University of California

Los Angeles

Artificial Neural Control for Nonlinear Systems

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

E. Robert Tisdale

1996

The dissertation of E. Robert Tisdale is approved.

Kirby A. Baker

David A. Rennels

Jack W. Carlyle

Milos Ercegovac, Committee Chair

University of California, Los Angeles

1996

To Joseph Skrzypek who worked on the hard problems.

Table of Contents

1 Introduction **1**

1.1 Problem 1

1.2 Approach 2

1.3 Test Problem 3

1.4 Scope 6

1.5 Organization 6

2 What Are Neural Networks? **9**

2.1 Computational Network 11

2.2 Properties 11

2.2.1 Simple Processing Units 11

2.2.2 Massive Parallelism 13

2.2.3 High Connectivity 13

2.3 Natural Neural Networks 14

2.4 Artificial Neural Networks 16

2.5 Learning 18

3 Artificial Neural Control **21**

3.1 Nonlinear Control Systems 21

3.2 Real-Time Recurrent Learning 27

3.3 Derivative Arithmetic 29

3.3.1 The Aircraft Model 31

3.4 The Backprop Algorithm 31

3.4.1 Feed-Forward 33

3.4.2 Backward Error Propagation 34

3.4.3 Update Weights and Biases 36

3.4.4 Partial Derivatives 36

3.5 Second Gradients 39

3.6 Normalization 46

3.7 Summary 48

4 Preliminary Investigations 50

5 Teaching Neural Networks to Fly Turns 58

5.1 Introduction 58

5.2 Configuration 61

5.2.1 Desired State 61

5.2.2 Network Inputs and Outputs 63

5.3 Training 67

5.3.1 The One Layer Network 68

5.3.2 The Two Layer Network 78

5.4 Stability 83

6 Conclusions 88

6.1 Derivative Arithmetic 89

6.2 Automatic Controller Design 90

6.3 Natural Neural Controllers 94

A Second Order Real-Time Recurrent Learning 95

A.1 Derivative Arithmetic 95

A.2 Real-Time Recurrent Learning 96

A.3 Backprop 97

References 100

List of Figures

1.1 The AIAA Aircraft Controls Design Challenge model is a very realistic computer flight simulator for high performance fighter aircraft. 4

2.1 A directed graph represents a computational network for the cross product $a \times b = c$ where $a_y b_z - a_z b_y = c_x$, $a_x b_z - a_z b_x = c_y$ and $a_x b_y - a_y b_x = c_z$. Numerical values are transmitted along the arcs to simple processing units at the nodes which compute products (*) and differences (-). 12

3.1 The nonlinear system which is to be controlled is called the plant. The block labeled z^{-1} represents a delay until the next time step. 22

3.2 The observer estimates the system state from the observable plant outputs and feeds them back to the controller. 23

3.3 An extended Kalman filter can be used to estimate the system state variables for a quasi-linear system. 23

3.4 The training algorithm adjusts the parameters in the control function so as to reduce the difference between the actual and desired output of the plant. 24

3.5 The system identification process maintains an accurate model of the system dynamics. 25

3.6 A simulator is used off-line to train an artificial neural network to perform the control function for a nonlinear system. Both the artificial neural network and the state transition function compute partial derivatives which the real-time recurrent learning algorithm use to adjust the network connection weights and biases at each time step. 26

3.7 Layer k computes $\sigma(b_k + W_k v_k = a_{k+1}) = v_{k+1}$. 33

3.8 The backward error propagation algorithm is used to adjust the biases and connection weights in multi-layer feed-forward artificial neural networks. The signals, v_k, feed forward through K layers. The output of each hidden layer is an input for the next layer. The equivalent errors, δ_k, feed backward through the layers and are used to update the biases, b_k, and the connection weights, W_k. 35

3.9 Gradient descent is unstable for $J(\theta) = \frac{1}{2}\theta^2$ when $\mu = 3$. 40

3.10 The performance function, $J(\theta)$, will generally contain a number of minima separated by maxima with inflection points between them. 42

4.1 A simple conventional controller was provided along with the flight simulator. Variable names from the computer simulation were used to label the signals and gains. The symbols represent nonlinear limiting functions. The Δ symbols represent unit time delays. . . 51

4.2 The first step in transforming the conventional controller into a neural controller replaces each of the nonlinear limit functions with equivalent normalized semi-linear functions (symbols) then scales the signal gains and adjusts the biases appropriately. 53

4.3 The second step in transforming the conventional controller into a neural controller replaces each of the internal state feed-back loops with equivalent loops which feed the outputs back into their respective subunits. 53

4.4 The third step in transforming the conventional controller into a neural controller places the feed-back gains and biases in the input layer and duplicates the computation of H2 and V2. The feed-forward network computes the internal state of the controller from delayed controller inputs and outputs. The layers are fully connected but only the non-zero biases and connection weights are shown. 55

4.5 The artificial neural network never learned to control altitude. The plot represents the best simulation result obtained. Most other simulations would end in less than five minutes by crashing the airplane. 56

5.1 The controller functions as a flight test maneuver autopilot. The artificial neural controller is trained to fly a 2 g coordinated right turn at Mach 1 while maintaining an altitude of 20 000 feet. The turn requires about half a minute to complete and has a radius of about 4 miles. The desired state of the aircraft is specified at intervals of 1.0 second (about 0.2 miles) ahead of the aircraft in order to permit the controller to look ahead. 59

5.2 The artificial neural network learned to fly a turn. 69

5.3 An artificial neural controller with just one layer of biases and connection weights (a linear controller) learned to fly turns with a radius of four miles at Mach 1 and 20 000 feet. The actual heading never deviates from the desired heading by more than about seven degrees. 74

5.4 The artificial neural controller must bank the aircraft to the right about 60 degrees in order to make a 2 g right turn. The turn is unsteady because the artificial neural controller attempts to regain lost altitude by rolling back to the left. 75

5.5 The aircraft looses over twenty feet per second in airspeed and almost one hundred feet in altitude. 76

5.6 The artificial neural controller applies about 0.3 degrees right rudder to minimize sideslip. 77

5.7 A bias and connection weight in the artificial neural controller were altered to control the throttle in the same manner as the conventional controller. This resulted in a more steady but tighter turn which slowly improved with training. 79

5.8 The two layer artificial neural controller did not perform much better than the single layer artificial neural controller. The actual heading exceeds the desired heading by more than seven degrees. The aircraft rolls out of the turn too late and overshoots the desired heading at the end of the maneuver. It looses almost 14 feet per second in airspeed and 180 feet in altitude which it does not completely regain until after it rolls out of the turn. 82

5.9 Mild air turbulence was modeled by adding noise $\dot{\nu}_{t+1}$ to $\dot{\alpha}_{t+1}$ and $\dot{\beta}_{t+1}$. The Fourier cosine amplitudes of the accumulated noise $\nu_{t+1} = \nu_t + \Delta t \dot{\nu}_{t+1}$ are shown above. 84

5.10 The artificial neural controller does not permit the affect of mild turbulence to accumulate. It acts to maintain the desired angle of attack and minimize sideslip. 85

5.11 Although the aircraft does complete the turn, the artificial neural controller is seriously affected by mild turbulence. 86

5.12 The artificial neural controller immediately suppresses the transients errors which occur when the simulation is initialized. The total weighted error increases before the maneuver begins. The controller trades airspeed for altitude anticipating a loss in altitude during the turn. It take advantage of the look ahead capability provided by the reference signal by starting to roll into the turn before the maneuver begins and roll out of the turn before the maneuver ends. 87

List of Tables

3.1 The aircraft model has only twelve state variables. 32

3.2 The aircraft model has only six control variables. 32

3.3 Summary of the real-time recurrent learning algorithm. 49

5.1 The relative importance of the difference between the desired and actual values of the aircraft state variables are set to zero if the difference has no importance. Otherwise, they are arbitrarily set to one. 62

5.2 Initial estimates for the network input biases and variations were obtained from various sources. 64

5.3 Initial estimates for the network output biases and variations were obtained from straight and level flight with the conventional controller. 65

5.4 The relative importance of the difference between the desired and actual values of the aircraft state variables was revised twice – once after the artificial neural controller had learned to fly turns with a 64 mile radius and again after it had learned to fly turns with a 4 mile radius. 71

5.5 Initial estimates for the network input variations were revised twice – once after the artificial neural controller had learned to fly turns with a 64 mile radius and again after it had learned to fly turns with a 4 mile radius. 72

5.6 Initial estimates for the network output variations were revised after the network had learned to fly turns with a 4 mile radius. . . 80

Acknowledgments

Lee Duke who first proposed the idea of using artificial neural networks to design flight test maneuver controllers automatically.

Abstract of the Dissertation

Artificial Neural Control for Nonlinear Systems

by

E. Robert Tisdale

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1996

Professor Milos Ercegovac, Chair

Research was conducted to test a new automatic design methodology for nonlinear controllers that can be used when an accurate conventional computer model exists for the plant (the nonlinear system which is to be controlled). The real-time recurrent learning algorithm was employed to train an artificial neural network to perform an unknown nonlinear control function which obtains the desired behavior from the plant. Two different methods are employed to compute partial derivatives for the real-time recurrent learning algorithm. Derivative arithmetic was used to compute the partial derivatives of the state variables at the next time step with respect to the state and control variables at the current time step. The backward error propagation algorithm was used to compute the partial derivatives of the control variables with respect to the current state variables and the control system parameters which are simply the network biases and connection weights.

The methodology was applied in the design of a flight test maneuver autopilot for high performance fighter aircraft. An artificial neural controller was installed in a realistic computer flight simulator and trained to fly coordinated 2 g turns while maintaining altitude and airspeed. It should be possible to teach an artificial neural controller to fly almost any standard flight test maneuver but training would probably require considerable engineering insight and knowledge of the system to configure the network and learning algorithm. Of course, this compromises the ultimate goal which is to completely automate the controller design process.

CHAPTER 1

Introduction

The objective of this research is to automate the design of nonlinear controllers for nonlinear systems. Computers are already used extensively in conventional controller design[Lew92] to assist the design engineer with some of the more tedious computational tasks. Automation of the entire design process is limited mostly to linear controllers for linear systems for which there is a large and elegant body of control theory. But because there is no comprehensive theory for nonlinear systems[NP], engineers are forced to invent ad hoc designs to control nonlinear systems. These designs usually require extensive testing then considerable refinement and adjustment in order to obtain acceptable performance while maintaining stability and reliability.

1.1 Problem

It isn't always possible to derive an expression which describes the control function for a nonlinear system analytically. The system is often described by a nonlinear differential equation, $\dot{x} = f(x, u)$, which implies the simultaneous solution of a system of nonlinear equations for the control inputs, u, required to produce the desired rate of change, $\dot{x}$, in the state variables, x. The desired rates of change are determined in turn by solving the differential equation beginning with some initial state and terminating in the desired state some time later.

Usually, only numerical solutions are possible.

1.2 Approach

One solution is to approximate the control function by a universal model with many adjustable parameters. This so-called black box approach requires no explicit knowledge of the control function. If we know how to change the parameters to improve control over the system, they can be adjusted incrementally until satisfactory behavior is obtained. This is precisely the approach that we have taken except that we avoid the time and expense of computation required for the more complicated universal models and restrict ourselves to those universal models which are artificial neural networks. The adjustable parameters in an artificial neural network are the biases and connection weights, and the process of estimating these parameters is called learning.

The idea of using universal models is not new but only became practical with the advent of high speed digital computers which could be used for parameter estimation. After fast, powerful computers became widely available around 1980, thousands of new artificial neural network applications appeared in the literature. The fact that these successes represent only a tiny fraction of attempts received little attention. Researchers seldom report their failures. They simply continue to search until they find a problem which they can solve. The methodology seems to work well for simple "toy" problems[NW90] and quasi-linear systems[FPD91, JS90] but has not yet been proven for complicated, highly nonlinear, real-world control problems.

The problem is that a large number of parameters are required to approximate complicated functions. Even if a method for estimating optimal values for each of the parameters can be devised, the complexity of the computation might be exponential in the number of independent parameters. In practice, the problem is circumvented using heuristic search techniques[1]. But these techniques cannot generally guarantee that an optimal or even a satisfactory set of values will be found for all of the parameters.

1.3 Test Problem

In order to test our methodology on an important real-world problem, we attempt to design flight test maneuver autopilots for high performance experimental fighter aircraft. Some flight test maneuvers are particularly difficult for human test pilots to fly, especially if the vehicle is a remotely piloted scale model. Maneuver autopilots have already been employed for this purpose[DJR86]. But autopilot design and development costs which are normally amortized over hundreds of production aircraft are usually prohibitive for one or two experimental models. Our hope is that an automatic design methodology would be cost effective for flight testing even if the resulting controller was not safe enough for commercial use.

Although high performance aircraft can be very complicated nonlinear systems, they should not be very difficult to control. Aircraft can tolerate a great deal of control error except when taking off or landing. There are only about a dozen state variables, half a dozen control variables and almost no hard constraints. The instruments which are standard equipment in almost all aircraft provide accurate information about almost all of the system state variables.

There is very little unclassified flight test data for high performance exper-

[1]Here, the term heuristic refers to methods such as hill-climbing which apply to search in a continuous multidimensional space and not just to search in graphs described by Pearl[Pea84].

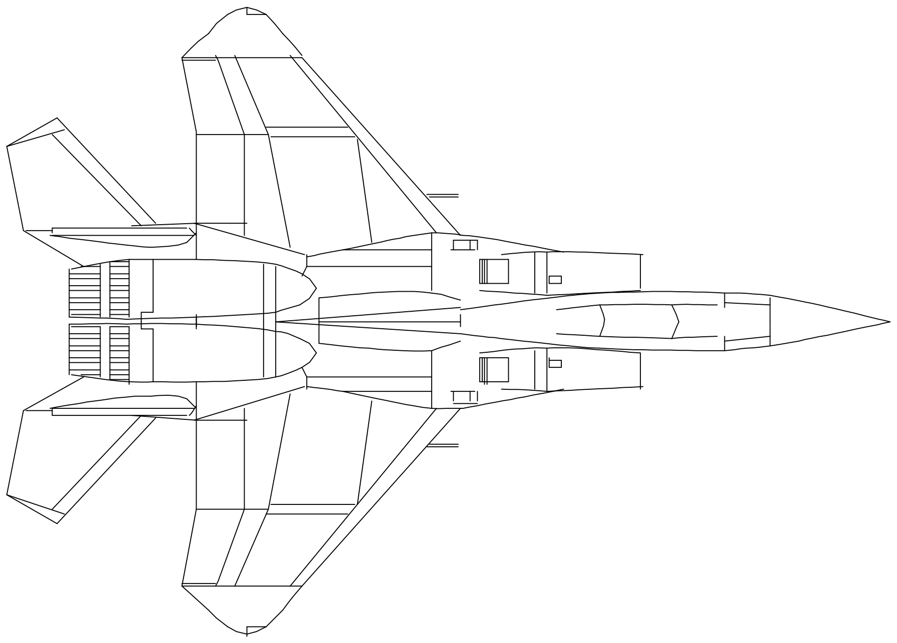


Figure 1.1: The AIAA Aircraft Controls Design Challenge model is a very realistic computer flight simulator for high performance fighter aircraft.

imental fighter aircraft, but there are very realistic flight simulators for high performance fighter aircraft like the one shown in figure 1.1. These simulators can be used to generate training data for an artificial neural controller. We use the AIAA Aircraft Controls Design Challenge model which is a computer program distributed as FORTRAN 66 source code from the NASA Dryden Flight Research Facility at Edwards Air Force Base.

We use the real-time recurrent learning algorithm to train the artificial neural controller. This algorithm employs a first order gradient descent method which requires estimates of the partial derivatives of the system state variables with respect to the state and control variables at the previous time step. These estimates can be obtained by a number of different methods.

One method is to differentiate the mathematical model symbolically. This usually results in an enormous number of complicated expressions which must be converted into code. Some researchers employ sophisticated symbolic manipulation software to eliminate errors and redundant code, but must repeat the whole process if there are any changes in the model. Others attempt to simplify the model in order to make the problem more manageable, but find it difficult to prove that the simplified model adequately represents the system. So the validity of the model is compromised.

Another method is to differentiate the computer model numerically as suggested by Dieudonne[Die78]. The system state variables are computed at the next time step for small perturbations in the the current state and control variables. The partial derivatives are estimated by the ratios of the differences. But this method is generally unreliable because it is too sensitive to small rounding errors.

Another method suggested by Barto[Bar90], Narendra[NP90] and Widrow[NW90] is to train a multi-layer feed-forward artificial neural network to model the system. If this works, the partial derivatives are easily and efficiently computed using the backward error propagation (backprop) algorithm[RHW86]. The network may also have the flexibility required to account for modeling errors and unmodeled behavior in an on-line adaptive control system. But this method discards all knowledge and understanding of the plant and is seldom sufficiently accurate for complicated nonlinear systems.

Because we have the source code for the computer model, we can use *derivative arithmetic* to compute the partial derivatives. Derivative arithmetic replaces the normal computer arithmetic with operations that compute and propagate the partial derivatives along with the normal results. This simple idea is the main original contribution of this research. It permits us to compute partial derivatives for the complete computer model without maintaining any additional source code.

1.4 Scope

The scope of this research is limited to artificial neural networks which are used as "black box" controllers for nonlinear systems for which an accurate conventional computer model already exists. It does not address the problem of system identification which, more generally, would be required to develop and validate a conventional computer model. The discussion provides just enough background in control system and artificial neural network theory to help orient readers who might not be experts in either discipline. The research is limited to artificial neural networks in engineering applications and any extension to natural neural networks is purely speculative.

The scope of the research is further restricted to controllers which do not, themselves, have internal state. The method requires that the source code for the computer model is available which means that the internal state of the system is accessible so no "observer" processes are required to estimate the state of the system. Conventional computer models are usually not easily adjustable so only accurate, valid conventional computer are considered and adaptive control is not relevant. re

1.5 Organization

Chapter 2 presents a working definition for neural networks in terms of the three properties which distinguish them from other computational networks. It includes a brief description of both natural and artificial neural networks and introduces learning as a process of statistical parameter estimation.

Chapter 3 details the mathematical models used for artificial neural control. Section 3.1 briefly describes nonlinear control systems and introduces some of the notation used in the sequel. Section 3.2 formally describes the real-time recurrent learning algorithm. Section 3.3 introduces derivative arithmetic and explains how it is used with the computer model. Section 3.4 reviews the backward error propagation algorithm and shows how it is used to compute the partial derivatives of the feed-forward network outputs with respect to network inputs as well as the network biases and connection weights. Section 3.5 shows how the second gradient might be employed to improve the performance and stability of a gradient descent method. Section 3.6 shows that the practice of "normalizing" the network inputs attempts to achieve the same effect as a constant second gradient method. Section 3.7 summarizes the real-time recurrent learning algorithm including normalization.

Chapter 4 describes the construction of an artificial neural controller by manipulating the gains employed in a simple conventional controller which was provided along with the flight simulator. This was important for two reasons. First, it demonstrated that an artificial neural network can control the aircraft. Second, it showed that a feed-forward network architecture was appropriate for the neural controller.

Chapter 5 describes the experiments that were designed to teach an artificial neural controller to fly turns along with the results of those experiments. Section 5.1 details the test maneuver. Section 5.2 describes the configuration of the artificial neural controller and the real-time recurrent learning algorithm. It specifies the details of the desired state, network inputs and outputs, input biases and variations, reference signal, output biases and variations, network initialization and input normalization. Section 5.3 presents the experiments and their results.

First, a one-layer artificial neural controller is taught to fly shallow turns then statistics generated by the simulator are used to revise parameters used by the learning algorithm. The artificial neural controller is taught to fly successively steeper turns until it can fly 2 g turns. The one layer artificial neural controller is embedded into a two layer network with a single hidden layer of identical nonlinear sigmoidal processing units. The two layer artificial neural controller performs almost exactly like the one layer network but it has a much greater capacity for learning. Section 5.4 presents an experiment designed to test the affect of noise (mild air turbulence) on the ability of the artificial neural controller to fly to fly 2 g turns. The controller does not act to reduce the difference between the desired and actual state but attempts to reduce the total squared error over the entire maneuver.

Chapter 6 presents the conclusions which can be drawn from this research. The artificial neural controller flies turns about as well as any human novice pilot. It should be possible to train artificial neural controllers to fly almost any standard flight test maneuver. Unfortunately, the methodology represents little improvement in automated controller design. The main problem seems to be that the first order learning algorithm is either too slow or unstable and we do not have sufficient computational resources to employ a higher order learning algorithm.

CHAPTER 2

What Are Neural Networks?

There is no universally accepted definition for neural networks. Most people have formed very vague notions about neural networks which they are unable to articulate but sometimes describe in terms of ideas that have been associated with them in the current literature. This is barely noticeable in casual conversation but in more serious discussions, people often cannot determine whether or not they disagree with each other much less whether or not some statement made about neural networks is accurate. There must be a clear, concise working definition for neural networks but so far they have eluded the search for a perfect sound bite which sums them up in a nutshell. The following definition:

> A neural network is a computational network with the following three properties:
>
> 1. simple processing units,
> 2. massive parallelism and
> 3. high connectivity.

abstracts the essential properties of neural networks without making any assumptions which unnecessarily narrow the definition but it requires considerable elaboration.

Many features commonly associated with neural networks are conspicuously absent from this definition.

- It does not imply that artificial neural networks which are the inventions of engineers are in any way derived or even inspired by natural neural networks which are the nervous systems of animals but applies equally to both artificial and natural neural networks.

- It does not mention learning. Despite the fact that the bulk of the literature on neural networks (including this discussion) concerns itself with learning, most neural networks do not learn. The function of natural neural networks is almost entirely determined by nature during development and changes very little over the life span of the host organism. As technology evolves, the current fascination with machines that learn should fade as engineers turn their attention to exploiting the raw computational power of artificial neural networks.

- It does not specify the function of the processing units. In particular, it does not specify uniform sigmoidal functions applied to a weighted sum of inputs. Each processing unit may be different and implement any function including a discontinuous or unbounded function of the inputs. The processing units in artificial neural networks bear little resemblance to the neurons in natural neural networks even when artificial neural networks are used to model the computational behavior of natural neural networks.

- It does not specify the network architecture. In particular, it does not specify a multi-layered feed-forward network architecture. The multi-layered feed-forward network architecture is merely a mathematical contrivance which is used to represent all other neural network architectures.

- It does not specify whether numeric or symbolic, analog or digital, continuous or discrete signals propagate through the network. Any combination of signaling methods is possible.

2.1 Computational Network

A computational network is the abstraction of a computer architecture which can be represented by a directed graph. Processing units at the nodes (vertices) operate on inputs transmitted along arcs (edges) directed into the node and produce outputs transmitted along arcs directed out of the node. Figure 2.1, for example, shows the diagram for a computational network for a simple arithmetical computation – the cross product $a \times b = c$ where $a_y b_z - a_z b_y = c_x$, $a_x b_z - a_z b_x = c_y$ and $a_x b_y - a_y b_x = c_z$.

2.2 Properties

Neural networks are distinguished from other computational networks only by the complexity, parallelism and connectivity of the processing units but there are no precise thresholds which discriminate between them. It is not as important to distinguish border line cases as it is to perceive the general relationship. A few examples of each property should suffice to establish good intuition about the differences.

2.2.1 Simple Processing Units

A processing unit which computes an elementary arithmetical operation (addition, subtraction, multiplication or division) is regarded as simple whereas few

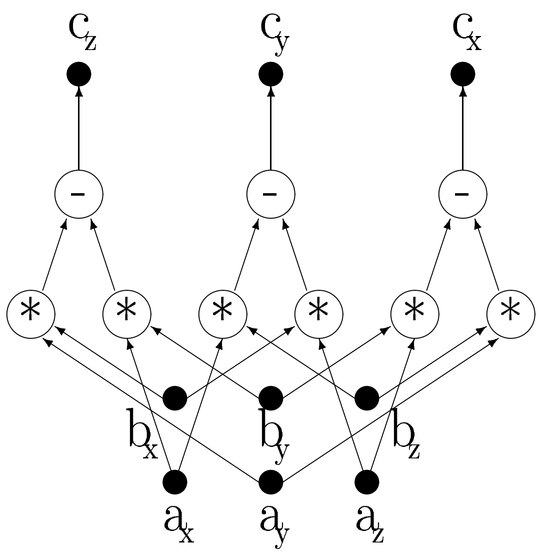


Figure 2.1: A directed graph represents a computational network for the cross product $a \times b = c$ where $a_y b_z - a_z b_y = c_x$, $a_x b_z - a_z b_x = c_y$ and $a_x b_y - a_y b_x = c_z$. Numerical values are transmitted along the arcs to simple processing units at the nodes which compute products (*) and differences (-).

people think that a general purpose microprocessor is simple. Neurons are the processing units in natural neural networks and are presumed to be simple even though no one is quite sure exactly what each one computes. It is generally assumed that the same value is distributed to every output of a simple processing unit because a processing unit which can compute and distribute different values for each of its outputs probably isn't simple.

2.2.2 Massive Parallelism

If the computations performed by any two processing units are independent of each other, they can execute in parallel. No particular timing mechanism is specified. The network may be implemented synchronously or asynchronously. A computational network which computes the matrix product $Ax = y$ where x and y are each vectors with thousands of elements is massively parallel because each element of y can be computed independently of every other element of y. The computational network which computes the cross product $a \times b = c$ has simple processing units and high connectivity but it isn't massively parallel because only three elements are computed.

2.2.3 High Connectivity

Adding connections increases the connectivity until it approaches the limiting case where there is a dedicated direct connection from each processor to every other processor. Typically, neurons are directly connected to thousands of other neurons in natural neural networks. Current technology is inadequate for building artificial neural networks with connectivity approaching that of natural neural networks. Processors in today's massively parallel computers have direct connections to only about a dozen of their nearest neighbors.

2.3 Natural Neural Networks

There are so many different kinds of natural neural networks that no one claims to know how they all work but we know enough about some of them to give a fairly accurate description of their behavior. The typical neuron in a vertebrate animal has three parts:

1. the soma or cell body which contains the nucleus,
2. a collection of branching nerve fibers called dendrites which transmit nerve impulses into the soma and
3. the axon which transmits nerve pulses out of the soma and may be quite long but eventually branches into terminal fibers which connect to the dendrites of other neurons through synaptic junctions.

The nerve impulses are changes in electrical potential across the cell membrane which may increase by about one hundred millivolts when they spike then decay again within a few milliseconds. They originate in the soma then travel on the order of ten meters per second along the axon and are broadcast over the terminal fibers. Nerve impulses are amplified (or attenuated) by the synapses then transmitted along a dendrite to the soma of the next neuron. Apparently, the soma behaves like a leaky integrator. If strong nerve impulses arrive at the soma at a sufficiently high rate, the potential across the cell membrane will increase until it reaches a threshold potential at which it begins to fire nerve impulses out along the axon. Usually, the rate of firing increases with increasing cell membrane potential but eventually saturates at some maximum rate.

No one knows what these nerve impulse signals really mean or how a natural neural network uses them to compute but there is one interpretation which explains how they can be used to compute anything that can be computed. This interpretation maintains that only the average firing rate of neurons is important and not the details or timing of individual nerve impulses. The firing rate is a frequency modulated signal which may have been naturally selected because it is less susceptible to interference, cross-talk and other noise in the neural environment. The synapses act as weights, w_i, for each input signal, x_i, and the leaky integrator effectively performs a simple sum of these weighted inputs so that the membrane potential at the soma is proportional to $\sum_i w_i x_i$ and the neuron fires at a rate proportional to $\sigma\left(\sum_i w_i x_i\right)$ where $\sigma(\cdot)$ is a sigmoid function. Almost any function can be approximated by a linear combination of the outputs, $y_j = \sigma\left(\sum_i w_{ij} x_i\right)$, given enough neurons and an appropriate set of connection weights, w_{ij}. The exact form of the sigmoid is unimportant.

A diagram of a natural neural network resembles a graph of a computational network except that people tend to associate the products, $w_{ij} x_i$, with the arcs because the synapses lie between the somata at the junction between dendrite and axon. Each neuron has perhaps as many as ten thousand synaptic connections to other neurons. If the human nervous system has as many as one hundred billion neurons, there could be upwards to 10^{15} synaptic connections. The somata are essential but their exact function can be safely ignored as long as it is consistent. It is the tiny but much more numerous synaptic connections that determine the exact function of the network. Still, it is not clear whether there are enough of them to account for all of the complexity of human behavior. Using this model for neurons, a digital computer capable of at least one million trillion floating-point multiplications each second is required to simulate the human brain in real time updating once each millisecond. Machines capable of one trillion floating-point operations each second are being introduced now. If performance continues to increase by a factor of ten every three years, the minimum required computational power should be available within two decades.

2.4 Artificial Neural Networks

Virtually all artificial neural networks are simulated on general purpose serial computers. Sometimes special purpose simulation accelerators are employed. A few attempts have been made to implement the network itself in hardware using electrical analog circuitry. Simple resistors, R_{ij}, tap inputs, u_i, which are electrical potentials and conduct electrical currents, u_i/R_{ij}, to an RC integration circuit at the input to an operational amplifier which may have a nonlinear output.

Progress in the implementation of artificial neural networks in hardware has been slow partly because of the limitations of current technology. It is difficult to manufacture precision resistors on a silicon substrate. Also, the two dimensional architecture severely restricts the number of connections between processing units. The connections occupy most of the surface of a chip leaving little room for the processing units. But an even better explanation is that development of artificial neural network hardware has been inhibited by competition from rapidly improving general purpose serial computers which can be used to simulate artificial neural networks.

Any neural network can be represented by an equivalent multi-layer feedforward artificial neural network. For this network architecture, the outputs, x_{k+1}, of layer $k+1$ are functions, $f_{k+1,i}(x_k) = x_{k+1,i}$, of the inputs, x_k, from layer k. The $f_{k+1,i}(\cdot)$ may all be different from each other and may be any linear or nonlinear function including $f_{k+1,i}(x_k) = x_{k,j}$ which simply passes an input through to the next layer. If the $f_{k+1,i}(\cdot)$ are functions of the activation, $a_{k+1,i} = \sum_j w_{k,ij} x_{k,j}$, then the network is called a multi-layer perceptron. An activation function, $f_{k+1,i}(a_{k+1,i})$, may be any function including a sigmoidal activation function, $\sigma_{k+1,i}(a_{k+1,i})$. A sigmoid is any bounded function which approaches a different value when the argument approaches positive infinity than when the argument approaches negative infinity. Any function with this general S-shaped appearance will do but other properties such as monotonicity or continuous, bounded derivatives are valued because they simplify learning algorithms.

Usually, an extra constant input, $x_{k,0} = 1$, is included in each layer so that the associated connection weights, $w_{k,i0}$, acts as a bias. As the weighted sum of all the other inputs increases past the threshold $-w_{k,i0}$, the activation changes sign from negative to positive. Common examples of sigmoid functions are the unit step function

$$\sigma(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } 0 \leq x \end{cases} \tag{2.1}$$

and the logistic function

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.2}$$

The problem with using these functions is that the output has a large bias which is highly correlated with the bias in every other input to the next layer especially the constant input. If there are very many such inputs, it usually means that the biases must be large but the connection weights must be small. If a first order gradient descent method like backprop is used to estimate the biases and connection weights, the learning rate must be reduced to estimate the small connection weights accurately which usually slows progress in approximating the large biases. A better choice of sigmoid functions is the signum function

$$\sigma(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } 0 \leq x \end{cases} \tag{2.3}$$

and the hyperbolic tangent function

$$\sigma(x) = \frac{e^{+x} - e^{-x}}{e^{+x} + e^{-x}}. \tag{2.4}$$

The output of the hyperbolic tangent is approximately equal to its argument if the argument is a very small value and approximately equal to the output of the signum function if the argument is a very large positive or negative value. Simply scaling the connection weights and biases appropriately permits it to serve either as a linear function or a step function.

A multi-layer perceptron can approximate any bounded continuous function with just one hidden layer of sigmoidal activation functions but may require an enormous number of units to obtain the desired accuracy. It is usually possible to maintain accuracy using fewer hidden units by adding additional hidden layers.

2.5 Learning

Learning is simply the process of estimating system parameters as employed in statistics, filter theory, systems science, operations research and a host of other disciplines – each of which has developed its own redundant jargon more or less in ignorance of the others. The process begins with a mathematical model $f(x;\theta) \approx y(x)$ for a system where the actual outputs, y, are approximated by a function, $f(\cdot)$, of the actual input variables, x, and adjustable system parameters, θ. The problem is to find a θ that minimizes the expected cost of the error $y - f(x;\theta)$ over all possible x. If the system is linear, $Ax = y$, there is a simple, direct method for estimating the matrix A which represents the system parameters. If the system is nonlinear, there is no general method for minimizing the error short of testing every possible θ. Even if the test is simple and the precision of each element of θ can be restricted, the complexity grows exponentially with the number of elements in θ and the problem quickly becomes intractable. Some sort of heuristic search is required to overcome this barrier. This usually means some kind of gradient descent or hill climbing algorithm when the elements of

θ represent continuous parameters. Although there have been some significant theoretical advances, the methods we use to optimize θ haven't changed since they were invented by Karl Friedrich Gauss nearly 200 years ago. What has changed is that we now have powerful computers which we can use to compute accurate estimates for many more parameters.

A mathematical model may be derived from first principles if everything about the system is known. But if the system is essentially a black box about which nothing is known except the inputs and outputs, universal models may be employed to approximate the behavior of the system. The most simple approach is to use a linear combination, $\sum \theta_i f_i(x) \approx y(x)$, of functions, $f_i(\cdot)$, which form a basis for the function space. In general, an infinite set of functions is required to span the function space so the trick is to pick a basis from which a small subset of functions are required to adequately approximate the behavior of the system. For example, the series $\sum \theta_i x^i = y$ truncated to n terms can be used to approximate any reasonable smooth, bounded function on the unit ball. Because the model is a linear function of the parameters, there is a simple direct method for estimating them. Another approach is to parameterize the basis functions themselves as in $\sum a_i x^{n_i} = y$ where the n_i are adjustable exponents. This complicates the parameter estimation process because the model is a nonlinear function of the n_i but it effectively permits a better choice of basis functions.

At first glance, a multi-layer perceptron with sigmoidal activation functions seems to be an unlikely model for anything except perhaps a natural neural network. In fact, it will serve as a universal model for any black-box system, $y(x)$, which is a function only of its inputs, x. The system parameters, θ, are simply the biases and connection weights, $w_{k,ij}$ and the processes of estimating them is called learning. If the system parameters are estimated from training data which are pairs of random vectors, (x, y), then the process is, at best, simply statistical estimation. In general, there must be at least as many distinct, independent training pairs as there are biases and connection weights in the network.

Learning in natural neural networks remains a matter of considerable speculation. There is strong evidence that the strength of synaptic connections is modified by learning but the mechanism which effects these changes is unclear. There is virtually no evidence for the kind of feedback network that seems to be essential for learning in artificial neural networks but this does not rule out communication via some other medium such as chemical diffusion.

CHAPTER 3

Artificial Neural Control

3.1 Nonlinear Control Systems

Our research is concerned with discrete-time difference equations which represent the equivalent continuous-time, ordinary differential equations on digital computers. A nonlinear system can be represented by a state transition equation,

$$x_{t+1} = f(x_t, u_t), \tag{3.1}$$

where the system state, x_{t+1}, at the next time step, $t+1$, is a nonlinear function, $f(\cdot)$, of the the system state, x_t, and the control inputs, u_t, at the current time step, t, and a measurement equation,

$$y_t = g(x_t), \tag{3.2}$$

where the observable system outputs, y_t, are a nonlinear function, $g(\cdot)$, of the system state, x_t. The nonlinear system which is to be controlled is called the *plant.* A block diagram of the plant is shown in figure 3.1.

The nonlinear system which controls the plant is called the controller. The controller outputs,

$$u_t = c(x_t, r_t; \theta_t), \tag{3.3}$$

are a nonlinear function, $c(\cdot)$, of the system state, x_t, the reference signal, r_t, and numerous adjustable parameters, θ_t. This is the function that we wish to

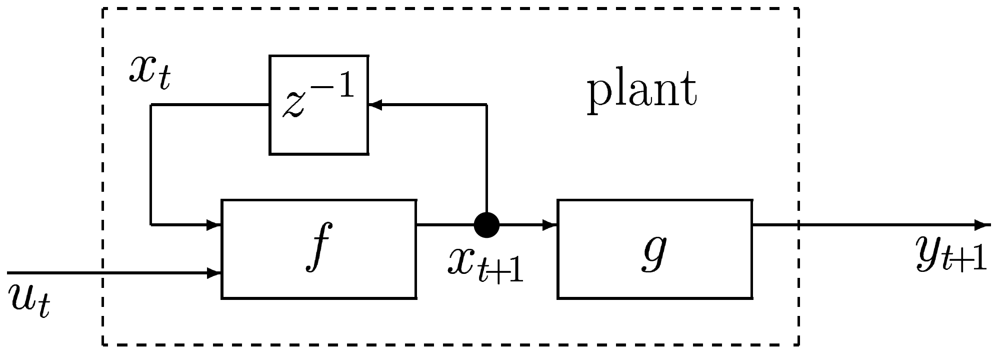


Figure 3.1: The nonlinear system which is to be controlled is called the plant. The block labeled z^{-1} represents a delay until the next time step.

approximate with an artificial neural network. The adjustable parameters are the network biases and connection weights. More generally, the control function is also a function of the internal state of the controller. However, these controllers are beyond the scope of this research since the stability of the internal states can be very sensitive to small parameter estimation errors.

The state of the system to be controlled must be *observable*. However, the system state generally cannot be measured directly, and another process, called the *observer*, is required to generate an estimate of the system state, $\hat{x}_{t+1}$, from the observable system outputs, y_{t+1}. The block diagram in figure 3.2 shows the signal flow from the controller to the plant to the observer and back again to the controller completing the feedback control loop.

More generally, the output of the observer, $\hat{x}_{t+1} = \phi(y_{t+1}, \hat{x}_t, u_t)$, will be a nonlinear function, $\phi(\cdot)$, of the plant output, y_{t+1}, the previous state estimate, $\hat{x}_t$, and the previous control input, u_t. For example, if the plant is a linear or quasi-linear system, a Kalman filter or an extended Kalman filter can perform the function of the observer as shown in the block diagram in figure 3.3. The innovations, $\alpha_{t+1} = y_{t+1} - \tilde{y}_{t+1}$, are used to improve the estimate of the plant

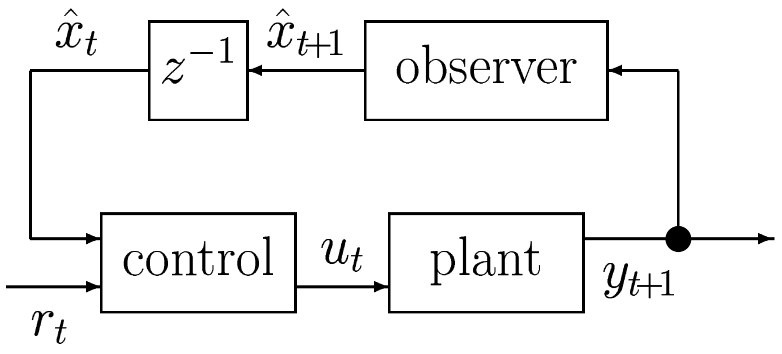


Figure 3.2: The observer estimates the system state from the observable plant outputs and feeds them back to the controller.

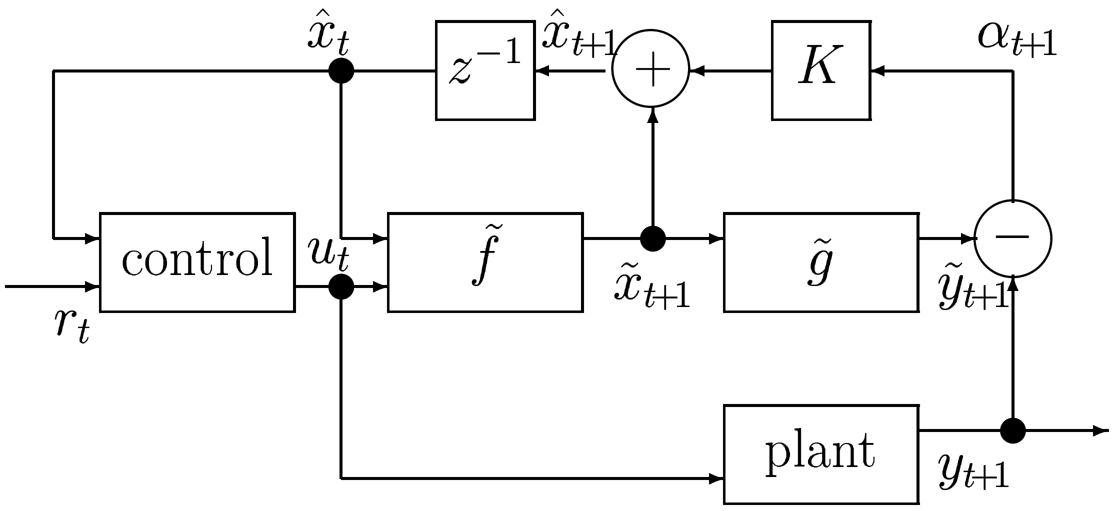


Figure 3.3: An extended Kalman filter can be used to estimate the system state variables for a quasi-linear system.

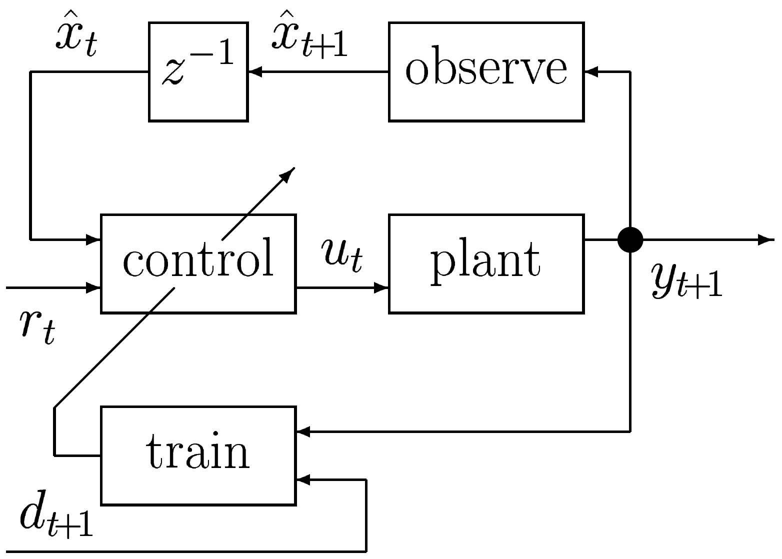


Figure 3.4: The training algorithm adjusts the parameters in the control function so as to reduce the difference between the actual and desired output of the plant.

state variables, $\hat{x}_{t+1} = \tilde{x}_{t+1} + K\alpha_{t+1}$, at the next time step. The accuracy of the estimation process depends upon the accuracy of the mathematical model specified by $\tilde{f}(\cdot)$ and $\tilde{g}(\cdot)$.

The design of observers is beyond the scope of this research, but for the purpose of training the artificial neural controller, it is assumed that it is possible to accurately estimate the system state variables from the system outputs. The block diagram in figure 3.4 shows that the training algorithm adjusts the parameters in the control function so as to reduce the difference between the actual output, y_{t+1}, and the desired output, d_{t+1}, of the plant.

Although the design of adaptive controllers is beyond the scope of this research, the same approach could be used on-line to compensate for minor modeling errors and small changes in plant dynamics. The main difference is that generally a system identification process will be required to automatically main-

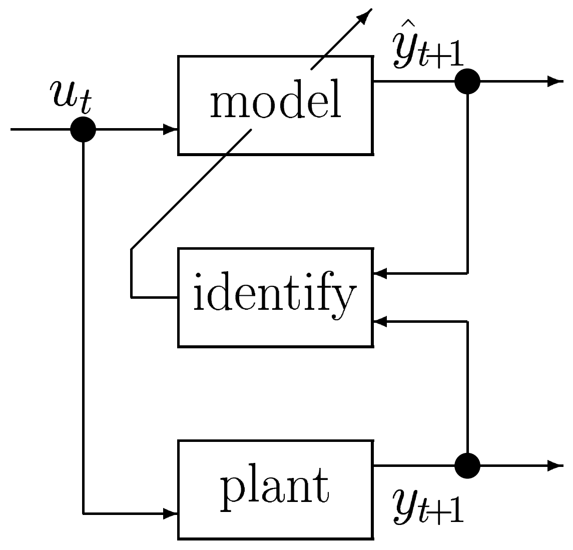


Figure 3.5: The system identification process maintains an accurate model of the system dynamics.

tain a sufficiently accurate model of the plant which can be used to estimate the partial derivatives of the state variables with respect to the state and control variables. The system identification process shown in the block diagram in figure 3.5 uses the difference between the outputs of the model and plant to update the model. The problem is that conventional models are seldom flexible enough to account for unexpected behavior in the actual plant. We have already rejected the idea of substituting an artificial neural network for the plant model. But it is possible to place an artificial neural network in parallel or in tandem with a conventional model which may be able to augment or modify the conventional model enough to compensate for small errors.

Figure 3.6 shows a block diagram of the method used in this research to train an artificial neural network with an off-line simulator to perform the control function, u_t, for a nonlinear system. It is assumed that an accurate mathematical model and observer exist for the system so $\tilde{f}(\cdot) = f(\cdot)$ and $\hat{x}_t = x_t$. Along with the

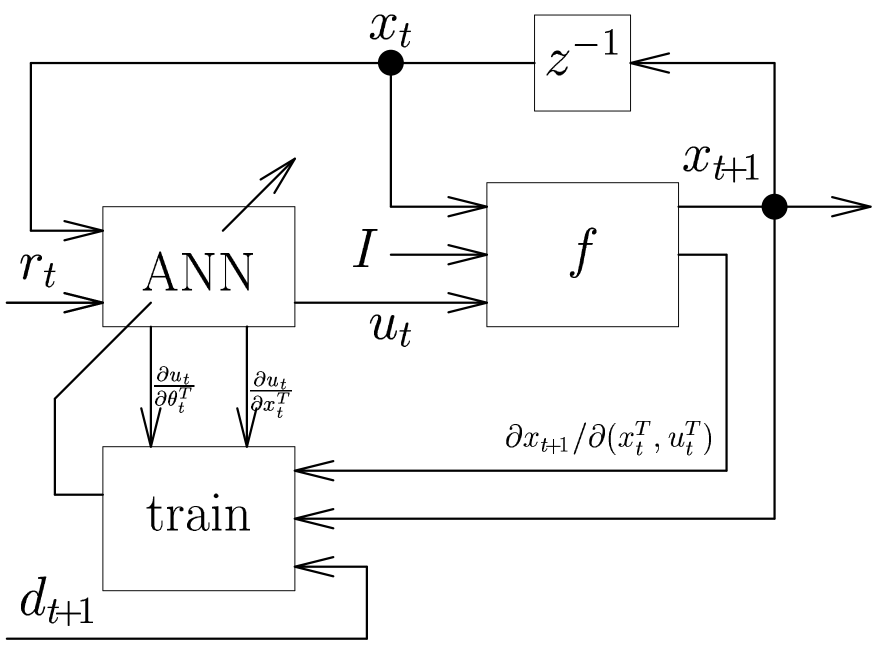

x_t
z^{-1}
x_{t+1}
ANN
r_t
I
f
u_t
$\frac{\partial u_t}{\partial \theta_t^T}$
$\frac{\partial u_t}{\partial x_t^T}$
$\partial x_{t+1}/\partial(x_t^T, u_t^T)$
train
d_{t+1}

Figure 3.6: A simulator is used off-line to train an artificial neural network to perform the control function for a nonlinear system. Both the artificial neural network and the state transition function compute partial derivatives which the real-time recurrent learning algorithm use to adjust the network connection weights and biases at each time step.

normal results, both the artificial neural network and the state transition function compute partial derivatives which the real-time recurrent learning algorithm use together with the difference between the actual and desired state of the system to adjust the network connection weights and biases at each time step.

3.2 Real-Time Recurrent Learning

Real-time recurrent learning[WZ92] (a. k. a. dynamic backpropagation[NP90]) is equivalent to backpropagation-through-time[Wer90] but maintains an estimate of the *total* partial derivatives of the system state with respect to the network biases and connection weights instead of recording the network inputs and outputs at every time step.

The objective of the real-time recurrent learning algorithm is to optimize a performance measure,

$$J = \sum_t J_{t+1}, \tag{3.4}$$

by reducing, at each step t, the loss,

$$J_{t+1} = L(d_{t+1}, x_{t+1}), \tag{3.5}$$

which is a nonlinear function, $L(\cdot)$, of the desired state, d_{t+1}, and the actual state, x_{t+1}, of the plant at time $t+1$. It is often simply a function of the difference between the desired and actual state of the plant as in the weighted square error,

$$L(d_{t+1}, x_{t+1}) = \frac{1}{2}\left(d_{t+1} - x_{t+1}\right)^T W_{t+1}\left(d_{t+1} - x_{t+1}\right), \tag{3.6}$$

where the weight matrix, W_{t+1}, is usually a positive semi-definite symmetric matrix which specifies the relative importance of each component of the error vector, $d_{t+1} - x_{t+1}$. The error correlation matrix $W^{-1} = \left\langle \left(d_{t+1} - x_{t+1}\right)\left(d_{t+1} - x_{t+1}\right)^T \right\rangle$, which might be estimated using the average of the $(d_{t+1} - x_{t+1})\,(d_{t+1} - x_{t+1})^T$, may be used to help calculate the weight matrix.

Since the actual state is determined, in part, by the controller, the performance is also a function of the control function parameters, θ. Real-time recurrent learning is simply a gradient descent algorithm,

$$\theta_{t+1} = \theta_t - \mu \cdot \frac{\bar{\partial} J_{t+1}}{\bar{\partial} \theta_t}, \tag{3.7}$$

where the rate of descent, μ, is a small number. The gradient,

$$\frac{\bar{\partial} J_{t+1}}{\bar{\partial} \theta_t} = \frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_t} \cdot \frac{\partial J_{t+1}}{\partial x_{t+1}}, \tag{3.8}$$

of the performance with respect to the control function parameters is a total derivative composed of the total derivative of the system state with respect to the control function parameters and the partial derivative of the performance with respect to the system state. If the performance function is the weighted square error, then $\partial J_{t+1}/\partial x_{t+1} = -W_{t+1}\,(d_{t+1} - x_{t+1})$.

Substituting the learning rate, $\eta = \mu \cdot (t+1)$, and equation 3.8 into equation 3.7,

$$\theta_{t+1} = \theta_t - \frac{\eta}{t+1} \cdot \frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_t} \cdot \frac{\partial J_{t+1}}{\partial x_{t+1}}, \tag{3.9}$$

shows that it is possible to account for the effect that all previous control function parameter estimation errors have on the performance at the current time step with a minimum of historical data by computing $\bar{\partial} x_t^T / \bar{\partial} \theta_t$ from a recurrence relationship. Since the change in θ will be small at each time step, the total derivative of the system state with respect to the control function parameters is

$$\frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_{t+1}} \cong \frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_t} = \frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial} \theta_t} \cdot \left(\frac{\partial x_{t+1}}{\partial (x_t^T, u_t^T)} \right)^T \tag{3.10}$$

where

$$\frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial} \theta_t} = \left[\frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t}, \frac{\bar{\partial} u_t^T}{\bar{\partial} \theta_t} \right]$$

and

$$\frac{\partial x_{t+1}}{\partial (x_t^T, u_t^T)} = \left[\frac{\partial x_{t+1}}{\partial x_t^T}, \frac{\partial x_{t+1}}{\partial u_t^T}\right].$$

The total derivative of the control function with respect to its parameters is

$$\frac{\bar{\partial} u_t^T}{\bar{\partial} \theta_t} = \frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \frac{\partial u_t^T}{\partial x_t} + \frac{\partial u_t^T}{\partial \theta_t}. \tag{3.11}$$

Expanding the right side of expression 3.10 and substituting equation 3.11,

$$\begin{aligned}
\frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_{t+1}} &\cong \frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \frac{\partial x_{t+1}^T}{\partial x_t} + \frac{\bar{\partial} u_t^T}{\bar{\partial} \theta_t} \cdot \frac{\partial x_{t+1}^T}{\partial u_t} \\
&= \frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \frac{\partial x_{t+1}^T}{\partial x_t} + \left(\frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \frac{\partial u_t^T}{\partial x_t} + \frac{\partial u_t^T}{\partial \theta_t}\right) \cdot \frac{\partial x_{t+1}^T}{\partial u_t} \\
&= \frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \left(\frac{\partial x_{t+1}^T}{\partial x_t} + \frac{\partial u_t^T}{\partial x_t} \cdot \frac{\partial x_{t+1}^T}{\partial u_t}\right) + \frac{\partial u_t^T}{\partial \theta_t} \cdot \frac{\partial x_{t+1}^T}{\partial u_t},
\end{aligned} \tag{3.12}$$

then renaming the state transition matrix, $A = \partial x_{t+1}/\partial x_t^T$, and the control matrix, $B = \partial x_{t+1}/\partial u_t^T$, makes it possible to write

$$\frac{\bar{\partial} x_{t+1}^T}{\bar{\partial} \theta_{t+1}} \cong \frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} \cdot \left(A^T + \frac{\partial u_t^T}{\partial x_t} \cdot B^T\right) + \frac{\partial u_t^T}{\partial \theta_t} \cdot B^T. \tag{3.13}$$

Derivative arithmetic can be used to compute A and B if there are not too many state or control variables. The backward error propagation algorithm can be used to compute $\left(\partial u_t^T/\partial x_t\right) \cdot B^T$ and $\left(\partial u_t^T/\partial \theta_t\right) \cdot B^T$.

3.3 Derivative Arithmetic

Derivative arithmetic computes partial derivatives of program variables with respect to n scalar variables ξ_j where $j \in \{1, 2, \ldots, n\}$. Each scalar variable, x, is replaced by an $n+1$ element vector variable,

$$\begin{aligned}
x^T &= (x_0, x_1, x_2, \ldots, x_n) \\
&= (x_0, \partial x_0/\partial \xi_1, \partial x_0/\partial \xi_2, \ldots, \partial x_0/\partial \xi_n),
\end{aligned} \tag{3.14}$$

which is just the original scalar, x_0, augmented with the n partial derivatives, $\partial x_0/\partial \xi_j$.

Each scalar operation and function is replaced by an operation which computes the partial derivatives along with the normal scalar result. The expression $z = x + y$ is evaluated by performing the normal scalar addition,

$$z_0 = x_0 + y_0,$$

followed by each of the partial derivatives,

$$z_j = \frac{\partial z_0}{\partial \xi_j} = \frac{\partial x_0}{\partial \xi_j} + \frac{\partial y_0}{\partial \xi_j} = x_j + y_j.$$

The expression $z = \exp(x)$ is evaluated by computing the value of the normal scalar function,

$$z_0 = \exp(x_0),$$

followed by each of the partial derivatives,

$$z_j = \frac{\partial z_0}{\partial x_0} \cdot \frac{\partial x_0}{\partial \xi_j} = z_0 x_j.$$

The partial derivatives play no part in comparison operations. The expression $x < y$ is equivalent to $x_0 < y_0$. The ξ_j are also replaced by vectors with $\xi_{j,0} = \xi_j$ and usually initialized with $\xi_{j,k} = \delta_{j,k}$ where $\delta_{j,k}$ is the Kronecker delta function since they are assumed to be independent.

Replacement of the normal computer arithmetic by derivative arithmetic is accomplished using any of the modern computer languages which support user defined types and operator overloading. A Fortran 90[MR90] compiler, the FORTRAN-SC[BRK87] precompiler or the AUGMENT[Cra76] precompiler could be used to perform this function since they are all supersets of the FORTRAN 66 programming language. We found it more convenient to convert the original

FORTRAN 66 source code into C++ source code using the `f2c`[FGM92] translator program. The `f2c` program replaces FORTRAN intrinsic data types `real` and `double precision` with C data types named `real` and `doublereal` which are normally replaced in turn by the built-in C data types `float` and `double` respectively but are replaced by derivative types instead. A C++ class definition for `real` and `doublereal` derivative arithmetic is the only requirement.

3.3.1 The Aircraft Model

At each time step, the partial derivatives of the system state and control variables,

$$\frac{\partial(x_t^T, u_t^T)}{\partial(x_t^T, u_t^T)^T} = I, \tag{3.15}$$

are set to 1 with respect to themselves and 0 with respect to each other. Since many of the zeros will propagate right through the computation to $\partial x_{t+1} / \partial(x_t^T, u_t^T)$, much computational effort is wasted. Still, the simulation time is only about eighteen times longer with derivative arithmetic because the aircraft model has only twelve state variables (see table 3.1) and six control variables (see table 3.2). Flight simulation with derivative arithmetic can still be accomplished in real time on modern microprocessors. But the artificial neural controller may have hundreds or even thousands of biases and connection weights. Another method must be used to compute the partial derivatives of the control vector, $\partial u_t^T / \partial \theta_t$, with respect to the parameter vector.

3.4 The Backprop Algorithm

Backward error propagation (backprop) is a gradient descent algorithm which achieves an advantage in efficiency by avoiding explicit computation of the gradient. Typically, an arbitrary sequence of input and output training pairs are used

	name	variable	description	units
1	P	p	roll rate	rad/sec
2	Q	q	pitch rate	rad/sec
3	R	r	yaw rate	rad/sec
4	V	V	airspeed	ft/sec
5	ALPHA	α	angle of attack	rad
6	BETA	β	sideslip	rad
7	THETA	θ	pitch	rad
8	PSI	ψ	heading	rad
9	PHI	ϕ	bank angle	rad
10	H	h	altitude	ft MSL
11	X	x	distance north	ft
12	Y	y	distance east	ft

Table 3.1: The aircraft model has only twelve state variables.

	name	variable	description	units
1	DA	δ_a	aileron	rad
2	DE	δ_e	elevon	rad
3	DT	δ_t		rad
4	DR	δ_r	rudder	rad
5	THRSTX1	T_{x1}	left throttle	degrees
6	THRSTX2	T_{x2}	right throttle	degrees

Table 3.2: The aircraft model has only six control variables.

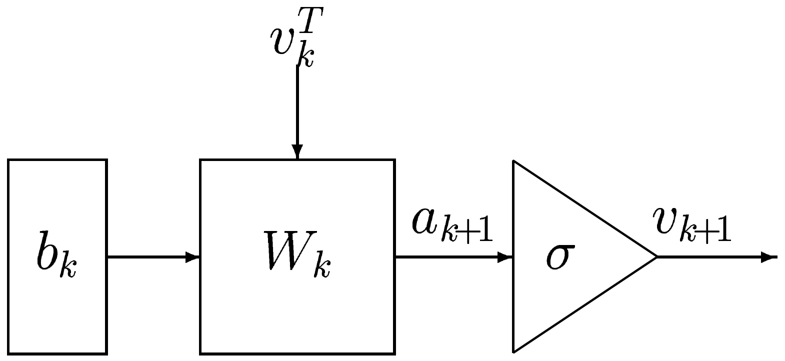


Figure 3.7: Layer k computes $\sigma(b_k + W_k v_k = a_{k+1}) = v_{k+1}$.

to incrementally adjust the network biases and connection weights in a multi-layer feed-forward artificial neural network. The input feeds forward through the layers of the network. The error between the desired and the actual output of the network propagates backward through the network to compute the equivalent error in each layer which is used to update the biases and connection weights. The algorithm can be reduced to just three simple operations:

1. feed forward,
2. back propagate and
3. update.

3.4.1 Feed-Forward

A K-layer feed-forward artificial neural network has K layers of biases and connection weights and $K+1$ layers of nodes including the input layer. The notation $N(n_0, n_1, \ldots, n_K)$ describes a network architecture with n_k nodes in each layer. The input and the output layers have n_0 and n_K nodes respectively. Figure 3.7 show a block diagram of the signal flow through layer k which computes

$$v_{k+1} = \sigma\left(a_{k+1} = [b_k, W_k]\begin{bmatrix}1\\ v_k\end{bmatrix}\right) \tag{3.16}$$

where vector b_k and matrix W_k are the biases and connection weights in layer k. The nonlinear vector function, $\sigma(\cdot)$, is usually a sigmoidal (S-shaped) function such as the hyperbolic tangent, $\tanh(\cdot)$, applied to each element of the activation vector, a_{k+1}, to compute the corresponding element, $v_{k+1,j} = \sigma(a_{k+1,j})$, of the signal vector, v_{k+1}. If, as for controllers, the outputs, v_K, are arbitrary continuous functions of the inputs, v_0, linear output functions replace the $\sigma(a_K)$ or they are simply omitted and the a_K are passed directly to the output. An artificial neural controller must have at least two layers of nodes with input signals

$$v_0 = \begin{bmatrix}x\\ r\end{bmatrix}_t \tag{3.17}$$

and output signals

$$u_t = v_K. \tag{3.18}$$

3.4.2 Backward Error Propagation

At each time step, the output error, Δv_K, is propagated backward through each layer to update the network connection weights and biases in previous layers. The block diagram shown in figure 3.8 shows the feed-forward signal through the network from inputs, v_0, to outputs, v_K, and the feed-back error signal through the network from the output layer, $K-1$, to the input layer, 0. The backprop algorithm computes the equivalent error,

$$\delta_{k-1} = \frac{\partial v_k^T}{\partial a_k} W_k^T \delta_k, \tag{3.19}$$

where

$$\frac{\partial v_k^T}{\partial a_k} = \operatorname{diag}\left\{\frac{\partial v_{k,j}}{\partial a_{k,j}}\right\} = \operatorname{diag}\left\{\sigma'\left(a_{k,j}\right)\right\} = \operatorname{diag}\left\{v'_{k,j}\right\}$$

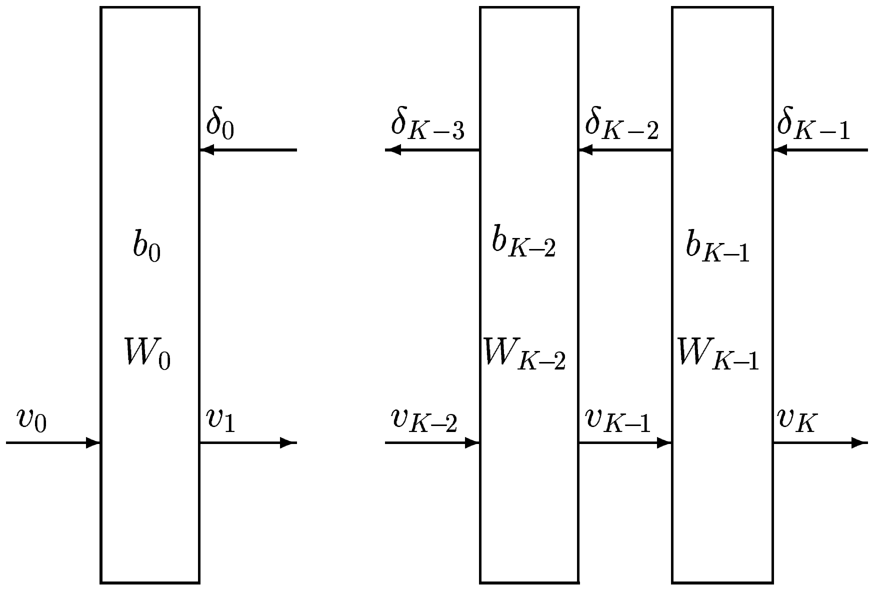


Figure 3.8: The backward error propagation algorithm is used to adjust the biases and connection weights in multi-layer feed-forward artificial neural networks. The signals, v_k, feed forward through K layers. The output of each hidden layer is an input for the next layer. The equivalent errors, δ_k, feed backward through the layers and are used to update the biases, b_k, and the connection weights, W_k.

is a diagonal matrix. The initial equivalent error,

$$\delta_{K-1} = \frac{\partial v_K^T}{\partial a_K} \Delta v_K, \tag{3.20}$$

is computed using the error at the output layer.

3.4.3 Update Weights and Biases

All that remains is to update the network biases,

$$b_k \leftarrow b_k + \eta \delta_k, \tag{3.21}$$

and connection weights,

$$W_k \leftarrow W_k + \eta \delta_k v_k^T, \tag{3.22}$$

where the learning rate, η, is a small number.

3.4.4 Partial Derivatives

A similar method can be used in the recurrent learning algorithm to back propagate B^T and avoid explicit computation of $\partial u_t^T / \partial x_t$ and $\partial u_t^T / \partial \theta_t$. First, we define long row vectors,

$$\vartheta_k^T = \text{row}\left[b_k, W_k\right], \tag{3.23}$$

by concatenating the rows of the $[b_k, W_k]$ matrix together. Then the transpose of the parameter vector,

$$\theta_t^T = [\vartheta_0^T, \vartheta_1^T, \ldots, \vartheta_{K-1}^T], \tag{3.24}$$

is constructed by concatenating the ϑ_k^T together. This construction permits us to partition

$$\frac{\bar{\partial} x_t^T}{\bar{\partial} \theta_t} = \begin{bmatrix} \frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_0} \\ \frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_1} \\ \vdots \\ \frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_{K-1}} \end{bmatrix}_t \tag{3.25}$$

and

$$\frac{\partial u_t^T}{\partial \theta_t} = \begin{bmatrix} \frac{\partial v_K^T}{\partial \vartheta_0} \\ \frac{\partial v_K^T}{\partial \vartheta_1} \\ \vdots \\ \frac{\partial v_K^T}{\partial \vartheta_{K-1}} \end{bmatrix}_t \tag{3.26}$$

so that it is possible to write independent expressions,

$$\left.\frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_k}\right|_{t+1} \cong \left.\frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_k}\right|_t \cdot \left(A^T + \left.\frac{\partial v_K^T}{\partial x}\right|_t \cdot B^T \right) + \left.\frac{\partial v_K^T}{\partial \vartheta_k}\right|_t \cdot B^T, \tag{3.27}$$

for each of the parts of expression 3.13. Now we can apply the chain rule to

$$\begin{aligned} \frac{\partial v_K^T}{\partial v_k} \cdot B^T &= \frac{\partial v_{k+1}^T}{\partial v_k} \cdot \frac{\partial v_K^T}{\partial v_{k+1}} \cdot B^T = \frac{\partial a_{k+1}^T}{\partial v_k} \cdot \frac{\partial v_{k+1}^T}{\partial a_{k+1}} \cdot \frac{\partial v_K^T}{\partial v_{k+1}} \cdot B^T \\ &= W_k^T \cdot \mathrm{diag}\left\{ v'_{k+1,j} \right\} \cdot \frac{\partial v_K^T}{\partial v_{k+1}} \cdot B^T = W_k^T \cdot {\Delta_k}^T \end{aligned} \tag{3.28}$$

where

$${\Delta_{k-1}}^T = \mathrm{diag}\left\{ v'_{k,j} \right\} \cdot \frac{\partial v_K^T}{\partial v_k} \cdot B^T = \mathrm{diag}\left\{ v'_{k,j} \right\} \cdot W_k^T \cdot {\Delta_k}^T \tag{3.29}$$

and

$${\Delta_{K-1}}^T = \mathrm{diag}\left\{ v'_{K,j} \right\} \cdot \frac{\partial v_K^T}{\partial v_K} \cdot B^T = \mathrm{diag}\left\{ v'_{K,j} \right\} \cdot B^T. \tag{3.30}$$

The equivalent control matrix, Δ_k, is analogous to the equivalent error, δ_k, in the original version of the algorithm. Substituting $k = 0$ into equation 3.28 results in

$$\frac{\partial v_K^T}{\partial x} \cdot B^T = X^T \cdot \Delta_0^T \tag{3.31}$$

where the X matrix is obtained by partitioning the $W_0 = [X, R]$ connection weight matrix so that $W_0 v_0 = Xx + Rr$. The last term on the right hand side of expression 3.27,

$$\begin{aligned} \frac{\partial v_K^T}{\partial \vartheta_k} \cdot B^T &= \frac{\partial v_{k+1}^T}{\partial \vartheta_k} \cdot \frac{\partial v_K^T}{\partial v_{k+1}} \cdot B^T = \frac{\partial a_{k+1}^T}{\partial \vartheta_k} \cdot \frac{\partial v_{k+1}^T}{\partial a_{k+1}} \cdot \frac{\partial v_K^T}{\partial v_{k+1}} \cdot B^T \\ &= I \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix} \cdot \Delta_k^T = \left(\Delta_k \cdot I \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix}^T \right)^T = \Delta_k^T \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix} \end{aligned} \tag{3.32}$$

where the $\otimes$ operator denotes the Kronecker product[Gra81]. Substituting equation 3.31 and equation 3.32 into expression 3.27 we get

$$\left.\frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_k}\right|_{t+1} \cong \left.\frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_k}\right|_t \cdot \left(A^T + X^T \cdot \Delta_0^T\right) + \Delta_k^T \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix}. \tag{3.33}$$

Defining

$$G_k|_{t+1} = \frac{1}{t+1} \cdot \left.\frac{\bar{\partial} x^T}{\bar{\partial} \vartheta_k}\right|_{t+1} \tag{3.34}$$

permits us to write two new update rules,

$$G_k \leftarrow (1-\epsilon) \cdot G_k \cdot (A + \Delta_0 \cdot X)^T + \epsilon \cdot \Delta_k^T \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix} \tag{3.35}$$

where $\epsilon = 1/(t+1)$ and

$$\vartheta_k \leftarrow \vartheta_k + G_k \cdot e \tag{3.36}$$

where $e = -\eta \cdot \partial J_{t+1}/\partial x_{t+1}$.

Update rule 3.35 will become unstable as $\epsilon \to 0$ if the magnitude of any of the eigenvalues, λ_j, of the augmented state transition matrix, $A + \Delta_0 \cdot X$, are consistently larger than 1. This is generally true for systems like flight simulators which have state variables such as x and y which can increase without bound. It can be stabilized by restricting $\epsilon > 1/(1+\tau)$ where the decaying time $\tau \leq 1/\left(\max\{|\lambda_j|\} - 1\right)$. This causes the real-time recurrent learning algorithm to respond more strongly to recent experiences and "forget" the effect of experiences that occurred in the more distant past.

First order gradient descent algorithms like backward error propagation and real-time recurrent learning are practically useless unless all of the inputs are independent of each other and have about the same variance. But the inputs to systems like the artificial neural controller are generally highly correlated with each other and some vary several orders of magnitude more than others. The weights associated with large inputs usually need to be small while the weights associated with small inputs must be large. But since, at each step in the learning algorithm, the change in the weight is proportional to the input, the weights associated with the small inputs change slowly relative to the weights associated with the large inputs. The result is that the learning algorithm will be unstable for the weights associated with large inputs unless the learning rate, η, is so small that learning virtually stops for the weights associated with the small inputs. The ideal way to correct this problem is to use a second or higher order gradient descent algorithm. Unfortunately, these algorithms require more computer power than is readily available to us at this writing. We will instead employ a more simple but effective method which involves "normalizing" the inputs.

3.5 Second Gradients

The purpose of this section is to characterize the nature of instability in gradient descent algorithms then propose a general solution to the problem and use it to justify the practice of normalizing the inputs. Finally, update rules are provided which incorporate input normalization into the biases and connection weights.

Consider using a first order gradient descent algorithm to minimize a performance function,

$$J(\theta) = \iota + \kappa\theta + \frac{1}{2}\lambda\theta^2, \tag{3.37}$$

which is a second order polynomial in the adjustable parameter θ. At step $n+1$, the adjustable parameter

$$\begin{aligned} \theta_{n+1} &= \theta_n - \mu\frac{\partial J}{\partial \theta_n} = \theta_n - \mu(\kappa + \lambda\theta_n) = (1-\mu\lambda)\theta_n - \mu\kappa \\ &= (1-\mu\lambda)^n\theta_0 - \sum_{k=1}^{n}(1-\mu\lambda)^{n-k}\mu\kappa \end{aligned} \tag{3.38}$$

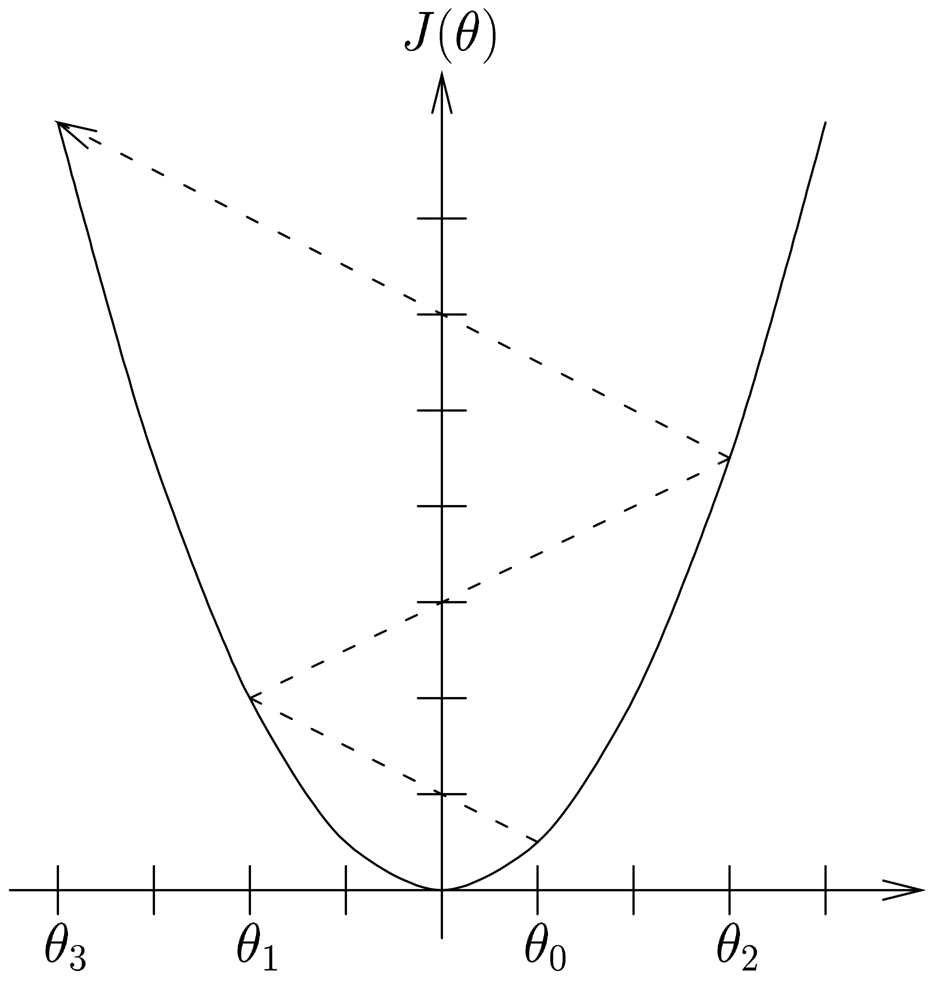


Figure 3.9: Gradient descent is unstable for $J(\theta) = \frac{1}{2}\theta^2$ when $\mu = 3$.

where

$$\frac{\partial J}{\partial \theta_n} = \left.\frac{\partial J(\theta)}{\partial \theta}\right|_{\theta_n}.$$

As shown in figure 3.9, the algorithm will be unstable and the adjustable parameter will grow exponentially unless $-1 < 1 - \mu\lambda < +1$ which means that we must restrict the rate of descent $0 < \mu < 2/|\lambda|$. Using the absolute value of the curvature, $\lambda/2$, ensures that the rate of descent will be positive and the algorithm will actually step down the gradient even if the performance curve is convex instead of concave.

A second order gradient descent method can be derived at step n from a second order approximation of the performance function

$$J(\theta_{n+1}) = J(\theta_n) + \frac{\partial J}{\partial \theta_n}\Delta\theta_n + \frac{1}{2}\frac{\partial^2 J}{\partial \theta_n^2}\Delta\theta_n^2 \tag{3.39}$$

where $\Delta\theta_n = \theta_{n+1} - \theta_n$ and $\Delta\theta_n^2 = (\theta_{n+1} - \theta_n)^2$. Setting the partial derivative of this approximation equal to zero with respect to the change in the adjustable parameter,

$$\frac{\partial J(\theta_{n+1})}{\partial \Delta\theta_n} = \frac{\partial J}{\partial \theta_n} + \frac{\partial^2 J}{\partial \theta_n^2}\Delta\theta_n = 0, \tag{3.40}$$

permits us to solve for the adjustable parameter,

$$\theta_{n+1} = \theta_n - \frac{\partial J/\partial\theta_n}{|\partial^2 J/\partial\theta_n^2|}, \tag{3.41}$$

which minimizes it. Apparently, the best choice for the rate of descent

$$\mu = \left|\frac{\partial^2 J}{\partial \theta_n^2}\right|^{-1} = \frac{1}{|\lambda_n|}. \tag{3.42}$$

If the performance function is the second order polynomial in equation 3.37, then the second order approximation in equation 3.39 is exact and $\mu = 1/|\lambda|$ specifies the rate of descent required to minimize the performance function in one step. But performance functions are not restricted to second order polynomials and may not even be convex everywhere in parameter space. More generally, they will contain a number of local minima separated by local maxima with inflection points between them as shown in figure 3.10. In this case, it is prudent to use a more conservative rate of descent

$$\mu = \frac{\eta}{|\lambda_n|} \tag{3.43}$$

where $0 < \eta < 1$ and limit the rate of descent

$$\mu = \frac{\eta}{\max\{|\lambda_n|, |\lambda_{\min}|\}} \tag{3.44}$$

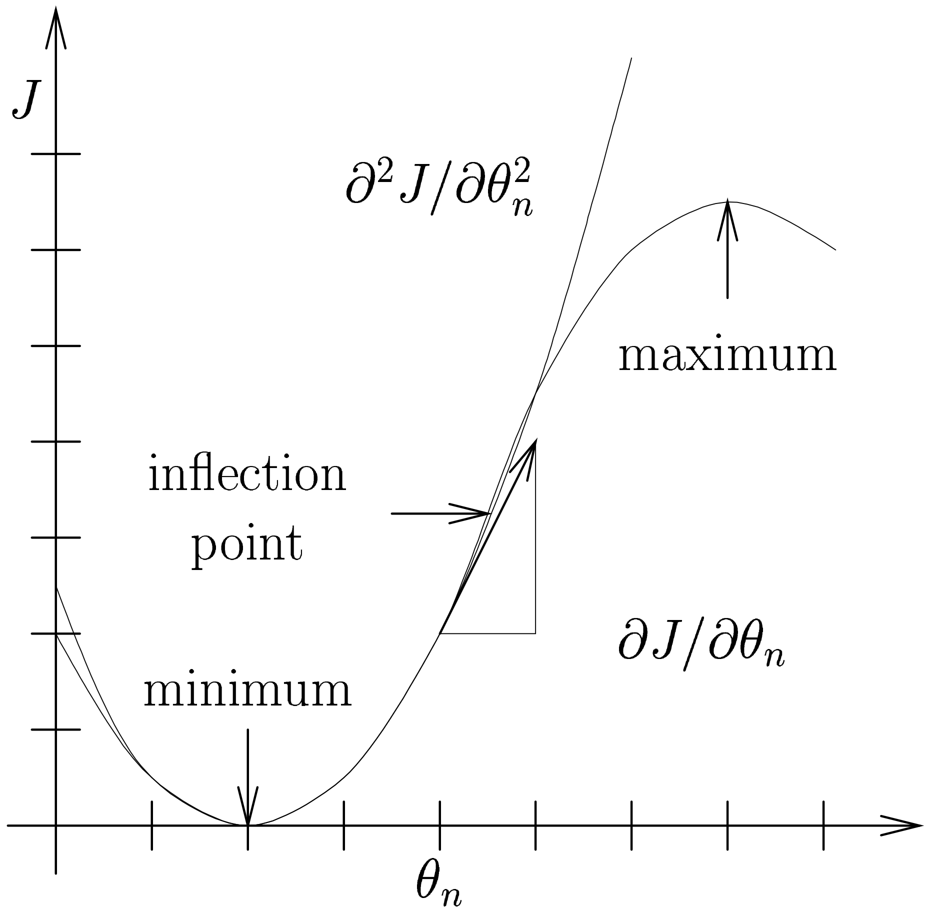

J
$\partial^2 J/\partial\theta_n^2$
maximum
inflection point
$\partial J/\partial\theta_n$
minimum
θ_n

Figure 3.10: The performance function, $J(\theta)$, will generally contain a number of minima separated by maxima with inflection points between them.

where $0 < |\lambda_{\min}|$ since the curvature of the performance function goes to zero when the adjustable parameter approaches an inflection point. For systems with a single adjustable parameter, it is usually sufficient to choose a constant rate of descent $\mu < 1/|\lambda_{\max}|$ where $\frac{1}{2}\lambda_{\max}$ is the maximum curvature of the performance curve. This value is usually easily obtained by trial and error because instability in the learning algorithm is so obviously apparent if the rate of descent is too large. Systems with several adjustable parameters pose additional problems.

If the system has several adjustable parameters, a second order gradient descent method can be derived at step n from a second order approximation of the performance function,

$$J(\theta_{n+1}) = J(\theta_n) + \Delta\theta_n^T \frac{\partial J}{\partial \theta_n} + \frac{1}{2}\Delta\theta_n^T \frac{\partial^2 J}{\partial \theta_n^2}\Delta\theta_n, \tag{3.45}$$

which is similar to the one in equation 3.39 except that θ represents a parameter vector and the second derivative,

$$\frac{\partial^2 J}{\partial \theta_n^2} = \frac{\partial}{\partial \theta}\frac{\partial J(\theta)}{\partial \theta^T}\bigg|_{\theta_n} = H_n,$$

is the Hessian matrix. Setting the partial derivatives of this approximation equal to zero with respect to the change in each of the adjustable parameters,

$$\frac{\partial J(\theta_{n+1})}{\partial \Delta\theta_n} = \frac{\partial J}{\partial \theta_n} + \frac{\partial^2 J}{\partial \theta_n^2}\Delta\theta_n = 0 \tag{3.46}$$

permits us to solve for the adjustable parameters,

$$\theta_{n+1} = \theta_n - \left[\frac{\partial^2 J}{\partial \theta_n^2}\right]^{-1} \frac{\partial J}{\partial \theta_n} \tag{3.47}$$

which minimize the performance function if it is a concave, second order polynomial. More generally, the Hessian matrix, H_n, will become singular near inflection points, negative definite at local maxima and have at least one negative eigenvalue near saddle points. Since the Hessian is a real symmetric matrix, it can be decomposed

$$H_n = Q_n^T \Lambda_n Q_n \tag{3.48}$$

into an orthogonal matrix, $Q_n = [e_{n,j}]$, composed of the unit eigenvectors, $e_{n,j}$, of H_n and a diagonal matrix, $\Lambda_n = \operatorname{diag}\{\lambda_{n,j}\}$, composed of the corresponding eigenvalues, $\lambda_{n,j}$, of H_n. This permits us to replace the solution in equation 3.47 with

$$\theta_{n+1} = \theta_n - \eta Q_n^T D_n^{-1} Q_n \frac{\partial J}{\partial \theta_n} \tag{3.49}$$

where

$$D_n = \operatorname{diag}\{\max\{|\lambda_{n,j}|, |\lambda_{\min}|\}\}. \tag{3.50}$$

Computing and decomposing the Hessian at every step in the algorithm can be very expensive especially for performance functions like

$$J(\theta) = \int_\Omega \mathcal{J}(x;\theta) d\Omega \tag{3.51}$$

which are integrals of a nonlinear density function, $\mathcal{J}(\cdot)$, of the state, x, over an extended subspace, Ω, of state space. It is often impossible to evaluate the continuous integral so one must settle for a sum of samples,

$$J(\theta) = \sum_k \mathcal{J}(x_k;\theta), \tag{3.52}$$

at discrete points, x_k, in state space which represent the subspace Ω. Since the partial sums,

$$J_n(\theta) = \sum_{k=1}^{n} \mathcal{J}(x_k;\theta), \tag{3.53}$$

already contain some of the information needed to estimate the adjustable parameters, some expense can be saved by applying the gradient descent algorithm to the partial sums while accumulating the total.

Suppose we have already found the adjustable parameters, θ_n, that minimize the partial sum, $J_n(\theta_n)$, at step n so that

$$\frac{\partial J_n(\theta_n)}{\partial \theta_n} = 0.$$

Then the partial sum at the next step,

$$J_{n+1}(\theta_n) = J_n(\theta_n) + \mathcal{J}(x_{n+1}; \theta_n), \tag{3.54}$$

can be used to approximate

$$J_{n+1}(\theta_{n+1}) = J_{n+1}(\theta_n) + \Delta\theta_n^T \frac{\partial J_{n+1}(\theta_n)}{\partial \theta_n} + \frac{1}{2}\Delta\theta_n^T \frac{\partial^2 J_{n+1}(\theta_n)}{\partial \theta_n^2}\Delta\theta_n \tag{3.55}$$

where the gradient

$$\frac{\partial J_{n+1}(\theta_n)}{\partial \theta_n} = \frac{\partial J_n(\theta_n)}{\partial \theta_n} + \frac{\partial \mathcal{J}(x_{n+1}; \theta_n)}{\partial \theta_n} = \frac{\partial \mathcal{J}(x_{n+1}; \theta_n)}{\partial \theta_n} \tag{3.56}$$

and the Hessian

$$H_n = \frac{\partial^2 J_{n+1}(\theta_n)}{\partial \theta_n^2} = \sum_{k=0}^{n} \frac{\partial^2 \mathcal{J}(x_{k+1}; \theta_n)}{\partial \theta_n^2} \approx \sum_{k=0}^{n} \frac{\partial^2 \mathcal{J}(x_{k+1}; \theta_k)}{\partial \theta_k^2} \tag{3.57}$$

if the adjustable parameters don't change very much and approximation 3.55 is accurate. As this partial sum accumulates, the elements of the Hessian can become quite large. It may instead be prudent to maintain a running average,

$$\frac{H_n}{n+1} \approx (1-\epsilon)\cdot\frac{H_{n-1}}{n} + \epsilon \cdot \frac{\partial^2 \mathcal{J}(x_{n+1}; \theta_n)}{\partial \theta_n^2} \tag{3.58}$$

where $\epsilon = 1/(n+1)$ if there is danger of floating point overflow. Since this running average is a real symmetric matrix, it can be decomposed,

$$\frac{H_n}{n+1} = Q_n^T \Lambda_n Q_n, \tag{3.59}$$

into an orthonormal matrix, $Q_n = P_n Q_{n-1}$, composed of the unit eigenvectors and a diagonal matrix, Λ_n, composed of the corresponding eigenvalues. Substituting equation 3.59 into equation 3.58 and solving for

$$\Lambda_n = P_n((1-\epsilon)\cdot\Lambda_{n-1} + \epsilon \cdot Q_{n-1}\frac{\partial^2 \mathcal{J}(x_{n+1}; \theta_n)}{\partial \theta_n^2}Q_{n-1}^T)P_n^T \tag{3.60}$$

provides us with insight into how we might compute this decomposition efficiently using the previous decomposition. If ϵ is small, then the new information only slightly perturbs Λ_{n-1}. A single sweep of Jacobi rotations, P_n, should be sufficient to almost zero the off-diagonal elements and reduce Λ_n nearly to diagonal form. Precision is not essential. A rough estimate of the eigenvalues and eigenvectors should be sufficient to implement the solution proposed in equation 3.49 and equation 3.50.

3.6 Normalization

Unfortunately, we are in no position to take advantage of second order learning algorithms for the same reason that second and higher order learning algorithms are generally avoided. The required computational resources are not readily available to us. Computational and storage requirements are proportional to the number of parameters for first order gradient descent methods but proportional to the *square* of the number of parameters for second order gradient descent methods. We can expect second order methods to become more popular as faster computers with large memories become more available and affordable. But for the time being, we propose to contrive a first order method which, in effect, substitutes a constant for the dynamic Hessian matrix, H_n, in a second order method. The effect is achieved by "normalizing" the inputs to a K-layer, feedforward artificial neural network.

The objective is to minimize

$$J = \frac{1}{2} \sum_t (d - v_K)_t^T \, W_t \, (d - v_K)_t \tag{3.61}$$

where d is the desired output, v_K is the actual output and W is a symmetric, positive, semi-definite matrix which specifies the relative importance of each of the outputs. With respect to the parameters ϑ_l in layer l, the gradient

$$\frac{\partial J}{\partial \vartheta_l} = -\sum_t \left.\frac{\partial v_K^T}{\partial \vartheta_l}\right|_t W_t \left(d - v_K\right)_t \tag{3.62}$$

where

$$\frac{\partial v_K^T}{\partial \vartheta_l} = \frac{\partial a_{l+1}^T}{\partial \vartheta_l} \cdot \frac{\partial v_K^T}{\partial a_{l+1}} = I_{n_{l+1}} \otimes \begin{bmatrix} 1 \\ v_l \end{bmatrix} \cdot \frac{\partial v_K^T}{\partial a_{l+1}} = \frac{\partial v_K^T}{\partial a_{l+1}} \otimes \begin{bmatrix} 1 \\ v_l \end{bmatrix}. \tag{3.63}$$

With respect to the parameters ϑ_k and ϑ_l in layers k and l, the second gradient

$$\frac{\partial^2 J}{\partial \vartheta_k \partial \vartheta_l^T} = \frac{\partial}{\partial \vartheta_k}\frac{\partial J}{\partial \vartheta_l^T} = \sum_t \left.\frac{\partial v_K^T}{\partial \vartheta_k}\right|_t W_t \left.\frac{\partial v_K}{\partial \vartheta_l^T}\right|_t - \sum_t \left(d - v_K\right)_t^T W_t \left.\frac{\partial^2 v_K}{\partial \vartheta_k \partial \vartheta_l^T}\right|_t. \tag{3.64}$$

Assuming that the second sum in equation 3.64 is negligible and substituting from equation 3.63 permits us to write

$$\begin{aligned} \frac{\partial^2 J}{\partial \vartheta_k \partial \vartheta_l^T} &\approx \sum_t \left.\frac{\partial v_K^T}{\partial a_{k+1}}\right|_t \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix}_t W_t \left.\frac{\partial v_K}{\partial a_{l+1}^T}\right|_t \otimes \begin{bmatrix} 1 \\ v_l \end{bmatrix}_t^T \\ &= \sum_t \left.\frac{\partial v_K^T}{\partial a_{k+1}}\right|_t W_t \left.\frac{\partial v_K}{\partial a_{l+1}^T}\right|_t \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix}_t \cdot \begin{bmatrix} 1 \\ v_l \end{bmatrix}_t^T. \end{aligned} \tag{3.65}$$

Assuming that we have no a priori information about the partial derivatives, $\partial v_K^T / \partial a_{k+1}$, of the actual outputs with respect to the activations, a_{k+1}, we should expect them to be independent of each other and $\frac{\partial v_K^T}{\partial a_{k+1}} W \frac{\partial v_K}{\partial a_{l+1}^T} = 0$ for all $l \neq k$. Furthermore, we expect the partial derivatives, $\partial v_K^T / \partial a_{k+1,i}$, of the actual outputs with respect to each element, $a_{k+1,i}$, of layer $k+1$ to be independent and equal because of the symmetry of the nodes in that layer. We also expect that the inputs, $v_{k,j}$, are all independent so that the only elements of the second derivative that accumulate any significant amount are along the diagonal,

$$\frac{\partial^2 J}{\partial \vartheta_k^2} = \frac{\partial^2 J}{\partial \vartheta_k \partial \vartheta_k^T} \propto I_{n_{k+1}} \otimes \operatorname{diag}\{1, \langle v_{k,j}^2 \rangle\}. \tag{3.66}$$

Since we are using the hyperbolic tangent, $\tanh(\cdot)$, we expect $\langle v_{k,j}^2 \rangle \approx 1$ for all $k > 0$. Only the variances, $\langle v_{0,j}^2 \rangle$, associated with the inputs differ significantly from 1. Apparently, the entire second derivative would be proportional to a huge identity matrix and inversion would be unnecessary if all the inputs were normalized, $v_{0,j} \leftarrow v_{0,j}/\sqrt{\langle v_{0,j}^2 \rangle}$, before they are used in the learning algorithm.

Before normalizing the inputs, it is important to try to remove some of the correlation between the inputs especially correlation with the constant input which appears as a constant bias in the variable inputs. This is usually accomplished by subtracting an appropriate linear combination of the correlated inputs. All of this preprocessing can be accomplished by prepending another layer to our network with uncorrelated and normalized outputs, $v_0 = b_{-1} + W_{-1} v_{-1}$, where v_{-1} are the raw inputs. For example, if

$$v_0 = \mathrm{diag}\left\{ \frac{1}{\sqrt{\langle v_{-1,j}^2 \rangle - \langle v_{-1,j} \rangle^2}} \right\} (v_{-1} - \langle v_{-1} \rangle)$$

then

$$b_{-1} = -\mathrm{diag}\left\{ \frac{1}{\sqrt{\langle v_{-1,j}^2 \rangle - \langle v_{-1,j} \rangle^2}} \right\} \langle v_{-1} \rangle$$

and

$$W_{-1} = \mathrm{diag}\left\{ \frac{1}{\sqrt{\langle v_{-1,j}^2 \rangle - \langle v_{-1,j} \rangle^2}} \right\} .$$

After training, we update the network input biases, $b_0 \leftarrow b_0 + W_0 b_{-1}$, and connection weights, $W_0 \leftarrow W_0 W_{-1}$, to incorporate the preprocessing step so that the network can accept the raw inputs directly.

3.7 Summary

A summary of the real-time recurrent learning algoritm for artificial neural controllers is shown in table 3.3. The algorithm is designed to accept an initial set of input biases and connection weights that incorporate the preprocessing step and temporarily transform them for the learning algorithm.

```
Pseudocode                                              Remarks

$W_0 \leftarrow W_0 W_{-1}^{-1}$                          Dis-incorporate
$b_0 \leftarrow b_0 - W_0 b_{-1}$                         Normalization
for $t \leftarrow 0$ to $\ldots$ do begin
    $v_0 = b_{-1} + W_{-1} \begin{bmatrix} x \\ r \end{bmatrix}$        Normalize Inputs
    for $k \leftarrow 0$ to $K-2$ do                      Feed Forward
        $v_{k+1} \leftarrow \tanh(b_k + W_k v_k)$
    $u \leftarrow b_{K-1} + W_{K-1} v_{K-1}$
    $\left[\, x \mid A \mid B \,\right] \leftarrow f\left(\left[\begin{array}{c|c} x & I \\ u & \end{array}\right]\right)$        Derivative Arithmetic
    $\Delta_{K-1} \leftarrow B$                           Back Propagate
    for $k \leftarrow K-1$ down to 1 do
        $\Delta_{k-1} \leftarrow \Delta_k W_k \mathrm{diag}\{1 - v_{k,j}^2\}$
    $\epsilon \leftarrow \frac{1}{t+1}$                   Update
    $e \leftarrow \eta \cdot W(d - x)$
    for $k \leftarrow 0$ to $K-1$ do begin
        $G_k \leftarrow (1-\epsilon) \cdot G_k \cdot (A + \Delta_0 \cdot X)^T + \epsilon \cdot \Delta_k^T \otimes \begin{bmatrix} 1 \\ v_k \end{bmatrix}$
        $\vartheta_k \leftarrow \vartheta_k + G_k \cdot e$
        end
    end
$b_0 \leftarrow b_0 + W_0 b_{-1}$                         Re-incorporate
$W_0 \leftarrow W_0 W_{-1}$                               Normalization
```

Table 3.3: Summary of the real-time recurrent learning algorithm.

CHAPTER 4

Preliminary Investigations

We were able to construct an artificial neural controller by manipulating the gains employed in a simple conventional controller which was provided along with the flight simulator. This was important for two reasons. First, it demonstrated that an artificial neural network can control the aircraft. Second, it showed that a feed-forward network architecture was appropriate for the neural controller.

Figure 4.1 shows a block diagram of the conventional controller. The purpose of the controller is to maintain a set altitude HSET, airspeed VSET and bank angle PHISET without sideslip. These reference signals are supplied as inputs to the controller along with six of the twelve state variables and two state variable derivatives, the vertical airspeed $\dot{h}$ and the acceleration $\dot{V}$, which are nonlinear functions of the state and control variables. A total of five control signals are output by four independent subunits. Two of the subunits produce linear functions of their inputs. The DR signal controls the rudder deflection which is used to minimize sideslip by turning the aircraft into the relative wind. The DA and DT signals control the aileron and differential elevon deflections respectively. Together, these deflections are used to bank and turn the aircraft. The other two subunits produce nonlinear functions of their inputs and internal state. The DE signal controls the deflection of the elevators (elevons) and the angle of attack. The THRSTX signal controls the throttle for both left and right engines. Together, these deflections determine the acceleration $\dot{V}$, and the rate of climb

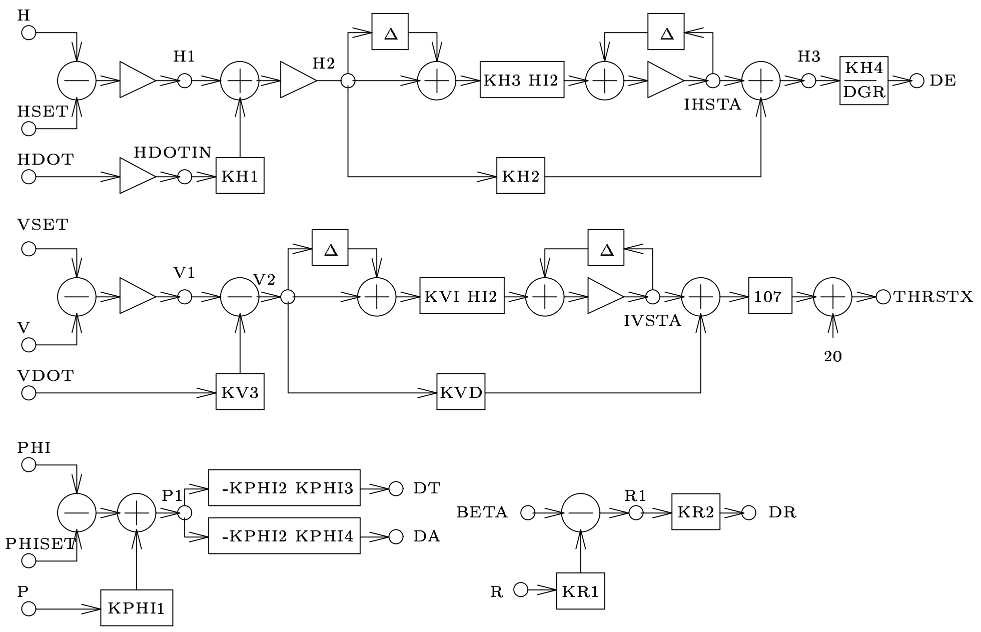


Figure 4.1: A simple conventional controller was provided along with the flight simulator. Variable names from the computer simulation were used to label the signals and gains. The symbols represent nonlinear limiting functions. The Δ symbols represent unit time delays.

$\dot{h}$.

The conventional controller was transformed into a feed-forward network in three steps. First, the nonlinear limit functions were replaced by equivalent normalized semi-linear functions

$$\sigma(x) = \begin{cases} -1, & x \leq -1 \\ x, & -1 < x < +1 \\ +1, & +1 \leq x \end{cases} \tag{4.1}$$

by scaling their respective input and output gains,

$$\begin{aligned}
\text{KH1} &\leftarrow \frac{\text{KHDOT}}{\text{KH}}\text{KH1} \\
\text{KH2} &\leftarrow \frac{\text{KH}}{20}\text{KH2} \\
\text{KH3} &\leftarrow \frac{\text{KH}}{20}\text{KH3} \\
\text{KH4} &\leftarrow \frac{20}{\text{DGR}}\text{KH4} \\
\text{KVD} &\leftarrow 2 \cdot \text{KV} \cdot \text{KVD} \\
\text{KVI} &\leftarrow 2 \cdot \text{KV} \cdot \text{KVI} \\
\text{KV3} &\leftarrow \frac{\text{KV3}}{\text{KV}} \\
\text{KV4} &\leftarrow \frac{107}{2},
\end{aligned}$$

and providing appropriate biasing,

$$\begin{aligned}
\text{IHSTA0} &= 1/4 \\
\text{IVSTA0} &= \text{-1},
\end{aligned}$$

as shown in the block diagram in figure 4.2. Second, the internal state feed-back loops were replaced by equivalent loops which feed the delayed DE and THRSTX outputs back into their respective subunits as shown in the block diagram in figure 4.3. Third, the conventional controller is replaced by a 4-layer, feed-forward, artificial neural network, $N(28, 12, 8, 6, 6)$, using the semi-linear sigmoid in equation 4.1. Delayed controller inputs and outputs are supplied to the network along

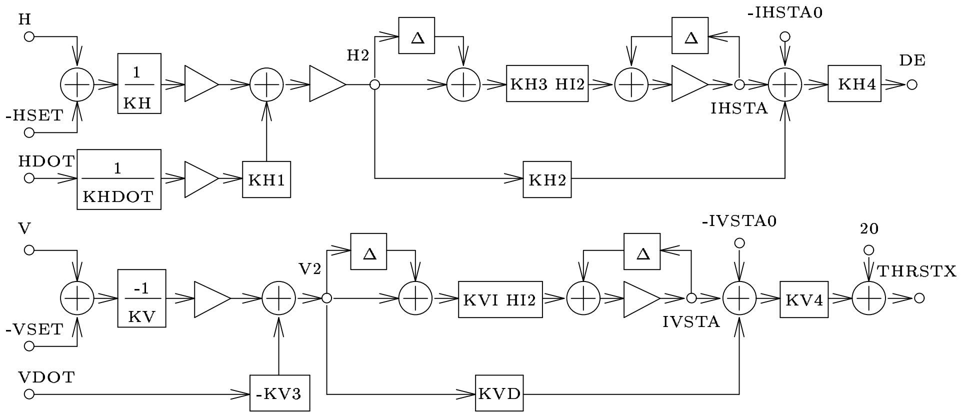


Figure 4.2: The first step in transforming the conventional controller into a neural controller replaces each of the nonlinear limit functions with equivalent normalized semi-linear functions (symbols) then scales the signal gains and adjusts the biases appropriately.

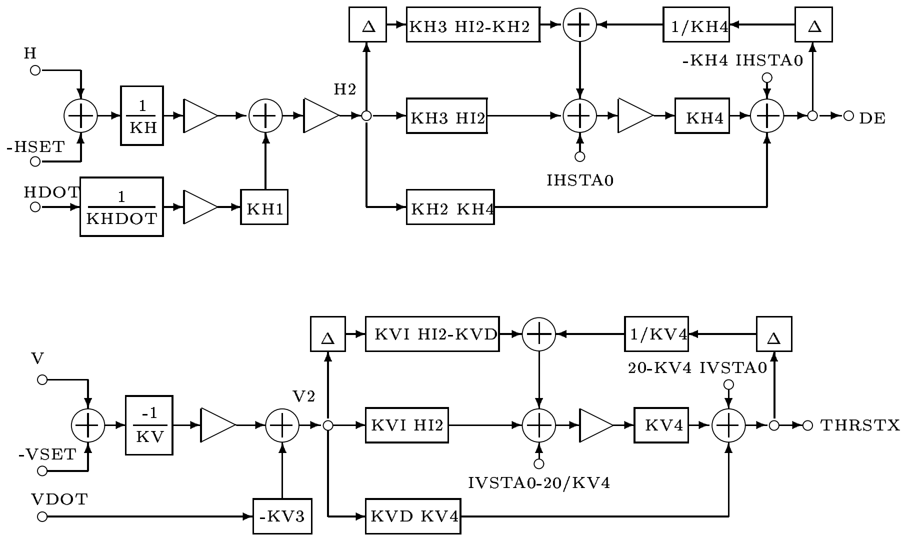


Figure 4.3: The second step in transforming the conventional controller into a neural controller replaces each of the internal state feed-back loops with equivalent loops which feed the outputs back into their respective subunits.

with the current controller inputs. The feed-back gains and biases for delayed outputs appear in the input layer and the computations for both H2 and V2 are duplicated for delayed inputs in order to compute the internal state of the controller. Since the sigmoids are linear near zero, signals can be passed through them by multiplying by a small number, e, then recovered without distortion in a subsequent layer by dividing by the same number. Part of the network is shown in the diagram in figure 4.4. The signals which pass through the linear subunits are similarly scaled so that they do not saturate the sigmoids during normal operation.

The resulting feed-forward network behaves exactly like the conventional controller but has a much greater capacity for learning and adaptation because it has 548 adjustable parameters which are almost all zero. However, it proved virtually impossible to train the network to perform as well as the conventional controller starting from arbitrary connection weights and biases. In fact, the backward error propagation algorithm proved to be unstable for any reasonable learning rate even when training began with an appropriate set of connection weights and biases despite the fact that it was able to reduce output errors to less than 0.1%. The network would, of course, learn the linear functions without difficulty. But altitude and airspeed control were never both satisfactory. Figure 4.5 shows a plot of both actual and desired altitude versus time for the best simulation result obtained after training an artificial neural network initialized with arbitrary connection weights and biases.

The problem can be understood by inspection of figure 4.2. The internal state variable,

$$\mathrm{IHSTA}_{t+1} = (1+\epsilon)\mathrm{IHSTA}_t + \mathrm{KH3}\cdot\mathrm{HI2}(\mathrm{H2}_{t+1} + \mathrm{H2}_t),$$

will saturate regardless of the value of H2 and overwhelm any direct contribution

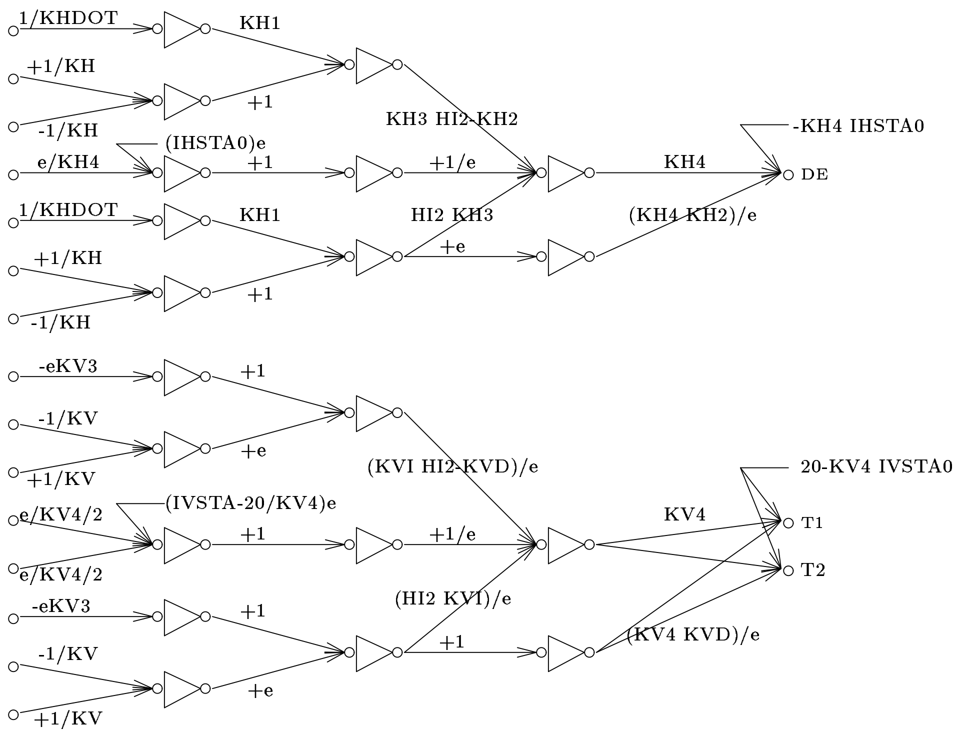


Figure 4.4: The third step in transforming the conventional controller into a neural controller places the feed-back gains and biases in the input layer and duplicates the computation of H2 and V2. The feed-forward network computes the internal state of the controller from delayed controller inputs and outputs. The layers are fully connected but only the non-zero biases and connection weights are shown.

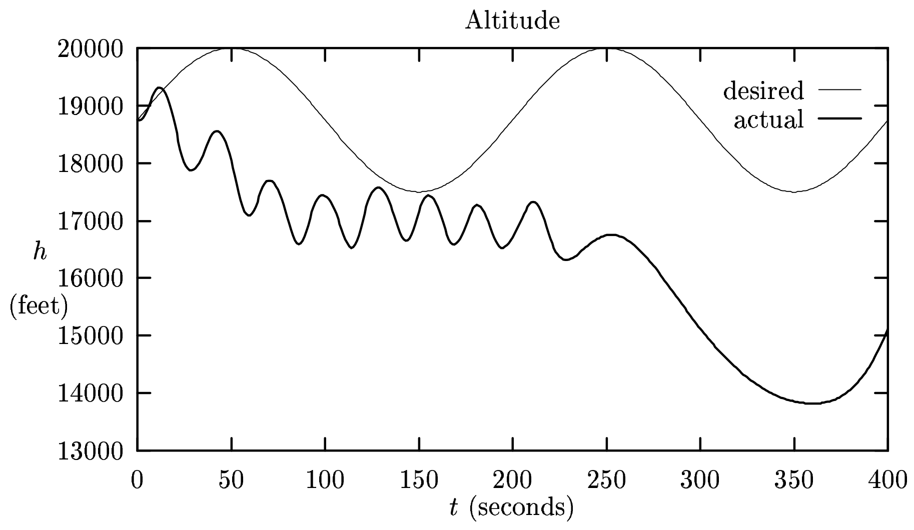


Figure 4.5: The artificial neural network never learned to control altitude. The plot represents the best simulation result obtained. Most other simulations would end in less than five minutes by crashing the airplane.

to DE from H2 through the gain KH2 = 0.08 if the error in the feed-back gain, $\epsilon > 2 \cdot \mathrm{KH3} \cdot \mathrm{HI2} = 0.0004$. The feed-forward network requires at least four connection weights and two biases to effect the feed-back gain. The backward error propagation algorithm can learn sensitive parameters like the feed-back gain to whatever degree of accuracy is required if the learning rate is sufficiently small. But as the learning rate is reduced, improvement in other less sensitive parameters virtually comes to a halt. The problem can be solved by using a more sophisticated (i.e., higher order) learning algorithm but such algorithms are beyond the scope of this research.

This problem is not unique to aircraft or even control systems. In general, complicated nonlinear systems will have at least a few parameters which are very sensitive to estimation errors. Similar difficulties can be expected when using artificial neural networks to identify these systems as suggested by Barto[Bar90]. This does not mean that Barto's approach is wrong. It only means that more sophisticated learning algorithms and significantly more computational power will be required to identify complicated, real-world systems. It appears that Barto's approach should only be used when virtually nothing is known about the system except the inputs and outputs. On the other hand, if a good model of the system already exists as in the case of the computer flight simulator, then it should be used and not discarded.

CHAPTER 5

Teaching Neural Networks to Fly Turns

5.1 Introduction

The purpose of this research is to test a new automatic design methodology for nonlinear controllers that can be used when an accurate conventional computer model exists for the plant (the nonlinear system which is to be controlled). The real-time recurrent learning algorithm[WZ92] is employed to train an artificial neural network to perform an unknown nonlinear control function that results in the desired behavior from the plant. Two different methods are employed to compute partial derivatives for the real-time recurrent learning algorithm. Derivative arithmetic is used to compute the partial derivatives of the state variables at the next time step with respect to the state and control variables at the current time step. The backward error propagation algorithm is used to compute the partial derivatives of the control variables with respect to the current state variables and the control system parameters which are simply the network biases and connection weights.

In order to test the methodology on an important, complicated real-world problem, the artificial neural controller was installed in a computer flight simulator for a high performance fighter aircraft and trained to fly a 2 g coordinated right turn at Mach 1 while maintaining an altitude of 20 000 feet. A diagram of the maneuver is shown in figure 5.1. It is a gentle turn which has a radius

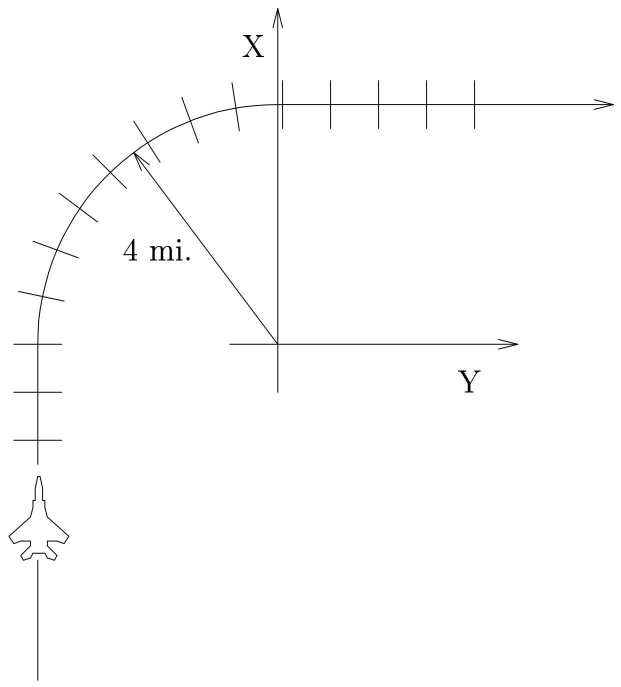


Figure 5.1: The controller functions as a flight test maneuver autopilot. The artificial neural controller is trained to fly a 2 g coordinated right turn at Mach 1 while maintaining an altitude of 20 000 feet. The turn requires about half a minute to complete and has a radius of about 4 miles. The desired state of the aircraft is specified at intervals of 1.0 second (about 0.2 miles) ahead of the aircraft in order to permit the controller to look ahead.

of about 4 miles and requires about half a minute to complete at the speed of sound.

This maneuver is interesting because it exercises all six control variables. In order to turn the aircraft, the controller must learn to bank the aircraft and pitch the nose up in order to “climb” around the turn. It must also increase power to maintain airspeed and altitude then apply right rudder to eliminate sideslip. Also, each control variable affects multiple state variables. For example, too little right rudder causes the aircraft to climb and increases drag which slows the aircraft and widens the turn.

Training an artificial neural controller to fly turns using the real-time recurrent learning algorithm proved to be exceptionally difficult. The problem is that the learning algorithm needs reasonable approximations for the input biases and variations, weight matrix and learning rate before it can begin to learn but the artificial neural controller must complete at least one maneuver before statistics are available which can be used to estimate these parameters accurately. Finding suitable initial estimates for these parameters requires considerable engineering insight and knowledge of the system. Of course, this compromises the ultimate goal which is to completely automate the controller design process.

Hundreds of tedious experiments were required to diagnose problems and to adjust the various parameters used by the learning algorithm. At first, almost none of the experiments worked but eventually the network began to show some evidence that it was learning. There is little point in chronicling all of these experiments or trying to explain why they failed. The remainder of this chapter is devoted to the strategies for configuring a network and learning algorithm that seemed to succeed.

5.2 Configuration

5.2.1 Desired State

The relative importance of the difference between the desired and actual value of the aircraft state variables are shown in table 5.1. The simulator uses a state vector which includes the time t, all twelve state variables in table 3.1 and the time derivatives of all thirteen of the above for a total of twenty six state variables. The desired airspeed V_d = Mach 1, sideslip $\beta_d = 0$ and altitude $h_d = 20\ 000$ feet are constant during the entire maneuver so the respective time derivatives should all be zero. In order to avoid ambiguity at $\psi = \pm\pi$, the desired heading is specified indirectly by the desired velocity in the x-direction $\dot{x}_d = V_d \cos(\psi_d(t))$ and in the y-direction $\dot{y}_d = V_d \sin(\psi_d(t))$. The desired pitch rate $q_d = \dot{\psi}_d\sqrt{3}/2$, yaw rate $r_d = \dot{\psi}_d/2$ and the the rate $\dot{\psi}_d$ at which the desired heading changes are constant so $\dot{q}_d = \dot{r}_d = \ddot{\psi}_d = 0.0$. The desired value of all the other state variables are zero but the angle of attack α, pitch angle θ, bank angle ϕ, distance north x and distance east y are free to assume whatever values are required to satisfy the other constraints.

The real-time recurrent learning algorithm was used as described in section 3.2 to minimize the weighted square error performance measure in equation 3.6. The weight matrix, W, is a diagonal matrix with zero elements along the diagonal corresponding to the unconstrained state variables including time t and the rate of change of time $dt/dt = 1$. All of the other elements were assigned the same arbitrary value (one) meaning that they all have the same relative importance.

	variable	relative importance	desired value	units
0	t_d	0.0	t	sec
1	p_d	0.0	0.0	rad/sec
2	q_d	0.0	$\frac{\sqrt{3}}{2}\dot{\psi}_d$	rad/sec
3	r_d	1.0	$\frac{1}{2}\dot{\psi}_d$	rad/sec
4	V_d	1.0	1037.9	ft/sec
5	α_d	0.0	α	rad
6	β_d	1.0	0.0	rad
7	θ_d	0.0	θ	rad
8	ψ_d	0.0	ψ_d	rad
9	ϕ_d	0.0	ϕ	rad
10	h_d	1.0	20000.0	ft MSL
11	x_d	0.0	x	ft
12	y_d	0.0	y	ft
13	$\dot{t}_d$	0.0	$\dot{t}$	
14	$\dot{p}_d$	0.0	0.0	rad/sec/sec
15	$\dot{q}_d$	0.0	$\frac{\sqrt{3}}{2}\ddot{\psi}_d$	rad/sec/sec
16	$\dot{r}_d$	0.0	$\frac{1}{2}\ddot{\psi}_d$	rad/sec/sec
17	$\dot{V}_d$	0.0	0.0	ft/sec/sec
18	$\dot{\alpha}_d$	0.0	0.0	rad/sec
19	$\dot{\beta}_d$	1.0	0.0	rad/sec
20	$\dot{\theta}_d$	0.0	0.0	rad/sec
21	$\dot{\psi}_d$	1.0	$\dot{\psi}_d$	rad/sec
22	$\dot{\phi}_d$	0.0	0.0	rad/sec
23	$\dot{h}_d$	1.0	0.0	ft/sec
24	$\dot{x}_d$	1.0	$V_d \cos(\psi_d(0))$	ft/sec
25	$\dot{y}_d$	1.0	$V_d \sin(\psi_d(0))$	ft/sec

Table 5.1: The relative importance of the difference between the desired and actual values of the aircraft state variables are set to zero if the difference has no importance. Otherwise, they are arbitrarily set to one.

5.2.2 Network Inputs and Outputs

The artificial neural controller is a multi-layer feed-forward network. There are a total of forty two inputs from the current state and the reference signal. There are five outputs which suffice to produce the six control variables in table 3.2 using a single throttle output for both the right and left throttles. The network outputs are not fed back into the network inputs so there is no state internal to the controller which can be used to help control the aircraft.

5.2.2.1 Input Biases and Variations

The initial estimates for the network input biases and variations shown in table 5.2 were obtained from various sources. The speed of sound is about 1037.9 feet per second at an altitude of 20 000 feet. The angle of attack and the pitch are equal at about 0.029 radians in straight and level flight at 20 000 feet and Mach 1 with the conventional controller. All of the other input biases were set to zero except, of course, $\dot{t} = 1$. The variation in the airspeed, altitude and vertical airspeed were taken from the conventional controller and set to 250 feet per second, 200 feet and 50 feet per second respectively. The variations for t, x, y and $\dot{t}$ were all set to 10^9 so that they would be ignored by the controller. The variations for all of the other inputs were set to one except for the variations for ψ and $\dot{V}$ which were arbitrarily set to 64 radians and 256 feet per second per second respectively reducing the effect of both inputs on the control function.

5.2.2.2 Reference Signal

A reference model provides the reference signal $r(t)$ which is the desired heading, $\psi_d(t + k\Delta t)$, of the aircraft at sixteen intervals of $\Delta t = 1.0$ seconds (about 0.2

	input	bias	variation	units
0	t	0.0	10^9	sec
1	p	0.0	1.0	rad/sec
2	q	0.0	1.0	rad/sec
3	r	0.0	1.0	rad/sec
4	V	1037.9	250.0	ft/sec
5	α	0.029	1.0	rad
6	β	0.0	1.0	rad
7	θ	0.029	1.0	rad
8	ψ	0.0	64.0	rad
9	ϕ	0.0	1.0	rad
10	h	20000.0	200.0	ft MSL
11	x	0.0	10^9	ft
12	y	0.0	10^9	ft
13	$\dot{t}$	1.0	10^9	
14	$\dot{p}$	0.0	1.0	rad/sec/sec
15	$\dot{q}$	0.0	1.0	rad/sec/sec
16	$\dot{r}$	0.0	1.0	rad/sec/sec
17	$\dot{V}$	0.0	256.0	ft/sec/sec
18	$\dot{\alpha}$	0.0	1.0	rad/sec
19	$\dot{\beta}$	0.0	1.0	rad/sec
20	$\dot{\theta}$	0.0	1.0	rad/sec
21	$\dot{\psi}$	0.0	1.0	rad/sec
22	$\dot{\phi}$	0.0	1.0	rad/sec
23	$\dot{h}$	0.0	50.0	ft/sec
24	$\dot{x}$	0.0	1037.9	ft/sec
25	$\dot{y}$	0.0	1037.9	ft/sec
26	$\psi_{d,0}$	0.0	1.0	rad
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
41	$\psi_{d,15}$	0.0	1.0	rad

Table 5.2: Initial estimates for the network input biases and variations were obtained from various sources.

	output	bias	variation	units
1	δ_a	0.0	1.0	rad
2	δ_e	-0.032	1.0	rad
3	δ_t	0.0	1.0	rad
4	δ_r	0.0	1.0	rad
5	T_x	77.3	90.0	degrees

Table 5.3: Initial estimates for the network output biases and variations were obtained from straight and level flight with the conventional controller.

miles) ahead of the aircraft. This gives the controller a "look-ahead" capability much like that provided by terrain-following or obstacle avoidance radar systems which permits the controller to anticipate and compensate for sluggish response to control action.

5.2.2.3 Output Biases and Variations

The initial estimates for the network output biases and variations shown in table 5.3 were obtained from straight and level flight with the conventional controller. The bias for the elevon deflection and throttle angle were -0.032 radians and 77.3 degrees respectively. All of the other biases were zero. The variation in the throttle angle was set to 90 degrees and the other variations were arbitrarily set to one.

5.2.2.4 Initializing the Network

Both network input and output biases and variations were used to initialize the artificial neural controller. First, the network biases and connection weights are assigned small random values with a normal distribution about zero. Then the network connection weights and biases are adjusted:

$$\begin{aligned} W_0 &\leftarrow W_0 \text{diag}\left\{\frac{1}{\Delta v_{0,j}}\right\} \\ b_0 &\leftarrow b_0 - W_0 \bar{v}_0 \end{aligned} \tag{5.1}$$

and

$$\begin{aligned} W_{K-1} &\leftarrow \text{diag}\left\{\Delta v_{K,j}\right\} W_{K-1} \\ b_{K-1} &\leftarrow \text{diag}\left\{\Delta v_{K,j}\right\} b_{K-1} + \bar{v}_K \end{aligned} \tag{5.2}$$

The adjustments to the input layer incorporate input normalization. The adjustments to the output layer produce controller outputs which permit the aircraft simulator to fly nearly straight and level for some time before the aircraft crashes so that the network has time to begin learning.

5.2.2.5 Input Normalization

The input biases $\bar{v}_0$ and variations Δv_0 are used to normalize the inputs to the artificial neural controller. First, the input biases are subtracted from the inputs

$$v_0 \leftarrow v_0 - \bar{v}_0. \tag{5.3}$$

Next, the heading is subtracted from each element of the reference signal

$$r \leftarrow r - \psi \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{5.4}$$

Finally, each input is divided by the respective variation

$$v_0 \leftarrow \text{diag}\left\{\frac{1}{\Delta v_{0,j}}\right\} v_0. \tag{5.5}$$

Normalization is incorporated into the input layer of the artificial neural controller and must be dis-incorporated from it

$$\begin{aligned} b_0 &\leftarrow b_0 + W_0 \bar{v}_0 \\ X_\psi &\leftarrow X_\psi + R \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \\ W_0 &\leftarrow W_0 \mathrm{diag}\left\{\Delta v_{0,j}\right\} \end{aligned} \tag{5.6}$$

before learning begins then re-incorporated into it

$$\begin{aligned} W_0 &\leftarrow W_0 \mathrm{diag}\left\{\frac{1}{\Delta v_{0,j}}\right\} \\ X_\psi &\leftarrow X_\psi - R \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \\ b_0 &\leftarrow b_0 - W_0 \bar{v}_0 \end{aligned} \tag{5.7}$$

after learning ends. Here X_ψ represents the column of W_0 corresponding to the heading input of v_0. The learning algorithm permits an optional "fudge factor" to be applied to the input variations in order to adjust the size of the variable inputs relative to the constant input which is always 1.

5.3 Training

It proved practically impossible to train a multi-layer network starting with small randomly perturbed network biases and connection weights. The connection weights in the output layer attenuate the output error as it propagates backward so strongly that the biases and connection weights in the input layer change very slowly. Similarly, the connection weights in the input layer attenuate the input signal so strongly that the signal produced by the hidden units is too weak to cause significant changes in the connection weights in the output layer. The only significant effect of the learning algorithm is to adjust the biases in the output layer until they are equal to the required output. The learning algorithm attempts to fly the desired trajectory but the aircraft eventually crashes. Initializing the network with larger biases and connection weights only causes the aircraft to crash more quickly.

It was relatively easy to train a linear controller to fly shallow turns. A linear controller is a single layer of biases and connection weights with no nonlinear output units. If the state of the aircraft does not deviate too far from straight and level flight, it behaves like a linear system and a linear controller is an adequate autopilot. The single layer network was important because it permitted us to verify that the derivative arithmetic was working correctly. After the single layer network learned to fly 2 g coordinated turns, it was converted into a two layer network which was used to verify that the derivatives would propagate backward through the output layer correctly.

5.3.1 The One Layer Network

5.3.1.1 Shallow Turns

Figure 5.2 shows the results of the first experiment. There are 64 seconds of straight and level flight before the maneuver begins. The maneuver is a shallow 90 degree turn to the right which should have a radius of about 64 miles and require about 512 seconds to complete. There are an additional 64 seconds of straight and level flight after the turn is completed. The simulator is initialized with the aircraft heading due north at Mach 1 and 20 000 feet but the aircraft begins to deviate from straight and level flight almost immediately. It looses over 1000 feet in altitude, 15 feet per second in airspeed and banks in excess of 90

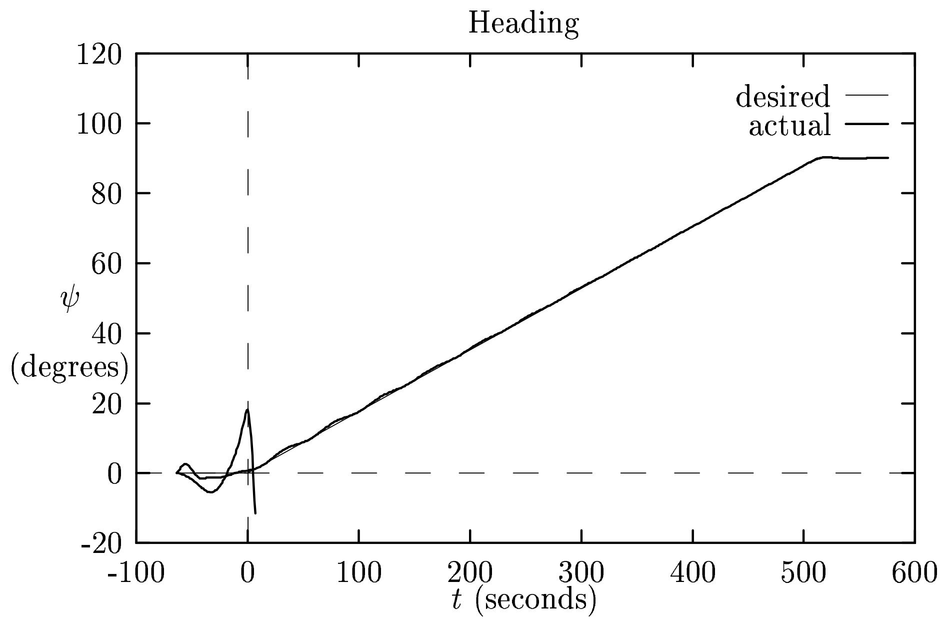


Figure 5.2: The artificial neural network learned to fly a turn.

degrees before the learning algorithm automatically reinitializes the simulation[1]. The aircraft begins to meander again but the artificial neural controller appears to have learned something from its first attempt to fly the maneuver. The aircraft begins to turn right and after about five minutes, it settles onto the desired flight path then transitions smoothly into straight and level flight heading due east at the end of the maneuver.

The minimum decay rate was 2^{-7} per time step which represents a half-life of about 2 seconds for G_0 at 50 time steps per second. A fudge factor of 2^{-6} was applied to the input variations to increase the size of the variable inputs relative to the constant input and discourage the learning algorithm from simply adjusting the network biases to equal the desired controller outputs. The learning rate, $\eta = 2^{-42}$, was determined by trial and error to be the highest rate for which

[1]Like any good flight instructor, the learning algorithm takes control away from the student autopilot well before the aircraft crashes not only to avoid a dangerous situation but also to avoid an experience that is useless or even harmful to the learning process.

the learning algorithm remained stable for the entire experiment. These same settings were used in four additional experiments:

1. a 90 degree left turn with an initial heading of 0 degrees,
2. a 90 degree right turn with an initial heading of −90 degrees,
3. a 90 degree left turn with an initial heading of 90 degrees and
4. a 90 degree right turn with an initial heading of 0 degrees

in an attempt to eliminate any dependence upon the initial heading or the direction of the turn.

5.3.1.2 Revisions

The output from the learning algorithm includes averages and deviations for the differences between the desired and actual state as well as all of the network inputs and outputs. These were used to calculate better estimates for the relative importance of the differences (table 5.4) and the relative variation in the inputs (table 5.5.) There were two revisions. The first revision was made just after the artificial neural controller had learned to fly turns with a 64 mile radius then the fudge factor was increased to 2^0 and the controller learned to fly turns with a 32, 16, 8 and 4 mile radius using, in each case, the largest learning rate for which the learning algorithm would remain stable. Again, the artificial neural controller was taught to fly both left and right turns starting at various headings. The second revision was made just after the artificial neural controller had learned to fly turns with a 4 mile radius.

	variable	first revision	second revision
0	t_d	0.0	0.0
1	p_d	0.0	$3 \cdot 10^5$
2	q_d	0.0	$2 \cdot 10^6$
3	r_d	$4 \cdot 10^9$	$2 \cdot 10^6$
4	V_d	100.0	0.2
5	α_d	0.0	0.0
6	β_d	$8 \cdot 10^9$	$5 \cdot 10^9$
7	θ_d	0.0	0.0
8	ψ_d	0.0	0.0
9	ϕ_d	0.0	0.0
10	h_d	100.0	$3 \cdot 10^{-2}$
11	x_d	0.0	0.0
12	y_d	0.0	0.0
13	$\dot{t}_d$	0.0	0.0
14	$\dot{p}_d$	0.0	0.0
15	$\dot{q}_d$	0.0	0.0
16	$\dot{r}_d$	0.0	0.0
17	$\dot{V}_d$	0.0	0.0
18	$\dot{\alpha}_d$	0.0	$4 \cdot 10^7$
19	$\dot{\beta}_d$	$1 \cdot 10^{11}$	$6 \cdot 10^{10}$
20	$\dot{\theta}_d$	0.0	$4 \cdot 10^7$
21	$\dot{\psi}_d$	$4 \cdot 10^9$	$3 \cdot 10^6$
22	$\dot{\phi}_d$	0.0	$3 \cdot 10^5$
23	$\dot{h}_d$	100.0	0.2
24	$\dot{x}_d$	1.0	1.0
25	$\dot{y}_d$	1.0	1.0

Table 5.4: The relative importance of the difference between the desired and actual values of the aircraft state variables was revised twice – once after the artificial neural controller had learned to fly turns with a 64 mile radius and again after it had learned to fly turns with a 4 mile radius.

	input	first revision	second revision	units
0	t	$1 \cdot 10^{+9}$	$1 \cdot 10^{+9}$	sec
1	p	$1 \cdot 10^{-3}$	$3 \cdot 10^{-2}$	rad/sec
2	q	$3 \cdot 10^{-4}$	$2 \cdot 10^{-2}$	rad/sec
3	r	$3 \cdot 10^{-3}$	$1 \cdot 10^{-2}$	rad/sec
4	V	3.0	10.0	ft/sec
5	α	$1 \cdot 10^{-4}$	$1 \cdot 10^{-2}$	rad
6	β	$3 \cdot 10^{-4}$	$7 \cdot 10^{-4}$	rad
7	θ	$1 \cdot 10^{-4}$	$6 \cdot 10^{-3}$	rad
8	ψ	0.1	1.0	rad
9	ϕ	0.1	0.4	rad
10	h	3.0	30.0	ft
11	x	$1 \cdot 10^{+9}$	$1 \cdot 10^{+9}$	ft
12	y	$1 \cdot 10^{+9}$	$1 \cdot 10^{+9}$	ft
13	$\dot{t}$	$1 \cdot 10^{+9}$	$1 \cdot 10^{+9}$	
14	$\dot{p}$	$1 \cdot 10^{-3}$	$7 \cdot 10^{-3}$	rad/sec/sec
15	$\dot{q}$	$6 \cdot 10^{-4}$	$1 \cdot 10^{-2}$	rad/sec/sec
16	$\dot{r}$	$3 \cdot 10^{-4}$	$2 \cdot 10^{-3}$	rad/sec/sec
17	$\dot{V}$	$1 \cdot 10^{-2}$	0.7	ft/sec/sec
18	$\dot{\alpha}$	$5 \cdot 10^{-5}$	$2 \cdot 10^{-3}$	rad/sec
19	$\dot{\beta}$	$7 \cdot 10^{-5}$	$2 \cdot 10^{-4}$	rad/sec
20	$\dot{\theta}$	$7 \cdot 10^{-5}$	$2 \cdot 10^{-3}$	rad/sec
21	$\dot{\psi}$	$3 \cdot 10^{-3}$	$2 \cdot 10^{-2}$	rad/sec
22	$\dot{\phi}$	$1 \cdot 10^{-3}$	$3 \cdot 10^{-2}$	rad/sec
23	$\dot{h}$	0.1	5.0	ft/sec
24	$\dot{x}$	1000.0	1037.9	ft/sec
25	$\dot{y}$	1000.0	1037.9	ft/sec
26	$\psi_{d,0}$	1.0	1.0	rad
27	$\psi_{d,1}$	1.0	1.0	rad
28	$\psi_{d,2}$	1.0	2.0	rad
29	$\psi_{d,3}$	1.0	3.0	rad
30	$\psi_{d,4}$	1.0	3.0	rad
31	$\psi_{d,5}$	1.0	4.0	rad
32	$\psi_{d,6}$	1.0	5.0	rad
33	$\psi_{d,7}$	1.0	6.0	rad
34	$\psi_{d,8}$	1.0	6.0	rad
35	$\psi_{d,9}$	1.0	7.0	rad
36	$\psi_{d,10}$	1.0	8.0	rad
37	$\psi_{d,11}$	1.0	9.0	rad
38	$\psi_{d,12}$	1.0	10.0	rad
39	$\psi_{d,13}$	1.0	10.0	rad
40	$\psi_{d,14}$	1.0	10.0	rad
41	$\psi_{d,15}$	1.0	10.0	rad

Table 5.5: Initial estimates for the network input variations were revised twice – once after the artificial neural controller had learned to fly turns with a 64 mile radius and again after it had learned to fly turns with a 4 mile radius.

5.3.1.3 Steep Turns

The 4 mile radius turns were very unsteady but slowly improved with training. Figure 5.3 shows the result obtained after several cycles through left and right turns at different initial headings. The turn is still unsteady but the actual heading never deviates from the desired heading by more than about seven degrees. The artificial neural controller must bank the aircraft to the right about 60 degrees in order to make a 2 g right turn. This converts part of the lift into a centripetal force to the right and reduces the resistance to gravity by one half. The aircraft begins to drop like a rock and the artificial neural controller responds by pulling back on the stick (decreasing δ_e) in order to increase the lift by increasing the angle of attack which slows the rate of descent but also tightens the turn. When the actual heading begins to exceed the desired heading, the controller attempts to regain some lost altitude by rolling back to the left. Figure 5.4 shows the rocking motion in the bank angle and the relationship between the altitude and the bank angle.

Figure 5.5 shows that the aircraft looses over twenty feet per second in airspeed and almost one hundred feet in altitude during the 2 g right turn. These are relatively small losses and the artificial neural controller performs well for a novice pilot but there appears to be a fundamental problem with the learning algorithm. The conventional controller uses the elevon deflection to control altitude and uses the throttle to control airspeed but every stick-and-rudder pilot uses the stick to control airspeed and the throttle to control altitude. Apparently, the learning algorithm couldn't decide which method was correct. It uses the elevon deflection to control both altitude and airspeed and ignores the throttle entirely.

Figure 5.6 shows that sideslip is negligible. The artificial controller had no trouble learning how to apply the correct deflection for the rudder (about -0.3

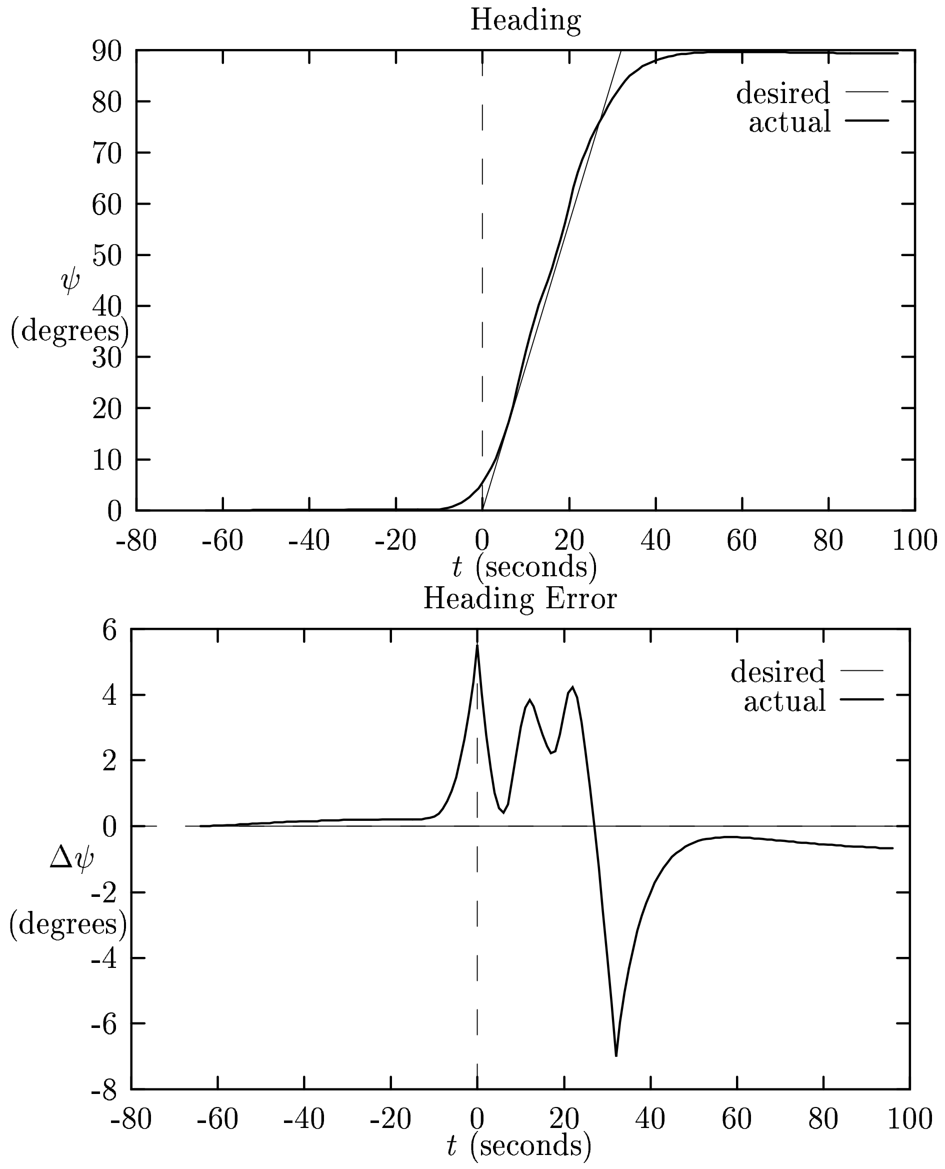
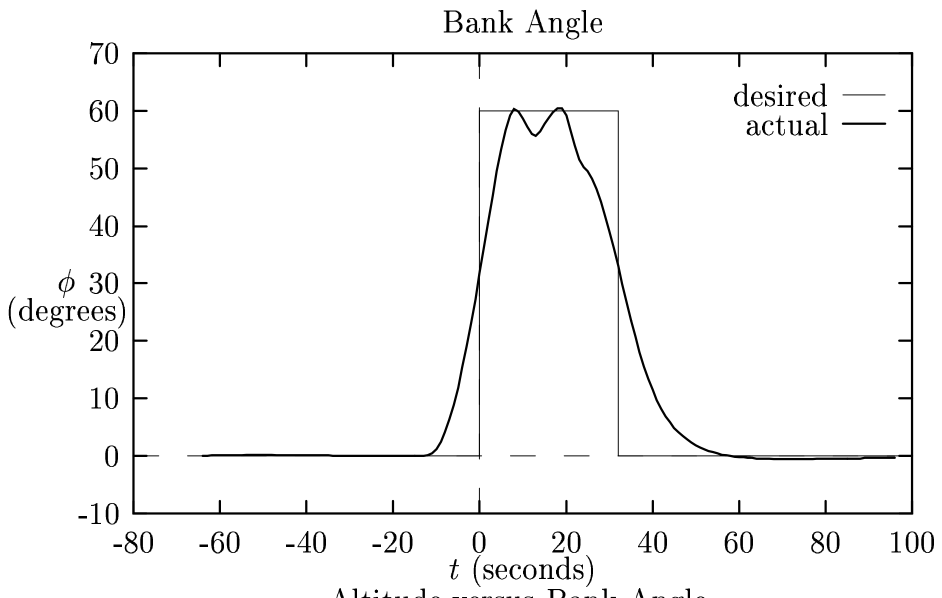


Figure 5.3: An artificial neural controller with just one layer of biases and connection weights (a linear controller) learned to fly turns with a radius of four miles at Mach 1 and 20 000 feet. The actual heading never deviates from the desired heading by more than about seven degrees.



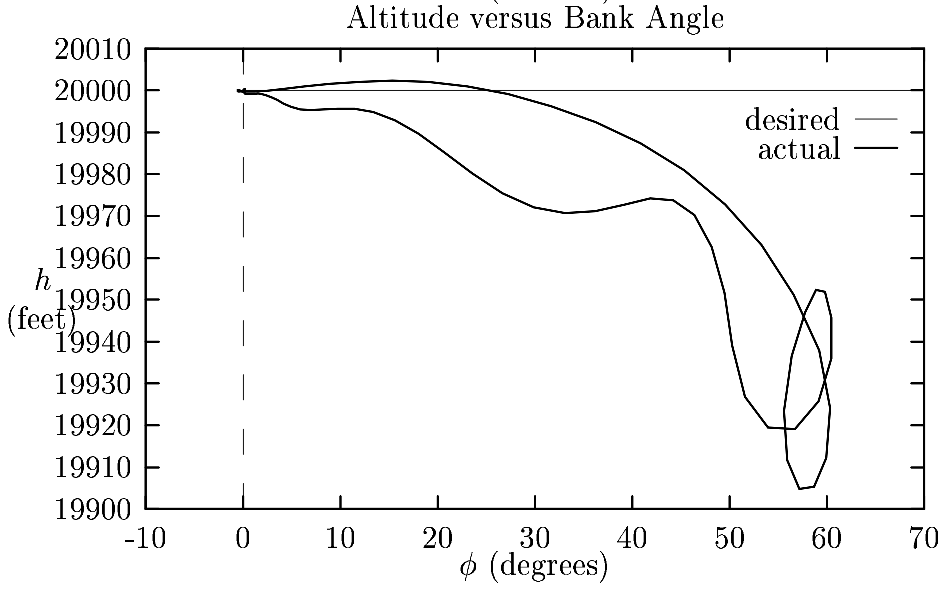


Figure 5.4: The artificial neural controller must bank the aircraft to the right about 60 degrees in order to make a 2 g right turn. The turn is unsteady because the artificial neural controller attempts to regain lost altitude by rolling back to the left.

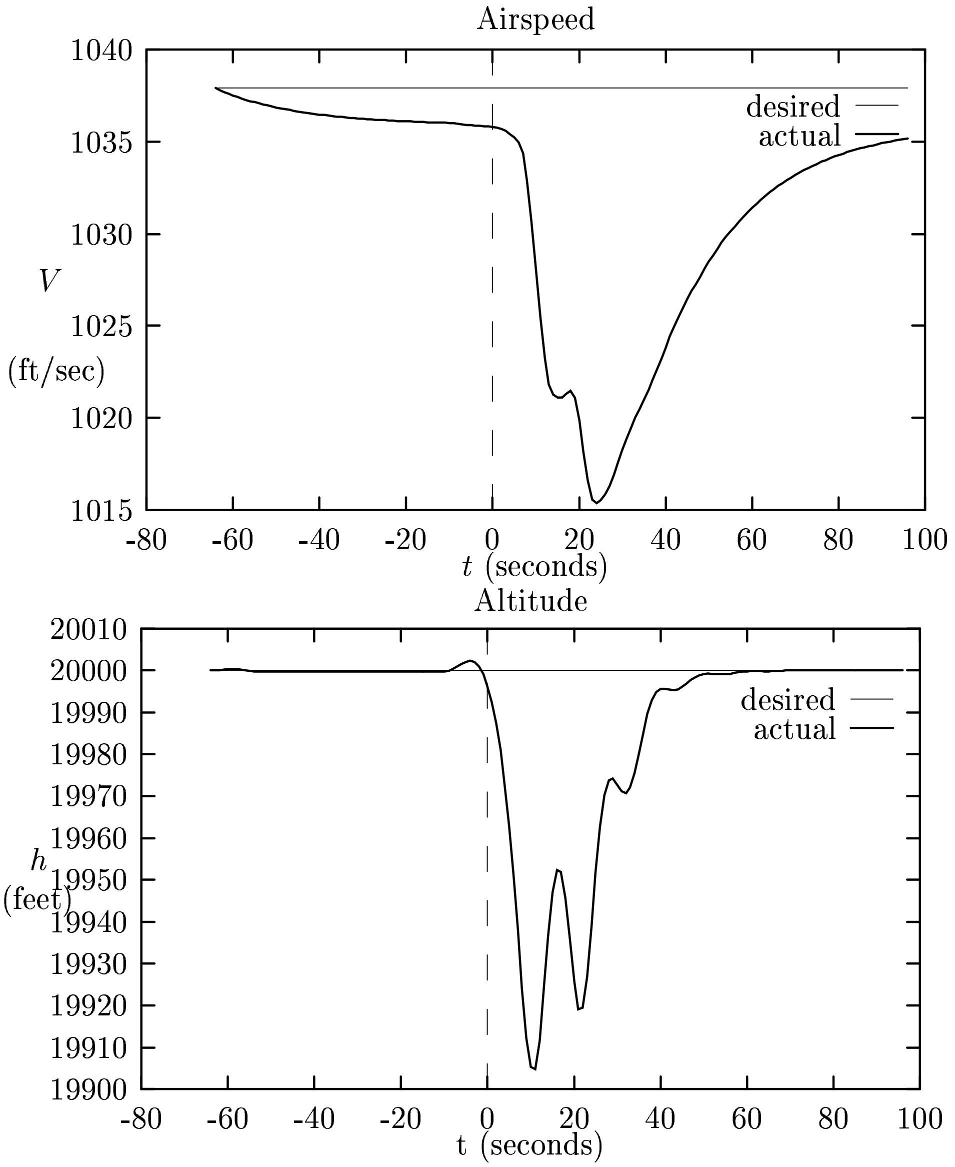


Figure 5.5: The aircraft looses over twenty feet per second in airspeed and almost one hundred feet in altitude.

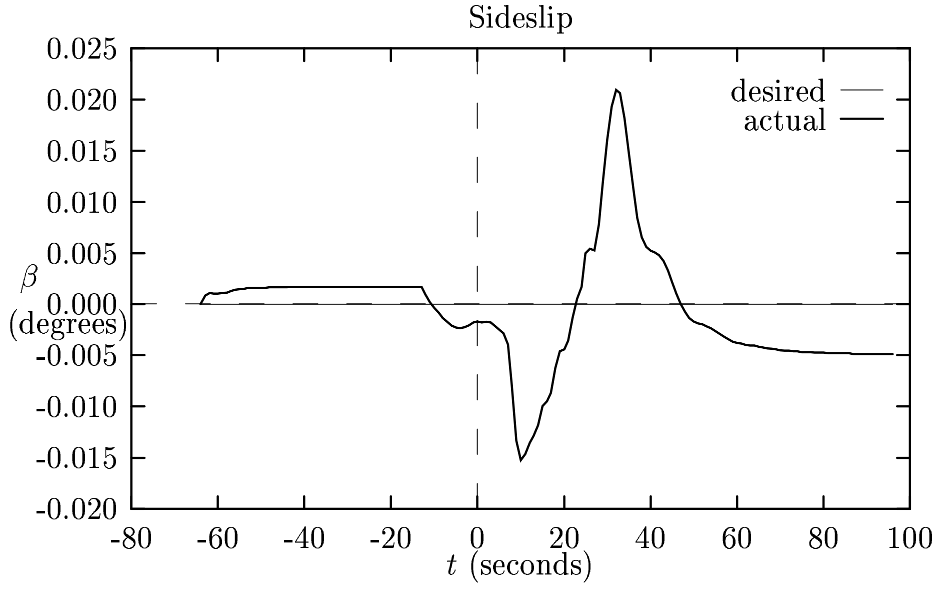


Figure 5.6: The artificial neural controller applies about 0.3 degrees right rudder to minimize sideslip.

degrees in this case) required to fly a coordinated turn.

Evidently, the artificial neural controller makes good use of the look ahead capability provided by the reference signal. The aircraft begins to roll into the turn almost 15 seconds before the beginning of the maneuver and begins to roll back out of the turn almost 15 seconds before the end of the maneuver. For any given control output except the throttle, the network connection weights associated with the reference signal all have about the same value. Since this is a linear controller, only the weights associated with the aileron and differential elevon deflections are useful for both right and left turns.

The biases and connection weights in the artificial neural controller were compared with the biases and gains in the conventional controller. They are about twice as large and have the the same sign for the elevon deflection but have no resemblance at all for the other control outputs. Apparently, the learning algorithm was using an entirely different strategy than the designer of the conventional controller for flying the aircraft. The gain (-0.535) from the conventional controller was substituted for the connection weight ($1.4 \cdot 10^{-6}$) in the artificial neural controller corresponding to $\partial T_x/\partial V$ and $0.535 \cdot 1037.9 \approx 555.3$ was added to network bias. Figure 5.7 shows the result of this alteration. The turn became more steady but also became tighter varying up to almost eight degrees even after extensive training.

Although the learning algorithm provides signals sufficient to adjust the connection weight corresponding to $\partial T_x/\partial V$ correctly, it also supplies other, much larger, alternating signals which, over time, tend to reduce the the connection weight to nearly zero. It may be possible to devise a maneuver which isolates the dependency of the airspeed on throttle angle but this investigation was not essential to the research and was set aside. The alteration to the throttle control output was retained, however, in all subsequent experiments because it permitted improvement of the other controls to continue.

5.3.2 The Two Layer Network

The single layer network, $N(42,5)$, was embedded in a two layer network, $N(42,42,5)$, with single hidden layer of 42 identical nonlinear sigmoidal processing units which compute the hyperbolic tangent of their respective activations. First, the input normalization is dis-incorperated from and the output normalization was re-incorperated into the single layer network using the revised output variations shown in table 5.6. Two different initial configurations for the two layer network were tested. In the first configuration, the input layer is a 42×42 identity matrix with a small amount of noise added onto it and the output layer is the normalized

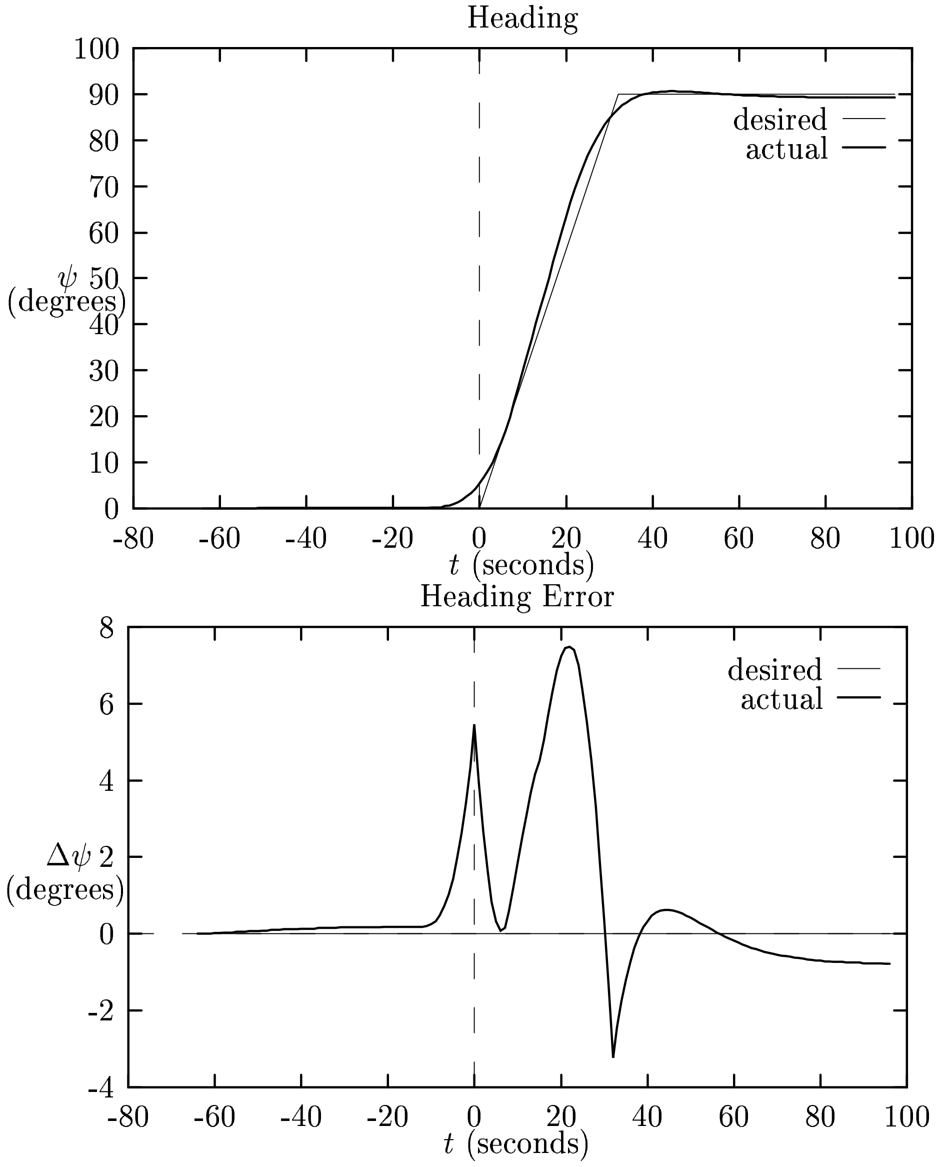


Figure 5.7: A bias and connection weight in the artificial neural controller were altered to control the throttle in the same manner as the conventional controller. This resulted in a more steady but tighter turn which slowly improved with training.

	output	revision	units
1	δ_a	0.002	rad
2	δ_e	0.010	rad
3	δ_t	0.002	rad
4	δ_r	0.002	rad
5	T_x	2.000	degrees

Table 5.6: Initial estimates for the network output variations were revised after the network had learned to fly turns with a 4 mile radius.

single layer network.

$$\begin{aligned} b_1 &\leftarrow b_0 \\ W_1 &\leftarrow W_0 \\ b_0 &\leftarrow 0 \\ W_0 &\leftarrow I_{42\times 42} \end{aligned} \tag{5.8}$$

In the second configuration, both input and output layers were initialized with a small amount of noise then the normalized single layer network was added to the first five rows of the input layer and a 5×5 identity matrix was added to the first five columns of the output layer connection weight matrix.

$$\begin{aligned} b_1 &\leftarrow 0 \\ W_1 &\leftarrow [I_{5\times 5}, 0] \\ b_0 &\leftarrow b_0 \\ W_0 &\leftarrow \begin{bmatrix} W_0 \\ 0 \end{bmatrix} \end{aligned} \tag{5.9}$$

The input layer biases and connection weights were multiplied by a small number (0.125) and the output layer connection weights were divided by the same number. Finally, the input normalization was re-incorperated into the input layer and the output normalization was dis-incorporated from the output layer.

The initial configuration of the two layer artificial neural controller behaves almost exactly like the single layer network embedded within it but it has a much greater potential for learning. This expanded learning capability increased simulation time by a factor of ten but, unfortunately, neither initial configuration of the two layer network was improved much by additional training. Part of the problem may be that the initial configuration represents a local minimum in the performance function but this is difficult to determine because so many of the eigenvalues of the second gradient are very small. The small variation in the outputs from the hidden layer was a more obvious problem. These variations can't be much larger than about 0.1 in the initial (linear) configuration without squashing the activation signal but many of them were much smaller. In an attempt to accelerate learning, the row of the input layer matrix corresponding to variation $\Delta v_{1,j} < 0.1$ was multiplied by $0.1/\Delta v_{1,j}$ and the corresponding column of the output layer matrix was divided by $0.1/\Delta v_{1,j}$. This did accelerate the learning rate but did not improve the turns. Figure 5.8 shows the best turn executed by a two layer artificial neural controller. It did not perform much better than the single layer controller. The actual heading exceeds the desired heading by more than seven degrees but the learning algorithm is actually trying to minimize the deviation from the *true heading* which lags behind the actual heading by $\alpha \sin(\phi) \approx 3$ degrees when $\alpha = 3.5$ degrees and $\phi = 60$ degrees. The true heading undershoots and overshoots the desired heading by about the same amount – approximately five degrees. The aircraft rolls out of the turn too late and overshoots and undershoots the desired heading again at the end of the maneuver. The controller did not learn to increase the throttle in anticipation of a turn but simply responds to a loss in airspeed. The aircraft looses almost 14 feet per second in airspeed and 180 feet in altitude which it does not completely regain until after it rolls out of the turn.

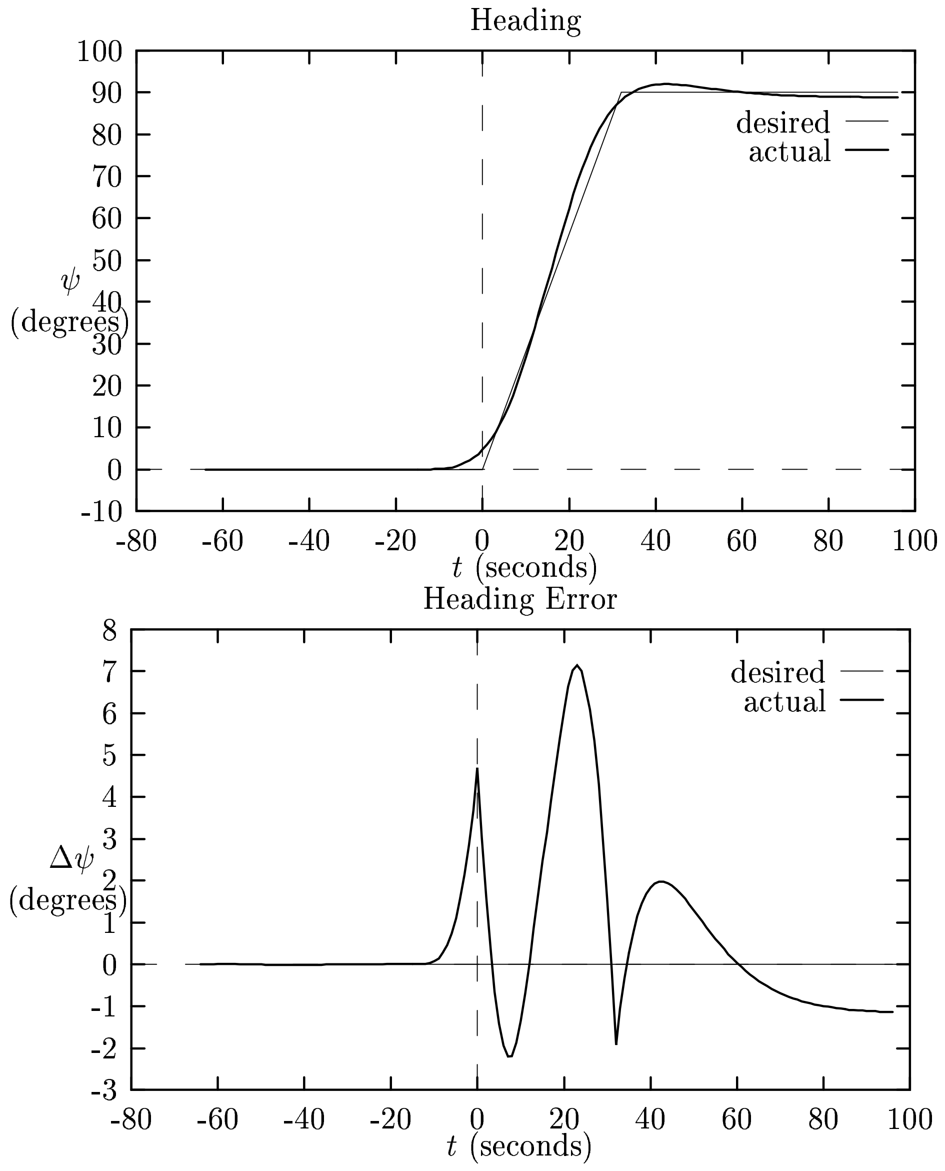


Figure 5.8: The two layer artificial neural controller did not perform much better than the single layer artificial neural controller. The actual heading exceeds the desired heading by more than seven degrees. The aircraft rolls out of the turn too late and overshoots the desired heading at the end of the maneuver. It looses almost 14 feet per second in airspeed and 180 feet in altitude which it does not completely regain until after it rolls out of the turn.

5.4 Stability

A controller must be able to tolerate some noise. In particular, and autopilot must be able to tolerate mild air turbulence. The simulator does not accomodate wind much less model air turbulence. In order to simulate wind shear gusting up to about 50 miles per hour, a small amount of noise $\dot{\nu}_{t+1}$ was added to the time rate of change $\dot{\alpha}_{t+1}$ and $\dot{\beta}_{t+1}$ in the angle of attack and sideslip respectively. In each case, the noise was computed using

$$\begin{aligned}\ddot{\nu}_{t+1} &= \rho_{t+1} - \Delta t\left(\gamma\ddot{\nu}_t + \omega^2\dot{\nu}_t\right) \\ \dot{\nu}_{t+1} &= \dot{\nu}_t + \Delta t\ddot{\nu}_{t+1} \qquad (5.10)\end{aligned}$$

where $\omega = \gamma = 2\pi$ per second, $\Delta t = 0.02$ seconds and ρ is normally distributed random noise with zero mean and $1/1024$ radians per second per second variance. Figure 5.9 shows the Fourier cosine amplitudes

$$\nu_t \approx 1.56 + \sum_{k=1} c_k \cos\left(\frac{\pi k t}{8192}\right)$$

of this noise if it were to accumulate using $\nu_{t+1} = \nu_t + \Delta t\dot{\nu}_{t+1}$ where $\nu_0 = \dot{\nu}_0 = \ddot{\nu}_0 = 0$. Figure 5.10 shows that the artificial neural controller does not permit the affect of mild turbulence to accumulate. It acts to maintain the desired angle of attack and minimize sideslip. Although the aircraft does complete the turn, figure 5.11 shows that the artificial neural controller is seriously affected by mild turbulence.

Figure 5.12

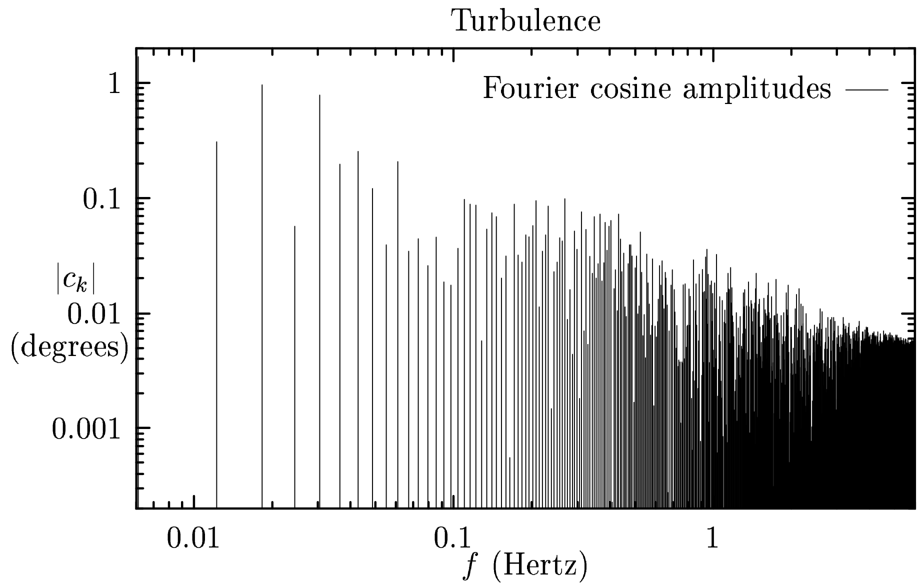
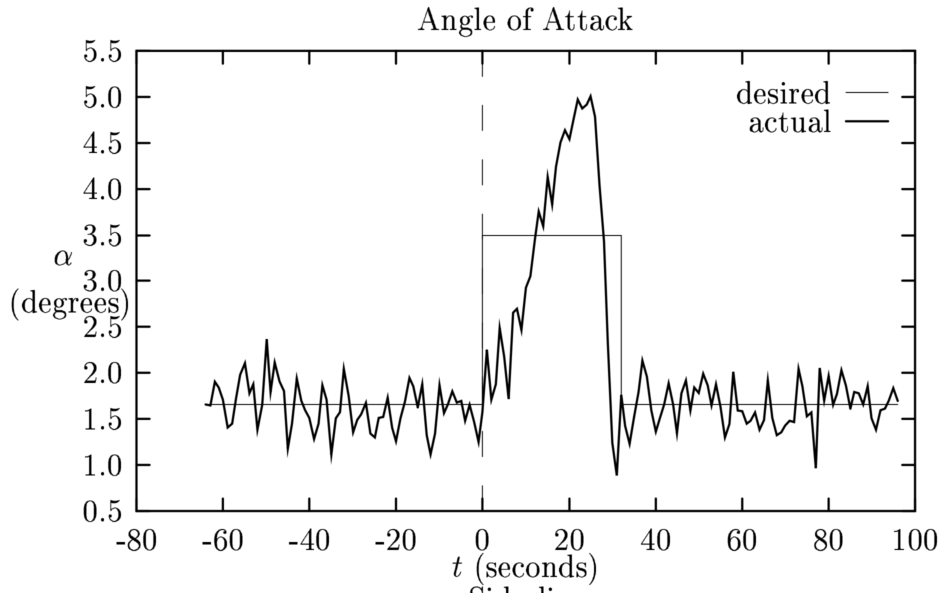


Figure 5.9: Mild air turbulence was modeled by adding noise $\dot{\nu}_{t+1}$ to $\dot{\alpha}_{t+1}$ and $\dot{\beta}_{t+1}$. The Fourier cosine amplitudes of the accumulated noise $\nu_{t+1} = \nu_t + \Delta t \dot{\nu}_{t+1}$ are shown above.



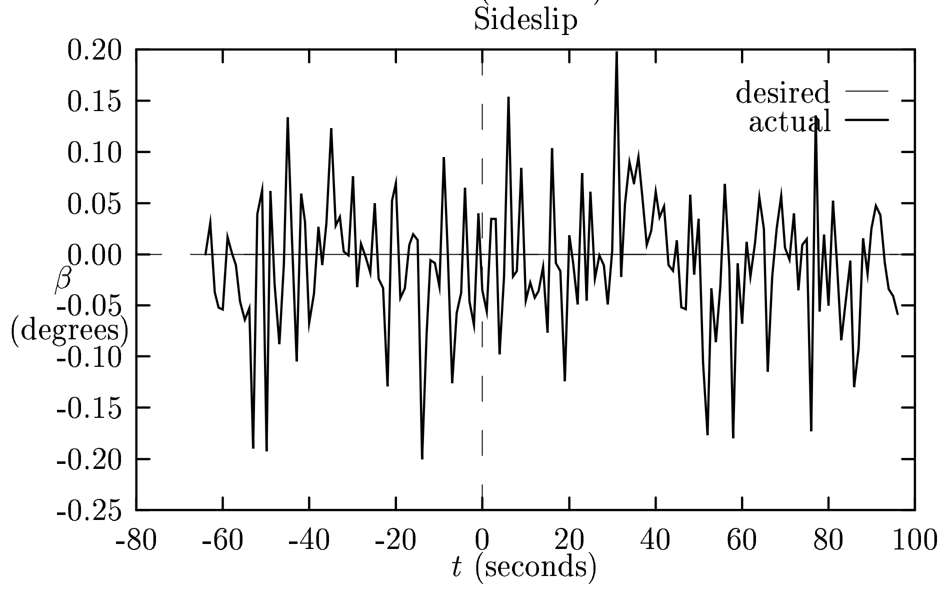


Figure 5.10: The artificial neural controller does not permit the affect of mild turbulence to accumulate. It acts to maintain the desired angle of attack and minimize sideslip.

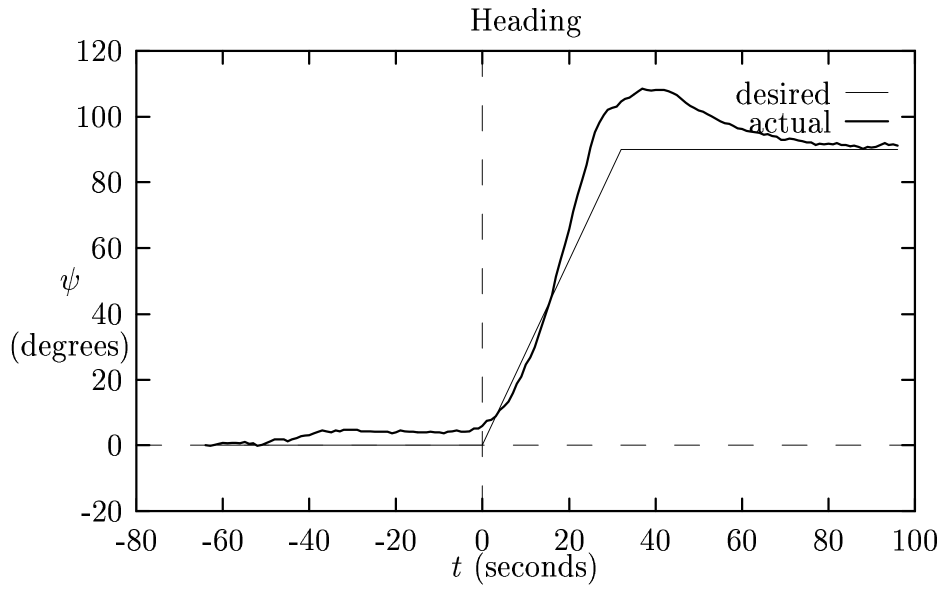


Figure 5.11: Although the aircraft does complete the turn, the artificial neural controller is seriously affected by mild turbulence.

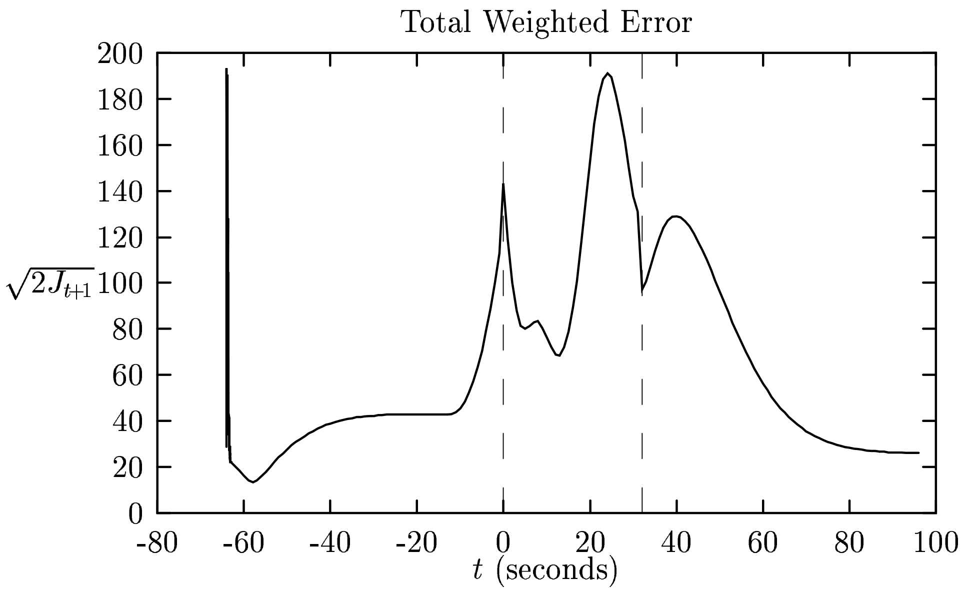


Figure 5.12: The artificial neural controller immediately suppresses the transients errors which occur when the simulation is initialized. The total weighted error increases before the maneuver begins. The controller trades airspeed for altitude anticipating a loss in altitude during the turn. It take advantage of the look ahead capability provided by the reference signal by starting to roll into the turn before the maneuver begins and roll out of the turn before the maneuver ends.

CHAPTER 6

Conclusions

There is something fascinating about science. One gets such wholesome returns of conjecture out of such a trifling investment of fact. *Mark Twain*

The man who sets out to carry a cat by its tail learns something that will always be useful and which never will grow dim or doubtful. *Mark Twain*

The original intent of this research was to test a novel variation of the real-time recurrent learning algorithm which uses derivative arithmetic to propagate partial derivatives through a conventional computer model of the system to be controlled. Derivative arithmetic appears to have performed flawlessly from the beginning and there is little more to conclude about it. Still, the artificial neural controller would not learn to fly turns. The learning algorithm was either too slow or unstable. Hundreds of experiments were required to diagnose the problem and adjust parameters just so that the artificial neural controller could begin to learn. This experience compels us to attempt to conclude something about the viability and the future of automatic controller design. No doubt, someone will attempt to extrapolate profound and probably erroneous conclusions about natural neural networks from this research. There is no way to prevent this but we feel obliged to indicate which of the more obvious implications are reasonable and which are not.

6.1 Derivative Arithmetic

The introduction of derivative arithmetic into the real-time recurrent learning algorithm is the single most important contribution of this research. It provides the controller design engineer with immediate access to partial derivatives of all variables with respect to state and control variables in the computer model with little more effort and expense than that required to re-compile the program source code. This means that it is now possible to use the real-time recurrent learning algorithm to train artificial neural controllers for any of thousands of existing simulators.

Derivative arithmetic itself is an ancient and obvious invention which is easy to implement but was always difficult to use until programming languages which support user defined types and operator overloading were introduced. Two of these new languages, Fortran 90 and C++, are especially important. Because the Fortran 77 language is a subset of Fortan 90 and the C language is a subset of C++, derivative arithmetic can easily be substituted for normal floating-point arithmetic in most of the existing scientific and engineering simulations.

Since few of the program variables in a typical simulation depend upon all of the state and control variables, much of the computational effort in derivative arithmetic is wasted computing and propagating zeros. This may or may not have a significant affect on the efficiency of the real-time recurrent learning algorithm. Derivative arithmetic in the flight simulator represented a small fraction of the total computational load while training the artificial neural controller using the real-time recurrent learning algorithm. There is very little incentive to improve the efficiency of partial derivative computation in simulations of comparable size and complexity.

6.2 Automatic Controller Design

Artificial neural controllers can be trained to mimic human operators but the ultimate goal of automatic controller design is to eliminate the need for human intervention in the design process. Theoretically, an artificial neural controller should be able to learn to control a complex nonlinear system autonomously. A demonstration of this capacity might be considered as evidence for artificial intelligence. This research was not designed to demonstrate autonomous learning. It was begun under the naive assumption that the artificial controller would learn autonomously. The results of our experiments seem to indicate that a first order real-time recurrent learning algorithm is inadequate for learning to control complex nonlinear systems like a flight simulator autonomously. We believe that a second order real-time recurrent learning algorithm would be able to learn autonomously and adjusted the first order algorithm based upon this conjecture. The fact that the artificial neural controller began to learn after these adjustments were made seems to support our conjecture. At this time, artificial neural control has no advantage over conventional control or even fuzzy control because a comparable amount of engineering insight and system knowledge is required from the design engineer to make the necessary adjustments. We expect that, eventually, the necessary computer resources will become available to test a second order real-time recurrent learning algorithm.

The results of this research show that the real-time recurrent learning algorithm can be used to train an artificial neural network to control a complex nonlinear system like a flight simulator for a high performance fighter aircraft.

This is not a "toy" simulator. It was used extensively for flight test simulation by NASA Ames-Dryden Flight Research Facility at Edwards Air Force Base in California. The simulation is not constrained in any way. The artificial neural network controls all six degrees of freedom. Turns are not trivial maneuvers. They are fairly difficult and demanding. The artificial neural controller learned that it must bank in the direction of the turn, increase the angle of attack to maintain altitude and apply rudder to minimize sideslip. It flies 2 g coordinated turns about as well as any novice pilot. It flies shallower turns better and slightly steeper turns less well. It will turn either left or right from any given initial heading to any given final heading. It should be possible to train an artificial neural controller to fly any standard flight test maneuver. It appears to be stable and can tolerate mild air turbulence.

A sincere effort was made to preserve the autonomy of the real-time recurrent learning algorithm. The artificial neural controller was provided with all of the system state variables and their derivatives along with a fairly rich reference signal which provided a look ahead capability for the desired heading. No preprocessing was applied to the inputs other than normalization. The real-time recurrent learning algorithm was expected to determine which inputs were relevant and train the artificial neural controller to use or ignore them for each control output by assigning appropriate weights to each input. The real-time recurrent learning algorithm does not employ a "teacher". The artificial neural controller does not mimic any human pilot or even a conventional autopilot. The learning algorithm trains the controller to minimize the difference between the desired and actual state of the aircraft over the entire maneuver. The design engineer influences the learning algorithm by specifying

- the desired value of each state variable at every time step during each maneuver,

- the weights applied to each of the differences between desired and actual state variables,
- the rate at which recent experience is forgotten and
- the learning rate.

Some human intervention in the automatic design process may always be necessary or at least desirable if the objective is to produce the best possible artificial neural controller in the least amount of time.

- Some sort of "bootstrapping" may be necessary. Unless the artificial neural controller is initialized in such a way as to produce reasonable control outputs, the simulation may never run long enough to allow the artificial neural controller to begin learning.
- It may not be practical to have the artificial neural controller compute all required nonlinear functions even if the learning algorithm could train it to produce them. Some (nonlinear) preprocessing of the inputs may help to speed up the learning process and improve control over the system.
- It will always be necessary for the controller design engineer to specify the performance function. It is hoped but there is no guarantee that any less engineering insight or knowledge of the system will be required to do this correctly.

The artificial neural controller would not learn when this research began. Hundreds of tedious experiments were conducted to diagnose problems. There are still no solutions for some of those problems.

- We were not able to train a two-layer artificial neural network to fly turns starting from small random biases and connection weights. When networks are initialized this way, they learn very slowly until the weights get larger. Perhaps we were simply too impatient to wait long enough for the network to begin learning. We tried initializing the network with larger biases and connection weights but this only caused the simulator to crash more quickly. We trained a one-layer artificial neural network instead and embedded it into a two-layer network in order to learn nonlinear control.

- The artificial neural controller never learned to control the throttle. This didn't seriously affect the ability of the artificial neural controller to learn to fly turns. Human pilots don't usually use the throttle when they first begin to learn to fly turns. There appears to be a fundamental problem involving the relationship between altitude and airspeed that the real-time recurrent learning algorithm was not able to resolve. It may be possible to design a maneuver which isolates the dependency of airspeed on power and train the artificial neural controller to use the throttle but this was not essential for this research. We adjusted the artificial neural controller to mimic the conventional controller.

- The two-layer artificial neural network with the embedded one-layer artificial neural controller never learned any significant nonlinear function. This may be partly due to the fact that 2 g turns can be flown by a linear controller don't demand much from a nonlinear controller. The two layer network also had about ten times as many biases and connection weights as the one layer network and required much more time to train. Although the network did seem to be learning slowly it didn't always appear to be improving. Again, the problem may simply have been that we were too impatient to continue training.

- We were never able to train an artificial neural controller which included its own outputs as part of the inputs. Theoretically, it should be possible to train a recurrent network to control a nonlinear system but none of the neural networks that learned to fly turns incorporated any kind of internal state.

- We never explored any method for stabilizing the update rule for the G_k matrices except limiting $\epsilon > 1/128$. This meant that the learning algorithm would forget recent experience after just a few seconds in real time. We don't think this had a serious impact on real-time recurrent learning and the learning algorithm must forget eventually but we were never able to test these assumptions.

6.3 Natural Neural Controllers

APPENDIX A

Second Order Real-Time Recurrent Learning

This appendix shows how to calculate the total second partial derivatives of the loss function J_{t+1} with respect to the network biases and connection weights at each time step so that they can be averaged to estimate the Hessian matrix which is required to implement a second order real-time recurrent learning algorithm. The objects are third order tensors and some of the notation used may be ambiguous taken out of context.

A.1 Derivative Arithmetic

The first step is to extend derivative arithmetic to compute and propagate the second derivatives of each variable with respect to the state and control variables so that the simulation would compute

$$\left[\begin{array}{c|c|c|c} x & A & B & M \end{array}\right] \leftarrow f\left(\left[\begin{array}{c|c|c} x & I & 0 \\ u & & \end{array}\right]\right) \tag{A.1}$$

where

$$M = \left[\begin{array}{cc} M_{xx} & M_{xu} \\ M_{ux} & M_{uu} \end{array}\right] = \left[\begin{array}{cc} \frac{\partial^2 x_{t+1}}{\partial x_t \partial x_t^T} & \frac{\partial^2 x_{t+1}}{\partial x_t \partial u_t^T} \\ \frac{\partial^2 x_{t+1}}{\partial u_t \partial x_t^T} & \frac{\partial^2 x_{t+1}}{\partial u_t \partial u_t^T} \end{array}\right] = \left[\begin{array}{cc} \frac{\partial}{\partial x_t} A & \frac{\partial}{\partial x_t} B \\ \frac{\partial}{\partial u_t} A & \frac{\partial}{\partial u_t} B \end{array}\right] \tag{A.2}$$

at each time step.

A.2 Real-Time Recurrent Learning

A second order real-time recurrent learning algorithm must compute the total second partial derivatives of the loss with respect to the network biases and connection weights

$$\begin{aligned}
\frac{\bar{\partial}^2 J_{t+1}}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} &= \frac{\bar{\partial} x_{t+1}^T}{\bar{\partial}\theta_t} \cdot \frac{\partial^2 J_{t+1}}{\partial x_{t+1} \partial x_{t+1}^T} \cdot \frac{\bar{\partial} x_{t+1}}{\bar{\partial}\theta_t^T} + \frac{\partial J_{t+1}}{\partial x_{t+1}^T} \cdot \frac{\bar{\partial}^2 x_{t+1}}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} \\
&= \frac{\bar{\partial} x_{t+1}^T}{\bar{\partial}\theta_t} \cdot W_{t+1} \cdot \frac{\bar{\partial} x_{t+1}}{\bar{\partial}\theta_t^T} - \left(d_{t+1} - x_{t+1}\right)^T W_{t+1} \cdot \frac{\bar{\partial}^2 x_{t+1}}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} \qquad \text{(A.3)}
\end{aligned}$$

which requires an estimate of the total second partial derivatives of the state variables with respect to the network biases and connection weights

$$\begin{aligned}
\frac{\bar{\partial}^2 x_{t+1}}{\bar{\partial}\theta_{t+1} \bar{\partial}\theta_{t+1}^T} \approx \frac{\bar{\partial}^2 x_{t+1}}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} &= \frac{\bar{\partial}}{\bar{\partial}\theta_t} \left(\frac{\partial x_{t+1}}{\partial (x_t^T, u_t^T)} \cdot \left(\frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial}\theta_t} \right)^T \right) \\
&= \frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial}\theta_t} \cdot \frac{\partial^2 x_{t+1}}{\partial (x_t^T, u_t^T)^T \partial (x_t^T, u_t^T)} \cdot \left(\frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial}\theta_t} \right)^T \\
&+ \frac{\partial x_{t+1}}{\partial (x_t^T, u_t^T)} \cdot \frac{\bar{\partial}^2 \left(x_t^T, u_t^T\right)^T}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} \\
&= \frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial}\theta_t} \cdot M \cdot \left(\frac{\bar{\partial}(x_t^T, u_t^T)}{\bar{\partial}\theta_t} \right)^T \\
&+ A \cdot \frac{\bar{\partial}^2 x_t}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} + B \cdot \frac{\bar{\partial}^2 u_t}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} \qquad \text{(A.4)}
\end{aligned}$$

which requires, in turn, the total second partial derivatives of the control variables with respect to the network biases and connection weights

$$\begin{aligned}
\frac{\bar{\partial}^2 u_t}{\bar{\partial}\theta_t \bar{\partial}\theta_t^T} &= \frac{\bar{\partial}}{\bar{\partial}\theta_t} \left(\frac{\partial u_t}{\partial (x_t^T, \theta_t^T)} \cdot \begin{bmatrix} \frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T} \\ I \end{bmatrix} \right) \\
&= \begin{bmatrix} \frac{\bar{\partial} x_t^T}{\bar{\partial}\theta_t} \,\Big|\, I \end{bmatrix} \cdot \frac{\partial^2 u_t}{\partial (x_t^T, \theta_t^T)^T \partial (x_t^T, \theta_t^T)} \cdot \begin{bmatrix} \frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T} \\ I \end{bmatrix} \\
&+ \begin{bmatrix} \frac{\bar{\partial} x_t^T}{\bar{\partial}\theta_t} \,\Big|\, I \end{bmatrix} \cdot \frac{\partial u_t}{\partial (x_t^T, \theta_t^T)} \cdot \frac{\partial}{\partial (x_t^T, \theta_t^T)^T} \begin{bmatrix} \frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T} \\ I \end{bmatrix} \\
&= \begin{bmatrix} \frac{\bar{\partial} x_t^T}{\bar{\partial}\theta_t} \,\Big|\, I \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial^2 u_t}{\partial x_t \partial x_t^T} & \frac{\partial^2 u_t}{\partial x_t \partial \theta_t^T} \\ \frac{\partial^2 u_t}{\partial \theta_t \partial x_t^T} & \frac{\partial^2 u_t}{\partial \theta_t \partial \theta_t^T} \end{bmatrix} \cdot \begin{bmatrix} \frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T} \\ I \end{bmatrix} + 0
\end{aligned}$$

$$
\begin{aligned}
&= \frac{\bar{\partial} x_t^T}{\bar{\partial}\theta_t}\cdot\frac{\partial^2 u_t}{\partial x_t \partial x_t^T}\cdot\frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T}+\frac{\bar{\partial} x_t^T}{\bar{\partial}\theta_t}\cdot\frac{\partial^2 u_t}{\partial x_t\partial\theta_t^T}\\
&+ \frac{\partial^2 u_t}{\partial\theta_t\partial x_t^T}\cdot\frac{\bar{\partial} x_t}{\bar{\partial}\theta_t^T}+\frac{\partial^2 u_t}{\partial\theta_t\partial\theta_t^T} \qquad \text{(A.5)}
\end{aligned}
$$

which requires the second partial derivatives of the control variables with respect to the network inputs, biases and connection weights.

A.3 Backprop

The backward error propagation algorithm may be used to obtain the second partial derivatives of the control variables, $u_t = v_K$, with respect to the network inputs, biases and connection weights. The calculation is decomposed into separate blocks for each combination of layer ℓ and layer k at time t.

The second partial derivatives of the control variables with respect to the network inputs

$$
\begin{aligned}
\frac{\partial^2 v_K}{\partial v_\ell \partial v_k^T} &= \frac{\partial a_{\ell+1}^T}{v_\ell}\cdot\frac{\partial v_{\ell+1}^T}{a_{\ell+1}}\cdot\frac{\partial^2 v_K}{\partial v_{\ell+1}\partial v_k^T}\\
&= W_\ell^T\cdot\operatorname{diag}\{v'_{\ell+1,j}\}\cdot\frac{\partial^2 v_K}{\partial v_{\ell+1}\partial v_k^T} \qquad \text{(A.6)}
\end{aligned}
$$

where $\ell < k$ and

$$
\begin{aligned}
\frac{\partial^2 v_K}{\partial v_k \partial v_k^T} &= \frac{\partial}{\partial v_k}\left(\frac{\partial v_K}{\partial v_{k+1}^T}\cdot\operatorname{diag}\{v'_{k+1,j}\}\cdot W_k\right)\\
&= W_k^T\cdot\operatorname{diag}\{v'_{k+1,j}\}\cdot\frac{\partial^2 v_K}{\partial v_{k+1}\partial v_{k+1}^T}\cdot\operatorname{diag}\{v'_{k+1,j}\}\cdot W_k\\
&+ \frac{\partial v_K}{\partial v_{k+1}^T}\cdot\left(W_k^T\cdot\operatorname{diag}\{v''_{k+1,j}\}\cdot W_k\right) \qquad \text{(A.7)}
\end{aligned}
$$

where $\operatorname{diag}\{v''_{k+1,j}\}$ is a diagonal third order tensor are computed first for layers $K-1$ down to 1. The second partial derivatives of the control variables with respect to the state variables

$$
\frac{\partial^2 v_K}{\partial x \partial x^T} = X^T\cdot\operatorname{diag}\{v'_{1,j}\}\cdot\frac{\partial^2 v_K}{\partial v_1\partial v_1^T}\cdot\operatorname{diag}\{v'_{1,j}\}\cdot X
$$

$$+ \quad \frac{\partial v_K}{\partial v_1^T} \cdot \left(X^T \cdot \mathrm{diag}\{v''_{1,j}\} \cdot X \right) \tag{A.8}$$

are then available. If the controller outputs were nonlinear functions of the output activations, backprop would begin with

$$\frac{\partial v_K}{\partial v_K^T} = I \tag{A.9}$$

and

$$\frac{\partial^2 v_K}{\partial v_k \partial v_K^T} = 0 \tag{A.10}$$

but, in this case, there are no output units so backprop begins with

$$\frac{\partial v_K}{\partial v_{K-1}^T} = W_{K-1} \tag{A.11}$$

and

$$\frac{\partial^2 v_K}{\partial v_k \partial v_{K-1}^T} = 0. \tag{A.12}$$

Next, we calculate the

$$\begin{aligned} \frac{\partial^2 v_K}{\partial \vartheta_\ell \partial v_k^T} &= \frac{\partial a_{\ell+1}^T}{\vartheta_\ell} \cdot \frac{\partial v_{\ell+1}^T}{a_{\ell+1}} \cdot \frac{\partial^2 v_K}{\partial v_{\ell+1} \partial v_k^T} \\ &= \begin{bmatrix} 1 \\ v_\ell \end{bmatrix} \otimes \mathrm{diag}\{v'_{\ell+1,j}\} \cdot \frac{\partial^2 v_K}{\partial v_{\ell+1} \partial v_k^T} \end{aligned} \tag{A.13}$$

to get the mixed second partial derivatives of the control variables with respect to the state variables and network biases and connection weights

$$\frac{\partial^2 v_K}{\partial x \partial \vartheta_k^T} = X^T \cdot \mathrm{diag}\{v'_{1,j}\} \cdot \frac{\partial^2 v_K}{\partial v_1 \partial \vartheta_k^T} \tag{A.14}$$

and the second partial derivatives of the control variables with respect to the network biases and connection weights

$$\frac{\partial^2 v_K}{\partial \vartheta_\ell \partial \vartheta_k^T} = \begin{bmatrix} 1 \\ v_\ell \end{bmatrix} \otimes \mathrm{diag}\{v'_{\ell+1,j}\} \cdot \frac{\partial^2 v_K}{\partial v_{\ell+1} \partial \vartheta_k^T}. \tag{A.15}$$

If there are nonlinear output units, backprop begins with

$$\frac{\partial^2 v_K}{\partial v_{K-1} \partial \vartheta_{K-1}^T} = W_{K-1}^T \cdot \mathrm{diag}\{v_{K,j}''\} \otimes \left[\ 1 \mid v_{K-1}^T\ \right] \tag{A.16}$$

and

$$\frac{\partial^2 v_K}{\partial \vartheta_{K-1} \partial \vartheta_{K-1}^T} = \begin{bmatrix} 1 \\ v_{K-1} \end{bmatrix} \otimes \mathrm{diag}\{v_{K,j}''\} \otimes \left[\ 1 \mid v_{K-1}^T\ \right] \tag{A.17}$$

otherwise,

$$\frac{\partial^2 v_K}{\partial v_{K-1} \partial \vartheta_{K-1}^T} = 0, \tag{A.18}$$

$$\frac{\partial^2 v_K}{\partial v_{K-1} \partial \vartheta_{K-2}^T} = 0, \tag{A.19}$$

$$\frac{\partial^2 v_K}{\partial \vartheta_{K-1} \partial \vartheta_{K-1}^T} = 0, \tag{A.20}$$

backprop begins with

$$\frac{\partial^2 v_K}{\partial v_{K-2} \partial \vartheta_{K-2}^T} = W_{K-1}^T \cdot W_{K-1} \cdot \mathrm{diag}\{v_{K-1,j}''\} \otimes \left[\ 1 \mid v_{K-2}^T\ \right] \tag{A.21}$$

and

$$\frac{\partial^2 v_K}{\partial \vartheta_{K-2} \partial \vartheta_{K-2}^T} = \begin{bmatrix} 1 \\ v_{K-2} \end{bmatrix} \otimes \left(W_{K-1} \cdot \mathrm{diag}\{v_{K-1,j}''\} \right) \otimes \left[\ 1 \mid v_{K-2}^T\ \right]. \tag{A.22}$$

For a two layer network with no output units

$$\frac{\partial^2 v_2}{\partial x \partial x^T} = X^T \cdot \left(W_1 \cdot \mathrm{diag}\{v_{1,j}''\} \right) \cdot X, \tag{A.23}$$

$$\frac{\partial^2 v_2}{\partial x \partial \vartheta_0^T} = X^T \cdot \left(W_1 \cdot \mathrm{diag}\{v_{1,j}''\} \right) \otimes \left[\ 1 \mid v_0^T\ \right] \tag{A.24}$$

and

$$\frac{\partial^2 v_2}{\partial \vartheta_0 \partial \vartheta_0^T} = \begin{bmatrix} 1 \\ v_0 \end{bmatrix} \otimes \left(W_1 \cdot \mathrm{diag}\{v_{1,j}''\} \right) \otimes \left[\ 1 \mid v_0^T\ \right]. \tag{A.25}$$

[Bar90] Andrew G. Barto. "Connectionist Learning for Control: An Overview." In Thomas W. Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 1, pp. 5–58. The MIT Press, Cambridge, Massachusettes, 1990.

[BRK87] J. H. Bleher, S. M. Rump, U. Kulisch, M. Metzger, Ch. Ulrich, and W. Walter. "FORTRAN-SC: A Study of a FORTRAN Extension for Engineering and Scientific Computation with Access to ACRITH." *Computing*, **39**(93):93–110, 1987.

[Cra76] Fred D. Crary. "The AUGMENT Precompiler: I. User Information." Technical Summary Report #1469, Mathematics Research Center, University of Wisconsin-Madison, Wisconsin, April 1976.

[Die78] James E. Dieudonne. "Description of a Computer Program and Numerical Technique for Developing Linear Perturbation Models From Nonlinear Systems Simulations." NASA Technical Manual 78710, NASA, 1978.

[DJR86] Eugene L. Duke, Frank P. Jones, and Ralph B. Roncoli. "Development and Flight Test of an Experimental Maneuver Autopilot for a Highly Maneuverable Aircraft." NASA Technical Paper 2618, NASA Ames Research Center Dryden Flight Research Facility, Edwards, California, September 1986.

[FGM92] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. "A Fortran-to-C Converter." Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, New Jersey, May 1992.

[FPD91] L. A. Feldkamp, G. V. Puskorius, L. I. Davis, Jr., and F. Yuan. "Decoupled Kalman Training of Neural and Fuzzy Controllers for Automotive Systems." In *Proceedings of the Fuzzy and Neural Systems and Vehicle Applications '91 Conference*, Tokyo, Japan, November 1991.

[Gra81] Alexander Graham. *Kronecker Products and Matrix Calculus with Applications.* Ellis Horwood Series in Mathematics and Its Applications. Ellis Horwood Limited, Chichester, England, 1981.

[JS90] Charles C. Jorgensen and C. Schley. "A Neural Network Baseline Problem for Control of Aircraft Flare and Touchdown." In Thomas W.

Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 17, pp. 401–425. The MIT Press, Cambridge, Massachusettes, 1990.

[Lew92] Frank L. Lewis. *Applied Optimal Control & Estimation: Digital Design & Implementation.* Prentice Hall and Texas Instruments Digital Signal Processing Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1992.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, New York, New York, 1990.

[NP] Kumpati S. Narendra and Kannan Parthasarathy. "Identificaton and Control of Dynamical Systems Using Neural Networks." unpublished draft copy.

[NP90] Kumpati S. Narendra and Kannan Parthasarathy. "Identificaton and Control of Dynamical Systems Using Neural Networks." *IEEE Transactions Neural Networks*, **1**:4–27, March 1990.

[NW90] Derrick H. Nguyen and Bernard Widrow. "Neural Networks for Self-Learning Control Systems." *IEEE Control Systems Magazine*, pp. 18–23, April 1990.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1984.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning internal representations by error propagation." In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 1*, pp. 318–362. MIT Press, Cambridge, Massachusetts, 1986.

[Wer90] Paul J. Werbos. "Overview of Designs and Capabilities." In Thomas W. Miller, Richard S. Sutton, and Paul J. Werbos, editors, *Neural Networks for Control*, chapter 2, pp. 401–425. The MIT Press, Cambridge, Massachusettes, 1990.

[WZ92] Ronald J. Williams and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." In V. Rao Vemuri, editor, *Artificial Neural Networks: Concepts and Control Applications*, pp. 300–308. IEEE Computer Society Press, Los Alamitos, California, 1992.