Neuro-Remodeling via Backpropagation of Utility

K. Wendy Tang and Girish Pingle [1]
Department of Electrical Engineering
SUNY at Stony Brook, Stony Brook, NY 11794-2350.

ABSTRACT *Backpropagation of utility is one of the many methods for neuro-control. Its goal is to provide a series of control signals to maximize a utility function over time. In this paper, we demonstrate how to use the basic backpropagation and backpropagation through time algorithms as fundamental building blocks for backpropagation of utility. We also explore the parallel implementation of the algorithm on Intel's Paragon computer.*

The backpropagation of utility algorithm is composed of three subnetworks, the Action network, Model network, and an Utility network or function. Each of these networks includes a feedforward and a feedback component. A flow chart for the interaction of these components is included. To further illustrate the algorithm, we use backpropagation of utility for the control of a simple one-dimensional planar robot. We found that the success of the algorithm hinges upon a sufficient emulation of the dynamic system by the Model network. Furthermore, the execution time can be improved through pattern-partitioning on multiple processors.

1 Introduction

Neurocontrol is defined as the use of neural networks to emit control signals for dynamic systems. Most neurocontrol algorithms involve the incorporation of one or more backpropagation neural networks into the controller. Different approaches exist in the method of incorporating the neural network into the controller and of training and adaptation [1]. Among these approaches, there are five basic schemes: the *supervised control*, *direct inverse control*, *neural adaptive control*, *back-propagation of utility*, and *adaptive critic networks*. Werbos [2] provided a detailed summary of the five schemes including the pros and cons of each method. In this paper, our objective is to illustrate the theory of Backpropagation of Utility through a simple example, the control of a 1-D planar robot. To improve execution time, we also explore parallel implementation on an Intel Paragon parallel computer.

This article is organized as follows: Section 2 is a description of the *Backpropagation of Utility* algorithm. Section 3 illustrates how the algorithm is used for the control of a 1-D planar robot. Parallel implementation of the algorithms on an Intel Paragon parallel computer is explored in Section 4. Finally, conclusions and a summary are included in Section 5.

[1] The authors acknowledge and appreciate discussions with and contributions from Paul Werbos. This research was supported by the National Science Foundation under Grant No. ECS-9407363.
The authors also like to thank the Center for Scientific Computing of the Applied Mathematics Department in SUNY Stony Brook for the use of the Intel Paragon parallel computer.

2 Backpropagation of Utility Algorithm

The objective of the backpropagation of utility algorithm is to provide a set of *action* or *control signals* to a dynamic system to maximize a utility function over time. The utility function can be total energy, cost-efficiency, smoothness of a trajectory, etc. For expository convenience, we assume the notation $X(t)$ for system state at time t, $u(t)$ for the control signal, and $U(t)$ for the utility function which is usually a function of the system state.

The system is composed of three subsystems, an *Action* network, a *Model* network, and a *Utility* network, which is often represented as a performance function. The Action network is responsible for providing the control signal to maximize the utility function. This goal is achieved through adaptation of the internal weights of the Action network. Such adaptation is accomplished through backpropagation of various signals. For each iteration, there are feedforward and feedback components. In the feedforward mode, the Action network outputs a series of control signals, $u(t), t = 1, \ldots, T$ whereas adaptation of the internal weights is accomplished through the feedback mode.

The Model network provides an exact emulation of the dynamic system in a neural network format. Its function is two folded: (i) in the feedforward mode, it predicts the system state $X(t+1)$ at time $t+1$ for a given system state $X(t)$ and control signal $u(t)$ at time t; and (ii) in the feedback mode, it inputs the derivative of the utility function $U(t)$ with respect to the system state $X(t)$ and outputs the derivative of the utility with respect to the control signal, i.e., $\frac{\partial U(t)}{\partial u(t)}$ which is used for the adaptation of the Action network. The Utility network or function, on the other hand, provides a measure of the system performance $U(t)$ as a function of the system state, $X(t)$. In the feedforward mode, it calculates a performance value $U(t)$ and in the feedback mode, it identifies $\frac{\partial U(t)}{\partial X(t)}$ which is used by the Model network.

The basic idea is that assuming we have an exact model of the system formulated as a neural network (the Model network), we can use the backpropagation method to calculate the derivative of the utility function with respect to the control signal from the action network, i.e., $F_u(t) = \frac{\partial U(t)}{\partial u(t)}$. Such derivative is then used to calculate the gradient of the Utility with respect to the internal weights of the action network. Figure 1 shows a block-diagram representation of the system. The dashed lines represent the feedback mode, or derivative calculations.

The successful application of backpropagation of utility hinges upon an accurate Model network that represents the system. The establishment of such a Model network is accomplished through train-

ing with the basic backpropagation or Werbos' backpropagation through time algorithms [3]. Once an accurate Model network is obtained, the internal weights of the Action network is adapted to output a series of desired control action, according to the flow chart in Figure 3. In this flow-chart, **Action, Model, Utility** represent the feedforward components of the corresponding networks whereas **F_Utility, F_Model, F_Action** are the feedback components. Pseudo-computer codes for each of these blocks are included in [3, 4] and are not repeated here. The details of the construction of the Model network and the adaptation of the Action networks are described as follows.

2.1 Training of the Model Network

The establishment of a Model network that represents the system is accomplished through training with either the basic backpropagation or Werbos' backpropagation through time algorithm [3]. To speed up convergence, Jacob's delta-bar-delta rule [5] can be used for weight adaptation.

First, a sufficient number of training samples, T_M must be obtained. These training samples consists of m_M inputs ($X_{Mi}(t), \quad i = 1, \ldots, m_M, \quad t = 1, \ldots, T_M$), and n_M desired outputs ($Y_{Mi}(t), \quad i = 1, \ldots, n_M, \quad t = 1, \ldots, T_M$). The objective of a trained Model network is to emulate the dynamic system. In the feedforward mode, it outputs the system state $X(t+1)$ at time $t+1$ for a given system state $X(t)$ and control signal $u(t)$. That is , the inputs of the Model network, $X_M(t)$, consist of the system state and the control signal ($X(t)$ and $u(t)$). The outputs of the Model network is composed of the system state $X(t+1)$ at time $t+1$. A pseudo-code for training the Model network with backpropagation through time algorithm can be found in [4] and is not repeated here.

2.2 Adaptation of the Action Network

Upon completion of training of the Model network, we are ready for the adaptation of the Action network. In this stage, we adapt the weights of the Action network to output a series of desired control action $u_i(t), \ i = 1, \ldots, n$ for time period $t = 1, \ldots, T$. The desired system state is $X_{di}(t), \ i = 1, \ldots, m$. This adaptation process is accomplished through a number of iterations and is best described through the flow-chart shown in Figure 3.

There are basically six fundamental building blocks, **Action, Model**, and **Utility** in the feedforward mode; and **F_Utility, F_Model**, and **F_Action** in the feedback model. For each iteration, in the feedforward mode, a series of predicted control signals $u(t)$ for $t = 1, \ldots, T$ are provided by the **Action**

routine. These control signals are inputs to the **Model** routine which outputs the next system state $X(t+1)$ which is then used to calculate the Utility function.

In the feedback mode, the training samples are traversed backward. Since the Utility function is normally an explicit function of the system state, we can usually obtain $F_X(t) = \frac{\partial U(t)}{\partial X(t)}$ analytically. The value $F_X(t)$ is then input to the routine **F_Model** which corresponds to the feedback component of the Model network. **F_Action** is the next routine which takes the output $F_u(t) = \frac{\partial U(t)}{\partial u(t)}$ from the **F_Model** routine to calculate the gradient of the Utility function with respect to the weight-space, i.e., $F_W_{ij} = \frac{\partial U(t)}{\partial W_{ij}}$ for all weights W_{ij} of the Action network. Once the effect of all training samples are accounted for in F_W, delta-bar-delta rule is used to update the weights W, and the next iteration can be repeated. For simplicity, in Figure 3 we use a predefined value Max to determine the number of iterations. Other termination criteria such as a predefined utility value can also be used to determine the number of iterations. Again, pseudo-codes for these building blocks were included in [3, 4] and are not repeated here.

3 An Example: 1-D Robot Control

As an example, we consider a simple planar manipulator with one rotational joint (Figure 2). We assume, without loss of generality, that the robot links can be represented as point-masses concentrated at the end of the link. The link mass and length are respectively: $M = 0.1$ kg, $L = 1$ m. This simple dynamic system is governed by the equation:

$$\tau(t) = M\ L^2\ddot{\theta}(t) + M\ g\ L\ \cos(\theta(t)) \tag{1}$$

where $g = 9.81\text{m/s}^2$ is the gravitational constant. We consider that initially, at time $t = 0$ second, the state of the manipulator is $\theta_0 = \dot{\theta}_0 = \ddot{\theta}_0 = 0$, with $\tau_0 = 0.981$ Newtons. The neural network's task is to generate a series of control signals $u(t) = \tau\ (t),\quad t = \delta t,\ 2\delta t,\ \ldots,\ t_f = T \times \delta t = 2$ seconds $(\delta t = 0.02, T = 100)$ to drive the manipulator from the initial configuration θ_0 to $\theta_f = \theta(t = t_f) = 60°$ with the following desired trajectory specified by the quintic polynomial [6].

$$\begin{aligned}
\theta_d(t) &= \theta_0 + 10(\theta_f - \theta_0)(t/t_f)^3 - 15(\theta_f - \theta_0)(t/t_f)^4 + 6(\theta_f - \theta_0)(t/t_f)^5 \\
\dot{\theta}_d(t) &= 30(\theta_f - \theta_0)(t^2/t_f^3) - 60(\theta_f - \theta_0)(t^3/t_f^4) + 30(\theta_f - \theta_0)(t^4/t_f^5) \\
\ddot{\theta}_d(t) &= 60(\theta_f - \theta_0)(t/t_f^3) - 180(\theta_f - \theta_0)(t^2/t_f^4) + 120(\theta_f - \theta_0)(t^3/t_f^5)
\end{aligned} \tag{2}$$

The system consists of an *Action* network, a *Model* network, and an utility function. Like in supervised control, in backpropagation of utility, our goal is to train the Action network to provide a set of

control signal $u(t) = \tau(t)$; but unlike supervised training, the desired control signals $\tau_d(t)$ are *not* used as feedback. Instead, the training of the Action network is accomplished through feedback from the Model network and the Utility function. The Model network basically acts as a system emulator whereas the Utility function provides a performance measure. In the following subsections, we first describe how the Model network is trained, and later how to use the trained Model network for the adaptation of the Action network which provides a series of control signals for the specific task described here.

3.1 Training of the Model Network

Before the adaptation of the Action network begins, the Backpropagation of Utility algorithm involves first training of the Model network. Again, the Model network accepts as inputs the system state (i.e., $\theta(t)$, $\dot{\theta}(t)$, $\ddot{\theta}(t)$) and the control signal $\tau(t)$ at the current time. Its function is to provide the actual system state for the next time period ($\theta(t+1)$, $\dot{\theta}(t+1)$, $\ddot{\theta}(t+1)$). From our experimentation, we found that it is more efficient if the Model network is trained to generate the change in system state instead of the actual value. Therefore, we train the model network to generate $\delta\theta(t)$, $\delta\dot{\theta}(t)$, $\delta\ddot{\theta}(t)$. The system state of the next time period can then be computed:

$$\theta(t+1) = \theta(t) + \delta\theta(t), \quad \dot{\theta}(t+1) = \dot{\theta}(t) + \delta\dot{\theta}(t), \quad \ddot{\theta}(t+1) = \ddot{\theta}(t) + \delta\ddot{\theta}(t).$$

To obtain an adequate representation of the system, we need to train the Model network with sufficient number of training points. In this case, we use the basic backpropagation algorithm with delta-bar-delta rule. The network has two hidden-layers with ten nodes in each layer. As stated in [7], progressive training in which the number of training samples increases gradually helps to maintain stability and provide fast convergence. Therefore, we start the training on $T_M = 20$ samples and gradually increase to $T_M = 500$ training samples. Each training set consists of four inputs ($m_M = 4$): $\theta(t)$, $\dot{\theta}(t)$, $\ddot{\theta}(t)$, $\tau(t)$ and three desired outputs ($n_M = 3$): $\delta\theta_d(t)$, $\delta\dot{\theta}_d(t)$, $\delta\ddot{\theta}_d(t)$.

Each of these training samples is obtained by first generating a random system state $\theta(t)$, $\dot{\theta}(t)$ $\ddot{\theta}(t)$ with the following constraints:

$$\begin{aligned} \theta(t) &\in \{0,\ 2\pi\} \quad \text{radians}, \\ \dot{\theta}(t) &\in \{-3,\ 3\} \quad \text{radians/second}, \\ \ddot{\theta}(t) &\in \{-5,\ 5\} \quad \text{radians/second}^2. \end{aligned}$$

For the given system state, we compute or measure the corresponding torque value, $\tau(t - \delta t)$ and then generate a random $\delta\tau(t)$ with the constraint that

$$\delta\tau(t) \in \{-0.02,\ 0.02\} \text{ Newtons}$$

An Euler integrator [6] is then used to solve for the actual system state $\theta(t+1)$, $\dot{\theta}(t+1)$, $\ddot{\theta}(t+1)$ for the given $\tau(t) = \tau(t-\delta t) + \delta\tau(t)$ and $\theta(t), \dot{\theta}(t), \ddot{\theta}(t)$. The desired outputs of the training set are computed as:

$$\delta\theta_d(t) = \theta(t+1) - \theta(t), \quad \delta\dot{\theta}_d(t) = \dot{\theta}(t+1) - \dot{\theta}(t), \quad \delta\ddot{\theta}_d(t) = \ddot{\theta}(t+1) - \ddot{\theta}(t)$$

3.2 Adaptation of the Action Network

With the Model network successfully trained, we are ready for the adaptation of the Action network. As illustrated in Figure 3, the adaptation of the Action network involves both a feedforward and a feedback component. In the feedforward mode, the Action network accepts the desired system state, namely, $\theta_d(t), \dot{\theta}_d(t), \ddot{\theta}_d(t)$ as inputs. The output of the Action network is to provide the signal $\tau(t)$ to drive the manipulator. For efficient training, we choose to train the action network to generate $\delta\tau(t)$. The value $\tau(t)$ can then be computed by:

$$\tau(t) = \tau(t-1) + \delta\tau(t). \tag{3}$$

where $t = \delta t,\ 2\delta t,\ \ldots,\ T \times \delta t$ and in this example, $\tau(t=0) = \tau_0 = 0.981$ Newton.

The computed torque $\tau(t)$ (Equation 3) is then passed to the trained Model network which accepts the desired system state, $\theta_d(t), \dot{\theta}_d(t), \ddot{\theta}_d(t)$ along with $\tau(t)$ as inputs. As described in the previous section, the output of the Model network indicates the change of the system state from its input state, i.e, $\delta\theta(t)$, $\delta\dot{\theta}(t)$, $\delta\ddot{\theta}(t)$. The actual system state for the next sample can then be computed according to:

$$\theta(t+1) = \theta_d(t) + \delta\theta(t), \quad \dot{\theta}(t+1) = \dot{\theta}_d(t) + \delta\dot{\theta}(t), \quad \ddot{\theta}(t+1) = \ddot{\theta}_d(t) + \delta\ddot{\theta}(t).$$

The last step in the feedforward mode is to compute the "utility" or performance of the action network. Since our objective here is tracking control, we use the utility function

$$U(t) = \frac{1}{2}\sum_{t=1}^{T} (\theta(\mathrm{t}) - \theta_d(\mathrm{t}))^2 + (\dot{\theta}(\mathrm{t}) - \dot{\theta}_d(\mathrm{t}))^2 + (\ddot{\theta}(\mathrm{t}) - \ddot{\theta}_d(\mathrm{t}))^2. \tag{4}$$

After a series of $\tau(t)$ and the corresponding $U(t)$ are produced, in the feedback mode, the gradient of the Utility with respect to system state is:

$$\frac{\partial U(t)}{\partial \mathbf{X}(t)} = [\theta(t) - \theta_d(t)] + [\dot{\theta}(t) - \dot{\theta}_d(t)] + [\ddot{\theta}(t) - \ddot{\theta}_d(t)].$$

This result is used by the Model network (F_Model routine) to determine $F_u(t) = \frac{\partial U(t)}{\partial u(t)}$ which is then used to determine $\frac{\partial U(t)}{\partial W_{ij}}$, the gradient of the utility with respect to the weight space. Basically, the idea is to change the output of the action network in the direction of $F_u(t)$ by adjusting its weights.

In our implementation, we found that the adaptation process is more robust if the weights of the Action network are adjusted through multiple iterations for the same $F_u(t)$ computed by the F_Model routine. This is due to the fact that steepest descent, in general, takes multiple iterations to achieve a particular desired output. Therefore, in this example, we have modified the feedback mode of the flowchart in Figure 3 to include an inner loop of iterations to adjust the internal weights of the Action network for a given $F_u(t)$ from the F_Model routine. Figure 4 shows the modified feedback component. The choice of the value for the number of inner iteration, Max_In depends on the problem. In this example, we have use both $Max_In = 1,000$ and $10,000$.

Figure 5 plots the generated torque

$$\tau(t) = \tau(t-1) + \delta\tau(t)$$

versus time where $\delta\tau(t)$ is generated by the Action network. Note that at iteration one (Iter=1), the generated torque is far from the desired value. But through multiple iterations, the generated torque gradually converges to the desired value. The iteration number shown here corresponds to the number of outer iterations. Figure 6 plots the error of the generated torque with the desired value $|\tau_d(t) - \tau(t)|$ after approximately 5,000 iterations. From this graph, the maximum error is bounded by 0.02 Newtons.

Again, unlike basic supervised control, the Backpropagation of Utility algorithm does not require the desired value $\tau_d(t)$ be available to the Action network. They are used here only to illustrate the performance of the action network. These figures show clearly that the weights of the action network is adapting to generate a forecast of the desired control signals based solely on the feedback signals $F_u(t)$ from the **F_Model** routine.

4 Parallel Implementation

To improve the execution time, we explore parallel implementation of these algorithms. Both the basic and backpropagation through time algorithm (the building blocks of backpropagation of utility) can be parallelized by two techniques - *node partitioning* and *pattern partitioning* [8]. Node-partitioning implies that the entire network is partitioned among different processors, each computing for the whole set of training samples. Pattern-partitioning, on the other hand, partitions the training patterns among the processors with each one representing the entire network. Our preliminary investigation found that node-partition helps to reduce execution time only for large networks. In our example, both the Action and Model network only have 10 hidden nodes in each layer. For such small networks, the

communication overhead involved in node-partitioning actually slows down the overall execution time. We, therefore, consider only the pattern-partitioning scheme.

In our implementation of the pattern partitioning scheme, training samples are equally divided among the number of processors. That is, for T training sets, and N_p number of processors, each processor computes both the feedforward and the feedback components of the $T_p = \frac{T}{N_p}$ training samples. (We assume that N_p divides T).

At the end of the backward loop, the weight changes computed based on the subset of the training samples of each processor are broadcasted. Once this information is received by all processors, the total weight changes F_W_{ij} are computed at every processor:

$$F_W_{ij} = \sum_{k=1}^{N_p} F_W_{ij}(k)$$

where $F_W_{ij}(k)$ is the weight gradient of processor k computed based on its own subset of training samples. Upon obtaining the total weight gradient, delta-bar-delta rule is applied at all processors to update the weights which completes one iteration. A flow chart for the basic backpropagation with pattern partitioning in included in Figure 8. The case of backpropagation through time can be obtained in a similar manner [4]. Figure 7 plots the execution time per outer iteration versus different number of processors. The amount of inner iterations is $Max_In = 1,000$. (See Figures 3 and 4 for the definition of outer and inner iterations).

We observe that, initially, the execution time decreases with increasing number of processors, but when the number of processors is greater than four, the execution time starts to increase. We attribute this phenomenon to the amount of communication among the processors when the samples are partitioned into too many processors. In particular, the local weight gradient ($F_W(k)$ for processor k, $k = 1, \ldots, N_p$ processors) needs to be broadcasted to all before each Delta-Bar-Delta routine can be called (see flowcharts in Figures 3 and 4). Because of our modified feedback mode, for each outer iteration, there are $Max_In = 1,000$ number of inner iterations. Each of this inner iteration requires each processor to broadcast its local weight gradient to all. In other words, eventually, the communication overhead associated with multiple processors will outweigh the advantages of parallel execution and the execution time per iteration starts to increase.

To provide an approximate comparison of the performance of the pattern-partitioning scheme, we implemented the algorithm on different computer platforms (Intel Paragon; Sun Sprac II, and Sun Sparc LX) for the 1-D example in Section 3. For the single-processor machines (Sun Sparc II and LX), we

executed the compiler-optimized program only when a single-user is logged on. The execution time for one outer iteration with $1,000$ inner iterations are 10.2 sec, 43.7 sec, and 84.8 sec for the 4-node Paragon, Sun Sparc II, and Sun Sparc Lx, respectively. The 4-node Paragon implementation indeed gives the best performance. Furthermore, we believe as the size of the problem grows larger, say bigger network, more training samples, the advantages of parallel execution will be more pronounced and the difference between multiple- and single-processor implementation will increase.

5 Conclusions

Backpropagation of Utility is one of the methods for neuro-control. Its goal is to provide a series of control signals to maximize a utility function. Basically, the system is composed of three subnetworks, the *Action* network, *Model* network and the *Utility* network which sometimes can be represented as a simple *Utility* function. Each of these networks has the feedforward components **Action**, **Model** and **Utility** and the feedback components **F_Action**, **F_Model** and **F_Utility**, respectively. The algorithm involves first training of the Model network to emulate the dynamic system and later adaptation of the internal weights of the Action network to generate a series of control signals. Such adaptation involves interactions of the three networks and are best described in the flow chart of Figure 3. To further illustrate the algorithm, we use the algorithm to control a $1-D$ planar robot. We showed that the Action network is capable of generating a series of control signals that maximize the utility function.

In short, backpropagation of utility is a simple neuro-control technique that uses a neural network (the Model network) to emulate the dynamic system and to provide proper feedback to adjust the weights of the Action network. It differs from supervised control in that the desired control signals are *not* needed in the feedback mode. However, the main drawback of the algorithm is its slow execution time. To alleviate this problem, we investigated parallel implementation of the algorithm on multiple processors of Intel's Paragon parallel computer. In conclusion, we believe that backpropagation of utility with parallel implementation is a powerful tool for neurocontrol or neuromodeling.

References

[1] D. Psaltis, A. Sideris, and A. Yamamura. "A Multilayered Neural Network Controller". *IEEE Control Systems Magazine*, pages 17–21, April 1988.

[2] Paul J. Werbos. Neurocontrol and Supervised Learning: an Overview and Evaluation. In D.A. White and D.A. Sofge, editors, *Handbook of Intelligent Control*, pages 65–89. Van Nostrand Reinhold, 1992.

[3] Paul J. Werbos. "Backpropagation Through Time: What It Does and How to Do It". *Proceedings of the IEEE*, 78(10):1550–1560, October 1990.

[4] K. W. Tang and Girish Pingle. Exploring Neuro-Control with Backpropagation of Utility. In *Ohio Aerospace Institute Neural Network Symposium and Workshop*, pages 107–137, Athens, Ohio, August 21-22 1995.

[5] Robert A. Jacobs. "Increased Rates of Convergence Through Learning Rate Adaptation". *Neural Networks*, 1:295–307, 1988.

[6] J. J. Craig. *Introduction to Robotics, mechanics and Control*. Addison-Wesley Publishing Co., New York, NY, 1986.

[7] K. W. Tang and H-J Chen. A Comparative Study of Basic Backpropagation and Backpropagation Through Time Algorithms. Technical Report TR-700, State University of NY at Stony Brook, College of Engineering and Applied Sciences, November 1994.

[8] V. Kumar, S. Shekhar, and M.B. Amin. "A Scalable Parallel Formulation of the Backpropagation Algorithm for Hypercubes and Related Architecture". *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1073–1089, October 1994.

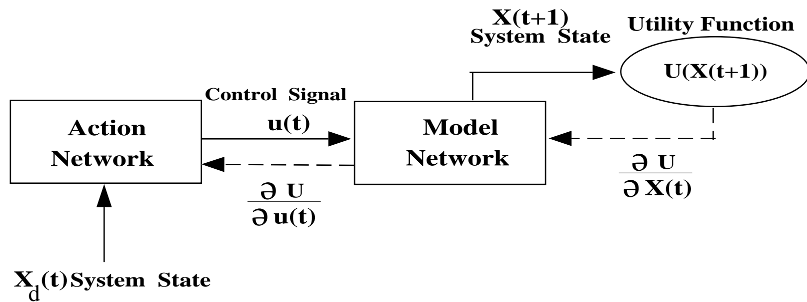

Figure 1: A Backpropagation of Utility System

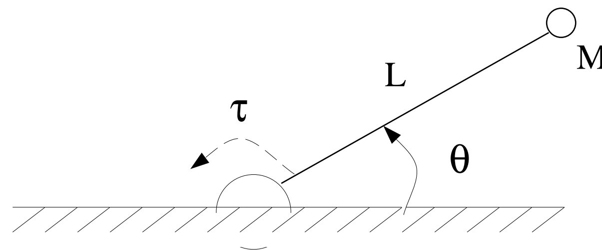

Figure 2: A 1-D Planar Robot

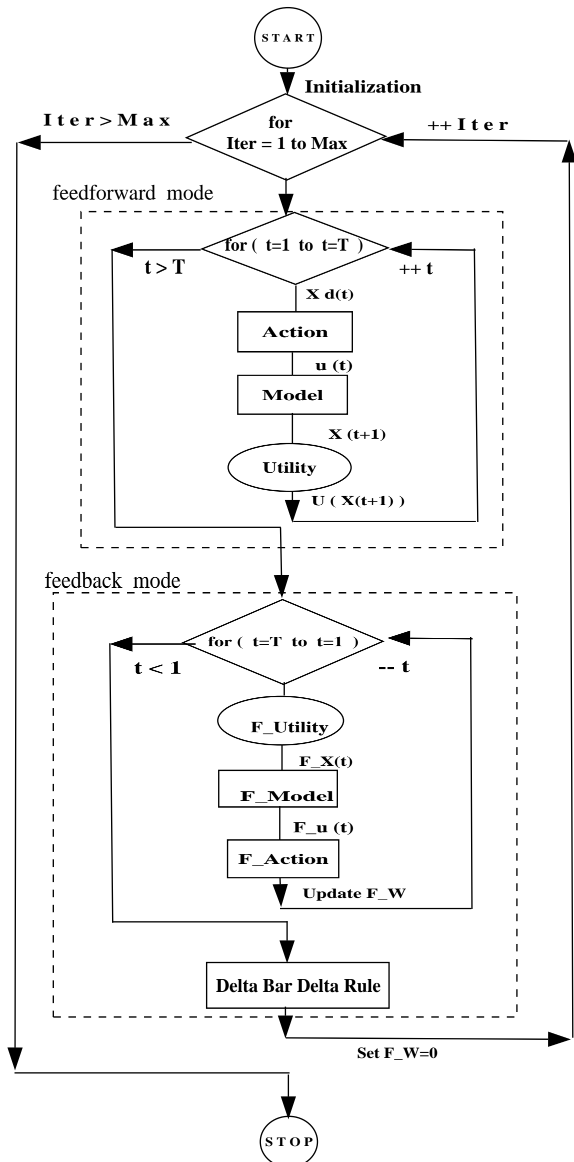

Figure 3: Adaptation of Action Network

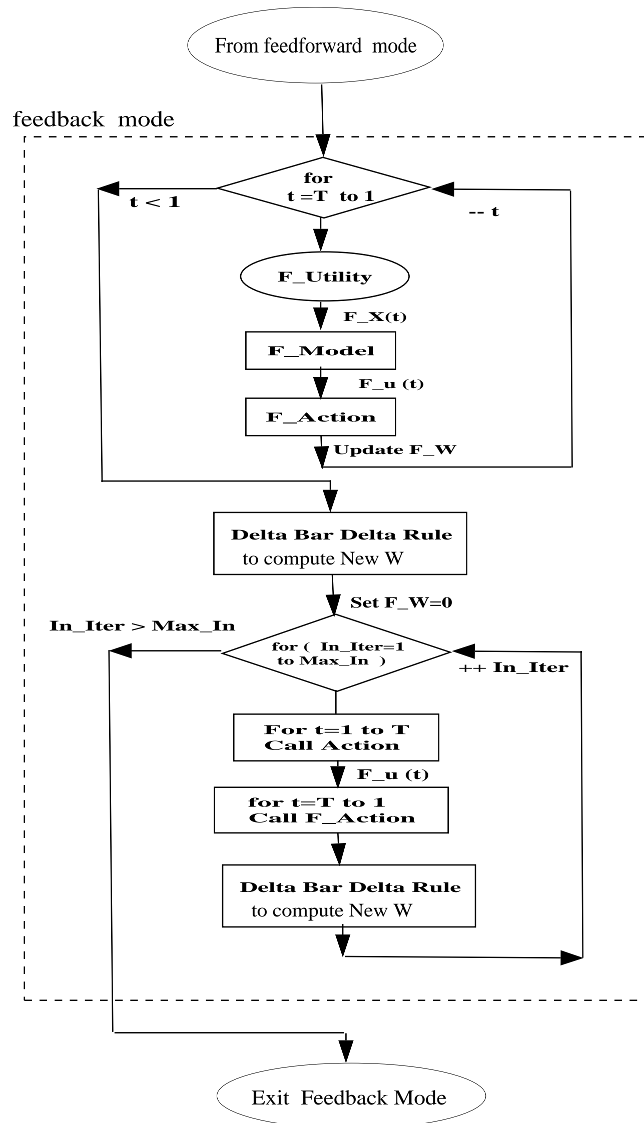

Figure 4: Modified Feedback Mode

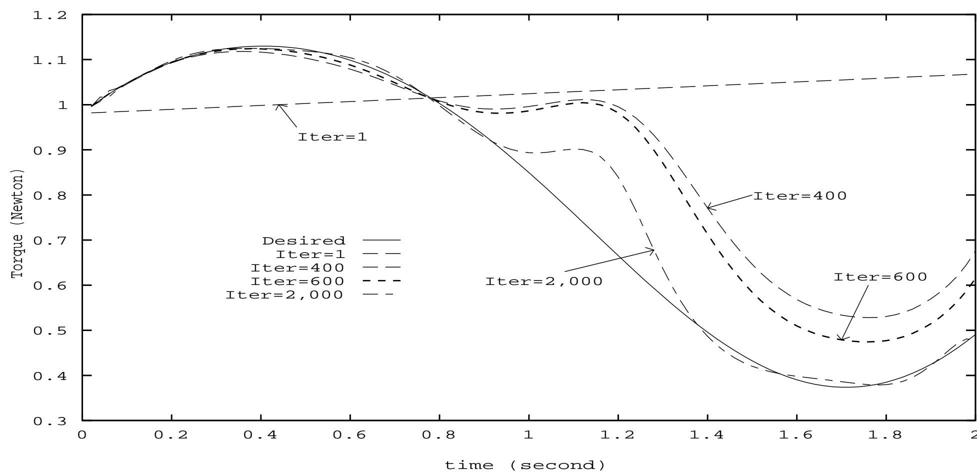

Figure 5: Generated Torque at Different Iterations

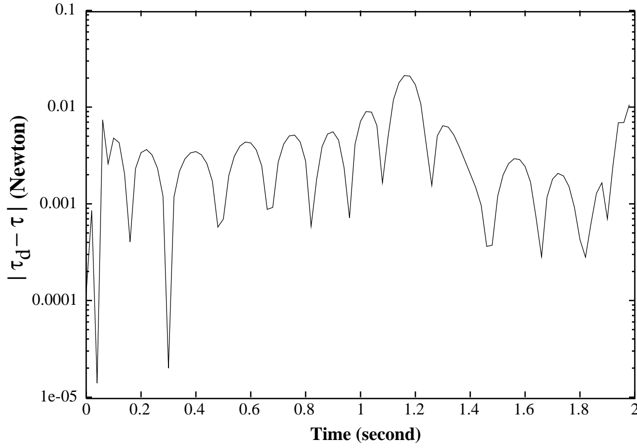

Figure 6: Errors in Control Signal

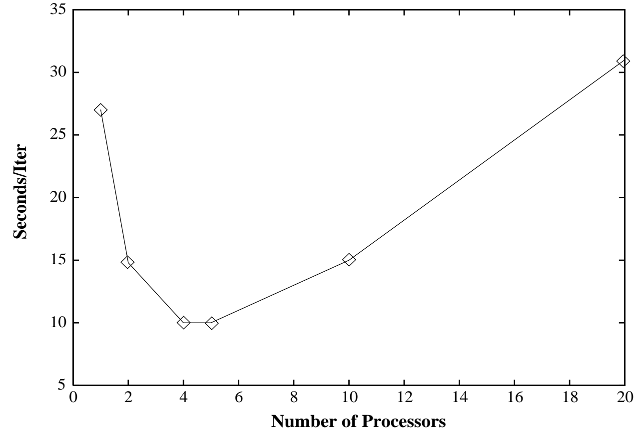

Figure 7: Execution Time for Backpropagation of Utility

START

Initialization

Divide the T training samples among the Np processors, each having Tp = T/Np training samples

Concurrent Computation on processor k=1 to Np

for Iter = 1 to Max

Iter > Max

++ Iter

for t=1 to Tp

t > Tp

++ t

Feedforward Computation

for t=Tp to 1

t < 1

-- t

Feedback Computation Update F_W(k) at processor k

Broadcast F_W (k) to all other processors

Reset F_W (k)=0

Receive F_W (q) , $q \neq k$ from all other processors

$$F_W = \sum_{k=1}^{Np} F_W\{k\}$$

Delta Bar Delta Rule

STOP

Figure 8: Pattern Partitioning of Basic Backpropagation