Reinforcement Learning with a Hierarchy of Abstract Models

Satinder P. Singh
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
singh@cs.umass.edu

Abstract

Reinforcement learning (RL) algorithms have traditionally been thought of as trial and error learning methods that use actual control experience to incrementally improve a control policy. Sutton's DYNA architecture demonstrated that RL algorithms can work as well using simulated experience from an environment model, and that the resulting computation was similar to doing one-step lookahead planning. Inspired by the literature on hierarchical planning, I propose learning a hierarchy of models of the environment that abstract temporal detail as a means of improving the scalability of RL algorithms. I present H-DYNA (Hierarchical DYNA), an extension to Sutton's DYNA architecture that is able to learn such a hierarchy of abstract models. H-DYNA differs from hierarchical planners in two ways: first, the abstract models are learned using experience gained while learning to solve other tasks in the same environment, and second, the abstract models can be used to solve stochastic control tasks. Simulations on a set of compositionally-structured navigation tasks show that H-DYNA can learn to solve them faster than conventional RL algorithms. The abstract models also serve as mechanisms for achieving transfer of learning across multiple tasks.

Introduction

Planning systems solve problems by determining a sequence of actions that would transform the initial problem state to the goal state. In a similar manner, problem-solving agents or controllers[1], that have to learn to control an external environment, incorporate planning when they use a model of the control problem to determine an action sequence, or an open loop control policy, prior to the actual process of controlling the environment. Recent work on building real-time controllers has highlighted the shortcomings of planning algorithms: their inability to deal with uncertainity, stochasticity, and model imperfection without extensive recomputation. Some researchers have proposed reactive controllers (e.g., Schoppers 1987) that dispense with planning altogether and determine actions directly as a function of the state or sensations. Others (e.g., Dean & Boddy 1988) have proposed control architectures that use anytime algorithms, i.e., use the results of partial planning to determine the action in a given state.

Sutton (1991) has noted that reactive controllers based on reinforcement learning (RL) can plan continually, caching the results of the planning process to incrementally improve the reactive component. Sutton's (1990) DYNA architecture is one such controller that learns a control policy as well as a model of the environment. Whenever time permits, simulated experience with the model is used to adapt the control policy (also see Barto et al. 1991). As noted by Sutton (1991), the computation performed by the RL algorithm on simulated experience is similar to executing a one-step lookahead planning algorithm. The difference between traditional planning algorithms and RL is that in RL the results are cached away into an evaluation function that directly and immediately affects the control policy.

The inability of RL-based controllers to scale well to control tasks with large state or action spaces has limited their application to simple tasks (see Tesauro 1992 for an exception). An approach to scaling RL algorithms can be derived from the research on hierarchical planning (e.g., Sacerdoti 1973). Most hierarchical planners assume access to a hierarchy of abstract models of the problem state-space. They first plan in the highest level, and then move down the hierarchy (and if necessary back up) to successively refine the abstract plan until it is expressed solely in terms of primitive actions. Although using abstract models is not new to RL (e.g., Chapman & Kaelbling 1991), such research has focussed on abstracting structural detail. I present a RL-based control architecture that learns a hierarchy of abstract models that, like hierarchical planners, abstract temporal detail.

[1] In this paper I will use the terms agent and controller interchangeably.

Scaling Reinforcement Learning Algorithms

Unlike planning-based controllers, RL-based controllers are embedded in an optimal control framework (Barto et al. 1990). Thus, the RL agent has to learn a sequence of actions that not only transforms an external dynamic environment to a desired goal state[2], but also improves performance with respect to an objective function. Let S be the set of states of the environment and $\mathcal{A}_1$ be the set of primitive actions[3] available to the agent in each state. In this paper, I focus on RL agents that have to learn to solve Markovian Decision Tasks (MDTs), where at each time step t the agent observes the state of the environment, s_t, and executes an action, a_t. As a result, the agent receives payoff R_t and the state of the environment changes to s_{t+1} with probability $P_{s_t,s_{t+1}}(a_t)$. The objective function, J, considered in this paper is the discounted sum of payoff over an infinite horizon, i.e., $J(i) = \sum_{t=0}^{t=\infty} \gamma^t R_t$. The discount factor, $0 \leq \gamma \leq 1$ causes immediate payoffs to be weighted more than future payoffs. A closed loop control policy, which is a function assigning actions to states, that maximizes the agents objective function is an optimal control policy.

If a model of the environment is available, i.e., the transition probabilities and the payoff function are known, conventional dynamic programming (DP) algorithms (e.g., Ross 1983) can be used to find an optimal policy. If a model of the environment is not available to the agent, RL algorithms that approximate DP, such as Sutton's (1988) temporal differences (TD), can be used to approximate an optimal policy. An essential component of all DP-based algorithms[4] for solving MDTs is determining an optimal value function $V^* : S \rightarrow \Re$, that maps states to scalar values such that in each state the actions that are greedy with respect to V^* are optimal. DP-based RL algorithms use repeated experience at controlling the environment to incrementally update $\hat{V}$, an estimate of the optimal value function.

The basic algorithmic step common to most DP-based learning algorithms is that of a "backup" in which the estimated value of a successor state is used to update the estimated value of the predecessor state. For example, the TD algorithm uses the state transition at time t to update the estimate as follows: $\hat{V}_{t+1}(x_t) = (1.0-\alpha)\hat{V}_t(x_t)+\alpha[R_t+\gamma\hat{V}_t(x_{t+1})]$, where α is the learning rate parameter. Bertsekas and Tsitsiklis (1989) show that under certain conditions the order of the backups over the state space is not important to the convergence of some DP algorithms (also see Barto et al. 1991) to the optimal value function. The rate of convergence, though, can differ dramatically with the order of the backups, and a number of researchers have used heuristics and domain knowledge to change the order of backups in order to accelerate learning of the value function (e.g., Kaelbling 1990; Whitehead 1991).

While the inability of RL algorithms to scale well involves many issues (Singh 1992a; Barto & Singh 1990), the one of relevance to this paper is that most RL algorithms perform backups at the scale of the primitive actions, i.e., actions executable in one-time step in the real world. At that fine a temporal scale problems with large state spaces can require too many backups for convergence to the optimal value function. To do backups at longer time scales requires a model that makes predictions at longer time scales, i.e., makes predictions for abstract actions that span many time steps in the real world.

One way to abstract temporal detail would be to simply learn to make predictions for all possible sequences of actions of a fixed length greater than one. However, the combinatorics of that will outweigh any resulting advantage. Furthermore, it is unlikely that there is a single frequency that will economically capture all that is important to predict. In different parts of the state space of the environment-model, "interesting" events, i.e., events that merit prediction, will occur at different frequencies. Any system identification technique that models the environment at a fixed frequency will be inefficient as compared to a system identification technique that can construct a variable temporal resolution model (VTRM), i.e., a model with different temporal resolutions in different parts of the state space.

Learning Abstract Models for Multiple Tasks

If the agent is to simply learn to solve a single task, the computational cost of constructing a VTRM may not be worthwhile (see Barto and Singh 1990). My approach to the scaling problem is to consider problem-solving agents that have to learn to solve multiple tasks and to use repeated experience at solving these tasks to construct a hierarchy of VTRMs that could then be used to accelerate learning of subsequent tasks. Besides, building sophisticated autonomous agents will require the ability to handle multiple tasks/goals (Singh 1992b). Determining the useful abstract actions for an arbitrary set of tasks is difficult, if not impossible.

In this paper, I consider an agent that has to learn to solve a set of undiscounted ($\gamma = 1$), compositionally-structured MDTs labeled $T_1, T_2, \ldots, T_n$. Each task requires the agent to learn the optimal path through a sequence of desired states. For example, task $T_i = [x_1 x_2 \cdots x_m]$, where $x_j \in S$ for $1 \leq j \leq m$. Task

[2] In some optimal control problems the goal is to follow a desired state trajectory over time. I do not consider such tasks in this paper.

[3] For ease of exposition I assume that the same set of actions are available to the agent from each state. The extension to the case where different sets of actions are available in different states is straightforward.

[4] Algorithms based on policy iteration are an exception.

T_i requires the agent to learn the optimal trajectory from any start state to x_m via intermediate states $x_1, x_2, \ldots, x_{m-1}$ in that order. The MDTs are compositionally structured because they can be described as a temporal sequence of simpler tasks each of which is an MDT in itself. Thus, the task of achieving desired intermediate state x optimally is the *elemental* MDT $X = [x]$ defined over the same environment. Without loss of generality, I will assume that the n composite MDTs are defined over a set of N intermediate states labeled $x_1, x_2, \ldots, x_N$. Equivalently, the n composite MDTs are defined over N elemental MDTs labeled $X_1, X_2, \ldots, X_N$.

The payoff function has two components: $C(x, a)$, the "cost" of executing action a in state x, and $r(x)$, the "reward" for being in the state x. It is assumed that $C(x, a) \leq 0$, and is independent of the task being performed by the control agent, while the reward for being in a state will in general depend on the task. For task T_i, the expected payoff for executing action a in state x and reaching state y is $R_i(x, a, y) \doteq r_i(y) + C(x, a)$. Further, I assume that $r_i(y) > 0$ iff y is the final goal state of task T_i; $r_i(y) = 0$, elsewhere.

For a set of compositionally-structured MDTs, determining the abstract actions for the hierarchy of VTRMs is relatively straightforward. The abstract actions for the second level[5] VTRM should be the elemental MDTs $X_1, X_2, \ldots, X_N$. Thus, the abstract action X_1 would transform the environment state to $x_1 \in S$. The expected payoff associated with this abstract action can be acquired using experience at solving MDT X_1. Note that these abstract actions are a *generalized* version of macro-operators (Iba 1989; Korf 1985) because unlike macros which are open loop sequences of primitive actions that transform the initial state to a goal state and can handle only deterministic tasks, the abstract actions I define are closed loop control policies that transform the environment from any state to a goal state and can handle stochastic tasks. Furthermore, unlike macro-operators, these abstract actions are embedded in an optimal control framework and could be learned incrementally.

Consider two levels of the hierarchy of VTRMs: M-1, the lowest level VTRM whose action set $\mathcal{A}_1$ is the set of primitive actions, and M-2, the second level VTRM whose action set $\mathcal{A}_2 = \{X_1, X_2, \ldots, X_N\}$. The set of states remains S at all levels of abstraction. Note, that predicting the consequences of executing an abstract action requires learning both the state transition for that abstract action and its expected payoff. Under the above conditions the following restatement of a proposition, proved in Singh 1992a, is true:

Proposition: If the abstract environment-model (M-2) is defined with the abstract actions $X_1, X_2, \ldots, X_N$, and the costs assigned to the abstract actions are those that would be incurred under the optimal policies for the corresponding MDTs, then for all composite tasks T_i, $\forall s \in S$, $V_i^2(s) = V_i^*(s)$.

V_i^2 is the value function for task T_i that is learned by doing DP backups exclusively in the abstract model M-2, and V_i^* is the optimal value function for task T_i. The above proposition implies that after learning the abstract model, backups can be performed in the abstract model alone to learn the optimal value function.

Hierarchical DYNA

The Hierarchical DYNA (H-DYNA) architecture is an extension of Sutton's (1990) DYNA architecture. H-DYNA, shown in Figure 1, consists of a hierarchy of VTRMs, a policy module for each level of the hierarchy, and a single evaluation function module. Note that the actual environment itself is shown in parallel with the M-1 to emphasize their interchangeability. The evaluation module maintains an estimate, $\hat{V}_i$, of the optimal value function for each task T_i. Let the action set for the VTRM at level i be denoted $\mathcal{A}_i$. For task T_i, let $\hat{R}_{ij}(x, a_j, y)$ denote the estimate of the expected payoff for executing action $a_j \in \mathcal{A}_j$ in state x and reaching state y. For the VTRM at level one the backups will be over state pairs connected by primitive actions, and the payoffs are available directly from the environment. For VTRMs at levels > 1, the payoffs associated with the abstract actions will have to be learned over time using experience with the tasks corresponding to the abstract actions. The policy module at level j keeps a weight function for each task T_i, $w_{ij} : S \times \mathcal{A}_j \rightarrow \Re$. For level j, the probability of executing action a for task T_i is:

$$P_j(i, a) = \frac{e^{w_{ij}(x,a)}}{\sum_{a' \in \mathcal{A}_j} e^{w_{ij}(x,a')}},$$

for $x \in S$ and $a \in \mathcal{A}_j$

When solving task T_i, the primitive control actions to be executed are always determined by the policy module at level 1. Whenever time permits, a backup is performed using experience in any one of the hierarchy of environment models. For both real (level = 1) and simulated experiences on task T_i the evaluation function and the policy module involved (say level j) are updated as follows:

$$\begin{aligned}
\hat{V}_i(x) &= \hat{V}_i(x) + \alpha\{\hat{R}_{ij}(x, a, y) + \gamma \hat{V}_i(y) - \hat{V}_i(x)\} \\
w_{ij}(x, a) &= w_{ij}(x, a) + \alpha[\hat{R}_{ij}(x, a, y) + \gamma \hat{V}_i(y) \\
&\quad - \hat{V}_i(x)],
\end{aligned}$$

where $a \in \mathcal{A}_j$ transforms state $x \in S$ to state $y \in S$ and results in a payoff of $\hat{R}_{ij}(x, a, y)$. For real experiences, the task T_i is automatically the current task being performed by the agent and the state x is the

[5] Given the recursive nature of the definition of composite tasks, the abstract actions for levels higher than two can be defined as the composite tasks themselves.

current state. However, for simulated experience in the abstract models, an arbitrary task and an arbitrary state can be chosen. If there is no data available for the chosen state-task pair in the abstract model, no backup is performed.

In addition, for every real experience, i.e., after executing a primitive action, the transition probabilities for that action in M-1 are updated using a supervised learning algorithm. Cumulative statistics are kept for all the states visited while performing the task T_i in the real environment. When the agent finishes task T_i, i.e., when the final state for task T_i is achieved, all the environment models that have that task as an abstract action are updated. For example, after the agent finishes elemental task X_1, the abstract model M-2 is updated because $X_1 \in \mathcal{A}_2$. See Appendix for details.

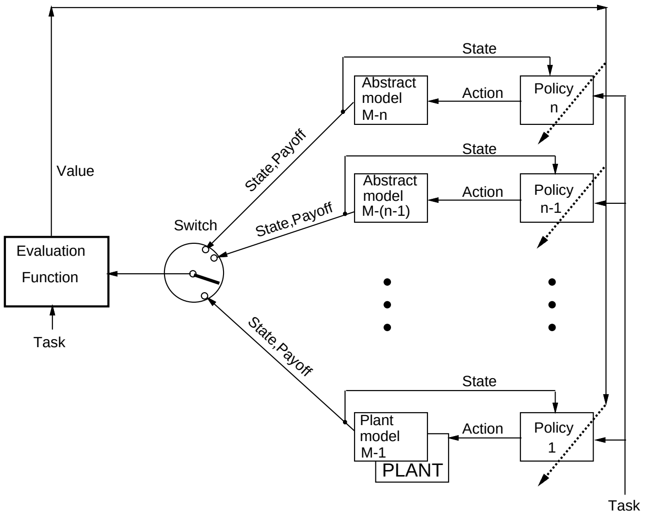

Figure 1: The Hierarchical Dyna (H-DYNA) architecture.

Multiple Navigational Tasks

I illustrate the advantages of the H-DYNA architecture on a set of deterministic navigation tasks. Figure 2 shows an 8×8 grid room with three goal locations designated A, B and C. The robot is shown as a filled circle and the filled squares represent obstacles. In each state the robot has 4 actions: UP, DOWN, LEFT and RIGHT. Any action that would take the robot into an obstacle or boundary wall does not change the robot's location. There are three elemental tasks: "visit A", "visit B", and "visit C", labeled $[A]$, $[B]$ and $[C]$ respectively. Three composite tasks: $T_1 = [AB]$, $T_2 = [BC]$, and $T_3 = [ABC]$ were constructed by temporally concatenating the corresponding elemental tasks. It is to be emphasized that the objective is not merely to find any path that leads to the goal state, but to find the least-cost path. The six different tasks, along with their labels, are described in Table 1.

Table 1: Task Set

Label	*Description*	*Decomposition*
$[A]$	visit A	$[A]$
$[B]$	visit B	$[B]$
$[C]$	visit C	$[C]$
T_1	visit A and then C	$[AC]$
T_2	visit B and then C	$[BC]$
T_3	visit A, then B and then C	$[ABC]$

The cost associated with all the state-action pairs was fixed at -0.25 for all tasks. For each task a value of 1.0 was associated with the final goal state for that task. Thus, no intermediate reward was provided for successful completion of subtasks. In the simulations, I will consider only two levels of the hierarchy, i.e., VTRMs M-1 and M-2.

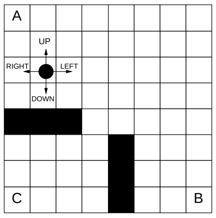

Figure 2: The Grid Room: See text for details.

Simulation 1

This simulation was designed to illustrate two things: first, that it is possible to solve a composite task by doing backups exclusively in M-2, and second, that it takes fewer backups to learn the optimal value function by doing backups in M-2 as compared to the number of backups it takes in M-1. To learn M-2, I first trained H-DYNA on the three elemental tasks $[A]$, $[B]$ and $[C]$. The system was trained until M-1 had learned the expected payoffs for the primitive actions and M-2 has learned the expected payoffs for the three elemental tasks. This served as the starting point for two separate training runs for task T_3.

For the first run, only M-1 was used to generate information for a backup. For the second run the same learning parameters were used, and only M-2 was used to do the backups. To make the conditions as similar as possible for the comparison, the order in which the states were updated was kept the same for both runs by choosing predecessor states in a fixed order. After each backup, the absolute difference between the estimated value function and the previously computed optimal value function was determined. This absolute error was summed over all states for each backup and then averaged over 1000 backups to give a single data point. Figure 3 shows the learning curves for the two runs. The dashed line shows that the value function for the second run converges to the optimal value function.

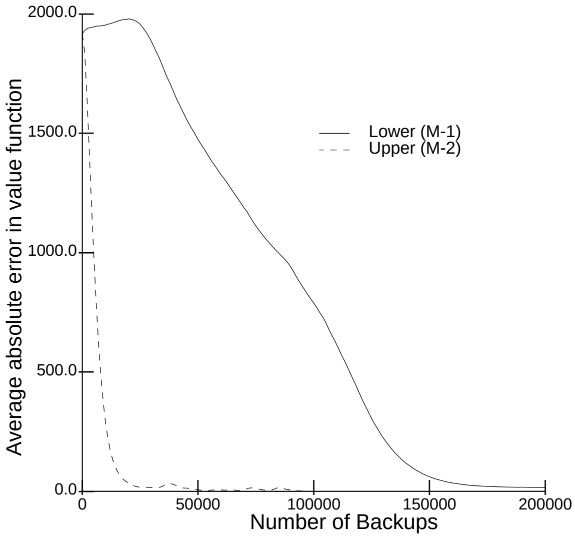

Figure 3: Learning curves: See text for details

The two curves show that it takes far fewer backups in M-2 than M-1 for the value function to become very nearly-optimal.

Simulation 2

This simulation was conducted on-line to determine the effect of increasing the ratio of backups performed in M-2 to the backups performed in the real world. The robot is first trained on the 3 elemental tasks for 5000 trials. Each trial started with the robot at a randomly chosen location, and with a randomly selected elemental task. Each trial lasted until the robot had either successfully completed the task, or until 300 actions had been performed. After 5000 trials H-DYNA had achieved near-optimal performance on the three elemental tasks. Then the three composite tasks (See Table 1) were included in the task set. For each trial, one of the six tasks was chosen randomly, the robot was placed in a random start state and the trial continued until the task was accomplished or there was a time out. The tasks, T_1 and T_2 were timed out after 600 actions and the task T_3 after 800 actions.

For this simulation it is assumed that controlling the robot in real-time leaves enough time for the agent to do n backups in M-2. The purpose of this simulation is to show the effect of increasing n on the number of backups needed to learn the optimal value function. No backups were performed in M-1. The simulation was performed four times with the following values of n: 0, 1, 3 and 10. Figure 4 shows the results of the four different runs. Note that each backup performed in M-2 could potentially take much less time than a backup performed in the real world. Figure 4 displays the absolute error in value function plotted as a function of the number of backups performed. This results of this simulation show that even when used on-line, backups performed in M-2 are more effective in reducing the error in the value function than a backup in the real world.

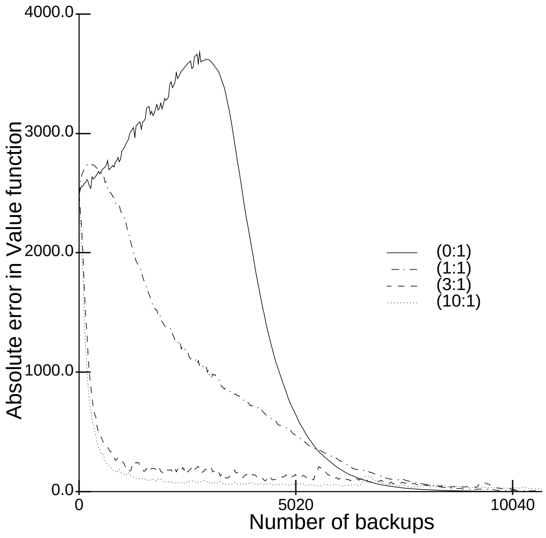

Figure 4: On-line Performance: See text for details.

Discussion

The idea of using a hierarchy of models to do more efficient problem solving has received much attention in both the AI and the control engineering community. H-DYNA is related to hierarchical planning techniques in that search for a solution is conducted in a hierarchy of models. However unlike hierarchical planning, where the models are given beforehand and the search is conducted off-line and sequentially, H-DYNA learns the model using on-line experience and can conduct the search in parallel. While the connection between DP backups and one-step lookahead planning was first emphasized by Sutton (1991) in his DYNA architecture, H-DYNA takes this one step further by demonstrating that doing backups in an abstract model is similar to multi-step planning in general and hierarchical planning in particular. H-DYNA as a hierarchical control architecture has the advantage of being continually "reactive" (much like DYNA) and at the same time performs deep lookahead searches using the abstract models.

The use of compositionally-structured tasks made the problem of figuring out the "interesting" subtasks simple. In addition the optimal solutions to composite tasks could be expressed in terms of the optimal solutions to the elemental tasks. Both of the above conditions will not be true for a more general set of MDTs. However, it may still be possible to discover significant "landmark" states using experience at solving multiple tasks in an environment and to use these states as goal states for the abstract actions to form VTRMs. Doing RL backups in such VTRMs could then quickly lead to near-optimal solutions to new tasks. Such solutions could then be optimized using further experience at solving that task. Further research is needed to test these intuitions.

Acknowledgments

I thank Andy Barto, Rich Sutton, Steve Bradtke, Richard Yee, and Vijay Gullapalli for their guidance and help. This material is based upon work supported by the Air Force Office of Scientific Research, Bolling AFB, under Grant AFOSR-89-0526 and by the National Science Foundation under Grant ECS-8912623.

Appendix

Learning abstract environment models

I keep a list of all states visited in the real world while performing a elemental task T_i. At each time step, the list is checked to see if the current state is in the list, if so, its cumulative cost is set to zero, else the current state is added into the list with cumulative cost zero. The current payoff is added into the cumulative payoff of all the states in that list. When the goal state is reached, the next state for all the states in the list is known as well as their cumulative payoff. As the policy evolves over time the cumulative payoff will change, but the next state will remain fixed. In this method the list can grow to contain the entire state set. Furthermore, searching for the current state is expensive, though efficient data structures (Union-Find) algorithms can be used. This method will be infeasible for large state sets.

A more incremental method is to fix the list size to, say, k, and then only keep the last k states in that queue and manage it in a FIFO manner. Thus when a state x is moved out of the queue, the next state for x is set to be the next state stored for the current state in M-2 and the cumulative payoff to that stored in the queue added to the cumulative payoff stored for the current state in M-2.