}$$

$$= R(s, a, s') + dR(s', a', s'') + d^2V^{\pi}(s'') \tag{5}$$

$$= R(s, a, s') + dR(s', a', s'') + d^2R(s'', a'', s''') + ... \tag{6}$$

This being a standard interpretation of value, e.g. the Discounted Cash Flow model within finance, where the discount rate is [1/(1+risk free rate)]. (Of course, the probabilistic model applies a wider range of cases in finance as well). DP works as follows:

1. Create an arbitrary policy $\pi$.
2. Evaluate $\pi$ to find $V^{\pi}$
3. Improve $\pi$ by being greedy with respect to $V^{\pi}$ for every possible state.
4. Check to see if $\pi$ has changed; if so, go to 2; if not, stop.

For a given $\pi$, we can find $V^{\pi}$ according to the following algorithm ( [12] p. 92):

```
Initialize V(s) = 0 for all s;
    Repeat:
      e <- 0
      For each state s,
           v <- V(s)
           V(s) <- Bellman(s, V)
           e <- max(e, |v - V(s)|)
      Until e < delta
 Return V
```

Where "delta" is some small positive number, which all differences must be less than; and Bellman(s, V) is the function V(s) as defined above in 2, and V is here a table of values, where the values in V are used by Bellman to define new values for V. Initially, a state's value will get updated with the rewards that the agent would get in the next state, weighted according to the probability of entering that state. As multiple iterations over the states are made, however, these rewards get discounted and propagated back to earlier and earlier states by the use of $dV(s')$ in the update; this continues until all

states get close enough (according to the choice in delta) to the real value function for $V^{\pi}$.

Note that this method of evaluation is made possible by the fact that we have a model of the environment. Without this information, we would not be able to iteratively sweep through the states to determine the apt updates for $V(s)$. Once we have the updated $V^{\pi}$ in hand, we improve upon our policy with $\pi'$ defined as follows:

$$\pi'(s,a) = 1, a \in \{a : argmax_a \sum_{s'} P_{s,a,s'}[R(s,a,s') + dV^{\pi}(s')]\} \tag{7}$$

and $0, otherwise$ Where we again use our model of the environment, but now use the value function for the policy as well. In general this update is greedy in the sense that it assigns a probability of 1 to the action with the greatest likelihood of taking them to the state with the most value, according to our current policy's evaluation. (To break ties, we can replace 1 with $1/|argmax_a \sum_{s'} P_{s,a,s'}[R(s,a,s') + dV^{\pi}(s')]|$, in order to weight equivalent maximal options equally; in most cases ties will be rare[1]. It turns out that if we just keep improving like this, we will converge to the optimal policy; so when our policy stops changing, we can stop iterating. At this time we define the optimal policy as above, with its optimal value function, $V^*(s)$

A similar method allows us also to converge on a value for any given ¡state, action¿ pair. The value for a state action pair is indicated with $Q(s,a)$. The Bellman optimality equation for $Q^*$ is:

$$Q*(s,a) = \sum_{s'} P_{s,a,s'}[R(s,a,s') + dQ*(s',a')] \tag{8}$$

This equation will be important when we come to consider Q-learning in a later section.

## 2.3 The Curse of Dimensionality

Bellman, for whom our above equation is named, coined the phrase the curse of dimensionality [2], as a depiction of the fact that as the number of dimensions in a state space grows, the number of possible states in that space

[1]This is the extension of the deterministic case presented in [12] to the stochastic case; the legitimacy of this extension is mentioned at (p. 97).

grows combinatorially. This turns out to be a problem in many disciplines, but the problem is pronounced in DP for the simple fact that each time we update V(s) or $\pi(s, a)$ in our above procedures, we have to loop through all the states. Naturally, this is not feasible if the state space is large, in which case we may well do better to sample from the agent's possible trajectories through state space.

If one has a model - as in the cases to which DP applies - a standard means of generating such samples is by Monte Carlo Simulation[2]. Here we start our agent in a given state; we stochastically determine which action the agent will take, according to the current policy $\pi(s, a)$; we stochastically determine what state the agent ends up in next, according to $P(s|s', a)$; then determine their reward in that state, according to R(s'); and we continue like this until the agent reaches a stopstate.

When we reach a stop state, we update an array, Returns(s), for any and all s, by appending the total discounted return that we gathered in that episode, after having visited s the first time. We then update our function V(s) as such:

$$V(s) = AVG(Returns(s)) \tag{9}$$

i.e. the average of the returns-per-episode (after visiting that state the first time in that episode). Then just like the case of DP, we update our policy to favor visiting those states which have a higher V(s). This particular algorithm is called the "first-visit" Monte Carlo method (, p. 113). There are variations on this theme, but the most important point that this brings up is the idea of trading off between exploitation and exploration. In the above algorithm, you will notice, we trace one trajectory through state space before we update our policy. If we were to take our V(s) after one such episode, and update $\pi(s, a)$ in the same manner as we did for DP above, we could actually be stuck with the same non-optimal policy for all eternity! Assume that our V(s) is initialized with 0, for all s (as is common), and assume the rewards are positive and non-zero at each step. Assume, moreover, that in this case, the transitions from state and action to the next state are deterministic, i.e. where $P(s|s', a)$ is only either zero or one. In such cases, if you choose one action, a1, it will take you to state s1; and if you choose another action, a2,

[2]CITATION Bar03 ł1033 Note that although one needs a model to create a simulated trial via Monte Carlo Simulation, the complexity of these models may be significantly less than the models required by DP ( [12] pp. 129-130)

it will take you to another state, s2. But if you chose a1 first, then V(s1) will get a higher value than V(s2); since s2 not being visited at all means V(s2) will still be zero. But if you never visit s2 at all - and you wouldn't if $V(s1) > V(s2)$ and you follow the strict greedy policy - it is certainly hard to determine what the real value of s2 is. Our update of $\pi(s, a)$ in this case could effectively rule out visiting a state we should visit.

It turns out that in order for our algorithm to converge in the limit, it needs to continue to explore all states, indefinitely. A common way to do this is to break any decision point into two; with some probability 1-q, you will choose among those actions which lead you to states with the best V(s) values according to your current estimate; and with probability q, you will choose among all the available actions with uniform probability. This allows you to exploit what you know, with probability 1-q, and explore with probability q. In many cases, q will be dynamic, and take on smaller and smaller values as the number of samples increases; this can lead to quicker convergence, without losing anything in theory.

It is important at this stage to take note of a few details. First, although Monte Carlo Simulation assumes that you have a model of the environment with which to generate samples, the learning method described in this section does not necessitate that the samples come from a Monte Carlo Simulation. If, for instance, you have a MDP that can be broken down into discrete episodes and you have a data set where each record is a record of a distinct episode - whether it be generated or naturally observed - then you can certainly apply the learning method described here in either case. The problem is typically that naturally observed data sets typically provide a small sample size for this method to be effective; this is one thing which reinforcement learning has the promise of overcoming, and may bring with it faster convergence to optimal results when data is plentiful. Secondly, although the method of learning from samples has a clear benefit over needing a complete model, the method traced here has its limitations as well. Perhaps the primary limitation is that it does not provide any "in game" learning. For instance, if we wish to use reinforcement learning to model biological organisms - and indeed, the inspiration for reinforcement learning is largely due to its appeal as a natural form of learning - our organism would do well to learn from reinforcement while it was alive, and not have to wait for the next life (with its forthcoming policy improvement) before it modified its behavior. The ability to learn as you play is something which reinforcement learning also has the promise of providing.

## 2.4 Classic Reinforcement learning

Reinforcement Learning, like the learning discussed in the previous section, is a method of learning from sample experience; but like Dynamic Programming, Reinforcement Learning involves bootstrapping. Unlike DP, the bootstrapping in RL is a means to attain faster convergence to the optimal policy from the available sample experience; and unlike the Monte Carlo Learning methods, RL provides the prospect of achieving "in game" results and faster overall convergence. There are many variations on even the classical forms of RL, but here I will concentrate on only one, called Q-learning.

As with all forms of RL, Q-learning uses the rewards attained in the course of action to guide future action (i.e. reinforce those behaviors that led to rewards); this in itself is not unlike the method of the previous section, but here we actually update the value function on the fly, by turning the Bellman equation into an assignment. Q-learning, as the name suggests, focuses on the function Q(s, a), which was introduced earlier; the Bellman optimality equation is converted into the assignment:

$$Q(s,a) = Q(s,a) + alpha[r + dmax_a Q(s',a') - Q(s,a)] \tag{10}$$

The idea here is that the updated value of Q(s,a) should be the old value of Q(s,a), plus the difference between the value of Q(s, a) that is indicated by the current sample (i.e. $r + dmax_a Q(s', a)$ ) and the current estimate (Q(s, a)), multiplied by the learning rate alpha (to temper changes to Q(s,a)).

This bootstrapping mechanism, allows updates to be made to the value of Q(s,a) on the fly, immediately after a new reward, r, is received. We do not have to wait until the end of an episode to update this value, and can therefore modify our policy using Q(s, a), in the middle of a game. How we modify our policy, of course, has certain restrictions. That is, we must not simply use the greedy strategy of choosing the action $max_a Q(s, a)$, when in any state s. If we want our policy to converge to the optimal policy, we must continue to explore our options. Hence, a mixed strategy of following Q(s, a) and choosing any action whatsoever (as described in the previous section) may be appropriate - though other algorithms are available.

Q-learning is among the class of learning algorithms which are called temporal difference algorithms. They are called this for the fact that the update of the value function is based on the difference between a previous estimate of the value function and a new instance (the sample). The above

algorithm is also among the sub-class of TD(0) algorithms. We can expand the above algorithm to be among the class TD(n), for any n, as follows:

$$Q(s,a) = Q(s,a)+alpha[r_0+dr_1+d^2r_2+...+d^{n-1}r_{n-1}+d^n max_a Q(s_n,a)-Q(s,a)] \quad (11)$$

In this case, we run our update of Q(s, a) n steps after we execute a in s, i.e. after we have reaped all of the rewards in n-step window following our agents decision to a in s; we then add to that the discounted current estimate of $max_a Q(s_n, a)$, where $s_n$ is the state we find ourselves in currently (n steps after being in s). Different values of n can make a significant difference, depending on the problem involved. It is also worth noting that this amounts to a unification of the method in the previous section (a Monte Carlo method) with Temporal Difference methods: if you suppose that an action executed in a state less than n steps from the finish of the game gets updated simply with the discounted rewards accrued after that action (and before the end of the game), then if n is infinity, you effectively have a method similar to that of the previous section[3].

# 3 Two Problems with Classical Reinforcement Learning

While the general framework for reinforcement learning is intuitive and provides significant benefits over the more traditional Dynamic Programming and Monte Carlo approaches to solving MDPs, it comes with limitations of its own. In this section we will address some of those limitations, as well as various attempts which have been made to overcome them.

## 3.1 Dealing with Non-Markov Decision Processes

The classical theoretical results in reinforcement learning rely on the agent/environment instantiating a Markov Decision Process (MDP). This is an inherent limitation, since there are at least two other possibilities which may come into play in real world applications. The first possibility is that the state signal may not be a perfect indicator of the environment, but instead result

[3]Though not exactly: this is an every-visit method, rather than the first-visit method discussed previously; but nonetheless, the idea is very similar

from a stochastic function, which takes the environmental variables as input; such models are called Partially Observable MDPs (POMDPs). Second, it may be that the world just does not have the Markov property; instead, the next state may be the probabilistic result of the immediately prior state, as well as the state at any other time in history; within this truly non-Markov case, there are naturally different classes of situations, depending on how far back in time is relevant; but provided all the past needs to be in the current state representation, we likely have an intractable problem you cannot make into a standard MDP.

We may want to handle POMPDs and non-Markov processes separately. However, the non-Markov case may often be understood as simply an instance of a POMDP. Note that if the relevant history always falls within a constant sized window of time, we can make a MDP out of this process. We can do this by simply including the history of the original state signal over this window, in a new state signal that we will use to make decisions going forward. Often, however, we will not want to include all of these values in the signal, in order to minimize complexity. In that case, our state signal may include some variables which are a probabilistic function of the history, but not the whole window of history - that is, you have a POMPD. Furthermore, the crux of the problem in handling POMPDs is that we have to incorporate our history of observations for the sake of judging our current state - and thereby making a better decision as to what to do. So fundamental to our problem in either the POMPD or the pure Non-Markov case is the incorporation of history into our calculations to help determine what will happen next [4]. As a result, our concern here will be how to incorporate the history of the state signal to provide better predictions in general.

As in the case of DP (with the full model of the MDP), if one has a full model of the POMDP, there are precise results that can be achieved. The results are achieved in a fashion similar to Dynamic Programming because if we take the belief state with respect to the environment as the state signal itself, we effectively have a new MDP ( [6]). Although belief states involve high complexity and continuity, they can be simplified in various ways [5], which may make them feasible as a method in large applications without a model. An alternative is to incorporate history explicitly; such was the

[4]It is worth noting that you can roughly understand the difference in this way: in the Non-Markov case, we try to use history to extend our experience in time; while in the case of POMPDs, we use history to extend our experience in space.

[5]For an example, see [14]

approach of McCallum. From 1994-1996 McCallum presented a number of different methods to handling memory; these various approaches culminated in a single, synthesized approach, as represented in the U-Tree algorithm ( [7]), which will be our focus here [6]

The basics of the U-Tree algorithms are not hard to understand; but the details are not so simple. We will concentrate on the basics, and discuss some of the difficulties in the details. At a very high level, the algorithm compares a window of the recent history of the state, including the present state, to previous examples that occurred within that window (including the "present" at the time); it will find the examples which match the best (within some threshold) to the current situation. It will treat these examples themselves as the relevant "current state", s; and determine the appropriate action to take, according to the action which maximizes the Q(s,a) function.

The relevant components in the history and current signal are represented as nodes in a tree. The current instance will filter down to a leaf in the tree pending the tests at each node; for instance, a node may check to see if two time steps prior, the variable was ¡ or ¿ or = to 0, and may branch appropriately. It will go through a series of such tests, until it reaches a leaf; the instances stored at this leaf node will be the instances which are relevantly similar to the current instance; and altogether will represent the state, s, which is under consideration when we choose the action $max_a Q(s, a)$.

The calculation of this value function, Q(s,a) takes place according to "one step of dynamic programming on the Q-values" [7] as follows:

$$Q(s, a) = R(s, a) + P(s, a, s')max_{a'}Q(s', a') \tag{12}$$

Where, R(s, a) is calculated as the average among the instances stored at this node, of the rewards attained taking action a. P(s,a,s') is calculated as the count calculated as:

"the number of instances in this node, which took action a, whose successor is stored in the leaf-node s'"
---
"the number of instances in this node, which took action a"

[6]This paper is presumably a partial report on his thesis (which I could not get my hands on); It should be said that McCallum's approach promised to do more than simply handle memory; it promises to handle selective perception, i.e. the problem of choosing when to act so as to gather more information before proceeding on a course (e.g. going out of the way to look for a landmark in a maze). I will have little to say about this subtle issue here; though clearly, McCallum does provide one approach to this.

[7] [7] section 3.2 step 3

Both of these values can be stored and updated upon the arrival of a new instance (after the action is taken, and next state observed). This much is fairly straightforward. What is more difficult to understand is how the tree is to be formed in the first place. These are the details mentioned above; and they are not trivial.

At the start, the tree is empty, but for the root-node. After some pre-specified number of steps, k (and then again after successive k), the algorithm evaluates whether it is beneficial to split a given leaf node, on what attributes, and at which point in time (i.e. the attribute at its present state or at its state some steps prior to the current time index). This act of splitting is actually a matter of deciding whether to add branches to the "on line" tree; these branches currently reside "in the fringe", as parts of a parallel, deeper tree not used for online categorization. The Kolmogorov-Smirnov test is used to judge whether branches on the fringe come from different distributions, and hence should be added to the online tree for the sake of better predicting value. Most of the subtleties involve computational techniques for judging the fringe[8]. McCallum's tests with the algorithm indicate its relevance; and the algorithm remains very important today -finding application in some of the most recent work on hierarchical reinforcement learning [9], which is what we turn to now.

## 3.2 Return of the Curse

It is clear that our work to incorporate history to make better decisions did little to help with the problem of complexity. The need to keep track of history only added dimensions to the fold. The curse gets no better as we try to scale to larger applications from which we expect more. In fact, in larger applications, we have not only issues regarding the complexity of the state; another aspect to the curse is the complexity of actions. Actions can be complex in both their span over space (concurrent actions) and their span over time (plans). To deal with complexity in time, researchers have introduced Hierarchical Reinforcement Learning. The basic idea is that an agent will make decisions to execute sub-policies which can then execute their course going forward without the need for incremental decisions (and thereby computing resources) along the way; in this way, they will not need

[8] [7]section 3.2

[9] I will address HRL in general here; the applications of McCallum's algorithms are to be found in ( [4]) and ( [1])

to make complex decisions at each step - or at least, the complexity will be decreased. These sub-policies effectively help form a hierarchy, with a master policy at the top, primitive actions at the bottom, and any amount of sub-policies and sub-subpolicies, etc. in between. If we are to handle decisions over sub-policies in a uniform manner with actions themselves, we must understand actions/sub-policies as capable of taking time to execute, and not simply executed between states; such a situation raises the need to introduce another sort of process: the semi-Markov Decision Processes (SMDPs).

Following ( [13]) in their initial presentation of the idea [10], such sub-policies are called options. An option consists of a set of states in which the option can be chosen, S, the policy that will be followed should the option be chosen, $\pi$, and a stochastic termination condition, $\beta(s)$, which determines the probability that the option will terminate when one enters a given state, s. In their simplest form, such options are Markov Options, however, a more flexible form of option is the Semi-Markov Option; the latter may consider the history of the states and actions which have taken place since the initiation of the option. Semi-Markov options are necessary in particular for options which have a set expiration after some time[11]. The general form of the update for our new Q function, resembles the original update closely:

$$Q(s,o) = Q(s,o) + alpha[r + d^k max_{o'} Q(s',o') - Q(s,o)] \qquad (13)$$

The only substantive change here is the inclusion of k as the exponent of the discount value. This exponent is necessary because r is the sum of discounted values during the course of o on this trial. We also assume here that o is restricted to a certain set of options; in particular that o is selected from the set O of available options. It can be shown that this update rule will converge to ${Q_O}^*(s,o)$, which is the value function for the optimal policy restricted to options in O within an SMDP[12].

[10]The work of ( [8]) and ( [3]) beg mention here; a synopsis of these different approaches can be found in ( [1]) pp. 54-61. I chose the "options" approach because I believe it the most general and gentle introduction to HRL.

[11]This point is made in ( [13]) on page 7 (which may actually be p.188 in the journal); these authors fail to make the simple point that if we allow for the agent to use state as external memory, this can be avoided with a simple counter variable; using external memory in RL (at least for other purposes) has naturally been proposed elsewhere ( [9]).

[12]Page 9 in the copy I have; likely 190 in the journal.

A further extension of this work is to allow the option to not only terminate according to $\beta(s)$, but moreover, allow the agent to interrupt this option in order to start along a different course of action. The scheme for doing this is also presented in ( [13]). Furthermore, important work is being done to deal with complexity in space within the framework of Hierarchical learning - i.e. concurrency - as well. Researchers Rohanimanesh and Mahadevan[13], among others, have introduced a framework which complements the hierarchical solution to complexity in time; and a natural extension of such concurrency is multi-agent systems, which are also being addressed[14].

We will not pursue these topics here, though they will clearly be important in many cases. Instead, we will simply raise some issues within the general framework of hierarchical learning, which may be the most important to resolve first. These issues are quite simple to understand: if we have an agent who is capable of various actions and capable of experiencing various things, we would like for this agent to be able to learn options; not simply in the sense of when to choose a given option, but in the sense of being able to abstract the option for themselves, in order to be able to choose it going forward. Moreover, we would like, if possible, to understand this abstraction in a uniform manner with state abstraction; or at least, understand how these two forms of abstraction work together[15].

One way for the agent to learn such options is reward shaping. Reward shaping hands out rewards for reaching states that are deemed as "progress" toward the actual goal; something like rewarding a checkers player for jumping an opponent and taking a piece. It is not the final goal of winning the game, but its progress. Unfortunately, in some cases, it can lead the agent to exploit a "glitch" without ever getting closer to the real goal[16]. And of course, even if such reward shaping; we would like a more organic solution to option discovery; since this approach still rests on "the hand of God" to shape the rewards. Despite this problem being well known, progress remains

[13]Cited in ( [1]) page 61ff

[14]See ( [1]) page 64ff

[15]As Sutton, et al, acknowledges: "Key issues such as transfer between subtasks, the source of sub-goals, and integration with state abstraction remain incompletely understood"( [13]).

[16]( [10]) mention the case of programming an automated soccer player to be rewarded for "getting possession" of the ball when trying to score a goal, in which the player "learns" to vibrate against the ball (p. 787); the issue of reward shapes may be understood as the fundamental problem with the human condition.

uncertain; there are a few attempts to address the issue, but I do not think enough progress is being made [17].

# 4 Conclusion: Toward a Full Treatment

Clearly the framework of Semi-Markov Options, by definition, allows for the influence of history, so within the options framework, it should seem clear that we can use the U-Tree algorithm to maintain this history. This in fact is what ( [5]) have done. There approach is to build a distinct U-Tree for each option. The leaves of these trees then represent the relevant states for the option, on the basis of which they can decide what to do when that option is in use. Furthermore, they allow for intra-option learning - that is, the ability for one option to learn from the results of another, "When several options operate in the same part of state space and choose among the same actions"(p. 4). This enhances the effectiveness of the algorithm significantly (p. 6). As they understand it, there are a set of options, with their respective U-Trees and above the options sits a single policy. The U-Tree dynamics apply entirely within a single option; in particular, the adding of nodes to the U-Tree from the "fringe" only takes place within a given option, since the policy itself does not represent a tree. But we could clearly imagine handling the policy itself as the root of a tree, with the options underneath. In such a case, we could understand there to be two tasks involved in the construction of one overall tree: the splitting of leaves to add nodes from the fringe, as usual; and the splitting of leaves to add options. Option nodes are only different in the sense that once they are decided upon, they will serve as entry points for the classification of new instances until control is either

1. kicked back to the option above them (or the overall root "policy") in the tree according to $\beta(s)$ or

2. the option above requests termination ( [13], p.19).

In this framework, the issue of learning options becomes the issue of when to create option nodes. This does not present a method describing when one

[17]( [14] ), in their conclusion, note that they intend to approach this problem, at least within a more minimal scope; but there is no clear follow-up on this promise; ( [1]) p. 71, list some of the attempts and make it clear that there are serious limitations on these methods.

should create such an option node and stands as a minor addition to what I have found in the literature. But I think it at least puts structure around the problem, which is intuitive to grasp. Moreover, with this structure, we should be able to see that the treatment of state abstraction and the treatment of action abstraction will fall within a single framework; which was one of our desiderata. But naturally it remains to be seen if this framework is rich enough to encapsulate the instances we are interested in; whether an apt criteria for adding option nodes can be determined; and whether the algorithms involved can be efficient enough for practical purposes. At any rate, this is what I hope to determine through further search of the literature and my own investigations in reinforcement learning.

## 4.1 Acknowledgments

In general, most of the first half is distilled from [12], which is the best book for the basics of RL. The second half largely had [1] as its guide through the more recent literature. The book Handbook of Learning and Approximate Dynamic Programming [11] undoubtedly contains much that I should know, and would likely have been significant to the content of this paper, had it been available to me earlier.

# References

[1] Barto, A. G., Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. Discrete Event Dynamic Systems: Theory and Applications, 13 , 41-77.

[2] Bellman, R. (1961). Adaptive Control Processes. Princeton: Princeton University Press.

[3] Dietterich, T. G. (2000). Heirarchical reinforcement learning with the maxq value function decomposition. Journal of Artificial Intelligence Research , 227-303.

[4] Hernandez, N., Mahadevan, S. (2001). Hierarchical memory-based reinforcement learning. Proceedings of Neural Information Processing System.

[5] Jonsson, A., Barto, A. (2001). Automated State Abstraction for Options Using the U-tree Algorithm. Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference (pp. 1054- 1060). Cambridge, MA: MIT Press.

[6] Kaelbling, L., Littman, M., Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. Artificial Intelligence vol. 101 , 99-134.

[7] McCallum, A. K. (1996). Learning to use selective attention and short-term memory n sequential tasks. From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (pp. 315-24). MIT Press.

[8] Parr, R., Russell, S. (1998). Reinforcement learning with hierarchies of machines. Advances in Neural Information Processing: Proceedings of the 1997 Conference. Cambridge, MA: MIT Press.

[9] Peshkin, L., Meuleau, N., Kaelbling, L. P. (1999). Learning Policies with External Memory.

[10] Russell, S., Norvig, P. (2003). Artificial Intelligence: A Modern Approach. Upper Saddle River, NJ: Pearson Education Inc.

[11] Si, J., Barto, A., Powell, W. B., Wunsch III, D. (2004). Handbook of Learning and Approximate Dynamic Programming. Piscataway, NJ: IEEE Press.

[12] Sutton, R. S., Barto, A. G. (1998). Reinforcement Learning: an introduction. Cambridge: MIT Press.

[13] Sutton, R. S., Precup, D., Singh, S. (1999). Between MDPS and Semi-MDPS: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence, 112 , 181-211.

[14] Theocharous, G., Kaelbling, L. P. (2004 (NIPS-03)). Approximate Planning in POMDPS with Macro-Actions. Advances in Neural Information Processing Systems 16, Vancouver.