Learning a Decision Rule for Monitoring Tasks with Deadlines*

Eric A. Hansen and Paul R. Cohen

Computer Science Technical Report 92-80

Experimental Knowledge Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

A real-time scheduler or planner is responsible for managing tasks with deadlines. When the time required to execute a task is uncertain, it may be useful to monitor the task to predict whether it will meet its deadline; this provides an opportunity to make adjustments or else to abandon a task that can't succeed. This paper treats monitoring a task with a deadline as a sequential decision problem. Given an explicit model of task execution time, execution cost, and payoff for meeting a deadline, an optimal decision rule for monitoring the task can be constructed using stochastic dynamic programming. If a model is not available, the same rule can be learned using temporal difference methods. These results are significant because of the importance of this decision rule in real-time computing.

---

* This research was supported by DARPA contract F49620-C-89-00113; by the Air Force Office of Scientific Research under the Intelligent Real-time Problem Solving Initiative, contract AFOSR-91-0067; and by Augmentation Award for Science and Engineering Research Training, supplement to DARPA contract F30602-91-C-0076.

1. Introduction

In hard real-time systems, tasks must meet deadlines and there is little or no value in completing a task after its deadline. When the time required to execute a task is uncertain, it may be useful to monitor the task as it executes so that failure to meet its deadline can be predicted as soon as possible. Anticipating failure to meet a deadline provides an opportunity to adjust; either to initiate a recovery action, or simply to abandon a failing task to conserve limited resources.

A system that monitors ongoing tasks needs a decision rule to predict whether a task will meet its deadline. The Phoenix planner uses a decision rule for this called an "envelope" (Hart, Anderson, & Cohen, 1990) which defines a range of expected performance over time. Envelopes are represented by a two-dimensional graph that plots progress toward completion of a task as a function of time; for example, in Figure 1 the shaded region represents an envelope. When the actual progress of a monitored task falls outside the envelope boundary, failure to meet the task's deadline is predicted and an appropriate action is taken.

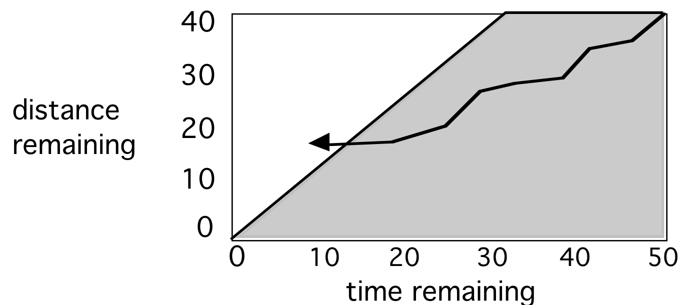

Figure 1. A Phoenix envelope. The arrow represents the trajectory of a task When it falls outside the shaded region, a recovery action is initiated.

Monitoring progress to anticipate failure to meet a deadline is useful in other contexts besides monitoring plans. For example, real-time problem-solvers monitor their progress to determine whether to adjust their strategies to make sure a solution is ready by a deadline (Durfee &Lesser, 1988; Lesser, Pavlin, & Durfee, 1988). Dynamic schedulers for real-time operating systems monitor task execution so they can anticipate failure to meet a deadline in time to abort a failing task and conserve resources (Haben & Shin, 1990).

Despite the many contexts in which it is useful to monitor tasks with deadlines, no principled method has yet been developed for constructing optimal decision rules, that is, rules to predict whether a task will meet a deadline. The envelopes used by the Phoenix planner were handcrafted and

adjusted by trial and error to get them to work well. Similar decision rules used in other systems have also been heuristic or constructed in an ad hoc way.

This paper describes how an optimal decision rule can be constructed. As an example, it works through the construction of a rule for the simple case in which the only option for a failing task is to abandon it. When recovery options are available a more complex decision rule is called for, but constructing it is a straightforward extension of the methods described here for constructing the rule in its simplest form.

Monitoring a task over the course of its execution requires a sequence of related decisions; each of these decisions is not whether to continue the task to the end, but whether to continue it a little longer before monitoring and reconsidering whether to continue it further. A task should be continued as long as the expected value of continuing it until the next monitoring action is greater than the expected value of abandoning it. Knowing the expected value of continuing a task until it is monitored again requires knowing the expected value of continuing it after that, and so on recursively until the deadline. In this sense, constructing an optimal decision rule is a sequential decision problem.

Section 2 describes how to use dynamic programming to unravel this recursive relationship backwards from the deadline. Using dynamic programming to construct the decision rule requires an explicit model of probable task execution time, execution cost, and the payoff for meeting the deadline. Section 3 describes machine learning techniques that can learn the decision rule if a model is not available. Section 4 discusses the generality of these results and possible extensions.

2. Constructing the Decision Rule Using Dynamic Programming

Dynamic programming is an optimization technique for solving problems that require making a sequence of decisions. A decision rule for monitoring task execution can be constructed using dynamic programming by assuming a task is monitored at discrete time steps; any interval of time can correspond to a time step.

The decision problem is formalized as follows. A state is represented by two variables, (t,d), where t is a non-negative integer that represents the number of time steps remaining before the deadline and d is a non-negative integer that represents the part of the task (the "distance") that remains to be completed. (This presupposes some unit of progress in terms of which completion of a task can be measured.) The decision to continue a task or abandon it is represented by a binary decision variable, $a \in \{continue, stop\}$.

The amount of a task likely to be executed in one time step is represented by a set of state transition probabilities, $P_{(t,d),(t-1,d-k)}(continue)$. In this notation, the first subscript, (t,d), is the state at the beginning of the time step and the second subscript, $(t-1,d-k)$, is the state at the end of the time step, where k is the number of units of the task completed during the time step. The argument is the action taken at the beginning of the time step.

In addition to this stochastic state-transition model, a function, $R(t,d,a)$, specifies the single-step payoff for action, a, taken in state, (t,d). The execution cost per time step is $R(t,d,continue)$; while $R(t,d,stop)$ is the value for finishing a task by its deadline (when $t \geq 0, d = 0$) or zero if the deadline is not met ($t = 0, d \geq 1$).

The objective is to maximize the expected value of the sum of the single-step payoffs for the course of a task. The difficulty is that the payoff for finishing a task by its deadline is not received until the end of the task, although it must be considered in making earlier decisions about whether to continue the task. Dynamic programming solves sequential decision problems with "delayed payoff" by constructing an evaluation function that provides *secondary reinforcement*; the key idea is that expected cumulative value over the long term is maximized by choosing, at each step, the action that maximizes the evaluation function in the short term.

The evaluation function for this decision problem is expressed by the recurrence relation:

$$V(t,d) = \max\left\{R(t,d,stop), E\left[V(t-1,d-k)\right] + R(t,d,continue)\right\}$$

Expanding the term for the expected value of the next state, this becomes:

$$V(t,d) = \max\left\{R(t,d,stop), \sum_k \left[P_{(t,d),(t-1,d-k)}(continue) \cdot V(t-1,d-k)\right] + R(t,d,continue)\right\}$$

Dynamic programming systematically evaluates this recurrence relation to fill in a table of values, $V(t,d)$, that represents the evaluation function.

In the course of evaluating this recurrence relation, the optimal action in each state (whether to stop or continue) is determined. The decision rule is defined implicitly by the evaluation function, as follows.

$$\pi(t,d) = \begin{cases} continue \;\; if \;\; V(t,d) > 0 \\ stop \;\; otherwise \end{cases}$$

That is, continue if the expected value of continuing exceeds zero, the value for stopping. (In the terminology of dynamic programming, a decision rule is also referred to as a policy function.)

This formal notation describes the decision rule in its most general form. However it leaves the state transition probabilities and costs used to compute the rule unspecified; they are fit to the decision problem being modeled. There are two ways to obtain these probabilities and costs. One is to specify them given some exogenous knowledge about the task; the other is to learn them automatically. We will consider these in turn.

For the sake of having an example to work through in this paper, we make the following arbitrary assumptions. Each time step is regarded as a single Bernoulli trial. This makes the state transition probabilities:

$$P_{(t,d),(t-1,d-1)}(continue) = p$$
$$P_{(t,d),(t-1,d)}(continue) = 1 - p$$

where p is the probability of "success" in a Bernoulli trial. Completing a task of size d is equivalent to succeeding in d Bernoulli trials, and completing the task of size d in time t is equivalent to succeeding in d out of t trials. So task execution time is binomially distributed with a mean and variance proportional to time.

Because a binomial distribution is approximately normal when $tp(1-p) \geq 10$, and because it seems likely that in many cases task execution time is approximately normally distributed, this is a plausible model. The mean and variance of the binomial probability function can be fit to the actual mean and variance of task execution time by choosing the value of p so that the equation,

$$p = 1 - Variance(execution\ time) / Mean(execution\ time)$$

is satisfied, and by choosing the scale of the distance step so that the mean proportion of a task executed in one time step is $p \cdot (\text{distance step})$.

As a payoff function, we assume:

$$\begin{aligned} R(0,d,stop) &= 0 \quad for\ d \geq 1 \\ R(t,0,stop) &= R \quad for\ t \geq 0 \\ R(t,d,continue) &= E \quad for\ t,d \geq 1 \end{aligned}$$

where R is the reward for finishing a task by its deadline and E is the execution cost per time step. Like the binomial state-transition model, this parameterized payoff function is very general and likely to fit many

situations. However it too is only an example; any payoff function could be used.

Given these state-transition probabilities and payoffs, the evaluation function for this decision problem has the following simple recursive definition:

$$V(0,d) = 0 \quad \textit{for} \ d \geq 1$$
$$V(t,0) = R \quad \textit{for} \ t \geq 0$$
$$V(t,d) = \max\{0, p \cdot V(t-1,d-1) + (1-p) \cdot V(t-1,d) + E\} \quad \textit{for} \ t,d \geq 1$$

The evaluation function for parameter values, $p = 0.5$, $F = 100$ and $E = -1$, is shown in Figure 2.

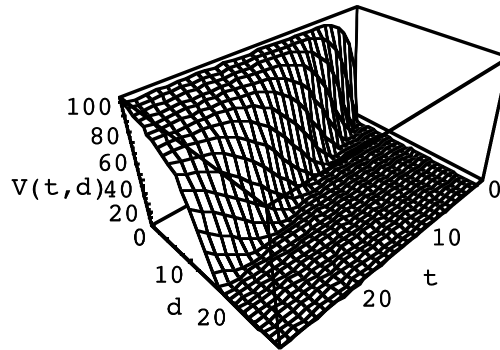

Figure 2. Evaluation function computed by dynamic programming.

The decision rule is represented implicitly by the evaluation function, since a task is continued as long as the expected value of the current state is greater than zero.

In the following graph the decision rule is projected onto two dimensions and extended out to a starting time of 200 before the deadline; this shows that for $d \geq 50$ it is not worth even beginning the task because the expected cost of completing the task is greater than the potential reward for finishing it.

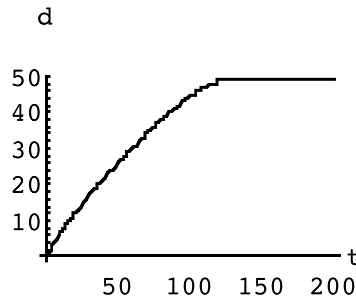

Figure 3. Representation of decision rule computed by dynamic programming. If the trajectory of a task goes above the line, the task is abandoned.

The shape of this decision rule is strikingly similar to the shape of the envelopes constructed by trial and error and used by the Phoenix planner.

3. Learning the Decision Rule Using Temporal Difference Methods

When an accurate model of the state transition probabilities and costs is available, an optimal decision rule can be computed off-line using dynamic programming. When an accurate model is not available, however, there are still machine learning methods that can gradually adapt a decision rule until it converges to one that is optimal. Called temporal difference (TD) methods (Barto, Sutton, & Watkins, 1990), they solve the so-called "temporal credit-assignment problem" inherent in sequential decision problems with delayed payoff by, in effect, approximating dynamic programming.

It can be shown that the recurrence relation for an evaluation function constructed by dynamic programming only holds true for a decision rule (i.e., policy) that is optimal. This provides the basis for TD methods. TD methods learn an evaluation function in addition to a decision rule. In our monitoring example, the recurrence relation that defines the evaluation function is

$$V(t,d) = E\left[V(t-1,d-k)\right] + R(t,d,continue)$$

Any measured difference during training between the values of the two sides of the equation is treated as an "error" that is used to adjust the decision rule. The TD error measured after each time step is:

$$V(t,d) - V(t-1,d-k) - R(t,d,continue)$$

The difference in this definition of TD error and the recurrence relation for the evaluation function is that the expected value, $E\left[V(t-1,d-k)\right]$, is replaced by $V(t-1,d-k)$. This is necessary because the expected value cannot be computed without a model of the state-transition probabilities. However TD training works because the learned value of each state regresses towards the weighted average of the values of its successor states, where the weightings reflect the conditional probabilities of the successor states. So in the limit, the value of each state converges to the expected value of its sucessor states plus the single-step payoff, and hence converges to the expected cumulative value.

By adapting the decision rule to minimize the TD error, an optimal decision rule is gradually learned along with an evaluation function. Adjusting the decision rule changes the evaluation function, which in turn serves as feedback to the learning algorithm for continued adjustment of the decision rule; the two are adapted simultaneously. In most cases of TD

learning, the decision rule and evaluation function must be represented separately. However this example is particularly straightforward because the simple relationship between the two ---the decision rule is defined by a simple threshold on the evaluation function--- makes it possible to represent both by the same function.

A complication is that learning takes place only as long as a task is continued, and so only inside the "boundary" of the decision rule. If this boundary is inadvertently set too conservatively, it cannot be unlearned unless a task is occasionally continued from a state outside the boundary to see what happens. This is characteristic of trial-and-error learning; occasionally actions that appear suboptimal must be taken so that the relative merits of actions can be assessed. This is managed by including a stochastic element in the decision rule, for example:

$$\pi(t,d) = \begin{cases} \textit{continue if } \left(V(t,d) + \left(random(2.0) - 1\right)\right) > 0 \\ \textit{stop otherwise} \end{cases}$$

The decision to use TD methods for training is independent of the decision about what function representation and learning algorithm to use. We show this by describing two different representations for the evaluation function and two different learning algorithms.

3.1 Table Representation and Linear Update Rule

If we represent the evaluation function by a two-dimensional table, as we have for dynamic programming, then the values in the table can be adjusted by the following learning rule:

$$V(t,d) \ := \ V(t,d) + \alpha \cdot error \cdot V(t,d)$$

This learning rule increments or decrements the value of the current state by an amount proportional to the TD error, defined as:

$$error = R(t,d,continue) + V(t-1,d-k) - V(t,d)$$

as well as proportional to a learning rate, α (in this example, set to 0.1). This linear learning rule minimizes the TD error by gradient descent.

A training regimen consists of repeatedly starting a task from a random state, (t,d), and continuing it until it finishes or is abandoned; each task counts as a learning trial. For the purpose of generating a learning curve, performance is measured by comparing the evaluation function computed by stochastic dynamic programming to the table of values learned by TD

training and measuring the mean square error. This comparison gives rise to the learning curve shown in Figure 4.

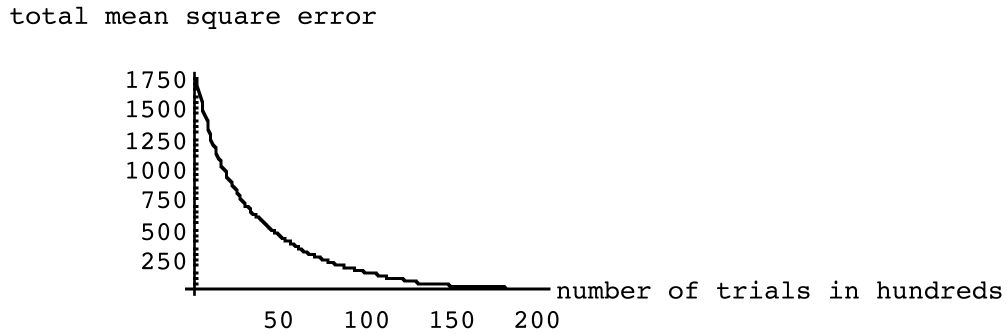

Figure 4. Learning curve for TD training, using a table representation and linear update rule.

The learned evaluation function, shown in Figure 5, is nearly identical to the optimal evaluation function computed by dynamic programming.

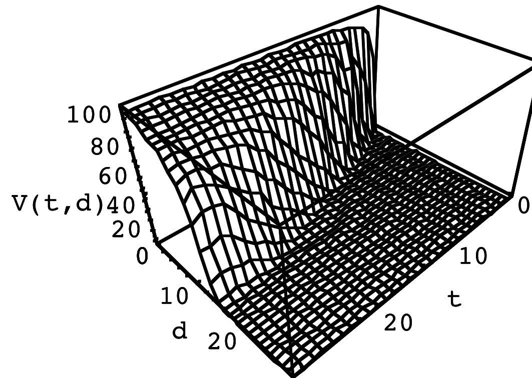

Figure 5. Evaluation function learned by TD training, using a table representation and linear update rule.

3.2 Connectionist Representation and Error Backpropagation Rule

The problem with representing the evaluation function by a table is that it requires $O(n^2)$ storage, where n is the number of time steps from the start of the task to its deadline. A more compact function representation would be better, although it still must be capable of representing a nonlinear function. One possibility is a feedforward connectionist network trained by the error

backpropagation rule. The simplest network possible for this problem is a single neuron with two inputs and a sigmoid activation function. It corresponds to the formula

$$V(t,d) = \frac{1}{1 + e^{-(w_1 t + w_2 d + w_3)}}$$

where w_1, w_2, and w_3 are the learned weights. This simple representation turns out to work surprisingly well. Trained by temporal differences using backpropagation, the learning curve in Figure 6 illustrates that it converges many times faster than when a table represenation and linear update rule are used.

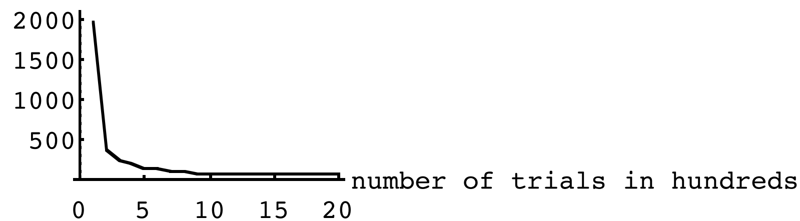

Figure 6. Learning curve for TD training of neuron by error backpropagation.

The reason convergence is faster is that the parameterized function represented by the neuron makes generalization possible; in addition, it is able to closely approximate the optimal evaluation function. The learned evaluation function is shown in Figure 7.

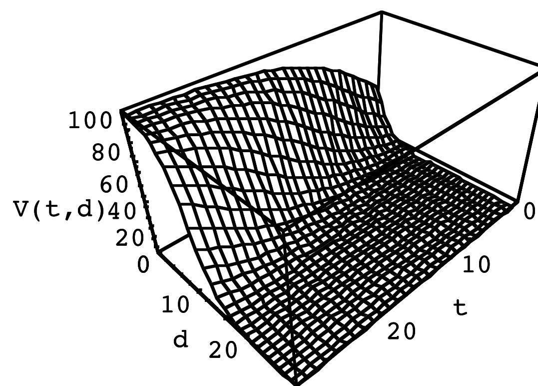

Figure 7. Evaluation function learned by TD training of a single neuron by error backpropagation.

Besides being more space-efficient, a connectionist network also has the advantage of being able to represent a continuous evaluation function, instead of the discrete function presupposed by a table representation. It allows for generalization as well because the possible input values are not limited by the size of the table.

4. Discussion

This paper treats monitoring tasks with deadlines as a sequential decision problem, which makes available a class of methods based on dynamic programming for constructing a decision rule for monitoring. When an explicit model of the state-transition probabilities and costs is available, the rule can be constructed off-line using stochastic dynamic programming. Otherwise it can be learned on-line using TD methods that approximate dynamic programming.

It makes sense to construct a decision rule such as the one described in this paper for tasks that are repeated many times or for a class of tasks with the same behavior. This allows the rule to be learned, if TD methods are relied on; or for statistics to be gathered to characterize a probability and cost model, if dynamic programming is relied on. However if a model is known beforehand, or can be estimated, a decision rule can also be constructed for a task that executes only once.

The time complexity of the dynamic programming algorithm is $O(n^2)$, where n is the number of time steps from the start of the task to its deadline; however the decision rule may be compiled once and reused for subsequent tasks. The time complexity of TD learning, $O(n)$, is mitigated by the possibility of turning learning off and on. The space overhead of representing an evaluation function by a table is avoidable by using a more compact function representation, such as a connectionist network.

Besides the fact that the approach described in this paper is not computationally intensive, it has other advantages. It is conceptually simple. The decision rule it constructs is optimal, or converges to the optimal in the case of TD learning. It works no matter what probability model characterizes the execution time of a task and no matter what cost model applies, and so is extremely general. Finally, it works even when no model of the state transition probabilites and costs is available, although a model can be taken advantage of.

These results can be extended in a couple obvious ways. The first is to factor in a cost for monitoring. In this paper we assume monitoring has no cost, or its cost is negligible. This allows monitoring to be nearly continuous, in effect, for a task to be monitored each time step. Others who have

developed similar decision rules have also assumed the cost of monitoring is negligible. However in some cases the cost of monitoring may be significant, so in another paper we show how this cost can be factored in (forthcoming). Once again we use dynamic programming and TD methods to develop optimal monitoring strategies.

The second way in which this work can be extended is to make the decision rule more complicated. In this paper we analyzed a simple example in which the only alternative to continuing a task is to abandon it. But recovery options may be available as well. A dynamic scheduler for a real-time operating system is unlikely to have recovery options available, but an AI planner or problem-solver is almost certain to have them (Lesser, Pavlin & Durfee, 1988; Howe, 1992). The way to handle the more complicated decision problem this poses is to regard each recovery option as a separate task characterized by its own probability model and cost model; so at any point the expected value of the option can be computed. Then instead of choosing between two options, either continuing a task or abandoning it, the choice includes the recovery options as well. The rule is simply to choose the option with the highest expected value.

Acknowledgments

This research is supported by the Defense Advanced Research Projects Agency under contract #F49620-89-C-00113; by the Air Force Office of Scientific Research under the Intelligent Real-time Problem Solving Initiative, contract #AFOSR-91-0067; and by Augmentation Award for Science and Engineering Research Training, PR No. C-2-2675. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

References

Barto, A.G.; Sutton, RS.; and Watkins, C.J.C.H., 1990. Learning and sequential decision making. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*.. M. Gabriel and J. W. Moore (Eds.), MIT Press, Cambridge, MA. Pp. 539-602.

Durfee, E. and Lesser, V., 1988. Planning to meet deadlines in a blackboard-based problem solver. In *Hard Real-Time Systems*. J. Stankovic and K. Ramamrithan (Eds.), IEEE Computer Society Press, Los Alamitos, CA. Pp. 595-608.

Haben, D. and Shin, K., 1990. Application of real-time monitoring to scheduling tasks with random execution times. In *IEEE Transactions on Software Engineering* 16(2): 1374-1389.

Hart, D.M.; Anderson, S.D.; and Cohen, P.R., 1990. Envelopes as a vehicle for improving the efficiency of plan execution. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*. K. Sycara (Ed.), San Mateo, CA: Morgan-Kaufman, Inc. Pp. 71-76.

Howe, A., 1992. Analyzing failure recovery to improve planner design. In *Proceedings AAAI-92*. Pp. 387-392.

Lesser, V.; Pavlin, J.; and Durfee, E., 1988. Approximate processing in real-time problem-solving. In *AI Magazine* 9(1): 49-61.