Appeared in *Proceedings of the Seventh Int. Conf. on Machine Learning*, pp. 216-224, Morgan Kaufmann, 1990.

# Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming

**Richard S. Sutton**
GTE Laboratories Incorporated
Waltham, MA 02254
sutton@gte.com

## Abstract

This paper extends previous work with Dyna, a class of architectures for intelligent systems based on approximating dynamic programming methods. Dyna architectures integrate trial-and-error (reinforcement) learning and execution-time planning into a single process operating alternately on the world and on a learned model of the world. In this paper, I present and show results for two Dyna architectures. The Dyna-PI architecture is based on dynamic programming's policy iteration method and can be related to existing AI ideas such as evaluation functions and universal plans (reactive systems). Using a navigation task, results are shown for a simple Dyna-PI system that simultaneously learns by trial and error, learns a world model, and plans optimal routes using the evolving world model. The Dyna-Q architecture is based on Watkins's Q-learning, a new kind of reinforcement learning. Dyna-Q uses a less familiar set of data structures than does Dyna-PI, but is arguably simpler to implement and use. We show that Dyna-Q architectures are easy to adapt for use in changing environments.

## 1 Introduction to Dyna

How should a robot decide what to do? The traditional answer in AI has been that it should deduce its best action in light of its current goals and world model, i.e., that it should *plan*. However, it is now widely recognized that planning's usefulness is limited by its computational complexity and by its dependence on an accurate world model. An alternative approach is to do the planning in advance and compile its result into a set of rapid *reactions*, or situation-action rules, which are then used for real-time decision making. Yet a third approach is to *learn* a good set of reactions by trial and error; this has the advantage of eliminating the dependence on a world model. In this paper I briefly introduce *Dyna*, a class of simple architectures integrating and permitting tradeoffs among these three approaches.

Dyna architectures use machine learning algorithms to approximate the conventional optimal control technique known as *dynamic programming (DP)* (Bellman, 1957; Ross, 1983). DP itself is not a learning method, but rather a computational method for determining optimal behavior given a complete model of the task to be solved. It is very similar to state-space search, but differs in that it is more incremental and never considers actual action *sequences* explicitly, only single actions at a time. This makes DP more amenable to incremental planning at execution time, and also makes it more suitable for stochastic or incompletely modeled environments, as it need not consider the extremely large number of sequences possible in an uncertain environment. Learned world models are likely to be stochastic and uncertain, making DP approaches particularly promising for learning systems. Dyna architectures are those that learn a world model online while using approximations to DP to learn and plan optimal behavior.

Intuitively, Dyna is based on the old idea that planning is like trial-and-error learning from hypothetical experience (Craik, 1943; Dennett, 1978). The theory of Dyna is based on the theory of DP (e.g., Ross, 1983) and on DP's relationship to reinforcement learning (Watkins, 1989; Barto, Sutton & Watkins, 1989, 1990), to temporal-difference learning (Sutton, 1988), and to AI methods for planning and search (Korf, 1990). Werbos (1987) has previously argued for the general idea of building AI systems that approximate dynamic programming, and Whitehead (1989) and others (Sutton & Barto, 1981; Sutton & Pinette, 1985; Rumelhart et al., 1986) have presented results for the specific idea of augmenting a reinforcement learning system with a world model used for planning.

## 2 Dyna-PI: Dyna by Approximating Policy Iteration

I call the first Dyna architecture *Dyna-PI* because it is based on approximating a DP method known as *policy iteration* (Howard, 1960). The Dyna-PI architecture consists of four components interacting as shown in Figure 1. The *policy* is simply the function formed by the current set of reactions; it receives as input a description of the current state of the world and produces as output an action to be sent to the world. The *world* represents the task to be solved; prototypically it is the robot's external environment. The world receives actions from the policy and produces a next state output and a reward output. The overall task is defined as maximizing the long-term average reward per time step (cf. Russell, 1989). The architecture also includes an explicit *world model*. The world model is intended to mimic the one-step input-output behavior of the real world. Finally, the Dyna-PI architecture includes an *evaluation function* that rapidly maps states to values, much as the policy rapidly maps states to actions. The evaluation function, the policy, and the world model are each updated by separate learning processes.

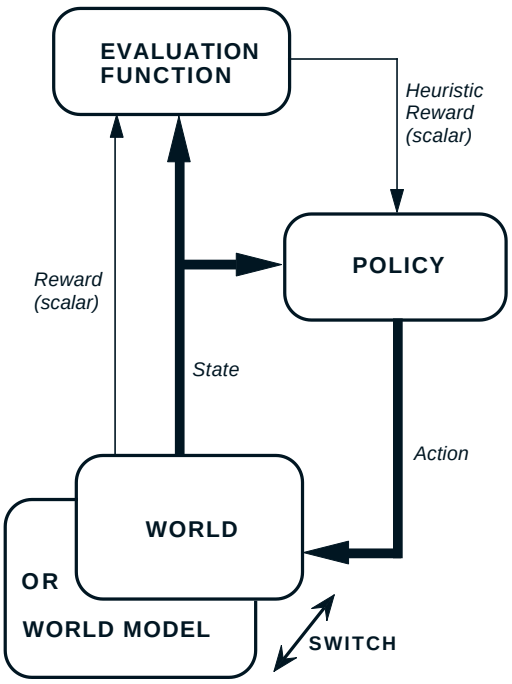


Figure 1: Overview of the Dyna Architecture. With the world in place as shown we have reinforcement learning; with the world model switched in place of the world we have planning.

For a fixed policy, Dyna-PI is simply a reactive system. However, the policy is continually adjusted by an integrated planning/learning process. The policy is, in a sense, a *plan*, but one that is completely conditioned by current input. The planning process is incremental and can be interrupted and resumed at any time. It consists of a series of shallow seaches, each typically of one step ply, and yet ultimately produces the same result as an arbitrarily deep conventional search. I call this *relaxation planning*. Dynamic programming is a special case of this.

Relaxation planning is based on continually adjusting the evaluation function in such a way that credit is propagated to the appropriate steps within action sequences. Generally speaking, the evaluation $e(x)$ of a state $x$ should be equal to the best of the states $y$ that can be reached from it in one action, taking into consideration the reward (or cost) $r$ for that one transition:

$$e(x) \quad \text{“}=\text{”} \quad \max_{a \in Actions} E\{r + e(y) \mid x, a\}, \qquad (1)$$

where $E\{\cdot \mid \cdot\}$ denotes a conditional expected value and the equal sign is quoted to indicate that this is a condition that we would like to hold, not one that necessarily does hold. If we have a complete model of the world, then the right-hand side can be computed by looking ahead one action. Thus we can generate any number of training examples for the process that learns the evaluation function: for any $x$, the right-hand side of (1) is the desired output. If the learning process converges such that (1) holds in all states, then the optimal policy is given by choosing the action in each state $x$ that achieves the maximum on the right-hand side. There is an extensive theoretical basis from dynamic programming for algorithms of this type for the special case in which the evaluation function is tabular, with enumerable states and actions. For example, this theory guarantees convergence to a unique evaluation function satisfying (1) and that the corresponding policy is optimal (Ross, 1983).

The evaluation function and policy need not be tables, but can be more compact function approximators such as decision trees, $k$-$d$ trees, connectionist networks, or symbolic rules. Although the existing theory does not apply to these machine learning algorithms directly, it does provide a theoretical foundation for exploring their use in this way. This kind of planning also extends conventional state-space planning in that it is applicable to stochastic and uncertain worlds and to non-boolean goals.

The above discussion gives the general idea of relaxation planning, but not the exact form used in

policy iteration and Dyna-PI, in which the policy is adapted simultaneously with the evaluation function. The evaluations in this case are not supposed to reflect the value of states given optimal behavior, but rather their value given current behavior (the current policy). As the current policy gradually approaches optimality, the evaluation function also approaches the optimal evaluation function. In addition, Dyna-PI is a *Monte Carlo* or *stochastic approximation* variant of policy iteration, in which the world model is only sampled, not examined directly. Since the real world can also be sampled, by actually taking actions and observing the result, the world can be used in place of the world model in these methods. In this case, the result is not relaxation planning, but a trial-and-error learning process much like reinforcement learning (see Barto, Sutton & Watkins, 1989, 1990). In Dyna-PI, both of these are done at once. The same algorithm is applied both to real experience (resulting in learning) and to hypothetical experience generated by the world model (resulting in relaxation planning). The results in both cases are accumulated in the policy and the evaluation function.

There is insufficient room here to fully justify the algorithm used in Dyna-PI, but it is quite simple and is given in outline form in Figure 2. The algorithm is based on a version of (1) modified to discount later as opposed to immediate reward:

$$e(x) \quad \text{“}=\text{”} \quad \max_{a \in Actions} E\{r + \gamma e(y) \mid x, a\}, \qquad (2)$$

where $\gamma$, $0 \leq \gamma < 1$, is the *discount rate*. Whereas (1) is limited to tasks that end with a clear termination event, such as the finding of a goal state or the end of a board game, (2) can be used for tasks that continue indefinitely, with rewards and/or penalties arriving on each step. Algorithms based on (2) are meant to estimate and maximize the expected value of a discounted sum of future reward:

$$E\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| x\right\},$$

where $r_1, r_2, r_3, \ldots$ is the sequence of future rewards. This is a standard optimization criterion in dynamic programming and Markov decision processes.

## 3 A Navigation Task

As an illustration of the Dyna-PI architecture, consider the task of navigating the maze shown in the upper right of Figure 3. The maze is a 6 by 9 grid of possible locations or states, one of which is marked as the starting state, “S”, and one of which is marked as the goal state, “G”. The shaded states act as barriers and cannot be entered. All the other states are distinct and completely distinguishable. From each there are four possible actions: UP, DOWN, RIGHT, and LEFT, which change the state accordingly, except where such a movement would take the take the system into a barrier or outside the maze, in which case the location is not changed. Reward is zero for all transitions except for those into the goal state, for which it is +1. Upon entering the goal state, the system is instantly transported back to the start state to begin the next trial.[1] None of this structure and dynamics is known to the Dyna-PI system a priori.

1. Decide if this will be a real experience or a hypothetical one.
2. Pick a state $x$. If this is a real experience, use the current state.
3. Choose an action: $a \leftarrow Policy(x)$
4. Do action $a$; obtain next state $y$ and reward $r$ from world or world model.
5. If this is a real experience, update world model from $x$, $a$, $y$ and $r$.
6. Update evaluation function so that $e(x)$ is more like $r + \gamma e(y)$; this is temporal-difference learning.
7. Update policy—strengthen or weaken the tendency to perform action $a$ in state $x$ according to the error in the evaluation function: $r + \gamma e(y) - e(x)$.
8. Go to Step 1.

Figure 2. Inner Loop of the Dyna-PI Algorithm. These steps are repeatedly continually, sometimes with real experiences, sometimes with hypothetical ones.

In this demonstration, the world was assumed to be deterministic, that is, to be a finite-state automaton, and the world model was implemented simply as next-state and reward tables that were filled in whenever a new state-action pair was experienced (Step 5 of Figure 2). The evaluation function was also implemented as a table and was updated (Step 6) according to the simplest temporal-difference learning method: $e(x) \leftarrow e(x) + \beta\big(r + \gamma e(y) - e(x)\big)$, where $\beta$ is a positive learning-rate parameter. The policy was implemented as a table with an entry $w_{xa}$ for every pair of state $x$ and action $a$. Actions were selected (Step 3) stochastically according to a Boltzmann distribution: $P(a|x) = e^{w_{xa}} / \sum_j e^{w_{xj}}$. The policy was updated (Step 8) according to: $w_{xa} \leftarrow w_{xa} + \alpha\big(r + \gamma e(y) - e(x)\big)$. For

[1] In fact, the goal state is never entered; the UP action from the state below produces a reward of +1 and sends the system directly to the start state.

hypothetical experiences, states were selected (Step 2) at random uniformly over all states previously encountered. The initial values of the evaluation function $e(x)$ and the policy table entries $w_{xa}$ were all zero; the initial policy was thus a random walk. The world model was initially empty; if a state and action were selected for a hypothetical experience that had never been experienced in reality, then the following steps (Steps 4–7) were simply omitted.

In this instance of the Dyna-PI architecture, real and hypothetical experiences were used alternately (Step 1). For each experience with the real world, $k$ hypothetical experiences were generated with the model. Figure 3 shows learning curves for $k = 0$, $k = 10$, and $k = 100$, each an average over 100 runs. The $k = 0$ case involves no planning; this is a pure trial-and-error learning system entirely analogous to those used in some reinforcement learning systems (Barto, Sutton & Anderson, 1983; Sutton, 1984; Anderson, 1987). Although the length of path taken from start to goal falls dramatically for this case, it falls much *more* rapidly for the cases including hypothetical experiences, showing the benefit of relaxation planning using the learned world model. For $k = 100$, the optimal path was generally found and followed by the fourth trip from start to goal; this is very rapid learning. The parameter values used were $\beta = 0.1$, $\gamma = 0.9$, and $\alpha = 1000$ ($k = 0$) or $\alpha = 10$ ($k = 10$ and $k = 100$). The $\alpha$ values were chosen roughly to give the best performance for each $k$ value.

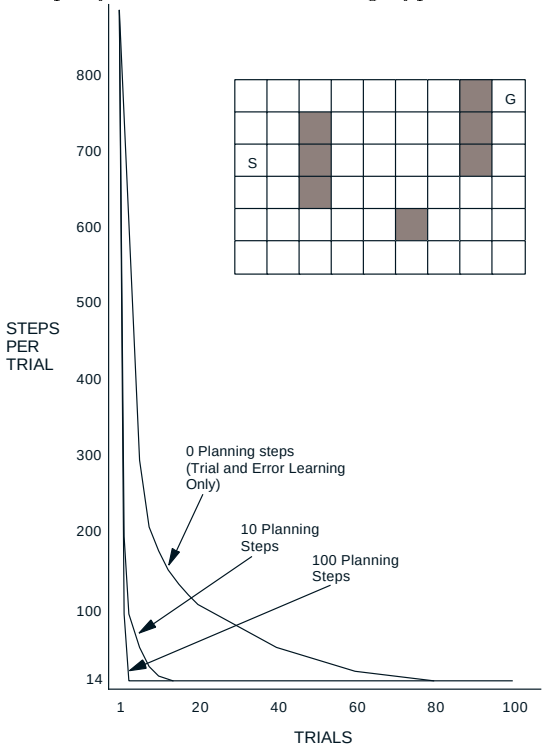


Figure 3. Learning Curves for Dyna-PI Systems on a Simple Navigation Task. A trial is one trip from the start state "S" to the goal state "G". The more hypothetical experiences ("planning steps") using the world model, the faster an optimal path was found.

Figure 4 shows why a Dyna-PI system that includes planning solves this problem so much faster than one that does not. Shown are the policies found by the $k = 0$ and $k = 100$ Dyna-PI systems half-way through the second trial. Without planning ($k = 0$), each trial adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, the first trial also learned only one step, but here during the second trial an extensive policy has been developed that by the trial's end will reach almost back to the start state. By the end of the third or fourth trial a complete optimal policy will have been found and perfect performance attained.

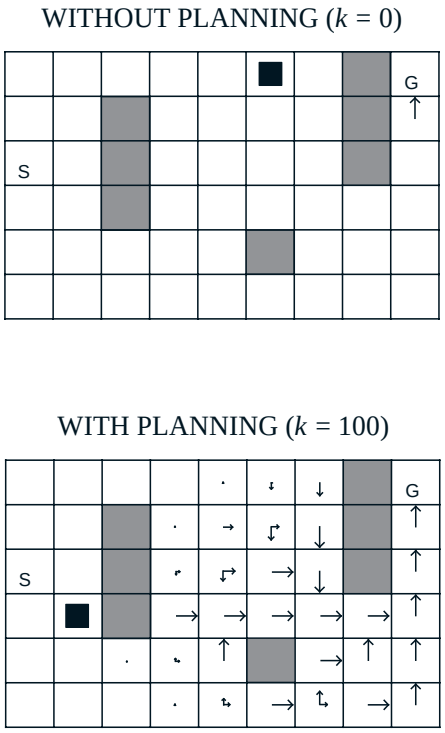


Figure 4. Policies Found by Planning and Non-Planning Dyna-PI Systems by the Middle of the Second Trial. The black square indicates the current location of the Dyna-PI system. The arrows indicate action probabilities (excess over the smallest) for each direction of movement.

## 4 Problems of Changing Worlds

Suppose that, after a Dyna-PI system has learned the optimal path from start to goal, a new barrier is added that blocks the optimal path. The Dyna-PI system described above will run into the block and then try the formerly effective action many hundreds of times. Eventually, the correct new path may be found, but the process is very slow. It seems inappropriately slow in that the system's world model is updated immediately. Even though the world model knows that the formerly good action is now poor, this is not reflected in the system's behavior for a long time. I call this the *blocking problem*.

Part of the problem is that the alternative actions are never tried, even hypothetically, because the policy assigns them a probability of zero. The model knows these actions are better, but this has no effect unless they are tried. One idea for solving this problem is to allow hypothetical actions to be selected according to a more liberal policy than that used to select real actions. The simplest case of this is that in which hypothetical actions are selected at random uniformly. If this is done, a small adjustment must be made to the evaluation update (Step 6). Recall that the evaluation function is supposed to represent the value of each state given the current policy. If hypothetical are selected uniformly, then the bias toward the current policy must be introduced explicitly. To do this, the evaluation update (Step 6), on hypothetical steps only, is altered to be weighted by the current action probability: $e(x) \leftarrow e(x) + \beta\big(r + \gamma e(y) - e(x)\big)P(a|x)$. In empirical studies we have indeed found this to be an improvement on the original algorithm, substantially improving the robustness of its convergence onto optimal behavior. However, this does not solve the blocking problem: the system still takes many hundreds of actions into an added barrier before finally finding a way around it.

Now consider a second sort of change in the environment. Suppose, after the optimal path has been learned, a barrier is removed that permits a shorter path from start to goal. The simple Dyna-PI system introduced above is unable to take advantage of such a shortcut; it never wavers from the formerly optimal path and thus never discovers that the former obstacle is gone. I call this the *shortcut problem*. In seeking to improve the Dyna-PI system to handle blocks, we might also seek to improve it to handle shortcuts. What is needed here is some way of continually testing the world model. In the next section we introduce a slightly different architecture that handles both kinds of changes with little increase in complexity.

## 5 Dyna-Q: Dyna by Q-learning

The Dyna-PI architecture is in essence the reinforcement learning architecture that my colleagues and I developed (Sutton, 1984; Barto, Sutton & Anderson, 1983) *plus* the idea of using a learned world model to generate hypothetical experience and to plan. Watkins (1989) subsequently developed the relationships between the reinforcement-learning architecture and dynamic programming (see also Barto, Sutton & Watkins, 1989, 1990) and, moreover, proposed a slightly different kind of reinforcement learning called *Q-learning*. The *Dyna-Q* architecture is the combination of this new kind of learning with the Dyna idea of using a learned world model to generate hypothetical experience and achieve planning.

Whereas the original reinforcement learning architecture maintains two fundamental memory structures, the evaluation function and the policy, Q-learning maintains only one. That one is a cross between an evaluation function and a policy. For each pair of state $x$ and action $a$, Q-learning maintains an estimate $Q_{xa}$ of the value of taking $a$ in $x$. The value of a *state* can then be defined as the value of the state's best state-action pair:

$$e(x) \stackrel{\text{def}}{=} \max_a Q_{xa}.$$

In general, the Q-value for a state $x$ and an action $a$ should equal the expected value of the immediate reward $r$ plus the discounted value of the next state $y$:

$$Q_{xa} \quad \text{“}=\text{”} \quad E\{r + \gamma e(y) \mid x, a\}. \tag{3}$$

To achieve this goal, the updating steps (Steps 6 and 7 of Figure 2) are implemented by

$$Q_{xa} \leftarrow Q_{xa} + \beta\big(r + \gamma e(y) - Q_{xa}\big). \tag{4}$$

This is the only update rule in Q-learning. We note that it is very similar though not identical to Holland's (1986) bucket brigade and to Sutton's (1988) temporal-difference learning.

So far, the Dyna-Q architecture is slightly simpler than the Dyna-PI architecture. Two data structures have been replaced with one (which is no larger than one of the original two), and one update rule and one parameter ($\alpha$) have been eliminated. However, Q-learning generally requires additional complexity in determining the policy from the Q-values, as we discuss below. One advantage of Q-learning is that it requires no special adjustments if the action selection during hypothetical experience is different from the current policy. Watkins (1989) has shown that the Q-values will converge properly whatever policy is used,

either hypothetically or in reality, as long as all state-action pairs are repetitively tried. In the following experiments, actions were selected at random uniformly (Step 3) on hypothetical experiences.

The simplest way of determining the policy on real experiences is to deterministically select the action that currently looks best—the action with the maximal Q-value. However, as we show below, this approach alone suffers from inadequate exploration and can not solve the shortcut problem. In his work with Q-learning, Watkins implemented the policy probabalistically using a Boltzmann distribution: $P(a|x) = e^{\alpha Q_{xa}} / \sum_j e^{\alpha Q_{xj}}$. An annealing process was added in which $\alpha$ tended to infinity so that even a small difference between Q-values would eventually lead to the best action being selected with probability one. That approach, however, recreates the problem of loss of variability in behavior such that shortcuts can not be found.

To deal directly with the shortcut problem, a new memory structure was added that keeps track of the degree of uncertainty about each component of the model. For each state $x$ and action $a$, a record is kept of the number of time steps $n_{xa}$ that have elapsed since $a$ was tried in $x$ in a real experience. The square root $\sqrt{n_{xa}}$ is used as a measure of the uncertainty about $Q_{xa}$.[2] To encourage exploration, each state-action pair is given an *exploration bonus* proportional to this uncertainty measure. For real experiences, the policy is to select the action $a$ that maximizes $Q_{xa} + \epsilon\sqrt{n_{xa}}$, where $\epsilon$ is a small positive parameter. This method of encouraging variety is very similar to that used in Kaelbling's (in preparation) interval-estimation algorithm.

However, this approach alone does not take advantage of the planning capability of Dyna architectures. Suppose there is a state-action pair that has not been tested in a long time, but which is far from the currently preferred path, and thus extremely unlikely to be tried even with the exploration bonus discussed above. In a Dyna system, why not expect the system to *plan* an action sequence to go out and test the uncertain state-action pair? If there is genuine uncertainty, then there is potential benefit in going out and trying the action, and thus forming such a plan is simply rational behavior and should be done. It turns out that there is a simple way to do this in Dyna-Q. The exploration bonus of $\epsilon\sqrt{n_{xa}}$ is used not in the policy, but in the update equation for the Q-values. That is, (4) is replaced by:[3]

$$Q_{xa} \leftarrow Q_{xa} + \beta\big(r + \epsilon\sqrt{n_{xa}} + \gamma e(y) - Q_{xa}\big). \qquad (5)$$

In addition, the system is permitted to hypothetically experience actions is has never before tried, so that the exploration bonus for trying them can be propagated back by relaxation planning. This can be done by starting the system with a non-empty initial model. In the experiments with Dyna-Q systems reported below, actions that had never been tried were assumed to produce zero reward and leave the state unchanged.

## 6 Changing-World Experiments

Experiments were performed to test the ability of Dyna systems to solve blocking and shortcut problems. Three Dyna systems were used: the Dyna-PI system presented earlier in the paper, a Dyna-Q system including the exploration bonus (5), called the *Dyna-Q+* system,[4] and a Dyna-Q system without the exploration bonus (4), called the *Dyna-Q−* system. All systems used $k = 10$. For the Dyna-PI system, the other parameters were set as in the navigation experiment. For the Dyna-Q systems, they were set at $\beta = 0.5$, $\gamma = 0.9$, and $\epsilon = 0.001$.

The blocking experiment used the two mazes shown in the upper portion of Figure 5. Initially a short path from start to goal was available (first maze). After 1000 time steps, by which time the short path was usually well learned, that path was blocked and a longer path was opened (second maze). Performance under the new condition was measured for 2000 time steps. Average results over 50 runs are shown in Figure 5 for the three Dyna systems. The graph shows a *cumulative* record of the number of rewards received by the system up to each moment in time. In the first 1000 trials, all three Dyna systems found a short route to the goal, though the Dyna-Q+ system did so significantly faster than the other two. After the short path was blocked at 1000 steps, the graph for the Dyna-PI system remains almost flat, indicating that it was unable to obtain further rewards. The Dyna-Q systems, on the other hand, clearly solved the blocking problem, reliably finding the alternate path after about 800 time steps.

[2] The use of the square root is heuristic but not arbitrary, as the standard deviation of all stationary, cumulative random processes increases with the square root of the number of cumulating steps.

[3] Note that this differs from (4) only on hypothetical experiences, as $n_{xa} = 0$ on real experiences.

[4] In these experiments, the Dyna-Q+ system selected the action $a$ in each state $x$ that maximized $Q_{xa} + \epsilon\sqrt{(n_{xa})}$, but we have since found that equally good performance can be obtained simply by picking the action with maximal $Q_{xa}$.

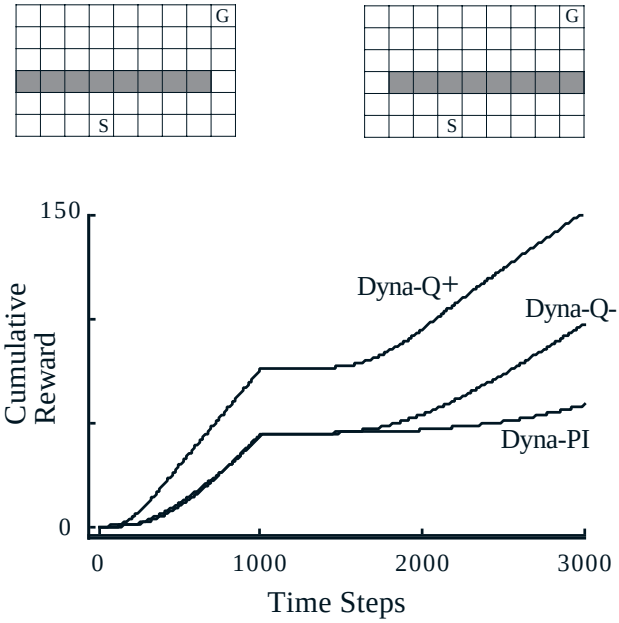


Figure 5. Average Performance of Dyna Systems on a Blocking Task. The left maze was used for the first 1000 time steps, the right maze for the last 2000. Shown is the cumulative reward received by a Dyna system at each time (e.g., a flat period is a period during which no reward was received).

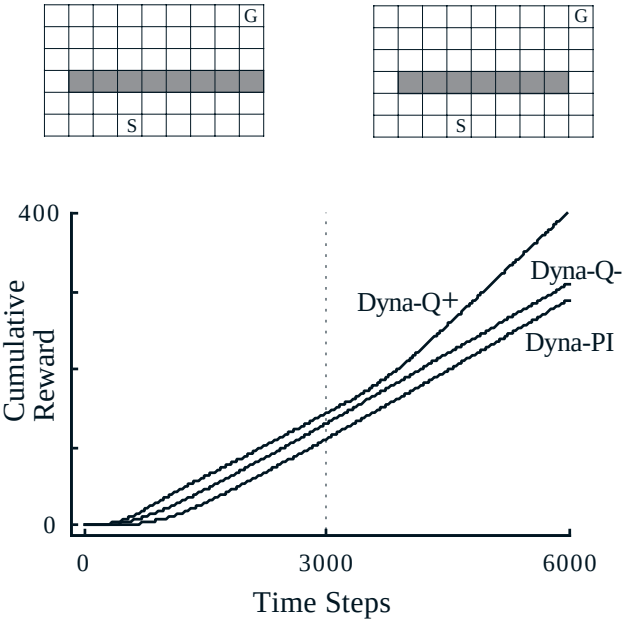


Figure 6. Average Performance of Dyna Systems on a Shortcut Task. The left maze was used for the first 3000 time steps, the right maze for the last 3000. Shown is the cumulative reward received by a Dyna system at each time (e.g., the slope corresponds to the *rate* at which reward was received).

The shortcut experiment began with only a long path available (first maze of Figure 6). After 3000 times steps all three Dyna systems had learned the long path, and then a shortcut was opened without interferring with the long path (second maze of Figure 6). The lower part of Figure 6 shows the results. The increase in the slope of the curve for the Dyna-Q+ system, while the others remain constant, indicates that it alone was able to find the shortcut. The Dyna-Q+ system also learned the original long route faster than the Dyna-Q− system, which in turn learned it faster than the Dyna-PI system. However, the ability of the Dyna-Q+ system to find shortcuts does not come totally for free. Continually re-exploring the world means occasionally making suboptimal actions. If one looks closely at Figure 6, one can see that the Dyna-Q+ system actually acheives a slightly lower *rate* of reinforcement during the first 3000 steps. In a static environment, Dyna-Q+ will eventually perform worse than Dyna-Q−, whereas, in a changing environment, it will be far superior, as here. One possibility is to use a meta-level learning process to adjust the exploration parameter $\epsilon$ to match the degree of variability of the environment.

One strength of the Dyna approach is that it applies to stochastic problems as well as deterministic ones. We have explored this direction in recent work, but are not yet ready to present systematic results. The basic idea is to learn a model which predicts not a determinstc next state and next reward, but rather a probability distribution over next states and next rewards. In the simple cases we have explored, this reduces to counting the number of times each possible outcome has occurred. In hypothetical experiences, the expected value on the right of (3) is then estimated using the sample statistics. A slightly different exploration bonus is also needed. Promising preliminary results have so far been obtained for simple problems involving random autonomous agents and stochastic state transitions (e.g., action UP takes the system to the state above 80% of the time, and to a random neighboring state 20% of the time).

Further results are needed for a thorough comparison of Dyna-PI and Dyna-Q architectures, but the results presented here suggest that it is easier to adapt Dyna-Q architectures to changing environments.

## 7 Limitations and Conclusions

The simple illustrations presented here are clearly limited in many ways. The state and action spaces are small and denumerable, permitting tables to be used for all learning processes and making it feasible for the entire state space to be explicitly explored. For large state spaces it is not practical to use tables or to visit all states; instead one must represent a limited amount of experience compactly and generalize from it. Both Dyna architectures are fully compatible with the use of a wide range of learning methods for doing this.

We have also assumed that the Dyna systems have explicit knowledge of the world's state. In general, states can not be known directly, but must be estimated from the pattern of past interaction with the world (Rivest & Schapire, 1987; Mozer and Bachrach, 1990). Dyna architectures can use state estimates constructed in any way, but will of course be limited by their quality and resolution. A promising area for future work is the combination of Dyna architectures with egocentric or "indexical-functional" state representations (Agre & Chapman, 1987; Whitehead, 1989).

Yet another limitation of the Dyna systems presented here is the trivial form of search control used. Search control in Dyna boils down to the decision of whether to consider hypothetical or real experiences, and of picking the order in which to consider hypothetical experiences. The tasks considered here are so small that search control is unimportant, and thus it was done trivially, but a wide variety of more sophisticated methods could be used. Particularly interesting is the possibility of using Dyna architectures at higher levels to make these decisions.

Finally, the examples presented here are limited in that reward is only non-zero upon termination of a path from start to goal. This makes the problem more like the kind of search problem typically studied in AI, but does not show the full generality of the framework, in which rewards may be received on any step and there need not even exist start or termination states.

Despite these limitations, the results presented here are significant. They show that the use of an internal model can dramatically speed trial-and-error learning processes even on simple problems. Moreover, they show how planning can be done with the incomplete, changing, and oftimes incorrect world models that are contructed through learning. Finally, they show how the functionality of planning can be obtained in a completely incremental manner, and how a planning process can be freely intermixed with reaction and learning processes. I conclude that it is not necessary to choose between planning systems, reactive systems and learning systems. These three can be integrated not only into one system, but into a single algorithm, where each appears as a different facet or different use of that algorithm.

**Acknowledgments**

The author gratefully acknowledges the extensive contributions to the ideas presented here by Andrew Barto, Chris Watkins and Steve Whitehead. I also wish to also thank the following people for ideas and discussions: Paul Werbos, Luis Almeida, Ron Williams, Glenn Iba, Leslie Kaelbling, John Vittal, Charles Anderson, Bernard Silver, Oliver Selfridge, Judy Franklin, Tom Dean and Chris Matheus.

**References**

Agre, P. E., & Chapman, D. (1987) Pengi: An implementation of a theory of activity. *Proceedings of AAAI-87*, 268–272.

Anderson, C. W. (1987) Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning*, 103–114. Morgan Kaufmann, Irvine, CA.

Barto, A. G., Sutton R. S., & Anderson, C. W. (1983) Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics 13*: 834–846.

Barto, A. G., Sutton, R. S., & Watkins, C. J. C. H. (1989) Learning and sequential decision making. COINS Technical Report 89-95, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. Also to appear in *Learning and Computational Neuroscience*, M. Gabriel and J.W. Moore (Eds.), MIT Press, 1990.

Barto, A. G., Sutton, R. S., & Watkins, C. J. C. H. (1990) Sequential decision problems and neural networks. In *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan Kaufmann, San Mateo, CA.

Bellman, R. E. (1957) *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Craik, K. J. W. (1943) *The Nature of Explanation*. Cambridge University Press, Cambridge, UK.

Dennett, D. C. (1978) Why the law of effect will not go away. In *Brainstorms*, by D. C. Dennett, 71–89, Bradford Books, Montgomery, Vermont.

Howard, R. A. (1960) *Dynamic Programming and Markov Processes*. Wiley, New York.

Kaelbling, L. (in preparation) Learning in Embedded Systems. Stanford Computer Science Ph.D. Dissertation.

Korf, R. E. (1990) Real-Time Heuristic Search. *Artificial Intelligence 42*: 189–211.

Mozer, M. C., & Bachrach, J. (1990) Discovering the structure of a reactive environment by exploration. In *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan Kaufmann, San Mateo, CA. See also Technical Report CU-CS-451-89, Dept. of Computer Science, University of Colorado at Boulder 80309.

Rivest, R. L., & Schapire, R. E. (1987) A new approach to unsupervised learning in deterministic environments. *Proceedings of the Fourth International Workshop on Machine Learning*, 364–375. Morgan Kaufmann, Irvine, CA.

Ross, S. (1983) *Introduction to Stochastic Dynamic Programming*. Academic Press, New York.

Rumelhart, D. E., Smolensky, P., McClelland, J. L., & Hinton, G. E. (1986) Schemata and sequential thought processes in PDP models. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume II*, by J. L. McClelland, D. E. Rumelhart, and the PDP research group, 7–57. MIT Press, Cambridge, MA.

Russell, S. J. (1989) Execution architectures and compilation. *Proceedings of IJCAI-89*, 15–20.

Sutton, R. S. (1984) Temporal credit assignment in reinforcement learning. Doctoral dissertation, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

Sutton, R.S. (1988) Learning to predict by the methods of temporal differences. *Machine Learning 3*: 9–44.

Sutton, R.S., Barto, A.G. (1981) An adaptive network that constructs and uses an internal model of its environment. *Cognition and Brain Theory Quarterly 4*: 217–246.

Sutton, R.S., Pinette, B. (1985) The learning of world models by connectionist networks. *Proceedings of the Seventh Annual Conf. of the Cognitive Science Society*, 54–64. Lawrence Erlbaum, Hillsdale, NJ.

Watkins, C. J. C. H. (1989) *Learning with Delayed Rewards*. PhD thesis, Cambridge University Psychology Department.

Werbos, P. J. (1987) Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, Jan-Feb.

Whitehead, S. D. (1989) Scaling reinforcement learning systems. Technical Report 304, Dept. of Computer Science, University of Rochester, Rochester, NY 14627.