In Meyer, J.-A., Roitblat, H., and Wilson, S., editors, *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 148–155. MIT Press.

# Two Methods for Hierarchy Learning in Reinforcement Environments

**Mark Ring**
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
(ring@cs.utexas.edu)

## Abstract

This paper describes two methods for hierarchically organizing temporal behaviors. The first is more intuitive: grouping together common sequences of events into single units so that they may be treated as individual behaviors. This system immediately encounters problems, however, because the units are binary, meaning the behaviors must execute completely or not at all, and this hinders the construction of good training algorithms. The system also runs into difficulty when more than one unit is (or should be) active at the same time. The second system is a hierarchy of *transition* values. This hierarchy dynamically modifies the values that specify the degree to which one unit should follow another. These values are continuous, allowing the use of gradient descent during learning. Furthermore, many units are active at the same time as part of the system's normal functionings.

## 1 Introduction

The importance of hierarchy in adaptive systems that perform temporal tasks has been noted often (Albus, 1979; Wilson, 1989; Roitblat, 1991; Wixson, 1991; Drescher, 1991; Schmidhuber, 1992; Singh, 1992). Dawkins (1976) described the intrinsic importance and variety of hierarchy in animal behavior, and he noted several arguments that demonstrate the merit of hierarchy as a general design principle.

In some systems (Albus, 1979; Roitblat, 1991), hierarchy is developed as an efficient method for modularizing temporal tasks such that they can be accomplished in a general way: in order to perform each task, a sequence of sub-tasks must somehow be performed. In other systems (Wilson, 1989; Drescher, 1991; Schmidhuber, 1992; Singh, 1992), hierarchy is used to organize concept learning. In these cases, sequences or sequential tasks are learned by combining already known sequences or tasks into new ones. In this paper, I will focus on the latter class. Specifically, I will explore two methods for learning *behavior* hierarchies, where a behavior is a sequence of perceptions and/or actions.

The motivation for hierarchy learning is one of efficiency and increased learning potential. As Dawkins pointed out (Dawkins, 1976, p. 16), complex organisms probably evolved from simpler organisms that embodied many of the sub-assemblies required to build the more complex organisms. One might also speculate that organisms capable of performing certain elaborate behaviors may have evolved from simpler organisms that performed less elaborate components of these more complex behaviors. One often finds in the neural network literature the tendency to preprogram into one's learning systems everything that one *can* preprogram easily, because the system might find exactly these aspects of the task very difficult, and it might then learn the remainder of the task—that which is difficult to preprogram—fairly easily (see for example Bachrach, 1983, ch. 5). But even the simplest organisms are often so difficult for artificial systems to simulate, that there is a vast expanse of behavior both too difficult to hard-wire, and too difficult to learn. In these cases, what is needed is a system specifically designed to learn layer after layer of behaviors on top of those that it has already learned, each possibly more complex than the preceding ones (see also Drescher (1991) for a related discussion). With a hierarchical approach such as this in mind, it is conceivable to imagine simulated agents where the foundations necessary for learning a difficult task are not hard-wired into the circuitry of the system, but taught to the continuously evolving adaptive system.

In this paper, I investigate two methods that attempt to capture some aspects of hierarchical learning. In section 2, I will describe a unit devised for learning sequential behavior hierarchies. In section 3, I will describe a continuous version of this unit that overcomes problems encountered with the first method while being simpler to implement.

## 2 Method I: Behavior Hierarchies

My first attempt at behavior hierarchies was an animat controlled by a system of units.[1] In this system, each unit represents a specific behavior sequence. Some units represent primitive behaviors: a single sensation, or a single action. Other, higher-level units represent a sequence of two primitive units, and still higher-level units represent sequences of any two lower units. The system executes a behavior by choosing the unit that stands for that behavior. If a primitive action is chosen, that action is attempted in an external environment. If a primitive sensation is chosen, the system attempts to perceive that sensation. If a higher-level unit is chosen, it is decomposed into the two units that it represents; then the first unit's behavior is executed, followed by the second unit's. At any time, a new higher-level behavior might be added to the system's abilities by creating a new unit that represents a sequence of two units already in the system.

An example should clarify all this. Suppose the system could sense heat and coldness, light and darkness, and that it could move one step to the north, south, east, or west. Its primitive sensation units would be: SH (sense heat), SC (sense cold), SL (sense light), and SD (sense darkness). Its primitive actions would be: MN (move north), MS (move south), MW (move west), and ME (move East). A new behavior could be created that combined, for example, ME and SC. The new unit would be called: <ME, SC>, and it would represent the behavior, "Move East and see if it's cold". After this unit is created, another new unit might also be formed: <<ME, SC>, MS> (move east and see if it's cold; if it is, move south). As can be seen in the last example, the rest of a sequence is executed only if the part executed so far has been successful. This allows testing the environment and acting on the results: <SD, MW> (see if it's dark, and if it is, move west).

Behaviors are chosen randomly at first in an effort to achieve a reward. When a reward is received, the system learns that the most recently chosen behaviors may be worth repeating. The system must therefore keep track of the choices it has made and the level of reinforcement it has received for these choice-sequences. To do this, the entire system is embedded in a neural network, where the connections between units record this information. The stronger the connection from one unit to another, the more likely execution of the first followed by the second will result in reward. Therefore, in order to determine which behavior the system should execute next, every unit representing a behavior that has just completed votes for a successor with its weights: [2]

$$a_i(t+1) = \sum_j w_{ij} o_j(t), \tag{1}$$

where $o_j(t)$, the output of unit $j$ at time $t$, is 1.0 if the behavior represented by unit $j$ completed at time $t$, and is 0.0 otherwise; $w_{ij}$ is the weight of the connection from $i$ to $j$; and $a_j(t+1)$ is the resulting degree to which the system believes unit $j$'s behavior should occur at the next time-step. When a unit is chosen at the next time-step, $t+1$, the choice is biased by these values: though stochastic, the unit chosen is *probably* the one with the highest activation.

The connection weights are set by use of the delta rule (Widrow and Hoff, 1960), amplified by the reinforcement signal:

$$\Delta w_{ij}(t) = \eta R(t) o_j(t)(T_i(t) - a_i(t)) \tag{2}$$

That is, the weight change at time $t$ of the connection from unit $j$ to unit $i$ is equal to the product of the learning rate, $\eta$, the current reward-level $R(t)$, the current output of unit $j$, and the difference between the activation of unit $i$ and its target, $T_i(t)$. The target of a unit is simply its output at the next time-step, i.e. $T_i(t) = o_i(t+1)$. (They are given different names in equation 1 for the benefit of those already familiar with the delta rule).

This rule states that if some unit B is chosen after another unit A's behavior completes, then wait to see if B's behavior completes. If it does not complete (or if unit B had not been chosen), then the weight from A to B is decreased by an amount proportional to the current reward and unit B's expectation. (So if unit B was highly expected, but it did not complete, the change is large, causing the system to learn not to expect B as much following A.) But if unit B's behavior does complete, then the weight is increased by an amount proportional to the reward received and the amount by which B fell short of its target, 1.0. (The smaller the expectation, the more the weight will be changed, causing the system to increase its expectation of B following A.)

Figure 1 presents a very simple example of how the system could be used. The animat begins at position 1 in the maze. It will receive a reward if it makes it to the asterisk in position 6. From position 5, ME *should* become highly activated because the animat will receive a reward if it moves east. But how does it know it's in position 5? It knows it's in position 5 if it senses a light. Therefore, the sequence SL→ME may frequently be followed by reward, so the connection from SL to ME will become strong. After a while,

[1] I do not have the space here to give all the fine-grained details of this first system. I can however give a general overview and describe the intended behavior of the system from a high-level perspective, depicting the units as something akin to macro-operators. The details of the implementation are somewhat cumbersome, but are given in *slightly* more detail in a previous paper (Ring, 1991).

[2] It is possible for more than one behavior to complete at the same time. For example: SC, <ME,SC>, and <MN,<ME,SC>> would all finish whenever <MN,<ME,SC>> finishes.

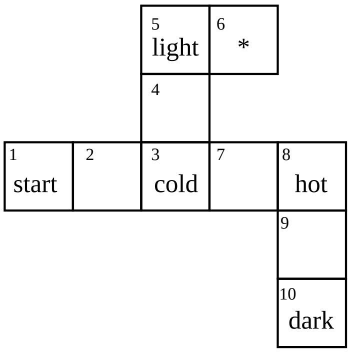


Figure 1: A maze for an animat. The animat would start in position 1 and would receive a reward in position 6. Other labels show what sensations the animat could perceive in different parts of the maze.

a new unit might be formed, <SL, ME>, to encapsulate this behavior. As units are used more and more in the same sequence again and again, the connections between them get stronger, and other new units might be created, such as <SC, <MN, MN>> (useful in position 3), <SH, <MW, MW>> (useful in position 8), and <<<ME, ME>,<SC, <MN, MN>>>, <SL, MR>> (useful in position 1), for example. The units clearly resemble macro-operators, though they are to be used in reinforcement environments with no explicit goals.

Reinforcement, in fact can be done easily: simply reward the last *n choices* made, and even though the behaviors represented by these choices may span a large period of time, reinforcement is spread smoothly across that time span. This is accomplished by slightly modifying equation 2 to:

$$\begin{aligned} \Delta w_{ij}(t) &= o_j(t)(T_i(t) - a_i(t)) + \sigma \Delta w_{ij}(t-1) \qquad (3) \\ w_{ij}(t) &= \eta R(t) \Delta w_{ij}(t) + w_{ij}(t-1) \end{aligned}$$

where $0 \leq \sigma \leq 1$ is a decay parameter that discounts previous weight changes in favor of more recent ones. The $\Delta w$'s are therefore a *trace* of weight changes that decay exponentially—like the eligibility trace presented in (Barto et al., 1983). The trace constantly accrues weight changes over time, biased toward the most recent ones, but the changes are only applied to the weights when a reward is received.

## 2.1 A Different Approach is Needed

There are problems with this approach. First and most importantly, these units are binary: the behaviors either execute, or they do not. This is a discontinuity that keeps the gradient descent performed by the delta rule from working effectively. Learning tends to be chaotic. Second, it's possible for multiple behaviors to complete simultaneously, as mentioned above, or to *begin* simultaneously. For example, if many sensations are impinging on the system at the same time, the sequence in which they are sensed is irrelevant. In these cases many units could be formed that are functionally identical (grouping together the same sensations), but structurally different (grouping them together in different orders). And third, there may be many ways of achieving the same end. A behavior that takes the animat from home to work, for example, may meet contingencies on the way. A traffic light may be red when a green light was expected. Yet there is no way to encode this contingency within a single unit such that one action is taken when the light is green, and another is taken when the light is red, both ending with the animat arriving at work. If two behaviors could be chosen at the outset, in one of which a green light is expected, and in the other a red light is expected, then the system would merely be inefficient, requiring enormous numbers of units to encode every possible combination of contingencies. But when only one sequence of behaviors can be chosen at a time, encoding contingencies is not just inefficient; it's impossible. A solution to some of these problems is described in the next section.

# 3 Method II: Transition Hierarchies

The second method of organizing behaviors hierarchically also uses a pool of units in the form of a neural network. This time, however, only the primitive units represent behaviors (as they did with Method I). The higher-level units now represent *transitions* between lower level units. Instead of combining together behaviors as was done with Method I, these units represent the *degree to which one behavior should follow another* at any particular time—they therefore resemble "higher order" units to some extent (c.f. Pollack, 1991; Giles, 1992; and Watrous, 1992).

Take for example Figure 2a. In one case (position 9) the animat should go south when it senses light, while in another case (position 5), it should go north. To decide whether to move north or south after the light, it is sufficient to know whether the animat sensed heat or cold in the previous step. Therefore, a unit can be built, <SL, MN> that causes the connection weight from SL to MN to be strong after sensing heat, but to be weak after sensing coldness. (The connection weight from one unit, A, to another, B, as with Method I, indicates how much the system believes unit B should be chosen directly following unit A.) Another unit, <SL, MS>, can be built to increase the weight from SL to MS after sensing coldness and to decrease it when sensing heat.

The same thing can be done for transitions among higher-level units. For example, in figure 2b, <SL, MN> and <SL, MS> cannot be correctly predicted from the sensation of heat or cold alone, but require knowledge of what happened in the previous step. Thus, a new unit might be built <SH, <SL, MS>> that sets to a high value the weight from SH to <SL, MS> immediately

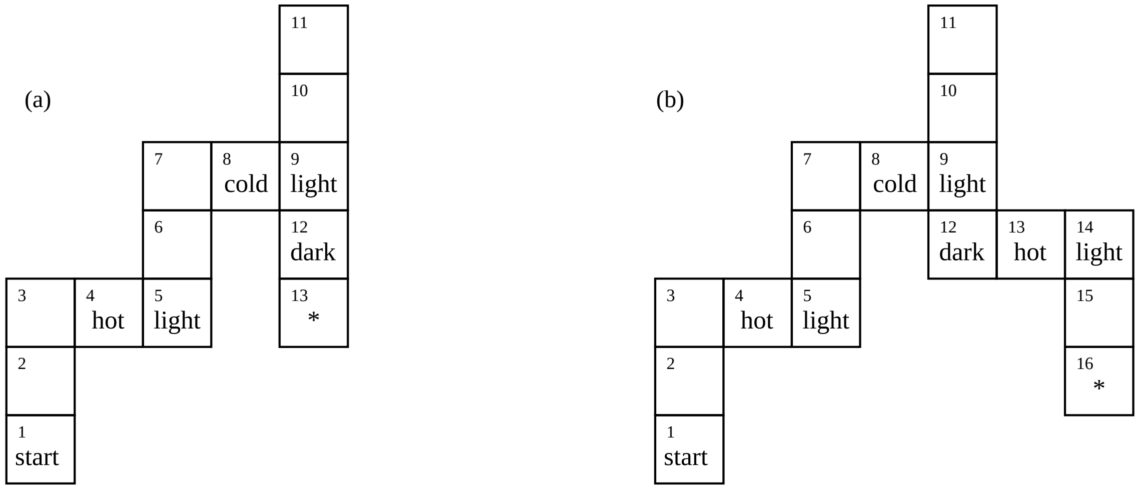


Figure 2: Maze (a) shows the need for two new units: <SL, MS>, to modify the connection from SL (sense light) to MS (move south), and <SL, MN>, to modify the connection from SL (sense light) to MN (move north). In maze (b), new units are needed to modify the connections from SH (sense heat) to <SL, MS> and to <SL, MN>.

following the sensation of darkness (position 12), but sets the weight to a low value otherwise. Clearly, this kind of hierarchy construction can continue indefinitely.

Mathematically, the activation of the units can be expressed as:

$$a_i(t+1) = \sum_j \hat{w}_{ij}(t) o_j(t)$$

$$\hat{w}_{ij}(t) = \begin{cases} w_{ij} + a_{<i,j>}(t-1) & \text{if a unit <i,j>} \\ & \text{for } w_{ij} \text{ exists} \\ w_{ij} & \text{otherwise} \end{cases}$$

The output of the unit, $o_j(t)$, is the same as its target, $T_i(t)$, and the targets are different for primitive and higher-level units. For a primitive unit, the target is 1.0 if the behavior it represents completes and 0.0 otherwise (as with Method I). For higher-level units, it is given by the learning rule:

$$\begin{aligned} \Delta w_{ij}(t) &= o_j(t-\tau_i)(T_i(t) - a_i(t)) \\ T_{<ij>}(t) &= a_{<ij>}(t) + \Delta w_{ij}(t) \end{aligned} \tag{4}$$

The value $\tau_i$ depends on how "high" in the hierarchy unit $i$ is: if $i$ is a primitive unit, $\tau_i$ is 1. If $i$ is a higher-level unit, modifying the connection from unit A to unit B, it is $1 + \tau_B$. The higher in the hierarchy unit $i$ is, the larger $\tau_i$ is, and the longer the time lag is between when the unit is activated and when its target arrives. This learning rule accomplishes actual gradient descent in the error space, unlike the learning rule of the first system.

New units are now created for different reasons from the units of Method I. If one unit is reliably activated after another, there is no reason to interfere with the connection between them. Only when the transition is *unreliable*, that is, when the connection weight should be different in different circumstances, is a unit required to predict the correct value.

In order to decide when a transition is unreliable, a statistical record is kept of each connection. Two long-term averages are maintained: the average change made to the connection, and the average *magnitude* of the change. When the average change is small, but the average magnitude is large, this implies that the learning algorithm is changing the weight of the connection back and forth by a large amount. Therefore, a criteria value, $\Theta$, is chosen arbitrarily, and when

$$\frac{\overline{|\Delta w_{ij}|}}{\epsilon + |\overline{\Delta w_{ij}}|} > \Theta$$

(where $\epsilon$ simply keeps the denominator from being zero), a new unit is created for $w_{ij}$. That is, when the average size of weight-change divided by the size of the average weight-change (plus $\epsilon$) is larger than $\Theta$ for some connection $w_{ij}$, then this connection is judged to be unreliable, and a new unit is created for it. This new unit can now attempt to learn the circumstances under which the connection weight should be one value, and when it should be another. A similar technique for creating new nodes in a non-temporal neural network is given in (Wynn-Jones, 1993).

The resulting system has much in common with the original one, but there are differences. First, this new system has no discontinuities. Higher-level units may come on to a greater or lesser degree, not just completely on or completely off. Second, the behavior hierarchies built by the new system are different from those built by the old system, and these new hierarchies are distributed; execution of a sequence of behaviors, say ME→SC→MS (move east, then if it's cold move south), might require no hierarchy at all! If this sequence always occurs whenever ME is chosen, only ME needs to be chosen, and the rest will follow (since the connection weights from ME to SC and from SC to MS will be large). If, on the other hand, there are *different* sequences that can follow ME, say SC→MS and SD→MW, (where MS always follows SC, and MW always follows SD), then these can be distinguished with new units: <ME, SC> and <ME, SD>. The sequence ME→SC→MS can then be executed by activating *both* ME *and* <ME, SC> at the same time, while *negatively* activating <ME, SD>. Thus, what took only one activation in the first system now takes three simultaneous activations, but they accomplish the same end.

Because of its distributed nature, the new system can handle situations that troubled the first system. Contingencies are a good example. Let's say the desired behavior is: move east, and if it's cold, move south, but if it's hot, move north. This can be done by activating *all* of the following units: ME, <ME, <SC, MS>>, and <ME, <SH, MN>> (while negatively activating any conflicting <ME, ...> units).

### 3.1 Reinforcement Learning

Reinforcement learning was fairly intuitive in the first system in that it spread reinforcement back over a small number of "choices" while possibly spreading it over an unlimited number of primitive behavior executions. A reinforcement learning scheme in the new system can do nearly the same thing by dynamically modifying the $\sigma$ of equation 3. This trace can be modified such that unexpected events are remembered strongest at the time of the reward by setting $\sigma$ equal to a normalized error value for the current time-step.

$$\sigma(t) = (1 - \frac{1}{n}\sum_i |T_i(t) - a_i(t)|)$$

($n$ is the number of units in the system.) The weight changes are now accumulated in $\Delta' w_{ij}$ (because of the use of $\Delta w_{ij}(t)$ in equation 4).

$$\begin{aligned} \Delta' w_{ij}(t) &= \Delta w_{ij}(t) + \sigma(t)\Delta' w_{ij}(t-1) \\ w_{ij}(t) &= \eta R(t)\Delta' w_{ij}(t) + w_{ij}(t-1) \end{aligned}$$

With this method, if everything in the current time-step is expected exactly as it occurred, the trace of weight changes, $\Delta' w_{ij}(t)$, will remain essentially the same as it was in the previous time-step, but if there is a large error, then the trace is altered significantly. This way, when a reward arrives, it applies the $\Delta' w$'s from the last few time-steps where expectations were not met, no matter how many time-steps have occurred in the mean time.

These methods of reinforcement for both Method I and Method II are really quite old-fashioned. There are many newer and better approaches based on temporal difference (TD) methods (Sutton, 1988) and dynamic programming, for example (Barto et al., 1983; Barto et al., 1989; Sutton, 1990; Barto et al., 1991), that are much more efficient. There is a major obstacle to using these newer, better approaches for reinforcement learning in the hierarchical systems presented here, however: they all depend on accurate knowledge of the state of the environment. It is conceivable that these methods could be used anyway, if one were to allow the hierarchical system to learn also to predict a discounted reinforcement signal, as is used in the TD methods. But despite some work on joining temporal difference methods with hierarchical learning, for example (Singh, 1992; Wixson, 1991), it is not immediately obvious how to tailor the TD approach to the hierarchical systems presented in this paper, (whereas it was relatively straightforward to tailor the reinforcement learning methods given above to these hierarchical systems). I am, however, currently working on doing just this and hope that TD methods allow improvement over the somewhat ad hoc reinforcement learning techniques presented above.

## 4 Results

It should be noted that the tasks described in the previous sections were presented for explanatory purposes. Certainly there are known techniques, such as recurrent neural networks, that could solve such simple tasks. The important issue that is *not* solved by recurrent neural networks is the implementation of hierarchy. In particular, one would like to have a system that is capable of learning from its environment and using the skills that it has gained so far for the purposes of solving still more difficult tasks and learning even more elaborate skills. Neural networks, on the other hand, are typically renowned for their ability to *forget* when learning something new.

Dawkins (Dawkins, 1976) presented a method of hierarchically reducing the description of symbol sequences without loss of information. This method, similar to the principles exploited here, is also used by Schmidhuber (1992) to build hierarchies within a neural network, though the network is of fixed size and not capable of indefinite extension. The Recurrent Cascade Correlation (RCC) algorithm however is not so limited. This algorithm also learns sequences by adding new units, and it has been shown capable of incremental learning (Fahlman, 1991). It does not build explicit behavior hierarchies, however, and it cannot be used for reinforcement learning because of the intricacies of its training algorithm.

Despite the fact that it was designed for tasks that required hierarchy learning, Method II has been tested on some traditional sequential tasks that did not require hierarchy learning, and it has compared favorably with other systems. One such task, reported in more detail in (Ring, 1993), was a finite state grammar learning problem that required use of previous sense information to

| Time-step: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sequence 1: | X | A | B | X | C | D | E | F | G | H | I | J | K |
| Sequence 2: | Y | A | B | Y | C | D | E | F | G | H | I | J | K |

Table 1: Training Set for "Gap" task with a gap of two.

| | Mean number of Training sets required by: | | | |
|---|---|---|---|---|
| gap | Standard recurrent net | Mozer network | Method II | Units created |
| 2 | 468 | 328 | 4 | 10 |
| 4 | 7406 | 584 | 6 | 15 |
| 6 | 9830 | 992 | 8 | 19 |
| 8 | $> 10000$ | 1312 | 10 | 23 |
| 10 | $> 10000$ | 1630 | 12 | 27 |

Table 2: Comparison of Learning Methods on "Gap" Task.

determine the correct output. For this task, reinforcement learning was *not* used—$R(t)$ was constant for all $t$. Instead, the task was supervised. That is, the targets were given by a teacher at every time-step. Traditional recurrent networks, including RCC, have been tested on the same task by other researchers (Cleeremans et al., 1989; Fahlman, 1991). The Method II algorithm learned the task approximately 100 times faster than the best results reported by the other researchers. (That is, the other networks needed to see 100 times more training examples before learning the task.) The number of new units in the network after learning completed was, however, between two and twenty times greater than the number of hidden units used in the networks reported by the other researchers.

The finite state grammar task was solvable by keeping track of information from only one time-step into the past. Method II was also tried on (again supervised) tasks that required maintaining information over a much larger time span, and the network quickly learned these tasks as well. For example, tasks of the form given in Table 1 have been given to the network to compare against the recurrent network of Mozer (Mozer, 1992). These two sequences formed a single traning set given to the network. The network was given as sensory input each element in the sequence one at a time until the end of the sequence was reached. At that point, the activations of the higher-level units were reinitialized and the second sequence was given to the network. At each time-step, the network's target output was the item of the sequence that the network would be given as input on the following time-step. Thus, at time-step 1, while experiencing sequence 2, sensory line A would be activated (i.e. set to 1.0), and all others would be quiescent (i.e. set to 0.0). The network's target output would be B. Because the network will always be asked to predict B when its input is A, this part of the task is fairly simple. However, at time-step 2, the network will see B as input and must predict either X or Y, depending upon which sequence it is currently sensing, and this information is contained two steps back in time. Thus, the network must remember what it saw at time-step 0 to correctly predict what it will see at time-step 3.

The difficulty of the task can clearly be increased by lengthening the duration between the initial X or Y stimulus at time-step 0 and its recurrence later on in the sequence. Mozer used sequences generated in this manner to compare his algorithm to traditional recurrent neural networks, which he realized were extremely poor at learning extended time delays. He compared the algorithms on sequences with gaps of between two and ten time-steps from the initial X or Y and its reappearance. The results of his tests, together with the results from Method II, are printed in table 2. Mozer noticed that the number of training sets his architecture required to learn a sequence scaled roughly linearly with the size of the gap. Method II also scaled linearly, but it learned the task approximately 100 times faster than Mozer's network. I tested the network at gaps of up to 24 and found the identical scaling behavior (i.e. 26 training set presentations were required, and 49 new units were built).

It should be noted that the parameters were optimized for these particular tasks and would not necessarily be the best parameters for a different task. Specifically, it is not always beneficial for units to be created as quickly as they were for this task. It should also be noted that Method II learns the precise number of time-steps between when information becomes available and when it will be used. It does so by creating a hierarchy that spans the gap exactly. Mozer's network, on the other hand, maintains more general knowledge and is able to keep information for use over broader periods of time. In the end, one would expect Mozer's network to generalize better than the network of Method II. Part of my current work is to provide the Method II algorithm with the same flexibility that Mozer's network has.

## 5 Summary and Conclusions

Two methods for hierarchically organizing temporal events have been described. The first is more intuitive: grouping together common sequences of events into single units so that they may be treated as individual behaviors. This system contains discontinuities, however, and it runs into difficulty when more than one unit is (or should be) active at the same time. The second system can dynamically modify the values that specify the degree to which one unit should follow another. These values are continuous, allowing the use of gradient descent during learning. Furthermore, many units are active at the same time as part of the system's normal functionings. The new system is actually more powerful than the first, while being easier to train.


## Acknowledgements

This work was supported by NASA Johnson Space Center Graduate Student Researchers Program training grant, NGT 50594. I would like to thank Kadir Liano and my advisor, Robert Simmons, for many helpful discussions. I would also like to thank Pavilion Technologies, Inc. for their generous contribution of computer time and office space required to complete much of this work.


# References


Albus, J. S. (1979). Mechanisms of planning and problem solving in the brain. *Mathematical Biosciences*, 45:247–293.

Bachrach, J. R. (1992). *Connectionist Modeling and Control of Finite State Environments.* PhD thesis, Department of Computer and Information Sciences, University of Massachusetts.

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91–57, Computer Science Department, University of Massachusetts at Amherst.

Barto, A. G., Sutton, R. S., and Anderson., C. W. (1983). Neuron-like elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846.

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1989). Learning and sequential decision making. Technical Report COINS Technical Report 89–95, University of Massachusetts at Amherst, Department of Computer and Information Science.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381.

Dawkins, R. (1976). Hierarchical organisation: a candidate principle for ethology. In Bateson, P. P. G. and Hinde, R. A., editors, *Growing Points in Ethology*, pages 7–54. Cambridge: Cambridge University Press.

Drescher, G. L. (1991). *Made-Up Minds: A Constructivist Approach to Artificial Intelligence.* Cambridge, Massachusetts: MIT Press.

Fahlman, S. E. (1991). The recurrent cascade-correlation architecture. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. San Mateo, California: Morgan Kaufmann Publishers.

Giles, C. L., Miller, C. B., Chen, D., Sun, G. Z., Chen, H. H., and Lee, Y. C. (1992). Extracting and learning an unknown grammar with recurrent neural networks. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 317–324. San Mateo, California: Morgan Kaufmann Publishers.

Mozer, M. C. (1992). Induction of multiscale temporal structure. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 275–282. San Mateo, California: Morgan Kaufmann Publishers.

Pollack, J. B. (1991). The induction of dynamical recognizers. *Machine Learning*, 7:227–252.

Ring, M. B. (1991). Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In Birnbaum, L. A. and Collins, G. C., editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 343–347. Morgan Kaufmann Publishers.

Ring, M. B. (1993). Learning sequential tasks by incrementally adding higher orders. In Giles, C. L., Hanson, S. J., and Cowan, J. D., editors, *Advances in Neural Information Processing Systems 5*, pages 115–122. San Mateo, California: Morgan Kaufmann Publishers.

Roitblat, H. L. (1991). Cognitive action theory as a control architecture. In Meyer, J. A. and Wilson, S. W., editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 444–450. MIT Press.

Schmidhuber, J. (1992). Learning unambiguous reduced sequence descriptions. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 291–298. San Mateo, California: Morgan Kaufmann Publishers.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3/4).

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

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Porter, B. W. and Mooney, R. J., editors, *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann Publishers.

Watrous, R. L. and Kuhn, G. M. (1992). Induction of finite-state languages using second-order recurrent networks. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*, pages 309–316. San Mateo, California: Morgan Kaufmann Publishers.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, pages 96–104. IRE. New York.

Wilson, S. W. (1989). Hierarchical credit allocation in a classifier system. In Elzas, M. S., Ören, T. I., and Zeigler, B. P., editors, *Modeling and Simulation Methodology*. Elsevier Science Publishers B.V.

Wixson, L. E. (1991). Scaling reinforcement learning techniques via modularity. In Birnbaum, L. A. and Collins, G. C., editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 368–372. Morgan Kaufmann Publishers.

Wynn-Jones, M. (1993). Node splitting: A constructive algorithm for feed-forward neural networks. *Neural Computing and Applications*, 1(1):17–22.