CONTROL APPLICATIONS USING NEURAL NETWORKS

by

Barton Earl Showalter

S. B., Massachusetts Institute of Technology

(1987)

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1988

Signature of Author [illegible]
Department of Aeronautics and Astronautics

Approved by [illegible]
William D. Goldenthal
Technical Supervisor, CSDL

Certified by [illegible]
Professor Wallace E. VanderVelde
Thesis Supervisor

Accepted by [illegible]
Professor Harold Y. Wachman
Chairman, Departmental Graduate Committee

Control Applications Using Neural Networks

by

Barton Earl Showalter

This research applies artificial neural networks to the field of control resulting in an augmented approach to control system design. A feedforward network, or perceptron, contains a connected mesh of non-linear, thresholding neurons that accept external inputs and compute an output based on their interconnection weights. This network operates as a variable-gain compensator in a control loop receiving the system error and outputting a control action to the inverted pendulum on a cart. A teaching algorithm adjusts the neuron interconnection weights as the controller is "exercised" through a number of position commands to the cart. The experimental results illustrate the non-linear computational advantages of the neurons and demonstrate the ability of the network controller to adapt quickly to poorly modelled or time-varying dynamics.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I owe a special thanks to Professor Wally VanderVelde, my thesis supervisor, for his technical guidance and patience. I hope he has learned something about neural networks through our friendly association.

Thanks also to my friends at C. S. Draper Laboratory, especially the basketball crew, to whom I apologize again for not playing in the big game. Many thanks to Bill Goldenthal, my technical supervisor, who approached me with an exciting thesis topic and allowed me to pursue some original ideas. I am also indebted to Jim Cervantes, the Lisp guru, for hours of free-lance hacking and software consulting that somehow brought the Neural Network Controller to fruition. Another rising star on the Symbolics, Doug Eberman, also provided me with a nice graphing package and overall software support.

As for general sounding block and MacYahtzee companion, that distinction must go to my officemate and Texas-Ex, Walter Baker. The "can I erase this?" board, Speed Reversi, and afternoon outings with Gambi will be greatly missed.

Thanks to Carol, Ifeanyi and the kids for being my family in Boston. Finally, thanks to Mom, Dad, and the sibs for all their long-distance support during my five year vacation at MIT.

To My Wife

LIST OF SYMBOLS

E	error between actual and commanded plant state
$f(net_j)$	neuron output function at the value of net_j
$f'(net_j)$	slope of the neuron output function at the value of net_j
F	control force on cart (N)
g	gravity (9.8 m/sec^2)
$G1$	gain matrix for network inputs
$G2$	gain matrix for network outputs
$K1$, $K2$	gains on the classical controller X loop
$K3$, $K4$	gains on the classical controller θ loop
L	pole length (m)
L'	effective pole length (m)
M	cart mass (kg)
m	pole mass (kg)
M_p	percent overshoot
$\mathbf{net}$	vector containing all neuron internal states
net_j	sum of the weighted inputs or internal state of neuron j
$\mathbf{o}$	vector containing all neuron external states
o_j	output or external state of neuron j
p_{ik}	propagation term from neuron i to output neuron k
p_{kk}	propagation term for output neuron k
R	residual between actual plant state and model reference state
R_h	residual of input neuron h
s	Laplace operator

t_j	target output of neuron j
$\boldsymbol{u}$	network input vector
$\boldsymbol{U}$	plant input vector containing control commands
$\boldsymbol{U_M}$	model reference vector containing target plant control commands
w_{ij}	connection weight from neuron *i* to neuron *j*
$\Delta w_{ij}(p+1)$	current connection weight change
$\Delta w_{ij}(p)$	previous connection weight change
X	cart position (m)
X_c	commanded cart position
$\dot{X}$	cart velocity (m/sec)
$\dot{X}_c$	commanded cart velocity
$\ddot{X}$	cart acceleration (m/sec^2)
$\boldsymbol{y}$	network output vector
$\boldsymbol{Y}$	plant output vector containing state variables
$\boldsymbol{Y_D}$	desired state of the plant
$\boldsymbol{Y_M}$	model reference vector containing target plant state
δ_j	error of neuron *j*
$\delta_j^{(h)}$	error of neuron *j* for the residual input *h*
ε	scalar adaptive error
ζ_θ	damping coefficient of classical controller θ loop
ζ_x	damping coefficient of classical controller X loop
η	learning coefficient
θ	pole position (rad)
θ_c	commanded pole position
$\dot{\theta}$	pole velocity (rad/sec)
$\dot{\theta}_c$	commanded pole velocity

$\ddot{\theta}$	pole acceleration (rad/sec^2)
λ	constant determining steepness of the sigmoid function
μ	momentum coefficient
μ_c	coefficient of friction between cart wheels and track
μ_p	coefficient of friction between pole and pivot
π_{hk}	dynamic sign for plant state Y_h due to control command U_h
$\omega_{n\theta}$	natural frequency of classical controller θ loop
ω_{nx}	natural frequency of classical controller X loop

1 INTRODUCTION

The emerging technology of neural networks offers intriguing computational advantages to the field of control system design. A collection of interconnected, non-linear neurons provides a parallel processing structure that can build an input/output mapping of arbitrary form. Furthermore, simple methodologies can adjust the neuron interconnection weights to teach the network new configurations. These properties, among others, motivate this research to use a neural network as an adaptive compensator in a control loop. An informed approach to the problem requires a good background knowledge of the history of neural networks briefly discussed in Section 1.1. Section 1.2 defines the desirable characteristics of neural networks relating to control system design. The specific expectations of this research appear in Section 1.3 with particular emphasis on the problem conception.

1.1 THE HISTORY OF NEURAL NETWORKS

For many years a debate between *symbolism* and *connectionism* has raged in the field of artificial intelligence. Symbolism uses a series of symbolically coded messages in a computer program to solve problems analytically or heuristically. Connectionism relies on communication by excitatory and inhibitory signals passed between simple neuron-like processing nodes known collectively as a *neural network*. These two fields have seen research interest in them wax and wane over the past forty years. Today both fields remain strong foundations of artificial intelligence, but a resurgence in connectionism offers many new possible applications, especially in the fields of signal processing and automatic control.

The first major spark in the field of neural networks was a paper entitled *A Logical*

Calculus of the Ideas Immanent in Nervous Activity written by McCulloch and Pitts in 1943. Their "neuro-logical networks" with linear threshold elements attempted to exploit the computational nature and structure of biological nerve cells. In 1947, the same authors wrote a second paper, *How We Know Universals*, describing the first practical application of a "neural network" to recognize particular spatial patterns invariant of geometrical transformations. With these two revolutionary papers, the field of neural networks was born.

Donald Hebb's book, *The Organization of Behavior*, laid the foundations of learning and internal representation in networks. To this day, many learning algorithms for simple feedforward networks find their basis in Hebb's work. Minsky in 1951 built the first "mind-like" machine that used a reinforcement-based learning rule to teach a collection of forty electronic units. However, by the end of the 1950's, the advent of the serial computer shifted research emphasis in artificial intelligence from neural networks and learning towards effective heuristic-based programs.

With the publication of *Principles of Neurodynamics* by Rosenblatt in 1962, interest in neural networks again swelled. The author proved important convergence properties of simple feedforward networks with non-linear neurons. These *perceptrons* could solve many interesting problems using the simple methodology that punishes the effects of individual neurons which fail to contribute to a desirable output. This learning algorithm was not foolproof as it failed mysteriously on seemingly simple exercises. Later, Minsky and Papert in their book, *Perceptrons*, addressed this problem concluding that the limitations of such an architecture to *learn* is inherent in its ability to *represent*.

The pessimism of *Perceptrons* again took the field of connectionism into an ebb of activity for most of the 1970's, but the 1980's saw yet another revival in learning machines. Hopfield published a number of important papers on his completely interconnected symmetric network which associated inputs to outputs. Recently, the two volume work of Rumelhart and McClelland and The PDP Research Group called *Parallel*

Distributed Processing proposed a new algorithm to teach perceptrons. This new rule, termed "backpropagation", uses input/output pairs to teach a perceptron with hidden neurons. This was a significant development since a perceptron with hidden layers sandwiched between the input and output layer is capable of more complex mappings.

1.2 ADVANTAGES OF NEURAL NETWORKS

There are a number of reasons for interest in neural networks and their application to controller design. Of great importance is the way in which a network stores empirical knowledge in the neuron interconnection weights. Information is smeared across a number of units that collectively arrive at an answer. If one particular neuron fails, the effect on the stored memory is minimal and if subsequent neurons fail, the system degrades gracefully. This is a property usually not found in other adaptive approaches. Furthermore, the perceptron constructs memory through association, giving it the ability to operate effectively in a region it has yet to explore.

Neural networks are also capable of learning good behavior in new or changing environments. With the work of Rumelhart and McClelland, a perceptron of sufficient hidden layers can be taught any conceivable transfer from input to output. The application of the teaching rules is very simple and permits greater flexibility than other current adaptive routines.

Finally, the actual development of network controllers in hardware offers a tremendous speed advantage due to the parallel processing of the nodes. Hopfield investigated implementing his network on a microchip and found staggering improvements in processing time.

1.3 CONTROL APPLICATIONS

The main thrust of this research is to develop a control system that is capable of adapting quickly to poorly modelled environments using neural network technology. In the past, researchers have approached this problem in a number of ways, each with its own set of limitations. One method, called gain scheduling, builds a string of controllers designed to operate about specific *linearized* conditions. As the plant moves from one envelope into another the algorithm switches controllers. This brute force method requires not only good knowledge of all the possible operating regions, but also increased computer memory. Since it is constrained to a fixed table of controllers, the methodology does not allow "on the fly" adjustments. Other approaches involve current adaptive techniques which have failed to perform adequately in many situations, often "blowing up" after a long period of good performance.

This paper offers a new approach that uses the neural network as a *variable-gain compensator* in the control loop. The network receives the system error and outputs the appropriate control commands to the plant. Initially, the plant may perform poorly, but after "exercising" the network controller the weights adjust to achieve good performance. If the plant's dynamics change over time, the network senses a degradation in performance and makes the necessary weight changes. Assuming a reasonable range of operating conditions, the network continuously reviews the plant performance and adjusts the weights, resulting in an effective controller for all operating regimes.

2 PERCEPTRONS

This chapter provides background information concerning the internal mechanics of the perceptron. A neural network is a mesh of neurons that exchange signals across directed, weighted connections. A perceptron is a special network of neurons arranged in layers. Each layer only receives inputs from downstream layers and can only output to upstream layers. Section 2.1 introduces the neuron and its related processing functions. The effect of hidden layers on generating an input/output mapping are presented in Section 2.2. Section 2.3 discusses the perceptron as a controller. Section 2.4 introduces network propagation algorithms using recursive methods. Finally a brief overview of teaching a neural network is offered in Section 2.5. The material focuses mainly on the perceptron and its properties with some general comments on neural networks.

2.1 NEURONS

A neuron is an independent processing unit with an arbitrary number of inputs and one output. A signal path between neurons carries a connection weight or strength. An input to a neuron is the product of the activation along an input line and the associated weight. Each neuron performs two functions:

1) combine the weighted inputs
2) apply the output function

These two functions transform the input or internal state of the neuron to an output or external state. The aggregate effect of the functions is the input/output characteristic of the neuron which can be linear or non-linear. Linear neurons offer simplified mathematics, but lack the important thresholding capabilities of their non-linear counterparts.

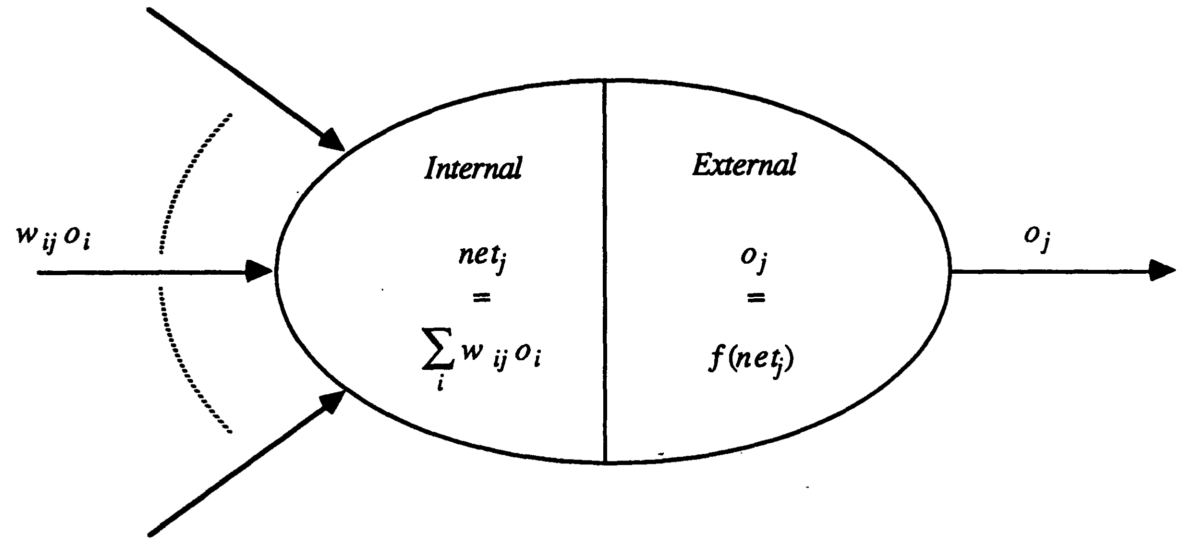


Figure 2.1 -Model of a Neuron

2.1.1 Summation Function

The internal state of the neuron is equal to the sum of the weighted inputs.

$$net_j = \sum_i w_{ij} o_i$$

Each incoming line contributes the product of the signal strength and connection weight to the neuron internal state.

2.1.2 Output Function

The output function alters the internal state of the neuron by applying a thresholding function.

$$o_j = f(net_j)$$

There are a number of candidate functions which limit the neuron output.

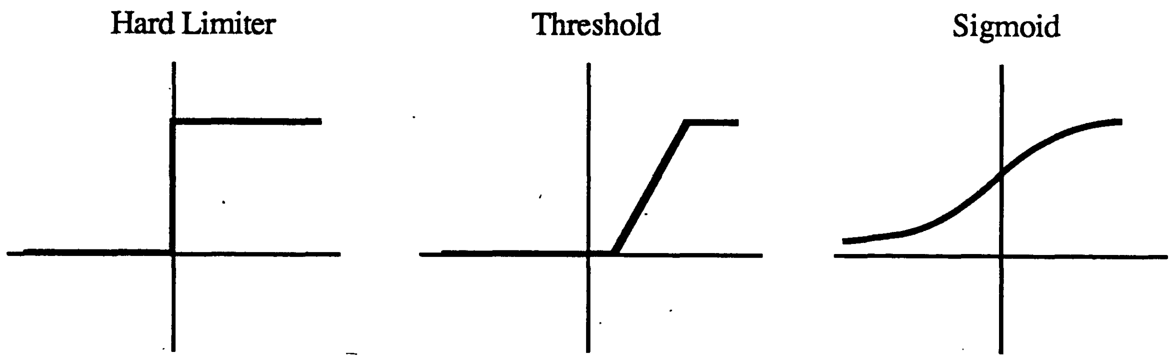


Figure 2.2 - Output Functions Ranging from 0 to 1

The hard limiter allows the output to assume only two values with a discontinuity at zero. The threshold contains a linear region, but the transition from linear to the limit values is again discontinuous. The sigmoid function is both nondecreasing and differentiable - a property Rumelhart and McClelland refer to as semi-linear. For input values near zero, the sigmoid appears linear and flattens gradually for large negative and positive inputs. Each of the function's output in Figure 2.2 ranges from 0 to 1, but to achieve symmetry and full representation for positive and negative values, the output must include values between -1 and 1.

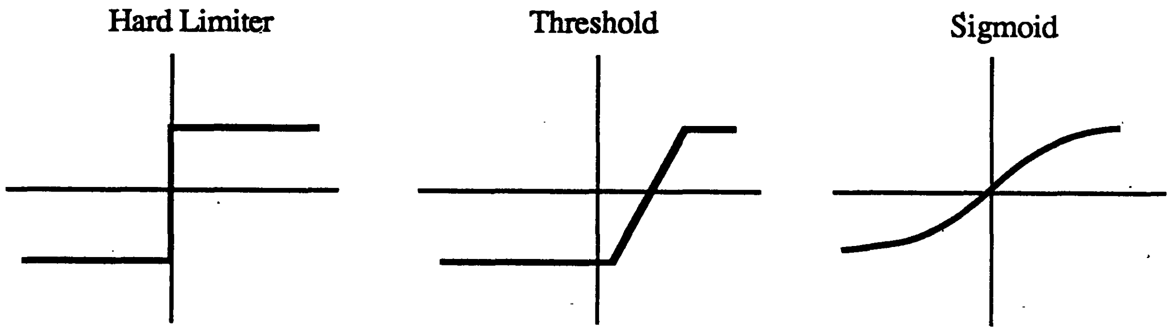


Figure 2.3 - Shifted Output Functions Ranging from -1 to 1

The output function used in this research is the sigmoid in Figure 2.3 with the following equation:

$$f(x, \lambda) = 2\left(\frac{1}{1 + e^{-\lambda x}} - 0.5\right)$$

where x is the input to the sigmoid and λ is a constant determining the steepness of the slope at the origin.

2.2 PERCEPTRON SUBSTRUCTURE

The perceptron substructure depends on the network interface to the outside world. This interface consists of a set of designated input neurons whose internal states are fed from an outside source and a set of output neurons whose external states are the network output. In a perceptron, a layer of distinctly input neurons is in the first column, a layer of distinctly output neurons is in the last column, and any number of hidden layers lie in between. By convention all lines carrying network input and all lines transmitting network output carry a weight of unity. Therefore, the weights that change must connect two neurons.

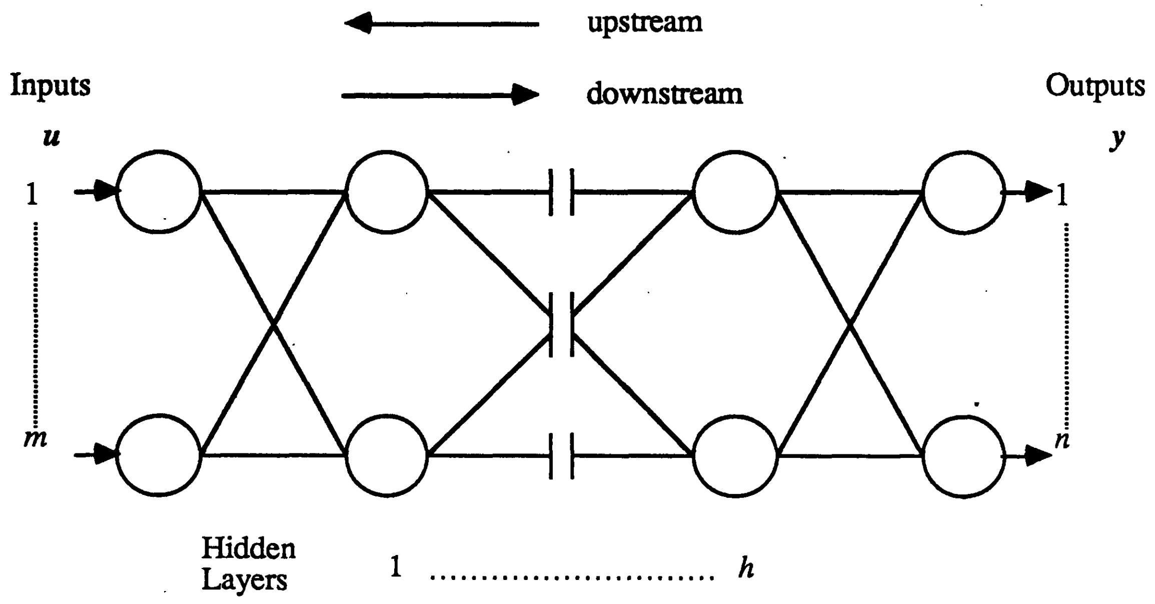


Figure 2.4 - The Perceptron

The perceptron in Figure 2.4 contains *m* inputs, *n* outputs, *p* neurons, and *h* layers. The network input vector ***u*** and the output vector ***y*** are defined as:

$$\boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ \cdot \\ u_m \end{pmatrix} \qquad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{pmatrix}$$

The vectors of the neuron internal state ***net*** and external state ***o*** are defined as:

$$\boldsymbol{net} = \begin{pmatrix} net_1 \\ net_2 \\ \cdot \\ \cdot \\ \cdot \\ net_p \end{pmatrix} \qquad \boldsymbol{o} = \begin{pmatrix} o_1 \\ o_2 \\ \cdot \\ \cdot \\ \cdot \\ o_p \end{pmatrix}$$

The output vector ***y*** is a subset of the output vector for all neurons, ***o***.

2.3 THE PERCEPTRON AS A CONTROLLER

The perceptron as a compensator in a control loop establishes a mapping between its input and output neurons. The input neurons receive the measured state variables of the plant and the output neurons transmit the control variables to the plant. A simple problem with two inputs and one output can be visualized in two-dimensions as a phase plot. The range of the two input values are graphed along the x and y axis. For any state of the plant there exists a value on the graph for the output.

$$u = f(x,y)$$

Though the visualization of the decision region is difficult, this argument can be generalized to any number of m inputs and n outputs.

The power of a perceptron to solve a particular mapping from input to output increases with the addition of hidden layers. Furthermore, the flexibility of a perceptron depends on the complexity of the decision shape it can create in a mapping space. A simple example attempts to separate the two classes A and B in a two-dimensional mapping space. A perceptron with no hidden layer can only separate the classes by a hyperplane (or line in two dimensions). A perceptron with one hidden layer effectively separates the classes using open or closed convex regions, whereas a perceptron with two or more hidden layers can form an arbitrary boundary.

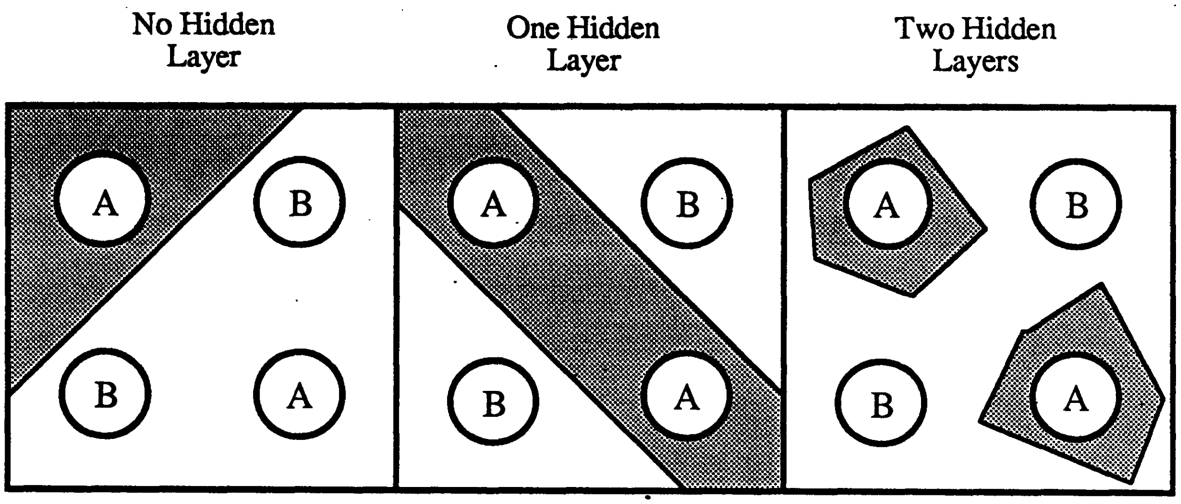


Figure 2.5 - Possible Decision Regions for Various Perceptron Structures

Since the number of hidden layers defines the input/output capabilities of the network, the complexity of the desired controller dictates the network structure. If the expected phase plane requires only linear separation of control regions, no hidden layers are necessary. As the control problem gets more non-linear, hidden layers can be added to allow for more convoluted control decision regions. However, since additional layers allow for many possible decision regions, converging on the desired set of weights becomes very difficult with more layers.

2.4 NETWORK PROPAGATION

A feedforward network propagates by updating the external states of the neurons as the internal states change. This research uses two methods to propagate the perceptron.

1) complete propagation
2) one-step propagation

Complete propagation processes each layer sequentially from left (input layer) to right (output layer). Propagation is not finished until the input signals have filtered through the network to give an associated set of outputs. The outputs are a function of the inputs and the connection weights. One-step propagation allows a signal on any line to pass through only one neuron for each time step. The outputs are now functions of the inputs, the connection weights, and the effect of past inputs held on internal lines within the network. By maintaining the impact of past inputs on the present output, the network experiences a form of memory different from resident neuron memory.

The computer simulation uses the techniques of one-step and complete propagation in an object-oriented environment. Each neuron is abstracted as a data structure and can be told to collect its inputs, process them, and output the resulting signal. This is a local neuron update. The propagation of the perceptron depends on the order in which the local neuron updates are executed. The methodologies are identical for both linear and non-linear neurons.

2.4.1 Complete Propagation

To achieve a complete propagation the neurons must be updated beginning with the input layer. The methodology continues updating the neurons by progressing through the network layers to the output layer. This has the effect of carrying the network input *downstream* from the input layer to the output layer. The output of the last neuron reaches steady-state given a constant input.

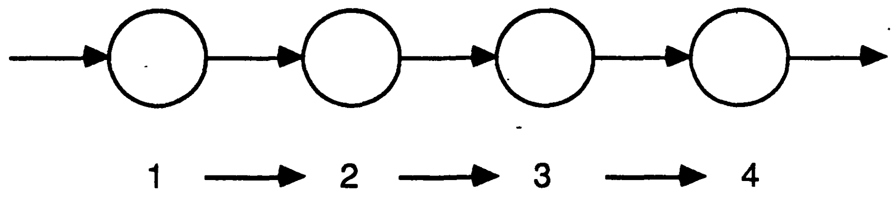


Figure 2.6 - Complete Propagation Sequence

2.4.2 One-Step Propagation

One-step propagation begins on the output layer and works *upstream* to the input layer. In this fashion, each neuron fires using old external states of neurons yet to update. This technique does not require storing of old values to achieve memory in the system.

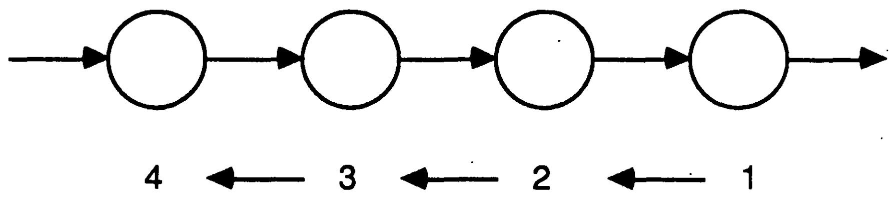


Figure 2.7 - One-Step Propagation Sequence

2.5 TEACHING

Teaching a network means changing the neuron connection weights. Not all neural networks are taught, some begin with a defined set of weights which never change. The basic learning rule for a perceptron proposed by Hebb increases the weights along the paths whose signals tended to produce the desired output. A weight along a path whose signal caused the output to diverge from the desired could be decreased or kept unchanged. Hebbian learning is rarely used in this simple form today, but almost all teaching algorithms are descended from this manner of adjusting the connection strength.

This research teaches the perceptron using a group of learning rules known as *error propagators*. Each of these algorithms involve a number of steps.

1) Evaluate the performance of the network
2) Generate a local error at the point of evaluation
3) Propagate the local error to all contributing neurons
4) Change the weights between neurons according to their error

Error propagators employ a gradient descent in the local error with respect to weight changes.

Rumelhart and McClelland's learning rule, called *backpropagation*, evaluates the network performance using desired input/output pairs. The local error originates at the network *output*. These errors are then recursively "backward propagated" to all the contributing neurons. The connection weight between two neurons changes according to the error of both neurons. When the network operates as a compensator in a control loop, backpropagation requires knowledge of the network output or control action given an input or state of the plant. This constrains the weights to converge on a preset control law.

This research proposes a new algorithm, called *forpropagation*, which generates a local error based on a comparison of the actual and target network *input*. This evaluation uses a desired plant state or network input instead of a desired control action to the plant. The local error originates at the inputs and is "forward propagated" to the output neurons using a term which relates the change in input to an associated change in output. Forpropagation then proceeds like backpropagation using the neuron errors to adjust the weights. The weights then converge on values that result in a target response of the plant regardless of the necessary control action.

3 NETWORK TEACHING STRATEGIES

This chapter compares two methods for teaching neural networks which operate as compensators in a control loop. Both algorithms, classified as error propagators, use gradient descent techniques to adjust the connection weights of a perceptron with semi-linear neurons. Backpropagation is a popular method effective in many problems, but not well-suited for control applications. Since backpropagation requires input/output pairs, the network output must be known which requires prior knowledge of the desired compensator. This constrains the network to converge on a predetermined set of weights. As a result, a new technique termed forpropagation specifically addresses the problem of teaching a compensator. Section 3.1 discusses the architecture of a controller with a network compensator. Section 3.2 introduces issues related to the general method of error propagation teaching algorithms. Sections 3.3 and 3.4 give quick overviews of the backpropagation and forpropagation algorithms.

3.1 THE NEURAL NETWORK AS A CONTROLLER

The neural network control loop (Figure 3.1) consists of a neural network in series with a plant. The network receives a desired state of the plant Y_D minus the actual state Y and passes the controlling inputs U to the plant.

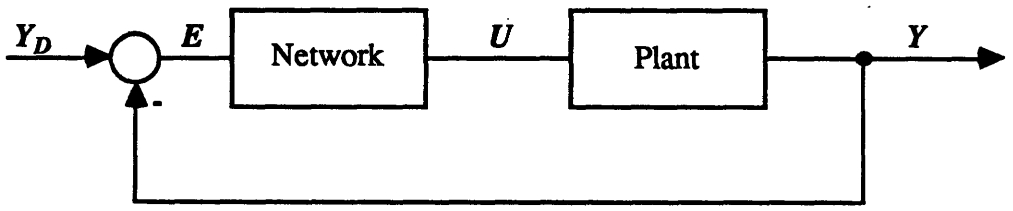


Figure 3.1 - Network Control Loop

The network acts as a compensator in the control loop. By changing the connection weights the network can produce any desirable transfer between input and output. If the network maintains memory either from resident neuron memory or through one-step propagation, then it is a dynamic compensator. A network continually taught while operating within the control loop is an adaptive compensator.

The connection weights change according to a learning rule which uses an evaluation of the network's performance as a controller. One current approach to this problem uses the backpropagation algorithm which evaluates the network *output* based on a desired input/output pair. Unfortunately, this requires complete knowledge of the desired control function. At best, the network will learn a target controller but cannot adjust the compensator parameters on-line.

The main theoretical contribution of this research is a second approach to teaching a network in a control loop. This new approach, named forpropagation, produces an evaluation at the network *input* based on the difference between the actual state variables and their commanded values. By tracking the plant state exclusively there is no need to specify an initial controller, the network converges on the weights necessary to control the plant. If the dynamics vary over time, the network senses the error and adjusts the weights accordingly.

3.2 TEACHING NEURAL NETWORKS USING ERROR PROPAGATION

The method for teaching networks using error propagation produces a gradient descent in error with respect to weight changes. At every step, the routine calculates the partial derivatives $\frac{\partial E}{\partial w_{ij}}$ of the total error with respect to each weight then moves a certain distance in the direction of the negative gradient vector $-\frac{\partial E}{\partial W}$. Assuming no problems with local minima, the error will ultimately reach zero. The following sections introduce topics common to all error propagation methods.

3.2.1 The Lesson

Error propagation of all types adheres to a strategy of data presentation with weight changes known as the lesson structure. Sometimes the teaching session occurs off-line in a procedure called *fixed learning*. After the network is taught, it is used in an application with fixed weights. Another common method involves on-line or *adaptive learning*. Here the weights are continuously adjusted while the network performs its function.

3.2.2 Learning Rate and Momentum

The learning coefficient η dictates how quickly the weights change. Ideally this is as high as possible without leading to oscillations or instability in the weight changes. An effective way to increase the learning rate while filtering out high frequency oscillations is to include a momentum coefficient μ which includes the effect of the last weight change on the current change. This keeps the weights from adjusting drastically due to high-frequency noise. Some researchers set $\mu + \eta = 1$, while others let the coefficients vary freely. A typical setting would be $\mu = 0.9$ and $\eta = 0.1$.

3.2.3 Periodic Updating

Often it is desirable to accumulate the weight change over several presentations of data and then incrementally adjust the weights. Sejnowski [27, 28] used this procedure when he presented the letters of a word sequentially and then adjusted the weights once for each word according to the summed contribution of each letter. This method allows the network to treat a sequential set of data as one.

3.2.4 Weight Initialization

Weight initialization with most teaching rules is a crucial step incorporating prior information. With error propagation the network can start as a clean slate. However, it is necessary to initialize the weights to small random values to break the symmetry of the network and allow connections to assume different weights. Usually the weights are uniformly distributed over a small range of positive and negative numbers.

The closer the initial weights are to the desired final answer, the faster the network converges. In addition, if the starting weights are much larger than the target values, the network receives misleading information from the plant resulting in weight instability. Assuming the neurons operate in their linear region, the network weights could be initialized using the best information of the final control law. This represents a "first-cut linear guess" at the desired compensator.

3.3 BACKPROPAGATION

Backpropagation in feedforward networks employs the generalized delta rule to teach a network input/output pairs. The method adjusts the connection weights by evaluating the error of the output layer. A teaching session using backpropagation cycles through a number of input/output pairs. For each presentation, the weights change slightly to register the effect of that pair on the final network.

After a sufficient teaching period, the network possesses the ability to associate an input with a corresponding output. For example, suppose the network is taught using desired outputs that are the squares of the corresponding inputs. Also assume only the pairs of odd numbers and their squares are used to teach the network. In theory, after the teaching is complete, the network will not only output the squares of the odd number inputs, but it should also build the association to estimate the even number squares.

3.3.1 Teaching Controllers Using Backpropagation

Figure 3.2 shows the network control loop augmented by a backpropagation adapter. Since the network adjusts the weights while attempting to control the plant this is a form of adaptive learning. The reference in the figure represents the *desired compensator* for the plant. The network at best can only mimic the reference structure. If the reference compensator is poorly designed or the plant dynamics are not well known, the network controller cannot perform well. The adapter receives the residual ***R*** and applies the rules of backpropagation to the network.

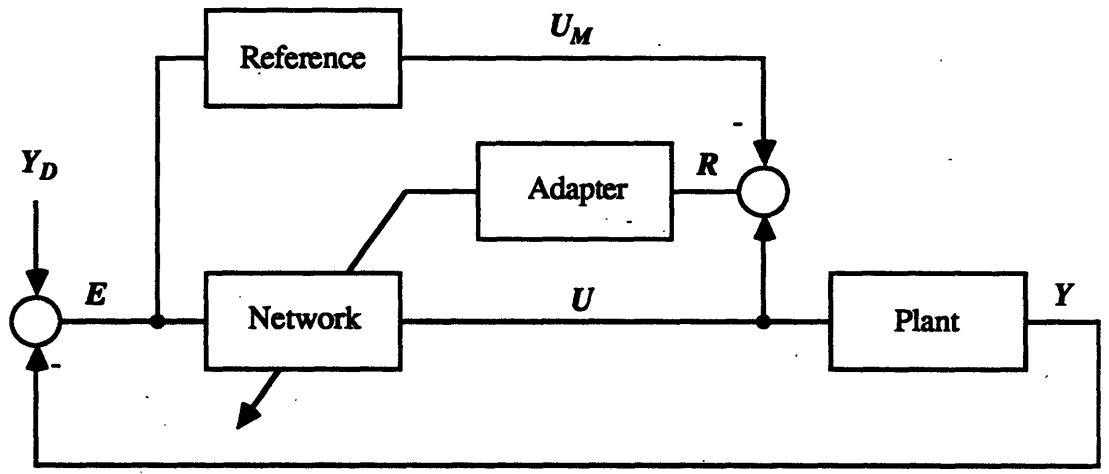


Figure 3.2 - Backpropagation Control Loop

This design builds a compensator into the associative memory of the network. When the network operates as a controller, it associates certain states of the plant with approximate control actions. Unfortunately, the approach requires knowledge of a controller and assumes no plant modelling errors. In most control systems, the desired state of the plant is known but not the desired control action. This need for a desired output severely limits the capabilities of backpropagation as a teacher for a controller.

3.3.2 Applying Backpropagation

Teaching a network using backpropagation, or any other neural network teaching algorithm, requires adherence to a number of rules. Foremost, the network must be a perceptron with semi-linear neurons. Special attention must be paid to the ordering of steps to change the weights as well as the setting of a number of teaching parameters.

Backpropagation is applied in two steps. First, the routine generates the neuron errors recursively starting at the output using the following equations for output and hidden neurons:

$$\textit{output neurons} \qquad \delta_j = (t_j - o_j)\, f'(net_j)$$

$$\textit{hidden neurons} \qquad \delta_j = f'(net_j) \sum_k \delta_k w_{jk}$$

where t_j is the target output and o_j is the actual output. Second, the weights are changed according to:

$$\Delta w_{ij}(p+1) = \eta\, \delta_j o_i + \mu\, \Delta w_{ij}(p)$$

where η is the learning coefficient, μ the momentum coefficient, $\Delta w_{ij}(p)$ is the weight change made during the previous lesson, and $\Delta w_{ij}(p+1)$ is the weight change for the current lesson.

3.4 FORPROPAGATION

Forpropagation adjusts the connection weights based on an evaluation of the *inputs*. The weights continually adjust as the network attempts to control the plant. If the error grows too large or the plant becomes unstable, the system reinitializes and a new lesson begins. After a number of lessons, the weights converge on values that produce a desired response in the plant. Forpropagation is a natural methodology allowing the user to specify a performance for the plant. If the plant dynamics change, the network senses the error and adjusts the weights to again achieve the desired response.

3.4.1 Teaching Controllers Using Forpropagation

Figure 3.3 shows the network control loop now augmented by a forpropagation adapter. The reference in the figure represents the overall *desired plant response*. The adapter receives a residual based on the desired and actual plant states and applies the rules of forpropagation to the network.

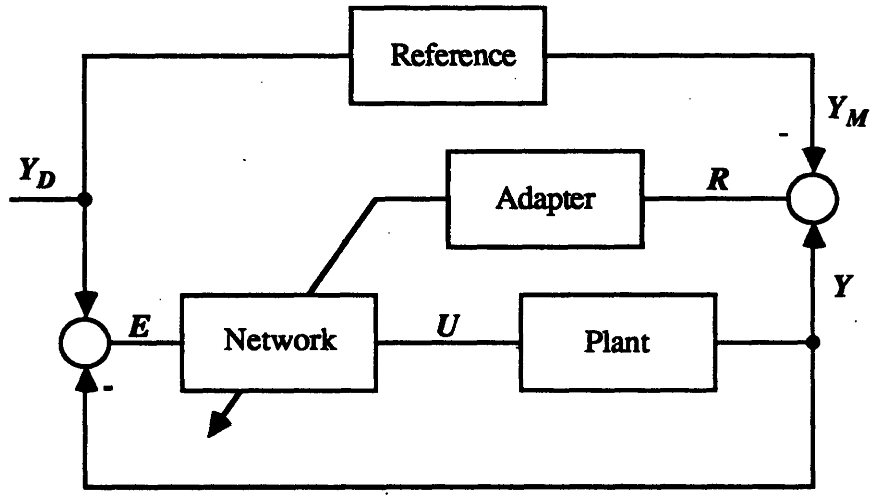


Figure 3.3 - Forpropagation Control Loop

Each of the major components of Figure 3.3 are discussed at length in the upcoming chapters.

3.4.2 Applying Forpropagation

Forpropagation, like backpropagation, requires a perceptron with semi-linear neurons. Issues related to gradient descent techniques also remain unchanged. A complete derivation of the forpropagation algorithm explaining all terms and notation appears in Chapter 7.

Forpropagation performs three operations. First, it calculates the propagation terms beginning at the output using the two equations for the output and hidden neurons:

output neurons $$p_{kk} = f'(net_k)$$

hidden neurons $$p_{ik} = f'(net_i) \sum_j w_{ij} p_{jk}$$

Second, the routine generates the neuron errors δ for both the hidden and output neurons using the residual vector $\boldsymbol{R}$.

output neurons $$\delta_k^{(h)} = -R_h \pi_{hk} p_{kk}$$

hidden neurons $$\delta_j^{(h)} = -R_h \sum_{k}^{outputs} \pi_{hk} p_{jk}$$

where π_{hk} is defined as the dynamic sign term. A neuron error represents how forpropagation changes the neuron's weighted sum input (i.e. - its upstream connection weights) to decrease the value of the residual. Finally, the weights are adjusted using the equation:

$$\Delta w_{ij}(p+1) = \eta\, o_i \sum_{h}^{residuals} \delta_j^{(h)} + \mu \Delta w_{ij}(p)$$

where η is the learning coefficient, μ the momentum coefficient, $\Delta w_{ij}(p)$ is the weight change made during the previous lesson, and $\Delta w_{ij}(p+1)$ is the weight change for the current lesson.

4 THE PLANT

The dynamic problem chosen for the neural network controller is the inverted pendulum on a cart. This chapter defines the plant and its state variables and presents the derivation of the equations of motion.

4.1 PROBLEM DEFINITION

A cart rests on a straight and level track (Figure 4.1) with an inverted pole attached to its center by a pivot. The cart must remain on the track and the pole can move only in the vertical plane of the track. A control force F can be applied on either side of the cart at its center of mass. One frictional force acts between the cart wheels and the track and another between the pole and the pivot.

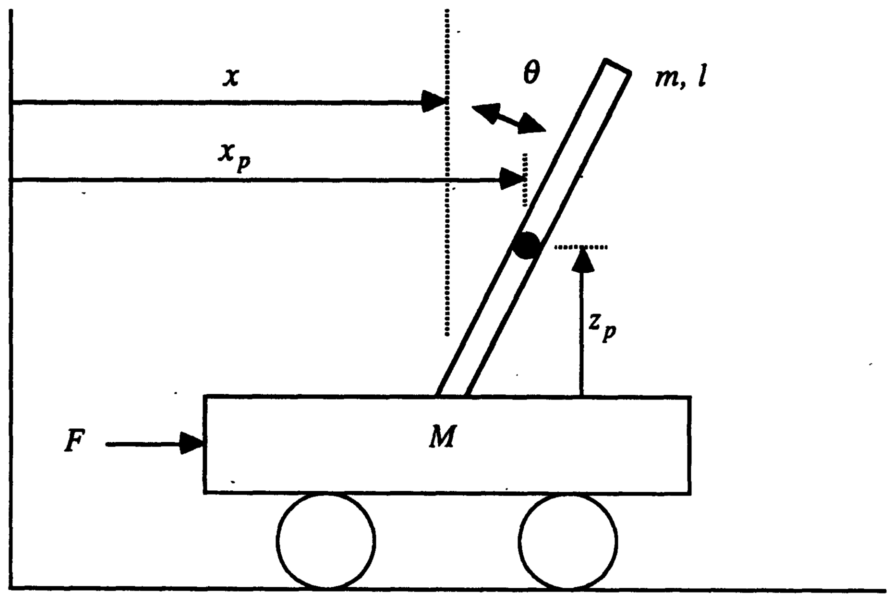


Figure 4.1 - The Cart-Pole Problem

Full-state feedback includes the cart's position X and velocity $\dot{X}$ and the pole's angular position θ and velocity $\dot{\theta}$. The two measurements X_p and Z_p are used in the Lagrangian derivation of the equations of motion. The important parameters include:

g	=	gravity (9.8 m/sec^{2})
M	=	mass of cart
m	=	mass of pole
L	=	half length of pole
μ_c	=	coefficient of friction of cart on track
μ_p	=	coefficient of friction of pole on cart

4.2 EQUATIONS OF MOTION

To derive the equations of motion of the cart-pole problem using Lagrangians, first define the coordinate transformations X_p and Z_p, and their first derivative:

$$X_p = X + L\sin\theta$$

$$\dot{X}_p = \dot{X} + L\cos\theta\,\dot{\theta}$$

$$Z_p = L\cos\theta$$

$$\dot{Z}_p = -L\sin\theta\,\dot{\theta}$$

Now write the kinetic energy of the system E_k including terms for the cart's translational energy and the pole's translational and rotational energy.

$$E_k = \frac{1}{2}M\dot{X}^2 + \frac{1}{2}m\left(\dot{X}_p^2 + \dot{Z}_p^2\right) + \frac{1}{18}m^2L^4\dot{\theta}^2$$

Substituting the definitions of X_p and Z_p and simplifying:

$$E_k = \frac{1}{2}(M+m)\dot{X}^2 + \frac{2}{3} m L^2 \dot{\theta}^2 + m L \cos\theta \dot{X} \dot{\theta}$$

The potential energy of the system E_p depends only on the pole.

$$E_p = m g Z_p = m g L \cos\theta$$

Now write the Lagrangian of the system.

$$\Lambda = E_k - E_p = \frac{1}{2}(M+m)\dot{X}^2 + \frac{2}{3} m L^2 \dot{\theta}^2 + m L \cos\theta \dot{X} \dot{\theta} - m g L \cos\theta$$

The forcing terms for X and θ are:

$$\Xi_x = F - \mu_c sgn(\dot{X})$$

$$\Xi_\theta = -\mu_p \dot{\theta}$$

The force balance equations in both X and θ are defined as:

$$\frac{d}{dt}\left(\frac{\partial \Lambda}{\partial \dot{X}}\right) - \frac{\partial \Lambda}{\partial X} = \Xi_x$$

$$\frac{d}{dt}\left(\frac{\partial \Lambda}{\partial \dot{\theta}}\right) - \frac{\partial \Lambda}{\partial \theta} = \Xi_\theta$$

Evaluating the above equations and rearranging gives the two equations in $\ddot{X}$ and $\ddot{\theta}$ that describe the motion of the cart-pole.

$$\ddot{X}(t) = \frac{F(t) + m L\left(\dot{\theta}^2(t)\sin\theta(t) - \ddot{\theta}(t)\cos\theta(t)\right) - \mu_c sgn\left(\dot{X}(t)\right)}{M+m}$$

$$\ddot{\theta}(t) = \frac{g\sin\theta(t) + \cos\theta(t)\left(\frac{-F(t) - m L \dot{\theta}^2(t)\sin\theta(t) + \mu_c sgn\left(\dot{X}(t)\right)}{M+m}\right) - \frac{\mu_p \dot{\theta}(t)}{m L}}{L\left(\frac{4}{3} - \frac{m \cos^3\theta(t)}{M+m}\right)}$$

These are the two equations used in the cart-pole simulation. For a given state of the plant, the routine first determines $\ddot{\theta}$ and $\ddot{X}$ and then uses a numerical integration scheme to determine the pole and cart positions and velocities. A simplified set of dynamic equations assume:

- no friction ($\mu_c = 0$ and $\mu_p = 0$)
- small angles ($sin\theta = \theta$ and $cos\theta = 1$)
- cart mass is much greater than pole mass ($M + m = M$ and $m = 0$)

Now the equations defining the motion of the cart-pole are:

$$\ddot{\theta} = \frac{g}{L'}\theta - \frac{F}{L'M}$$

$$\ddot{X} = \frac{F}{M}$$

where L' is the effective pole length defined as:

$$L' = \frac{J + ML^2}{ML} = \frac{\frac{1}{3}ML^2 + ML^2}{ML} = \frac{4}{3}L$$

These simplified state equations are used in Chapter 6 to motivate an architecture for the neural network compensator.

5 THE REFERENCE

The plant and the reference controller run simultaneously and are given identical commanded positions. The reference is a simplified plant model (no friction, nominal dynamic parameters) with a linear control law that responds in a desired manner to control commands. The teaching algorithm adjusts the network weights "on the fly" according to the difference between the reference and plant states. If the deviations between the two responses becomes significant, the system is reset and a new lesson begins. After the teaching is completed, the network weights should converge on values that result in similar performance of the reference and plant.

5.1 NEED FOR THE REFERENCE

The reference is necessary due to the formulation of the forpropagation algorithm. The equations adjust the weights to achieve a gradient descent of some measure. If that measure is the *system error* ***E***, the weights constantly change to decrease ***E***. This results in unbounded weights since a newly commanded position of the cart is perceived as a discontinuous increase in ***E***. With each teaching pass, forpropagation changes the weights to move the cart faster and faster to the commanded value. Eventually the gains of the network compensator become too large and the system reaches instability.

As an alternative approach, forpropagation uses the residual ***R*** which is the difference between the actual plant state and the desired reference state. The reference and the plant receive identical commanded positions and the reference "shows" the plant the appropriate response. Here forpropagation is well-suited to null the residual by adjusting the connection weights. If the cart is moving too fast with respect to the reference, the weights change to slow the cart down; and, if the cart lags the reference, the weights change to speed up the response. The reference always provides the identical response characteristics for the plant to follow. If significant changes are made to the plant dynamics, the weights will again adjust to mimic the reference.

5.2 BACKGROUND ON MODEL REFERENCE

This section discusses a number of issues related to teaching with a reference. Most of the observations result from experimental work with the neural network controller simulation.

5.2.1 Initial Information

Ideally, the initial values of the network weights are as close as possible to the final weights after convergence. In this situation, the controller slightly adjusts its internal representation to satisfy the performance requirements. However, if the initial network weights are significantly different from the target, the controller must go through an intensive and carefully structured learning session.

5.2.2 System Reset

System reset is a necessary action when teaching the network in a "carefully structured learning session." As the deviations between the plant and reference increases, the chance for spurious weight change and instability also increases. Consider the simple example when the cart-pole begins at $X = 0$ and is commanded to a positive position P. Assume the plant lags behind and the reference reaches P first. The reference then overshoots the commanded position and begins moving back to P with a negative $\dot{X}$ and a negative θ_c. Meanwhile the plant has yet to reach P and continues with a positive $\dot{X}$ and a positive θ_c. At that instant the network controller receives ill-advised information on the desired pole position and desired cart velocity. In this situation the system should be reset.

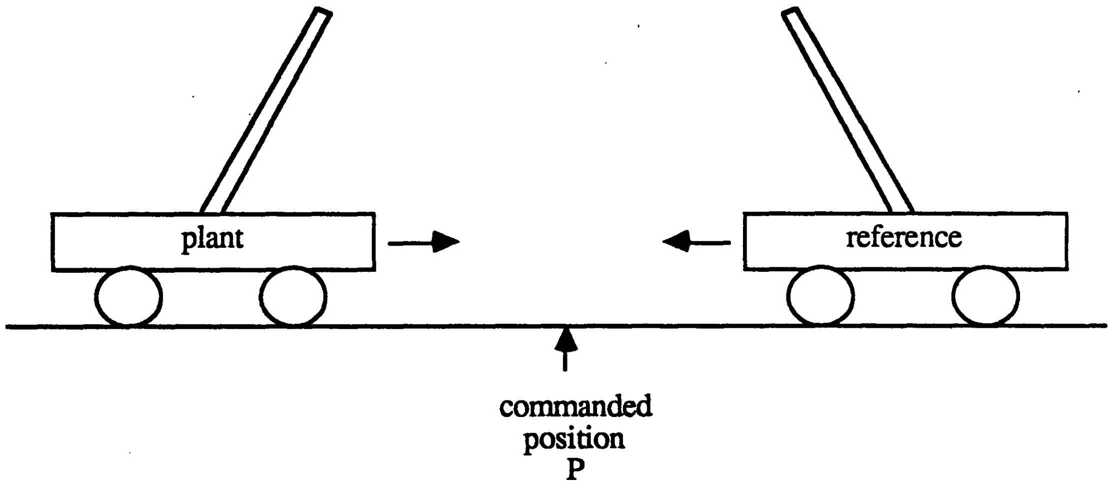


Figure 5.1 - The Lag Problem

The system also maintains a reset limit on the absolute pole position and velocity to insure no teaching past the "point of no return". Here the maximum control force F cannot recover a falling pole. This issue of resetting the system for another lesson is critical and should be addressed with much care.

5.2.3 State Variables to Track

Tracking a state variable means comparing its value with the reference over time and generating an error used in adjusting the weights. In the cart-pole problem there are four possible state variables to track: pole position, pole velocity, cart position, and cart velocity.

Tracking Only Cart Position and Velocity

The final objective is to control the cart position so it seems logical to choose to track only the cart position and its velocity. The only requirement on the pole is that it remains erect regardless of the time history of its position and velocity. Furthermore, with a change in cart mass or pole length, the network controller will undoubtedly require a different response in the pole to achieve the same response in cart position. This may present a problem when the network controller commands an unrealistic and irrecoverable pole position. However, the thresholding effects of the neurons do limit this action.

Tracking All State Variables

This approach is not recommended unless problems arise with keeping the pole upright. Perhaps an acceptable approach would be to leniently track the pole position and velocity to insure balancing.

5.2.4 Weight Update Interval

The weight update interval is a crucial setting in the learning algorithm. One approach updates the weights much faster than the system's highest natural frequency. This offers quick learning and recovery, but is very sensitive to higher order disturbances and noise which often lead to instability. Another approach allows the network controller to perform for a long interval of time without changing the weights. This "mimic and adjust" procedure produces a more stable system impervious to higher order effects, but relies on many lessons to teach the desired weights.

5.2.5 Weight Update Test

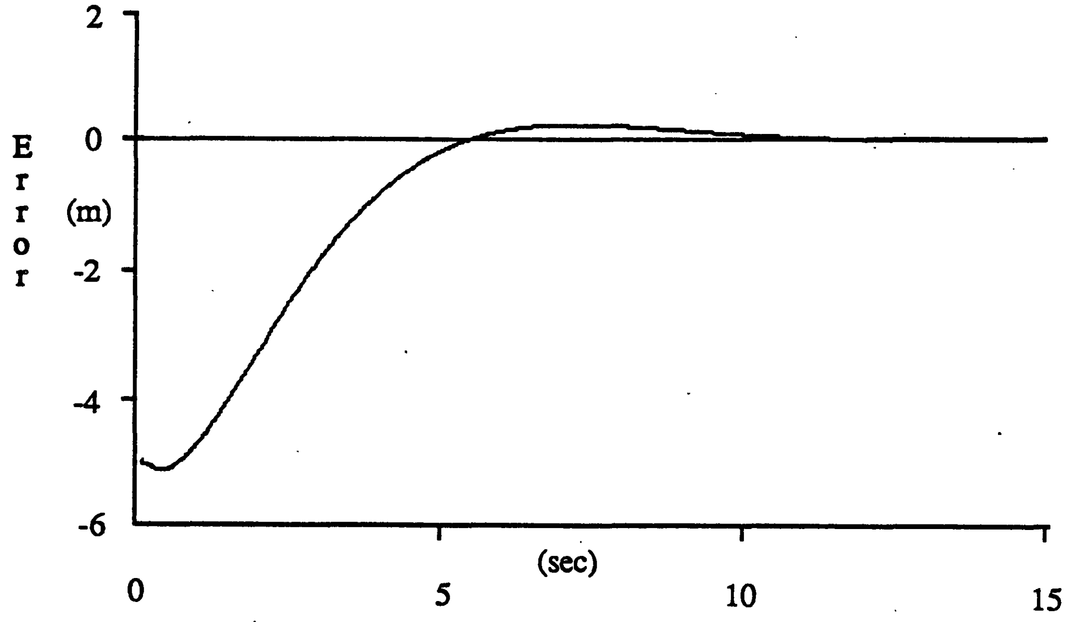
The simulation uses two modes of update. The mandatory or forced update always adjusts the weights at the scheduled time regardless of the performance of the network controller. This approach continuously adjusts the weights even when the plant closely follows the reference. The second method, the optional update, changes the weights only if the adapter error is above a certain threshold.

5.3 DESIGN OF THE REFERENCE

The reference controller design assumes nominal dynamic parameters - a cart mass M of 1 kilogram and a pole length L of 1 meter. The controller commands F in the range of ±10 Newtons and limits θ_c to ±15 degrees. The controller was designed using pole placement to achieve the quickest possible response in X with a ζ of approximately 0.7.

$$F = 69.3\,\theta + 13.2\,\dot{\theta} + 4.9\,\dot{X} + 2.0\,(X - X_c)$$

The reference response, used by the forpropagation algorithm, first reaches an X_C of 5 m in 5.44 sec, overshoots by 4.4%, and settles to within 2% in 9.56 sec.



Graph 5.1 - The Reference Response

The task of this controller is to produce a reference response used in the forpropagation algorithm. The controller operates on an unchanged plant *model* of $M = 1$ kg and $L = 1$ m, producing the identical reference response for all of the experiments. As the plant dynamic parameters change, the network controller attempts to mimic this response from the reference. *Plant* perturbations in pole length or cart mass that take place in the control loop are not reflected in the reference *model*, thus the reference response *always* remains the same.

6 THE NETWORK

This chapter introduces the operation of the neural network as a fixed-gain compensator for the plant in a feedback loop. The network topology is specific to the control problem. In this research, the number of input neurons and output neurons match the number of state variables and control variables, respectively. Furthermore, with the cart-pole problem, the network infrastructure resembles the inner and outer loop organization of the classical compensator.

6.1 THE NETWORK CONTROL LOOP

Figure 6.1 shows the detailed version of the network control loop introduced in the Chapter 3. The positive gain matrices G_1 and G_2 appear along with the localized variable naming convention.

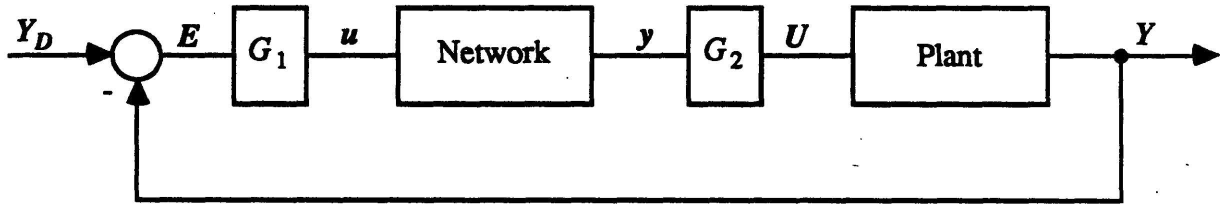


Figure 6.1 - The Network Control Loop

6.1.1 Variable Definitions

Network states are shown as lower case letters, consistent with neural network theory development in Chapter 2. Plant states are shown as upper case letters. Define the plant input control vector U, the plant output state vector Y, and the plant desired state vector Y_D as:

$$U = \begin{pmatrix} U_1 \\ U_2 \\ \cdot \\ \cdot \\ \cdot \\ U_n \end{pmatrix} \qquad Y = \begin{pmatrix} Y_1 \\ Y_2 \\ \cdot \\ \cdot \\ \cdot \\ Y_m \end{pmatrix} \qquad Y_D = \begin{pmatrix} Y_{D1} \\ Y_{D2} \\ \cdot \\ \cdot \\ \cdot \\ Y_{Dm} \end{pmatrix}$$

The network receives the system error $\boldsymbol{E}$ equal to the desired state Y_D minus the actual state $\boldsymbol{Y}$ and passes the controlling inputs $\boldsymbol{U}$ to the plant. The output $\boldsymbol{y}$ of the network is a function of the input $\boldsymbol{u}$, the connection weights, and sometimes the old output states.

6.1.2 The Neural Network

The network box contains the network structure, interconnection weights, and propagation function for the perceptron. Complete propagation allows the network to settle to a steady state for each time step. The network output is a function of only the current inputs and the weights. One-step propagation passes a signal one layer forward for each time step. The output for this type of propagation is a function of the current input, the network weights, and the effects of previous input values held on signal lines within the network. This propagation incorporates memory and is used later in a design of a partial-state feedback controller. Techniques for complete or one-step propagation of linear or non-linear perceptrons appear in Chapter 2.

6.1.3 The Plant

The plant contains the control problem dynamics. It receives a control action and outputs its state variables. A full-state feedback controller uses all independent state variables to control the plant whereas a partial-state feedback controller uses only a subset. The experimental work of this paper uses an inverted pole on a cart as the plant (see Chapter 4).

6.1.4 The Gain Matrices

The gain matrices $G1$ and $G2$ contain positive, non-zero entries along the diagonal. They are used to appropriately scale the variables of the plant and network. The gains in G_1 are chosen to keep the network input bounded by the sigmoid function. A gain on a state variable causes the sigmoid to saturate at a particular value for that variable. This saturation point depends on the desired operation envelope of the plant. The experiments of Chapter 9 used the following four gains in G_1 for the state variables of the cart-pole.

Gains in G_1	
θ	8.0
$\dot{\theta}$	4.0
X	0.3
$\dot{X}$	0.9

The gains in G_2 determine the relationship between the network outputs and the controlling inputs to the plant. For the cart-pole problem the single network output is multiplied by a factor of 10 to produce a control force range of ± 10 Newtons.

6.2 OPERATION OF THE NETWORK CONTROL LOOP

At each time step, the simulation integrates the equations of motion of the plant and updates the control action by propagating the network. At a less frequent rate, the program adjusts the network interconnection weights based on the performance of the controller. Nominally, the equations of motion and control action are updated at 25 Hz while the weights are adjusted at 5 Hz. After 5 time steps of normal fixed-gain operation, the network weights are adjusted. Discussion of the weight change algorithm appears in the Chapter 7.

There are three ordered events for one complete time step of the network control loop. First, the system determines the error vector:

$$E = Y_D - Y$$

Next, it propagates the network, either complete or one-step, using the input vector:

$$u = G_1 E$$

The full-state controllers use complete propagation which allows the network to settle to a steady-state output. Partial-state controllers require a memory of past input values and use one-step propagation. Finally, the system numerically integrates the plant dynamics using the new controlling input vector:

$$U = G_2 y$$

6.3 CONTROLLER ARCHITECTURES

This section establishes the infrastructure of the network used as a full-state compensator for the cart-pole problem. Since all necessary state variables are available to control the plant, the compensator requires no memory and uses complete propagation. Steps are taken to develop controllers using classical methods which in turn motivate the design of analogous network structures. Section 6.3.1 derives a classical controller to balance the pole and proposes a network counterpart. Section 6.3.2 develops a compensator to control the cart position while balancing the pole and again presents an analogous network structure. Since the purpose of this section is to determine the general structure of the network compensator, a simple model of the cart-pole will suffice.

6.3.1 The Full-State Pole Position Controller

The initial task of the cart-pole problem is to balance the pole, or ideally, control the pole position. First, the problem is solved using classical control techniques and then an analogous design of a network is proposed.

<u>Classical Control Technique</u>

Using the simplified state equations of the cart-pole found in Chapter 4, the expression governing θ including the forcing term F is:

$$\ddot{\theta} - \frac{g}{L'}\theta + \frac{F}{L'M} = 0$$

where g is the force of gravity, L' is the effective length of the pole, and M is the mass of the cart. Rearrange and differentiate to get the transfer function in terms of the Laplace operator s.

$$\frac{\dot{\theta}}{F} = \frac{-\frac{s}{L'M}}{s^2 - \frac{g}{L'}}$$

The resulting control loop is:

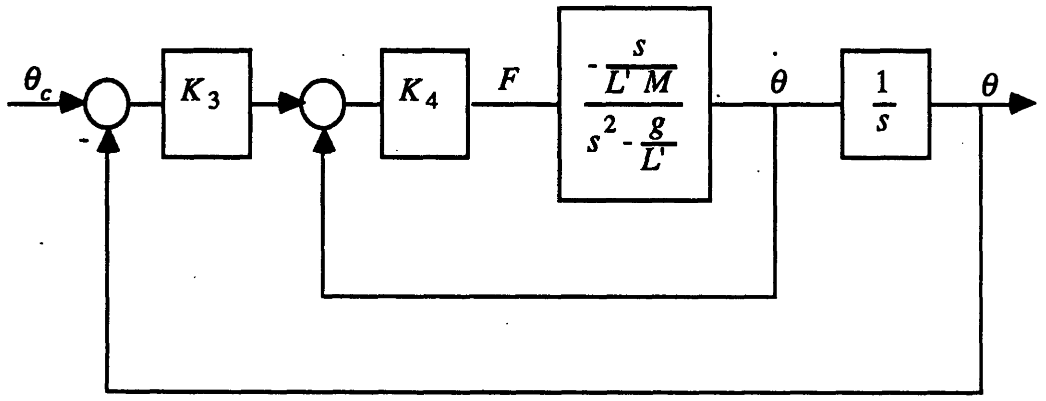


Figure 6.2 - Theta Control Loop

A judicious choice of $K3$ and $K4$ places the two closed loop poles in any desired configuration. From the block diagram in Figure 6.2, assuming θ_c is zero, F can be rewritten as:

$$F = -K_3 K_4 \theta + K_4 \dot{\theta}$$

Combine the above equation with the simplified state equation and factor out θ.

$$\theta \left[s^2 + \frac{K_4}{L' M} s - \left(\frac{g}{L'} + \frac{K_3 K_4}{L' M} \right) \right] = 0$$

Compare this result to the general second order equation:

$$s^2 + 2 \zeta_\theta \omega_{n\theta} s + \omega_{n\theta}^2 = 0$$

where ζ_θ is the damping coefficient and $\omega_{n\theta}$ is the natural frequency. The two equations determining $K3$ and $K4$ for a given choice of ζ_θ and $\omega_{n\theta}$ are:

$$K_3 = -\frac{\omega_{n\theta}^2 + \frac{g}{L'}}{2 \zeta_\theta \omega_{n\theta}}$$

$$K_4 = 2 \zeta_\theta \omega_{n\theta} L' M$$

The preceding derivation of the θ controller using classical techniques does not constrain the control input F. First choose a desired ζ_θ (usually .707) and then in an iterative procedure choose the desired $\omega_{n\theta}$ and check if the resulting gains give a reasonable control action. To realize a large $\omega_{n\theta}$ requires a large control input F. After readjustment of $\omega_{n\theta}$ to fit within the control limits of the system, the final transfer function is:

$$\frac{\theta}{\theta_c} = \frac{\omega_{n\theta}^2 + \frac{g}{L'}}{s^2 + 2 \zeta_\theta \omega_{n\theta} s + \omega_{n\theta}^2}$$

Analogous Network Controller

The equivalent neural network controller requires two gains on the angular position and velocity. The final network structure for the θ controller is simple requiring only two input neurons and one output neuron.

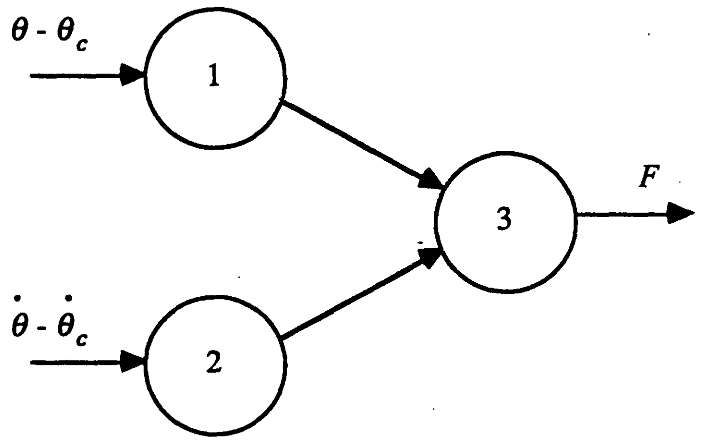


Figure 6.3 - Network Structure of Theta Controller

Assuming $\dot{\theta}_c$ is zero and the neurons operate in their linear region, the corresponding control law as a function of the perceptron weights is:

$$F = w_{23}\dot{\theta} + w_{13}(\theta - \theta_c)$$

6.3.2 The Full-State Cart Position Controller

The next task is to integrate the results of the θ controller into a complete design to control the cart's position while balancing the pole. Again the method involves the classical approach leading to an equivalent network structure.

Classical Control Technique

The design of the control loop for X requires an important assumption. If the natural frequency of the θ loop is sufficiently larger than the natural frequency of the X loop, then there are two approximations. First, the transfer from θ_c to θ in the X loop is unity. With respect to the slow X loop, the θ loop maintains virtually no error. Secondly, the design of the X loop controller ignores the second order term in θ.

$$\cancel{\ddot{\theta}} - \frac{g}{L'}\theta + \frac{\ddot{X}}{L'} = 0$$

This equation, using the first assumption, becomes:

$$\ddot{X} = g\,\theta = g\,\theta_c$$

The block diagram for the X loop is now second order and can be written as:

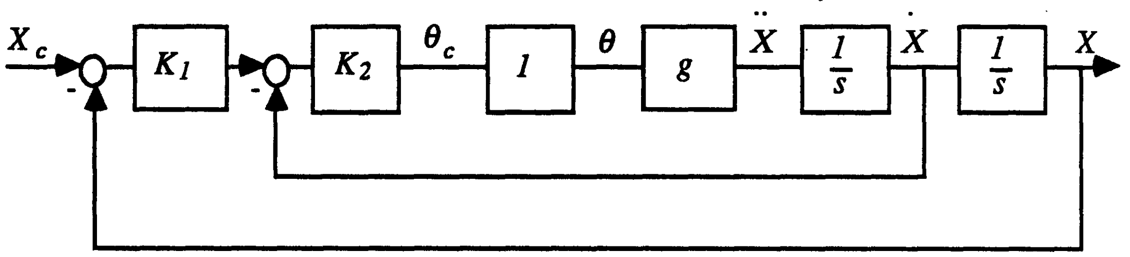


Figure 6.4 - Simplified X Control Loop

This problem with the important assumptions is identical to the θ loop design but with a simpler plant. Wise choices of K_1 and K_2 will place the two closed loop poles in any desired configuration. With X_c treated as zero, the equation governing X is now a function of θ_c.

$$\ddot{X} = g\,\theta_c = -K_1K_2gX - K_2g\dot{X}$$

Rearrange and factor out X:

$$X\left[s^2 + K_2 g s + K_1 K_2 g\right] = 0$$

Again compare this equation to the standard second order system to find the equations for K_1 and K_2 given a ζ_x and ω_{nx}.

$$K_1 = \frac{\omega_{nx}}{2\,\zeta_x}$$

$$K_2 = \frac{2\,\zeta_x\,\omega_{nx}}{g}$$

The X control law does not explicitly limit the internally commanded θ. Again, choose an ω_{nx} and determine if it is attainable. A large ω_{nx} in the X loop requires a large θ_c which the controller may not be able to handle. It is important to define reasonable operating regions for the controller or include a limiter in the loop.

After choosing the two gains for the inner loop θ controller and the two gains for the outer loop X controller, the final control law is:

$$F = K_4\dot{\theta} - K_3K_4\theta - K_2K_3K_4\dot{X} - K_1K_2K_3K_4(X - X_c)$$

also in block diagram form:

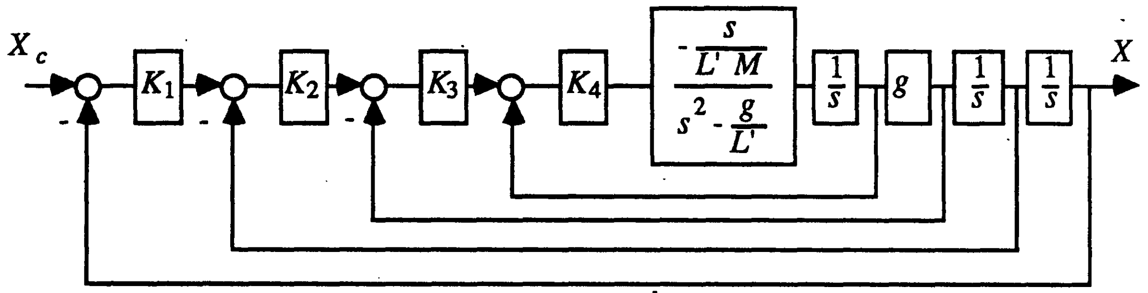


Figure 6.5 - Full X Control Loop

and as a transfer function:

$$\frac{X}{X_c} = \frac{\omega_{nx}^2\left(\omega_{n\theta}^2 + \frac{g}{L'}\right)}{s^4 + 2\zeta_\theta\omega_{n\theta}s^3 + \omega_{n\theta}^2 s^2 + \left[2\zeta_x\omega_{nx}\left(\omega_{n\theta}^2 + \frac{g}{L'}\right)\right]s + \omega_{nx}^2\left(\omega_{n\theta}^2 + \frac{g}{L'}\right)}$$

Analogous Network Controller

The network controller adopts the same assumptions as the previous section. The inner θ controller dynamics do not effect the outer X controller. As in the classical approach, the X loop commands a value for θ. This commanded θ is held internally in the network configuration. Layering the two loops gives the final structure.

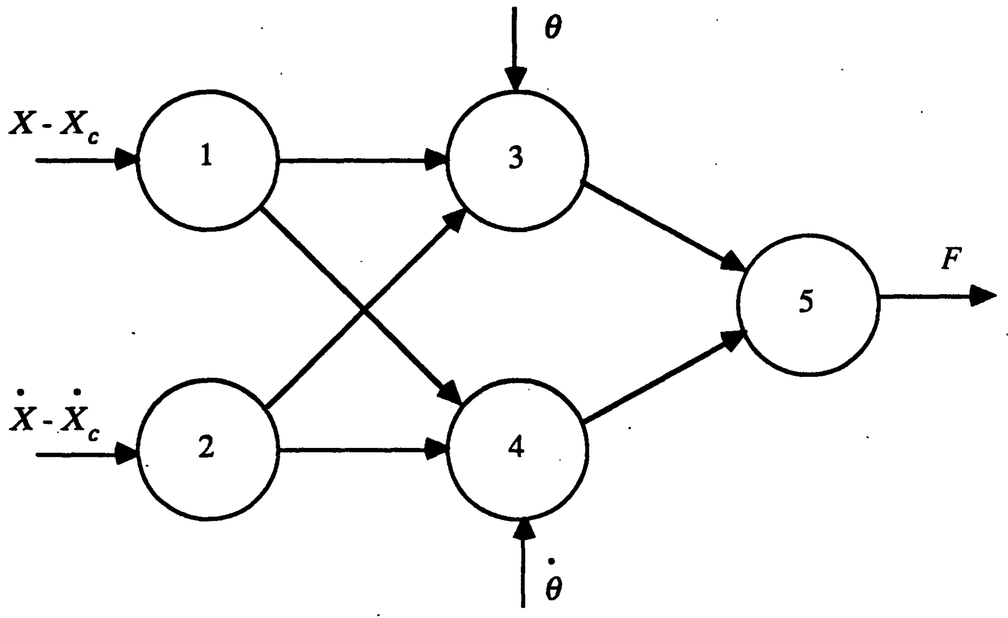


Figure 6.6 - Network Structure of X controller

Each neuron is associated with a state variable of the plant. The first layer of neurons receives the difference between the cart's desired and actual position and velocity. In this research, the value for $\dot{X}_c$ is always set to zero. The next layer receives an external signal of the actual angular position and velocity of the pole. The network internally generates the *negative* of the desired signals for pole position and velocity and feeds these values into the second layer of neurons. As a result, nodes 3 and 4 receive the difference between the desired and actual pole position and velocity. The control law as a function of the network weights with the neurons operating in the linear region is:

$$\begin{aligned} F &= (w_{35} w_{13} + w_{45} w_{14})(X - X_c) \\ &\quad + (w_{35} w_{23} + w_{45} w_{24})\dot{X} + w_{35}\theta + w_{45}\dot{\theta} \end{aligned}$$

Since there are six weights to determine four control gains, there is not a unique solution of weights for a particular control law.

7 THE ADAPTER

The adapter implements the forpropagation algorithm to adjust the network weights according to a measure of error. This error or *residual* is the difference between the measured state variables of the plant and their corresponding desired values. Section 7.1 presents the architecture of the forpropagation control loop. Section 7.2 introduces a summary of the operation of the adapter with emphasis on the three steps necessary to apply forpropagation. Finally, Section 7.3 shows that the forpropagation weight change algorithm performs a gradient descent.

7.1 THE FORPROPAGATION CONTROL LOOP

The forpropagation control loop appears in Figure 7.1. The network input u and output y are lower case whereas the plant input U and output Y are upper case. G_1 and G_2, discussed in Section 6.1.4, are the gain matrices that scale the variables of the plant and network.

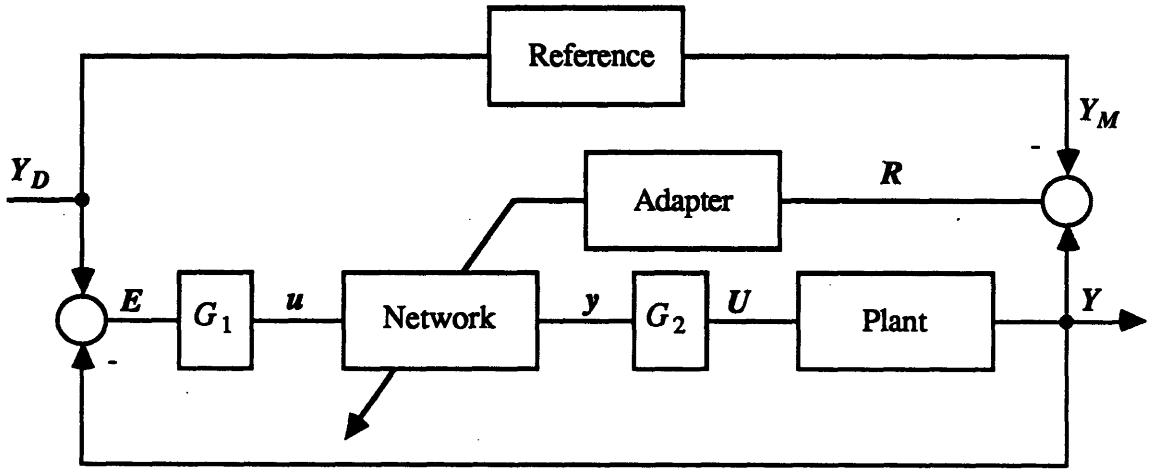


Figure 7.1 - Forpropagation Control Loop

The model reference receives the commanded state of the plant Y_D and outputs the desired plant response $\boldsymbol{Y_M}$. The adapter uses $\boldsymbol{R}$, the difference between the actual plant output $\boldsymbol{Y}$ and $\boldsymbol{Y_M}$, to adjust the network weights. This residual $\boldsymbol{R}$ has the same dimensions as the network input $\boldsymbol{u}$.

$$\boldsymbol{R} = \begin{pmatrix} Y_1 - Y_{M_1} \\ Y_2 - Y_{M_2} \\ \cdot \\ \cdot \\ \cdot \\ Y_m - Y_{M_m} \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathrm{R_m} \end{pmatrix}$$

The adaptive error ε is a scalar and is defined as:

$$\varepsilon = \frac{1}{2}\boldsymbol{R}^T\boldsymbol{R}$$

As ε decreases, the actual plant response closely resembles the reference response. The weights are continually updated until the adaptive error ε is below a particular threshold.

7.2 OPERATION OF THE ADAPTER

The adapter changes the individual neuron connection weights while the network operates as a compensator in the control loop. The adapter receives the residual $\boldsymbol{R}$, calculates the adapter error ε, and determines whether the weights should change. If ε is above a preset threshold, the weights are adjusted according to the forpropagation algorithm.

Forpropagation, derived and discussed in Section 7.3, performs a gradient descent in the adaptive error ε with respect to network weight changes. A summary of the weight adjustment algorithm proceeds in three steps. Step 1 calculates the *propagation terms p* beginning at the output using the two equations for the output and hidden neurons.

hidden neurons $$p_{ik} = f'(net_i) \sum_j w_{ij} p_{jk}$$

output neurons $$p_{kk} = f'(net_k)$$

Step 2 generates the ***neuron errors*** δ for both the hidden and output neurons using the residual vector $\boldsymbol{R}$.

hidden neurons $$\delta_j^{(h)} = -R_h \sum_k^{outputs} \pi_{hk} p_{jk}$$

output neurons $$\delta_k^{(h)} = -R_h \pi_{hk} p_{kk}$$

where π_{hk} is defined as the ***dynamic sign term.*** Finally, Step 3 adjusts the weights using the equation:

$$\Delta w_{ij}(p+1) = \eta \, o_i \sum_h^{residuals} \delta_j^{(h)} + \mu \Delta w_{ij}(p)$$

where η is the learning coefficient, μ the momentum term, $\Delta w_{ij}(p)$ is the weight change made during the previous lesson, and $\Delta w_{ij}(p+1)$ is the weight change for the current lesson.

After making the weight changes using these three steps, the newly taught network is used in the control loop. If the adjustments do not reduce ε below a preset threshold, the process is repeated.

7.3 FORPROPAGATION ALGORITHM DERIVATION

The following is a detailed derivation of forpropagation. Many of the techniques of gradient descent parallel the work of Rumelhart and McClelland [26]. The specific subscript h is used in conjunction with the residual input R. Output neurons are denoted by subscript k, while subscripts i and j refer to any neuron. The weight that connects neuron i

to a downstream neuron j is denoted by w_{ij}. The output of neuron i is o_i and the sum of its weighted inputs is net_i.

7.3.1 Gradient Descent

A gradient descent in the adapter error ε with respect to a weight change can now be developed by showing that:

$$-\frac{\partial \varepsilon}{\partial w_{ij}} \ \alpha \ \textit{weight change equation}$$

This relationship insures that a small change in the weight w_{ij} results in a small decrease in the overall adapter error ε. Using the chain rule, expand the left side into partial fractions.

$$-\frac{\partial \varepsilon}{\partial w_{ij}} = -\frac{\partial \varepsilon}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}} = \delta_j^{(h)}\frac{\partial net_j}{\partial w_{ij}}$$

The first term $\delta_j^{(h)}$, defined as the *error* of neuron j for the residual R_h, is evaluated in Section 7.3.2. Express the second term using the fact that the network input net_j is the sum of the upstream neuron outputs o_g multiplied by the connection weights w_{gj}.

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_g w_{gj} o_g\right)$$

The partial derivative is nonzero for only one term in the summation, so the equation can be simplified and evaluated.

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial\left(w_{ij} o_i\right)}{\partial w_{ij}} = o_i$$

The weight change equation is proportional to the output o_i of neuron i multiplied by the error $\delta_j^{(h)}$ of neuron j.

$$-\frac{\partial \varepsilon}{\partial w_{ij}} = \delta_j^{(h)} o_i$$

7.3.2 The Neuron Error

The neuron error is defined as:

$$\delta_j^{(h)} = -\frac{\partial \varepsilon}{\partial net_j}$$

This expression can be expanded using partial fractions.

$$\delta_j^{(h)} = -\frac{\partial \varepsilon}{\partial net_j} = -\frac{\partial \varepsilon}{\partial R_h}\frac{\partial R_h}{\partial net_j}$$

The error states how forpropagation must change the neuron's internal state, or weighted sum input net_j, to decrease the adapter error ε. A neuron contributes an error term to ε for each residual R_h. To determine the error of neuron j for a particular residual R_h, sum over all the network outputs y_k the weight change effects.

$$\delta_j^{(h)} = -\frac{\partial \varepsilon}{\partial R_h}\sum_k^{outputs}\frac{\partial R_h}{\partial y_k}\frac{\partial y_k}{\partial net_j}$$

This expression states that forpropagation must adjust a neuron's upstream weights in proportion to the effect of those weights on output control commands. Evaluate the term relating the change in ε with respect to a change in a residual R_h using the definition of the adapter error.

$$\frac{\partial \varepsilon}{\partial R_h} = \frac{\partial\left(\frac{1}{2}R^T R\right)}{\partial R_h} = \frac{\partial\left(\frac{1}{2}R_1^2 + \frac{1}{2}R_2^2 + \ldots + \frac{1}{2}R_m^2\right)}{\partial R_h} = R_h$$

Now the equation can be written as:

$$\delta_j^{(h)} = -R_h\sum_k^{outputs}\frac{\partial R_h}{\partial y_k}\frac{\partial y_k}{\partial net_j}$$

The next two sections discuss the terms after the summation sign.

The Dynamic Sign

The inputs and outputs of a perceptron controller are tied to the plant. It is necessary to know at least a crude approximation of how a network output (actuator command) effects a residual input. The π_{hk} term "completes the loop" and relates the residual R_h to an output y_k. This research only uses the *sign information* of the relationship between the plant input and output.

First, define π_{hk} as the change in a residual R_h with respect to a change in the network output y_k.

$$\pi_{hk} = \left.\frac{\partial R_h}{\partial y_k}\right)_*$$

The subscript * means "while operating about a nominal condition". This important restriction insures that π_{hk} maintains the true dynamic information of the plant. In the example of the cart-pole, if the pole falls past the "point of no return", any allowable control force on the cart will not move the pole back to the balanced position. This instance leads to an incorrect calculation of the π_{hk} term that relates the pole position to the control force. Now assume π_{hk} is computed when the pole is close to the vertical. Here a positive control force F results in a decrease in θ and vice-versa for a negative F. This π_{hk} determined about a nominal operating condition contains the correct dynamic information of the cart-pole plant.

Rewrite the equation for π_{hk} using the definition of the residual $R_h = Y_h - Y_{Mh}$ and the relationship $U_k = c\, y_k$ where c is a positive constant in the gain matrix G_2 (see Figure 7.1).

$$\pi_{hk} = \left.\frac{\partial R_h}{\partial y_k}\right)_* = \left.\frac{\partial\left(Y_h - Y_{Mh}\right)}{\partial\left(\frac{U_k}{c}\right)}\right)_*$$

Since the model reference response Y_{Mh} is unchanged by the input to the plant U_k, the equation becomes:

$$\pi_{hk} = c \left.\frac{\partial Y_h}{\partial U_k}\right)_*$$

In its simplest form π_{hk} contains the sign information of the impulse response of the plant state Y_h for the controlling input to the plant U_k.

$$\pi_{hk} = sgn\left[\left.\frac{\partial Y_h}{\partial U_k}\right)_*\right]$$

During the experiments of this research, π_{hk} does not change but could be updated periodically using plant perturbations in real-time. Each update would represent a local linearization of the plant controls to the state variables.

<u>The Propagation Term</u>

The final term in the equation for the neuron error $\delta_j^{(h)}$ is $\frac{\partial y_k}{\partial net_j}$ which relates the change in the output of neuron k due to a change in the input of an upstream neuron j. Since the network output y_k is the external state of the output neuron k, this expression can be generalized for any neuron i and any downstream neuron j.

$$p_{ij} = \frac{\partial o_j}{\partial net_i}$$

This is the definition of the linearized sensitivity term or *propagation term* p_{ij}. Essentially, p_{ij} represents the effects of connection weights and saturation functions along the path between neurons i and j. A simple case will illustrate the idea. Figure 7.2 represents a signal path from neuron 1 to neuron 3. The interconnection weights appear in the boxes.

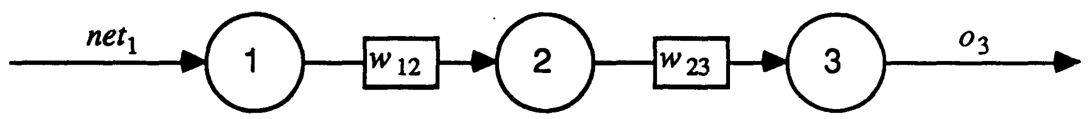


Figure 7.2 - Simple Feedforward Path

The final expression for p_{13} contains interconnection weights and the effects of neuron saturation. Expand p_{13} using the chain rule.

$$p_{13} = \frac{\partial o_3}{\partial net_1} = \frac{\partial o_3}{\partial net_3}\frac{\partial net_3}{\partial o_2}\frac{\partial o_2}{\partial net_2}\frac{\partial net_2}{\partial o_1}\frac{\partial o_1}{\partial net_1}$$

Each factor of the form:

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j} f(net_j) = f'(net_j)$$

contributes the effect of neuron saturation which is always positive. The factor $f'(net_j)$ is the slope of the sigmoid function at the point net_j which is very small for extremely negative or positive inputs and reaches a maximum for an input of zero (see Figure 2.3). Each factor of the form:

$$\frac{\partial net_j}{\partial o_i} = \frac{\partial}{\partial o_i}\left(\sum_g w_{gj} o_g\right) = \frac{\partial \left(w_{ij} o_i\right)}{\partial o_i} = w_{ij}$$

contributes the interconnection weight w_{ij} to the product. The final expression for p_{13} is:

$$p_{13} = w_{12} w_{23} f'(net_1)\, f'(net_2)\, f'(net_3)$$

This simple exercise suggests a recursive procedure to find all of the propagation terms for a network. First, generate the propagation term for each output neuron.

$$p_{kk} = f'(net_k)$$

Then working back from the output layer, recursively compute the propagation terms from the hidden and input neurons to the output neurons.

$$p_{ik} = f'(net_i) \sum_j w_{ij} p_{jk}$$

Final Expression for the Neuron Error

Now the error of a hidden neuron $\delta_j^{(h)}$ for the residual R_h can be written in a simpler form.

$$\delta_j^{(h)} = -R_h \sum_{k}^{outputs} \pi_{hk} p_{jk} \qquad \textit{(hidden neuron)}$$

where π_{hk} is the dynamic sign term between the residual h and the output neuron k, and p_{jk} is the propagation term from the input of neuron j to the output of neuron k. The error for an output neuron is simpler since it only registers the effect of its own output on the residual h.

$$\delta_k^{(h)} = -R_h \pi_{hk} p_{kk} \qquad \textit{(output neuron)}$$

As an example consider a network with a single output neuron k. For each residual h, the neuron error for both the hidden neurons and the one output neuron simplifies to:

$$\begin{array}{ll} \textit{hidden} & \delta_j^{(h)} = -R_h \pi_{hk} p_{jk} \\ \textit{output} & \delta_k^{(h)} = -R_h \pi_{hk} p_{kk} \end{array}$$

7.3.3 Weight Change Equation

Begin with the original definition of the weight change equation.

$$\textit{weight change equation} \ \alpha \ -\frac{\partial \varepsilon}{\partial w_{ij}} = -\frac{\partial \varepsilon}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Using the definitions developed in the previous sections, sum over the weight changes for each residual input to the network. This involves summing the errors for all the residuals h at neuron j.

$$-\frac{\partial \varepsilon}{\partial w_{ij}} = -\frac{\partial \varepsilon}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}} = o_i \sum_{h}^{residuals} \delta_j^{(h)}$$

This equation states that an interconnection weight changes in proportion to the activation along its line, represented as o_i, multiplied by the sum of the errors at the downstream neuron *j*.

The final equation for the weight change $\Delta w_{ij}(p+1)$ incorporates a teaching coefficient η, a momentum coefficient μ, and the value of the weight change during the previous application of forpropagation $\Delta w_{ij}(p)$. The variable *p* indexes the discrete events that change the network weights. The teaching coefficient is analogous to the step size of the weight change. The weights change in direct proportion to η. The momentum coefficient μ multiplied by the previous weight change provides a smoothing effect to the weight adjustments over time. The final equation for adjusting the weights is:

$$\Delta w_{ij}(p+1) = \eta\ o_i \sum_{h}^{residuals} \delta_j^{(h)} + \mu \Delta w_{ij}(p)$$

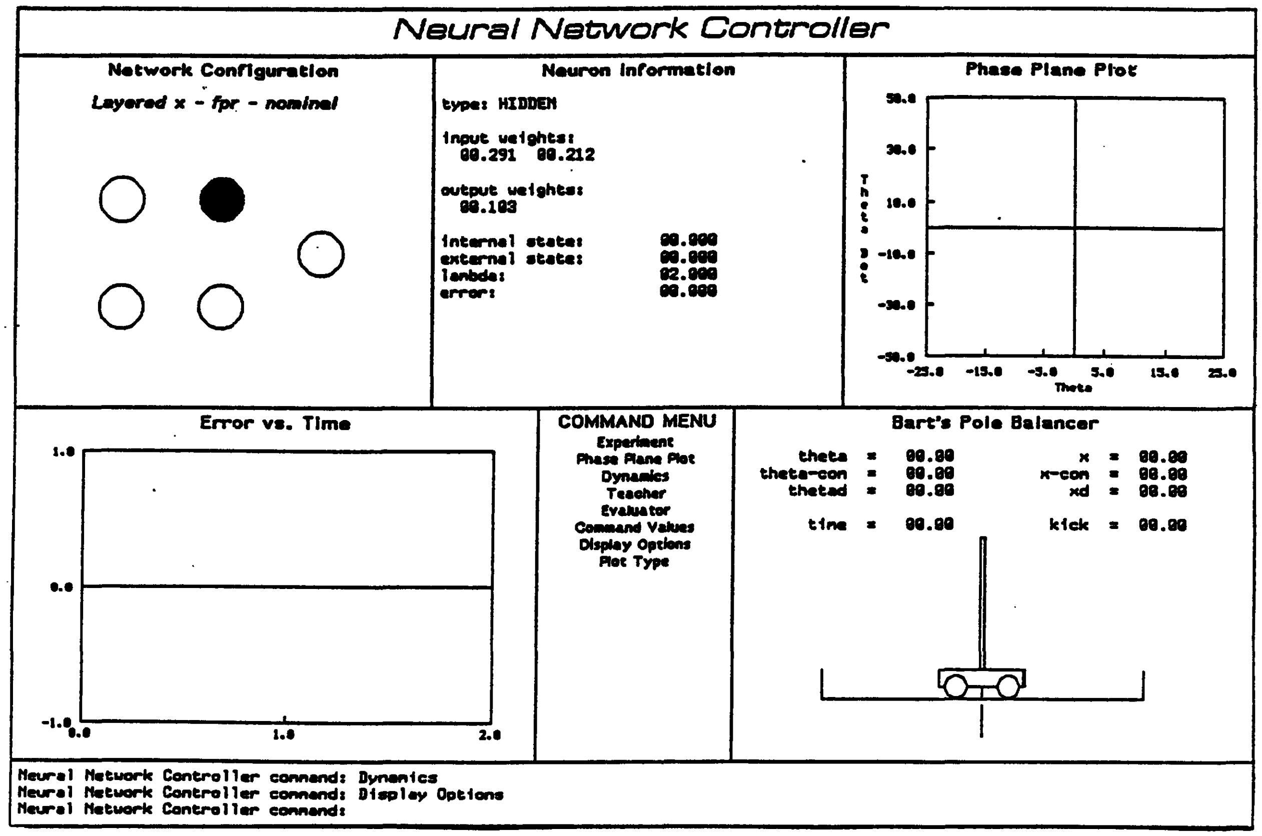
This concludes the derivation and confirms that forpropagation is a gradient descent teaching algorithm.

8 IMPLEMENTATION

The Neural Network Controller Simulator, written in Common Lisp, runs on the Symbolics 3600 computer. The program is organized into modules which perform specific tasks. This chapter reviews the operation of the Neural-Network-Controller, Pole-Exec, Forprop, Pole-Dynamics, and Big-Graph modules. Source listings of the Pole-Exec, Forprop, and Pole-Dynamics programs appear in the Appendices.

8.1 NEURAL-NETWORK-CONTROLLER

This Neural-Network-Controller directs the high level control of the simulator. In addition to organizing the graphical displays, it also contains the system menu.



```
type: HIDDEN

input weights:
  00.291  00.212

output weights:
  00.103

internal state:        00.000
external state:        00.000
lambda:                02.000
error:                 00.000
```

```
    theta  =  00.00            x  =  00.00
theta-con  =  00.00        x-con  =  00.00
   thetad  =  00.00           xd  =  00.00

     time  =  00.00         kick  =  00.00
```

```
Neural Network Controller command: Dynamics
Neural Network Controller command: Display Options
Neural Network Controller command:
```

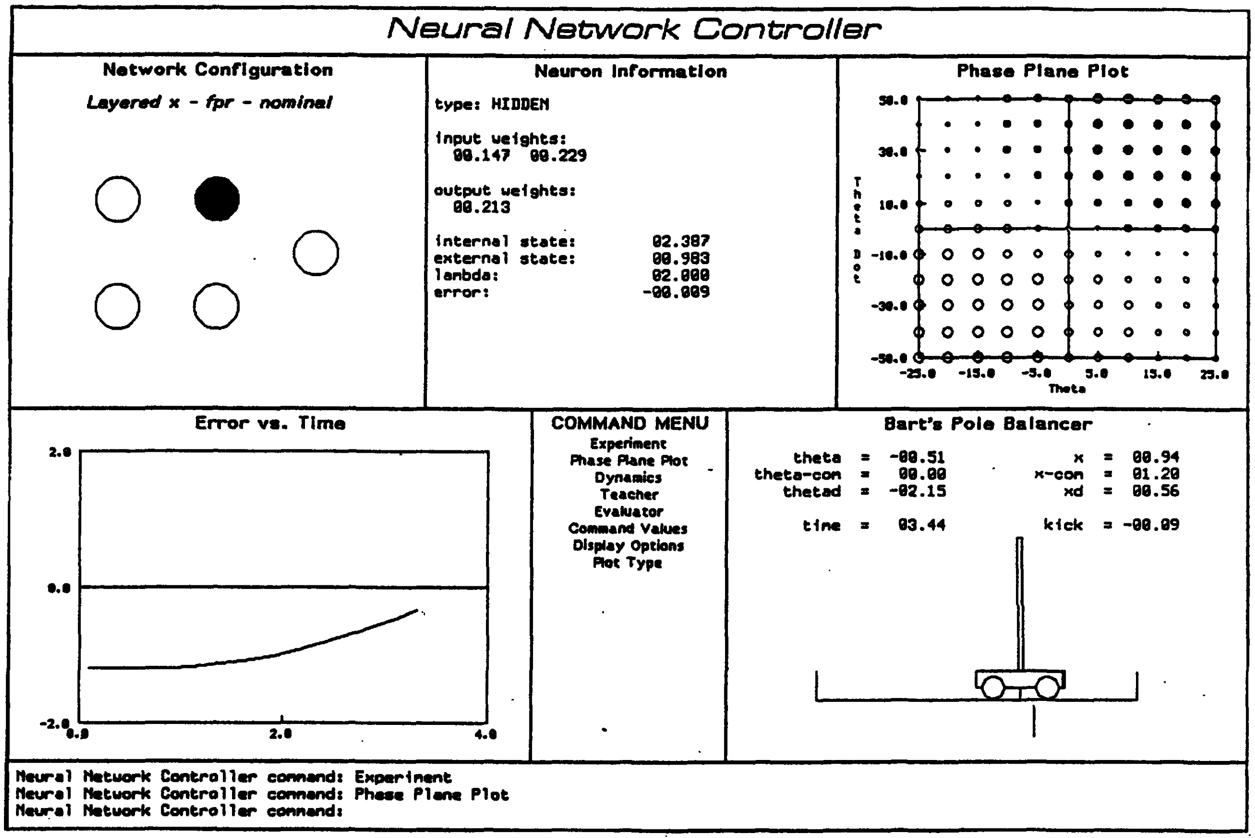
Figure 8.1 - Initial Display of the Simulator

Figure 8.1 shows the display of the simulator before running an experiment called *Nominal*. The window is arranged into six panes:

- Network Configuration
- Neuron Information
- Phase-Plane Plot
- General Plot
- System Menu
- Cart-Pole

The Network Configuration pane displays the experiment name and the infrastructure of the network. Information on the connection weights and state of the darkened neuron appears in the adjacent Neuron Information pane. The user can click on a neuron and display the statistics or change the neuron parameters while the simulation is running. The Phase Plane Plot pane shows the current control decision for the cart-pole in θ - $\dot{\theta}$ space. The General Plot pane displays the user's choice of single or multiple value plots of the cart-pole state variables. The System Menu contains all of the high-level simulator commands that allow the user to adjust on-line any of the parameters related to the forpropagation algorithm, the cart-pole simulator, and the display. The Cart-Pole pane displays an animated picture of the plant. The vertical line below the track indicates the commanded position of the cart. The user can command a new desired cart position "on the fly" by dragging the line with the mouse.

Figure 8.2 now shows a snapshot of the experiment in progress. Dark circles in the Phase Plane Plot represent a positive control force F and light circles a negative F. The magnitude of F is proportional to the radius of the circle. The General Plot Pane indicates the cart position error is approaching the zero horizontal line. The Cart Pane displays related dynamical information and shows the cart moving towards the commanded position.



```
type: HIDDEN

input weights:
  00.147  00.229

output weights:
  00.213

internal state:          02.387
external state:          00.983
lanbda:                  02.000
error:                  -00.009
```

```
    theta  =  -00.51              x  =  00.94
theta-con  =   00.00          x-con  =  01.20
   thetad  =  -02.15             xd  =  00.56

     tine  =   03.44           kick  = -00.09
```

```
Neural Network Controller connand: Experinent
Neural Network Controller connand: Phase Plane Plot
Neural Network Controller connand:
```

Figure 8.2 - Snapshot of the *Nominal* Experiment

8.2 POLE-EXEC

The Pole-Exec runs the main execution loop of the simulator. For each loop, the program performs the following operations:

- check if the pole has fallen
- integrate the plant's equations of motions
- teach the network
- propagate the network

Pole-Exec also allows the user to create, stop, start, and reset a number of user-defined experiments.

8.3 FORPROP

Forprop contains the software to create, propagate, and teach networks. The program builds a network given the following characteristics:

- the number and type of neurons in each layer
- the connections between each layer or neuron
- the initialization of the interconnection weights
- the type of propagation function

All experiments for this research use neurons whose internal function sums the weighted inputs and whose output function is sigmoidal. One common sigmoid function used by Rumelhart and McClelland ranges from 0 to 1:

$$f(x, \lambda) = \frac{1}{1 + e^{-\lambda x}}$$

but does not allow explicit representation of positive and negative values. The shifted sigmoid function used for this research ranges from -1 to 1:

$$f(x, \lambda) = 2\left(\frac{1}{1 + e^{-\lambda x}} - 0.5\right)$$

where the constant λ determines the steepness of the sigmoid.

The neuron interconnections are directed, weighted signal lines between two neurons. They are usually specified through simple layer-to-layer connections where each neuron in an upstream layer feeds a signal to each neuron in a downstream layer. The code also handles neuron-to-neuron and neuron-to-layer connections. Any signals of neurons from downstream layers to neurons of upstream layers is forbidden in the perceptron structure. The interconnection weights can be initialized over a uniform distribution range or they can be set to specific values.

The software supports both complete and one-step propagation. A network is propagated with external inputs fed into any of the neurons. These network inputs are scaled to take advantage of the limiting characteristics of the neurons. The propagation routine receives a list of values to be input into specific neurons, propagates the neurons with those inputs, and returns the external state of the output neurons.

The Forprop module also contains all the teacher routines necessary to adjust the network weights using forpropagation. First, the program calculates the propagation terms and the neuron errors given the current state of the network weights and plant. Then the weights are adjusted using the equations derived in Chapter 7. The cart-pole simulation runs at 25 Hz and the teacher adjusts the weights at 5 Hz, or every fifth time step. For all experiments, the learning coefficient η is set to 0.01 and the momentum coefficient μ is set to 0.9.

8.4 POLE-DYNAMICS

Pole-Dynamics contains all of the cart-pole simulation software. The main purpose of this program is to update the equations for cart acceleration $\ddot{X}$ and the pole acceleration $\ddot{\theta}$.

$$\ddot{X}(t) = \frac{F(t) + m\,L\left(\dot{\theta}^{2}(t)\,sin\theta(t) - \ddot{\theta}(t)\,cos\theta(t)\right) - \mu_c sgn\left(\dot{X}(t)\right)}{M + m}$$

$$\ddot{\theta}(t) = \frac{g\sin\theta(t) + \cos\theta(t)\left(\frac{-F(t) - mL\dot{\theta}^{2}(t)\sin\theta(t) + \mu_c \operatorname{sgn}\left(\dot{X}(t)\right)}{M+m}\right) - \frac{\mu_p \dot{\theta}(t)}{mL}}{L\left(\frac{4}{3} - \frac{m\cos^3\theta(t)}{M+m}\right)}$$

The following simple integration filters compute the velocities and positions of the cart and pole:

$$\dot{h}(t+1) = \left(\frac{\ddot{h}(t+1) + \ddot{h}(t)}{2.0}\right)\Delta t + \dot{h}(t)$$

$$h(t+1) = \left(\frac{\dot{h}(t+1) + \dot{h}(t)}{2.0}\right)\Delta t + h(t)$$

Pole-Dynamics uses the same dynamic update equations to simulate the classical controller and the model reference controller. This module also allows for uniform distribution noise to be added to any state variable.

8.5 BIG-GRAPH

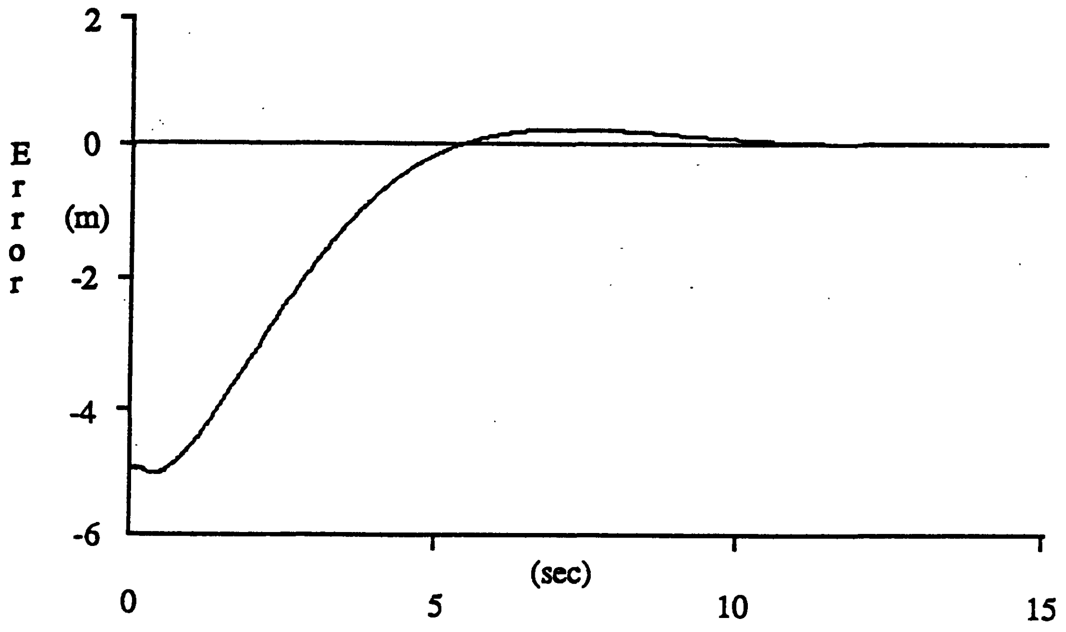
Big-Graph allows the user to plot any state variable versus time. The routine supports dual graphs which will be used extensively in Chapter 9 to superimpose the reference and network controller responses.

9 EXPERIMENTAL RESULTS

This chapter compares the neural network controller to the fixed-gain classical controller. The initial results show the network controller converges on the reference response when the plant dynamic parameters are identical to the model reference parameters. Furthermore, the network controller, while still being taught by the nominal reference response, adapts quickly to significant changes in cart mass and pole length. For the same parameter perturbations, the classical controller fails to balance the pole. Section 9.1 discusses the limitations of the classical controller while Section 9.2 presents experiments showing the adaptability and performance of the neural network controller.

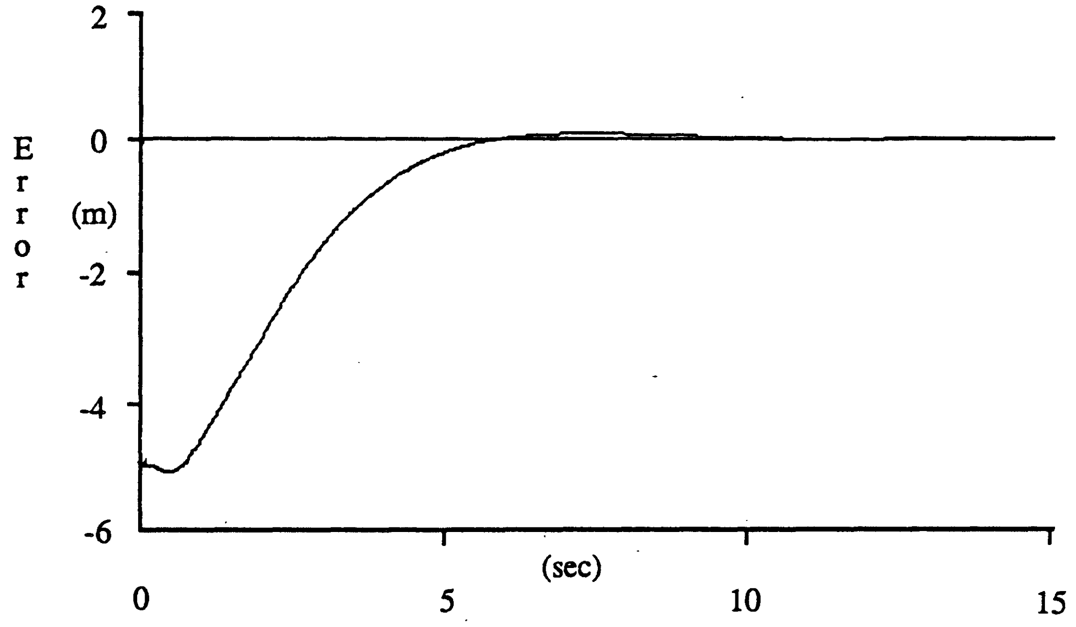
9.1 THE CLASSICAL CONTROLLER

With a commanded cart position X_c of 5 m, a cart mass M of 1 kg, and a pole length L of 1 m, the classical controller gives the following response in cart position X.



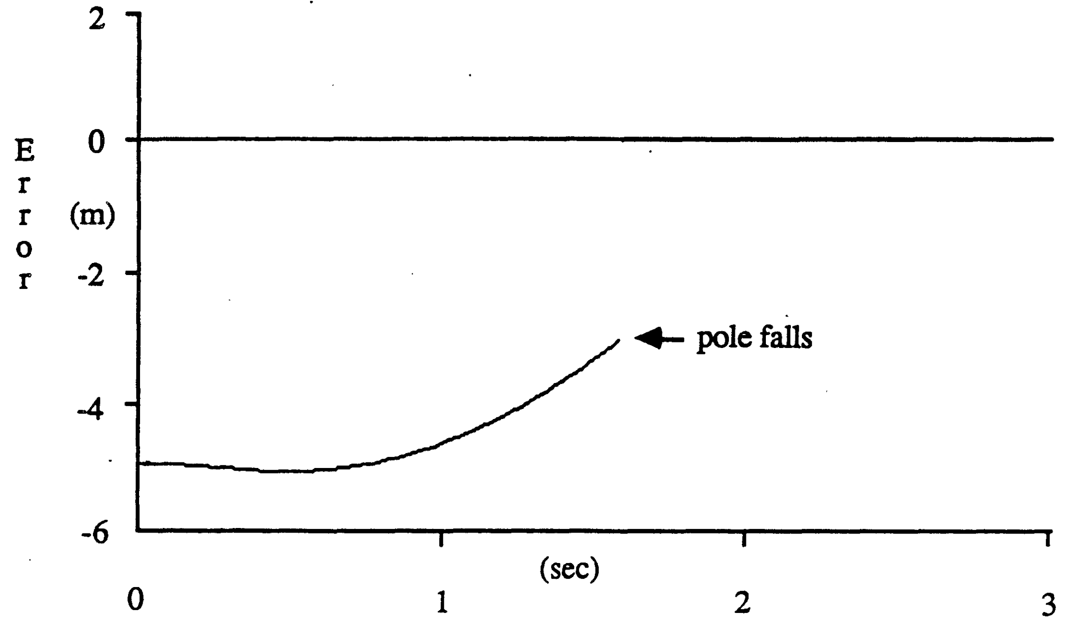
Graph 9.1 - *Classical* - Response in X (M = 1 kg, L = 1 m)

The classical controller has fixed gains so any significant parameter changes result in inadequate performance and possible failure to balance the pole. First, consider increasing the mass of the cart by 100% to 2 kg.



Graph 9.2 - *Classical* -Response in X ($M = 2$ kg, $L = 1$ m)

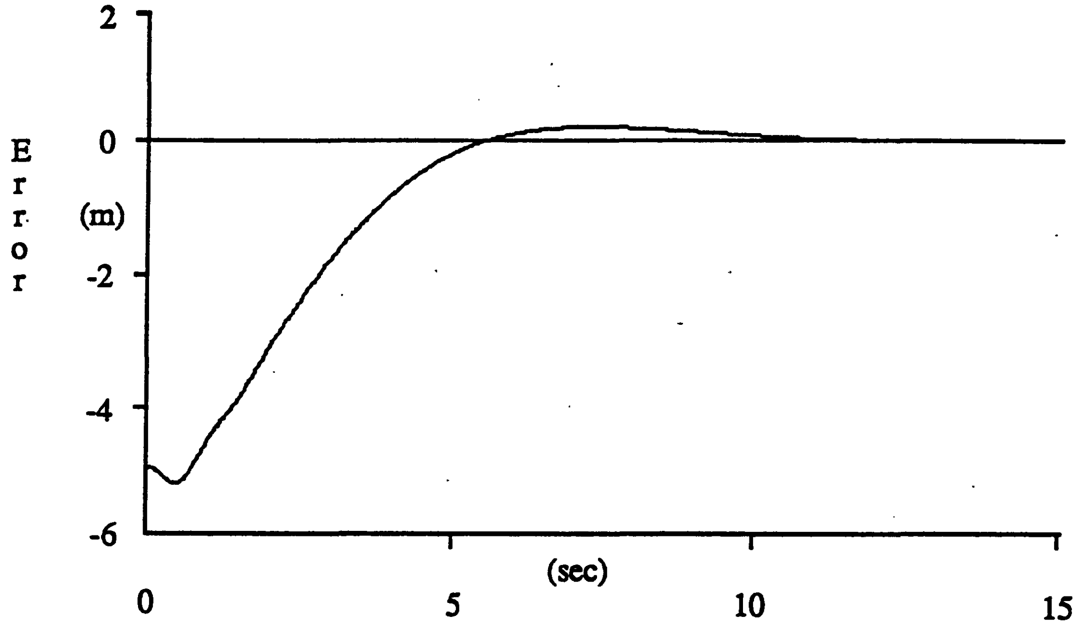
The classical controller still performs well. Now increase M to 3 kg.



Graph 9.3 - *Classical* - Response in X ($M = 3$ kg, $L = 1$ m)

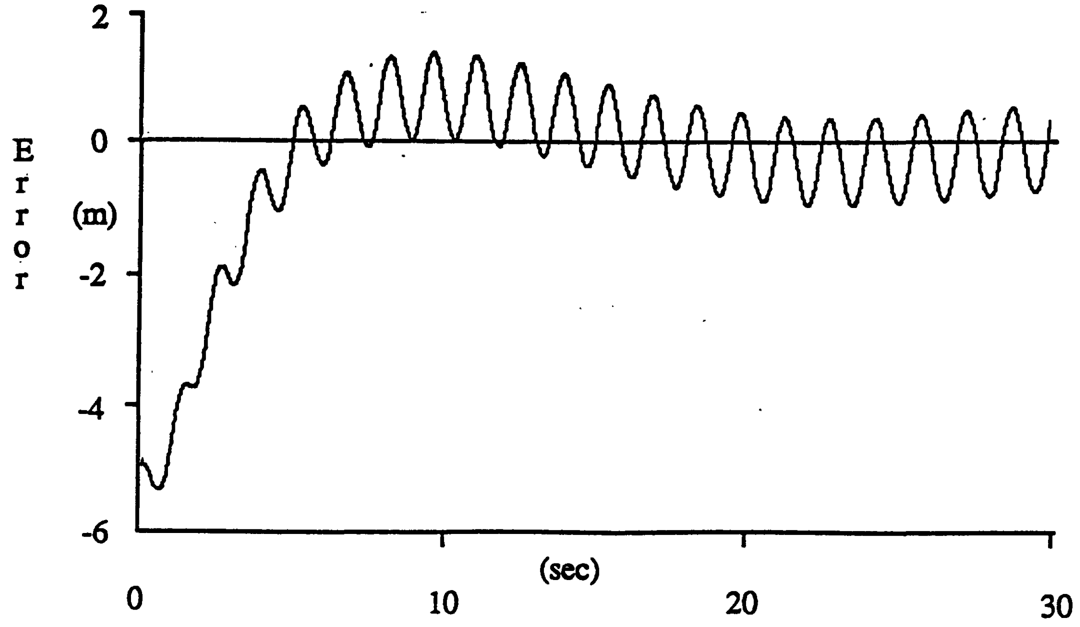
The added mass of the cart causes the classical controller to fail. At 1.6 sec, the pole is already at 57.3 degrees, an irrecoverable situation given the control force saturates at ±10 Newtons.

Now consider increases in the pole length L while keeping M at 1 kg.



Graph 9.4 - *Classical* - Response in X (M = 1 kg, L = 2 m)

The classical controller handles L = 2 m. Next, increase the pole length to 3 m.

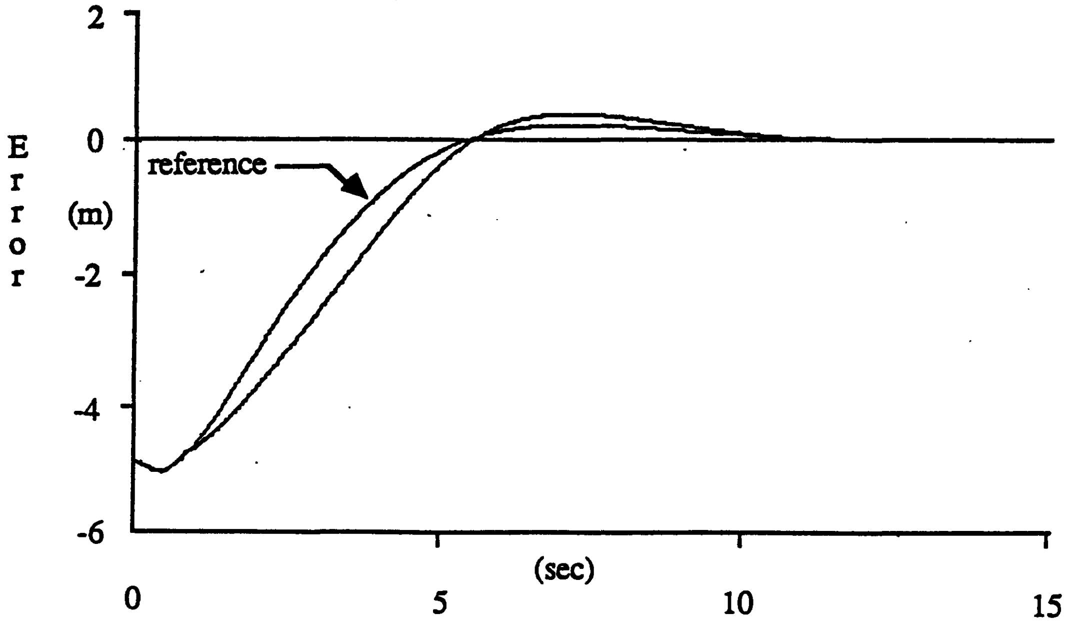


Graph 9.5 - *Classical* - Response in X (M = 1 kg, L = 3 m)

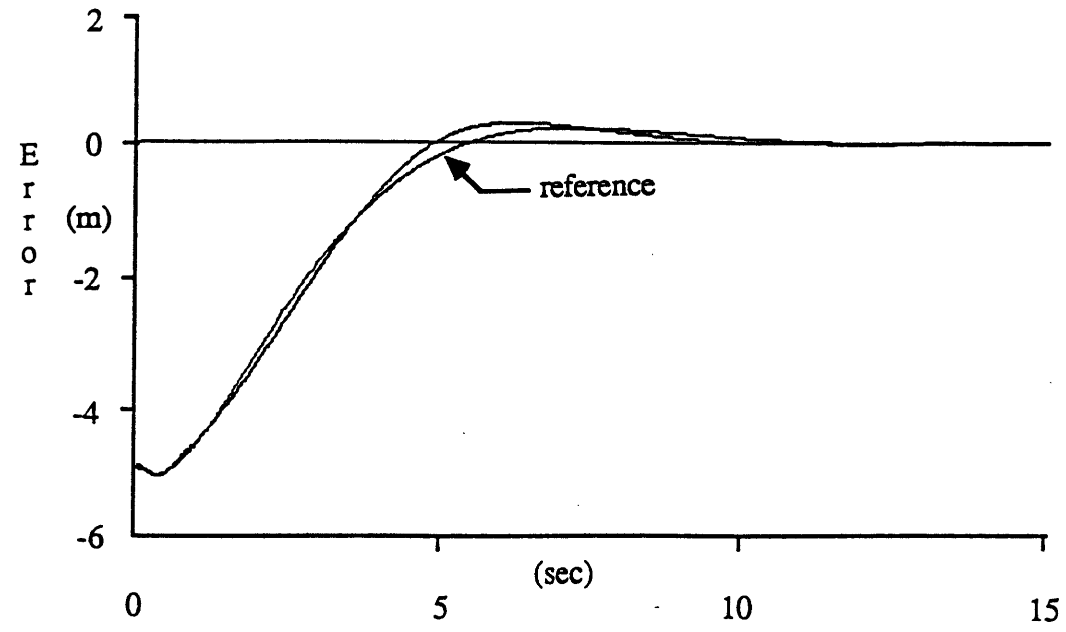
The added length of the pole causes an oscillation (1.32 m peak-to-peak) and a substantially longer settling time. Despite keeping the pole upright, the classical controller reveals severe limitations at this parameter setting.

9.2 THE NEURAL NETWORK CONTROLLER

This section takes the neural network controller through a series of experiments which illustrate distinct advantages over the classical controller. The first network controller experiment called *Linear-Initialization* confirms that the interconnection weights converge to a desired configuration. For this experiment, the weights are initialized using the reference controller gains assuming the neurons operate in the *linear* region. The network attempts to control a plant with dynamic parameters ($M = 1$ kg, $L = 1$ m) identical to the model reference. To teach the network, the user gives the controller a string of 5 m commands allowing for settling time in between. The superimposed graphs of the initial network response and the reference response are almost the same, but the assumption of linear neurons is not valid. After five commands, the weights converge to give a response very close to the reference.

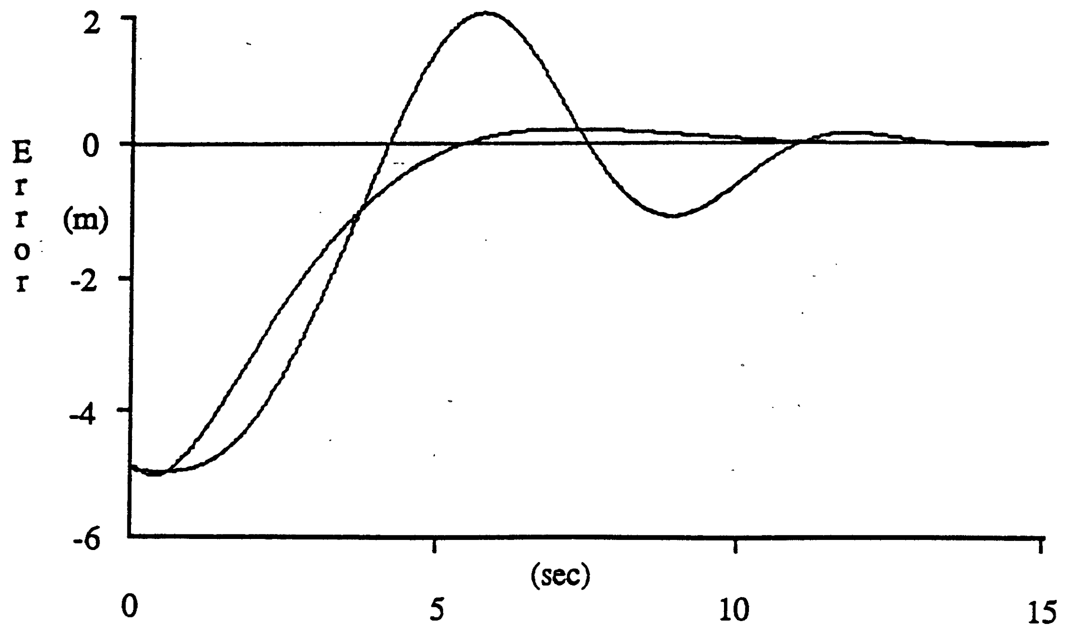


Graph 9.6 - *Linear-Initialization* - 1st Response in X ($M = 1$ kg, $L = 1$ m)



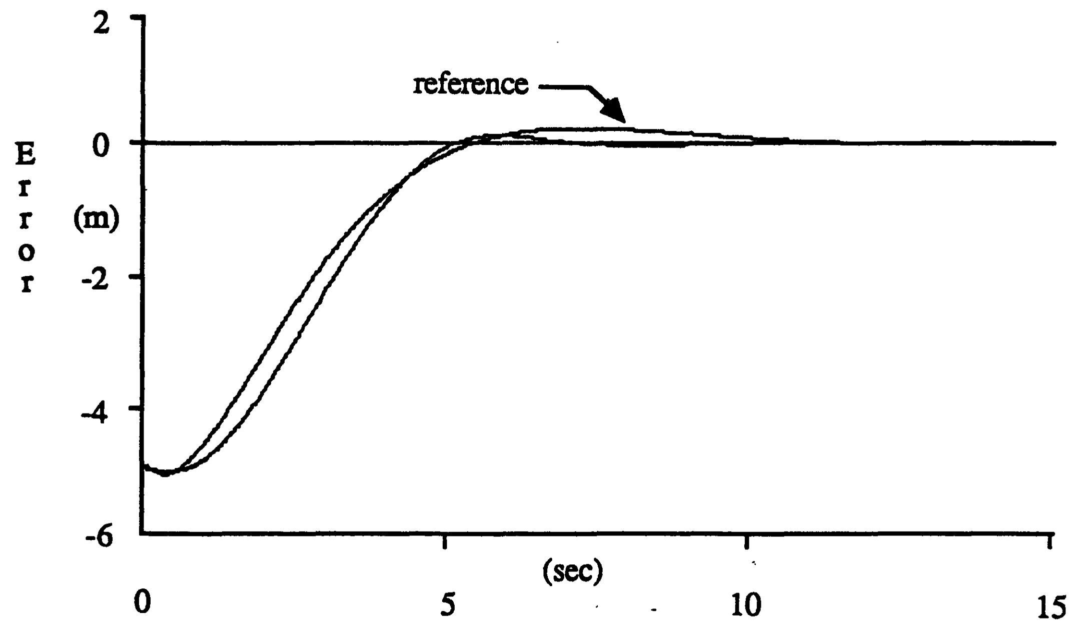
Graph 9.7 - *Linear-Initialization* - 5th Response in X ($M = 1$ kg, $L = 1$ m)

Nominal is similar to *Linear Initialization* except now the weights begin at small *random* positive values. The pole does not fall despite these initial weights and after five commands the network converges on a desirable weight configuration.



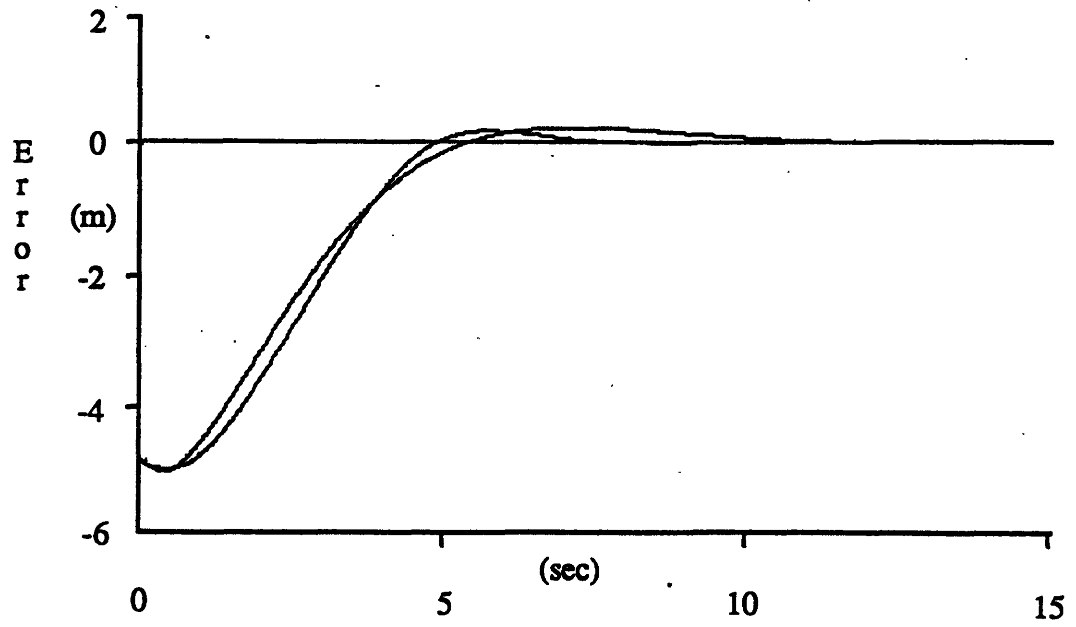
Graph 9.8 - *Nominal* - 1st Response in X ($M = 1$ kg, $L = 1$ m)

The second response reduces the percent overshoot M_p from 42.5% to 2.5%.



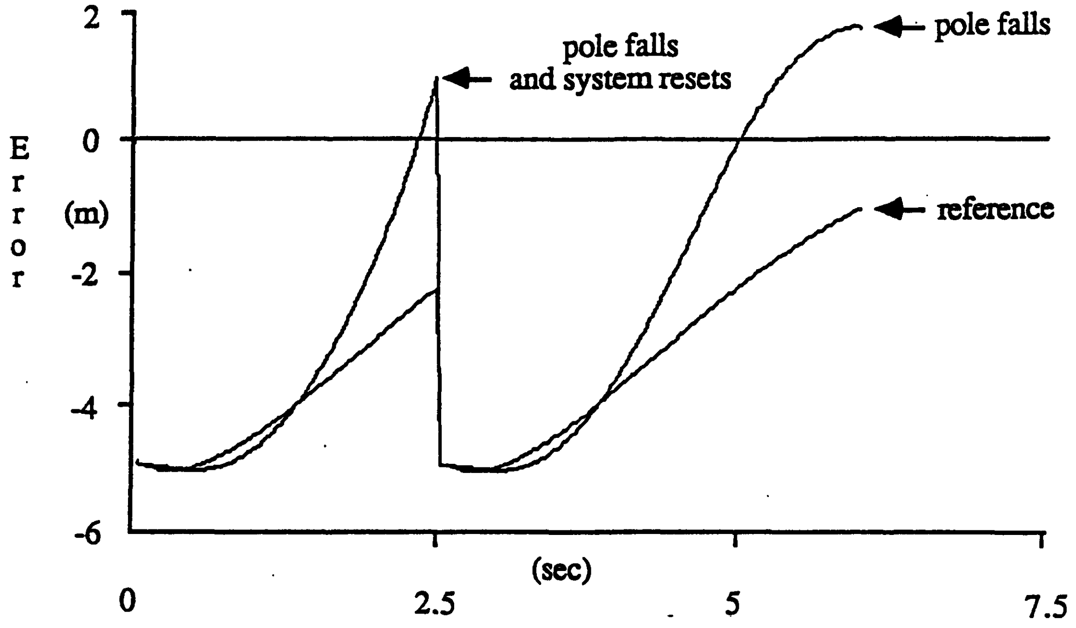
Graph 9.9 - *Nominal* - 2nd Response in X (M = 1 kg, L = 1 m)

Finally, the fifth response has an overshoot of 4.3% compared to the reference M_p of 4.4%.



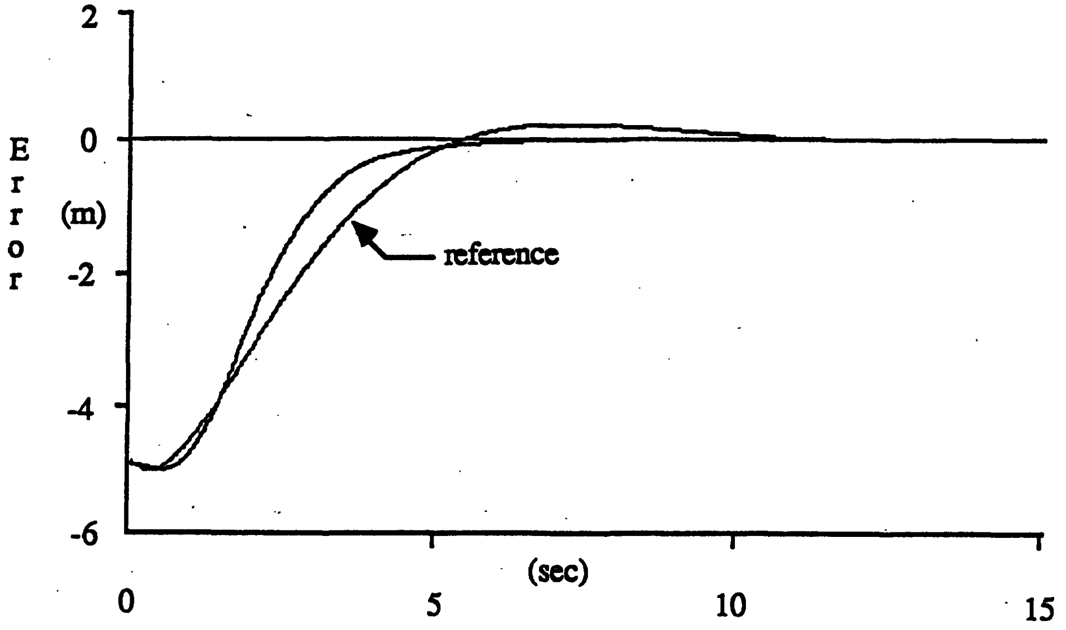
Graph 9.10 - *Nominal* - 5th Response in X (M = 1 kg, L = 1 m)

Linear Big M uses the initial weights in the *Linear Initialization* experiment but now the cart mass is set to 3 kg. The classical controller fails to keep the pole upright with this parameter setting. The network also fails to balance the pole on the first two tries.



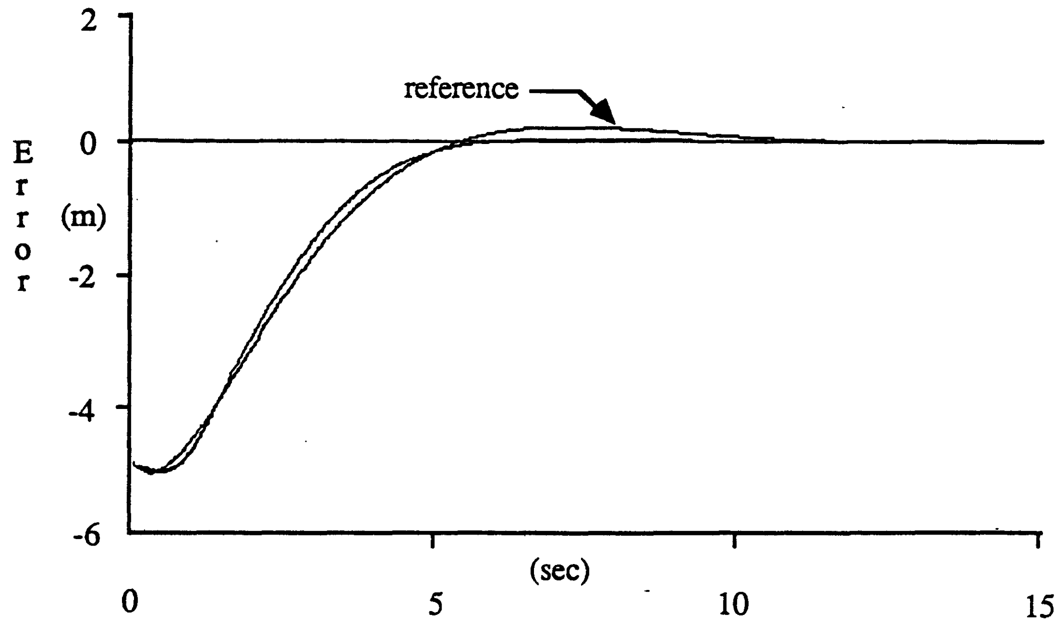
Graph 9.11 - *Linear Big M* - 1st and 2nd Response in *X* (*M* = 3 kg, *L* = 1 m)

The network controller keeps the pole upright on the third response, but the curve diverges from the reference in its early stages.



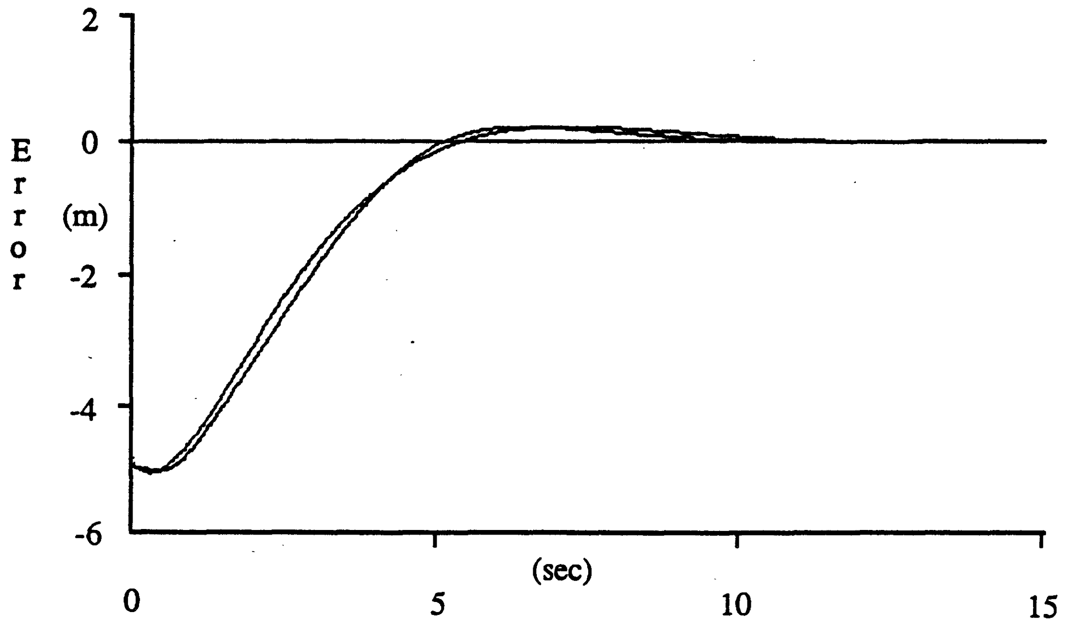
Graph 9.12 - *Linear Big M*- 3rd Response in *X* (*M* = 3 kg, *L* = 1 m)

An intermediate fifth response shows good progress in following the reference,



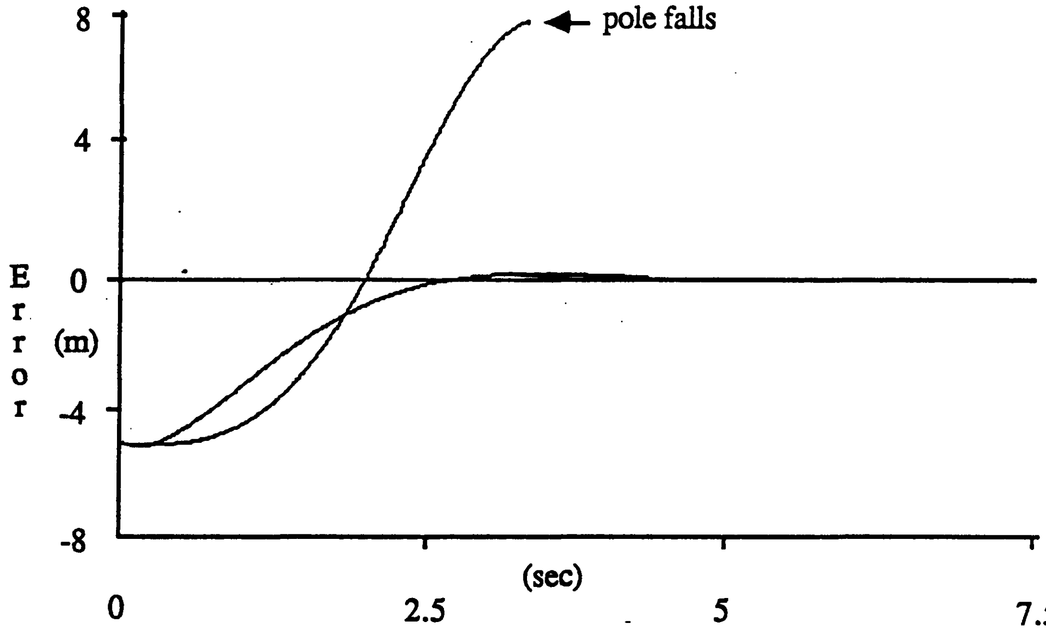
Graph 9.13 - *Linear Big M*- 5th Response in *X* (*M* = 3 kg, *L* = 1 m)

and eventually the weights settle to give a good seventh response.



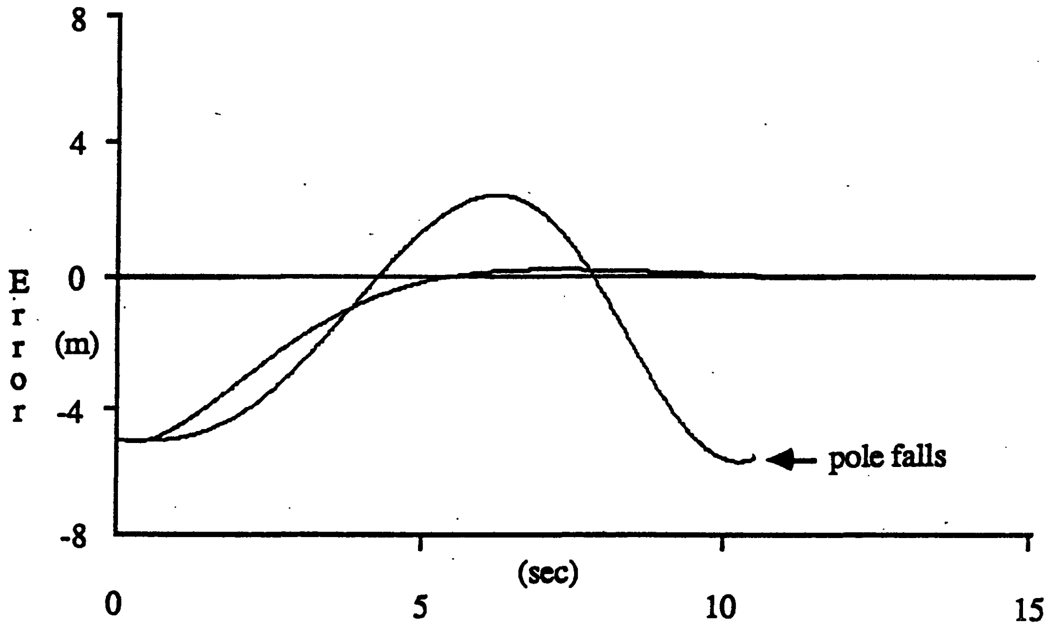
Graph 9.14 - *Linear Big M* - 7th Response in *X* (*M* = 3 kg, *L* = 1 m)

Random Big M uses random initial weights as in the *Nominal* experiment with a cart mass of 3 kg. The network fails to balance the pole on the first try after overshooting the desired cart position by 170.0%.



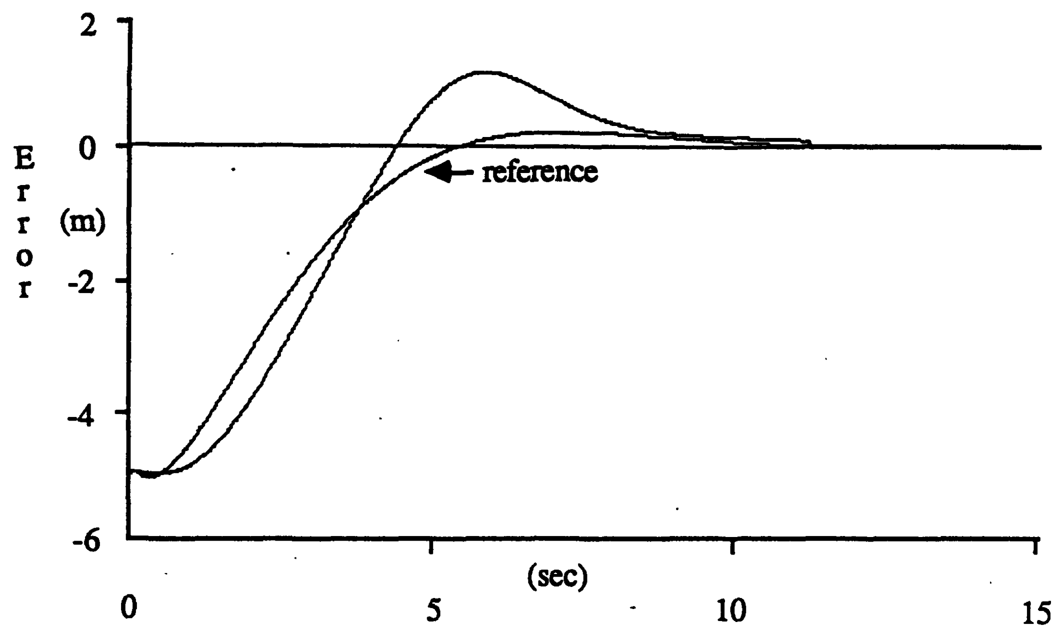
Graph 9.15 - *Random Big M* - 1st Response in X ($M = 3$ kg, $L = 1$ m)

The pole also falls in the second response after the cart overshoots X_c by 52.5% and overcompensates on the recovery.



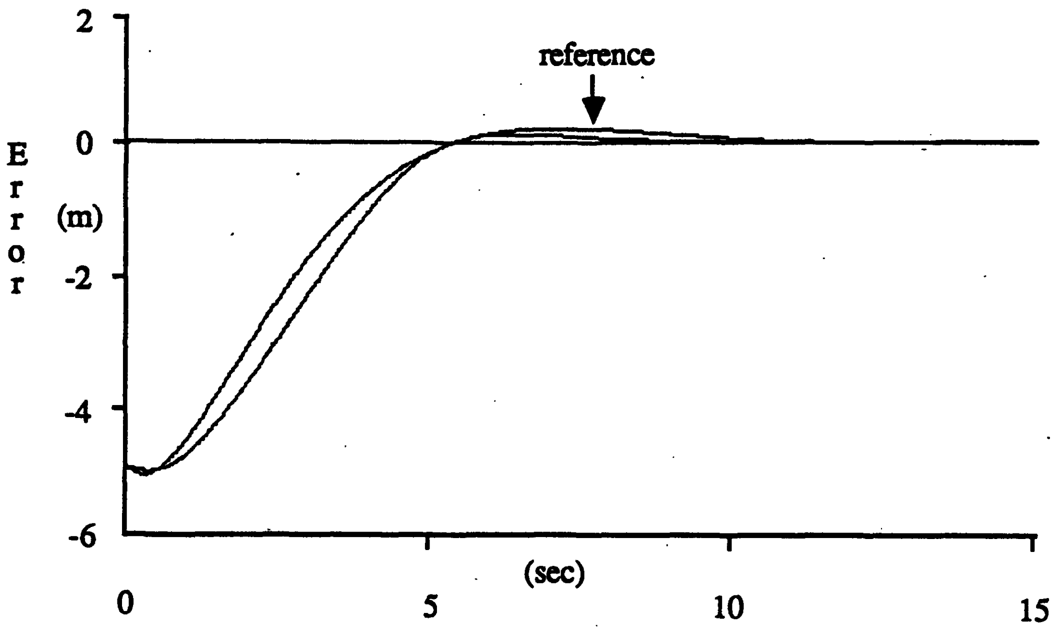
Graph 9.16 - *Random Big M*- 2nd Response in X ($M = 3$ kg, $L = 1$ m)

The third response successfully reaches the desired cart position with an M_p of 24.5% and a settling time 1.5 sec longer than the reference response



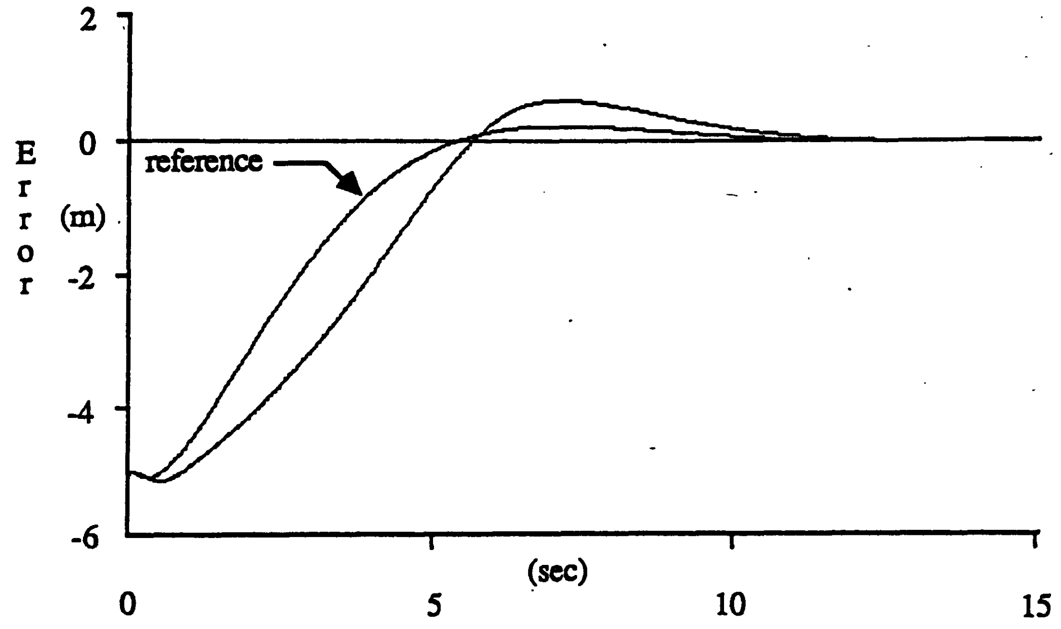
Graph 9.17 - *Random Big M*- 3rd Response in *X* ($M = 3$ kg, $L = 1$ m)

The weights eventually settle to give a tenth response that closely follows the reference.



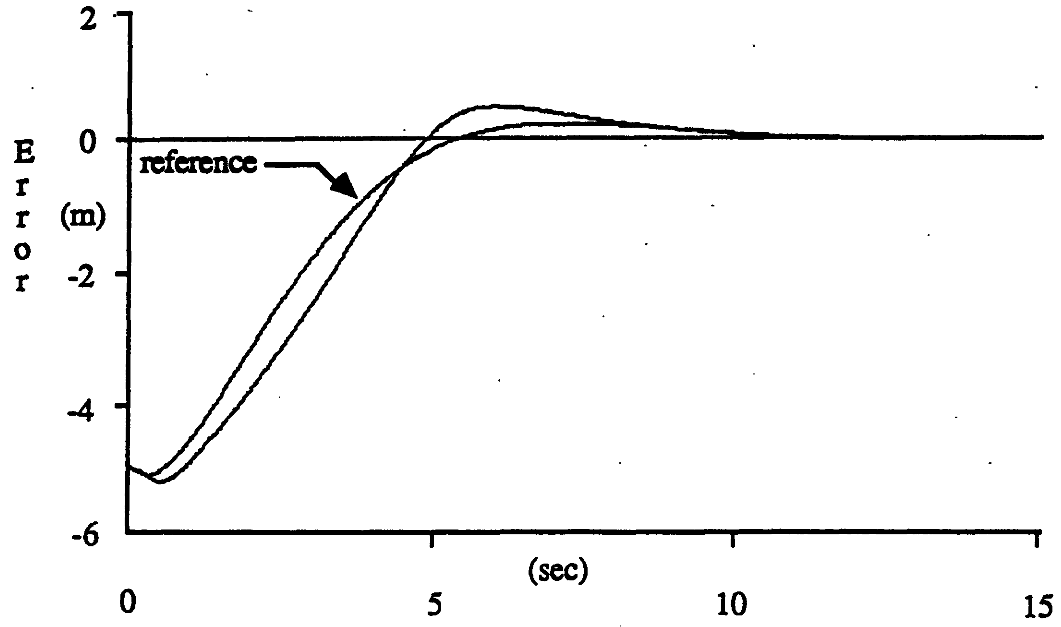
Graph 9.18 - *Random Big M* - 10th Response in *X* ($M = 3$ kg, $L = 1$ m)

Linear Big L begins with an initial linear guess for the weights and attempts to balance a cart-pole with an *M* of 1 kg and an *L* of 3 m. With these settings the classical controller experiences a slow settling time with a large oscillation (1.32 m peak-to-peak) in steady state. The initial response of the plant overshoots the desired cart position by 13.3%.



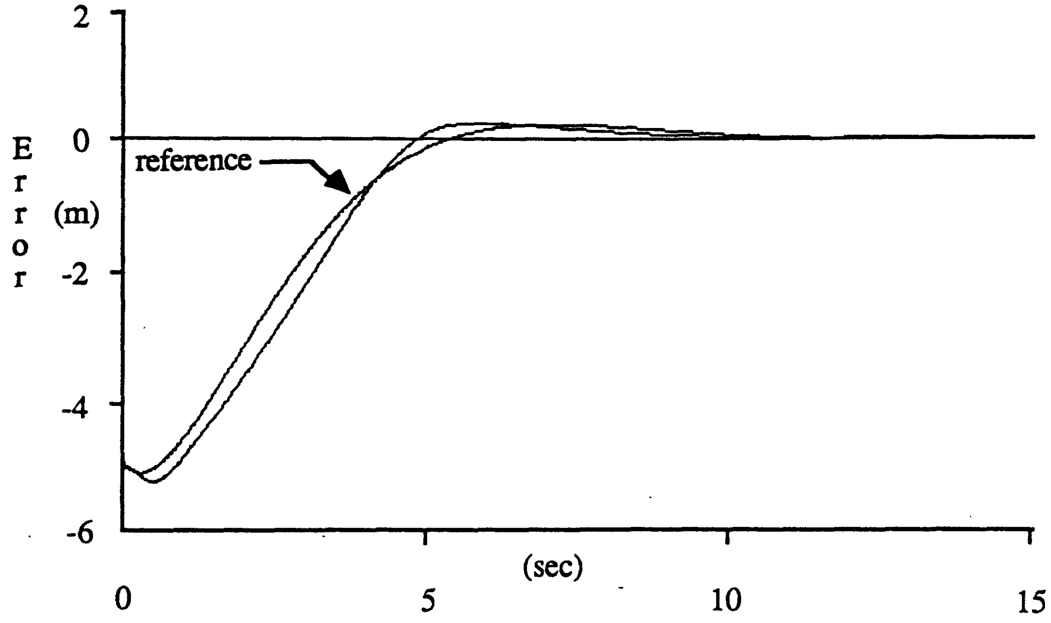
Graph 9.19 - *Linear Big L* - 1st Response in *X* ($M = 1$ kg, $L = 3$ m)

The second shows an improvement especially in the early stages of the response, and



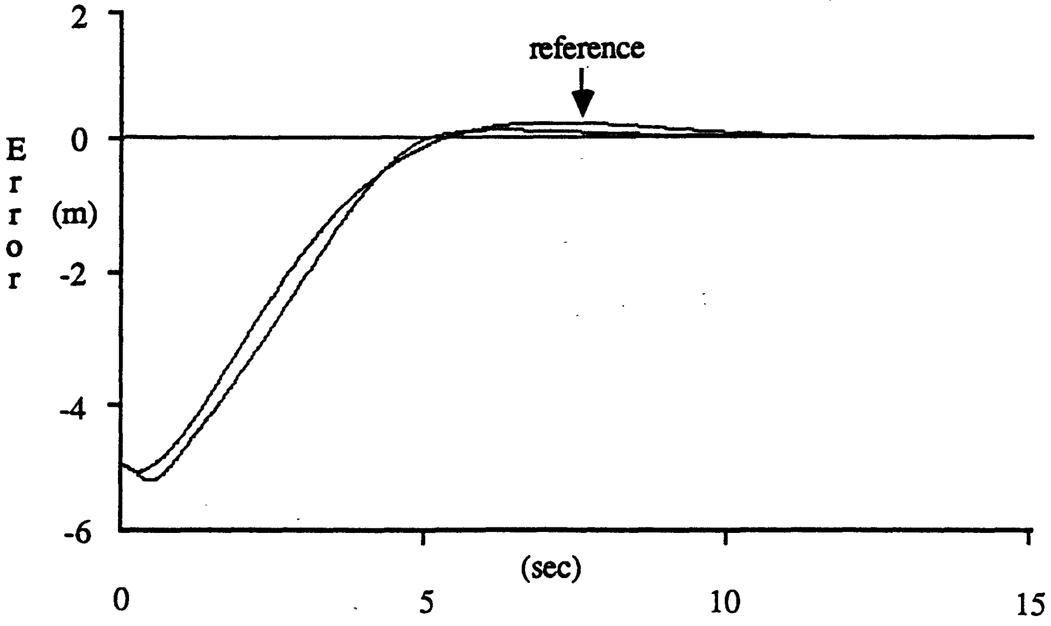
Graph 9.20 - *Linear Big L* - 2nd Response in *X* ($M = 1$ kg, $L = 3$ m)

the third response follows the reference rather well with an overshoot of 4.8% compared to the reference M_p of 4.4%.



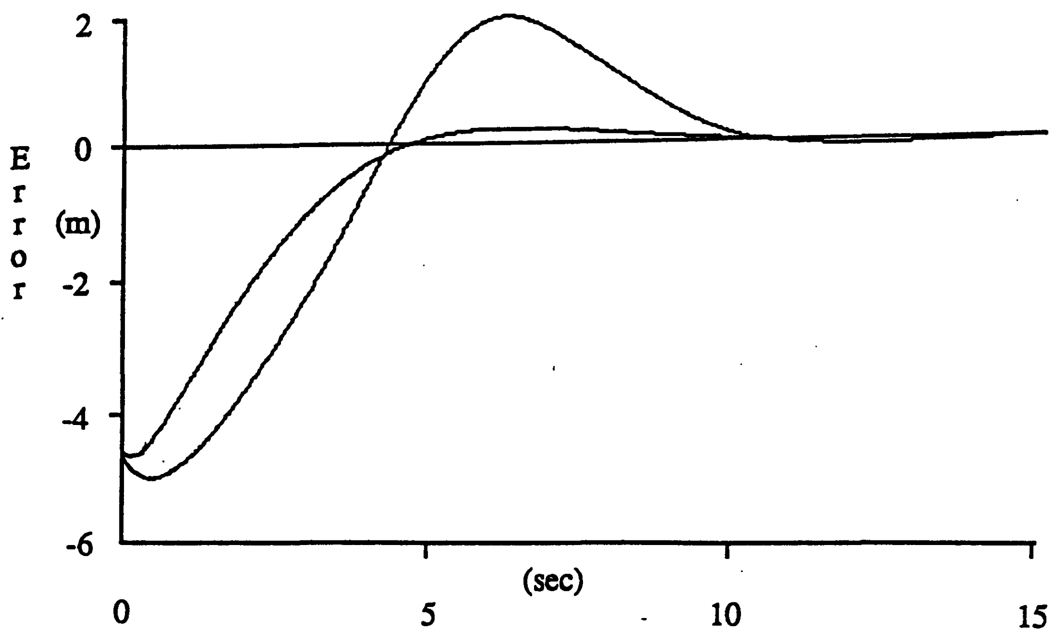
Graph 9.21 - *Linear Big L* - 3rd Response in X (M = 1 kg, L = 3 m)

Finally, the fourth response mimics the reference despite a larger dip at the beginning due to the increased pole length.



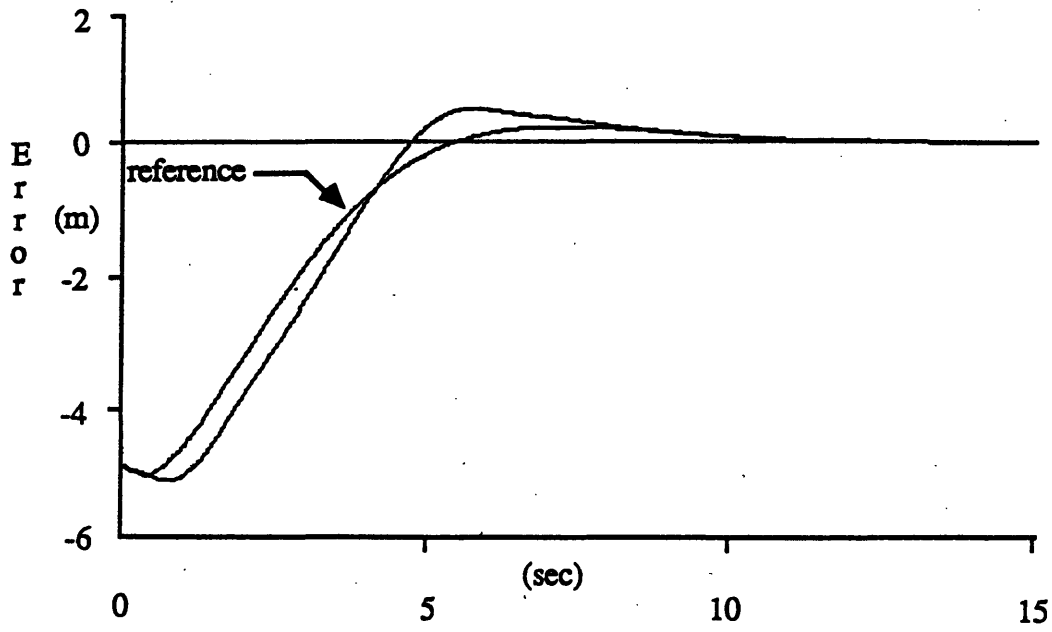
Graph 9.22 - *Linear Big L* - 4th Response in X (M = 1 kg, L = 3 m)

Random Big L begins with small random weights and attempts to balance the cart-pole with an M of 1 kg and an L of 3 m. The cart-pole overshoots the desired position on the first response by 41.5%.



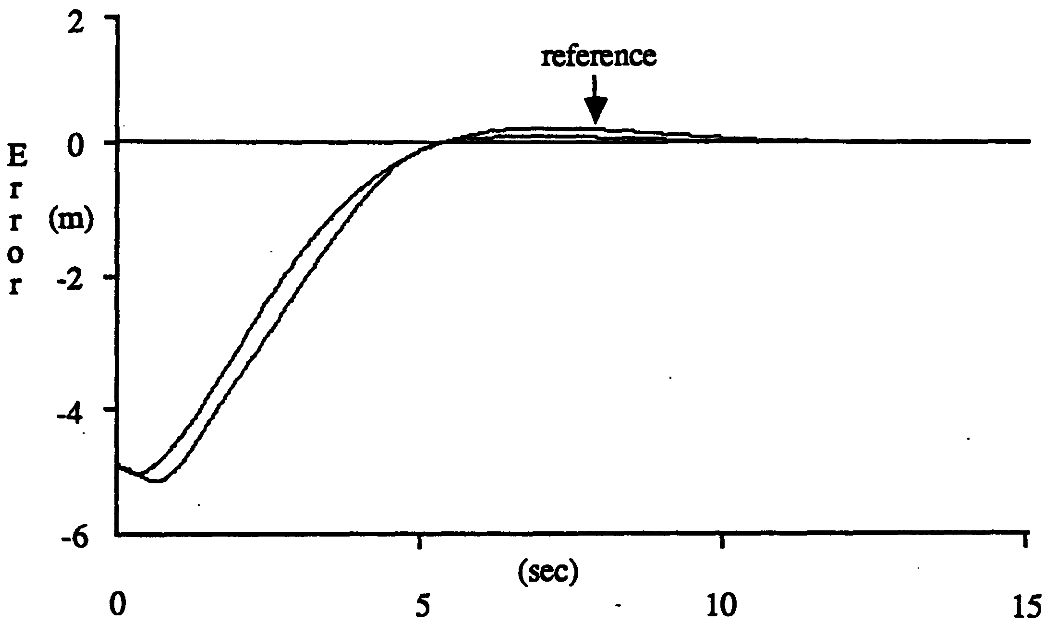
Graph 9.23 - *Random Big L* - 1st Response in *X* (M = 1 kg, L = 3 m)

The second response decreases the overshoot to 11.3%, and



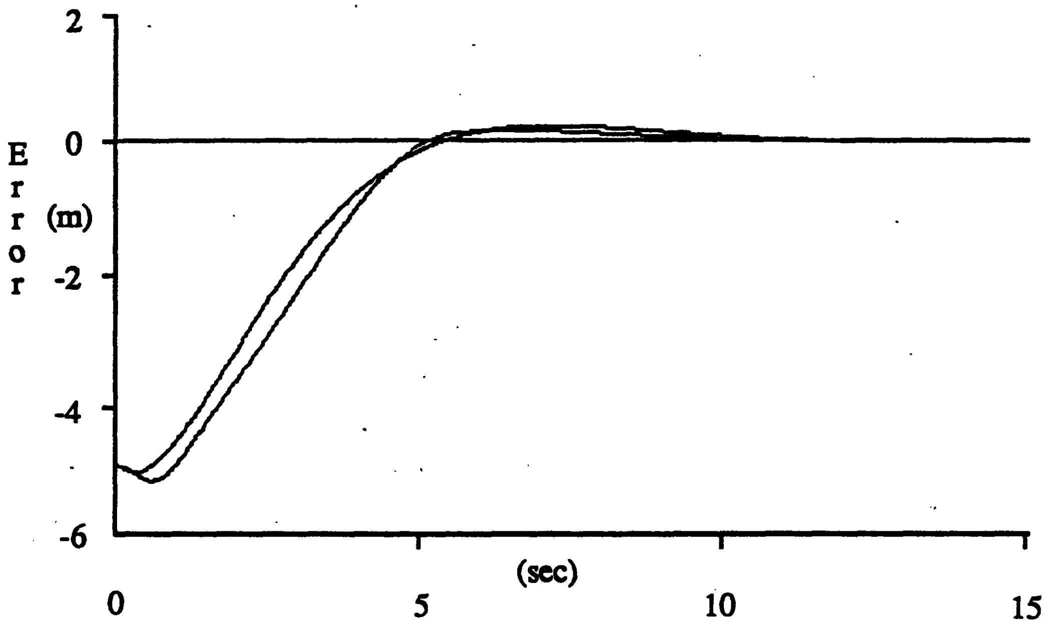
Graph 9.24 - *Random Big L* - 2nd Response in *X* (M = 1 kg, L = 3 m)

the third response almost duplicates the reference.



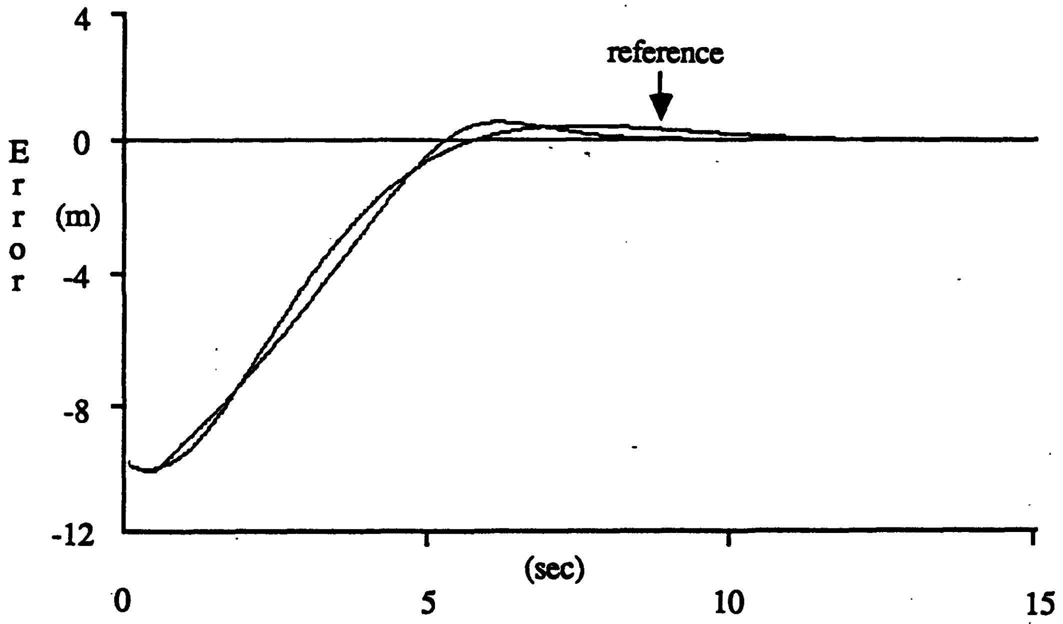
Graph 9.25 - *Random Big L* - 3rd Response in X ($M = 1$ kg, $L = 3$ m)

After only four position commands to the cart-pole, the network controller learns to imitate the response of the reference.



Graph 9.26 - *Random Big L* - 4th Response in X ($M = 1$ kg, $L = 3$ m)

The last experiment, *Big* X_C, demonstrates the thresholding effects of the neurons for large commanded cart positions. After *Linear Initialization* converges on a set of weights, the cart is commanded to 10 m. For any X_C larger than 5 m, the classical controller must use limiters on θ_c to keep the pole balanced. The network controller relies only on the thresholding characteristics of the neurons.



Graph 9.27 - *Big* X_c - Response in X (M = 1 kg, L = 1 m)

3 SUMMARY OF THE RESULTS

Each of these experiments highlights an important capability of the network controller. In general, an experiment takes less than ten commands to converge on a good set of weights, and often the plant and reference responses are close after only two or three commands. The network typically converges quicker with an initial linear guess of the weights than with originally random weights.

The following two tables summarize the results of the experiments that started with random weights. A measure of similarity between the plant and reference responses is the difference in percent overshoot M_p. For all the experiments, the network controller attempts to duplicate the *target* overshoot of 4.4% of the reference response. Table 9.1 displays the results of the experiments where M was increased and L remained at 1 m. A "*" indicates the network controller failed to balance the pole.

	Classical Controller		Network Controller				
cart mass *M*	balance the pole?	over-shoot *Mp*	overshoot for each response ("*" indicates failure to balance the pole) *Mp*				
kg		%	1	2	3	4	5
1.0	yes	4.4	42.5	2.5	3.8	4.0	4.3
1.5	yes	3.0	52.5	7.0	6.8	5.5	4.4
2.0	yes	1.6	56.3	0.0	1.3	1.3	2.5
2.5	yes	0.6	*	48.8	3.8	3.5	4.6
3.0	no	-	*	*	24.5	0.5	1.3

Table 9.1 - Cart Mass Perturbations

Despite the decrease in M_p with increased cart mass, the classical controller must apply an almost constantly saturated force F to balance the pole when M is larger than 2.0 kg. Towards the fifth response using the network controller the overshoot for each setting of M approaches the reference M_p of 4.4%. Table 9.2 presents data on the experiments where L was increased and M remained at 1 kg.

	Classical Controller		Network Controller				
pole length *L*	steady state oscillation	over-shoot *Mp*	overshoot for each response *Mp*				
m	m (p-to-p)	%	1	2	3	4	5
1.0	none	4.4	42.5	2.5	3.8	4.0	4.3
1.5	none	3.4	20.2	19.5	11.3	8.3	5.5
2.0	none	4.0	45.5	15.5	9.5	5.5	3.8
2.5	none	4.0	50.0	8.3	8.0	3.6	3.7
3.0	1.32	26.8	41.5	11.3	10.0	2.5	3.8

Table 9.2 - Pole Length Perturbations

At L larger than 2.0 m the classical controller waves the pole back and forth before settling on the commanded cart position. At L = 3.0 m the classical controller fails to reach the commanded position due to a severe oscillation in steady state. All of the fifth responses for the network controller show an M_p close to the target value of 4.4%.

The final weights of the experiment offer important insight into the operation of the neural network controller. Since the network configuration contains six adjustable weights (see Figure 9.1) and the compensator for the cart-pole only requires 4 gains, there is not a unique set of desired weights. Forpropagation attempts to "smear" the information of a four-gain compensator into six weights.

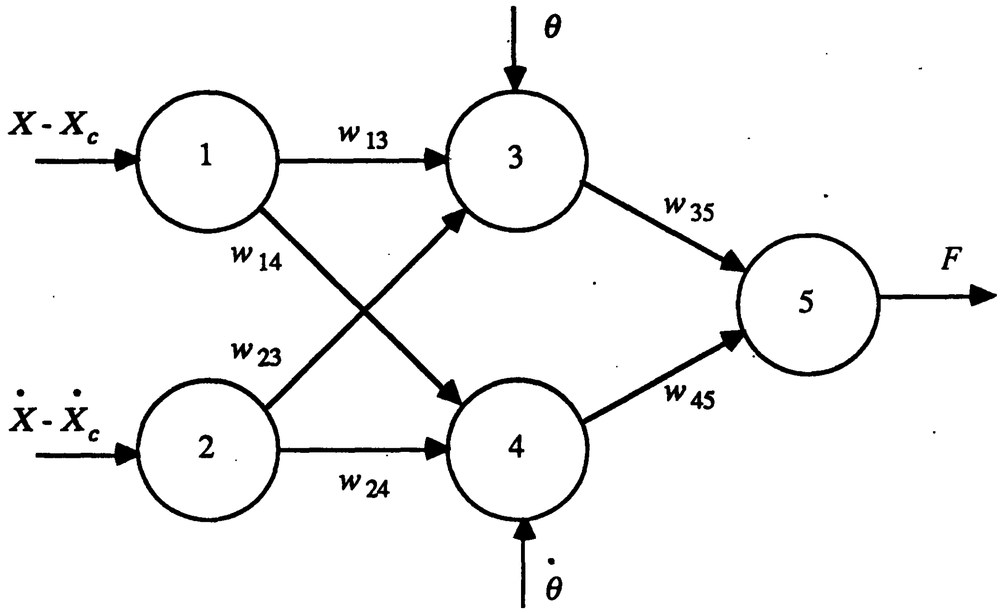


Figure 9.1 - Network Configuration

Table 9.3 shows the final value of the weights for some of the experiments.

L	M	initial weights	Weights After Convergence					
m	kg		$w13$	$w14$	$w23$	$w24$	$w35$	$w45$
1.0	1.0	linear	0.73	0.63	0.61	0.52	1.02	0.65
1.0	1.0	random	0.47	0.51	0.47	0.65	1.00	0.66
1.0	3.0	linear	0.61	0.55	0.60	0.57	1.12	0.56
1.0	3.0	random	0.51	0.57	0.41	0.35	1.21	0.91
3.0	1.0	linear	0.64	0.63	0.52	0.54	1.29	0.85
3.0	1.0	random	0.67	0.64	0.57	0.51	1.28	0.94

Table 9.3 - Network Weights

As the cart mass M or the pole length L increases, w_{35} and w_{45} consistently converge to larger values. This suggests the network controller learns to quickly adjust to small deviations in θ and $\dot{\theta}$ when the plant dynamics make it more difficult to balance the pole.

10 RECOMMENDATIONS AND CONCLUSIONS

The neural network controller performs well under the various exercises of Chapter 9. This demonstrates the ability of forpropagation to seek and maintain weight configurations capable of controlling the cart-pole in various operating envelopes. Section 10.1 presents closing comments on the experiments and general statements on the findings of this research. Work in this field is far from complete and Section 10.2 offers some new avenues of investigation to pursue.

10.1 CONCLUSIONS

The first experiment, *Linear-Initialization,* confirmed the stability of the forpropagation algorithm. As the network controller approached the response of the reference, the weights adjusted less dramatically and any small changes over later responses integrated out to near zero. *Nominal* demonstrated the ability of the network controller to learn a good weight configuration given no prior knowledge. Positive results insured the gradient search technique invoked by forpropagation would work even with initial weights far from the target. *Linear Big M* and *Linear Big L* showed the network controller surpassing the capabilities of the fixed-gain classical controller in the presence of a large change in cart mass or pole length. However, the weights for these experiments began with a linear guess giving the network important initial information. Finally, in the *Nominal Big M* and *Nominal Big L* experiments, the network controller outperformed the classical controller given no prior knowledge.

Big X_C revealed the effect of the non-linear neurons in the network. After the weights converged in the *Linear-Initialization* experiment, the cart-pole was commanded to 10 m, a task the classical controller would rely on limiters to perform. The network controller, with the aid of the non-linear neurons, did not command a pole angle greater than 15 degrees and moved the cart to the desired position. This result shows that limiting effects, artificially produced in the classical controller, are naturally inherent in the network controller.

The network inner-outer loop infrastructure proved essential for a successful controller. The first layer received $X - X_c$ and $\dot{X}$ and passed commanded positions for θ and $\dot{\theta}$ to the second layer. This separation of layers by inner and outer loop construction can be used in many applications where this assumption is valid. Due to this infrastructure, the network contained six adjustable weights as opposed to the four gains of the reference controller. The network used this redundancy to "smear" the knowledge of the controller into six weights. This common property of neural networks builds weight configurations that allow small contributions from all neurons.

Overall, the forpropagation algorithm teamed with the network structure accomplished tasks a similar network taught using backpropagation could not perform. The forpropagation teaching sessions did not require any initial knowledge but merely a reference to show the general response. Furthermore, the plant was only tracked in X and $\dot{X}$ allowing θ and $\dot{\theta}$ to take on any time responses necessary. The action of the pole over a response to a commanded cart position can vary greatly depending on the pole length and cart mass. In the *Big M* and *Big L* experiments, the commanded pole positions were modest compared to the classical controller.

10.2 RECOMMENDATIONS FOR FURTHER WORK

Through the course of this work, two important fields of study related to network controllers were identified but not investigated thoroughly. All previous experiments assumed full use of state variables, but Section 10.2.1 suggests a new approach to implement partial-state feedback for the cart-pole controller. Section 10.2.2 presents the idea of adjusting the reference to always insure peak system performance.

10.2.1 Partial-State Feedback Controllers

The architectures for partial-state feedback controllers closely resemble their full-state counterparts but require memory to estimate the unavailable state variables. Consequently, these controllers use *one-step propagation* which allows values from two time steps to be differenced to estimate a velocity for the cart or pole. First, this section considers the network that merely balances the pole. A value for the pole velocity is estimated by differencing two pole positions over time. After determining $\dot{\theta}$, the controller looks similar to the full-state version with two weighted inputs summed at the output neuron to arrive at a control action F.

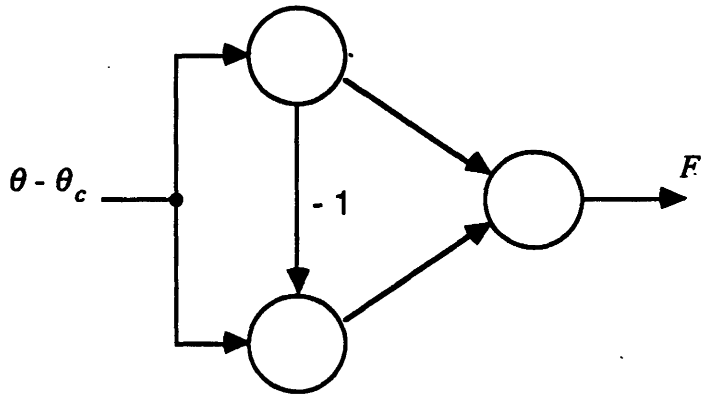


Figure 10.1 - Proposed Network Structure of Partial-State θ Controller

The partial-state X controller, which balances the pole while regulating the cart position, cascades two triangular forms of the partial-state θ controller. The first neuron triad receives $X - X_c$ and differences previous inputs to arrive at an estimate for $\dot{X}$. Signals proportional to X and $\dot{X}$ are then multiplied by weights and summed at an intermediate *linear* neuron. This linear neuron also receives θ and outputs a value proportional to $\theta - \theta_c$ to the next triad of neurons. Again, differencing gives an estimate of $\dot{\theta}$. The final control action is a function of $\theta - \theta_c$ and $\dot{\theta}$ multiplied by weights.

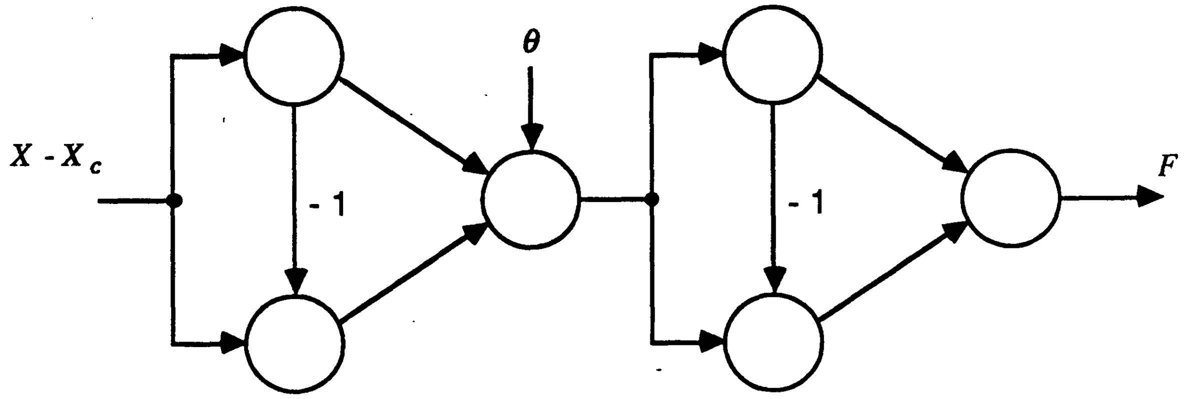


Figure 10.2 - Proposed Network Structure of Partial-State *X* Controller

This architecture, though slightly more complicated than the full-state feedback network, implements the same representation assuming the differencing operation is relatively accurate.

10.2.2 Adjustable Reference

For all experiments the reference remained the same. This was adequate to prove convergence qualities and performance capabilities of the forpropagation algorithm. However, a new approach that adjusts the reference can maintain optimal performance in all operating environments. If the dynamic problem becomes simpler, then the control action *F* could give a better response than the current reference. An algorithm could make small adjustments to the reference depending on a time history of the control action. If *F* rarely peaks, the reference response can be improved and a full range of control action can yield a better response. However, if *F* is constantly saturating and the network controller is struggling, the reference response can be slightly degraded.

APPENDIX

The appendix contains the important source code listings of the neural network simulator. The three programs, Pole-Exec, Forprop, and Pole-Dynamics, do not represent all of the code necessary for the simulator. Various programs dealing only with graphics, support routines, or simple utility functions do not appear. This section provides the top-level and critical lower-level routines for the simulator.

A.1 POLE-EXEC.LISP

Pole-Exec runs the top-level loop of the simulator and handles all operations on experiments. The user can create, stop, run, and reset the experiments defined in this program.

POLE-EXEC FLAVOR

```
(defflavor pole-exec
    ((process)
    (process-run-f)
    (net-type)
    (net)
    (dynamics (make-instance 'pole-state))
    (teacher (make-instance 'teacher))
    (network-pane)
    (pane-b)
    (phase-pane)
    (menu-pane)
    (duration-pane)
    (cart-pane)
    (keyboard-io-pane)
    (network-pane-update-f t)
    (pane-b-update-f t)
    (phase-pane-update-f nil)
    (phase-pane-x-abs 7)
    (phase-pane-y-abs 7)
    (duration-pane-update-f t)
    (duration-pane-update-inc-f t)
    (cart-pane-update-f nil)
    (discrete-f t)
    (lesson 0)
    (count 0)
    (suspend-f)
    (scale 1.0)
    (scale-exp -.005)
    (grade 0.0)
    (total-count 0)
    (final-count 3000)
    (neuron-update-f)
    (neuron-on-display)
    (old-circle-data nil)
    (pathname "M:>bes>test.lisp")
    (banner "this is a test"))
    ()
  :initable-instance-variables
  :writable-instance-variables)
```

POLE-EXEC METHODS

```
(defmethod (run-experiment pole-exec) (controller)
  (if process
    (mom-menu "Experiment Already Running, Big Dummy" '(nil "Abort" nil))
    (progn
     (setq process-run-f t)
     (setq process
          (process-run-function "Neural Loop Process" 'run-loop self)))))

(defmethod (stop-experiment pole-exec) ()
  (cond (process-run-f
        (setq process-run-f nil)
        (si:process-sleep 120)
        (setq process nil))
       (t (mom-menu "No Experiment Is Running" '(nil "Abort" nil)))))

(defmethod (reset-experiment pole-exec) ()
  (if process-run-f  (stop-experiment self))
  (setq lesson 0)
  (setf (pole-state-lesson dynamics) 0)
  (setq suspend-f nil)
```

```
 (initialize-weights net)
 (balance dynamics)
 (balance (pole-state-reference dynamics))
 (setf (pole-state-time dynamics) 0.0)
 (setf (pole-state-desired-x dynamics) 0.0)
 (setf (pole-state-time (pole-state-reference dynamics)) 0.0)
 (setq neuron-update-f nil)
 (cond-every
  (pane-b-update-f
   (init pane-b))
  (network-pane-update-f
   (init network-pane net))
  ((not (pole-state-clascon-f dynamics))
   (init phase-pane)
   (setf phase-pane-update-f nil)))
 (init duration-pane net)
 (init cart-pane dynamics))
```

BLIP READER METHODS

```
(defmethod (loop-net-blip-reader pole-exec) ()
 (loop do (handle-neuron-click self (send network-pane :any-tyi))))

(defmethod (handle-neuron-click pole-exec) (blip)
 (if (listp blip)
   (if (> (length blip) 2)
     (if (eq (car blip) ':typeout-execute)
       (cond ((or (= (second blip) 0) (= (second blip) 1))
              (setq neuron-on-display (car (third blip)))
             (if old-circle-data
              (progn
                 (send network-pane ':draw-filled-in-circle
                    (first old-circle-data)
                    (second old-circle-data)
                    (third old-circle-data)
                   tv:alu-andca)
                 (send network-pane ':draw-circle
                    (first old-circle-data)
                    (second old-circle-data)
                    (third old-circle-data))))
              (send network-pane ':draw-filled-in-circle
                 (second (third blip))
                 (third (third blip))
                 (fourth (third blip)))
```

```
            (setq old-circle-data (cdr (third blip)))
             (show-weight neuron-on-display pane-b)
             (setq neuron-update-f (= 0 (second blip))))
           ((= 2 (second blip)) (change-neuron (car (third blip))))
          ((= 3 (second blip))
           (if old-circle-data
            (progn
                (send network-pane ':draw-filled-in-circle ;erase black circle
                   (first old-circle-data)
                   (second old-circle-data)
                   (third old-circle-data)
                  tv:alu-andca)
                (send network-pane ':draw-circle  ;redraw white circle
                   (first old-circle-data)
                   (second old-circle-data)
                   (third old-circle-data))
              (setq old-circle-data nil)))
            (setq neuron-on-display (network-threshold net))
            (show-weight neuron-on-display pane-b)
           (setq neuron-update-f t))))))
```

MAIN EXECUTION LOOP

```
(defmethod (run-loop pole-exec) ()
  (send network-pane :set-io-buffer (tv:make-io-buffer 1024))
  (send network-pane :select)
  (if (pole-state-write-stats-f dynamics)
    (append-to-streams `(,pathname) banner))
  (loop while process-run-f
     do
   (cond ((not suspend-f)
         (setq lesson (1+ lesson))
          (setf (pole-state-lesson dynamics) (1+ (pole-state-lesson dynamics)))
          (if (member (plot-pane-plot-type duration-pane) '(2 3))
            (init duration-pane net))
         (setq count 0)
          (setf (pole-state-duration dynamics) 0)
         (balance dynamics)
          (balance (pole-state-reference dynamics))
         (set-kick dynamics
                (* (pole-state-initial-kick dynamics) (if (zerop (mod lesson 2)) -1 1)))
          (set-kick (pole-state-reference dynamics)
                (* (pole-state-initial-kick dynamics) (if (zerop (mod lesson 2)) -1 1)))
         (setq scale 1.0))
    (t (setq suspend-f nil)))
(loop while (and process-run-f
            (not (funcall (teacher-trip-function teacher) dynamics)))
   do
  (setq count (1+ count))
  (setf (pole-state-duration dynamics) (* count (pole-state-dt dynamics)))
  (if (and (pole-state-write-stats-f dynamics)
        (zerop (mod count (pole-state-write-stats-frequency dynamics))))
    (append-to-streams `(,pathname)
                   "~% time  =  ~2,2,6$  x-com = ~2,2,6$  kick  = ~2,2,6$~%
                         theta  =  ~2,2,6$  thetad = ~2,2,6$  x  = "
                  (pole-state-time dynamics)
                  (pole-state-desired-x dynamics)
                  (pole-state-kick dynamics)
                  (pole-state-theta dynamics)
                  (pole-state-thetad dynamics)
                  (pole-state-x dynamics)
                  (pole-state-xd dynamics)))
  (if cart-pane-update-f
    (update cart-pane cart-pane dynamics))
  (if phase-pane-update-f
    (update phase-pane (pole-state-theta dynamics) (pole-state-thetad dynamics)
        (/ (pole-state-kick dynamics) 10.0)))
  (zl:selectq (plot-pane-plot-type duration-pane)
   (1 (update duration-pane lesson (* count (pole-state-dt dynamics))))
   (2 (update duration-pane (* count (pole-state-dt dynamics))
          (- (pole-state-x dynamics) (pole-state-desired-x dynamics))))
   (3 (update duration-pane (* count (pole-state-dt dynamics))
          (- (pole-state-x dynamics) (pole-state-desired-x dynamics))
          (* count (pole-state-dt dynamics))
          (- (pole-state-x (pole-state-reference dynamics))
            (pole-state-desired-x (pole-state-reference dynamics))))))
  (handle-neuron-click self (send network-pane :any-tyi-no-hang))
  (if neuron-update-f (update pane-b neuron-on-display))
  (cond ((pole-state-clascon-f dynamics)
        (if (pole-state-time-fudge-f dynamics) (cl:sleep 0.04))
       (update dynamics)
       (set-kick dynamics (clascon dynamics)))
    (t
      (update dynamics)
       (update (pole-state-reference dynamics) dynamics)
       (evaluate dynamics net scale)
       (if (teacher-teacher-on-f teacher) (teach teacher net dynamics))
       (propagate net (process-inputs dynamics))
       (set-kick dynamics (* 10.0 (first (get-outputs net))))
       (set-kick (pole-state-reference dynamics)
             (clascon (pole-state-reference dynamics) dynamics))))
    finally
      (if (and process-run-f duration-pane-update-f)
        (zl:selectq (plot-pane-plot-type duration-pane)
          (1 (update duration-pane lesson (* count (pole-state-dt dynamics))))
          (2 (update duration-pane (* count (pole-state-dt dynamics))
                 (- (pole-state-x dynamics) (pole-state-desired-x dynamics))))
          (3 (update duration-pane (* count (pole-state-dt dynamics))
                 (- (pole-state-x dynamics) (pole-state-desired-x dynamics))
                 (* count (pole-state-dt dynamics))
                 (- (pole-state-x (pole-state-reference dynamics))
                   (pole-state-desired-x (pole-state-reference dynamics))))))
        (if (not process-run-f) (setq suspend-f t))
        (send keyboard-io-pane :select)))))
```

EXPERIMENTS

```
(defmethod (create-experiment-menu pole-exec) ()
  (zl:selectq (mom-menu "Choose Experiment"
              '(nil
                "Theta and x Classical Controller"
                "Layered x - fpr - linear-initialization"
                "Layered x - fpr - nominal"
                "Layered x - fpr - linear big M"
                "Layered x - fpr - nominal big M"
                "Layered x - fpr - linear big L"
                "Layered x - fpr - nominal big L"
               nil))

   (1 (setq net
        (create-network
          '((1 input neuron2 (3.0)) (1 output neuron2 (5.0)))
         '((I-to-I 0 1))
          '((net nil nil uniform (0.0 0.3)))
          'propagate-feedforward))
    (setf (network-experiment-name net) "  Theta and x Classical Controller")
    (send network-pane :clear-window)
    (setq network-pane-update-f nil)
    (send pane-b :clear-window)
    (setq pane-b-update-f nil)
    (send phase-pane :clear-window)
    (setq phase-pane-update-f nil)
    (setf (pole-state-clascon-f dynamics) t))
```

```
(2 (setq net
      (create-network
        '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
        (1 output neuron2 (2.0)))
       '((l-to-l 0 1) (l-to-l 1 2))
        '((l-to-l-custom 0 1 nil ((0.548 0.0) (0.670 0.0)))
        (l-to-l-custom 1 2 nil ((1.216) (0.462))))
       'propagate-feedforward))
  (setf (network-experiment-name net) "     Layered x - fpr - linear-initialization")
  (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
  (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
  (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
  (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
  (setf (pole-state-theta-continuous-scale dynamics) 8.0)
  (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
  (setf (pole-state-x-continuous-scale dynamics) 0.30)
  (setf (pole-state-xd-continuous-scale dynamics) 0.90)
  (setf (pole-state-process-inputs-function dynamics)
    'layered-x)
  (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
  (setf (teacher-learning-coefficient teacher) 0.01)
  (setf (teacher-update-interval teacher) 5)
  (setf (teacher-teach-function teacher) 'fpr))

(3 (setq net
      (create-network
        '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
        (1 output neuron2 (2.0)))
       '((l-to-l 0 1) (l-to-l 1 2))
        '((l-to-l 0 1 uniform (0.0 0.3))
        (l-to-l 1 2 uniform (0.0 0.3)))
       'propagate-feedforward))
  (setf (network-experiment-name net) "     Layered x - fpr - nominal")
  (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
  (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
  (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
  (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
  (setf (pole-state-theta-continuous-scale dynamics) 8.0)
  (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
  (setf (pole-state-x-continuous-scale dynamics) 0.30)
  (setf (pole-state-xd-continuous-scale dynamics) 0.90)
  (setf (pole-state-process-inputs-function dynamics)
    'layered-x)
  (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
  (setf (teacher-learning-coefficient teacher) 0.01)
  (setf (teacher-update-interval teacher) 5)
```

```
 (setf (teacher-teach-function teacher) 'fpr))

(4 (setq net
      (create-network
        '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
         (1 output neuron2 (2.0)))
       '((l-to-l 0 1) (l-to-l 1 2))
        '((l-to-l-custom 0 1 nil ((0.548 0.0) (0.670 0.0)))
         (l-to-l-custom 1 2 nil ((1.216) (0.462))))
        'propagate-feedforward))
 (setf (network-experiment-name net) "    Layered x - fpr - linear big M")
 (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
 (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
 (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
 (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
 (setf (pole-state-theta-continuous-scale dynamics) 8.0)
 (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
 (setf (pole-state-x-continuous-scale dynamics) 0.30)
 (setf (pole-state-xd-continuous-scale dynamics) 0.90)
 (setf (pole-state-process-inputs-function dynamics)
    'layered-x)
 (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
 (setf (pole-state-mc dynamics) 3.0)
 (setf (teacher-learning-coefficient teacher) 0.01)
 (setf (teacher-update-interval teacher) 5)
 (setf (teacher-teach-function teacher) 'fpr))

(5 (setq net
      (create-network
        '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
         (1 output neuron2 (2.0)))
       '((l-to-l 0 1) (l-to-l 1 2))
        '((l-to-l 0 1 uniform (0.0 0.3))
         (l-to-l 1 2 uniform (0.0 0.3)))
        'propagate-feedforward))
 (setf (network-experiment-name net) "    Layered x - fpr - nominal big L")
 (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
 (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
 (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
 (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
 (setf (pole-state-theta-continuous-scale dynamics) 8.0)
 (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
 (setf (pole-state-x-continuous-scale dynamics) 0.30)
 (setf (pole-state-xd-continuous-scale dynamics) 0.90)
 (setf (pole-state-process-inputs-function dynamics)
    'layered-x)
```

```
  (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
  (setf (pole-state-mc dynamics) 3.0)
  (setf (teacher-learning-coefficient teacher) 0.01)
  (setf (teacher-halt-function teacher) 'halt1)
  (setf (teacher-update-interval teacher) 5)
  (setf (teacher-teach-function teacher) 'fpr))))

(6 (setq net
       (create-network
         '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
          (1 output neuron2 (2.0)))
        '((l-to-l 0 1) (l-to-l 1 2))
         '((l-to-l-custom 0 1 nil ((0.548 0.0) (0.670 0.0)))
          (l-to-l-custom 1 2 nil ((1.216) (0.462))))
         'propagate-feedforward))
  (setf (network-experiment-name net) "     Layered x - fpr - linear big L")
  (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
  (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
  (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
  (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
  (setf (pole-state-theta-continuous-scale dynamics) 8.0)
  (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
  (setf (pole-state-x-continuous-scale dynamics) 0.30)
  (setf (pole-state-xd-continuous-scale dynamics) 0.90)
  (setf (pole-state-process-inputs-function dynamics)
     'layered-x)
  (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
  (setf (pole-state-l dynamics) 1.5)
  (setf (teacher-learning-coefficient teacher) 0.01)
  (setf (teacher-update-interval teacher) 5)
  (setf (teacher-teach-function teacher) 'fpr))

(7 (setq net
       (create-network
         '((2 input neuron2 (2.0 1.0)) (2 hidden neuron2 (2.0 1.0))
          (1 output neuron2 (2.0)))
        '((l-to-l 0 1) (l-to-l 1 2))
        '((l-to-l 0 1 uniform (0.0 0.3))
         (l-to-l 1 2 uniform (0.0 0.3)))
        'propagate-feedforward))
  (setf (network-experiment-name net) "     Layered x - fpr - nominal big L")
  (setf (pole-state-x-neuron dynamics) (nth 0 (network-sequentials net)))
  (setf (pole-state-xd-neuron dynamics) (nth 1 (network-sequentials net)))
  (setf (pole-state-theta-neuron dynamics) (nth 2 (network-sequentials net)))
  (setf (pole-state-thetad-neuron dynamics) (nth 3 (network-sequentials net)))
  (setf (pole-state-theta-continuous-scale dynamics) 8.0)
```

```
    (setf (pole-state-thetad-continuous-scale dynamics) 4.0)
    (setf (pole-state-x-continuous-scale dynamics) 0.30)
    (setf (pole-state-xd-continuous-scale dynamics) 0.90)
    (setf (pole-state-process-inputs-function dynamics)
       'layered-x)
    (setf (pole-state-evaluate-function dynamics) 'evaluate-4ref)
    (setf (pole-state-l dynamics) 1.5)
    (setf (teacher-learning-coefficient teacher) 0.01)
    (setf (teacher-halt-function teacher) 'halt1)
    (setf (teacher-update-interval teacher) 5)
    (setf (teacher-teach-function teacher) 'fpr))))
```

A.2 FORPROP.LISP

Forprop defines all flavors (data structures) and methods (functions on data structures) related to neurons, networks, and teachers. The program contains routines to create, propagate (complete or one-step), and teach (forpropagation or backpropagation) the network

NEURON FLAVOR

```
(defflavor neuron
      (id-type
       (activator 'activate-summer)
       outputter
       (internal-state 0.0)
       (last-internal-state 0.0)
       (external-state 0.0)
       (last-external-state 0.0)
       (error 0.0)
       (last-error 0.0)
      lambda
       dynamic-sign
       (max-sigmoid-input 10.0)
       (min-sigmoid-input -10.0))
      (node)
  :initable-instance-variables
```

```
 :writable-instance-variables)

(defflavor neuron1
     ((outputter 'output-sigmoid1))
     (neuron)
 :initable-instance-variables
 :writable-instance-variables)

(defflavor neuron2
     ((outputter 'output-sigmoid2))
     (neuron)
 :initable-instance-variables
 :writable-instance-variables)

(defflavor neuron-linear
     ((outputter 'output-linear))
     (neuron)
 :initable-instance-variables
 :writable-instance-variables)
```

NEURON METHODS

```
(defmethod (round-x-state neuron1) ()
 (if (> external-state 0.5) 1 0))

(defmethod (round-x-state neuron2) ()
 (if (> external-state 0.0) 1 0))

(defmethod (set-ex-state neuron) (ex-state)
 (setf last-external-state external-state)
 (setf external-state ex-state))

(defmethod (set-error neuron) (new-error)
 (setf last-error error)
 (setf error new-error))
```

NEURON-TEACHERS

BACKPROPAGATION

```
(defmethod (teach-backprop-errors neuron) ()
 (setq error
      (* (deriv-output-sigmoid self)
           (loop for neuron-alist in output-alist
           for n = (first neuron-alist)
           for weight = (cdr neuron-alist)
           summing (* weight (neuron-error n))))))

(defmethod (teach-backprop-weights neuron) (from-neuron teacher)
 (let* ((old-weight (get-old-weight from-neuron self))
       (weight (get-weight from-neuron self))
       (last-weight-change (- weight old-weight)))
   (set-weight
    from-neuron self (+ old-weight
                   (+ (* (teacher-learning-coefficient teacher)
                        error (neuron-external-state from-neuron))
                      (* (teacher-momentum-coefficient teacher)
                      last-weight-change))))))
```

FORPROPAGATION

```
(defmethod (teach-forprop-errors neuron) ()
 (setq error
      (* (deriv-output-sigmoid self)
         (loop for neuron-alist in input-alist
             for n = (first neuron-alist)
             for weight = (cdr neuron-alist)
             summing (* weight (neuron-error n))))))

(defmethod (teach-forprop-weights neuron) (from-neuron teacher)
 (let* ((old-weight (get-old-weight from-neuron self))
       (weight (get-weight from-neuron self))
       (last-weight-change (- weight old-weight)))
   (set-weight
    from-neuron self (+ old-weight
                   (+ (* (teacher-learning-coefficient teacher)
                        (neuron-external-state from-neuron)
                       (neuron-error from-neuron))
                      (* (teacher-momentum-coefficient teacher)
                      last-weight-change))))))
```

FORPROPAGATION WITH REFERENCE (FPR)

```
(defmethod (teach-fpr-weights neuron) (to-neuron teacher)
 (let* ((old-weight (get-old-weight self to-neuron))
       (weight (get-weight self to-neuron))
       (last-weight-change (- weight old-weight)))
   (set-weight
     self to-neuron (+ old-weight
                  (+ (* (teacher-learning-coefficient teacher)
                      (deriv-output-sigmoid self)
                       (deriv-output-sigmoid to-neuron)
                   error)
                    (* (teacher-momentum-coefficient teacher)
                     last-weight-change))))))
```

NEURON-INITIALIZERS

```
(defmethod (initialize neuron) (arglist)
 (setf lambda (first arglist))
 (setf dynamic-sign (second arglist)))
```

NEURON-ACTIVATORS

```
(defmethod (activate-summer neuron) (&optional (ext-input 0.0))
 (let ((temp (+ (loop for input-neuron-alist in (node-input-alist self)
                summing (* (cdr input-neuron-alist)
                         (neuron-external-state (first input-neuron-alist))))
           ext-input)))
   (setf last-internal-state internal-state)
   (setf internal-state temp)))
```

NEURON-OUTPUTTERS

```
(defmethod (output neuron) ()
 (funcall outputter self))

(defmethod (output-sigmoid1 neuron) ()
 (setf last-external-state external-state)
 (setf external-state
     (/ 1.0 (+ 1.0 (exp (min max-sigmoid-input
                       (max min-sigmoid-input
                        (* -1.0 internal-state lambda)))))))

(defmethod (output-sigmoid2 neuron) ()
 (setf last-external-state external-state)
 (setf external-state
    (* 2.0
      (- (/ 1.0 (+ 1.0
               (exp (min max-sigmoid-input
                     (max min-sigmoid-input
                         (* -1.0 internal-state lambda)))))) 0.5))))

(defmethod (output-linear neuron) ()
 (setf last-external-state external-state)
 (setf external-state internal-state))
```

DERIVATIVE OUTPUT FUNCTIONS

```
(defmethod (deriv-output-sigmoid neuron) ()
 (funcall (intern (string-append 'deriv- (neuron-outputter self))) self))

(defmethod (deriv-output-sigmoid1 neuron) ()
 (* external-state (- 1.0 external-state)))

(defmethod (deriv-output-sigmoid2 neuron) ()
 (let ((val (exp (min max-sigmoid-input
               (max min-sigmoid-input
                  (* -1.0 internal-state lambda))))))
   (/ (* 2.0  lambda val) (square (+ 1.0 val)))))
```

NETWORK FLAVOR

```
(defflavor network
     ((inputs)              LIST OF NEURONS CONNECTED AS INPUTS
      (hiddens)             LIST OF INTERNAL NEURONS
      (outputs)             LIST OF NEURON CONNECTED AS OUTPUT
      (threshold)           ONE THRESHOLD NEURON
      (sequentials)         DIRECTED LIST OF ALL NEURONS
     (layers)               LIST OF LAYERS
      (teacher-list)        LIST OF TEACHER LOOPS
      creation-list         DEFINES STRUCTURE OF NET (NEURONS)
      connection-list       DEFINES CONNECTIONS OF NEURONS
      weight-init-list      DEFINES INITIAL CONNECTION WEIGHTS
     teacher-init-list              DEFINES HOW TO TEACH NET
     propagator                     FULL OR ONE-STEP PROPAGATION FUNCTION
     experiment-name)
   ()
 :initable-instance-variables
 :writable-instance-variables)
```

NETWORK METHODS

```
(defun create-network
     (creation-list connection-list weight-init-list propagator)
  (let ((net (make-instance 'network
                     :creation-list creation-list
                     :connection-list connection-list
                     :weight-init-list weight-init-list
                     :propagator propagator)))
   (initialize net)
   (connect net)
   (initialize-weights net)
   net))

(defun create-network-from-file (filename)
  (with-open-file (stream filename
                    :direction :input
                   :characters t)
   (create-network
    (read stream)
    (read stream)
    (read stream)
    (read stream)
    (read stream))))

(defmethod (initialize network) ()
  (loop for layer-process-list in creation-list
       for number-to-create = (first layer-process-list)
       for id-type = (second layer-process-list)
       for neuron-type = (third layer-process-list)
       for creation-args = (fourth layer-process-list)
       do (if (equal id-type 'threshold)
             (let ((new-neuron (make-instance neuron-type)))
              (initialize new-neuron creation-args)
              (setf (neuron-id-type new-neuron) id-type)
              (setf threshold new-neuron))
             (loop for n from 1 to number-to-create
                for new-neuron = (make-instance neuron-type)
                do (initialize new-neuron creation-args)
                  (setf (neuron-id-type new-neuron) id-type)
                 (setf sequentials
                      (append sequentials (list new-neuron)))
                collecting new-neuron into layer-list
                finally (let ((layer (create-layer layer-list)))
                      (setf layers (append layers (list layer)))
                      (cond ((equal id-type 'input)
                           (setf inputs layer-list))
                           ((equal id-type 'hidden)
                             (setf hiddens (append hiddens (list layer-list))))
                           ((equal id-type 'output)
                            (setf outputs layer-list)))))))

(defmethod (connect network) ()
  (loop for connection in connection-list
      for conn = (first connection)
      for from-num = (second connection)
      for to-num = (third connection)
      do (cond ((equal conn 'l-to-l)
               (connect (nth from-num layers) (nth to-num layers)))
              ((equal conn 't-to-net)
               (loop for neuron-to in sequentials
                   do (connect threshold neuron-to)))
              ((equal conn 'n-to-n)
               (connect (nth from-num sequentials) (nth to-num sequentials)))
              ((equal conn 'n-to-l)
               (connect-node-to-layer (nth from-num sequentials) (nth to-num layers)))
              ((equal conn 'l-to-n)
               (connect-layer-to-node (nth from-num layers) (nth to-num sequentials))))))

(defmethod (initialize-weights network) ()
  (loop for weight-init in weight-init-list
      for type = (first weight-init)
      for from-num = (second weight-init)
      for to-num = (third weight-init)
      for weight-func = (intern (string-append 'initialize-weights- (fourth weight-init)))
      for weight-func-args = (fifth weight-init)
      do (cond ((equal type 'net)
              (loop for neuron in sequentials
                  do (loop for output-neuron-alist in (node-output-alist neuron)
                         for output-neuron = (first output-neuron-alist)
                         do (set-weight neuron output-neuron
                                  (funcall weight-func weight-func-args) t))))
             ((equal type 'net-t)
                (loop for neuron in (append sequentials (list threshold))
                     do (loop for output-neuron-alist in (node-output-alist neuron)
                           for output-neuron = (first output-neuron-alist)
                           do (set-weight neuron output-neuron
                                   (funcall weight-func weight-func-args) t))))
               ((equal type 'net-t-custom)
                (loop for neuron in (append sequentials (list threshold))
                   for weight-list in weight-func-args
                     do (loop for output-neuron-alist in (node-output-alist neuron)
                           for output-neuron = (first output-neuron-alist)
                         for weight in weight-list
                           do (set-weight neuron output-neuron weight t))))
              ((equal type '1-to-l)
               (let ((from-layer (nth from-num layers))
                    (to-layer (nth to-num layers)))
                 (loop for from-neuron in (layer-nodes from-layer)
                     do (loop for to-neuron in (layer-nodes to-layer)
                           do (if (assoc to-neuron (node-output-alist from-neuron))
                               (set-weight
                                 from-neuron to-neuron
                                   (funcall weight-func weight-func-args) t))))))
               ((equal type '1-to-l-custom)
               (let ((from-layer (nth from-num layers))
                    (to-layer (nth to-num layers)))
                 (loop for from-neuron in (layer-nodes from-layer)
                     for weight-list in weight-func-args
                     do (loop for to-neuron in (layer-nodes to-layer)
                          for weight in weight-list
                            do (set-weight from-neuron to-neuron weight t)))))))
```

NETWORK PROPAGATORS

```
(defmethod (propagate network) (input-values)
  (funcall propagator self input-values))

(defmethod (propagate-feedforward network) (input-values)
  (if threshold (set-ex-state threshold 0.1))
  (loop for nlayer from 0
      for neuron-list in (append (list inputs) hiddens (list outputs))
      do (loop for nneuron from 0
             for neuron in neuron-list
             do (let ((ext-input (nth nneuron (nth nlayer input-values))))
               (if ext-input
                  (activate-summer neuron ext-input)
                  (activate-summer neuron))
                (output neuron)))))

(defmethod (propagate-feedforward-step network) (input-values)
  (if threshold (set-ex-state threshold 0.1))
  (loop for nlayer from (1- (length layers))
       for neuron-list in (append (list outputs) hiddens (list inputs))
       do (loop for nneuron from 0
              for neuron in neuron-list
              do (let ((ext-input (nth nneuron (nth nlayer input-values))))
                 (if ext-input
                    (activate-summer neuron ext-input)
                    (activate-summer neuron))
                 (output neuron)))))
```

NETWORK BACKPROP METHODS

```
(defmethod (teach-backprop-errors network) ()
  (loop for neuron-list in (zl:reverse hiddens)
       do (loop for neuron in neuron-list
              do (teach-backprop-errors neuron))))

(defmethod (teach-backprop-weights network) (teacher)
  (if threshold
    (loop for to-neuron-alist in (node-output-alist threshold)
         for to-neuron = (first to-neuron-alist)
         do (teach-backprop-weights to-neuron threshold teacher)))
  (loop for from-neuron-list in (append (list inputs) hiddens)
       do (loop for from-neuron in from-neuron-list
              do (loop for to-neuron-alist in (node-output-alist from-neuron)
                     for to-neuron = (first to-neuron-alist)
                     do (teach-backprop-weights to-neuron from-neuron teacher)))))
```

NETWORK FORPROP METHODS

```
(defmethod (teach-forprop-errors network) ()
  (loop for neuron-list in (append hiddens (list outputs))
       do (loop for neuron in neuron-list
              do (teach-forprop-errors neuron))))

(defmethod (teach-forprop-weights network) (teacher)
  (if threshold
   (loop for to-neuron-alist in (node-output-alist threshold)
        for to-neuron = (first to-neuron-alist)
         do (teach-forprop-weights to-neuron threshold teacher)))
 (loop for from-neuron-list in (append (list inputs) hiddens)
      do (loop for from-neuron in from-neuron-list
              do (loop for to-neuron-alist in (node-output-alist from-neuron)
                      for to-neuron = (first to-neuron-alist)
                       do (teach-forprop-weights to-neuron from-neuron teacher)))))
```

NETWORK FPR METHODS

```
(defmethod (teach-fpr-weights network) (teacher)
 (loop for neuron-list in (append (list inputs) hiddens)
      do (loop for from-neuron in neuron-list
              do (loop for to-neuron-alist in (node-output-alist from-neuron)
                      for to-neuron = (first to-neuron-alist)
                       do (teach-fpr-weights from-neuron to-neuron teacher)))))
```

GENERAL NETWORK METHODS

```
(defmethod (get-outputs network) ()
 (loop for neuron in outputs
      collecting (neuron-external-state neuron)))

(defmethod (get-rounded-outputs network) ()
 (loop for neuron in outputs
      collecting (round-x-state neuron)))

(defmethod (get-externals-of-layer network) (nlayer)
 (loop for neuron in (layer-nodes (nth nlayer layers))
      collecting (neuron-external-state neuron)))

(defmethod (get-internals-of-layer network) (nlayer)
 (loop for neuron in (layer-nodes (nth nlayer layers))
      collecting (neuron-internal-state neuron)))

(defmethod (get-errors-of-layer network) (nlayer)
 (loop for neuron in (layer-nodes (nth nlayer layers))
      collecting (neuron-error neuron)))

(defmethod (set-output-error-with-actual-desired network) (actual-values desired-values)
 (loop for output-neuron in outputs
    for actual-value in actual-values
    for desired-value in desired-values
    for x-state = (neuron-external-state output-neuron)
    do (set-error output-neuron (* (deriv-output-sigmoid output-neuron)
                       (- desired-value actual-value)))))

(defmethod (set-input-error-with-actual-desired network) (actual-values desired-values)
  (loop for input-neuron in inputs
    for actual-value in actual-values
    for desired-value in desired-values
    for x-state = (neuron-external-state input-neuron)
    do (set-error input-neuron (* (deriv-output-sigmoid input-neuron)
                       (- desired-value actual-value)))))

(defmethod (set-output-error-with-error network) (errors)
  (loop for output-neuron in outputs
    for error in errors
    do (setf (neuron-error output-neuron) (* (deriv-output-sigmoid output-neuron) error))))

(defmethod (set-input-error-with-error network) (errors)
  (loop for input-neuron in inputs
    for error in errors
    do (setf (neuron-error input-neuron) (* (deriv-output-sigmoid input-neuron) error))))

(defmethod (set-input-and-threshold-error-with-error network) (errors)
  (loop for input-neuron in (append inputs (list threshold))
    for error in errors
    do (setf (neuron-error input-neuron) (* (deriv-output-sigmoid input-neuron) error))))

(defmethod (calc network) (input)
  (propagate self input)
  (get-outputs self))
```

TEACHER FLAVOR

```
(defflavor teacher
    ((teach-function)
     (teacher-on-f t)
     (learning-coefficient 0.1)
     (momentum-coefficient 0.9)
     (update-interval 1)
     (update-count 0)
     (trip-function 'simple-trip)
     (halt-function 'no-halt))
```

```
   ()
  :initable-instance-variables
  :writable-instance-variables)
```

TEACHER METHODS

```
(defmethod (teach teacher) (network dynamics)
  (cond (teacher-on-f
        (setq update-count (1+ update-count))
        (if (= 0 (mod update-count update-interval))
          (if (not (funcall halt-function dynamics))
            (funcall teach-function self network))))))

(defmethod (backprop teacher) (network)
  (teach-backprop-errors network)
  (teach-backprop-weights network self))

(defmethod (forprop teacher) (network)
  (teach-forprop-errors network)
  (teach-forprop-weights network self))

(defmethod (fpr teacher) (network)
  (teach-fpr-weights network self))
```

TEACHER MENU

```
(defmethod (teacher-menu teacher) ()
  (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7|))
  (tv:choose-variable-values-locally
    '(("teacher-function" :choose
      (backprop forprop forprop-wts fpr bpr) teach-function)
     ("teacher on" :boolean teacher-on-f)
     ("learning coefficient" :number learning-coefficient)
     ("momentum coefficient" :number momentum-coefficient)
     ("update interval" :number update-interval)
     ("trip function" :choose (no-trip simple-trip trip1) trip-function)
     ("halt function" :choose (no-halt halt1) halt-function))
    "Teacher Variables          "))
```

WEIGHT INITIALIZATION FUNCTIONS

```
(defun initialize-weights-gaussian (arglist)
  (gaussian (first arglist) (second arglist)))

(defun initialize-weights-uniform (arglist)
  (si:random-in-range (first arglist) (second arglist)))

(defun initialize-weights-constant (arglist)
  (first arglist))
```

A.3 POLE-DYNAMICS.LISP

Pole-Dynamics uses simple numerical techniques to integrate the equations of motion of the cart-pole. This code also runs the classical controller and maintains the system reset responsibilities.

POLE-STATE FLAVOR

```
(defflavor pole-state
      ((reference (make-instance 'pole-state
                        :reference nil))
       (theta-wander-max 0.2)
       (thetad-wander-max 0.4)
       (x-wander-max 0.4)
       (xd-wander-max 0.6)

       (g -9.8)
       (mc 1.0)
       (mp 0.01)
       (l 0.5)
       (uc 0.0)
       (up 0.0)
       (dt 0.02)
       (time 0.0)

       (duration 0.0)
       (duration-history (make-array 10000 :element-type 'float :initial-element 0.0))
```

```
(lesson 0.0)
 (lesson-history (make-array 10000 :element-type 'float :initial-element 0.0))

(kick 0.0)
 (kick-history (make-array 10000 :element-type 'float :initial-element 0.0))
(last-kick 0.0)
(initial-kick 0.0)

(x 0.0)
 (x-history (make-array 10000 :element-type 'float :initial-element 0.0))
(last-x 0.0)
 (desired-x 0.0)
 (desired-x-history (make-array 10000 :element-type 'float :initial-element 0.0))

(xd 0.0)
 (xd-history (make-array 10000 :element-type 'float :initial-element 0.0))
(last-xd 0.0)
 (desired-xd 0.0)
 (desired-xd-history (make-array 10000 :element-type 'float :initial-element 0.0))

(xdd 0.0)
 (xdd-history (make-array 10000 :element-type 'float :initial-element 0.0))
(last-xdd 0.0)

(theta 0.0)
 (theta-history (make-array 10000 :element-type 'float :initial-element 0.0))
(last-theta 0.0)
 (desired-theta 0.0)
 (desired-theta-history (make-array 10000 :element-type 'float :initial-element 0.0))

 (thetad 0.0)
 (thetad-history (make-array 10000 :element-type 'float :initial-element 0.0))
 (last-thetad 0.0)
 (desired-thetad 0.0)
 (desired-thetad-history (make-array 10000 :element-type 'float :initial-element 0.0))

 (thetadd 0.0)
 (thetadd-history (make-array 10000 :element-type 'float :initial-element 0.0))
 (last-thetadd 0.0)

 (theta-continuous-scale 10.0)
 (thetad-continuous-scale 2.5)
 (x-continuous-scale 5.0)
 (xd-continuous-scale 10.0)

 (kick-noise-mag 3.0)
```

```
      (theta-noise-mag 0.01)
      (thetad-noise-mag 0.01)
      (x-noise-mag 0.02)
      (xd-noise-mag 0.02)
      (kick-noise-f)
      (theta-noise-f)
      (thetad-noise-f)
      (x-noise-f)
      (xd-noise-f)

      (theta-neuron)
      (thetad-neuron)
      (x-neuron)
      (xd-neuron)
      (layer-f)

      (clascon-f)
      (cc-theta-gain 69.3)
      (cc-thetad-gain 13.2)
      (cc-x-gain 2.0)
      (cc-xd-gain 4.9)
      (cc-max-kick 10.0)
      (cc-max-theta-com 0.1745)
      (time-fudge-f nil)

      (process-inputs-function)
      (process-reference-inputs-function)
      (filter-on)
      (evaluate-function)
      (show-stats-f nil)
      (write-stats-f nil)
      (write-stats-frequency 1)
      (draw-theta-c-f nil))
   ()
  :initable-instance-variables
  :writable-instance-variables)
```

POLE-STATE METHODS

```
(defun create-plant ()
  (make-instance 'pole-state))

(defmethod (update-x pole-state) ()
  (setq last-x x)
```

```
 (if filter-on (setq x (integrator xd last-xd last-x dt))
   (setq x (+ x (* xd dt)))))

(defmethod (update-xd pole-state) ()
 (setq last-xd xd)
 (if filter-on (setq xd (integrator xdd last-xdd last-xd dt))
   (setq xd (+ xd (* xdd dt)))))

(defmethod (update-theta pole-state) ()
 (setq last-theta theta)
 (if filter-on (setq theta (integrator thetad last-thetad last-theta dt))
   (setq theta (+ theta (* thetad dt)))))

(defmethod (update-thetad pole-state) ()
 (setq last-thetad thetad)
 (if filter-on (setq thetad (integrator thetadd last-thetadd last-thetad dt))
   (setq thetad (+ thetad (* thetadd dt)))))

(defmethod (update-xdd pole-state) ()
 (without-floating-underflow-traps
  (setq last-xdd xdd)
  (setq xdd (/ (+ kick
                (* mp l (- (* (square thetad) (sin theta)) (* thetadd (cos theta))))
               (- (* uc (sgn xd))))
             (+ mc mp)))))

(defmethod (update-thetadd pole-state) ()
 (without-floating-underflow-traps
  (setq last-thetadd thetadd)
  (setq thetadd  (/ (+ (- (* g (sin theta)))
                   (* (cos theta)
                    (/ (+ (- kick)
                          (- (* mp l (square theta) (sin theta)))
                         (* uc (sgn xd)))
                       (+ mc mp)))
                   (- (/ (* up thetad) mp l)))
                  (* l (- 1.333333 (/ (* mp (square (cos theta)))
                                   (+ mc mp))))))))

(defmethod (update-noise pole-state) ()
 (if kick-noise-f
   (setq kick (+ kick (- (random (* kick-noise-mag 2.0)) kick-noise-mag))))
 (if theta-noise-f
   (setq theta (+ theta (- (random (* theta-noise-mag 2.0)) theta-noise-mag))))
 (if thetad-noise-f
   (setq thetad (+ thetad (- (random (* thetad-noise-mag 2.0)) thetad-noise-mag))))
 (if x-noise-f
   (setq x (+ x (- (random (* x-noise-mag 2.0)) x-noise-mag))))
 (if xd-noise-f
   (setq xd (+ xd (- (random (* xd-noise-mag 2.0)) xd-noise-mag)))))

(defmethod (theta-com pole-state) ()
 (cond (layer-f
       (if (= time 0.0) 0.0
          (+ (neuron-internal-state theta-neuron) theta)))
      (clascon-f
       (max (- cc-max-theta-com)
           (min cc-max-theta-com
              (/ (+ (* cc-x-gain (- x desired-x))
                    (* cc-xd-gain xd))
                   (- cc-theta-gain)))))
      (t desired-theta)))

(defmethod (record-history pole-state) ()
 (let ((i (round (/ time dt))))
   (if (<= i 9999)
     (progn
       (setf (aref duration-history i) duration)
       (setf (aref lesson-history i) lesson)
       (setf (aref kick-history i) kick)
       (setf (aref x-history i) x)
       (setf (aref desired-x-history i) desired-x)
       (setf (aref xd-history i) xd)
       (setf (aref desired-xd-history i) desired-xd)
       (setf (aref xdd-history i) xdd)
       (setf (aref theta-history i) theta)
       (setf (aref desired-theta-history i) desired-theta)
       (setf (aref thetad-history i) thetad)
       (setf (aref desired-thetad-history i) desired-thetad)
       (setf (aref thetadd-history i) thetadd)))))

(defmethod (update pole-state) (&optional dynamics)
 (record-history self)
 (setq time (+ time dt))
 (update-noise self)
 ;; This order is very important!!
  (update-x self)
 (update-xd self)
 (update-theta self)
 (update-thetad self)
 (update-thetadd self)
 (update-xdd self)
 (if dynamics (setf desired-x (pole-state-desired-x dynamics))))

(defmethod (balance pole-state) ()
 (setq x 0.0)
 (setq last-x 0.0)
 (setq xd 0.0)
 (setq last-xd 0.0)
 (setq xdd 0.0)
 (setq last-xdd 0.0)
 (setq theta 0.0)
 (setq last-theta 0.0)
 (setq thetad 0.0)
 (setq last-thetad 0.0)
 (setq thetadd 0.0)
 (setq last-thetadd 0.0)
 (setq kick 0.0)
 (setq last-kick 0.0))
```

CLASSICAL CONTROLLER

```
(defmethod (clascon pole-state) (&optional dynamics)
 (max (- cc-max-kick)
     (min cc-max-kick
         (+ (* cc-theta-gain theta)
           (* cc-thetad-gain thetad)
           (max (- (* cc-theta-gain cc-max-theta-com))
               (min (* cc-theta-gain cc-max-theta-com)
                 (+ (* cc-x-gain (- x
                              (if dynamics
                                  (pole-state-desired-x dynamics)
                                desired-x)))
                   (* cc-xd-gain xd))))))))
```

PROCESS INPUTS FUNCTIONS

```
(defmethod (process-inputs pole-state) (&optional args)
 (funcall process-inputs-function self args))

(defmethod (continuous-theta-thetad pole-state) (&optional args)
 (if args
   (let ((theta-in (first args))
       (thetad-in (second args)))
```

```
     (list (list (* theta-in theta-continuous-scale) (* thetad-in thetad-continuous-scale))))
    (list (list (* (- theta desired-theta) theta-continuous-scale)
       (* (- thetad desired-thetad) thetad-continuous-scale)))))

(defmethod (continuous-theta-thetad-x-xd pole-state)
         (&optional args)
 (if args
    (let ((theta-in (first args))
        (thetad-in (second args)))
     (list (list (* theta-in theta-continuous-scale)
        (* thetad-in thetad-continuous-scale) 0.0 0.0)))
    (list (list (* theta theta-continuous-scale)
       (* thetad thetad-continuous-scale)
       (* x x-continuous-scale)
       (* xd xd-continuous-scale)))))

(defmethod (layered-x pole-state)
         (&optional args)
 (if args
    (let ((theta-in (first args))
        (thetad-in (second args)))
     (list (list 0.0 0.0)
        (list (* theta-in theta-continuous-scale)
           (* thetad-in thetad-continuous-scale))))
    (list (list      (* (- x desired-x) x-continuous-scale)
          (* (- xd desired-xd) xd-continuous-scale))
       (list (* theta theta-continuous-scale)
          (* thetad thetad-continuous-scale)))))
```

TEACHER TRIP FUNCTIONS

```
(defmethod (no-trip pole-state) ()
 nil)

(defmethod (simple-trip pole-state) ()
 (or
   (> (abs theta) theta-abs)
   (> (abs thetad) thetad-abs)))

(defmethod (trip1 pole-state) ()
 (or
   (> (abs theta) theta-abs)
   (> (abs thetad) thetad-abs)
   (> (abs (- theta (pole-state-theta reference))) theta-wander-max)
```

```
  (> (abs (- thetad (pole-state-thetad reference))) thetad-wander-max)
  (> (abs (- x (pole-state-x reference))) x-wander-max)
  (> (abs (- xd (pole-state-xd reference))) xd-wander-max)))
```

TEACHER HALT FUNCTIONS

```
(defmethod (no-halt pole-state) ()
 nil)

(defmethod (halt1 pole-state) ()
 (or
  (< (abs (- x (pole-state-x reference))) 0.2)
  (< (abs (- xd (pole-state-xd reference))) 0.2)))
```

POLE-STATE MENUS

```
(defmethod (noise-menu pole-state) ()
 (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7| |t8| |t9| |t10|))
 (tv:choose-variable-values-locally
  '(
   ("theta noise on ? " :boolean theta-noise-f)
   ("theta noise magnitude" :number theta-noise-mag)
   ("thetad noise on ? " :boolean thetad-noise-f)
   ("thetad noise magnitude" :number thetad-noise-mag)
   ("x noise on ? " :boolean x-noise-f)
   ("x noise magnitude" :number x-noise-mag)
   ("xd noise on ? " :boolean xd-noise-f)
   ("xd noise magnitude" :number xd-noise-mag)
   ("kick noise on ? " :boolean kick-noise-f)
   ("kick noise magnitude" :number kick-noise-mag))
  "Noise Parameters              "))

(defmethod (state-variables-menu pole-state) ()
 (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7| |t8| |t9| |t10|))
 (tv:choose-variable-values-locally
  '(("theta" :number theta)
   ("desired-theta" :number desired-theta)
   ("theta-d" :number thetad)
   ("theta-dd" :number thetadd)
   ("x" :number x)
   ("desired-x" :number desired-x)
   ("x-d" :number xd)
```

```
    ("x-dd" :number xdd)
    ("kick" :number kick))
   "State Variables        "))

(defmethod (dynamic-parameters-menu pole-state) ()
  (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7| |t8| |t9| |t10| |t11| |t12| |t13| |t14|))
  (tv:choose-variable-values-locally
   '(("gravity" :number g)
    ("mass of cart" :number mc)
    ("mass of pole" :number mp)
    ("length of pole" :number l)
    ("friction cart on track" :number uc)
    ("friction pole on cart" :number up)
    ("initial kick on cart" :number initial-kick)
    ("integration step" :number dt)
    ("filter on?" :boolean filter-on)
    ("elapsed time" :number time)
    ("theta step multiplier" :number theta-step-mult)
    ("max theta step" :number max-theta-step)
    ("x step multiplier" :number x-step-mult)
    ("max x step" :number max-x-step))
   "Dynamic Parameters              "))

(defmethod (command-values-menu pole-state) ()
  (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7| |t8| |t9| |t10|))
  (tv:choose-variable-values-locally
   '(("desired-theta" :number desired-theta)
    ("desired-x" :number desired-x))
   "Command Values        "))

(defmethod (clascon-menu pole-state) ()
  (declare (special |t1| |t2| |t3| |t4| |t5| |t6| |t7| |t8| |t9| |t10|))
  (tv:choose-variable-values-locally
   '(("clascon running" :boolean clascon-f)
    ("theta gain" :number cc-theta-gain)
    ("thetad gain" :number cc-thetad-gain)
    ("x gain" :number cc-x-gain)
    ("xd gain" :number cc-xd-gain)
    ("max kick" :number cc-max-kick)
    ("max theta command" :number cc-max-theta-com))
```

REFERENCES

[1] Athans, M. and F.C. Schweppe, "Gradient Matrices and Matrix Calculations," Technical Note 1965-53, Lincoln Laboratory, Lexington, MA, November 17, 1965.

[2] Baker, Walter L., "State-Space Formulation of a Neural Network," Personal notes, C. S. Draper Laboratory, Cambridge, MA, February 9, 1988.

[3] Barto, Andrew G. and Richard S. Sutton, and Peter S. Brouwer, "Associative Search Network: A Reinforcement Learning Associative Memory," *Biological Cybernetics*, Vol. 40, pp. 201-211, 1981.

[4] Barto, Andrew G. and Richard S. Sutton, "Landmark Learning: An Illustration of Associative Search," *Biological Cybernetics*, Vol. 42, pp. 1-8, 1981.

[5] Barto, Andrew G., "Adaptive Neural Networks for Learning Control: Some Computational Experiments," *Proceedings of the IEEE Workshop on Intelligent Control*, 1985.

[6] Brewer, John W., "Kroneker Products and Matrix Calculus in System Theory," IEEE Transactions on Circuits and Systems, Vol. CAS-25, No. 9, September 1978.

[7] Coleman, Bernard D. and Victor J. Mizel, "Generalization of the Perceptron Convergence Theorem," *Brain Theory Newsletter*, Vol. 1, No. 4, May, 1976.

[8] Friedland, Bernard, *Control System Design*, McGraw-Hill Inc., New York, NY, 1986.

[9] Gelb, Arthur (ed), *Applied Optimal Estimation*, The MIT Press, Massachusetts Institute of Technology, Cambridge, MA, 1974.

[10] Hall, Steven Ray, "A Failure Detection Algorithm for Linear Dynamic Systems," Ph.D Thesis, Massachusetts Institute of Technology, June 1985.

[11] Hopfield, J. J. and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, Vol. 52, pp 141-152, 1985.

[12] Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of the Sciences USA*, Vol. 79, pp 2554-2558, April 1982.

[13] Hopfield, J. J., "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons," *Proceedings of the National Academy of the Sciences USA*, Vol. 81, pp 3088-3092, May 1984.

[14] Hopfield, John J. and David W. Tank, "Computing with Neural Circuits: A Model," *Science*, Vol. 233, pp. 625-633, August 8, 1986.

[15] Jorgensen, Chuck and Chris Matheus, "Catching Knowledge in Neural Nets," *AI Expert*, December 1986.

[16] Kailath, Thomas, *Linear Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1980.

[17] Kandel and Schwartz, *Principles of Neural Science 2 ed.*, Elsevier Science Publishing Co., 1985.

[18] Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, Num. 4598, May 13, 1983.

[19] Lee, Y. W., *Statistical Theory of Communication*, John Wiley and Sons, New York, NY, 1960.

[20] Levy, Bernard C. and Milton B. Adams, "Global Optimization with Stochastic Neural Networks," presented at IEEE First International Conference on Neural Networks, San Diego, CA, June 21-24, 1987.

[21] Lippman, R. P., "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, Vol.4, No. 2, pp 4-22, April 1987.

[22] Minsky, M. and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA, 1969.

[23] Ogata, Katshuhiko, *Modern Control Engineering*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1970.

[24] Palm, G., "On Representation and Approximation of Nonlinear Systems," *Biological Cybernetics*, Vol. 31, pp. 119-124, 1978.

[25] Pao, Yoh-Han, "A Connectionist-Net Approach to Autonomous Machine Learning of Effective Process Control Strategies," Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1987.

[26] Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing. Vol. 1: Foundations, Vol. 2: Psychological and Biological Models*, MIT Press, Cambridge, MA, 1986.

[27] Sejnowski, T. J. and C. R. Rosenberg, "Parallel Networks That Learn to Pronounce English Text," *Complex Systems*, Vol. 1, pp. 145-168, 1987.

[28] Sejnowski, Terrance J. and Charles R. Rosenberg, "NETtalk: A Parallel Network that Learns to Read Aloud," The Johns Hopkins University Electrical Engineering and Computer Science Department, Technical Report JHU/EECS-86/01, Baltimore, MD, 1986.

[29] Simmons, George F., *Differential Equations*, McGraw-Hill Inc., New York, NY, 1972.

[30] Stevens, Charles S., "The Neuron," *Scientific American*, September 1979.

[31] Strang, Gilbert, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, MA, 1986.

[32] Sutton, Richard S. and Brian Pinette, "The Learning of World Models by Connectionist Networks," *Proceedings of the 7th Annual Conference on Cognitive Science*, August 1985.

[33] Sutton, Richard S., "Temporal Credit Assignment in Reinforcement Learning," Ph.D Thesis, University of Massachusetts at Amherst, February 1984.

[34] Thomas, George B. Jr. and Ross L. Finney, *Calculus and Analytic Geometry*, Addison-Wesley Publising Co., Reading, MA, May 1982.