In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

Comparison of Adaptive Neural Network Controllers for a Non-Linear Robotic Manipulator

by

Ian Showalter

B. Eng

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

June, 2003

© Copyright
2003, Ian Showalter

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88896-7

Our file Notre référence

ISBN: 0-612-88896-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

Abstract

This thesis presents a comparison of four previously presented neural network based control strategies for the trajectory control of robot manipulators. The neural networks learn the inverse dynamics of a robotic manipulator without any a priori knowledge of the manipulator inertial parameters or any a priori knowledge of the equation of dynamics. The first neural network type is a two step feedback error learning process that does not learn on line, the second is of similar type but has a small stack of previous input output pairs that are used to train the network on-line. The third is an on-line learning CMAC type, and the fourth is a statistical approximation type that also learns on-line. Training strategies and difficulties with on-line training are discussed. Simulation of a two degree of freedom serial link manipulator allows comparison of the effectiveness of the algorithms. Results show various performances ranging from effective to very good.

Keywords

Neural Network, Neural Control, CMAC, History Stack Algorithm, Statistical Approximation, Simultaneous Perturbation, Back Propagation.

Acknowledgments

I would like to start by expressing my sincere gratitude to my supervisor, Professor Schwartz for his time, generosity, patience, and faith in my work. Thank you for making research and studies in adaptive control so fascinating. Your assistance is greatly appreciated. Thank you also to Professor Aitken for always being available to help, and for your patience during the infinite times when the lab was covered in robot bits.

I would also like to thank my friends who have been supportive and encouraging, Veronique, Steve, Bevan, Brian, &c. Thank you for your help during times of panic, code ignorance. My friends at Mike's place, thank you for your patience during periods of insufficient sleep and extended grumpiness.

I want to thank Veronique specially for introducing me to the fabulous LyX software ($\text{L}\text{A}\text{T}\text{E}\text{X}$ front-end) which allowed me to write this thesis without having to spend months learning all the $\text{L}\text{A}\text{T}\text{E}\text{X}$ code, and without having to spend endless nights cursing certain commercial word processing packages and their respective designers (and equation inabilities, and crash-abilities). Thank you to all that have helped thesis writers everywhere by producing the wonderful free-ware/share-ware packages; MikTeX, Cygwin, Ghostscript, and of course LyX .

Finally, I want to thank Melissa and everyone at Motion Control Systems for their help in repairing the robot electrical, and electronics, and Fawad for his help in rebuilding the robot.

Contents

1	Introduction	1
2	The Neural Networks	4
2.1	Neural Network Basics	4
2.1.1	Multi-layer Neural Networks	5
2.1.2	The Neural Network Training Algorithm [2]	6
2.2	History Stack Algorithm (HSA)	11
2.3	Cerebral Model Arithmetic Controller (CMAC)	13
2.4	Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)	17
3	Neural Network Pre-Training	20
3.1	Problems With On-Line (Sequential) Training	20
3.2	Neural Network Pre-Training	29
3.2.1	Stage One Pre-Training	30
3.2.2	Stage Two Pre-Training	32
3.3	The Target System	32
3.4	Simulation And Pre-Training Of The Individual Neural Networks . .	35
3.4.1	The PD Controller	36
3.4.2	Non-Adaptive Neural Network	39
3.4.3	History Stack Algorithm (HSA)	45
3.4.4	Cerebral Model Arithmetic Controller (CMAC)	45
3.4.5	Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)	54

3.5	Non-Adaptive Comparison Results	54
4	Results Of Simulation With Adaptation	56
4.1	Results of Simulation With Gripper Arm Mass Twice Original Value .	57
4.1.1	The PD Controller	57
4.1.2	Non-Adaptive Neural Network	60
4.1.3	History Stack Algorithm (HSA)	63
4.1.4	Cerebral Model Arithmetic Controller (CMAC)	63
4.1.5	Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)	68
4.2	Results Of Simulation With Gripper Arm Mass Five Times Original Value	76
4.2.1	The PD Controller	76
4.2.2	Non-Adaptive Neural Network	76
4.2.3	History Stack Algorithm (HSA)	81
4.2.4	Cerebral Model Arithmetic Controller (CMAC)	86
4.2.5	Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)	86
4.3	Adaptive (Changed System) Comparison Results	86
5	Conclusions	97
5.1	Recommendations For Future Work	99

List of Figures

2.1	Structure Of A Basic Neuron (Perceptron).	5
2.2	Multi-layer Neural Network Architecture.	6
2.3	A Simple Example Of A Multi-layer Neural Network.	7
2.4	Block Diagram of the CMAC Neural Network [5]	14
2.5	SPSA Neural Network Flowchart.	19
3.1	The Controller Model for each Manipulator Joint [2].	21
3.2	Desired Position and Actual Position of Sequentially On-Line Trained NANN.	22
3.3	Neural Network and PD Controller Output of Sequentially On-Line Trained NANN.	23
3.4	A Neural Network Output Weight of Sequentially On-Line Trained NANN.	24
3.5	Non-pre-trained CMAC Desired and Actual Position.	26
3.6	Non-pre-trained SPSA Desired and Actual Position.	27
3.7	Non-pre-trained HSA Desired and Actual Position.	28
3.8	Stage One Method Of Training [2].	31
3.9	Stage Two Method Of Training [2].	33
3.10	Schematic Of The Two Degree Of Freedom Robot, Used For The Sim- ulations.	34
3.11	PD Controller Block Diagram.	36
3.12	PD Controller Desired and Actual Position.	37
3.13	PD Controller Trajectory Error.	38

3.14 NANN Desired and Actual Position After Stage One Training.	40
3.15 NANN Trajectory Error After Stage One Training.	41
3.16 NANN Desired and Actual Position After Stage Two Training.	42
3.17 NANN Trajectory Error After Stage Two Training (Note Angle Scaling x10 ⁻³).	43
3.18 NANN Output Versus PD Controller Output.	44
3.19 CMAC Desired and Actual Position After Stage One Training.	48
3.20 CMAC Trajectory Error After Stage One Training.	49
3.21 CMAC Neural Network Output Versus PD Controller Output After Stage One Training.	50
3.22 CMAC Desired and Actual Position After Stage Two Training.	51
3.23 CMAC Trajectory Error After Stage Two Training.	52
3.24 CMAC Neural Network Output Versus PD Controller Output After Stage Two Training.	53
4.1 PD Controller Desired and Actual Position of Changed System (Mass x2).	58
4.2 PD Controller Trajectory Error of Changed System (Mass x2).	59
4.3 NANN Desired and Actual Position of Changed System (Mass x2).	60
4.4 NANN Trajectory Error of Changed System (Mass x2).	61
4.5 NANN Neural Network Output Versus PD Controller Output of Changed System (Mass x2).	62
4.6 HSA Desired and Actual Position of Changed System (Mass x2).	64
4.7 HSA Trajectory Error of Changed System (Mass x2).	65
4.8 HSA Neural Network Output Versus PD Controller Output During On-Line Training of Changed System (Mass x2).	66
4.9 HSA RMS Error of Changed System (Mass x2).	67
4.10 CMAC Desired and Actual Position of Changed System (Mass x2).	68
4.11 CMAC Trajectory Error of Changed System (Mass x2).	69
4.12 CMAC Neural Network Output Versus PD Controller Output of Changed System (Mass x2).	70

4.13 CMAC RMS Error of Changed System (Mass x2).	71
4.14 SPSA Desired and Actual Position of Changed System (Mass x2). . .	72
4.15 SPSA Trajectory Error of Changed System (Mass x2).	73
4.16 SPSA Neural Network Output Versus PD Controller Output of Changed System (Mass x2)	74
4.17 SPSA RMS Error of Changed System (Mass x2).	75
4.18 PD Controller Desired and Actual Position of Changed System (Mass x5).	77
4.19 PD Controller Trajectory Error of Changed System (Mass x5).	78
4.20 NANN Desired and Actual Position of Changed System (Mass x5). . .	79
4.21 NANN Trajectory Error of Changed System (Mass x5).	80
4.22 NANN Neural Network Output Versus PD Controller Output of Changed System (Mass x5).	81
4.23 HSA Desired and Actual Position of Changed System (Mass x5). . . .	82
4.24 HSA Trajectory Error of Changed System (Mass x5).	83
4.25 HSA Neural Network Output Versus PD Controller Output During On-Line Training of Changed System (Mass x5).	84
4.26 HSA RMS Error of Changed System (Mass x5).	85
4.27 CMAC Desired and Actual Position of Changed System (Mass x5). . .	87
4.28 CMAC Trajectory Error of Changed System (Mass x5).	88
4.29 CMAC Neural Network Output Versus PD Controller Output of Changed System (Mass x5).	89
4.30 CMAC RMS Error of Changed System (Mass x5).	90
4.31 SPSA Desired and Actual Position of Changed System (Mass x5). . .	91
4.32 SPSA Trajectory Error of Changed System (Mass x5).	92
4.33 SPSA Neural Network Output Versus PD Controller Output of Changed System (Mass x5).	93
4.34 SPSA RMS Error of Changed System (Mass x5).	94

List of Tables

4.1	Changed System Comparison Results.	95
-----	--	----

Chapter 1

Introduction

Artificial neural networks have been used for modeling and control of non-linear systems over the last few decades with some success. They are among the most effective tools available for modeling or controlling such systems without requiring linearization of the controlled system. Artificial neural networks promise many advantages over conventional modeling or control. Several algorithms have been proposed to overcome the inability of neural networks to adapt on-line, but there has so far been no comparison of effectiveness or efficiency between these different algorithms. This thesis presents a comparison of four neural network based control strategies, three of which are capable of on-line training and adaptation. The above strategies are compared based on fastest training (convergence of weights to optimal values), least overshoot and steady-state error, and lastly computational time for each iteration/time-step. Here, a time-step refers to the time between each sampling of the manipulator position. The neural networks in this thesis are used to control the target system based on the desired trajectory, they are not used to generate the desired trajectory, and hence trajectory is used in this thesis to refer to the angular position of the actuators.

The History Stack Algorithm (HSA) [3] is an adaptation of a standard back-propagation neural network. A small stack of past input-output pair data is kept to train the neural network between time-steps while it is controlling the manipulator on-line. This FIFO stack of past data is updated every time-step, with the oldest pattern being discarded and a new pattern added. Each pattern that is in the stack

is selected at random between each time-step to train the neural network and update the weights. These input-output pair patterns represent an approximation of the manipulator transfer function, and allow the neural network to adapt to plant changes in real-time. Applying the supervised learning rule to all the patterns in the stack at each time-step, causes the error in the network weight vector to be minimized, corresponding to the mapping represented in the History Stack patterns. This algorithm uses each pattern several times before it is discarded, improving potential learning performance at each time-step, but increasing the controller's computational load. This controller requires pre-training before it can effectively control the target system.

The Cerebral Model Arithmetic/Articulated Controller (CMAC) [5] originally described by James Albus is a result of what was known of the cerebellum at the time. It differs from the other neural networks presented in this thesis, in that the three weight layers of this network are not represented by two-dimensional matrices, as they are in the others, instead the first layer quantizes the inputs, which are then used as a coordinate index value into the second layer matrix (which has as many dimensions as there are inputs). A neighbourhood constant is used to select a ten dimensional region of weights around the coordinate index. The third layer simply sums the second layer weights to produce the neural network output. CMAC algorithms typically take many fewer iterations to converge than back-propagation algorithms, and do not require a pre-training period prior to operation as a controller. CMAC algorithms provide these advantages at the expense of requiring massive use of memory to store the large second layer weight matrices.

The Simultaneous Perturbation Stochastic Approximation (SPSA) [11] neural network uses a stochastic approximation by simultaneous perturbation to minimize the weight error. It is feed-forward, back-propagation learning type similar to those presented above. For the first time period (a time period being several time-steps), the neural network output is generated by feed-forward propagation. For the second time period, the output is by feed-forward propagation with a zero-mean small random number having been added to each weight (simultaneous perturbation), subtraction

of the latter from the former provides weight update values. This process is repeated constantly during operation. The derivative of the error function is therefore not required, only values of the error itself. This controller does not necessarily have to be pre-trained before operating adaptively.

The Non-Adaptive Neural Network which is used for comparison purposes is a two-stage feedback error learning process that does not learn on-line. This neural network is a three layer feed-forward network with a two-step feedback error learning rule. This neural network cannot be trained using data sequentially in time, because this causes the weights to oscillate in phase with the desired trajectory, and they will therefore not converge to the desired values. By randomly selecting samples of the data that was collected on-line to train the neural network, this problem is resolved. This makes it difficult to train the neural network on-line, which would allow it to adapt to a changing environment. This neural network was simulated to provide an example of a non-adaptive neural network for comparison with the adaptive networks.

These algorithms will also be compared to a simple PD controller. The goal for these neural networks is to become a controller for a robotic manipulator. The neural network is to learn the inverse dynamics of the manipulator, and then its output is to be used as an inverse dynamics controller in combination with a PD controller for said manipulator. During the first stage of training, the neural network learns the output of the PD controller. In the second stage of training, the neural network operates in parallel with the PD controller, and is further trained by the output of the combined controller. Stage two is repeated until the neural network has achieved the desired accuracy. The target system's equations of inertial parameters and the equation of motion are assumed to be unknown for neural network training. Assumptions must be made about the target system; the complexity of the target system must be known in order to determine the number of weights for each layer of the neural networks.

Simulation of a two degree of freedom serial link manipulator allows comparison of the effectiveness of these algorithms. Results show performances ranging from effective to exceptional.

Chapter 2

The Neural Networks

This section first presents the concept of a neuron (perceptron), which is the fundamental building block of a neural network, then describes one of the most commonly used multi-layer neural network training methods, and finally presents the details of the adaptation algorithms that are used to train the networks while they are controlling the system on-line.

2.1 Neural Network Basics

The human nervous system is an interconnection of many millions of neuron cells, through which the signals of sensory perception pass back and forth to the brain. The Artificial Neural Network is an attempt at a mathematical conceptualization of these cells, their connections and interactions. A mathematical model of a single neuron and the synapses connected to this neuron is called a perceptron. The structure of a perceptron is presented in Figure 2.1.

Each perceptron accepts several input signals that are either signals from another perceptron, or input signals to the neural network proper. The input signals are multiplied by a weight and summed to provide a trigger to the neuron itself. If an input signal is positive, the connection is excitatory, if negative it is inhibitory. The summation of the input signals is used as input to the threshold procedure (or activation function), which is typically a non-linear function such as the sigmoid

Figure 2.1: Structure Of A Basic Neuron (Perceptron).

function $f(x) = \frac{1}{1+\exp(-x)}$. The output of the threshold procedure provides the activation of the perceptron. Perceptrons are typically interconnected in several layers to create a structure known as a neural network. The specific neural network structure is chosen based on its target application. Typically neurons contain a bias term that is added to the output before the activation function. These bias terms have been omitted in the neural networks presented in this thesis.

2.1.1 Multi-layer Neural Networks

Typically Neural Networks are constructed of several layers of perceptrons. The first layer accepts as its input, the input to the neural network. The remaining layer's of perceptrons each accept as input the output of the previous layers perceptrons. The output of the last layer is the output of the neural network. The first layer is commonly referred to as the input layer, and the last layer is called the output layer. Layers in between the input and output layers are known as the hidden layers. Figure 2.2 shows the architecture of a multi-layer neural network.

The weights that represent the connections between layers of neurons (perceptrons) in a multi-layer neural network can be viewed as parameter matrices. These parameters are the same as those of the system that is to be identified, or in this

Figure 2.2: Multi-layer Neural Network Architecture.

case the inverse of the system parameters as they are being used to control the system. Multi-layer neural networks are non-linear systems that are well suited for the identification of systems with unknown characteristics.

2.1.2 The Neural Network Training Algorithm [2]

The procedure for finding the best possible weight values for a neural network is called the learning rule. One of the most commonly used learning rules for multi-layer neural networks is the back-propagation rule or delta rule. The name back-propagation is used to describe this learning rule because there are two flows through the network weights. Feed-forward propagation is used to find the neural network output, and propagation backwards (back-propagation) through the weights is used to train them. To train the network, first the inputs are propagated forward through the network to find the output. The output is then compared to the desired output giving the error in output. This error is then used to update the output layers weights. The method for updating the output weights is different from that used to update the hidden layer weights.

The following is the mathematical procedure for updating both the output and hidden layer weights of a back-propagation neural network. For an example, a network

Figure 2.3: A Simple Example Of A Multi-layer Neural Network.

with three nodes in the input layer, a hidden layer of two nodes, and an output layer of one node will be examined. This network is presented in Figure 2.3.

Given the input vector $X_p = [X_{p1}, X_{p2}, \dots, X_{pn}]^T$, the total input to the j^{th} node in the first hidden layer is,

$$net_{pj}^h = \sum_{i=1}^n w_{ji}^h X_{pi} \quad (2.1)$$

where w_{ji}^h is the weight of the i^{th} node to the j^{th} node of the hidden layer, X_{pi} is the input to the i^{th} node from the p^{th} set of the input vectors to the perceptrons in the hidden layer, and h denotes the hidden layer. For the neural network defined for our example, input to the sigmoid function, denoted by γ on the node in Figure 2.3, to one in the hidden layer is,

$$net_{p1}^h = w_{11}^h X_{p1} + w_{12}^h X_{p2} + w_{13}^h X_{p3} \quad (2.2)$$

The output of the j^{th} hidden layer is then,

$$i_{pk} = f(net_{pj}^h) \quad (2.3)$$

where $f(.)$ is the sigmoid function in the cases presented here (or in general the activation function), and the total input to the k^{th} output node is,

$$net_{pk}^o = \sum_{j=1}^m w_{kj}^o i_{pj} \quad (2.4)$$

Here, m denotes the number of nodes in the hidden layer. For our example network, the input to the sigmoid function of the output layer node is,

$$net_{pi}^o = w_{11}^o i_{p1} + w_{12}^o i_{p2} \quad (2.5)$$

the superscript o referring to the output layer, and the output of the k^{th} node is,

$$O_{pk} = f(net_{pk}^o) \quad (2.6)$$

The total error is then the summation of the error of each node of the output layer, and the error on each output layer node is,

$$\delta_{pk} = (y_{pk} - O_{pk}) \quad (2.7)$$

where p refers to the p^{th} training vector, k is the k^{th} node of the output layer, O_{pk} is the output at that node, y_{pk} is the desired output, and the error is,

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 \quad (2.8)$$

The symbol r denotes the number of nodes in the output layer. The direction in which the weight must change is the gradient of E_p calculated with respect to the

changing weight. Therefore,

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 = \frac{1}{2} \sum_{k=1}^r (y_{pk} - O_{pk})^2 \quad (2.9)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(\text{net}_{pk}^o))^2 \quad (2.10)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(\sum_{j=1}^m w_{kj}^o i_{pk}))^2 \quad (2.11)$$

The partial derivative of E_p with respect to the weights is calculated using the calculus chain rule giving,

$$-\frac{\partial E_p}{\partial w_{kj}^o} = (y_{pk} - O_{pk}) f'_k(\text{net}_{pk}^o) i_{pk} \quad (2.12)$$

And hence to update the weights of the output layer, one can use the following,

$$w_{pk}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - O_{pk}) f'_k(\text{net}_{pk}^o) i_{pk} \quad (2.13)$$

where the factor η is the learning parameter or rate, the activation function is assumed to be sigmoidal, and the derivative of the sigmoid function is,

$$f' = f(1 - f) \quad (2.14)$$

the variable f being the sigmoid function itself. By the above assumption, the update of the output layer weight is,

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta \delta_{pk}^o i_{pk} \quad (2.15)$$

and the new variable,

$$\delta_{pk}^o = (y_{pk} - O_{pk}) f_k(\text{net}_{pk}^o) = \delta_{pk} f_k(\text{net}_{pk}^o) \quad (2.16)$$

has been introduced to reduce the complexity of the equation.

To update the hidden layer weights, the error of the hidden layer output and the

gradient of the total error with respect to each hidden layer must be calculated,

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 = \frac{1}{2} \sum_{k=1}^r (y_{pk} - O_{pk})^2 \quad (2.17)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(\sum_{j=1}^m w_{kj}^o i_{pj}))^2 \quad (2.18)$$

and,

$$i_{pj} = f(net_{pj}^h) = f(\sum_{i=1}^n w_{ji}^h X_{pi}) \quad (2.19)$$

The partial derivative of E_p with respect to the hidden layer weights is taken, and the equivalent partial derivative of the different terms is substituted, giving the gradient,

$$-\frac{\partial E_p}{\partial w_{ji}^h} = -\sum_{k=1}^r (y_{pk} - O_{pk}) f_k(net_{pk}^o) w_{kj}^o f_j(net_{pj}^h) X_{pj} \quad (2.20)$$

the equation for weight updating is then,

$$\Delta w_{ji}^h = \sum_{k=1}^r (y_{pk} - O_{pk}) f_k(net_{pk}^o) w_{kj}^o \eta f_j(net_{pj}^h) X_{pj} \quad (2.21)$$

Which can be simplified using Equation 2.16 to,

$$\Delta w_{ji}^h = \eta f_j(net_{pj}^h) X_{pj} \sum_{k=1}^r \delta_{pk}^o w_{kj}^o \quad (2.22)$$

This results in a hidden layer term,

$$\delta_{pj}^h = f_j(net_{pj}^h) \sum_{k=1}^r \delta_{pk}^o w_{kj}^o \quad (2.23)$$

And the weights in the hidden layer are updated by,

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \eta \delta_{pj}^h X_{pi} \quad (2.24)$$

After all the weights of the output layer have been updated, all the weights of the hidden layers are updated by back-propagation, a new training vector is generated

by feed-forward operation, and the process is repeated, the weights once again being updated according to the new error. This process is repeated until the error is within an acceptable range.

2.2 History Stack Algorithm (HSA)

The History Stack Algorithm (HSA) [3, 4] is made adaptive by having a small stack of past input-output pair data that is kept to train the neural network between time-steps while it is controlling the manipulator on-line. This stack of past data is updated every time-step, with old data being discarded and new data added. Input-output pair data are selected from this stack at random between each time-step to train the neural network. This allows the neural network to adapt to changes in the plant in real-time. The signals that are input to the neural network are the desired position, desired velocity and desired acceleration, measured position and measured velocity for each of the joints. The neural network outputs are the estimated inverse dynamic torques for each joint.

The short history of input-output pair patterns represents an approximation of the manipulator transfer function. It operates as a FIFO stack (of size n_p patterns chosen as described in Section 3.4.3) accepting at each time-step k a new pattern, and discarding the oldest pattern. A number n_c of patterns that are in the stack are used at each time-step to update the weights. The weight update equation has a distinction between time-step k and iteration q within that time-step. Each weight $w_{ji}(q, k)$ connecting node i to node j of the previous layer is calculated by,

$$w_{ji}(0, 0) = \xi_{ji} \quad (2.25)$$

$$w_{ji}(0, k) = w_{ji}(n_p n_c, k - 1) \quad (2.26)$$

$$w_{ji}(q, k) = w_{ji}(q - 1, k) + \alpha \varepsilon_i(p) + \beta (w_{ji}(q - 1, k) - w_{ji}(q - 2, k)) \quad (2.27)$$

The back-propagation error value $\varepsilon_i(k)$ corresponds to the input to the sigmoid function calculated for node i during presentation of the input output pair pattern

corresponding to index k . Constant α is the learning rate, β is the momentum rate, and ξ_{ji} is a random variable of zero mean and small variance. Variable p is the time-step index into the stack, and is randomly chosen from the range $k - n_p + 1, k$. Each time that a value of p is chosen, and the weights are updated, iteration counter q is incremented. When the supervised learning rule is applied to all the patterns in the stack for learning at each time-step, the learning rule causes the error in the weight vector of the network to be minimized, corresponding to the mapping represented in the History Stack patterns.

The trade off is that the History Stack uses each pattern $n_p n_c$ times before it is discarded, which improves potential learning performance at each time-step, but increases the controller's computational load by a factor of $n_p n_c$. By setting $n_c > 1$ to achieve large learning increment at each time-step, a reasonably sized stack can be achieved. A stack that is too large will keep patterns that are out of date. At start up, there is no information retained in the stack, so the algorithm must be able to run with a stack size of zero. To take advantage of the reduced computational load during this period, the number of patterns that are selected from the history stack to train the neural network weights $n < n_p, n_c$ can be made temporarily larger by substituting as,

$$n_t = \frac{n_p n_c}{n} \quad (2.28)$$

where n_t is rounded up to the nearest integer.

This accelerates the initial identification performance of the History Stack learning. This start up algorithm was not built into the implementation of the HSA used for these comparisons. Because of the small size of the history stack implemented, this start up algorithm would only help for fifty time-steps (the first one-half of a second), and as such it is not used in this implementation of the History Stack Algorithm.

2.3 Cerebral Model Arithmetic Controller (CMAC)

The CMAC type neural network is quite different from the other neural networks that were tested for this thesis. The three weight layers of this neural network are not represented by two-dimensional matrices, as they are in the others, instead the first layer quantizes the inputs. The signals that are input to the neural network are the desired position, desired velocity and desired acceleration, measured position and measured velocity for each of the joints. The second layer consists of the neural network weights. The quantized input signals from the first layer are combined to select a number of neighboring elements (weights) from the second layer. Each of the quantized input signals selects a point (weight) on its respective dimension of the second layer. The second layer weight matrix has as many dimensions as there are inputs. A number of neighbouring weights are selected as well. The number of elements (weights) selected is dependent upon the size of the neighbourhood constant. The third and output layer sums all the weight elements that were selected from the second layer to generate the CMAC neural network output. The estimated inverse dynamic torques for each joint are the outputs from the neural network. Back-propagation learning is simple as only one layer (the second layer) needs to be updated. The second layer weights are updated by adding to them the torque error (the torque error being the difference between the neural network output and the PD controller output) multiplied by the learning coefficient α .

The acronym CMAC stands for either Cerebral Model Arithmetic Controller or Cerebral Model Articulated Controller, both of which are considered correct, and was originally described in the 1970's by James Albus. The CMAC is a model of what was known about the cerebellum at that time. The CMAC algorithm has been applied as a controller in [7, 8], and is described for DSP and pattern recognition in [5]. Different learning rules have been compared in [9], and the CMAC has been compared to a self tuning regulator (STR) and a Lyapunov based model reference adaptive controller (MRAC) in [10]. Results from this last paper show that for a simple non-linear target system of an integrator, the CMAC approach called for the smallest control signal, and handled the non-linear system well, but was slower at converging to its minimal

Figure 2.4: Block Diagram of the CMAC Neural Network [5]

error state than the other adaptive controllers.

The CMAC algorithms typically take many fewer iterations to converge than back-propagation algorithms due to the localization of network weights based on the neural network inputs. Even though multi-layer networks (for example those similar to the other three neural networks presented in this thesis) have an error surface that can have a relative minima, they do not assure that the best or even a good solution has been found, because the back-propagation techniques are based on gradient search techniques. The CMAC algorithm has error surfaces that are localized based on its inputs. Therefore, each combination of inputs has its own error surface, and hence a good solution can be guaranteed.

Figure 2.4 shows a block diagram of the CMAC neural network. The input state space S is the set of all possible input vectors. Each vector in S is mapped to a set of C points in the conceptual memory A , this being accomplished by the quantization of the input values. By using a neighbourhood constant that selects weights around

the selected point in conceptual memory, the input vectors that are neighbours in S overlap in their C memory locations, and vectors that are far apart in S do not overlap in C . The sum of the absolute values of the differences in the components of two vectors is the measure of distance in the S space. The A memory is called conceptual because it is usually too large to be practical. To obtain a reasonable sized memory, a hash table is used to randomly map the conceptual weights to virtual weights as described in [5]. Hashing is a technique commonly used when the entries of stored data are sparsely distributed. The address of the large 'conceptual' memory is hashed to map it into an address in a smaller 'virtual' or physical memory denoted A' . This process can be described by,

$$\text{hashed (virtual memory) index} = (\text{conceptual index}) \bmod M$$

Here, 'mod' represents the modulo function which is simply the remainder after division. So $A \bmod B$ is simply the remainder after A is divided by B . The modulo function hashes the large virtual memory addresses into a smaller index on the range $[0, M]$. This makes for efficient use of memory, but now the problem arises that some of the indices will have the same conceptual memory location. This problem is known as a 'hash collision'. An unresolved hash collision has the effect of adding noise to the CMAC output. To help solve the problem of collisions, a more effective hashing algorithm was used in this CMAC implementation. This hashing algorithm is double-hashed open hashing [17], and is described by,

$$\text{hash}(k, i) = (h_1(k) + ih_2(k)) \bmod M \quad (2.29)$$

where the auxiliary functions are,

$$h_1(k) = k \bmod M \quad (2.30)$$

$$h_2(k) = 1 + (k \bmod M') \quad (2.31)$$

or all together,

$$\text{hash}(k, i) = ((k \bmod M) + i(k \bmod (M - 1))) \bmod M \quad (2.32)$$

The variable k is the virtual memory index i is an iterator that is incremented after a collision has been detected, and M' is chosen to be slightly less than M . This is a complex method of inserting sparse data of large size into a smaller memory space of size M by wrap-around when the element to be stored is in an address that is greater than M . The auxiliary function in Equation 2.30 is used to reduce collisions, and the auxiliary function in Equation 2.31 is used to resolve hashing collisions by providing alternative memory locations if a hashing collision has been detected. The use of a hash table is possible in the CMAC neural network because not all combinations of input values are possible, and only a fraction of all possible input vectors are typically seen by the system. If the size M of A' is too small for the number of unique input vectors that do appear, a 'collision' will appear. This is only problematic if there are a large number of collisions, and has the effect of noise on the CMAC output. The number of collisions in A' can be reduced by increasing M . Choice of M , and the size of A are described in Section 3.4.4.

The CMAC output is calculated as the average of all the adjustable weights from the field of excited neurons. The output of the CMAC neural network is therefore given by,

$$y(S) = \frac{1}{C} \sum_{i=1}^C w(A_i) \quad (2.33)$$

where C represents the neighborhood constant, and A_i are the elements of A chosen by the neighbourhood constant.

The CMAC neural network is trained using a Least Mean Squared adaptation by adjusting the weights as,

$$w_{new} = w_{old} + \alpha(y_d(S) - y(S)) \quad (2.34)$$

where the same value w_{new} is added to each of the C memory locations w_{old} , and α

is a constant learning rate.

2.4 Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)

The Simultaneous Perturbation Statistical Approximation neural network is a standard feed-forward, back-propagation learning type, but uses a stochastic approximation with simultaneous perturbation learning algorithm instead of the delta learning rule. The signals that are input to the neural network are the desired position, desired velocity and desired acceleration, measured position and measured velocity for each of the joints. The estimated inverse dynamic torques for each joint are the outputs from the neural network.

The Simultaneous Perturbation Statistical Approximation learning rule can be described by the following: For the first time period (several time-steps), the neural network output is generated by feed-forward propagation. For the second time period (several time-steps), the output is by feed-forward propagation with a zero-mean small random number having been added to each weight (simultaneous perturbation), subtraction of each of the first period errors from each of the second period errors provides the neural network with a number of weight update values. This process is repeated constantly during operation. The number of time steps chosen for each time period is critical. If the time period is too long, the error function will be overly influenced by the trajectory, if the time period is too short, perturbation will occur too often, and add too much error into the weights. The error function is not directly a result of the error, therefore on-line weight adaptation can be performed without worrying about of the weight oscillation problem that will be discussed in Section 3.1. Figure 2.5 shows a block diagram of the SPSA learning process.

The error function $J(w)$ is usually defined as a squared error of the plant. When the usual gradient method is used as a learning rule, the quantity $\partial J(w)/\partial w$ must be known. Then,

$$\frac{\partial J(w)}{\partial w} = \frac{\partial J(w)}{\partial y} \frac{\partial y}{\partial w} = (y - y_d) \frac{\partial f(u)}{\partial u} \frac{\partial u}{\partial w} \quad (2.35)$$

where y_d is the desired output of the plant. Here, $(y - y_d)$ is known, and $\partial u / \partial w$ can be calculated by using the back-propagation learning rule. Because this learning rule can estimate the gradient of the error function with respect to the neural network weights using only the values of the error function, this learning rule allows on-line adaptation to environmental changes in the objective plant (here in the robot).

This learning algorithm was originally proposed by J. C. Spall as an extension of Kiefer-Wolfowitz stochastic approximation [12, 13], and was introduced as a learning rule for use in neural networks by Y. Maeda [14]. The Simultaneous Perturbation learning rule is described by,

$$w(t) = (w_1(t), w_2(t), \dots, w_N(t))^T \quad (2.36)$$

$$s(t) = (s_1(t), s_2(t), \dots, s_N(t))^T \quad (2.37)$$

where $w(t)$ is the weight vector, and $s(t)$ is a sign vector whose components are ± 1 . The weight update vector is computed as follows,

$$\Delta w_i(t) = \frac{J(w(t) + cs(t)) - J(w(t))}{c} s_i(t) \quad (2.38)$$

The variable c is the magnitude of perturbation that is added to the weights, its choice being described in Section 3.4.5, and the weights are updated as,

$$w_i(t+1) = w_i(t) - \alpha \Delta w_i(t) \quad (2.39)$$

with α being the positive learning coefficient. So, only two values of the error function $J(w(t))$ and $J(w(t) + cs(t))$ are used to update the neural network weights, and there is no information about the objective plant included in the learning rule. The error function is defined as,

Figure 2.5: SPSA Neural Network Flowchart.

$$J(u(w)) = \sum_i (w_{2,i} - d_i)^2 \quad (2.40)$$

where d_i denotes the desired position of the arm. Without perturbation the controller is run through a time period, the value of the error $J(w(t))$ is retained for later use. Next perturbation is added to all weights, and the controller is operated for the next time period, giving the error $J(w(t) + cs(t))$. By using the Simultaneous Perturbation learning rule the weight modification vector and updated weights are obtained. This procedure is repeated. A flowchart of the learning algorithm is presented in Figure 2.5.

Chapter 3

Neural Network Pre-Training

The goal of neural network training is to produce a combined neural network and PD controller to control the system described in Section 3.3. This controller is presented in Figure 3.1.

Initial off-line sequential training is referred to as 'pre-training' in this thesis. Pre-training being non-sequential in nature, eliminates the correlation between time and consecutive input-output pairs, but does not allow the neural network to adapt to changing target system dynamics. The following section presents a detailed explanation of why the neural networks in this thesis were pre-trained, and not trained sequentially on-line.

3.1 Problems With On-Line (Sequential) Training

For our purposes, we shall define on-line training as the updating of the neural network weights sequentially between each time-step, while the neural network being trained may also be operating as the controller (in parallel with the PD controller) for the manipulator. Sequential on-line training of back-propagation neural networks is difficult because when presented with an input-output pair with which to update the weights, the neural network learning rule does not learn the inverse dynamics of the target system. Instead the learning rule updates the weights to follow the trajectory of the target system. When a standard delta rule back-propagation neural

Figure 3.1: The Controller Model for each Manipulator Joint [2].

network is used to estimate the inverse dynamics of a manipulator on-line, the neural network learns to estimate the output of the PD controller with minimal error. This creates the illusion that training is complete, and that the trained neural network can estimate the dynamics of the PD controller, and hence control the target system. However, when the weights are examined, it is apparent that they are not converging to an optimal value, and instead are oscillating in phase with the desired trajectory of the manipulator, as can be seen in Figures 3.3, and 3.4. Figures 3.3, and 3.4 were generated when the standard delta rule neural network was not pre-trained, but was trained on-line between each time step, while the manipulator was in operation, and being used as a controller for the manipulator. When training of the neural network controller ceased, the controller was not able to estimate the PD controller output. Figure 3.2 shows the desired position and actual position of the manipulator during on-line training. Figure 3.3 shows the neural network and PD controller output during on-line training. Figure 3.4 shows an output layer weight of the neural network during on-line training. It is clearly visible that the neural network weight presented in

Figure 3.2: Desired Position and Actual Position of Sequentially On-Line Trained NANN.

Figure 3.4 was simply oscillating in phase with the desired output, or neural network output presented in Figure 3.3. Since the neural network weight did not converge to the desired value, it clearly would be unable to control the manipulator after training has ceased.

To solve the problem of the inability of the standard delta rule neural network to learn on-line, the neural network was pre-trained non-sequentially off-line, with the input-output pairs being selected randomly (with respect to time) from a repository of past values. This standard delta rule neural network will be referred to as the Non-Adaptive Neural Network (NANN) in the remainder of this thesis.

It is desirable to have neural network controllers that can learn sequentially on-

Figure 3.3: Neural Network and PD Controller Output of Sequentially On-Line Trained NANN.

Figure 3.4: A Neural Network Output Weight of Sequentially On-Line Trained NANN.

line, with or without the requirement of being pre-trained. On-line sequential training allows the controller to adapt to dynamic changes in the target system. On-line sequentially trained neural networks that do not require pre-training allow a controller to be designed with only minimal assumptions about the target system. An intuitive method to overcome the on-line training problem and one that will facilitate adaptation of the neural network, is to keep a small repository (or stack) of previous input-output pair values so that there are several possible pairs from which one pair may be selected at random for training at each time-step. This will theoretically eliminate correlation between consecutive input-output pairs. This is the basis upon which one might arrive at the concept of the History Stack Algorithm (HSA). Another possible solution is to separate time from trajectory, and thereby eliminate correlation. The CMAC algorithm accomplishes this by using different areas of the weight matrix for different actual and desired trajectories and velocities. Yet another solution is to use a weight error calculation method that does not rely directly on the trajectory error. This concept is realized using the Simultaneous Perturbation Statistical Approximation (SPSA) algorithm.

The CMAC and SPSA algorithms are reportedly capable of controlling a target system while learning sequentially on-line without prior training[6][16]. Results of sequential on-line training of these neural networks and the HSA without prior training are presented below.

For sequential on-line training of the HSA, CMAC, and SPSA neural networks, the neural networks were operated in parallel with the PD controller. The error is then calculated based on the error between the recorded combined neural network and PD controller torque output, and the newly calculated neural network output (the error is the PD controller output), and the error is then back-propagated through the neural network to update the weights. On-line training consisted of 1,000 training sessions, each training session consisting of 10,000 time-steps. Each training session used the same trajectory. Error calculation and weight updating for each of the neural networks was performed in the manner that was described for each of the respective algorithms in Sections 2.2 to 2.4.

Figure 3.5: Non-pre-trained CMAC Desired and Actual Position.

Results from CMAC sequential on-line training are presented in Figure 3.5. During sequential on-line training of the CMAC, while the CMAC is being used as a controller, the CMAC causes a wild trajectory due to the influence of weights that were initially poorly trained. This means that the CMAC initially uses many weights. In order to perform this simulation, hashing collision protection had to be turned off to avoid overflowing the hash table. Therefore there will be many hashing collisions causing unpredictable trajectories at some times, and a generally noisy CMAC neural network output. CMAC pre-training is discussed in Section 3.4.4.

Results from the sequentially trained, but non-pre-trained SPSA algorithm are similarly poor. It seems that as the controller for this target system, the SPSA needs a to be pre-trained. Figure 3.6 shows the output of the non-pre-trained SPSA

Figure 3.6: Non-pre-trained SPSA Desired and Actual Position.

algorithm. As a result of not being pre-trained, the SPSA learning rule has built up a bias in the SPSA weights.

The HSA was not described as an algorithm that could be used without pre-training (as the CMAC and SPSA were), however it does seem logical that it could be used in such a manner. The output for the sequentially trained non-pre-trained HSA is shown in Figure 3.7. When not pre-trained, the output of the HSA is oscillatory in nature. This is likely due to the small size of the history stack itself, and repetitive use of its elements over a relatively small number of time-steps. The small size of the history stack causes correlation of error with time, and repetitive use of the stack elements causes the equivalent effect of using a large learning coefficient. If at each time-step, 5 elements of the history stack are used, then for 10,000 time-steps,

Figure 3.7: Non-pre-trained HSA Desired and Actual Position.

the HSA neural network weights are trained 50,000 times. During pre-training, the NANN weights are only trained 10,000 times per training period (10,000 time-steps). By comparison, the HSA learning coefficient is five times that of the NANN. With a history stack size of only 50 elements from which to randomly choose the non-sequential training input-output pair, correlation between error and time is much greater than that in the pre-training of the NANN (which effectively has a history stack of 10,000 elements, with one train per time-step). The problem related to the size of the history stack can be alleviated by using a larger history stack, but the trade off is that the algorithm will then be slower to react to changes in the dynamics of the target system, and the HSA will be trained for old system dynamics that may already have changed. The problem of repetitive use of the history stack elements can be solved by reducing the magnitude of the learning coefficient, but the HSA will once again be slow to react to the changing dynamics of the system.

As a result of the above problems, all of the neural networks presented in this document were pre-trained in two stages to provide better and more accurate comparison results. Pre-training solves the correlation problems by randomly selecting samples of the data that were collected during previous operation of the system to train the neural network. This makes it difficult to train the neural network on-line, which is required for the network to adapt to a changing environment. The following describes the pre-training of the neural networks.

3.2 Neural Network Pre-Training

Sequential training is one method of training the neural network. In this method the neural network is trained while the manipulator is in motion. At each time-step, the signals used to control the manipulator are input to the neural network. The neural network output is then compared to the PD controller output, and the difference is used to train the neural network. This sequential training method was tried initially, but the results (as discussed in Section 3.1) show the neural network does not learn the inverse dynamics of the manipulator system, but rather the neural

network weights simply oscillate in phase with the desired output. As a result the following training method was used to train the neural networks. The pre-training of the neural networks is divided into two stages.

The above two stage method of off-line non-sequential training is referred to in this thesis as pre-training, as it refers to the neural network being trained before it can be used as a controller in parallel with the PD controller in real time.

3.2.1 Stage One Pre-Training

In the first stage of pre-training [2], the neural network is trained using the output of the PD controller, which gives a good initial estimate of the inverse dynamics of the manipulator system. The target system is controlled only by the PD controller, and the neural network is trained to act as the PD controller by learning the PD controller output. Training is performed based on the error between the PD controller torque output and the neural network torque output. In fact, the desired torque of the neural network controller is the torque output of the PD controller. In Stage One training, the neural network is trained for 10,000 training sessions, where each training session is 10,000 time-steps in length. The number of training sessions for the first stage of training is equivalent that used in [2]. During each training session, the manipulator is moved through its workspace in a pre-filtered step function pattern of random amplitude. At each time-step, desired position, desired velocity, and desired acceleration as well as the measured position and measured velocity, and joint torques and forces for each actuator are stored in an array indexed by time-step for each of the 10,000 time-steps. At the end of the test trajectory the array of input-output data for each time-step is written to disk. Each of the input-output pairs are selected randomly from this array to train the neural network weights using the learning rule appropriate for the specific neural network algorithm. The error is then back-propagated through the neural network to update the weights. The process is repeated for 10,000 input-output pairs, and then the next training session is commenced. At the end of Stage One training, the neural network has learned to imitate the PD controller. Figure 3.8 shows a block diagram of stage one pre-training.

Figure 3.8: Stage One Method Of Training [2].

3.2.2 Stage Two Pre-Training

In Stage Two pre-training [2], the neural network weights trained from Stage One are used as an inverse dynamic feed-forward controller in parallel with the PD controller. The training trajectory for this stage is calculated in the same manner as that used for Stage One. There are 5000 training sessions in Stage Two training. The number of training sessions for the second stage of training is equivalent to that used in [2]. Each training session is 10,000 time-steps in length. During each training session, the combined PD and neural network controller controls the target system and all the inputs and outputs to the system for each of the 10,000 time-steps are recorded in an array for future use. To train the network, as in Stage One, input-output pairs are selected randomly from the array of 10,000 points and used to train the neural network weights using the learning rule appropriate for the specific neural network algorithm. The error is then back-propagated through the neural network to update the weights. The process is repeated for 10,000 input-output pairs, and then the next training session is commenced. When training of this stage is completed, the neural network is able to estimate the inverse dynamics of the manipulator, and the neural network has learned to control the target system on its own. Figure 3.9 shows a block diagram of Stage Two pre-training.

3.3 The Target System

The target system that was used to test and illustrate the design of these controllers, is a simulation of a manipulator with two degrees of freedom. Figure 3.10 shows the design of the manipulator used. Ignoring gravity, the equation of motion of the manipulator is as follows,

$$T = M(q)\ddot{q} + C(q, \dot{q}) \quad (3.1)$$

where,

Figure 3.9: Stage Two Method Of Training [2].

Figure 3.10: Schematic Of The Two Degree Of Freedom Robot, Used For The Simulations.

$$M(q) = \begin{bmatrix} m_1 l^2 + 2m_2 l^2 + 2m_2 l^2 \cos q_2 & m_2 l^2 + m_2 l^2 \cos q_2 \\ m_2 l^2 + m_2 l^2 \cos q_2 & m_2 l^2 \end{bmatrix} \quad (3.2)$$

$$C(q, \dot{q}) = \begin{bmatrix} -2m_2 l^2 \dot{q}_1 \dot{q}_2 \sin q_2 - m_2 l^2 \dot{q}_2^2 \sin q_2 \\ m_2 l^2 \dot{q}_1^2 \sin q_2 \end{bmatrix} \quad (3.3)$$

Here, m_1 is 1, m_2 is 2, and $l = l_1 = l_2$ is 1. Desired position, velocity, and acceleration are given as the output of a second order pre-filter, which is a critically damped system with a bandwidth of 2 radians per second. The pre-filter transfer function is,

$$w_m(s) = \frac{4}{s^2 + 4s + 4} \quad (3.4)$$

In state space this pre-filter is defined as,

$$\begin{bmatrix} \dot{q}_d \\ \ddot{q}_d \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -4 & -4 \end{bmatrix} \begin{bmatrix} q_d \\ \dot{q}_d \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} q_c \quad (3.5)$$

When the above pre-filter is transformed from the continuous domain into the digital domain with sampling frequency of 100Hz, it results in the following,

$$\begin{bmatrix} q_d(k+1) \\ \dot{q}_d(k+1) \end{bmatrix} = \begin{bmatrix} 0.9998 & 0.0098 \\ -0.0392 & 0.9606 \end{bmatrix} \begin{bmatrix} q_d(k) \\ \dot{q}_d(k) \end{bmatrix} + \begin{bmatrix} 0.0002 \\ 0.0392 \end{bmatrix} q_c(k) \quad (3.6)$$

Desired position commands are passed through the above filter to generate the desired angular position and velocity of the manipulator arms.

3.4 Simulation And Pre-Training Of The Individual Neural Networks

The simulated manipulator was subjected to a desired trajectory that was 10,000 time-steps (or 100 seconds) in length. Recall that a time-step was defined in for this thesis as the time between each sampling of the manipulators position, here this is 0.01 seconds. For each trajectory, 20 of 4,096 possible destinations were selected randomly for each manipulator arm. In other words, during each trajectory or training session, the manipulator arms were moved to 20 different positions. The transition time from one destination to the next is 5 seconds in duration, and the sampling rate is simulated at 100Hz. For Stage One pre-training there are a total of 10,000 training sessions each of 10,000 time-steps. Stage Two pre-training has 5,000 training sessions, each of 10,000 time-steps with the neural network being trained through a total of 15,000 sessions in total.

Figure 3.11: PD Controller Block Diagram.

3.4.1 The PD Controller

The PD controller was initially used to train the neural networks, and then used in parallel with the neural networks as a controller. The PD controller was also tested alone as a controller of the target system. The target system plant can be simplified to be a double integrator. A block diagram of the PD controller and the simplified target system structure is presented in Figure 3.11. The system presented in Figure 3.11 can be reduced to the transfer function,

$$\frac{q(s)}{q_d(s)} = \frac{sk_d + k_p}{s^2 + sk_d + k_p} \quad (3.7)$$

The PD controller constants are; $k_p = 60$, $k_d = 35$. The values k_p and k_d were determined to place a zero at $s = -1.7143$. The test results for the simulated target system with the PD controller being used alone as the target system controller are presented in Figures 3.12, and 3.13. Figure 3.12 shows the desired position compared to the actual position. Figure 3.13 shows the trajectory error. Maximum overshoot was 8.3%, and the RMS error was 0.0183.

Figure 3.12: PD Controller Desired and Actual Position.

Figure 3.13: PD Controller Trajectory Error.

3.4.2 Non-Adaptive Neural Network

This simulation was built from Zaryab Hamavand's weight training simulations for his Masters thesis [2]. Originally coded in Pascal, the code was translated into C++. This neural network is non-adaptive and is used to collect baseline data for comparison with the adaptive neural networks. The Non-Adaptive Neural Network will also be used to pre-train the HSA and SPSA neural networks. It is a three layer feed-forward network with a two-step feedback error learning method as presented in Section 2.1. The equations that describe the dynamics of this neural network were presented in Section 2.1.2, Equations 2.1 to 2.24. The neural network consists of two hidden layers each of 35 nodes, the input layer has 10 nodes and there are two nodes in the output layer. The size of each layer and number of layers were chosen to be the same as those used in [2]. The signals that are input to the neural network are the desired position, desired velocity, desired acceleration, measured position and measured velocity for each of the joints. The controller torques for each joint are the outputs from the neural network. Weights are adjusted by using a back-propagation algorithm. The neural network is then trained to learn the output of the PD controller as described in Section 3.2.1. Figure 3.14 shows the actual position compared to the desired position at completion of the Stage One pre-training. Figure 3.15 shows the trajectory error at completion of Stage One pre-training. The neural network closely approximates the desired output. At the end of Stage One pre-training, the RMS error was 0.0087, and the maximum overshoot was 4.43%.

At the end of the Stage Two pre-training, the controller performance is shown in the figures below. Figure 3.16 shows the desired and actual position of joint 1, note that the trajectory error is too small to be visible in this graph, while Figure 3.17 shows the trajectory error. Please note that this graph is scaled $\times 10^{-3}$. Figure 3.18 shows the neural network output and the PD controller output. Figure 3.18 shows that the neural network output is of much greater magnitude than the PD controller output indicating that the neural network is dominant and implying that the neural network is an estimator of the inverse dynamics of the manipulator. Maximum overshoot was 0.15%, and the RMS error was 4.8750×10^{-4} .

Figure 3.14: NANN Desired and Actual Position After Stage One Training.

Figure 3.15: NANN Trajectory Error After Stage One Training.

Figure 3.16: NANN Desired and Actual Position After Stage Two Training.

Figure 3.17: NANN Trajectory Error After Stage Two Training (Note Angle Scaling $\times 10^{-3}$).

Figure 3.18: NANN Output Versus PD Controller Output.

If a neural network that is destined to be a controller of an adaptive system is to be trained off-line, one would have to train a separate set of neural network weights for each possible variation in dynamics. A separate neural network must be available for each time that the manipulator has a change in its dynamics for example when it picks up an object. This decreases the neural network's control characteristics to be those akin to a gain scheduled controller. Large amounts of memory would therefore be required to maintain all the weight matrices for each possible change in dynamics.

3.4.3 History Stack Algorithm (HSA)

The HSA learning algorithm was described in Section 2.2. This neural network consists of two hidden layers each of 35 nodes, an input layer consisting of 10 nodes and an output layer with two nodes. The signals that are input to the neural network are the desired position, desired velocity, desired acceleration, measured position and measured velocity for each of the joints. The control torques for each joint are the outputs from the neural network. The weights are adjusted using the delta back-propagation learning rule. The error values used in the back-propagation learning rule are derived from the input-output pairs in the history stack as described in Section 2.2. The NANN weights are the same as the HSA weights, so the NANN pre-trained weights can be used as the pre-trained weights for the HSA. As a result, the pre-trained NANN was a starting point, as pre-training is required for this algorithm. Without pre-training, the HSA produces an output that oscillates. The results of an attempt at running this algorithm without pre-training are presented in Figure 3.7.

3.4.4 Cerebral Model Arithmetic Controller (CMAC)

Recall the CMAC as described in Section 2.3. The ten signals that are input to the CMAC neural network are the desired position, desired velocity, desired acceleration, measured position and measured velocity for each of the joints. The neural network consists of three layers, the first layer simply quantizes each of the input signals into one of 200 possible values. The second layer consists of 200^{10} weights. The exponent

value 10 is derived from the fact that there are ten inputs to the neural network. The quantized input signals from the first layer are combined to select a number of neighboring elements (weights) from the second layer. The number of elements (weights) selected is dependent on the size of the neighbourhood constant. The third and output layer sums all the elements that were selected from the second layer to generate the CMAC neural network output. The inverse dynamic torques for each joint are the outputs from the neural network. Weight training is simple as only one layer (the second layer) needs to be updated. The second layer weights are updated by adding to them the torque error multiplied by the learning coefficient α .

Besides the CMAC, all the other algorithms were simulated with only one neural network to control both actuators, but in this case two weight matrices were required, because the CMAC algorithm cannot produce more than one output signal without correlation problems between the output signals. With two neural networks to control the actuators and five inputs for each actuator, two ten-dimensional second layer weight matrices are required. Two five dimensional second layer weight matrices could not have been used, because each of the two neural networks (the two second layer weight matrices) must have knowledge of the position, velocity and acceleration of both of the manipulator arms in order for the CMAC to be an effective controller. If the input signals were quantized to a resolution of 35 (as in the source literature), then the size of each matrix is to be 35, and the matrices required would be of a size that is impractical to implement on a standard desktop computer, as this would result in two weight matrices of 35^{10} elements. Instead of quantizing the inputs to a resolution of 35, the resolution of 200 was chosen, as the former size was found to be of insufficient resolution to accurately map the input space. This means that the total memory required for weight matrices alone is 200^{10} . This is far too large to be practical on an affordable computer. Fortunately, due to localization of the weights with respect to the inputs, only a fraction of all possible input combinations actually being realizable, and only a small number of these possible weights are actually required. This means that a hash-table can be used to store the weight matrices, as was discussed in Section 2.3.

In the back-propagation neural networks, each weight is trained once for each time-step in each training session. Any given weight of the CMAC neural network is only trained if all ten inputs are the correct value to activate that particular weight. The CMAC weights are therefore only trained for a fraction of the number of times that the NANN weights are trained during pre-training. To rectify this problem the learning coefficient α is set to be higher in pre-training than it is in adaptive training. This speeds up the overall learning of the CMAC neural network. Hence during Stage One training where the torque error is the greatest, α is 0.01, during Stage Two training the torque error is slightly smaller so α is 0.005, and during adaptive training, where the torque error is small, α is 0.002. Therefore the CMAC requires less pre-training than if the learning coefficient was constant with respect to training iteration.

For Stage One pre-training, A was chosen to be 200, and M is 12,000,000, the learning coefficient α is 0.01, and the neighbourhood constant is 3. At the end of Stage One training, the maximum overshoot was 6.57%, and the RMS error was 0.0113. Figure 3.19 shows the desired and actual position of the manipulator motor 0 at the end of Stage One pre-training. Figure 3.20 shows the trajectory error of motor 0 after completion of Stage One pre-training. Figure 3.21 shows the CMAC neural network and PD controller outputs at the end of Stage One pre-training.

For Stage Two pre-training, A was chosen to be 200, and M is 12,000,000, the learning coefficient α is 0.005, and the neighbourhood constant is 3. Maximum overshoot was 5.85%, RMS error was 0.0087. Figure 3.22 shows the desired and actual position of the manipulator motor 0 at the end of Stage Two pre-training. Figure 3.23 shows the trajectory error of motor 0 after completion of Stage Two pre-training. Figure 3.24 shows the CMAC neural network and PD controller outputs at the end of Stage Two pre-training. The CMAC performance is poor in comparison to the other neural networks.

Figure 3.19: CMAC Desired and Actual Position After Stage One Training.

Figure 3.20: CMAC Trajectory Error After Stage One Training.

Figure 3.21: CMAC Neural Network Output Versus PD Controller Output After Stage One Training.

Figure 3.22: CMAC Desired and Actual Position After Stage Two Training.

Figure 3.23: CMAC Trajectory Error After Stage Two Training.

Figure 3.24: CMAC Neural Network Output Versus PD Controller Output After Stage Two Training.

3.4.5 Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)

This neural network consists of two hidden layers each of 35 nodes, the input layer consists of 10 nodes and there are two nodes in the output layer. The signals that are input to the neural network are the desired position, desired velocity, desired acceleration, measured position and measured velocity for each of the joints. The controller torques for each joint are the outputs from the neural network. This simulation is the same as that of the NANN, except that the delta rule learning algorithm was replaced with a statistical approximation learning algorithm. The SPSA is more difficult to train than delta rule neural networks, because selection of the learning coefficient, and magnitude of the perturbation have some effect on one another. There is also only a small range for each of these two values that produce weights that converge to their desired values. The learning coefficient must be smaller in magnitude (by several orders of magnitude) than the coefficient of perturbation, or the SPSA neural network weights will saturate. Some papers [15] have suggested that the learning coefficient must decrease in magnitude with time, and have further suggested algorithms to choose the learning coefficient decrease curve. Decreasing the learning coefficient with time is impractical for neural networks that must learn on-line, as more emphasis is put on earlier estimates of the dynamics that are changing with time. As a result a constant with respect to time learning coefficient is used here. The SPSA pre-trained weights are the same as the NANN pre-trained weights, the NANN can produce more accurate weights, and therefore the NANN pre-trained weights were used for the initial SPSA weights.

3.5 Non-Adaptive Comparison Results

The most accurate results produced from off-line pre-training are achieved by the NANN, RMS error at the end of Stage Two pre-training was 4.8750×10^{-4} . The least accurate results from off-line training were achieved using the CMAC algorithm, RMS error at the end of Stage Two pre-training was 0.0087. The RMS error for the PD

controller was 0.0183. Both the NANN and CMAC have however been able to improve on the PD controller's performance.

Despite the fact that both the CMAC and SPSA algorithms are reputed to be capable of being used as controllers without having been pre-trained [6][16], both of these algorithms that were simulated for this research were pre-trained. In fact it was found that for the target manipulator system being simulated here, neither the HSA, nor SPSA were capable of sequential learning on-line without having been pre-trained. Without pre-training, the non-pre-trained SPSA simply developed a bias in the weights, that caused a bias in torque output of the SPSA, and the non-pre-trained HSA developed oscillations in its output. It should be possible for the HSA to be trained sequentially on-line, because if the history stack is increased to be the same length as the number of time steps in a training session (10,000 here), and the learning coefficient is decreased to be the same magnitude as that used in the NANN, then the HSA is the same as the NANN during pre-training. In the cases of the HSA and the SPSA, the NANN weights were used to replace pre-training of the HSA and SPSA. The CMAC algorithm was separately pre-trained, as the CMAC weights are vastly different than those of the NANN. Pre-training was performed to allow proper comparison of all algorithms.

Chapter 4

Results Of Simulation With Adaptation

The test for adaptation was accomplished by changing the mass of one of the links of the manipulator before the test runs. For the first set of simulations the mass of the gripper arm was changed to be twice that of its actual value. Then the gripper arm's mass was changed to be five times its original mass. These changes were performed to simulate the gripper of the manipulator having picked up objects. All the neural networks are trained and compared using the same 'standardized' trajectory. Unlike during pre-training, where a trajectory is selected at random for each training session, during the adaptive training, the same standardized trajectory is used for each training session repeatedly. The neural networks were repeatedly trained using this trajectory in each training session so that the RMS error for each training session would be less than that of the previous training session, thereby showing convergence to the desired values.

Please note that at $t = 6000$ the desired position of the standardized trajectory puts both arms in the fully extended position, where the torque required to start or stop them is the greatest. This explains why there is a great deal of overshoot at $t = 6000$ when the mass of the gripper arm is increased.

All processing times reported are based on the simulations written in C++, running on Windows 2000 on a 1.5GHz Pentium 4 processor. The processing times

quoted were calculated using a stopwatch, and as such are not that accurate, however considering the differences in processing time required for each of the different neural networks, they are accurate enough to compare the algorithms.

The RMS error was calculated for each training session for the adaptive algorithms. The RMS error is the square root of the sum of the squared error at each time step. The RMS error can be written as,

$$E_{RMS} = \sqrt{\sum_{i=1}^{10000} e_i^2} \quad (4.1)$$

Where e_i is the trajectory error, or the difference between the desired position and actual position at time step i .

4.1 Results of Simulation With Gripper Arm Mass Twice Original Value

For the results presented in this section, the mass of the gripper arm was changed to be twice that of its original value.

4.1.1 The PD Controller

Figures 4.1 and 4.2 present the test results for the simulated system with the gripper mass changed to be twice its original value, with the PD controller being used alone to control the target system. Figure 4.1 shows the desired position compared to the actual position. Figure 4.2 shows the trajectory error. Maximum overshoot was 13.92%, and the RMS error was 0.0386.

The results depicted in these figures show that the PD controller is no longer as good of a controller for the changed system with doubled gripper arm mass, as it was for the original system. This is hardly unexpected, because the PD controller constants k_p , and k_d were calculated based on the original system. The PD controller constants could be recalculated to regain the performance shown on the original non-

Figure 4.1: PD Controller Desired and Actual Position of Changed System (Mass x2).

Figure 4.2: PD Controller Trajectory Error of Changed System (Mass x2).

Figure 4.3: NANN Desired and Actual Position of Changed System (Mass $\times 2$).

changed system in Section 3.4.1 by using the procedure discussed in Section 3.4.1.

4.1.2 Non-Adaptive Neural Network

Figure 4.3 shows the manipulator arm position generated by the combined PD and Neural Network controller compared to the desired position while it is controlling the target system when the mass of the gripper arm has changed. Figure 4.4 shows the trajectory error during testing. Note that the neural network was not re-trained to be a controller of the parameter changed system. The neural network closely approximates the desired output. Maximum overshoot was 1.98%, and the RMS error was 0.0046. Figure 4.5 shows that the neural network output is of much greater

Figure 4.4: NANN Trajectory Error of Changed System (Mass x2).

Figure 4.5: NANN Neural Network Output Versus PD Controller Output of Changed System (Mass $\times 2$).

magnitude than the PD controller output, implying that the neural network is still a good estimator of the inverse dynamics of the manipulator. The Non-Adaptive Neural Network is still a good controller, even though it is controlling a target system whose parameters have been changed drastically. Please note that the NANN was not trained to control the system it is controlling here. It has not been trained in any way to control the system with changed dynamics, only the original system that was described in Section 3.3. The computational time required for 10,000 time-steps was about 1 second of processor time.

4.1.3 History Stack Algorithm (HSA)

For this simulation the pre-trained non adaptive neural network was a starting point, as pre-training is required for this algorithm. It was found that good results could be achieved using a history stack of 50 elements, the momentum coefficient β is 0 (as the momentum coefficient has little or no effect), and a learning coefficient α is 0.1. At each time-step five of these history stack elements were chosen at random to train the neural network using the same back-propagation delta learning rule that was used to train the non-adaptive neural network. After 1,000 training sessions (or test trajectories as discussed in Section 3.4), the maximum overshoot was 0.68%, and the RMS error was 0.0035. The computational time required for one training session, or 10,000 time-steps was about 5 seconds of processor time. Figures 4.6 to 4.8 show the actual and desired position, trajectory error, and controller outputs for a standardized trajectory. The results shown here are similar to those found in the source literature [3]. Figure 4.9 shows the RMS error for each training session during the simulations. The RMS error has decreased to almost its minimal value after about fifty training sessions. The RMS error rises slightly towards the end of the simulation, however when the simulation was continued the RMS error decreased quickly back to its minimum value. The HSA RMS error is quite noisy when compared to the CMAC or SPSA RMS error, this is because of the large amount of correlation between input-output pairs due to the small size of the history stack. Increasing the size of the history stack should alleviate both of these problems.

4.1.4 Cerebral Model Arithmetic Controller (CMAC)

For the adaptive CMAC simulations, A was 200, M was 12,000,000, the learning coefficient α was 0.002, and the neighbourhood constant was 3. The computational time required for one training session, or 10,000 time-steps was 0.75 seconds of processor time. Figures 4.10, 4.11, and 4.12, show the desired and actual position of motor 0, trajectory error of motor 0, and the neural network and PD controller outputs

Figure 4.6: HSA Desired and Actual Position of Changed System (Mass x2).

Figure 4.7: HSA Trajectory Error of Changed System (Mass x2).

Figure 4.8: HSA Neural Network Output Versus PD Controller Output During On-Line Training of Changed System (Mass x2).

Figure 4.9: HSA RMS Error of Changed System (Mass x2).

Figure 4.10: CMAC Desired and Actual Position of Changed System (Mass x2).

respectively after 10,000 training sessions. Figure 4.13 shows the RMS error for each training session. After completion of the simulation, the RMS error was 0.0212, and maximum overshoot was 9.43%. The results shown here are once again similar to those presented in the source literature.

4.1.5 Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)

For the SPSA algorithm it was found that good results could be achieved with perturbation being applied after 10 time-steps, and with a learning coefficient α of

Figure 4.11: CMAC Trajectory Error of Changed System (Mass x2).

Figure 4.12: CMAC Neural Network Output Versus PD Controller Output of Changed System (Mass x2).

Figure 4.13: CMAC RMS Error of Changed System (Mass x2).

Figure 4.14: SPSA Desired and Actual Position of Changed System (Mass x2).

0.0000000002. The computational time required for one training session, or 10,000 time-steps was about 1 second of processor time. The results for the standardized trajectory are presented in Figures 4.14, 4.15, and 4.16. After 10,000 training sessions, the RMS error was 0.0058, maximum overshoot was 2.70%. The results presented here show a system that has about the same RMS error as that presented in the source literature, such as [15]. Figure 4.17 shows the RMS error for each training session during the simulations.

Figure 4.15: SPSA Trajectory Error of Changed System (Mass x2).

Figure 4.16: SPSA Neural Network Output Versus PD Controller Output of Changed System (Mass x2) .

Figure 4.17: SPSA RMS Error of Changed System (Mass x2).

4.2 Results Of Simulation With Gripper Arm Mass Five Times Original Value

For the results presented in this section, the mass of the gripper arm was changed to be five times that of its original value.

4.2.1 The PD Controller

The test results for the simulated target system with changed gripper mass, with the PD controller being used alone as the target system controller are presented in Figures 4.18, and 4.19. Figure 4.18 shows the desired position compared to the actual position. Figure 4.19 shows the trajectory error. Maximum overshoot was 30.89%, and the RMS error was 0.1430. The results depicted in these figures show that the PD controller is no longer a good controller for the changed system.

4.2.2 Non-Adaptive Neural Network

Figure 4.20 shows the combined PD and Neural Network controller output compared to the desired position while it is controlling the target system with the mass of the gripper arm changed. Figure 4.21 shows the trajectory error during testing. The neural network can still approximate the desired output, and improve upon the PD controller's performance. Maximum overshoot was 11.24%, and the RMS error was 0.0349. Figure 4.22 shows that the neural network output is larger in magnitude than the PD controller output, implying that the neural network is able to estimate the inverse dynamics of the manipulator. The Non-Adaptive Neural Network is still a good controller, even though it is controlling a target system whose parameters have been changed drastically. Please note that the NANN was not trained to control the system it is controlling here, nor has it been trained for the desired trajectory. It has not been trained in any way to control the system with changed dynamics, only the original system that was described in Section 3.3. The computational time required for 10,000 time-steps was about 1 second of processor time.

Figure 4.18: PD Controller Desired and Actual Position of Changed System (Mass x5).

Figure 4.19: PD Controller Trajectory Error of Changed System (Mass x5).

Figure 4.20: NANN Desired and Actual Position of Changed System (Mass x5).

Figure 4.21: NANN Trajectory Error of Changed System (Mass x5).

Figure 4.22: NANN Neural Network Output Versus PD Controller Output of Changed System (Mass x5).

4.2.3 History Stack Algorithm (HSA)

This simulation was performed in the same manner, and using the same values as the HSA presented in Section 4.1.3, only the value of the mass of the gripper arm is different. The computational time required for one training session, or 10,000 time-steps was about 5 seconds of processor time. Figures 4.23 to 4.25 show the actual and desired position, trajectory error, and controller outputs for a standardized trajectory respectively. After 10,000 training sessions the RMS error was 0.0317, maximum overshoot was 10.25%. Figure 4.26 shows the RMS error for each training session

Figure 4.23: HSA Desired and Actual Position of Changed System (Mass x5).

during the simulations. The RMS error has decreased to almost its minimal value after about fifty training sessions. The RMS error rises slightly towards the end of the simulation, however when the simulation was continued the RMS error decreased quickly back to its minimum value. The HSA RMS error is quite noisy when compared to the CMAC or SPSA RMS error, this is because of to the large amount of correlation between input-output pairs due to the small size of the history stack. Increasing the size of the history stack should alleviate both of these problems.

Figure 4.24: HSA Trajectory Error of Changed System (Mass x5).

Figure 4.25: HSA Neural Network Output Versus PD Controller Output During On-Line Training of Changed System (Mass x5).

Figure 4.26: HSA RMS Error of Changed System (Mass x5).

4.2.4 Cerebral Model Arithmetic Controller (CMAC)

For this simulation, A was 200, M was 12,000,000, the learning coefficient α was 0.002, and the neighbourhood constant was 3. The computational time required for one training session, or 10,000 time-steps was 0.75 seconds of processor time. Figures 4.27, 4.28, and 4.29, show the desired and actual position, trajectory error and neural network and PD controller outputs at the end of the simulation. Figure 4.30 shows the RMS error during on-line training with the mass of the gripper arm five times its original value. The RMS error has decreased to a minimum, suggesting that the CMAC has adapted to learn the inverse dynamics of the changed system. At the end of the simulation the RMS error was 0.0727, and maximum overshoot was 20.02%. The CMAC performance shown here is still an improvement upon that of the PD controller.

4.2.5 Simultaneous Perturbation Statistical Approximation Algorithm (SPSA)

This simulation was performed in the same manner, and using the same values as the SPSA presented in Section 4.1.5, only the value of the mass of the gripper arm is different. The computational time required for one training session, or 10,000 time-steps was about 1 second of processor time. The results for the standardized trajectory are presented in Figures 4.31, 4.32, and 4.33. After 1,000 training sessions, the RMS error was 0.0373, maximum overshoot was 12.81%. Figure 4.34 shows the RMS error for each training session during the simulations.

4.3 Adaptive (Changed System) Comparison Results

For the system with gripper arm mass twice its original value: At the end of training for 1,000 training cycles, the most accurate results produced from on-line training are

Figure 4.27: CMAC Desired and Actual Position of Changed System (Mass x5).

Figure 4.28: CMAC Trajectory Error of Changed System (Mass x5).

Figure 4.29: CMAC Neural Network Output Versus PD Controller Output of Changed System (Mass x5).

Figure 4.30: CMAC RMS Error of Changed System (Mass x5).

Figure 4.31: SPSA Desired and Actual Position of Changed System (Mass x5).

Figure 4.32: SPSA Trajectory Error of Changed System (Mass x5).

Figure 4.33: SPSA Neural Network Output Versus PD Controller Output of Changed System (Mass x5).

Figure 4.34: SPSA RMS Error of Changed System (Mass x5).

	Non-Pre-Trained		Pre-Training				Adaptive Training			
			Stage One		Stage Two		Mass x2		Mass x5	
	Maximum Overshoot	RMS Error	Maximum Overshoot	RMS Error	Maximum Overshoot	RMS Error	Maximum Overshoot	RMS Error	Maximum Overshoot	RMS Error
PD	n/a	n/a	n/a	n/a	8.30%	0.0183	13.92%	0.0386	30.89%	0.1430
NANN	4.80%	0.0094	4.43%	0.0087	0.15%	4.875 $\times 10^{-4}$	1.98%	0.0046	11.24%	0.0349
HSA	37.35%	0.1854	n/a	n/a	n/a	n/a	0.68%	0.0035	10.25%	0.0317
CMAC	10.16%	0.0229	6.57%	0.0113	5.85%	0.0087	9.43%	0.0212	20.02%	0.0727
SPSA	2.57%	0.0604	n/a	n/a	n/a	n/a	2.70%	0.0058	12.81%	0.0373

Table 4.1: Changed System Comparison Results.

achieved by the History Stack Algorithm, RMS error at the end of on-line training was 0.0035, next most accurate was the NANN which produced a surprisingly accurate RMS error of 0.0046 despite never having been trained for the changed system that it was controlling, followed by the SPSA with RMS error of 0.0058. The second to least accurate result was the CMAC controller with RMS error of 0.0212. The PD controller produced an RMS error of 0.0386.

For the system with gripper arm mass five times its original value: At the end of training for 1,000 training cycles, the most accurate results produced from on-line training are achieved by the History Stack Algorithm, RMS error at the end of on-line training was 0.0317, next most accurate was the NANN which produced a still somewhat surprisingly accurate RMS error of 0.0349, followed by the SPSA with RMS error of 0.0373. The second to least accurate result was the CMAC controller with RMS error of 0.0727. The PD controller produced an RMS error of 0.1430.

For computational time per iteration the CMAC algorithm was the fastest, followed by the NANN, and then the SPSA, and as expected the HSA required the most computational time per iteration. This is due to the fact that 5 training points are selected for each time-step in the HSA. The HSA converged to its best output in the shortest number of iterations beating the CMAC and the SPSA. Results for the comparison of the simulations of the algorithms on the changed target system are summarized in Table 4.1.

The CMAC requires the least computational time, but requires orders of magnitude more memory, and cannot produce a better error than the NANN, and indeed only slightly better than the PD controller alone. SPSA performance is not better than that of the NANN or the HSA. The surprising ability of the NANN (or standard delta-rule back-propagation neural network) to produce errors almost as small as the HSA, despite never having been trained for the target system it is controlling shows how effective even basic neural networks can be as controllers. The CMAC controller only produces a slightly better controller than the PD controller alone. Lastly, simulations were run with the gripper arm mass being ten times its original value. All of the controllers became unstable. Five times the original gripper arm mass is about the limit to which the gripper arm mass can be increased before the controllers and systems become unstable.

Chapter 5

Conclusions

The results presented in the previous two chapters show that the HSA, CMAC and SPSA algorithms are all capable of on-line learning and controlling of a changing system's dynamics. The biggest surprise in performance is the ability of the non-retrained NANN to control the changed system with gripper arm mass twice its initial value. The HSA clearly produces the best performance, it converges to its minimum error more quickly than the others, and produces the smallest RMS error, and smallest percent overshoot error. The CMAC converges to its minimum error almost as quickly as the HSA but cannot produce an RMS error or percent overshoot error that is smaller than the non-retrained NANN. The SPSA converges to its minimum error fairly slowly in comparison to the CMAC and HSA, but produces a much smaller RMS error and percent overshoot error than the CMAC.

When the CMAC is not pre-trained, but is used as a controller, it causes a rather wild trajectory due to poorly trained weights. This means that it initially uses many weights. To prevent overflow of the hash table, hashing collision protection must be turned off, causing many hashing collisions, unpredictable trajectories at some times, and generally noisy CMAC neural network output.

Results from the sequentially trained, but non-pre-trained SPSA algorithm are similarly poor. It seems that as the controller for this target system, the SPSA needs a head start. As a result of not being pre-trained, the SPSA learning rule built up a bias in the SPSA weights.

The HSA was not recommended as an algorithm that could be used without pre-training, however it does seem logical that it could be used in such a manner. When not pre-trained, the output of the HSA is oscillatory in nature. This is likely due to the small size of the history stack itself, and repetitive use of its elements over a relatively small number of time-steps. The small size of the history stack causes correlation of error with time. These respective problems could be alleviated by using a larger history stack, and reducing the magnitude of the learning coefficient. These solutions would however cause slower reactions to changes in the dynamics of the target system, and training to old system dynamics that have changed.

The most accurate results produced from off-line pre-training are achieved by the Non-Adaptive Neural Network. The least accurate results from off-line training were achieved using the CMAC algorithm.

Although the HSA, CMAC and SPSA are capable of on-line learning and therefore adapting to changing target system dynamics, they are not without their respective problems. The CMAC is hardly able to do better than the PD controller in performance, the SPSA is much better than the PD controller but cannot match the NANN (which has not been retrained for the target system), and the HSA is only slightly more effective than the non-retrained NANN in performance. The HSA requires considerable computational time, about five times more computation time than the NANN does while training. The CMAC algorithm requires huge amounts of memory to store the network weights, and even though it is slightly faster than the NANN, the large amount of weights means that a large amount of computational time is spent locating and updating the weights in the hash table and realistically the amount of memory required for the CMAC, even with hashing, is too large to be implementable at reasonable cost.

Even though there are problems associated with on-line learning, neural networks can be very accurate controllers. The HSA algorithm shows that neural networks can be trained on-line to be excellent controllers of changing systems. Neural controllers can be excellent controllers, and can be designed based on the fewest number of assumptions on the plant to be controlled. To produce a truly exceptional controller,

more accuracy and speed of convergence is required than what has been observed here.

5.1 Recommendations For Future Work

The work presented here is a comparison of a few of the adaptive neural network controllers that have been proposed over the last forty years. These controllers were simply simulated to produce the results presented here. Experimental results would provide a more conclusive comparison. Although one of the neural networks was clearly the best in terms of minimal trajectory error, each has its advantages and drawbacks.

For the HSA, it seems possible that dynamically changing the history stack size and learning coefficient based on the magnitude of the torque error could produce a more effective more accurate controller, and one that could possibly learn without the necessity of being pre-trained. The CMAC is one of the older neural networks, and has many promising properties, however the learning rule is rather ineffective in comparison to the delta back-propagation rule used in the NANN and the HSA. It is also comparatively ineffective next to the SPSA learning rule. It seems possible that the SPSA learning rule could be applied to the CMAC neural network, and that this hybrid could produce interesting results. The delta rule could possibly be applied to the CMAC, or simply a second order learning rule as opposed to the first order learning rule that is typically used with the CMAC. For the SPSA itself, there have been second order learning rules proposed, and implemented, and have been shown to give slightly more accurate results than those presented here.

There are many promising advantages provided by the massive parallelism, multi-parameter and brain-like nature of the neural network controller.

References

- [1] Z. Hamavand, and H.M. Schwartz, 'Trajectory Control of Robotic Manipulators by Using a Feedback-Error-Learning Neural Network', *Robotica*, Vol. 13, 1995, pp. 449-459.
- [2] Z. Hamavand, 'Trajectory Control of Robotic Manipulators By Using a Feedback-Error-Learning Neural Network', Masters Thesis, Carleton University, Ottawa, Ontario, August 1994.
- [3] P.M. Mills, A.Y. Zomaya, M.O. Tade, 'Application of Adaptive Neural Model-Based Control', *IEEE Proceedings of the 32nd Conference on Decision and Control*, San Antonio, Texas, USA, December 1993, pp. 2804-2809.
- [4] P.M. Mills, M.O. Tade, A.Y. Zomaya, 'A Hybrid Learning System for Identification and Control', *Proceedings of the Third Annual Conference of AI, Simulation, and Planning in High Autonomy Systems*, 1992, pp. 2-8.
- [5] F. Glanz, W. Miller, G. Kraft, 'An Overview of the CMAC Neural Network', *IEEE Conference on Neural Networks for Ocean Engineering*, 1991, pp. 301-308.
- [6] W.T. Miller, R.P. Hewes, F.H. Glanz, L.G. Kraft, 'Real-Time Dynamic Control of an Industrial Manipulator Using a Neural-Network-Based Learning Controller', *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 1, 1990, pp. 1-9.
- [7] L. C. Y. Chan, S. F. Asokanthan, 'CMAC Based Controller for Hydro-Mechanical Systems', *Proceedings of the 2001 American Control Conference*, 2001, pp. 4496-4501.

- [8] C.P. Bottura, M.C.M. Teixeira, M.J. Bordon, 'Stabilization of an Inverted Pendulum with CMAC in a Variable Structure Controller', Proceedings of the 1999 American Control Conference, 1999, pp.1042-1046.
- [9] L.G. Kraft, E. An, D.P. Campagna, 'Comparison of CMAC controller Weight Update Laws', Proceedings of the 28th IEEE Conference on Decision and Control, 1989, pp. 1746-1747.
- [10] L.G. Kraft, D.P. Campagna, 'Comparison of Convergence Properties of CMAC Neural Networks and Traditional Adaptive Controllers', Proceedings of the 28th IEEE Conference on Decision and Control, 1989, pp. 1744-1745.
- [11] Y. Maeda, 'Real-time control and learning using neuro-controller via simultaneous perturbation for flexible arm system', Proceedings of the American Control Conference, Anchorage, Alaska, USA, May 2002, pp. 2583-2588.
- [12] J. C. Spall, 'A Stochastic approximation technique for generation of maximum likelihood parameter estimates', Proceedings of the 1987 American Control Conference, 1987, pp. 1161-1167.
- [13] J. C. Spall, 'Multi-variable stochastic approximation using a simultaneous perturbation gradient approximation', IEEE Transactions on Automatic Control, Vol. 7, 1992, pp. 332-341.
- [14] Y. Maeda, H. Hirano, and Y. Kanata, 'A learning rule of neural networks via simultaneous perturbation and its hardware implementation', Neural Networks, Vol 8, 1995, pp. 251-259.
- [15] K. Patan, T.Parisini, 'Stochastic Learning Methods for Dynamic Neural Networks: Simulated and Real-Data-Comparisons', Proceedings of the American Control Conference, Anchorage, Alaska, USA, May, 2002, pp. 2577-2582.
- [16] Y. Maeda, R.J.P. De Figueiredo, 'Learning Rules for Neuro-Controller via Simultaneous Perturbation' IEEE Transaction on Neural Networks, Vol. 8, No. 5, 1997, pp. 1119-1130.

- [17] M. J. Atallah, *Algorithms and Theory of Computation Handbook*, CRC Press, New York, 1999, pp2-7, 2-8.