Multiobjective Reinforcement Learning Using Adaptive Dynamic Programming And Reservoir Computing

Mohamed Oubbati, Timo Oess, Christian Fischer, and Günther Palm

Institute of Neural Information Processing, 89069 Ulm, Germany.

Abstract. This paper introduces a multiobjective reinforcement learning approach which is suitable for large state and action spaces. The approach is based on actor-critic design and reservoir computing. A single reservoir estimates several utilities simultaneously and provides their gradients that are required for the actor enabling an agent to adapt its behavior in presence of several sources of rewards. We describe the approach in theoretical terms, supported by simulation results.

1 Introduction

In recent years, there has been an increasing interest in extending reinforcement learning (RL) techniques to multiobjective problems (MOP) [1]. In a single-objective RL the aim is to optimize one objective that is expressed as a function of a scalar reward, whereas in multiobjective RL (MORL) objectives are expressed as a vector with a reward element for each objective. The aim of any MORL method is to select policies that produce suitable trade-offs between the several objectives. A “good” trade-off can be defined in terms of Pareto dominance [2]: any feasible solution has to be nondominated by other solutions, i.e. there exists no other solution that will make an improvement in one objective without causing a degradation in at least one other objective. The set of nondominated solutions (also known as noninferior or Pareto optimal solutions) is denoted as the Pareto set, and its image in the objective space as Pareto front.

The majority of MORL algorithms proposed so far convert the MOP into a single-objective task and then find one optimal solution (e.g. [3, 4]). This is the easiest way to handle MOPs, but it is very unlikely to have a single solution that satisfies all objectives simultaneously. Moreover, this approach needs *a priori* information from the decision maker (e.g. defining the objective-ordering, or specifying objective weights), which is also a drawback, because the decision maker does not necessarily know the complexity of the learning process a priory, and may have too optimistic or pessimistic expectations. Alternative approaches are those that produce a set of solutions, also called pareto-based approaches (e.g. [5, 6]). Within pareto-based approaches the decision maker provides his input *a posteriori*. Although *a posteriori* decision may provide the decision maker with better insight about the relationships between the objectives, a disadvantage might be that the decision maker has to analyze a large amount of information and to choose one solution from too many feasible ones. Despite the important advances during the last years, research on MORL is still in its infancy, and there exist no study comparing the performance of different algorithms, and no standard test benchmarks are available for such a study [1]. Moreover, when considering continuous state spaces, most of the existing MORL approaches will suffer from the the commonly known "curse of dimensionality" [7], i.e. computational complexity increases exponentially with dimensionality of the application or the size of the state space.

The framework of adaptive dynamic programming (ADP) [8] adresses the "curse of dimensionality" by approximating the utility[1] using a system called "Critic" (usually a neural network). The decision-making process is usually reduced to a simple directive: "maximizing the *expected utility*" [9]. The utility is supposed to be related to the agent's preference, and expectation corresponds to something like the agent's belief related to the outcomes after performing a given action. Although considerable advances have occurred within ADPs (a recent review can be found in [10]), powerful computational tools that perform in real-time are still required.

Recurrent neural networks (RNNs) emerge as efficient and very promising tools to be implemented within ADPs. RNNs are universal approximators of dynamical systems [11] and can, indeed, be trained to approximate the expected utility. Moreover, they can exhibit continuous dynamics which is a suitable property to manage continuous state/action spaces that are quite common in real environments. Recently, we introduced a single objective RL approach in high dimentional spaces using ADP and the framework of reservoir computing (RC)[12]. RC consists of using a non-trainable RNN (the reservoir) that transforms the input stream into a high-dimensional states. Only a readout layer is then trained to map these states to a desired output. This concept reduces the complexity of training while preserving the recurrent property of the network. The proposed RC-ADP is a model-free approach, which uses RC to perform real-time estimation of the value function, and to calculate the gradients required for the actor [13, 14].

This paper aims to extend the RC-ADP for MOPs. A single reservoir estimates several utilities simultaneously and provides their gradients that are required for the actor enabling an agent to adapt its behavior in presence of several sources of rewards. The remainder of the paper is organized as follows. Sections 2 introduces the reservoir computing approach, and section 3 explains the extension of RC-ADP for MOPs. In section 4 we show our first preliminary results, and section 5 gives a general conclusion.

2 Reservoir Computing

Originally, reservoir computing has been introduced with two similar architectures: Echo State Networks (ESN) [15] and Liquid State Machines [16]. While most liquid state machines use spiking integrate-and-fire neurons with a dynamic synaptic connection model in the reservoir, the ESN standardly uses continuous valued sigmoid neurons. In this paper we adopt the ESNs as the main learning tool. An ESN has K inputs, a dynamic reservoir (DR) which contains N neurons and L output neurons. Activations of input neurons at time step k are $U_{in}(k) = (u_1(k), u_2(k), \ldots, u_K(k))$, of internal neurons are $X(k) = (x_1(k), \ldots, x_N(k))$, and of output neurons are $Y(k) = (y_1(k), \ldots, y_L(k))$. Weights for the input connection in a $(N\text{x}K)$ matrix are $W_{in} = (w_{ij}^{in})$, for the internal connection in a $(N\text{x}N)$ matrix are $W = (w_{ij})$, and for the connection to the output neurons in an L x $(K + N + L)$ matrix are $W_{out} = (w_{ij}^{out})$, and

[1] In computational intelligence, the term utility is often interpreted as a reward.

in a $(N\text{x}L)$ matrix $W_{back} = (w_{ij}^{back})$ for the connection from the output to the internal neurons. The activation of the reservoir neurons is updated according to

$$X(k+1) = f(W_{in}U_{in}(k+1) + WX(k) + W_{back}Y(k+1)) \tag{1}$$

where $f = (f_1, \ldots, f_N)$ are the internal neurons output sigmoid functions, and the input weights W_{in}, the reservoir weight matrix W and the output backpropagated weights W_{back} are generate randomly. The outputs are calculated as

$$Y(k+1) = f_{out}(W_{out}(U(k+1), X(k+1), Y(k))) \tag{2}$$

An essential condition for successful using of ESN is the "echo state" property. It is a property of the network prior to training, related to the weight matrices (W^{in}, W, W^{back}). A network (W^{in}, W, W^{back}) has echo states, if the current reservoir state $X(k)$ is uniquely determined by the history of the input/output data. The following procedure seems to give a practical solution to guaranty echo state property [15]:

1. The order of input and output neurons should be stated according to the task at hand.
2. Generate randomly the input weights W_{in} and output backpropagated weights W^{back}.
3. Generate randomly an internal weight matrix W_0.
4. Normalize W_0 with its spectral radius λ_{max} and put it in $W_1 : W_1 = \frac{1}{|\lambda_{max}|}W_0$.
5. Scale W_1 with a factor $0 < \alpha < 1$ and put the new internal matrix $W = \alpha W_1$ (in the remaining of this paper α is called the spectral radius).

If the echo state condition is met, only weights connections from the reservoir to the output (W_{out}) are to be adjusted.

2.1 Training

One simple way to train W^{out} is to use the least square (LS) method. It consists of the following steps [1]:

1. Compute the network states by presenting T input/output training sequence $(u(k), d(k))$:
$$X(k) = f(W_{in}U_{in}(k) + WX(k-1)) \tag{3}$$
where $k = 1, \ldots, T$.
2. Collect at each time the state $X(k)$ as a new row into a state collecting matrix M, and collect similarly at each time the sigmoid-inverted teacher output $tanh^{-1}D(k)$ into a teacher collection matrix C. After these collections, the matrix M has the size of $(T+1) \times (K+N+L)$, and the matrix C has the size of $(T+1) \times L$.
3. Adjust the output weights: Compute the pseudoinverse of M and put:
$$W^{out} = (M^{-1}C)^t \tag{4}$$
t: indicates transpose operation.

[1] The implementations in this paper use no back-connection W^{back} from the output to the reservoir and no connections from the input directly to the output.

The ESN is now trained off-line. For exploitation, the network can be driven by new input sequences and using equations (1) and (2).

Most of the time, however, it is computationally more efficient if we update the estimates in (4) recursively. The LS algorithm above can be extanded to the Recursive version (RLS). The recursive update of W^{out} is given by:

$$W^{out}(k) = W^{out}(k-1) + L(k)(Y_{desired}(k) - Y(k)) \tag{5}$$

where $Y_{desired}(k)$ is the desired mapping. The gain vector $L(k)$ is updated as

$$L(k) = P(k)X(k) = P(k-1)X(k)\left(1 + X^t(k)P(k-1)X(k)\right)^{-1} \tag{6}$$

and

$$P(k) = \left(I - L(k)X^t(k)\right)P(k-1) \tag{7}$$

$P(k)$ is usually referred to as the covariance matrix.

3 Multiobjective RC-ADP

We consider the coupled agent-environment as one dynamical system described by

$$s(k+1) = F[s(k), a(k)] \tag{8}$$

where $s \in \Re^n$ represents the state, $a \in \Re^m$ denotes the control action to make transitions between states, k the discrete time and F is in general a nonlinear function. Suppose that for each objective l one associates the performance index

$$J_l[s(i)] = \sum_{k=i}^{\infty} \gamma^{k-i} U_l[s(k), a(k)], \tag{9}$$

where U_l is the utility function for objective l and γ is a discount factor with $0 < \gamma < 1$. According to Bellman [17], the optimal cost-to-go function for objective l at time k is

$$J_l^*[s(k)] = \min_{a(k)} \{U_l[s(k), a(k)] + \gamma J_l^*[s(k+1)]\} \tag{10}$$

and the corresponding action $a^*(k)$ that achieves this optimal cost at time k is

$$a^*(k) = \arg\min_{a(k)} \{U_l[s(k), a(k)] + \gamma J_l^*[s(k+1)]\} \tag{11}$$

Obtaining $a^*(k)$ is a hard task since it depends upon solutions to the Hamilton-Jacobi-Bellman equation which is generally a nonlinear partial differential (or difference) equation [18]. Moreover, solutions to Hamilton-Jacobi-Bellman equation are computationally intractable, due to the curse of dimensionality [7]. The framework of ADP addresses this problem by using a system called “critic” to approximate J^* and to adapt a control law such that the utility U is maximized in the long run. The overall learning system we envisage to produce is depicted in Figure 1.

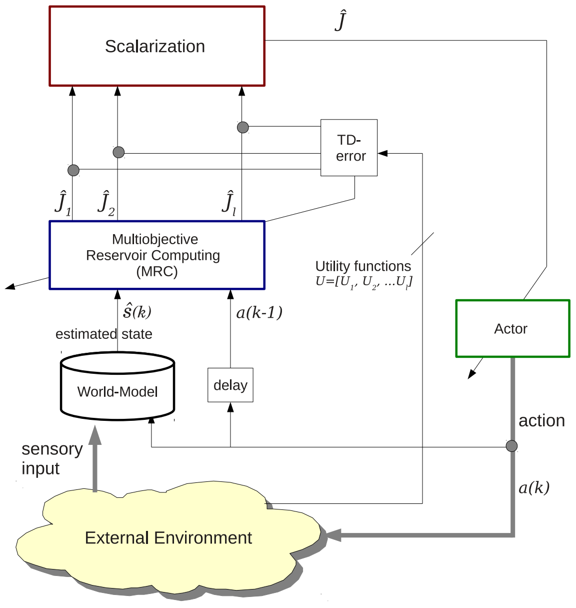


Fig. 1. Multi-objective optimization with RC-ADP

3.1 World Model

We consider a continuous state space $s(k)$ that reflects the position of the agent in its environment, and a continuous action space in a form of the agent's heading $a(k)$. In order to simulate a real environment, the state $s(k)$ is estimated by $\hat{s} \in R^3$ using three stationary landmarks. Each element of $\hat{s}$ is calculated as

$$\hat{s}_i = \frac{1}{80}\,||lm_i - pos|| = \frac{1}{80}d_i \tag{12}$$

where $\frac{1}{80}$ is a scaling factor relying on the size of the environment, and d_i is the distance between the agent and landmark i. We also add random noise with a signal-to-noise-ration of 2% to the measurments.

3.2 The Multiobjective Reservoir Computing

The activation of internal neurons is updated according to

$$X(k+1) = f(W_{in}U_{in}(k+1) + WX(k)) \tag{13}$$

where $U_{in} = s(k)$, X is the reservoir state, and $f = tanh()$. According to (10) the output weights W_{out} are adjusted recursively using the temporal difference (TD) learning algorithm [19], by minimizing the following error

$$\|E\| = \sum_k E_k = \sum_k [RC_l(k) - U_l(k) - \gamma RC_l(k+1)]^2 \tag{14}$$

After each adjustment of W_{out} (using the recursive algorithm of section 2), RC_l (the estimation of J_l is calculated as

$$RC_l(k) = f_{out}(W_l^{out} X(k))) \tag{15}$$

where f_{out}: Identity.

3.3 Scalarization

In this first version of multi-objective RC-ADP we combine all estimated $\hat{J}_l$ into one cost function $\hat{J}$.

$$\hat{J} = \sum_{i=1}^{l} w_i \hat{J}_i \tag{16}$$

where w_i are scalar weights from the intervall $[0, 1]$. The control policy is then used to optimize the combined cost function $\hat{J}$.

3.4 The actor

In the actual version of RC-ADP uses the control policy that minimizes the gradient of the utility

$$a^*(k+1) = a^*(k) \pm \delta \frac{\partial \hat{J}(k)}{\partial a^*(k)} \tag{17}$$

where δ is the learning rate, and the sign $\pm$ is related to the task at hand. The gradient of $\hat{J}(k)$ with respect to $a^*(k)$ can be computed using the chaine rule

$$\frac{\partial \hat{J}(k)}{\partial a^*(k)} = \frac{\partial \hat{J}(k)}{\partial s(k)} \frac{\partial s(k)}{\partial a^*(k)} \tag{18}$$

The term $\dfrac{\partial s(k)}{\partial a^*(k)}$ represents the model of the coupled system (8). The term $\dfrac{\partial \hat{J}(k)}{\partial s(k)}$ is calculated as follows

$$\frac{\partial \hat{J}(k)}{\partial s(k)} = \frac{\partial \hat{J}(k)}{\partial X(k)} \frac{\partial X(k)}{\partial s(k)} \tag{19}$$

From (15) we have

$$\frac{\partial \hat{J}(k)}{\partial X(k)} = W_{out}(k) \tag{20}$$

To compute the term $\dfrac{\partial X(k)}{\partial s(k)}$ we proceed as follows. We put

$$\Gamma = W_{in} s(k) + W X(k-1) \tag{21}$$

and using the chaine rule again for each element of $X = [x_1, x_2, \ldots x_N]$ we obtain

$$\frac{\partial x_i(k)}{\partial s(k)} = \frac{\partial f(\Gamma_i)}{\partial \Gamma_i}\frac{\partial \Gamma_i}{\partial S(k)} = (1 - x_i^2(k))w_i^{in} \tag{22}$$

$i = 1, \ldots N$ (the number of reservoir neurons), and $f = tanh()$ (the reservoir neurons output function).

From (19), (20) and (22) we obtain

$$\frac{\partial \hat{J}(k)}{\partial s(k)} = W_{out}(k)\{(I - X^2(k))W_{in}(k)\} \tag{23}$$

where I denotes the column vector of 1.

Equation (23) shows that the partial derivative of $\hat{J}$ with respect to s depends only on the W_{out} update and on the current reservoir state. Thus, in contrast to a typical layered neural networks, RC structure offers a simple way to calculate the $\hat{J}$ gradients. Thus, the power of RC-ACD lies in the fast training of RC and in the simplicity to propagate the required gradients to the actor.

4 Results

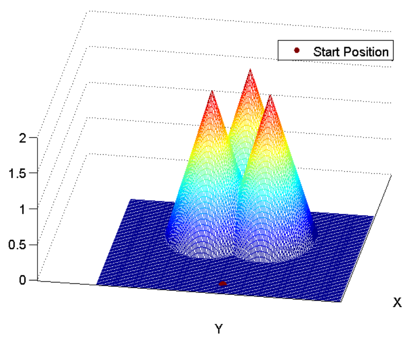
Fig. 2 illustrates a scenario of an agent that explores an environment containing three utility regions. Each utility is defined as following

$$U = \begin{cases} 2 * (1 - \frac{dist}{th}) & \text{, if } dist \leq th \\ 0 & \text{, if } dist > th \end{cases} \tag{24}$$

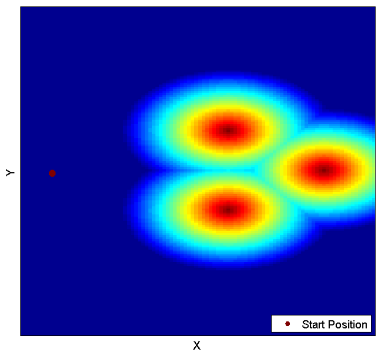
where $dist$ is the distance of the maximum utility to the agent, and th is a threshold from which the utility gradient is provided to the agent. Thus the utilities are not distributed over all the environment, i.e. they are not automatically provided with respect to each state/action.

In the simulations we put $th = 30$, and we set RC parameters as $\alpha = 0.65$, $N = 500$, the connectivity of the reservoir $c_{dr} = 30\%$ and the connectivity of the input weight matrice $c_i = 20\%$. We start the experiment with an untrained RC. The agent begins each run (episode) at the start position, and an episode ends if the agent drives beyond the borders of the environment or if it reaches the highest value of the utility, which is defined as 20% of the maximum value that could be reached. This was repeated for a variety of weights (w_1, w_2, w_3), and the resulting sets of solutions are shown in Fig. (3). The projection of the Pareto front onto the three objective planes are illustrated in Fig. 4. In Fig. 4 (b), the expected value for $\hat{J}_3$ linearly decreases once the value of $\hat{J}_1$ starts to increase. This means that the agent is not able to optimise both utility functions simultaneously. One reason is that the intersecting area of both utility functions is not large enough. However, the agent was able to optimise $\hat{J}_2$ with either $\hat{J}_3$ or $\hat{J}_1$ as shown in Figs. 4 (a) and (c).

The trajectories of the agent for two solutions from the obtained Pareto set are illustrated in Fig. 5. For the case $\hat{J} = 0.86\hat{J}_1 + 0.73\hat{J}_2 + 0.81\hat{J}_3$, the optimum is found approximately at the center of the three maxima. This was expected, since the weights are almost similar. In the next solution for the cost $\hat{J} = 0.04\hat{J}_1 + 0.17\hat{J}_2 + 0.87\hat{J}_3$, the agent ignores $\hat{J}_1$ and $\hat{J}_2$, and optimise $\hat{J}_3$.

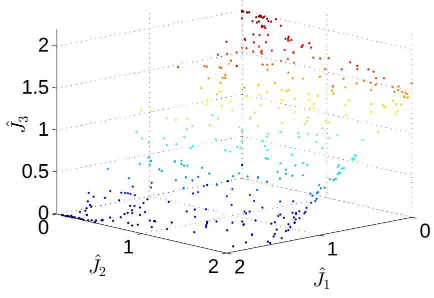


(a) 3D View

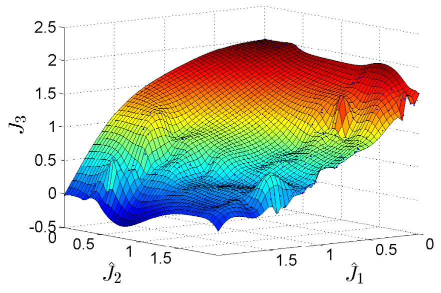


(b) Aerial View

Fig. 2. Utility Functions

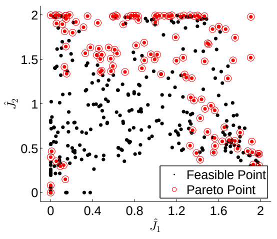


(a) Several feasible solutions

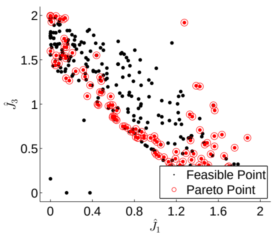


(b) Pareto Set

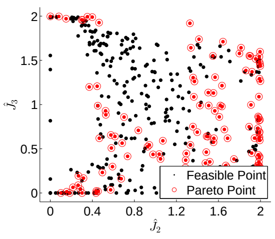
Fig. 3. The global Pareto-frontier solutions obtained across all 120 runs using multiobjective RC-ADP.



(a) Pareto set of $\hat{J}_2$ vs. $\hat{J}_1$

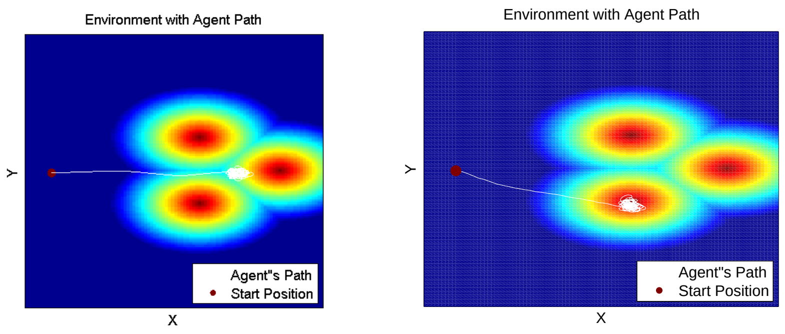


(b) Pareto set of $\hat{J}_3$ vs. $\hat{J}_1$



(c) Pareto set of $\hat{J}_3$ vs. $\hat{J}_2$

Fig. 4. Projection of the Pareto front onto the three objective planes. All black points are dominated by at least one red point



(a) Control policy for $\hat{J} = 0.86\hat{J}_1 + 0.73\hat{J}_2 + 0.81\hat{J}_3$

(b) Control policy for $\hat{J} = 0.04\hat{J}_1 + 0.17\hat{J}_2 + 0.87\hat{J}_3$

Fig. 5. Testing results of two solutions from the pareto set.

5 Conclusion

We have presented some preliminary results of the extended RC-ADP for solving multiobjective problems. A single reservoir was able to estimate several utilities simultaneously, and an actor adapts the policy in order to maximize the combined agent's utility. Our next step is to increase the complexity of the environment by adding some dynamics to the external world. Also, we are working on integrating a higher-level decision maker as a meta-learner to learn how to select the most performant policies between the pre-selected ones in the pareto set. It should consider a minimum necessary performance with regard to one or more utilities in selecting optimal policies.

References

1. P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Mach. Learn.*, 84(1–2):51–80, 2011.
2. Carlos A Coello Coello and Gary B Lamont. Applications of multi-objective evolutionary algorithms. *Advances in natural computation*, 1, 2004.
3. A. Castelletti, G. Corani, A. Rizzolli, R. Soncinie-Sessa, and E. Weber. Reinforcement learning in the operational management of a water system. In *IFAC Workshop on Modeling and Control in Environmental Issues*, pages 325–330, 2002.
4. Shie Mannor and Nahum Shimkin. A geometric approach to multi-criterion reinforcement learning. *Journal of Machine Learning Research*, 5:325–360, 2004.
5. C.R. Shelton. Importance sampling for reinforcement learning with multiple objectives. Technical Report No. 2001-003, Massachusetts Institute of Technology AI Lab, 2001.
6. L. Barrett and S. Narayanan. Learning all optimal policies with multiple criteria. In *Proc. of the 25th international conference on Machine learning*, ICML '08, pages 41–47, 2008.
7. S. Dreyfus and A. Law. *Art and Theory of Dynamic Programming*. Academic Press, Inc., Orlando, FL, USA, 1977.
8. F. Wang, H. Zhang, and D. Liu. Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, 4(2):39–47, 2009.

9. Jos Luis Bermudez. *Decision Theory and Rationality*. Oxford University Press, 2009.
10. Warren B. Powell. Perspectives of approximate dynamic programming. *Annals of Operations Research*, pages 1–38, 2012.
11. Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Network*, 6(6):801–806, 1993.
12. M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, August 2009.
13. M. Oubbati, J. Uhlemann, and G. Palm. Adaptive learning in continuous environment using actor-critic design and echo-state networks. In *From Animals to Animats 12. Lecture Notes in Computer Science.*, volume 7426, pages 320–329, 2012.
14. P. Koprinkova-Hristova, M. Oubbati, and G. Palm. Heuristic dynamic programming using echo state network as online trainable adaptive critic. *International Journal of Adaptive Control and Signal Processing*, 26(11), 2012.
15. H. Jaeger. The echo state approach to analysing and training recurrent neural networks. Technical Report 148, AIS Fraunhofer, St. Augustin, Germany, 2001.
16. W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput.*, 14(11):2531–2560, 2002.
17. R.E. Bellman. *Dynamic Programming*. NJ: Princeton Univ. Press, 1957.
18. D. Vrabie and F. Lewis. Neural network approach to continuous-time direct adaptive optimal control for partially unknown nonlinear systems. *Neural Networks*, 22(3):237–246, 2009.
19. Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.