Department of Electrical Engineering
College of Engineering and Applied Sciences
State University of New York at Stony Brook
Stony Brook, New York 11794-2350

Exploring Neuro-Control with Backpropagation of Utility

by

K. Wendy Tang and Girish Pingle

Technical Report # 713

June, 1995

Exploring Neuro-Control with 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. Pseudo-computer codes for each component and a flow chart for the interaction of these components are 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 of the algorithm can be improved through pattern-partitioning on multiple processors.

Keywords: Neuro-Control, Backpropagation of Utility, and Pattern-Partioning.

1 Introduction

Neurocontrol is defined as the use of neural networks to emit control signals for dynamic systems. Neural networks offer several advantages over conventional computing architectures [1]. Calculations are carried out in parallel yielding speed advantages and programming is done by training through examples. These networks are characterized by their *learning* and *generalization* capabilities [2]. The neural network "learns" the system model by training through a set of training patterns. Their inherently parallel architecture and trainability make neural networks attractive candidates for fast, real-time control with unknown dynamic models.

[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. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The authors also like to thank Professors James Glimm and Yue-Feng Deng in the Center for Scientific Computing of the Applied Mathematics Department in SUNY Stony Brook for the use of the Intel Paragon parallel computer.

The most dominant form of neural networks used is the multi-layer backpropagation network. It is a hierarchical design consisting of fully interconnected layers of neurons [3]. The weights associated with each neuron are updated by taking the gradient of the total squared error with respect to the weights and performing a gradient search of the weight space [4]. Errors are propagated backwards through the network, hence the name *back-propagation.* Despite its popularity, the main drawback of the basic backpropagation algorithm is its slow convergence rate. Various efforts are made to increase the rate of convergence, e.g., the Delta-Bar-Delta Rule [5]. By incorporating memory from previous time periods into current outputs, Werbos developed a more sophisticated version of the basic backpropagation algorithm, known as *backpropagation through time* [6]. In our previous work, we found that by combining the backpropagation through time algorithm with the Delta-Bar-Delta rule, the neural network provides more robust and faster learning [7].

Almost all neural network applications in robot control involve the incorporation of one or more backpropagation (basic or through time) neural networks into the controller. Different approaches exist in the method of incorporating the neural network into the controller and of training and adaptation [8]. 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 [9] 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 decrease the execution time, we also explore parallel implementation of the various algorithms on multiple nodes of an Intel 110-node Paragon parallel computer.

This article is organized as follows: Section 2 is a review of all the basic equations for the backpropagation algorithms (both the *basic* and the *through time* versions), including the delta-bar-delta rule for speeding up convergence. A simple network with six neurons is used to illustrate the validity of the equations. Then in Section 3, we show how these equations are used as fundamental building blocks for the *Backpropagation of Utility* algorithm. Section 4 illustrates how the algorithm is used for the control of a 1-D planar robot. Parallel imple-

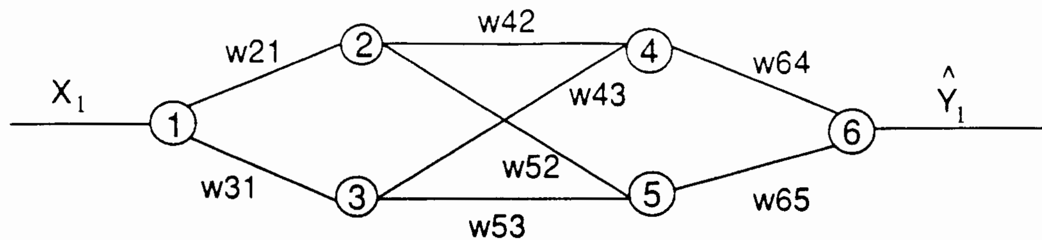


Figure 1: *A Six-Node Two-Layer Network.*

mentation of the algorithms on multiple processors of an Intel Paragon parallel computer is explored in Section 5. Finally, conclusions and a summary are included in Section 6.

2 Basic Equations

In this section, we present the basic equations for backpropagation and backpropagation through time algorithms. A simple example is used to illustrate the validity of these equations. For expository convenience, we assume there are m inputs, n outputs, H hidden nodes, and T training samples. The training inputs are presented to the network as $X_i(t), \quad i = 1, \dots, m, \quad t = 1, \dots, T$ and the corresponding desired outputs are $Y_i(t), \quad i = 1, \dots, n, \quad t = 1, \dots, T$. The detailed of the algorithm can be found in [6]. For the reader's convenience, they are also summarized here.

2.1 Basic Backpropagation Algorithm

The backpropagation algorithm is simply a tool for calculating the derivative of a function. The network equations consist of the feedforward and feedback components. During the feedforward mode, the network calculates an estimated output $\hat{Y}$ as a function of the inputs and the weights associated with the neurons. An error function is then produced by comparing $\hat{Y}$ with the desired output Y. In the feedback mode, the gradients of this error with respect to the weight space are identified. Subsequently, the weights are updated through the steepest descent method. More specifically, for training samples, $t = 1, \dots, T$,

the feedforward equations are:

$$x_i(t) = X_i(t) \qquad 1 \le i \le m \tag{1}$$

for $i = m+1$ to $i = m+H+n$,

$$\begin{cases} net_i(t) &= \sum_{j=1}^{i-1} W_{ij} x_j(t) \\ x_i(t) &= s(net_i(t)) \end{cases} \tag{2}$$

$$\hat{Y}_i(t) = x_{m+H+i}(t) \qquad 1 \le i \le n \tag{3}$$

The error of the network is obtained by comparing the actual and the desired outputs.

$$E = \sum_{t=1}^{T} E(t) = \sum_{t=1}^{T} \sum_{i=1}^{n} 0.5[\hat{Y}_i(t) - Y_i(t)]^2 \tag{4}$$

where $\hat{Y}_i(t)$ is the output of the neural network and $Y_i(t)$ is the desired outputs. This error is fed back to the network. The error gradient F_W_{ij} with respect to each weight, W_{ij} is calculated with the feedback equations:

For training samples, $t = 1, \ldots, T$, the feedback equations are:

$$F_\hat{Y}_i(t) = \frac{\partial E}{\partial \hat{Y}_i(t)} = \hat{Y}_i(t) - Y_i(t) \qquad i = 1, \ldots, n \tag{5}$$

for $i = m+H+n$ to $i = m+1$,

$$\begin{cases} F_x_i(t) &= F_\hat{Y}_{i-m-H}(t) + \sum_{j=i+1}^{m+H+n} W_{ji} * F_net_j(t) \\ F_net_i(t) &= s'(net_i) * F_x_i(t) \end{cases} \tag{6}$$

$$F_W_{ij} = \sum_{t=1}^{T} F_net_i(t) * x_j(t) \qquad i,j = 1, \ldots, m+H+n \tag{7}$$

where $s(z)$ is the sigmoidal transfer function and $s'(z)$ is the derivative of $s(z)$. Also,

$$s(z) = 1/(1+e^{-z}) \tag{8}$$

$$s'(z) = s(z) * (1 - s(z)) \tag{9}$$

Once F_W_{ij} (the gradient of E with respect to W_{ij}) is calculated, each weight is updated according to:

$$\text{New } W_{ij} = W_{ij} - \alpha * F_W_{ij} \quad i,j = 1, \ldots, m+H+n \tag{10}$$

where α is a constant called the *learning rate*.

2.2 An Example

In this section, we use a small network to illustrate the equations presented in Section 2.1 and to demonstrate that the backpropagation algorithm is simply a tool for derivative calculation. Consider the network with $m = 1$ input node, $n = 1$ output node, and two hidden layers each with two neurons. The network is layered and there is no connections for nodes of the same layer. The network with the weights associated at each node is shown in Figure 1.

According to Equations 2, the feedforward equations are:

$$\begin{array}{ll} x_1(t) = X_1(t) & \\ net_2(t) = W_{21} * x_1(t) & x_2(t) = s(net_2(t)) \\ net_3(t) = W_{31} * x_1(t) & x_3(t) = s(net_3(t)) \\ net_4(t) = W_{42} * x_2(t) + W_{43} * x_3(t) & x_4(t) = s(net_4(t)) \\ net_5(t) = w_{52} * x_2(t) + W_{53} * x_3(t) & x_5(t) = s(net_5(t)) \\ net_6(t) = W_{64} * x_4(t) + W_{65} * x_5(t) & x_6(t) = s(net_6(t)) \\ \hat{Y}_1(t) = x_6(t) & \end{array}$$

The idea of backpropagation is to adjust the weights $W_{21}, W_{31}, \ldots, W_{64}, W_{65}$ such that $\hat{Y}(t)$ is close to the desired $Y(t)$. To achieve this goal, we need to find the gradient of the error function with respect to the weights. From Equation 4, the error function is:

$$E = \sum_{t=1}^{T} [\hat{Y}_1(t) - Y_1(t)]^2$$

Using the Equations 6, for $t = 1, \ldots, T$, the backward equations are:

$$\begin{aligned} F_\hat{Y}_1(t) &= \hat{Y}_1(t) - Y_1(t) \\ F_x_6(t) &= F_\hat{Y}_1(t) \\ F_net_6(t) &= s'(net_6) * F_x_6(t) \end{aligned}$$

$$\begin{aligned} F_x_5(t) &= W_{65} * F_net_6(t) \\ F_net_5(t) &= s'(net_5) * F_x_5(t) \end{aligned}$$

$$\begin{aligned} F_x_4(t) &= W_{64} * F_net_6(t) \\ F_net_4(t) &= s'(net_4) * F_x_4(t) \end{aligned}$$

$$\begin{aligned} F_x_3(t) &= W_{43} * F_net_4(t) + W_{53} * F_net_5(t) \\ F_net_3(t) &= s'(net_3) * F_x_3(t) \end{aligned}$$

$$\begin{aligned} F_x_2(t) &= W_{42} * F_net_4(t) + w52 * F_net_5(t) \\ F_net_2(t) &= s'(net_2) * F_x_2(t) \end{aligned}$$

$$\begin{aligned} F_x_1(t) &= W_{21} * F_net_2(t) + W_{31} * F_net_3(t) \\ F_net_1(t) &= s'(net_1) * F_x_1(t) \end{aligned}$$

Once the F_net_i are known, the error gradient with respect to the different weights are:

$$F_W_{21} = \sum_{t=1}^{T} F_net_2(t) * x_1(t)$$

$$F_W_{31} = \sum_{t=1}^{T} F_net_3(t) * x_1(t)$$

$$F_W_{42} = \sum_{t=1}^{T} F_net_4(t) * x_2(t)$$

$$\vdots$$

$$F_W_{64} = \sum_{t=1}^{T} F_net_6(t) * x_4(t)$$

$$F_W_{65} = \sum_{t=1}^{T} F_net_6(t) * x_5(t)$$

The weights are then adjusted according to

$$\text{New } W_{ij} = W_{ij} - \alpha * F_W_{ij}, \quad i,j = 1,2,\ldots,6$$

To verify that F_W_{ij} indeed corresponds to $\frac{\partial E}{\partial W_{ij}}$, we calculate the gradient directly from the forward equations. As an example, consider $\frac{\partial E}{\partial W_{21}}$,

$$\frac{\partial E}{\partial W_{21}} = \sum_{t=1}^{T} \frac{\partial E}{\partial \hat{Y}_1(t)} \frac{\partial \hat{Y}_1(t)}{\partial W_{21}} \tag{11}$$

where

$$\begin{aligned}
\frac{\partial E}{\partial \hat{Y}_1(t)} &= [\hat{Y}_i(t) - Y_i(t)] = F_Y_1(t) \\
\frac{\partial \hat{Y}_1(t)}{\partial W_{21}} &= \underbrace{\frac{\partial \hat{Y}_i(t)}{\partial net_6(t)}}_{s'(net_6(t))} \frac{\partial net_6(t)}{\partial W_{21}} \\
\frac{\partial net_6(t)}{\partial W_{21}} &= [\underbrace{\frac{\partial net_6(t)}{\partial x_4}}_{W_{64}} \frac{\partial x_4(t)}{\partial W_{21}} + \underbrace{\frac{\partial net_6(t)}{\partial x_5}}_{W_{65}} \frac{\partial x_5(t)}{\partial W_{21}}] \\
\frac{\partial x_4(t)}{\partial W_{21}} &= \underbrace{\frac{\partial x_4(t)}{\partial net_4(t)}}_{s'(net_4(t))} \frac{\partial net_4(t)}{\partial W_{21}} \qquad \frac{\partial x_5(t)}{\partial W_{21}} = \underbrace{\frac{\partial x_5(t)}{\partial net_5(t)}}_{s'(net_5(t))} \frac{\partial net_5(t)}{\partial W_{21}} \\
\frac{\partial net_4(t)}{\partial W_{21}} &= [\underbrace{\frac{\partial net_4(t)}{\partial x_2}}_{W_{42}} \frac{\partial x_2(t)}{\partial W_{21}} + \underbrace{\frac{\partial net_4(t)}{\partial x_3}}_{W_{43}} \frac{\partial x_3(t)}{\partial W_{21}}] \\
\frac{\partial x_2(t)}{\partial W_{21}} &= \underbrace{\frac{\partial x_2(t)}{\partial net_2(t)}}_{s'(net_2(t))} \underbrace{\frac{\partial net_2(t)}{\partial W_{21}}}_{x_1(t)} \qquad \frac{\partial x_3(t)}{\partial W_{21}} = \underbrace{\frac{\partial x_3(t)}{\partial net_3(t)}}_{s'(net_3(t))} \underbrace{\frac{\partial net_3(t)}{\partial W_{21}}}_{x_1(t)}
\end{aligned} \tag{12}$$

Hence substituting Equations 12 to Equation 11, we have

$$\begin{aligned}
\frac{\partial E}{\partial W_{21}} &= \sum_{t=1}^{T} \underbrace{F_Y_1(t)\ s'(net_6(t))}_{F_net_6(t)}\ [\ W_{64}\ s'(net_4(t))\ W_{42}\ s'(net_2(t))\ x_1(t) \\
&\qquad + W_{65}\ s'(net_5(t))\ W_{52}\ s'(net_2(t))\ x_1(t)\ ] \\
&= \sum_{t=1}^{T} [\ \underbrace{\overbrace{F_net_6(t)\ W_{64}}^{F_x_4}\ s'(net_4(t))}_{F_net_4(t)} W_{42} + \underbrace{\overbrace{F_net_6(t)\ W_{65}}^{F_x_5}\ s'(net_5(t))}_{F_net_5(t)} W_{52}\ ]\ s'(net_2(t))\ x_1(t) \\
&= \sum_{t=1}^{T} \underbrace{[\ F_net_4(t)\ W_{42}\ +\ F_net_5(t)\ W_{52}\ ]}_{F_x_2(t)}\ s'(net_2(t))\ x_1(t) \\
&= \sum_{t=1}^{T} \underbrace{F_x_2(t)\ s'(net_2(t))}_{F_net_2(t)}\ x_1(t) \\
&= \sum_{t=1}^{T} F_net_2(t)\ x_1(t) = F_W_{21}
\end{aligned}$$

The rest of F_W_{ij} can be proved similarly. From these equations, we can see that the backpropagation algorithm is simply a tool for calculating the gradient of the error function with respect to the weight space.

2.3 Backpropagation Through Time Algorithm

Backpropagation through time was first proposed by Werbos [6]. It is basically an extension of the basic backpropagation algorithm but also considers memory from previous time periods. Mathematically, this is implemented through the introduction of a second set of weights W'. In our version of the backpropagation through time algorithm, we associated a weight W' at each hidden and output node. A more general version that includes a W' for each connection can be found in [6]. In our version, the second equation of the feedforward equations (Eq 2) is replaced by:

$$net_i(t) = \sum_{j=1}^{i-1} W_{ij} x_j(t) + W_i' x_i(t-1), \quad m < i \leq m + H + n \tag{13}$$

And the second equation of the feedback equations (Equation 6) is replaced by:

$$\begin{aligned} F_x_i(t) = F_\hat{Y}_{i-m-H}(t) + \textstyle\sum_{j=i+1}^{m+H+n} W_{ji} * F_net_j(t) \quad &+ W_i' * F_net_i(t+1) \\ &i = m + H + n, \ldots, m + 1 \end{aligned} \tag{14}$$

For adaptation of the W',

$$\begin{array}{ll} F_W'_i & = \sum_{t=1}^{T} F_net_i(t) * x_i(t) \\ \text{New } W'_i & = W'_i - \beta * F_W'_i \qquad i = m+1, \ldots, m+H+n \end{array} \tag{15}$$

where β is the constant learning rate for W'.

2.4 Delta-Bar-Delta Rule

To improve the convergence speed of the steepest descent/ascent method, Jacob proposed the delta-bar-delta algorithm [5]. Basically, the algorithm is a special case of the *Adaptive Learning Rate* (ALR) discussed in [9]. Every weight of the network is given its own learning rate and that the rates changes with time. According to [5], the learning rate update rule is:

$$\Delta\alpha_{ij}(t) = \begin{cases} \kappa & \text{if } \bar{\delta}_{ij}(t-1)\delta_{ij}(t) > 0 \\ -\phi\alpha_{ij}(t-1) & \text{if } \bar{\delta}_{ij}(t-1)\delta_{ij}(t) < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

where

$$\begin{array}{rcl} \delta_{ij}(t) & = & F_W_{ij} \\ \bar{\delta}_{ij}(t) & = & (1-\theta)\delta_{ij}(t) + \theta\bar{\delta}_{ij}(t-1) \\ \alpha_{ij}(t) & = & \alpha_{ij}(t-1) + \Delta\alpha_{ij}(t) \end{array}$$

In these equations, $\delta_{ij}(t)$ is the partial derivative of the error with respect to W_{ij} at time t and $\bar{\delta}_{ij}(t)$ is an exponential average of the current and past derivatives with θ as the base and time as the exponent[5]. If the current derivative of a weight and the exponential average of the weight's previous derivatives possess the same sign, the learning rate for that weight is incremented by a constant κ. If the current derivative of a weight and the exponential average of the weight's previous derivatives possess opposite signs, the learning rate for the weight is decremented by a proportion ϕ of its current value [5].

As discussed in [10], we found the best result comes from a combination of the back-propagation through time algorithm with the delta-bar-delta rule. In this case, $\beta_i(t)$, the learning rate for W'_i also changes with time. More specifically,

$$\Delta\beta_i(t) = \begin{cases} \kappa & \text{if } \bar{\gamma}_i(t-1)\gamma_i(t) > 0 \\ -\phi\beta_i(t-1) & \text{if } \bar{\gamma}_i(t-1)\gamma_i(t) < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

where

$$\begin{aligned}
\gamma_i(t) &= F_W_i' \\
\bar{\gamma}_i(t) &= (1-\theta)\gamma_i(t) + \theta\bar{\gamma}_i(t-1) \\
\beta_i(t) &= \beta_i(t-1) + \triangle\beta_i(t)
\end{aligned}$$

3 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 can often be 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 the delta-bar-delta rule with iterations. 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 $X(t+1)$ 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, 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.

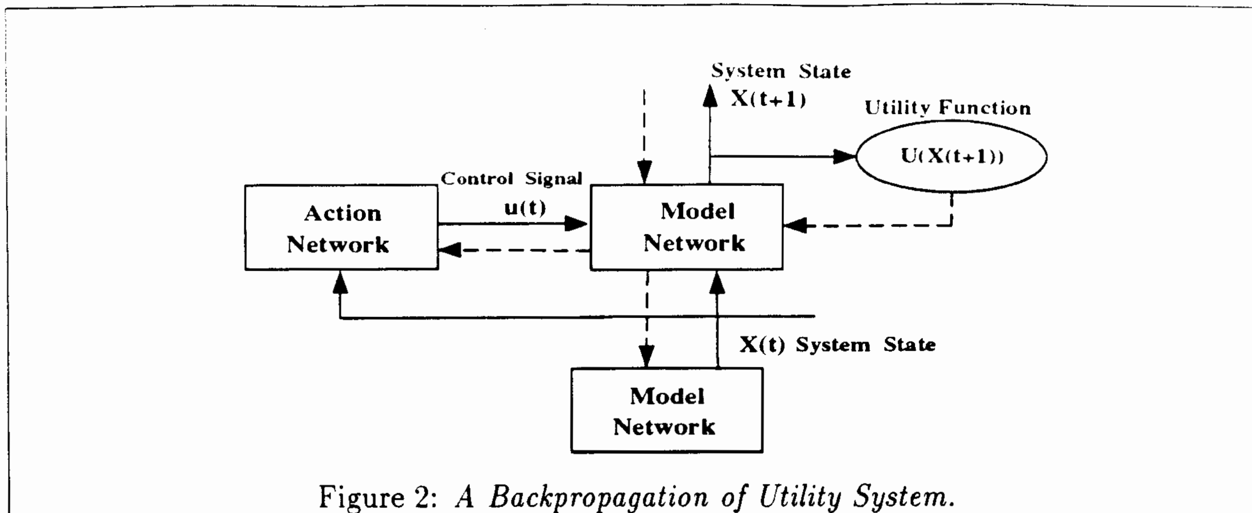


Figure 2: *A Backpropagation of Utility System.*

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 2 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 training with the basic backpropagation or backpropagation through time algorithms. 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. The details of the construction of the Model network and the adaptation of the Action networks are included in the following subsections.

3.1 Training of the Model Network

The establishment of a Model network that represents the system is accomplished through training with either the basic or the backpropagation through time algorithm combined with Jacob's delta-bar-delta rule discussed in Section 2.3.

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 $u(t)$ at time t. That is , $X_M(t)$ consists of $X(t)$ and $u(t)$ and $Y_M(t)$ is composed of the system state $X(t+1)$ at time $t+1$. For expository convenience, we assume there are H_M hidden nodes. A pseudo-code for training the Model network with backpropagation through time algorithm is presented in Table 1. The version for the basic algorithm can be obtained by setting $W' = 0$ for all nodes at all times.

3.2 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.

Assume $m_M, n_M, N_M, H_M, T_M, X_M, Y_M$ as defined in Section 2.
For $i, j = 1, \ldots, N_M + n_M$

W_{ij}, are randomly distributed between ± 1, $\delta_{ij} = \bar{\delta}_{ij} = 0$.

Also, $W_{ij} = 0$ for nodes i, j in the same layer and is fixed for all iterations.

Initialize, $W_i' = \gamma_i = \bar{\gamma}_i = 0$ for $i = m_M + 1, \ldots, N_M + n_M$.

for (Iteration =1 to Maximum Iteration)
{

Step 1: for $(t = 1$ to $t = T_M)$

$$\begin{cases} \text{for } (i = 1 \text{ to } i = m_M) \quad x_{Mi}(t) = X_{Mi}(t) \\ \text{for } (i = m_M \text{ to } i = N_M + n_M) \\ \quad \begin{cases} net_{Mi}(t) &= \sum_{j=1}^{i-1} W_{Mij} x_{Mj}(t) + W_{Mi}' x_{Mi}(t-1) \\ x_{Mi}(t) &= s(net_{Mi}(t)) \end{cases} \\ \text{for } (i = 1 \text{ to } i = n_M) \quad \hat{Y}_{Mi}(t) = x_{M(m+H+i)}(t) \end{cases}$$

Step 2: Compute the Error, E_M.

$$E_M = \sum_{t=1}^{T_M} E_M(t) = \sum_{t=1}^{T_M} \sum_{i=1}^{n_M} 0.5[\hat{Y}_{M\,i}(t) - Y_{M\,i}(t)]^2$$

Step 3: for $(t = T_M$ to $t = 1)$

$$\begin{cases} \text{for } (i = 1 \text{ to } i = n_M) \quad F_\hat{Y}_{M\,i}(t) = \hat{Y}_{M\,i}(t) - Y_{M\,i}(t) \\ \text{for } (i = N_M + n_M \text{ to } i = m_M + 1) \\ \quad \begin{cases} F_x_{M\,i}(t) = & F_\hat{Y}_{M\,(i-m-H)}(t) + \sum_{j=i+1}^{m+H+n} W_{M\,(ji)} * F_net_{M\,j}(t) \\ & + W_{M\,i}' * F_net_{M\,i}(t+1) \\ F_net_{M\,i}(t) = & s'(net_{M\,i}) * F_x_{M\,i}(t) \end{cases} \end{cases}$$

Step 4: For $i, j = 1, \ldots, N_M + n_M$, compute

$$(i)\; F_W_{M\,(ij)} = \delta_{ij} = \sum_{t=1}^{T} F_net_{M\,i}(t) * x_{M\,j}(t); \quad (ii)\; \Delta\alpha_{ij} = \begin{cases} \kappa & \text{if } \bar{\delta}_{ij}\delta_{ij} > 0 \\ -\phi\alpha_{ij} & \text{if } \bar{\delta}_{ij}\delta_{ij} < 0 \\ 0 & \text{otherwise.} \end{cases}$$

$$(iii)\; \bar{\delta}_{ij} = (1-\theta)\delta_{ij} + \theta\bar{\delta}_{ij} \quad \text{and} \quad (iv)\; \alpha_{ij} = \alpha_{ij} + \Delta\alpha_{ij}$$

Step 5: For $i = m + 1, \ldots, m + H + n$, compute

$$(i)\; F_W_{M\,i}' = \gamma_i = \sum_{t=1}^{T_M} F_net_{M\,i}(t) * x_{M\,i}(t); \quad (ii)\; \Delta\beta_i = \begin{cases} \kappa & \text{if } \bar{\gamma}_i\gamma_i > 0 \\ -\phi\beta_i & \text{if } \bar{\gamma}_i\gamma_i < 0 \\ 0 & \text{otherwise.} \end{cases}$$

$$(iii)\; \bar{\gamma}_i = (1-\theta)\gamma_i + \theta\bar{\gamma}_i \quad \text{and} \quad (iv)\; \beta_i = \beta_i + \Delta\beta_i$$

Step 6: New $W_{M\,(ij)} = W_{M\,(ij)} - \alpha_{ij} * F_W_{M\,(ij)}$ $\quad i, j = 1, \ldots, m + H + n$

New $W_{M\,i}' = W_{M\,i}' - \beta_i * F_W_{M\,i}'$ $\quad i = m + 1, \ldots, N_M + n_M$

}

Table 1: *A Pseudo-Code for Training of Model Network.*

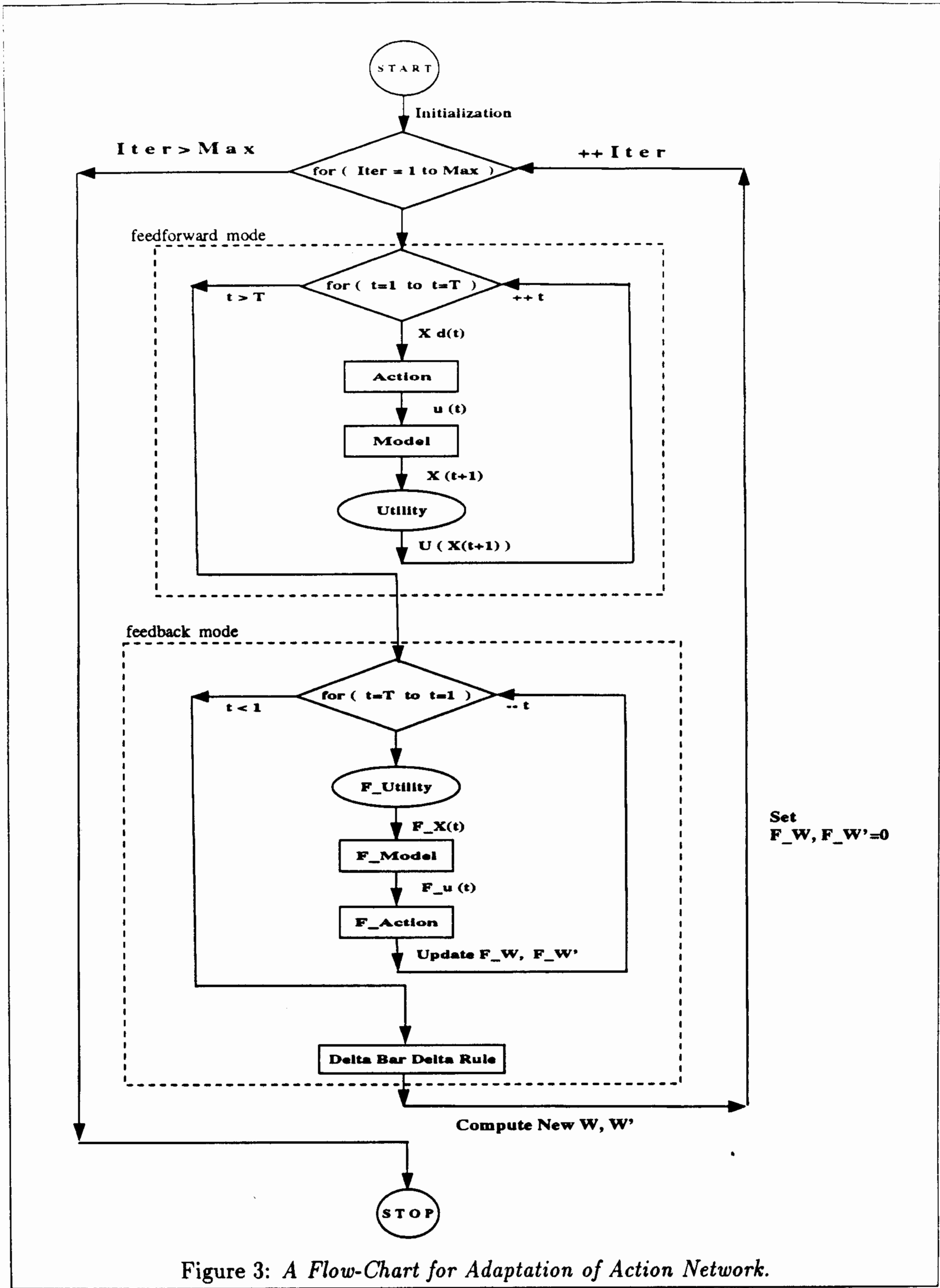


Figure 3: *A Flow-Chart for Adaptation of Action Network.*

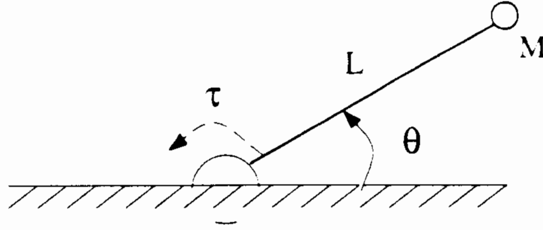


Figure 4: *A 1-D Planar Robot.*

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., $F_W_{ij} = \frac{\partial U(t)}{\partial W_{ij}}$ and $F_W'_i = \frac{\partial U(t)}{\partial W'_i}$ for all weights W_{ij} and W'_i of the Action network. Once the effect of all training samples are accounted for in F_W and F_W', delta-bar-delta rule is used to update the weights, W and W', and the system is ready for the next iteration.

Note that, for simplicity, in Figure 3 we use a predefined value Max to determine the number of iterations. However, this is not always the best strategy, other termination criteria such as a predefined utility value can also be used to determine the number of iterations. Pseudo-codes for these building blocks are included in Tables 2 to 5.

4 An Example: 1-D Robot Control

As an example, we consider a simple planar manipulator with one rotational joint (Figure 4). 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:

$$\tau(t) = M\ L^2\ddot{\theta}(t) + M\ g\ L\ \cos(\theta(t)) \tag{18}$$

where $g = 9.81\text{m/s}^2$ is the gravitational constant.

Assume m inputs, n outputs, H hidden nodes, T samples, and $N = m + H$.

Inputs:	$X_i(t)$	system state at time t	$i \in \{1, \ldots, m\}$
	W_{ij}	internal weights	$i, j \in \{1, \ldots, N + n\}$
	W'_i	internal weights	$i, j \in \{m + 1, \ldots, N_M + n_M\}$
Outputs:	$x_i(t)$	internal state	$i \in \{1, \ldots, N + n\}$
	$\hat{Y}_i(t) = u_i(t)$	control signals	$i \in \{1, \ldots, n\}$

Action$(X(t), W, W', x(t), \hat{Y}(t))$

{

Step 1: for $(i = 1$ to $i = m)$

$x_i(t) = X_i(t)$

Step 2: for $(i = m$ to $i = N + n)$

$$\begin{cases} net_i(t) &= \sum_{j=1}^{i-1} W_{ij} x_j(t) + W'_i x_i(t-1) \\ x_i(t) &= s(net_i(t) \end{cases}$$

Step 3: for $(i = 1$ to $i = n)$

$\hat{Y}_i(t) = u_i(t) = x_{(m+H+i)}(t)$

}

Table 2: *A Pseudo-Code for Subroutine Action.*

Let $m, n, T, X(t), u(t)$ be defined in Table 2.

Assume $m_M = m + n$ inputs, $n_M = m$ outputs, H_M hidden nodes, and $N_M = H_M + n_M$.

Inputs:	$X_{Mi}(t)$	contains $X(t)$ and $u(t)$	$i \in \{1, \ldots, m_M\}$
	$W_{M\,(ij)}$	resultant weights after training	$i, j \in \{1, \ldots, N_M + n_M\}$
	$W'_{M\,i}$	resultant weights after training	$i, j \in \{m + 1, \ldots, N_M + n_M\}$
Outputs:	$x_{Mi}(t)$	internal state of Model network	$i \in \{1, \ldots, N_M + n_M\}$
	$\hat{Y}_{Mi}(t)$	outputs to Utility function	$i \in \{1, \ldots, n_M\}$

Model$(X_M(t), W_M, W'_M, x_M(t), \hat{Y}_M(t))$

{

Step 1: for $(i = 1$ to $i = m_M)$

$x_{Mi}(t) = X_{Mi}(t)$

Step 2: for $(i = m_M$ to $i = N_M + n_M)$

$$\begin{cases} net_{Mi}(t) &= \sum_{j=1}^{i-1} W_{Mij} x_{Mj}(t) + W'_{Mi} x_{Mi}(t-1) \\ x_{Mi}(t) &= s(net_{Mi}(t)) \end{cases}$$

Step 3: for $(i = 1$ to $i = n_M)$

$\hat{Y}_{Mi}(t) = x_{M(m+H+i)}(t)$

}

Table 3: *A Pseudo-Code for Subroutine Model.*

Let $m, n, T, X(t), u(t)$ be defined in Table 2, and $m_M = m + n$, $n_M = m$.
Assume H_M hidden nodes, and $N_M = H_M + n_M$.

Inputs:	$F_X_i(t)$	from F_Utility	$i \in \{1, \ldots, n_M\}$
	$x_{M\,i}(t)$	internal state of Model network	$i \in \{1, \ldots, N_M + n_M\}$
	$W_{M\,(ij)}$	weights of Model network	$i, j \in \{1, \ldots, N_M + n_M\}$
	$W'_{M\,i}$	weights of Model network	$i, j \in \{m+1, \ldots, N_M + n_M\}$

Outputs: $F_u_i(t) = \frac{\partial U(t)}{\partial u_i(t)} \quad i \in \{1, \ldots, n\}$

F_Model($F_X(t), W_M, W'_M, x_M(t), F_u_i(t)$)
{
Step 1: for ($i = 1$ to $i = N_M$) $\quad F_x_{M\,i}(t) = 0.0$
Step 2: for ($i = 1$ to $i = n_M$) $\quad F_x_{M\,(N_M+i)}(t) = F_X_i(t)$
Step 3: for ($i = N_M + n_M$ to $i = 1$)

$$\begin{cases} F_x_{M\,i}(t) = F_x_{M\,i}(t) + \sum_{j=i+1}^{N_M+n_M} W_{M\,(ji)} * F_net_{M\,j}(t) \\ \qquad + W'_{M\,i} * F_net_{M\,i}(t+1) \\ F_net_{M\,i}(t) = s'(net_{M\,i}) * F_x_{M\,i}(t) \end{cases}$$

Step 4: for ($i = 1$ to $i = n$) $\quad F_u_i(t) = F_x_{M\,m+i}$
}

Table 4: *A Pseudo-Code for Subroutine F_Model.*

Let $m, n, T, X(t), u(t)$ be defined in Table 2.
Assume H_M hidden nodes, and $N_M = H_M + n_M$.

Inputs:	$F_u_i(t)$	from F_Utility	$i \in \{1, \ldots, n\}$
	$x_i(t)$	internal state of Action network	$i \in \{1, \ldots, N + n\}$
	$W_{(ij)}$	internal weights	$i, j \in \{1, \ldots, N + n\}$
	W'_i	internal weights	$i, j \in \{m+1, \ldots, N + n\}$

Outputs: $F_W_{ij}(t) = \frac{\partial U(t)}{\partial W_{ij}} \quad i, j \in \{1, \ldots, N + n\}$
$F_W'_i(t) = \frac{\partial U(t)}{\partial W'_i} \quad i \in \{1, \ldots, N + n\}$

F_Action($F_X(t), W_M, W'_M, x_M(t), F_u_i(t)$)
{
Step 1: for ($i = 1$ to $i = N$) $\quad F_x_i(t) = 0.0$
Step 2: for ($i = 1$ to $i = n$) $\quad F_x_{(N+i)}(t) = F_u_i(t)$
Step 3: for ($i = N + n$ to $i = 1$)

$$\begin{cases} F_x_i(t) = F_x_i(t) + \sum_{j=i+1}^{N+n} W_{(ji)} * F_net_j(t) + W'_i * F_net_i(t+1) \\ F_net_i(t) = x_i(t) * (1 - x_i(t)) * F_x_i(t) \\ F_W_{ij} = F_W_{ij} + F_net_i(t) * x_j(t) \quad j = 1, \ldots, n \end{cases}$$

Step 4: for ($i = 1$ to $i = n$) $\quad F_u_i(t) = F_x_{M\,m+i}$
}

Table 5: *A Pseudo-Code for Subroutine F_Action.*

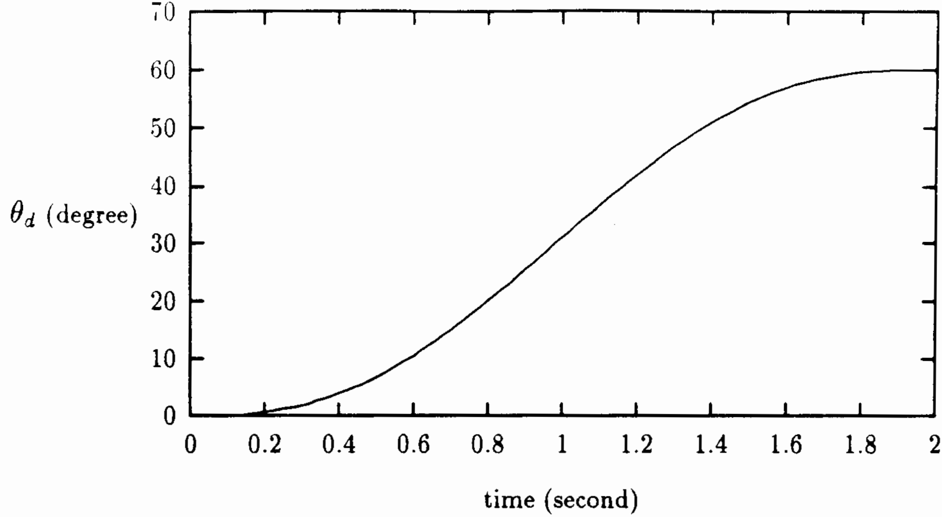


Figure 5: *Desired Trajectory of Manipulator.*

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 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 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 [11].

$$\begin{aligned}
\theta_d(t) &= \theta_0 + 10(\theta_f - \theta_0)(t/t_f)^3 - 15(\theta_f - \theta_0)(t/t_f)^4 + 6(\theta_f - \theta_0)(t/t_f)^5 \\
\dot{\theta}_d(t) &= 30(\theta_f - \theta_0)(t^2/t_f^3) - 60(\theta_f - \theta_0)(t^3/t_f^4) + 30(\theta_f - \theta_0)(t^4/t_f^5) \\
\ddot{\theta}_d(t) &= 60(\theta_f - \theta_0)(t/t_f^3) - 180(\theta_f - \theta_0)(t^2/t_f^4) + 120(\theta_f - \theta_0)(t^3/t_f^5) \qquad (19)
\end{aligned}$$

The desired angle profile of the system is shown in Figure 5. We assume the sampling period is $\delta t = 0.02$ second. That is, the number of samples is $T = t_f/\delta t = 100$.

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, which basically acts as a system emulator, and the Utility function which provides performance measurement. Using

the terminologies of Section 3, the system state corresponds to the position, velocity, and acceleration of the system, i.e., $\mathbf{X}(t) = (\theta(t), \dot{\theta}(t), \ddot{\theta}(t))$, the control signal is $u(t) = \tau(t)$.

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.

4.1 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 at the previous time instant (i.e., $\theta(t-\delta t)$, $\dot{\theta}(t-\delta t)$, $\ddot{\theta}(t-\delta t)$) and the control signal $\tau(t)$ at the current time. Its function is to provide the actual system state for the current time ($\theta(t)$, $\dot{\theta}(t)$, $\ddot{\theta}(t)$). 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 current time can then be computed:

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

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 (i.e, Table 1 with $W' = 0$ at all times). 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 ($m_M = 4$): $\theta(t-\delta t)$, $\dot{\theta}(t-\delta t)$, $\ddot{\theta}(t-\delta t)$, $\tau(t)$ and three desired outputs ($n_M = 3$): $\delta\theta_d(t)$, $\delta\dot{\theta}_d(t)$, $\delta\ddot{\theta}_d(t)$.

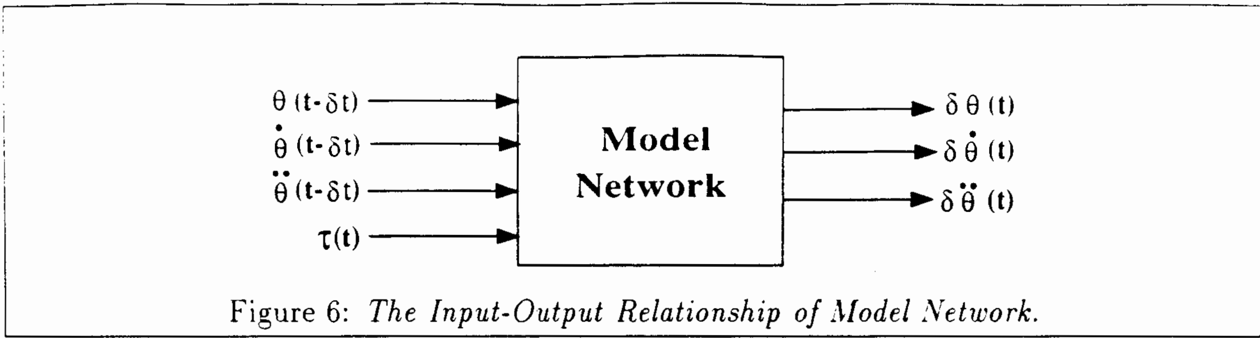


Figure 6: *The Input-Output Relationship of Model Network.*

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

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

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

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

Figure 6 summarizes the input-output relationship of the Model network.

4.2 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 $\tau(t)$

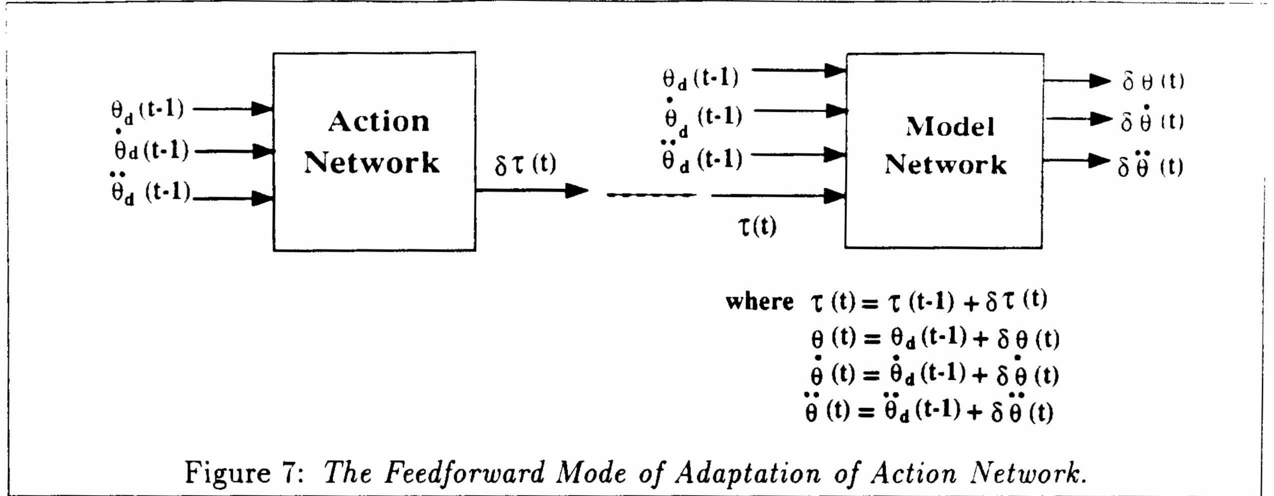


Figure 7: *The Feedforward Mode of Adaptation of Action Network.*

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:

$$\tau(t) = \tau(t - \delta t) + \delta\tau(t). \tag{20}$$

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 20) is then passed to the trained Model network which accepts the desired system state of the previous instant, $\theta_d(t-\delta t), \dot{\theta}_d(t-\delta t), \ddot{\theta}_d(t-\delta t)$ 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 can then be computed according to:

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

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

$$U(t) = \frac{1}{2}\sum_{t=1}^{T} (\theta(\mathrm{t} \times \delta\mathrm{t}) - \theta_d(\mathrm{t} \times \delta\mathrm{t}))^2 + (\dot{\theta}(\mathrm{t} \times \delta\mathrm{t}) - \dot{\theta}_d(\mathrm{t} \times \delta\mathrm{t}))^2 + (\ddot{\theta}(\mathrm{t} \times \delta\mathrm{t}) - \ddot{\theta}_d(\mathrm{t} \times \delta\mathrm{t}))^2. \tag{21}$$

The input-output relationship in this feedforward component of the adaptation of the Action network is summarized in Figure 7.

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 in turn is used to determine $\frac{\partial U(t)}{\partial W_{ij}}$, the gradient of the utility with respect to the weight space, W_{ij} and W'_i in the Action network according to the pseudo-code in Figure 3. Basically, the idea is to change the output of the action network in the direction of $F_u(t)$ by adjusting its weights (W and W').

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 actually 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 8 shows the modified feedback component. The choice of the value for the number of inner iteration, $Iner_Max$ depends on the problem. In this example, we have use both $Iner_Max = 1,000$ and $10,000$.

Figure 9 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 10 plots the error of the generated torque with the desired value $|\tau_d(t) - \tau(t)|$ after approximately 5,000 iterations. From this graph, we observe that the maximum error is about 0.02 Newtons. Better accuracy can be obtained by continuing the iterations.

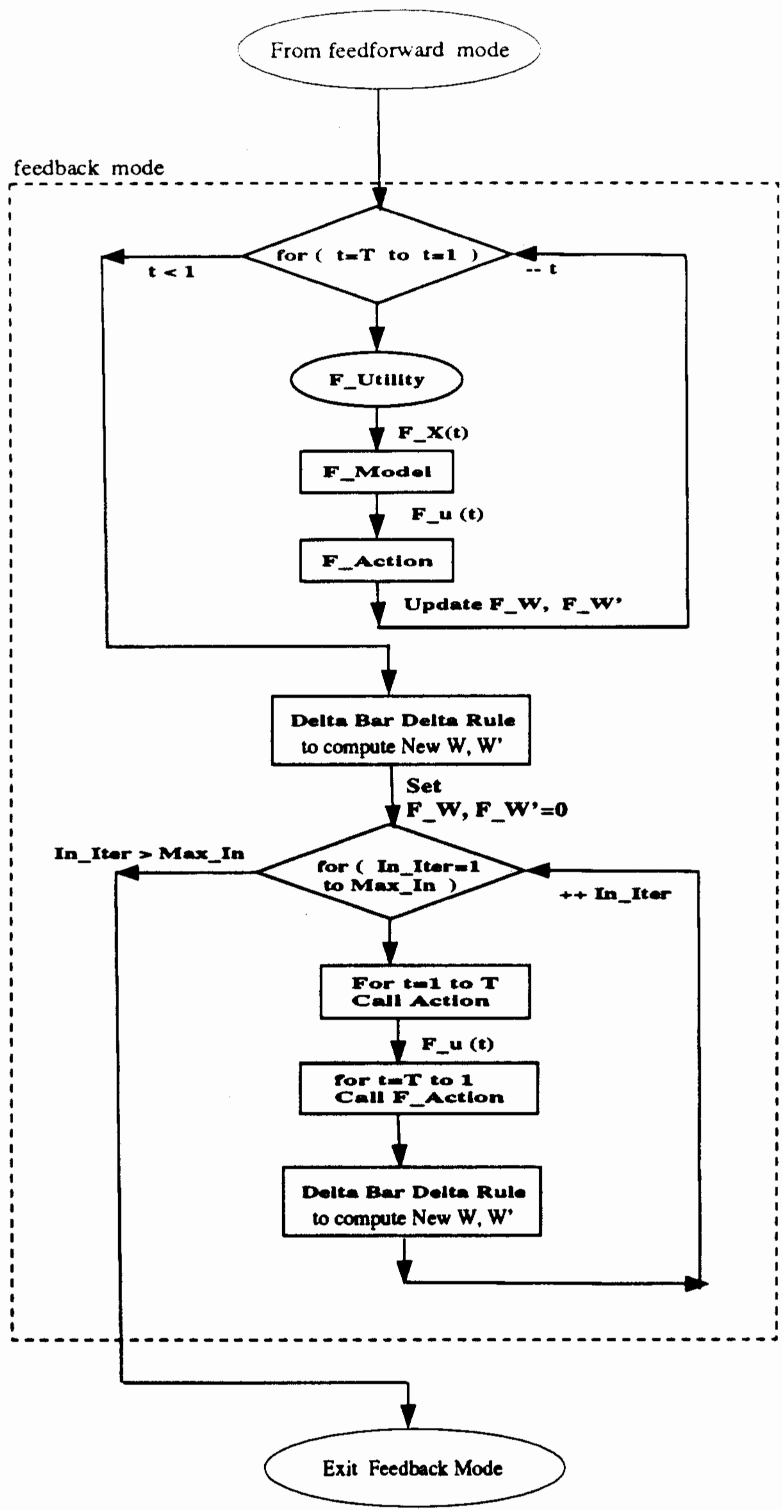


Figure 8: *A Flow-Chart for Modified Feedback Mode.*

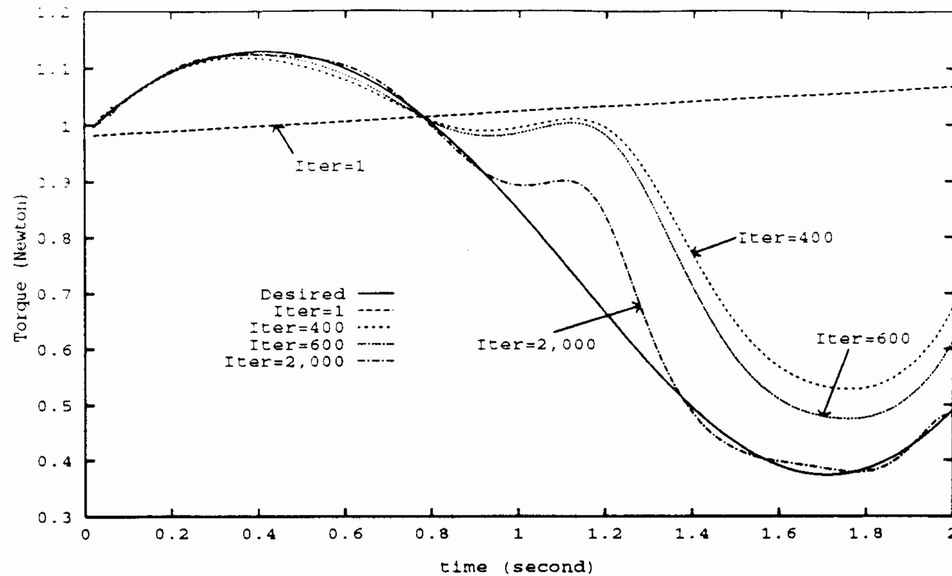


Figure 9: *Generated Torques at Different Iterations.*

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.

5 Parallel Implementation

In the previous sections, we have illustrated and demonstrated how the backpropagation of utility algorithm can be used to generate a series of control signals. The simple, one-dimensional planar robotic system was used as an example. Basically, the underlying concept of the algorithm is very simple. A neural network (the Model network) is used to emulate the dynamic system and provide proper feedback to adjust weights of the Action network. In short, the backpropagation of utility algorithm is a simple tool for neuro-control. However, the main drawback of the algorithm is the amount of execution time. For the one-dimensional manipulator system, in a Sun Sparc II computer running SunOs 4.1, it took approximately,

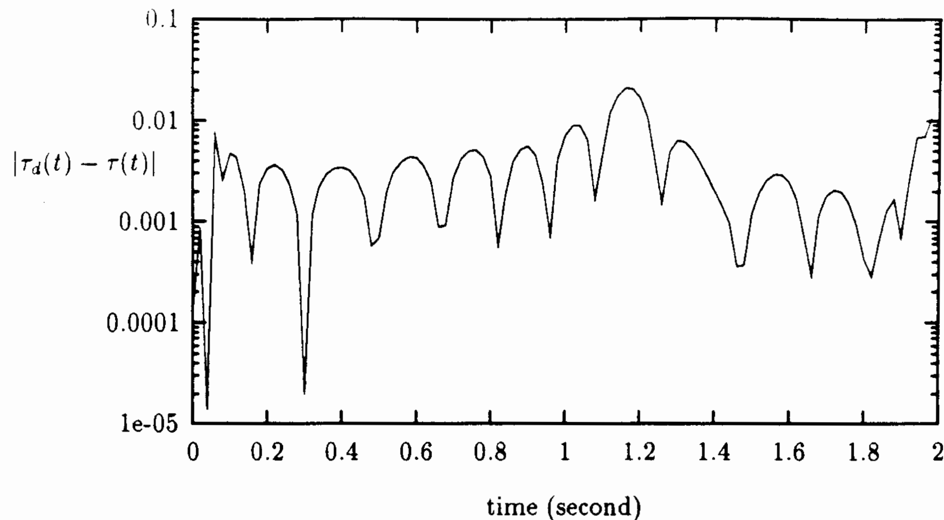


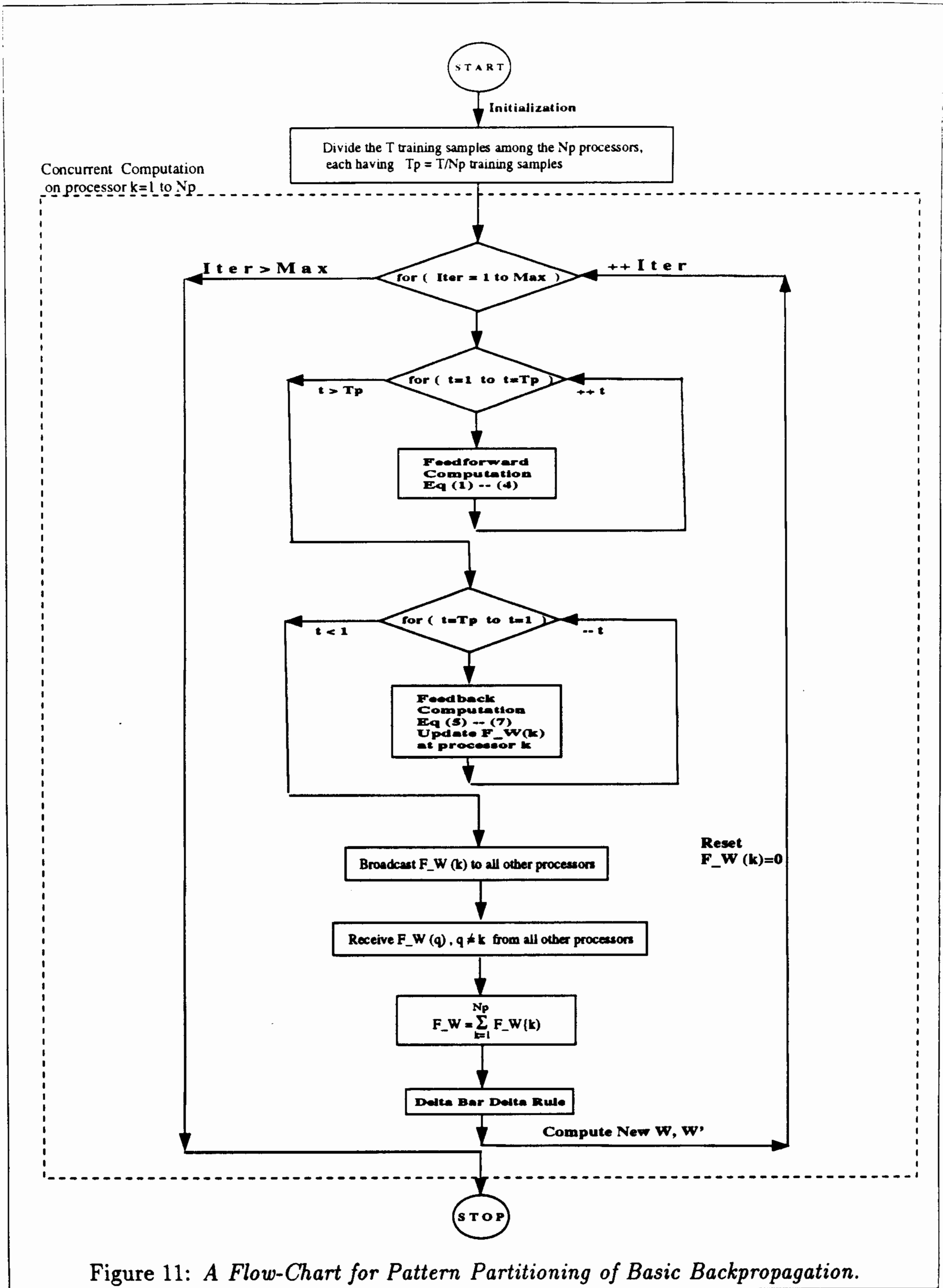
Figure 10: *Errors in Control Signals.*

43 seconds for each outer iteration with a given inner iteration $Inner_Iter = 1,000$. To improve the execution time, we explore parallel implementation of these algorithms. Both the basic and backpropagation through time algorithm can be parallelized by two techniques - *node partitioning* and *pattern partitioning* [12].

Basically, 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. In the next subsection, we include a brief description of node-partitioning. Section 5.2 describes pattern partitioning and provides execution time comparison for two examples.

5.1 Node-Partitioning

In node partitioning, nodes of the entire network are partitioned among different processors. Our strategy is to divide the number of hidden nodes in each hidden layer equally among the given number of processors. Because of the small number of nodes in the input and output



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

Figure 11: *A Flow-Chart for Pattern Partitioning of Basic Backpropagation.*

layers, these layers are assigned to a specific processor. For simplicity, we have each processor keep a copy of all weights of the network. In the forward loop, each processor calculates node activations $x_i(t)$ in parallel and broadcasts it to other processors for computation of next layer activations.

In the backward loop, each processor computes node errors and weight changes required for its incoming weights. Node errors are broadcasted to other processors for computation of previous layer errors. Each processor also computes total weight change required for all its incoming weights. At the end of backward loop, each processor broadcasts the weight changes calculated to update the other processor's buffer. (Each processor keeps a copy of all the weights in the network.) After this, Delta-bar-Delta rule is used to calculate new weights. 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 parallel implementation actually slows down the overall execution time. We, therefore, consider only the pattern-partitioning scheme.

5.2 Pattern-Partitioning

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}, F_W_i' are computed at every processor:

$$F_W_{ij} = \sum_{k=1}^{N_p} F_W_{ij}(k), \quad F_W_i' = \sum_{k=1}^{N_p} F_W_i'(k)$$

where $F_W_{ij}(k)$ and $F_W_i'(k)$ are the weight gradient of processor k computed based on its own subset of training samples. Upon obtaining the total weight gradient, delta-bar-delta

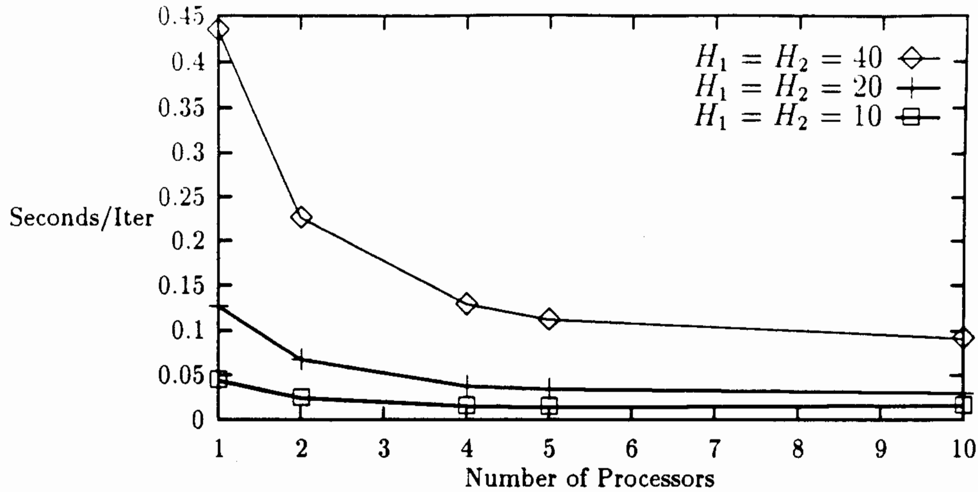


Figure 12: *Execution Time for Learning the Sine Function Using Pattern Partitioning.*

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 11. The case of backpropagation through time can be obtained in a similar manner with the proper inclusion of W'.

We implemented this pattern partitioning scheme for the basic backpropagation algorithm on Intel's 110-node Paragon parallel computer. As a simple test, we train two-hidden-layer neural networks of different sizes to learn $T = 100$ samples of the sine curve equally spaced over $(0, 2\pi)$. In particular, we investigated networks with $H_1 = H_2 = 10, 20, 40$ nodes in each layer, on $N_p = 1, 2, 4, 5, 10$ processors, with each processor having $T_p = 100, 50, 25, 20$ training samples respectively. Figure 12 shows the execution time per iteration on the different number of processors. From this figure, we observe that the execution time decreases with the number of processors. However, we believe that if the number of processor increase further, the execution time per iteration will not decrease continuously. We believe that eventually, the communication overhead associated with multiple processors will outweigh the advantages of parallel execution and the execution time per iteration will start to increase.

	Paragon (4 node)	PC (Pentium-90)	Sparc II	Sparcstation LX
Time/Iter	10.2 sec	14.3 sec	43.7 sec	84.8 sec

Table 6: *Comparison of Execution Time of Different Computing Platforms*

After testing the scheme through the learning of the sine function, we use the pattern partitioning scheme for the adaptation of the Action network discussed in Section 4.2. Figure 13 plots the execution time per outer iteration versus different number of processors. The amount of inner iterations is $Iner_Max = 1,000$. (See Figures 3 and 8 for the definition of outer and inner iterations).

We observe that, initially, the execution time also 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 large amount of communication among the 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 Figures3 and 8). Because of our modified feedback mode, for each outer iteration, there are $Iner_Max = 1,000$ number of inner iterations. Each of this inner iteration requires each processor to broadcast its local weight gradient to all.

To provide an approximate comparison of the performance of the pattern-partitioning scheme, Table 6 summarizes the execution time per outer iteration with inner iteration $In_Max = 1,000$ on different computer platforms for the adaptation of the Action network discussed in Section 4.2. For the single-processor machine, we executed the compiler-optimized program only when a single-user is logged on. We observe that 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.

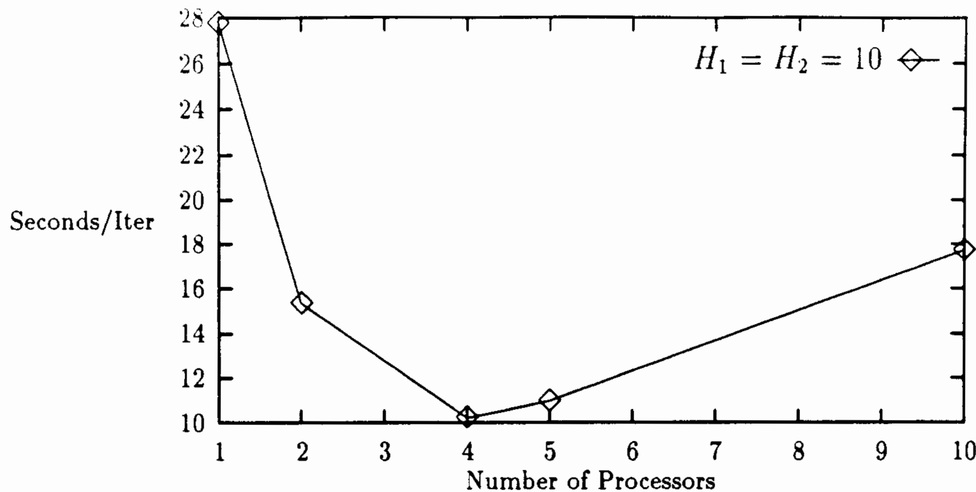


Figure 13: *Execution Time for Backpropagation of Utility.*

6 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. In this paper we demonstrated how to use the basic backpropagation and backpropagation through time algorithms as building blocks for the backpropagation of utility algorithm and explored parallel implementation of the algorithm on Intel's Paragon parallel computer.

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. We provided description of two parallel schemes, *node-partitioning* and *pattern-partitioning.* Basically, for node-partitioning, nodes of a neural network is partitioned among different processors, each computing the entire training set. Pattern-partitioning, on the other hand, partions the training patterns among different processors, each representing the entire network.

We found that for our example, with rather small networks (both the Action and Model networks are two-layered with ten hidden nodes on each layer), pattern-partioning is more appropriate. We first tested the pattern partioning scheme through the learning of a sine curve and then applied the technique for the control of one-dimensional robot with the Backpropagation of Utility algorithm. Comparison of the execution time for single and different number of multiple processors are included. We believe that the advantages of parallel implementation will be more pronounced for large problems that require bigger networks with more training samples.

References

[1] Richard L. Lippmann. "An Introduction To Computing With Neural Nets". *IEEE ASSP Magazine*, pages 4–22, April 1987.

[2] Judith E. Dayhoff. *Neural Network Architectures.* Van Nostrand Reinhold, New York, 1990.

[3] D. Rumelhart and J. McClelland. *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.

[4] B. Widrow and M.A. Lehr. "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation". *Proceedings of the IEEE*, 78(9):1415–1442, September 1990.

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

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

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

[9] 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.

[10] J. Tanomaru and S. Omatu. "Towards Effective Neuromorphic Controllers". In *IECON*, pages 1395–1400, Kobe, November 1991.

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

[12] 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.