Totally Model-Free Reinforcement Learning by Actor-Critic Elman Networks in Non-Markovian Domains

Eiji Mizutani and *Stuart E. Dreyfus*
eiji@biosys2.me.berkeley.edu, dreyfus@cimsim.IEOR.berkeley.edu
Dept. of Industrial Engineering and Operations Research, University of California at Berkeley, USA

Abstract

In this paper we describe how an **actor-critic reinforcement learning** agent in a non-Markovian domain finds an optimal sequence of actions in a **totally model-free** fashion; that is, the agent neither learns transitional probabilities and associated rewards, nor by how much the state space should be augmented so that the Markov property holds. In particular, we employ an Elman-type recurrent neural network to solve non-Markovian problems since an Elman-type network is able to **implicitly** and **automatically** render the process *Markovian*.

A standard "actor-critic" neural network model has two separate components: the action (actor) network and the value (critic) network. In animal brains, however, those two presumably may not be distinct, but rather somehow entwined. We thus construct one Elman network with two output nodes: actor node and critic node, and a portion of the shared hidden layer is fed back as the context layer, which functions as a history memory to produce **sensitivity** to non-Markovian dependencies.

The agent explores small-scale three and four-stage triangular path-networks to learn an optimal sequence of actions that maximizes total value (or reward) associated with its transition from vertex to vertex. The posed problem has deterministic transition and reward associated with each allowable action (although either could be stochastic) and is rendered non-Markovian by the reward being dependent on an earlier transition. Due to the nature of *neural model-free learning*, the agent needs many iterations to find the optimal actions even in small-scale path problems.

I. INTRODUCTION

Learning from reinforcement is a trial-and-error learning scheme whereby a computational agent learns to perform an appropriate action by receiving evaluative feedback through interaction with the world (or environment) without explicit instructions concerning correct actions. In this paper, we specifically mean **temporal difference (TD) reinforcement learning** (or **neuro-dynamic programming** [3]) simply by "reinforcement learning." This rules out a trial-and-error search strategy with no TD learning mechanisms, such as a standard genetic algorithm.

A learning agent may encounter a situation where it struggles to learn an optimal policy due to **non-Markovian** characteristics, which cannot be detected by the current

This paper will appear in Proceedings of the IEEE World Congress on Computational Intelligence (Wcci'98 / IJCNN'98 May, 1998).

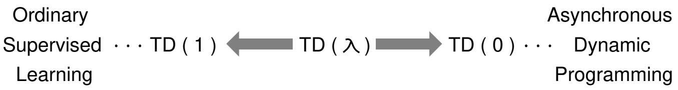

Fig. 1. *TD learning spectrum. TD(λ) migrates various degrees of the TD learning spectrum according to the value of λ, between two important computational schemes: "supervised learning" and "dynamic programming."*

perception alone, since they are correlated with certain past events. This is the so-called non-Markov (hidden) state problem. The agent in a non-Markovian domain cannot take an optimal next action by observing its current input only.

A variety of approaches toward non-Markovian problems have been discussed in the spirit of *TD reinforcement learning* [18; 13; 9; 15]. One of the approaches to make the agent manage non-Markovian situations is the *memory approach*, in which the agent retains **internal state** over time. The agent with internal state can distinguish between each different past history, and therefore can become **sensitive** to the non-Markovian dependencies that cause hidden states. When we focus on neural network (NN) function approximators, this requirement may be realized by networks with **context units** embedded in their own architectures, such as an Elman network [7] and a Jordan network [11]. *Note that, in any environment, the outcome at any given time may be affected arbitrarily by prior events. The agent may not know which prior events matter, and therefore the agent does not know how to enlarge the state space to render the process Markovian.* The context units, however, **implicitly** and **automatically** encode the history of the entire past, as far back as it goes. If the agent tries to **explicitly** learn what prior events matter, we would consider this **model-building** and not call such an agent **totally model-free**. Notice that the classical dynamic programming (DP) procedure requires an **explicit** state description to allow a DP solution based on the *principle of optimality* [6].

TD methods are a class of incremental learning procedures for prediction whereby credit is assigned based on the difference between temporally successive predictions [19]. Figure 1 summarizes the TD method, which compromises two extreme schemes: "ordinary supervised learning" and "asynchronous dynamic programming." In particular, we employ TD(0), in which only the most recent prediction affects weight parameter alterations. The TD method is

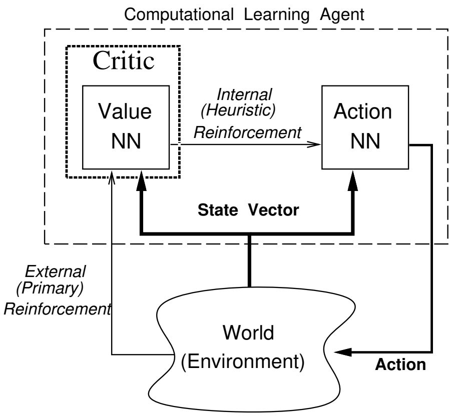

Fig. 2. *An ordinary AC architecture that uses neural network function approximators: the value NN and the action NN.*

employed to solve the temporal credit assignment problem in the predictor (or critic) part of *actor-critic* models [2].

II. ACTOR-CRITIC LEARNING

Two basic representative schemes for TD reinforcement learning are **Q-learning** [20] and **actor-critic(AC)** learning. Q-learning based on look-up tables has been proven to converge to optimal values and actions [22], whereas AC-learning has *not* been, except for one complex modified AC algorithm [4]. Besides the convergence theorem, there are fundamental differences in those two learning mechanisms. To seek the best policy, a Q-learning agent needs to learn *how bad* each bad action is. On the other hand, an AC agent does not pursue *how bad* the bad actions are; instead, it merely seeks to discover and evaluate good actions alone. As a consequence, when look-up tables are used for solving general *multiple-action* problems, an AC-learning agent can be much less memory intensive. Also, *such an AC agent's simpler way of adjusting behavior seems to us closer than Q-learning to what animal brains do.* Hence, we focus on AC concepts in this paper, although most of the previous work focused on Q-learning [13; 24].

A. An Adaptive Neural Critic Algorithm

The neural AC architecture usually consists of two NNs: the value NN and the action NN. The value NN approximates evaluation functions, mapping states to estimated values, whereas the action NN generates a plausible (or legal) action, mapping states to actions [12]. Figure 2 illustrates such AC networks. The adaptive critic receives external (primary) reinforcement from the world and transforms it into internal (heuristic) reinforcement. The critic is adaptive because its predictor component (value NN) is updated using TD methods. The action NN *attempts* to learn optimal control or decision-making skills. It does so by choosing actions probabilistically to produce **exploration**, hopefully converging to optimal actions with probability one. In this framework, an AC agent *attempts* to find both *optimal actions* and *optimal values*.

The weight update rule follows the usual error minimization scheme used in supervised learning by defining the squared error $E_{\mathrm{TD}} = \frac{1}{2} error^2$, where

$$
\begin{aligned}
error \;\equiv\; & \text{(value incurred by a selected action)} \\
+ \; & \gamma\text{(estimated value computed by value NN of observed successor state)} \\
- \; & \text{(estimated value computed by value NN of current state)},
\end{aligned}
$$

where γ is a discount rate. Both the action NN and the value NN are trained simultaneously using E_{TD}. The following procedure explains an implementation of the AC concept.

1. Observe the current state: $s \leftarrow$ current state s_n.
2. Use the value NN to compute current estimated value $V(s)$: $e \leftarrow V(s)$.
3. Select an action a_n by using the output of the action NN.
4. Execute the action a_n.
5. Observe the successor state t_n and reinforcement r_n.
6. Use the value NN to compute $V(t_n)$.
7. $E \leftarrow r_n + \gamma\ V(t_n)$.
8. Adjust the value NN by backpropagating the error ($= E - e$).
9. Adjust the action NN according to the error.

□

This procedure demonstrates how to use the output of a value NN to control the output of an action NN by means of backpropagation. To be precise, suppose we want to maximize the estimated value (i.e., reward). Each action should be chosen to maximize the sum of all future rewards. Thus given an action a_n, if the value of the next state $V(t_n)$ [or $\gamma\ V(t_n)$] plus the external reinforcement r_n is greater than the value of the current state $V(s_n)$ (i.e., $E - e > 0$), the action looks good, and therefore, that action should be reinforced. Conversely, if the reverse inequality holds, the action looks bad. (This is because we want to maximize the cumulative future rewards.) The action should therefore be inhibited.

B. SMALL-SCALE PATH PROBLEMS

To make the aforementioned AC concept more concrete, we consider a small-scale path problem configured in the triangular three-stage path network illustrated in Figure 3. It is a deterministic discrete-action environment, and the transition from vertex to vertex incurs a reward. The agent's objective, when the discount factor γ is one (which we assume), is to learn the optimal sequence of three actions that maximizes the total (reward) value (i.e., the sum of the values associated with each of the three steps plus the terminal value). We assume that the agent always starts at vertex A and moves diagonally to the right. The agent chooses either "action d"* (going diagonally downward) or "action u" (going diagonally upward) at each vertex. The

*Since the process is *deterministic*, by action d, an agent *always* goes diagonally downward, although the model-free learning agent neither knows nor uses this fact during its learning.

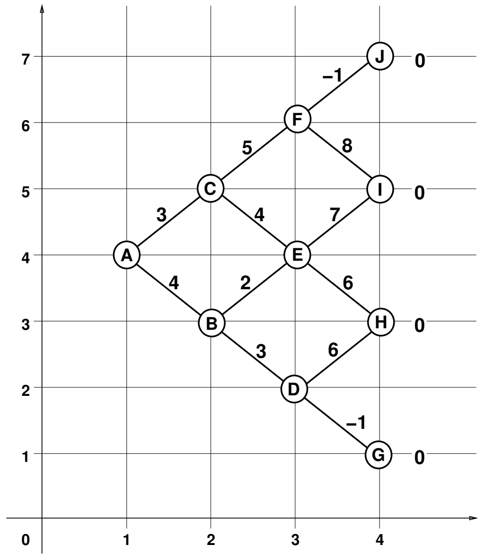

Fig. 3. *The configuration of a three-stage path problem in a coordinate system.*

environment accordingly informs the agent of the next vertex and the transitional value associated with the action taken. This process is repeated three times. For the third action, the terminal value provided by the world (environment) is used as the value $V(t_n)$ rather than the NN's output. This is the realization of the boundary conditions. Moreover, the process is *non-Markovian*, due to the following *additional reward rule* (or *bonus-rule*, in short):

> *If the third transition is the same as the first transition chosen, an additional value (bonus) of "9" is added to the transitional (reward) value traversed on the third step of the path to generate the value presented to the agent. If the third transition does not match the first transition, no bonus is accrued.*

This description of the "bonus-rule" should be viewed as a specialization for a "three-stage" problem. This rule can be generalized as "*a bonus is added if the transition at any stage matches the transition two stages before.*" It will be applied later to the four-stage problem.

Suppose the agent initially chooses action d, then u, then d. After the first action d, the transitional value "4" and the coordinates (2,3) at vertex B are presented to the agent. Likewise, after the second action u, the transitional value "2" and the coordinates (3,4) at vertex E are informed. At this stage, the accumulated value is "6 ($\leftarrow$ 4 + 2)." After the third action d, the value "15 ($\leftarrow$ 6 + bonus 9)" is presented, as well as the terminal value "0." Hence, this action sequence "*d-u-d*" gives the total (reward) value "21" (see Path 1 in Table 1).

Similarly, another action sequence "*u-u-d*" gives the total value "16" (Path 2 in Table 1), which was the maximum total value before the bonus-rule was introduced. Yet, due to the bonus-rule, another sequence "*u-d-u*" gives the maximum possible total value "23" (see Path 3 in Table 1). Of course, the bonus-rule and the incurred-reward data used

Table 1. *Sample paths followed by agent's sequence of three actions. The bonus value is set equal to "9."*

Example path								Total value
Path 1	A	$\xRightarrow{down}$ 4	B	$\xRightarrow{up}$ 2	E	$\xRightarrow{down}$ 6 + bonus	H	21
Path 2	A	$\xRightarrow{up}$ 3	C	$\xRightarrow{up}$ 5	F	$\xRightarrow{down}$ 8	I	16
Path 3	A	$\xRightarrow{up}$ 3	C	$\xRightarrow{down}$ 4	E	$\xRightarrow{up}$ 7 + bonus	I	23

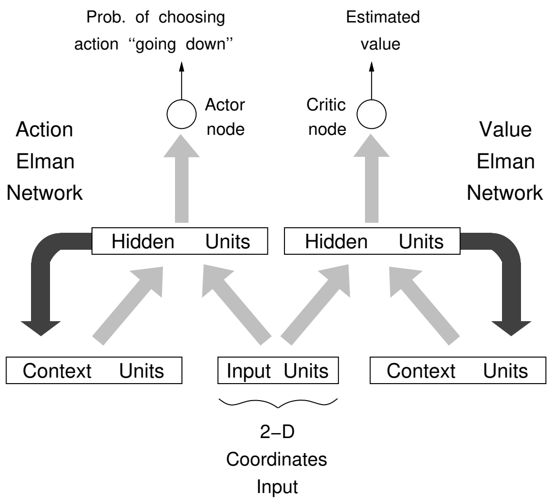

Fig. 4. *Two Elman networks that implement the actor-critic concepts. Darker-shaded feedback arrows have fixed weights of 1.0.*

by the environment are unknown to the agent, and are not explicitly learned during the procedure. (Concerning a similar path problem without the bonus-rule, see *Chapter 10 "Learning from reinforcement"* in ref. [10].)

The agent is in a non-Markovian domain because, in the path network pictured in Figure 3, the agent at vertex E does not know the previous vertex (B or C) just by observing the current state (vertex E) alone, which is crucial in making an optimal action in light of the bonus-rule. Whitehead and Ballard discussed a similar problem in world state representations [23]; the mapping from world states to the agent's state representations can be many-to-many in a complex system environment. A single state representation may represent multiple world states. They called this overlapping between the world and the agent's state representation "perceptual aliasing." By extension, an autonomous mobile robot should be able to deal with incorrect state descriptions generated by limited sensors [14].

C. *Actor-Critic Elman Networks for Solving Non-Markovian Problems*

For solving non-Markovian problems, we can use two distinct Elman-type networks for the value NN and action NN. Figure 4 illustrates this architecture. The context units **implicitly** encode the past history. Each different path to any vertex will have different context-unit values. Thus, this type of NN will be able to generate the *sensi-*

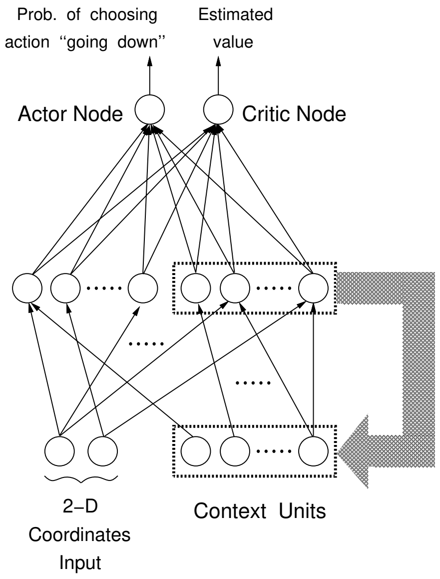

Fig. 5. *An actor-critic Elman network that implements the actor-critic learning concepts. Portion of the hidden activations (in dotted lines) is fed back to the context units via recurrent connections (shaded arrows), which have fixed weights of 1.0.*

tivity † of value and action to whatever in the prior path history is potentially relevant to environmental dynamics and reinforcement.

Another architecture shares the hidden and context layers between "actor" and "critic" modules. Figure 5 depicts such an architecture, in which there are two different output neurons. One output neuron is the so-called "critic" node, producing the current estimated total value, which indicates the expected quality of performance. The second neuron is the "actor" node, generating the current probability of action d "going down." This neuron has an activation function constrained to lie between zero and one. That architecture (Figure 5) combines the value NN and the action NN, which may be closer to biological neural networks in animal brains than the two separate network architecture shown in Figure 4. Furthermore, in the ordinary Elman-network, the entire hidden activations are fed back to the context layer. As the number of hidden nodes increases, that of context nodes increases. However, to capture history features, it may not be necessary to use the entire hidden layer as the context layer; only some portion of the hidden layer can be used as the context layer [13]. This idea is illustrated in dotted lines in Figure 5.

III. MODEL-FREE LEARNING

From the standpoint of environment (or world) modeling, reinforcement learning is roughly classified into the following two types:

1. Learning a world model by trial-and-error interaction with the world, and then basing optimal actions and values on the learned world model;
2. Learning optimal actions and values by trial-and-error interaction with the world without attempting to learn a world model.

Method 1 is **model-based** or **indirect** learning. Method 2 is **model-free** or **direct** learning.

Method 1 corresponds, in conformity with control engineering terminology, to a **system identification** procedure to form a world model [1; 5]. The agent is assumed to be able to observe states, actions, and reinforcement signals. It can therefore model the mapping from *actions* to *reinforcement signals*. More precisely, the world model is intended to model the input-output behavior of the dynamic world (or environment); given a state and an action, it is supposed to predict the received resultant reinforcement and next state [12; 20]. Method 1 describes a sequential training strategy whereby the world model is trained first and then frozen. Alternatively, it can be trained simultaneously with action and value NNs [8; 17].

Classical DP is categorized as a *model-based* approach because it uses world models, such as a transition model and a reward (or cost) model. To solve the aforementioned path problem with the bonus-rule, classical DP requires that one increase the arguments in the optimal value function so that the Markov property holds. Choosing proper arguments for the value function corresponds to enlarging the state space to make the process Markovian [6]. Those arguments must **explicitly** define the appropriate amount of information that includes the bonus-rule.

In contrast, model-free (Method 2) learning of optimal actions and values can be viewed as *modern DP* (like incremental DP [21]) because the scope of DP is basically characterized by seeking the optimal value function **asynchronously** and by using a backup property of relating an estimated value at a point to estimated values at the next points **successively**. This is a modern perspective of DP in the spirit of model-free learning, as seen in Q-learning.

In light of our path problem, two agents based on methods 1 and 2 can be rephrased as follows:

- The *model-based learning agent* either knows all relevant probabilistic information about *transitions and rewards* and knows how this information depends on past events, or estimates this information through observation to determine actions and values by a suitable version of classical dynamic programming or a similar optimization technique.
- The *model-free learning agent* attempts to construct policy and evaluation functions directly with no ability to predict transitions or immediate rewards resulting from its performed actions with no explicit memory other than memory of the current and the next state.

IV. SIMULATION

We first applied actor-critic Elman networks (Figures 4 and 5) to the three-stage path network illustrated in Figure 3. We then considered a four-stage path network, as shown in Figure 6. In the latter four-stage problem, the agent has

†This goal could also be achieved by a more animal-like dynamical neural model [8] with the change in neural activations assumed to be a function of current activations and net input from connected neurons. We employ an Elman network because it permits the efficient use of the backpropagation learning algorithm.

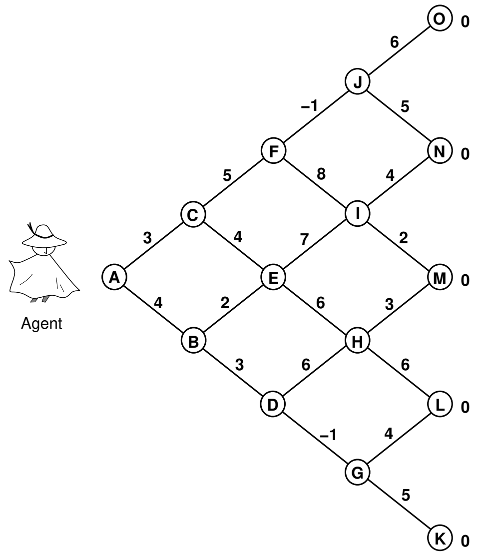

Fig. 6. *The configuration of the four-stage path problem.*

more chance to obtain "bonus" since *if the fourth transition is the same as the second transition chosen, an additional value (bonus) of "9" is accrued.* In the four-stage problem, the following path gives the double bonus and produces the maximum total value "34":

$$\mathrm{A} \underset{3}{\overset{up}{\Longrightarrow}} \mathrm{C} \underset{4}{\overset{down}{\Longrightarrow}} \mathrm{E} \underset{7+\text{bonus}}{\overset{up}{\Longrightarrow}} \mathrm{I} \underset{2+\text{bonus}}{\overset{down}{\Longrightarrow}} \mathrm{M}$$

For this four-stage problem, the input states are represented as two-dimensional coordinates in the same way as depicted in Figure 3; for instance, vertex H is expressed as (4,3) in the coordinate system.

Table 2 presents three experimental architectures of the actor-critic Elman networks—X, Y, and Z. They were trained by TD(0) in conjunction with the well-known *backpropagation learning rule with a momentum term* (0.8). Tables 3 and 4 show the results obtained by those networks. Due to the exploration property, action choice is stochastic during learning. Each value may not necessarily improve at each backup because the estimated value will be worse if a state has a correct value and the next state has a wrong one; the value will then temporarily go wrong. Only the average of many (wrong) values may be right, or may approach correct if the learning rate goes to zero appropriately. Since we employed a small fixed learning rate in the simulation, the value at each vertex in Tables 3 and 4 is averaged over the last 1,000 epochs.

If the value of the next state $V(t_n)$ plus the external reward r_n was *larger* than the value of the current state $V(s_n)$ (i.e., if the TD error was *positive*), the action looked *good* because the agent wanted to *maximize* the cumulative values (rewards), and therefore the probability of choosing it was increased. Conversely, if the TD error was negative, the action was inhibited. Through repeated passes, the actor-critic Elman nets learned the optimal actions as well as the estimated maximum values.

Table 2. *Three experimental actor-critic Elman network architectures.*

AC nets	Architecture	Hidden units	Context units	Total parameters
X	Action Elman net	5	5	108
	Value Elman net	5	5	
Y	Two-output Elman net	10	10	165
Z	Two-output Elman net	11	8	145

Table 3. *Results of the three-stage path problem obtained by three actor-critic Elman nets whose architectures X, Y, and Z, are presented in Table 2. P_{down} denotes "probability of action d" (i.e., the output of the actor node).*

AC nets	Estimated values	Vertex A	Vertex C	Vertex E	Required epoch
X	P_{down}	0.025	0.943	0.063	52,000
	Rewards	22.013	19.174	15.415	
Y	P_{down}	0.031	0.942	0.053	138,000
	Rewards	22.110	19.298	15.565	
Z	P_{down}	0.025	0.950	0.055	100,000
	Rewards	22.129	19.301	15.426	
Target values	P_{down}	0.0	1.0	0.0	–
	Rewards	23.0	20.0	16.0	

Table 4. *Results of the four-stage path problem obtained by three actor-critic Elman nets X, Y, and Z (see Table 2). P_{down} denotes "probability of action d" (i.e., the output of the actor node).*

AC nets	Estimated values	Vertex A	Vertex C	Vertex E	Vertex I	Required epoch
X	P_{down}	0.037	0.923	0.0248	0.9513	2,000,000
	Rewards	33.135	30.340	26.473	10.629	
Y	P_{down}	0.042	0.983	0.046	0.930	3,940,000
	Rewards	33.150	30.356	26.496	10.581	
Z	P_{down}	0.028	0.986	0.059	0.940	2,055,000
	Rewards	33.133	30.273	26.311	10.613	
Target values	P_{down}	0.0	1.0	0.0	1.0	–
	Rewards	34.0	31.0	27.0	11.0	

V. DISCUSSION

We implemented *totally model-free AC-learning* on the basis of Elman-net function approximators.

Tables 3 and 4 show that the networks provided outputs close to the desired values but converged *very slowly*. This characterizes *neural* totally model-free learning. Clearly, when the path-network was expanded from three-stage to four-stage, the required iterations were dramatically increased. This can be seen in the column "required epoch" in Tables 3 and 4. It should also be noted that the reward data (shown in Figures 3 and 6) and the parameter setups (learning rates and number of neurons) affected learning behaviors. Choosing proper parameters are still state-of-the-art and found by the process of trial and error. Some choices appeared to lock-in on non-optimal solutions, although the possibility of finding the optimal path after additional epochs cannot be ruled out; see Figure 7 for the sample AC-learning curve for the four-stage problem.

For our particular problem and Elman-type recurrent connections, two separate Elman-networks (AC-net X in Table 2 or Figure 4) appeared to be able to capture his-

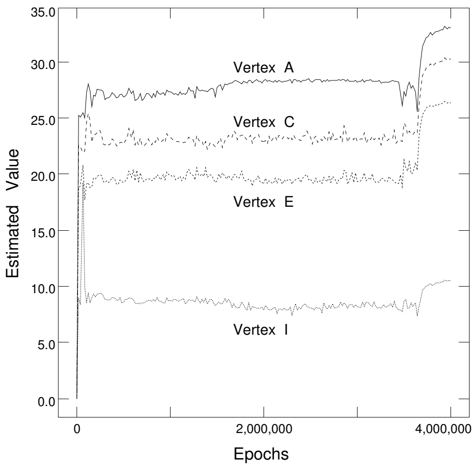

Fig. 7. *A sample AC-learning curve for the four-stage problem. An AC agent happened to lock-in on a sub-optimal solution by taking an action sequence "d-d-d-d," which produced an estimated value close to "29" at vetex A. After many excursions, however, the AC agent found the optimal action sequence "u-d-u-d," which produced an estimated value close to the maximum value "34" at vertex A. The value at each vertex is averaged over the last 1,000 epochs.*

tory features more quickly than one actor-critic Elman network with two output units (AC-nets Y and Z in Table 2 or Figure 5). This may be because the two outputs have completely different qualities: "action probability" and "estimated value," and thus those outputs may interfere with each other during learning, rather than help each other to grasp certain history features.

The discussed AC-learning concept can be presumably applied to making agents sensitive to some past events in a more practical setting; for instance, in learning *vehicle control actions* based on changing road lane marker trajectory [16].

VI. Conclusion

With an appropriate use of recurrence, the behavior of a recurrent-network agent can always be made sensitive to past history during its execution of a sequential task. Anything about past history that affects subsequent reinforcement or transition will potentially produce different learned behavior, whereas anything not relevant to subsequent reinforcement or transition will not do so. In this manner, optimal behavior can be learned for processes that are non-Markovian in current observable states without explicitly learning what enlargement of state would render the situation Markovian. This finding, together with the well-known fact that TD reinforcement learning need not know or learn a model for dynamics and reward in a Markovian environment, renders this learning *totally model-free*.

References

[1] A. G. Barto and S. P. Singh. On the computational economics of reinforcement learning. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proceedings of the 1990 Connectionist Models Summer School*, pages 35–44, San Mateo, CA., 1990. Morgan Kaufmann.

[2] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):834–846, 1983.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts, 1996.

[4] R. H. Crites and A. G. Barto. An actor/critic algorithm that is equivalent to Q-learning. In *Advances in Neural Information Processing Systems 7*, pages 401–408, San Mateo, CA, 1995. Morgan Kaufmann.

[5] J. del R. Millan and C. Torras. A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8(3):363–393, May 1992.

[6] S. E. Dreyfus and A. M. Law. *The art and theory of dynamic programming*, volume 130 of *Mathematics in Science and Engineering*. Academic Press Inc., 1977.

[7] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[8] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the theory of neural computation*. Addison-Wesley, Reading, MA, 1991.

[9] T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement learning algorithms for partially observable Markov decision. In *Advances in Neural Information Processing Systems 7*, pages 345–352, San Mateo, CA, 1995. Morgan Kaufmann.

[10] J.-.S. Roger Jang, C.-T. Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing: a computational approach to learning and machine intelligence*. Prentice Hall, Upper Saddle River, NJ, 1997.

[11] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 531–546, 1986.

[12] Long-Ji Lin. Self-improving reactive agents: Case studies of reinforcement learning frameworks. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 297–305, 1990.

[13] Long-Ji Lin and Tom M. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, School of Computer Science, Carnegie Mellon University, 1992.

[14] S. Mahadevan and J. Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In L. A. Birnbaum and G. C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 328–332, San Mateo, CA., 1991. Morgan Kaufmann.

[15] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, 1995. revised in 1996.

[16] E. Mizutani, T. Kozek, and L.O. Chua. Road lane marker extraction by motion-detector CNNs. In *Proceedings of the IEEE World Congress on Computational Intelligence (Wcci'98)*, Alaska, USA, May 1998. (will appear).

[17] J. Schmidhuber. Learning algorithmes for networks with internal and external feedback. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proceedings of the 1990 Connectionist Models Summer School*, pages 52–61, San Mateo, CA., 1990. Morgan Kaufmann.

[18] J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In R. P. Lippmann, J. E. Moody, and D. J. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 500–506, San Mateo, CA, 1991. Morgan Kaufmann.

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

[20] R. S. Sutton. Reinforcement learning architectures for animats. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 288–296, 1990.

[21] R. S. Sutton. Planning by incremental dynamic programming. In L. A. Birnbaum and G. C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 353–357, San Mateo, CA., 1991. Morgan Kaufmann.

[22] C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[23] S. D. Whitehead and D. H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7:45–83, 1991.

[24] Steven D. Whitehead and Long-Ji Lin. Reinforcement learning of non-Markov decision process. *Artificial Intelligence*, 73:271–306, November 1995.