Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

5-1997

Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method

Christian Peter Paintz
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Paintz, Christian Peter, "Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method" (1997). *Dissertations and Theses*. Paper 5792.

<https://doi.org/10.15760/etd.7663>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of Christian Peter Paintz for the Master of Science in Electrical and Computer Engineering were presented May 21, 1997, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

George G. Lendaris

W. Robert Daäsch

Gerardo Lafferriere
Representative of the Office of
Graduate Studies

DEPARTMENT APPROVAL:

Rolf Schaumann, Chair
Department of Electrical Engineering

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by

on

Aug. 29, 1997

Abstract

An abstract of the thesis of Christian Peter Paintz for the Master of Science in Electrical and Computer Engineering presented May 21, 1997

Title: Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method

This thesis discusses strategies for and details of training procedures for the Dual Heuristic Programming (DHP) methodology. This and other approximate dynamic programming approaches (HDP, DHP, GDHP) have been discussed in some detail in the literature, all being members of the Adaptive Critic Design (ACD) family. The example applications used here are the inverted pendulum problem and a fully nonlinear constant velocity bicycle steering model. The inverted pendulum has been successfully controlled using DHP, as reported in the literature. This thesis suggests and investigates several alternative DHP training procedures and compares their performance with respect to convergence speed and quality of resulting controller design. A promising modification is to introduce a real copy of the *criticNN* (*criticNN#2*) for making the “desired output” calculations, and very importantly, this *criticNN#2* is trained differently than is *criticNN#1*. The idea is to provide the “desired outputs” from a stable platform during an epoch while adapting the *criticNN#1*. Then at the end of the epoch, *criticNN#2* is made identical to the then-current adapted state of *criticNN#1*, and a new epoch starts. In this way, both the *criticNN#1* and the *actionNN* can be simultaneously trained on-line during each epoch, with a faster overall

convergence than the older approach. Further, the measures used herein suggest that a “better” controller design (the *actionNN*) results.

The learning strategy with the fastest learning was used to design a controller for a fully nonlinear, constant-velocity bicycle steering model. The controller's task here is to steer the car along a given trajectory on the road. The performance accomplished by the controller demonstrates the applicability of that learning strategy to highly nonlinear, complex plants.

TRAINING STRATEGIES FOR CRITIC AND ACTION
NEURAL NETWORKS
IN DUAL HEURISTIC PROGRAMMING METHOD

by

CHRISTIAN PETER PAINTZ

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University

1997

ACKNOWLEDGMENTS

The Author is sincerely grateful to Dr. George G. Lendaris, a trusted mentor over the last 2 years. The research reported in this thesis started out of his many excellent lectures on neural network technology and he devoted a great amount of energy over the last one and a half years to very helpful discussions in "Tuesday Night Meetings", which inspired this author's intellectual curiosity to reveal many otherwise obscure features of the problem. Through his good contacts with Accurate Automation Corporation he also provided the C code which was crucial for the implementation of the Nonlinear Steering Model.

Many other problems were solved by combining the ideas of the author with those of various people. The author is grateful for all the contributions involved.

This thesis is dedicated to Margot and Manfred, the author's parents.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGMENTS	II
LIST OF TABLES	VI
LIST OF FIGURES.....	VII
1. INTRODUCTION.....	1
1.1. History of Neural Computing.....	1
1.2. The Back-Propagation Paradigm.....	3
1.2.1. <i>Description of a Feedforward Network</i>	3
1.2.2. <i>Back-Propagating the Local Error</i>	5
1.2.3. <i>Minimizing the Global Error</i>	6
1.2.4. <i>The Global Error Function</i>	7
1.2.5. <i>Momentum Term</i>	8
1.2.6. <i>Summary of the Standard Back-Propagation Algorithm</i>	9
2. ADAPTIVE CRITIC DESIGNS (ACD)	10
2.1. Adaptive Critic Designs at the Autolanding Task.....	12
3. DUAL HEURISTIC PROGRAMMING (DHP)	14
3.1. Introduction.....	14
3.2. Equations to Update the Neural Nets.....	16

3.2.1. Equations to Update the ActionNN Weights.....	16
3.2.2. Desired Output for the CriticNN.....	18
3.3. Strategies for Training Procedures.....	19
3.3.1. The Convergence Process.....	19
3.3.2. Strategies for Training Procedures.....	20
3.3.3. Algorithms for the Above Strategies.....	21
4. EXPERIMENTAL RESULTS.....	27
4.1. The Inverted Pendulum Problem	27
4.1.1. Preliminary Comments.....	29
4.1.2. Performance.....	30
4.1.3. Bias Term.....	31
4.1.4. Convergence Speed.....	32
4.1.5. Quality of Resulting Controller Design.....	34
4.2. Steering Control of a 4-Wheel Vehicle.....	34
4.2.1. Bicycle Model.....	35
4.2.2. Implementing DHP/Strategy 4a for the Bicycle Steering Model.....	42
4.2.2.1. Preliminary Comments.....	
4.2.2.2. Preconditioning the ActionNN.....	
4.2.2.3. Dividing the State Space into Units of Training Time.....	
4.2.2.4. Selection Module.....	
4.2.2.5. Performance.....	

5. CONCLUSION.....	48
5.1. Future Research.....	48
5.2. Synopsis of Contribution.....	54
REFERENCES.....	55
APPENDIX A.....	57
APPENDIX B.....	65
Pascal Code for Inverted Pendulum simulation, using strategy 4a.	65

LIST OF TABLES

TABLE

I	Percentage of successful landings for additional wind gusts $N(0,1)$	13
II	Neural net structures for criticNN and actionNN	28
III	Learning Coefficients and Epoch Size	30
IV	Number of elements and transfer functions for actionNN and criticNN	42

LIST OF FIGURES

FIGURE

1.	Typical feed forward network layout	3
2.	A typical neural network processing element with hyperbolic tangent transfer function	4
3.	Schematic Diagram of important components of the DHP process.	15
4.	Schematic of pole balancer	26
5.	Accumulated cost during first pass through train data for the different strategies.	31
6.	Accumulated cost during second and third passes through train data for the different strategies.	32
7.	Accumulated cost $\{C(4), C(5) \text{ and } C(6)\}$ for test and generalization runs for the different strategies	33
8.	Bicycle Model Geometry	35
9.	Block Diagram for Bicycle Steering Model	41
10.	Preconditioned ActionNN	43
11.	Trajectory of Car on Road (friction coefficient = 1)	46
12.	Pole angles while strategy 4a adapts $l=1\text{m}$ controller design to $l=2.4\text{m}$ during first 30 sec. of adaptation. (5° deflection) m	49
13.	Trajectory of Car on Road (friction coefficient = 0.85)	50
14.	Trajectory of Car on Road (friction coefficient = 0.7)	51

A.1.	Algorithm 1	56
A.2.	Algorithm 2a	57
A.3.	Algorithm 2b	58
A.4.	Algorithm 3a	59
A.5.	Algorithm 4a	60
A.6.	Algorithm 4b	61
A.7.	Pole angle after -20° disturbance for pole length 1m and 2.4m	62
A.8.	Force applied by controller after -20° disturbance for pole length 1m and 2.4m	63

CHAPTER 1

INTRODUCTION

1.1 History of Neural Computing [12]

The human brain is a very complex computing device. The brain's thinking, remembering, and problem-solving capabilities have inspired many scientists to attempt computer modeling of its operation. One group of researchers has sought to create a mathematical model that matches the functionality of the brain in a very fundamental manner; the result has been neural computing.

The neuron is the fundamental "computing" unit of the nervous system and, in particular, the brain. Each neuron is a simple unit which receives and combines signals from many other neurons through input structures called dendrites. If the combined signal is strong enough, it activates the firing of the neuron, which produces an output signal; the path of the output signal is along a component of a cell called the axon. This simple transfer of information is chemical in nature, but it has electrical side effects which we can measure.

The brain consists of tens of billions of neurons densely interconnected. The axon (output path) of a neuron splits up and connects to dendrites (input paths) of other neurons through a junction referred to as a synapse. The transmission across this junction is chemical in nature and the amount of signal transferred depends on the amount of chemicals (neurotransmitters) released by the axon and received by the dendrites. This synaptic efficiency (or "strength") is what is modified when the brain

learns. The synapse combined with the processing of information in the neuron form the basic memory mechanism of the brain.

In 1943, a neurobiologist, Warren McCulloch, and a statistician, Walter Pitts, published a watershed paper titled "A Logical Calculus of Ideas Imminent in Nervous Activity." This paper was an inspiration which helped to launch many diverse fields. Marvin Minsky, one of the most prominent researchers in Artificial Intelligence, became enthralled with the idea of macroscopic intelligence from this paper. Later, this led to his interest in black-box macroscopic intelligence, the birthplace of expert systems. Frank Rosenblatt became intrigued with computations of the eye. This led to his invention of the perceptron. The perceptron is a pattern classification system which could identify both abstract and geometric patterns. A major limitation was the inability of the perceptron to represent the basic Exclusive OR (XOR) function.

In 1959, Bernard Widrow developed an adaptive linear element, called "adaline" (Adaptive Linear Neuron), based on simple neuron-like elements. The Adaline and a two-layer variant, the madaline were used for a variety of applications including speech recognition, character recognition, weather prediction, and adaptive control. Later the adaline was modified to produce a continuous rather than discrete output.

Paul Werbos invented the back-propagation methodology, a powerful paradigm for supervised learning.

1.2 The Back-Propagation Paradigm

1.2.1 DESCRIPTION OF A FEEDFORWARD NETWORK

The Back Propagation Paradigm typically involves a feed-forward network, which always has an input layer, an output layer and at least one hidden layer. There is no theoretical limit on the number of hidden layers but typically there will be one or two. Some work has been done which indicates that a maximum of four layers (three hidden plus one output layer) are required to solve arbitrarily complex pattern classification problems. Each layer is fully connected to the succeeding layer. During normal operation, information flows in the "forward" direction only indicated by the arrows in Fig.(1). During learning, information is also propagated back through the network and used to update the connection weights, see Fig. (2).

Figure 1: Typical feed forward network layout

To distinguish the elements from one layer to another, a clear notation is needed for describing the learning rule. A superscript in square brackets is used herein to indicate which layer of the network is being considered. The rest of the notation is as follows:

$x_j^{[s]}$ current output state of jth neuron in layer s

$w_{ji}^{[s]}$ weight on connection joining ith neuron in layer (s-1) to jth neuron in layer s

$I_j^{[s]}$ weighted summation of inputs to jth neuron in layer s

A back-propagation element therefore transfers its inputs as follows:

$$x_j^{[s]} = f\{\sum_i (w_{ji}^{[s]} * x_i^{[s-1]})\} = f\{I_j^{[s]}\} \quad (1)$$

where f is traditionally the sigmoid or hyperbolic tangent function but can be any differentiable function.

Figure 2: A typical neural network processing element with hyperbolic tangent transfer function.

The sigmoid function is defined as

$\text{sigmoid}(z) = (1.0 + e^{-z})^{-1}$
and the hyperbolic tangent is defined as

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

1.2.2 Back-Propagating the Local Error

A global error function E is defined for the network, with the requirement that it be a differentiable function of all the connection weights in the network, and typically involves the desired output and the actual output. The actual error function is unimportant for understanding the mechanism of back-propagation. The error value that is passed back through the layers is defined by

$$e_j^{[s]} = - \partial E / \partial I_j^{[s]} \quad (2)$$

We will see later that this can be considered a measure of the local error at processing element j in level s . Using the chain rule of differentiation twice in succession gives a relationship between the local error at a particular processing element at level s and all the local errors at the level $s+1$:

$$e_j^{[s]} = f' (I_j^{[s]}) * \sum_k (w_{kj}^{[s+1]} * e_k^{[s+1]}) \quad (3)$$

Note that in Eqn. (3), there is a layer after layer s ; therefore, Eqn. (3) can only be used for non-output layers.

If f is the sigmoid function, then its derivative can be expressed as a simple function of itself as follows:

$$f'(z) = (1.0 - f(z)) * f(z)$$

As with the sigmoid function, the derivative of hyperbolic tangent can also be expressed in terms of itself:

$$f'(z) = (1.0 - f(z)) * (1.0 + f(z))$$

With the hyperbolic tangent transfer function, the error propagation Eqn. (3) is modified to

$$e_j^{[s]} = (1 + x_j^{[s]}) * (1 - x_j^{[s]}) * \sum_k (w_{kj}^{[s+1]} * e_k^{[s+1]}) \quad (4)$$

The summation term in Eqns. (3) and (4) which is used to back-propagate errors is analogous to the summation term in Eqn. (1) which is used to forward propagate the input through the network. The main mechanism in a back-propagation network is to forward propagate the input through the layers to the output layer, determine the error at the output layer, and then propagate the errors back through the network from the output layer to the input layer using Eqn. (4) or more generally Eqn. (3).

1.2.3 Minimizing the Global Error

The objective of the learning process is to minimize the global error E of the system by modifying the weights. This section will show how to do this based on knowledge of the local error at each processing element.

Given the current set of weights $w_{ji}^{[s]}$, we need to determine how to increment or decrement each weight in order to decrease the global error. This can be done using a gradient descent rule as follows:

$$\Delta w_{ji}^{[s]} = -lcoef * (\partial E / \partial w_{ji}^{[s]}) \quad (5)$$

where lcoef is a learning coefficient. In other words, change each weight according to the size and direction of negative gradient on the error surface.

The partial derivatives in Eqn. (5) can be calculated directly from the local error values discussed in section 1.2.1, because by the chain rule and Eqn. (1):

$$\partial E / \partial w_{ji}^{[s]} = (\partial E / \partial I_j^{[s]}) * (\partial I_j^{[s]} / \partial w_{ji}^{[s]})$$

$$\partial E / \partial w_{ji}^{[s]} = -e_j^{[s]} * x_i^{[s-1]} \quad (6)$$

Combining Eqns. (5) and (6) together gives

$$\Delta w_{ji}^{[s]} = lcoef * e_j^{[s]} * x_i^{[s-1]} \quad (7)$$

1.2.4 The Global Error Function

The above discussion has assumed the existence of a global error function without specifying it. This function defines the local errors at the output layer so that they can be propagated back through the network. Suppose a vector I is presented at the input layer of the network and suppose the desired output $\underline{d}$ is specified by a teacher. Let $\underline{o}$ denote the actual output produced by the network with its current set of weights. Then one possible measure of the error in achieving that desired output is given by

$$E = 0.5 * \sum_k \{(d_k - o_k)^2\} \quad (8)$$

where the subscript k indexes the components of $\underline{d}$ and $\underline{o}$. Here, the raw local error is given by $d_k - o_k$, and the 0.5 scale factor is included in anticipation of the differentiation

step to be taken. From Eqn. (2), the "local error" at each processing element of the output layer is given by

$$\begin{aligned}
 e_k^{[0]} &= -\partial E / \partial I_k^{[0]} \\
 &= -(\partial E / \partial o_k) * (\partial o_k / \partial I_k) \\
 &= (d_k - o_k) * f'(I_k) \quad (9)
 \end{aligned}$$

The error E as defined in Eqn. (8) defines the global error of the network for a *particular* $(\underline{i}, \underline{d})$'s. An overall global error function can be defined by summing the errors over a collection of $(\underline{i}, \underline{d})$'s. In the former case, each time a particular $(\underline{i}, \underline{d})$ is shown, the back-propagation algorithm modifies the weights to reduce that particular component of the overall error function.

1.2.5 Momentum Term

One of the problems of gradient descent algorithms is setting an appropriate learning rate. Changing the weights as a linear function of the partial derivative as defined in Eqn. (5) makes the assumption that the error surface is locally linear, where "locally" is defined by the size of the learning coefficient. At points of high curvature, this linearity assumption does not hold and divergent behavior might occur near such points. It is therefore important to keep the learning coefficient low to avoid such behavior.

On the other hand, a small learning coefficient can lead to very slow learning. The concept of "momentum term" was introduced to help alleviate this problem. The delta weight Eqn. (7) is modified so that a portion of the previous delta weight is fed

through to the current delta weight:

$$\Delta w_{ji}^{[s]} = \text{lcoef} * e_j^{[s]} * x_i^{[s-1]} + \text{momentum} * \Delta w_{ji}^{[s]} \quad (10)$$

This acts as a low-pass filter on the delta weight terms since general trends are reinforced whereas oscillatory behavior cancels itself out. This allows a low learning coefficient but fast(er) learning.

1.2.6 Summary of the Standard Back-Propagation Algorithm

Given an input vector $\underline{i}$ and a desired output vector $\underline{d}$, do the following.

1. Present $\underline{i}$ to the input layer of the network and process it through to the output layer to obtain an output vector $\underline{o}$.
2. As this information propagates through the network, it will also set all the summed inputs i_j and output states x_j for each processing element in the network.
3. For each processing element in the output layer, calculate the local error as given in Eqn. (9) and then calculate the delta weight using Eqn. (7).
4. For each layer s , starting at the layer previous to the output layer and ending with the layer just following the input layer, and for each processing element in layer s , calculate the local error as given in Eqn. (4), then calculate the delta weight using Eqn. (7).

Chapter 2

ADAPTIVE CRITIC DESIGNS (ACD)

The simplest adaptive critic designs learn slowly on large problems but have generated many real-world success stories on difficult small problems. Complex adaptive critics may seem intimidating, at first, but according to [Werbos, 1992] they are the only design approach that shows serious promise of duplicating critical aspects of human intelligence.

Adaptive critic designs may be defined as designs that attempt to approximate dynamic programming in the general case. Dynamic programming [14], in turn, is the only exact and efficient method for finding an optimal strategy of action over time in a noisy, nonlinear environment. The cost of running true dynamic programming is proportional (or worse) to the number of possible states in the plant or environment; that number, in turn grows exponentially with the number of variables in the environment. Therefore, approximate methods are needed.

In all forms of dynamic programming, the user supplies a utility function U and a stochastic model of the plant or environment to be controlled. Dynamic programming is used to solve for another function, J , which serves as a secondary or strategic utility function. The key theorem is that any strategy of action that maximizes J in the short term will also maximize the sum of U over all future times. J is a function of $R(t)$, where R is a complete state description of the plant to be controlled at time t . Adaptive

critic designs are defined more precisely as designs that include a Critic network - a network whose output is an approximation to the J function, or to its derivatives, or to something very closely related to these two. The action network in an adaptive critic system is adapted so as to maximize J in the near-term future.

It is convenient to use backpropagation to get necessary derivatives of the error term with respect to training parameters and/or inputs of the network. Here we use back-propagation as a tool of getting required derivatives and also as a complete training algorithm. When the critic network learns, backpropagation of error signals can continue along their input pathway back to the action network. To the backpropagation algorithm, this input pathway just looks like another synaptic connection that needs weight adjustment. Thus, no desired action signal is necessary, nor is a desired trajectory needed to allow simple computation of desired actions.

Adaptive Critic Designs (ACD's) include heuristic dynamic programming (HDP), dual heuristic programming (DHP) and globalized DHP (GDHP) as well as their action dependent forms, further referred to as having prefix AD. All designs attempt to approximate dynamic programming.

A typical ACD consists of three neural nets -the critic, action and model. The critic net outputs a function J which is an approximation of the secondary utility function J (as in HDP, ADHDP and GDHP) or the derivatives of J with respect to the state variables R (as in DHP and ADDHP). This function has to be approximated because of intractable computational complexity to calculate it. The goal is to maximize or minimize J in the immediate future (in the next time step) which produces an

optimum U , the primary utility function in the long run. This goal is accomplished by the action network which outputs a control vector $\underline{u}$ that optimizes J .

An adaptation of the action network is based on derivatives of J with respect to the components of the vector $\underline{u}$. A straightforward way to get those derivatives is to use backpropagation through the other neural networks. The use of the backpropagation algorithm to find the derivatives of J is the key distinction between HDP and the well-known adaptive critic element [1].

2.1 Adaptive Critic Designs for an Autolanding Task

In [7], a problem is defined which involves a simplified model of a commercial aircraft which is to be landed in a specified touchdown region of a runway within given ranges of speed and pitch angle. The aircraft is subject to wind disturbances. The problem was complicated by shortening the touchdown region of the runway. A standard PID controller, backpropagation of utility (BU), ADHDP, HDP and DHP are compared for the same complicated version of the autolander problem. The most important conclusion which can be drawn from that paper is that going from the simplest Adaptive Critic Design: ADHDP, to the more advanced DHP, one can observe a significant improvement in the performance.

Table I shows some extracted representative results.

TABLE I:
PERCENTAGE OF SUCCESSFUL LANDINGS FOR ADDITIONAL WIND
GUSTS $N(0,1)$. [7]

<i>Achieved Performance</i>	<i>DHP</i>	<i>HDP</i>	<i>ADHDP</i>	<i>BU</i>	<i>PID</i>
tight success	72+	37	2	4	0
loose success	8	21.5	98	96	100
near misses	5	21.5	0	0	0

Chapter 3

DUAL HEURISTIC PROGRAMMING (DHP)

3.1 Introduction

This section describes the fundamentals and discusses strategies for and details of training procedures for the Dual Heuristic Programming (DHP) methodology, defined in [6]. The main recent reference on training procedures for ACDs is [2]. In this and the next section the author suggests and investigates several alternative procedures and compares their performance with respect to convergence speed and quality of resulting controller design. A promising modification is to introduce a **real** copy of the *critic*NN (*critic*NN#2) for making the “desired output” calculations, and very importantly, this ***critic*NN#2 is trained differently than is *critic*NN#1**. The idea is to provide the “desired outputs” from a stable platform during an epoch while adapting the *critic*NN#1. Then at the end of the epoch, *critic*NN#2 is made identical to the then-current adapted state of *critic*NN#1, and a new epoch starts. In this way, both the *critic*NN#1 and the *action*NN can be simultaneously trained on-line during each epoch, with a faster overall convergence than the older approach. Further, the measures used herein suggest that a “better” controller design (the *action*NN) results.

The secondary utility function $J(t)$ is defined in Eqn as follows:

$$J(t) = \sum_{k=t}^{\infty} \gamma^k * U(R(k), u(k)) \quad (11).$$

The term γ is a discount factor ($0 < \gamma \leq 1$), and U is the *primary* utility function, which

must be defined by the user for the specific application context.

For all further discussion herein, γ is assumed to be 1. In this case, Eqn. (11) is equivalent to

$$J(t) = U(t) + J(t+1) \quad (12)$$

A schematic diagram of important components of the DHP method is shown in Fig.

(3).

Figure 3: Schematic Diagram of important components of the DHP process.

The equations and techniques described are based on a discrete plant. The Euler Forward Method is used to solve the ordinary differential plant equations for the next time states. For DHP, at least two neural nets are needed, one for the *action*NN functioning as the controller, and one for the *critic*NN used to train the *action*NN. A third NN must be trained to copy the plant if an analytical description (model) of the

plant is not available.

$\mathbf{R}(t)$ [dim: n] is the state of the plant at time t . The control signal $\mathbf{u}(t)$ [dim: a] is generated by the *actionNN* in response to the input $\mathbf{R}(t)$. The signal $\mathbf{u}(t)$ is then asserted to the plant. As a result of this, the plant changes its state to $\mathbf{R}(t+1)$. The neural nets are updated using $\{\mathbf{u}(t), \mathbf{R}(t), \mathbf{R}(t+1)\}$ in the equations described in Section 3.2. The *criticNN* is needed to adapt the *actionNN* to the plant (model) and to the utility function. The (primary) utility function $U(\mathbf{R}(t), \mathbf{u}(t))$ expresses the objective of the control application, a potential example for the inverted pendulum problem being: "balance the pole upright and save energy by keeping the control vector's amplitude small".

It was discovered necessary to insert the $*1/\max_r$ modules to scale the n -dimensional state space R^n to $[-1,+1]^n$.

3.2 Equations to update the Neural Nets

The underpinnings of the DHP method are the equations for training the neural nets.

Therefore, it is important to understand how the two networks are updated.

3.2.1 Equations to Update the ActionNN Weights

The weights in the action network are updated with the objective of maximizing $J(t)$.

The control vector is defined as $\mathbf{u}(t)=[u_1(t), \dots, u_k(t), \dots, u_a(t)]^T$ and the state vector as

$\mathbf{R}(t)=[R_1(t), \dots, R_s(t), \dots, R_n(t)]^T$. The *basic* backpropagation algorithm is used (no

embellishments), wherein the actionNN's weight-adjustment increment is calculated via:

$$\Delta w_{ij}(t) = lcoef * \frac{\partial J(t)}{\partial w_{ij}(t)} \quad (13)$$

where lcoef is the learning coefficient and

$$\frac{\partial J(t)}{\partial w_{ij}(t)} = \frac{\partial (U(t) + J(t+1))}{\partial w_{ij}(t)} = \sum_{k=1}^a \frac{\partial J(t)}{\partial u_k(t)} * \frac{\partial u_k(t)}{\partial w_{ij}(t)} \quad (14)$$

Applying Eqn. (12), we have

$$\frac{\partial J(t)}{\partial u_k(t)} = \frac{\partial U(t)}{\partial u_k(t)} + \frac{\partial J(t+1)}{\partial u_k(t)} \quad (15)$$

where

$\frac{\partial U(t)}{\partial u_k(t)}$ is the partial derivative of the primary utility function $U(t)$ with respect to

the component $u_k(t)$, and

$$\frac{\partial J(t+1)}{\partial u_k(t)} = \sum_{s=1}^n \frac{\partial J(t+1)}{\partial R_s(t+1)} * \frac{\partial R_s(t+1)}{\partial u_k(t)} \quad (16)$$

We abbreviate:

$$\frac{\partial J(t+1)}{\partial R_s(t+1)} = \lambda_s(t+1) \quad (17)$$

The term $\lambda_s(t+1)$ is approximated by the critic, as response to the input $\mathbf{R}(t+1)$.

$\frac{\partial R_s(t+1)}{\partial u_k(t)}$ can be calculated from analytical equations of the plant, if they are available

or by backpropagation through a third neural net which has been trained to

copy the plant.

3.2.2 Desired Output for the CriticNN

To train the critic NN, whose output is λ , a value has to be calculated for the role of "desired output", here called λ^* . Recalling equations (11) and (12), and making use of equation (17), we write

$$\lambda_s^*(t) = \frac{\partial J(t)}{\partial R_s(t)} = \frac{\partial (U(t) + J(t+1))}{\partial R_s(t)}$$

This resolves to

$$\begin{aligned} \lambda_s^*(t) = & \frac{\partial U(t)}{\partial R_s(t)} + \sum_{j=1}^a \left(\frac{\partial U(t)}{\partial u_j(t)} * \frac{\partial u_j(t)}{\partial R_s(t)} \right) + \sum_{k=1}^n \left(\frac{\partial J(t+1)}{\partial R_k(t+1)} * \frac{dR_k(t+1)}{dR_s(t)} \right) \\ & + \sum_{k=1}^n \left[\sum_{j=1}^a \left[\frac{\partial J(t+1)}{\partial R_k(t+1)} * \frac{\partial R_k(t+1)}{\partial u_j(t)} * \frac{\partial u_j(t)}{\partial R_s(t)} \right] \right] \end{aligned} \quad (18)$$

where:

$\frac{\partial U(t)}{\partial R_s(t)}$ is the derivative of the primary utility function $U(t)$ with respect to $R_s(t)$,

$\frac{\partial u_j(t)}{\partial R_s(t)}$ is calculated by backpropagation through the *action* NN,

$\frac{\partial J(t+1)}{\partial R_k(t+1)}$ is approximated by the critic itself as response to $\mathbf{R}(t+1)$, i.e., is $\lambda_k(t+1)$.

For training the *critic* NN, the error components are calculated as follows:

$$E_s = (\lambda_s^*(t) - \lambda_s(t))^2 \quad (19)$$

3.3 Strategies for Training Procedures

This section discusses various procedures for using the neural net update equations given in the previous section.

3.3.1 The Convergence Process

To derive the training strategies, we take a closer look at the convergence process. In the present notation, $\lambda(\mathbf{R})$ is the mapping performed by the *critic*NN, $\lambda^*(\mathbf{R})$ is the desired output for the *critic*NN ["calculated" by using the *critic*NN's output in response to $\mathbf{R}(t+1)$], and $\lambda^\wedge(\mathbf{R})$ is the "solution" (that we don't know) of the Bellman equation and is the target for the other two λ 's. $\lambda^\wedge(\mathbf{R})$ is a function of the state $\mathbf{R}$ and doesn't change for a time invariant plant. Since we update the *critic*NN, $\lambda(\mathbf{R})$ and $\lambda^*(\mathbf{R})$ (which is calculated using the updated *critic*NN) change over time; $\lambda^*(\mathbf{R})$ is supposed to converge to $\lambda^\wedge(\mathbf{R})$, and $\lambda(\mathbf{R})$ is adjusted in order to converge to $\lambda^*(\mathbf{R})$. I.e., $\lambda(\mathbf{R}) \rightarrow \lambda^*(\mathbf{R}) \rightarrow \lambda^\wedge(\mathbf{R})$. One can imagine this as a tracking problem, where λ tracks λ^* . The better the *critic*NN "solves" the Bellman equation [i.e.: $\lambda(\mathbf{R}) \rightarrow \lambda^\wedge(\mathbf{R})$] the better the *action*NN will approximate an optimal controller.

Eqn. (18) for $\lambda^*(t)$ is our principal focus here. As an aside, if we substitute $\lambda(t)$ (i.e., without the superscript) in the left side of Eqn. (18), the *critic*NN is considered 'converged' when this new equation holds true for all s and all subsequent t 's, i.e. when $\lambda(\mathbf{R}) = \lambda^*(\mathbf{R})$ [t can be thought of as an index in the sequence of states].

For convenience of discussion, we paraphrase Eqn. (18) as follows:

$$\begin{aligned}
\lambda^*_s(t) = & \{\sim Utility\} + \sum_{j=1}^a [\{\sim Utility\} \bullet \{\sim Action\}] \\
& + \sum_{k=1}^n [\{\sim Critic(t+1)\} \bullet \{\sim Plant\}] \\
& + \sum_{k=1}^n \left[\sum_{j=1}^a [\{\sim Critic(t+1)\} \bullet \{\sim Plant\} \bullet \{\sim Action\}] \right]
\end{aligned}$$

In a neural network implementation, the $\{\sim Action\}$ terms are calculated via the action

NN, and the $\{\sim Critic(t+1)\}$ terms are calculated via the critic NN.

3.3.2 Strategies for Training Procedures

Strategy 1. "Straight" application of the equation.

Strategy 2. Basic two-stage process: [3]

i) Hold the actionNN parameters constant for a specified number N of computation cycles, while adjusting parameters in the criticNN.

ii) Then, hold the *critic*NN parameters constant for a specified number M of computation cycles, while adjusting the *action*NN parameters. Return to i.

(We here let N=M and use the familiar term "epoch" as a name for this set of cycles)

Strategy 3. Modified two-stage process (as in Strategy 2, however first stage is substantially modified):

i) We noted earlier that the task of the *critic*NN [which performs $\lambda(\mathbf{R})$] is to learn

$\lambda^*(\mathbf{R})$. For the present strategy, we temporarily *suspend* adjustments to the process that calculates $\lambda^*(\mathbf{R})$ to give the process that calculates λ a chance to accomplish its adaptation. To do this, we create a new copy of the *criticNN*; the original is called *criticNN#1* and the copy is called *criticNN#2*. This is different from the "copy" in the cited references; their "copy" is for convenience rather than substance. In the present case, we hold *criticNN#2*'s parameters constant during the epoch; we use this (non-adapted) *criticNN#2* to calculate $\lambda^*(\mathbf{R})$ and use these values of $\lambda^*(\mathbf{R})$ to train *criticNN#1*. As in strategy 2., the *actionNN* parameters are held constant during this epoch. At the end of this epoch, the weights of *criticNN#2* are set equal to those of *criticNN#1*.

ii) Same as for strategy 2. above.

Strategy 4. Single-stage process, based on suspended adaptation of *criticNN#2*.

We maintain the notion of an epoch as in strategies 2. and 3.. However, in this strategy, both the action and the *criticNN#1* are adjusted during each computational cycle. As before, *criticNN#2* is not adapted during the epoch; its weights are set equal to those of *criticNN#1* at the end of the epoch. The next epoch starts the same process over.

3.3.3 Algorithms for the above Strategies

Strategies 1. and 4. are single-stage processes. Strategies 2. and 3. are 2-stage processes. The 2-stage processes are (here) said to comprise a sequence of "flip/flop" epochs. In the "flip" epoch, only the *criticNN* is updated; in the "flop" epoch, only the *actionNN* is updated (on-line mode). The variations in Strategies 2. and 3. occur in the

flip epoch; the flop epoch process is the same for both. The flip and flop epochs are here equal in length.

Flop epoch algorithm (adapts the *action*NN in on-line mode):

1. Scale $\mathbf{R}(t)$ and apply it to the *action*NN, obtain $\mathbf{u}(t)$;
2. Apply $\mathbf{u}(t)$ to the plant and obtain $\mathbf{R}(t+1)$;
3. Scale $\mathbf{R}(t+1)$ and apply it to the *critic*NN (both #1 and #2 are identical during this epoch), obtain $\lambda(t+1)$;
4. Calculate/execute weight changes for *action*NN per Section 3;
5. If $t < \text{epoch}$, increment t and go to 1;
6. Go to flip epoch.

Strategy 1: Single stage, concurrent training of actionNN and criticNN

1. Scale $\mathbf{R}(t)$ and apply it to the *action* NN, obtain $\mathbf{u}(t)$
2. Apply $\mathbf{u}(t)$ to the plant and obtain $\mathbf{R}(t+1)$
3. Scale $\mathbf{R}(t+1)$ and apply it to the *critic* NN, obtain $\lambda(t+1)$
4. Calculate desired output $\lambda^*(t)$ for *critic* NN
5. Calculate *action* NN weight changes
6. Execute *action* NN weight changes
7. Scale $\mathbf{R}(t)$ and apply it to *critic* NN
8. Calculate and execute weight changes on *critic* NN
9. Increment t and go to 1.

Strategy 2a: Basic flip/flop strategy(a): On-line training of criticNN during flip epoch / on-line training of action NN during flop epoch.

Flip epoch algorithm:

1. Scale $R(t)$ and apply it to the *action* NN, obtain $u(t)$
2. Apply $u(t)$ to the plant and obtain $R(t+1)$
3. Scale $R(t+1)$ and apply it to the *critic* NN, obtain $\lambda(t+1)$
4. Calculate desired output $\lambda^*(t)$ for *critic* NN
5. Scale $R(t)$ and apply it to *critic* NN
6. Calculate and execute weight changes on *critic* NN
7. If $t < \text{epoch}$, increment t and go to 1.
8. Go to flop epoch

Strategy 2b: Basic flip/flop strategy(b): batch training of critic NN during flip epoch / on-line training of action NN during flop epoch.

Flip epoch algorithm:

1. Scale $R(t)$ and apply it to the *action* NN, obtain $u(t)$
2. Apply $u(t)$ to the plant and obtain $R(t+1)$
3. Scale $R(t+1)$ and apply it to the *critic* NN, obtain $\lambda(t+1)$
4. Calculate desired output $\lambda^*(t)$ for *critic* NN
5. Scale $R(t)$ and apply it to *critic* NN

6. Calculate and accumulate *critic* NN weight changes in a weight store matrix Δw
7. If $t < \text{epoch}$, increment t and go to 1.
8. Execute weight changes stored in matrix Δw to *critic* NN
9. Go to flop epoch

Strategy 3a. Modified flip/flop strategy (a): On-line train *critic*NN#1 with suspended adaptation of *critic*NN#2.

Flip epoch algorithm:

1. Scale $R(t)$ and apply it to the *action*NN, obtain $u(t)$;
2. Apply $u(t)$ to the plant and obtain $R(t+1)$;
3. Scale $R(t+1)$, apply it to *critic*NN#2, obtain $\lambda(t+1)$;
4. Calculate desired output (t) for *critic*NN#1;
5. Scale $R(t)$ and apply it to *critic*NN#1;
6. Calculate/execute weight changes for *critic*NN#1;
7. If $t < \text{epoch}$, increment t and go to 1;
8. Set *critic*NN#2=*critic*NN#1;
9. Go to flop epoch

Strategy 3b. Modified flip/flop strategy (b):Batch train *critic*NN#1 with suspended adaptation of *critic*NN#2.

This entry is included for completeness. Batch training here gives same results as **2b**.

Strategy 4. Single-stage. On-line train *actionNN* and *criticNN#1* with suspended adaptation of *criticNN#2*.

4a. Use *criticNN#2* to update both, *actionNN* and *criticNN#1*.

1. Scale $\mathbf{R}(t)$ and apply it to *actionNN*, obtain $\mathbf{u}(t)$;
2. Apply $\mathbf{u}(t)$ to the plant and obtain $\mathbf{R}(t+1)$;
3. Scale $\mathbf{R}(t+1)$, apply it to *criticNN#2*, obtain $\lambda(t+1)$;
4. Calculate desired output (t) for *criticNN#1*;
5. Calculate/execute weight changes for *actionNN* per Section 3;
6. Scale $\mathbf{R}(t)$ and apply it to *criticNN#1*;
7. Calculate/execute weight changes for *criticNN#1*;
8. If $t < \text{epoch}$, increment t and go to 1;
9. Set $\text{criticNN\#2} = \text{criticNN\#1}$ and go to 1.

4b. Use *criticNN#2* to update *criticNN#1*; use *criticNN#1* to update *actionNN*.

1. Scale $\mathbf{R}(t)$ and apply it to the *actionNN*, obtain $\mathbf{u}(t)$;
2. Apply $\mathbf{u}(t)$ to the plant and obtain $\mathbf{R}(t+1)$;
3. Scale $\mathbf{R}(t+1)$, apply it to *criticNN#2*, obtain $\lambda(t+1)$;
4. Calculate desired output (t) for *criticNN#1*;

5. Scale $\mathbf{R}(t+1)$, apply it to *criticNN#1*, obtain $\lambda(t+1)$;
6. Calculate/execute weight changes for *actionNN* per Section 3;
7. Scale $\mathbf{R}(t)$ and apply it to *criticNN#1*;
8. Calculate/execute weight changes for *criticNN#1*;
9. If $t < \text{epoch}$, increment t and go to 1;
10. Set *criticNN#2*=*criticNN#1* and go to 1.

Chapter 4

EXPERIMENTAL RESULTS

4.1 The Inverted Pendulum Problem

The well known "inverted pendulum" or "pole balancer" problem, is used here as the test bed. This "plant" has been successfully controlled using DHP, as reported in [4].

Figure 4: Schematic of pole balancer

This cart/pole system is modeled by the following Eqn. [1]:

$$R_6(t+1) = \frac{g \sin \Theta + \cos \Theta \left[\frac{-F - ml\dot{\Theta}^2 \sin \Theta + \mu_c \operatorname{sgn}(\dot{x})}{m_c + m} \right] - \frac{\mu_p \dot{\Theta}}{ml}}{l \left[\frac{4}{3} - \frac{m \cos^2 \Theta}{m_c + m} \right]}$$

$$R_3(t+1) = \frac{F + ml \left[\dot{\Theta}^2 \sin \Theta - \ddot{\Theta} \cos \Theta \right] - \mu_c \operatorname{sgn}(\dot{x})}{m_c + m}$$

$$R_1(t+1) = R_1(t) + \tau * R_2(t)$$

$$R_2(t+1) = R_2(t) + \tau * R_3(t)$$

$$R_4(t+1) = R_4(t) + \tau * R_5(t)$$

$$R_5(t+1) = R_5(t) + \tau * R_6(t)$$

where:

$$R_1(t) = x(t) \quad R_4(t) = \Theta(t)$$

$$R_2(t) = \dot{x}(t) \quad R_5(t) = \dot{\Theta}(t)$$

$$R_3(t) = \ddot{x}(t) \quad R_6(t) = \ddot{\Theta}(t)$$

and:

$\tau=0.05$ sec, sampling rate,

$g= 9.8 \text{ m/s}^2$, acceleration due to gravity,

$m_c=1.0 \text{ kg}$, mass of cart,

$m=0.1 \text{ kg}$, mass of pole,

$l=0.5\text{m}$, half pole length,

$\mu_c=0.0005$, coefficient of friction of cart on track,

$\mu_p=0.000002$, coefficient of friction of pole on cart

$F=$ force applied to cart's center of mass at time t

We define the utility function $U(t) = -0.25 * \left(R_4(t) - \text{desired angle} \right)^2$. Balancing the pole means desired angle=0.

4.1.1 Preliminary Comments

While coding of the model equations and of the basic Backprop NN paradigm is straightforward, getting the DHP process to converge turned out being a tedious task. The available literature doesn't offer full details (page limitations). We started with Strategy 1 and 2 (per [3]), and had to discover useful values for learning coefficients and the epoch size. Further, we discovered the efficacy of scaling the state space, and that using bias terms in the *critic*NN and particularly in the *action*NN gave problems. After this, exploring the strategies we here report followed more easily.

We simulated the inverted pendulum with the Euler iteration method, used analytical equations to compute the utility function U, and used an *action*NN and a *critic*NN. The neural network structures used are shown in Table II.

TABLE II
NEURAL NET STRUCTURES FOR CRITICNN AND ACTIONNN.

<i>Net</i>	<i>Input layer</i>	<i>Hidden layer</i>	<i>Output layer</i>
action	6 PE linear transfer	3 PE tanh transfer	1 PE const.*tanh transfer
critic	6 PE linear transfer	6 PE tanh transfer	6 PE linear transfer

These were trained using the basic Backpropagation algorithm. The number of PE's are specific to the plant (model). It was observed that the use of the hyperbolic tangent transfer function over the sigmoid transfer function was preferable because the natural range of variable values is positive and negative.

4.1.2 Performance:

For the pole-balancer test bed, the training procedure was to randomly initialize all the NNs [weight range: $(-.01, .01)$], and to then provide a specified sequence of starting angles (with zero being the "desired" angle), allowing the system to train on each starting angle for a specified number of seconds. The measure used for comparing the various DHP strategies takes the values achieved by the primary utility function during training and accumulates these over the sequence of starting angles; the measure is here called $C(j)$, where j labels a separate pass through the sequence of angles. In a sense, this measure incorporates the convergence speed of the DHP strategy as well as the quality of the controller's actions along the way.

The sequence of starting angles used for the training was: (5, -10, 20, -5, -20, 10) [degrees from vertical]. The system was allowed to train on each starting angle for 30 seconds. The same sequence was run 3 times [measures $C(1)$, $C(2)$, $C(3)$], with cumulative learning. For measuring the quality of the resulting controller at the end of training, the same sequence was applied one more time, with no learning [measure $C(4)$].

To test generalization capability of the resulting *action*NN (controller) design, a test sequence of starting angles was presented (-23, -18, -8, 3, 13, 23) and the results measured in the same way as above [$C(5)$]. In this case, the measure incorporates the speed of achieving balance and quality of the controller's actions to achieve this. The results were very encouraging, so a more aggressive generalization test was performed via a second test sequence: (-38, -33, 23, 38) [$C(6)$], reported below. It is remarkable

that in additional tests, strategy 4a and 4b controllers successfully generalized out to 48°.

The following table shows the parameters used for each DHP strategy for the results presented. Each was selected in order to yield lowest C(1) and highest probability of convergence. Note that the *action*NN is set up with a faster training rate than the *critic*NN. We observed that each strategy did best for a certain ratio of training rates of these two neural nets.

TABLE III
LEARNING COEFFICIENTS AND EPOCH SIZE

<i>Strategy</i> <i>Train parameter.</i>	<i>1</i>	<i>2a</i>	<i>2b/3b</i>	<i>3a</i>	<i>4a</i>	<i>4b</i>
critic NN: learn coeff	0.03	0.03	0.02	0.2	0.15	0.2
action NN: learn coeff	0.1	0.2	0.5	0.6	0.6	0.5
epoch: time steps	na	3	5	3	5	5

4.1.3 Bias Term

It is usually good practice to also input to each element of the hidden and output layer a so called constant bias term. The weight connection to this constant term adds an additional degree of freedom to the processing elements. An important observation made here is that the bias term makes it much harder to accomplish convergence for the criticNN and especially for the actionNN. Experiments with Strategy 1 showed that a bias term equal to 1 allows convergence with very small learning coefficients only. In all the following reported experiments, a bias term equal to zero was used.

4.1.4 Convergence Speed

All strategies succeeded in Learning to balance the pole without dropping it any more, based as one pass through the train data. However, not every training attempt converged to a solution; empirically, each of the strategies successfully learned in about 80 % of the trials, each trial starting with a new random initialization. Figures A.1 through A.6 in Appendix A show the pole angle (in degrees) over time for a successful learning process for each training strategy for the first pass through the train data.

Because of the random initialization of all weights, the results vary for separate runs. Fig. (5), (6) and (7) show the performance measures, $C(j)$, averaged over four successfully converged separate training runs. Fig. (5) shows the accumulated cost during the 1st pass through train data.

Figure 5: Accumulated cost during first pass through train data for the different strategies.

Strategies 4a and 4b accomplish the fastest learning. Strategy 4a is capable of learning

to balance the pole during the first 10 seconds of operation, see Fig. (A.5).

Fine tuning of the controller (the actionNN) is accomplished during the second and third pass through the train data. Fig. (6) shows the accumulated cost during the second and third passes through the train data.

We note that in all cases except strategy1, the accumulated costs were lowest during the third train pass . In additional experiments, the neural nets were trained for more passes through the train data. The lowest observed value for any $C(j)$ was 2.1..

Figure 6: Accumulated cost during second and third pass through train data for the different strategies.

Although convergence speed was the major criterion for optimizing the NN setup, we note here that the reported parameters (Table II and Table III) also yield the highest probability of convergence, which is at about 80% for all strategies.

4.1.5 Quality of Resulting Controller Design

After training the nets successfully, adaptation was stopped and no more weight changes took place. The actionNN was used as a non-adaptive controller. Fig. (7) shows the values for C(4), C(5) and C(6) as defined in Section 4.1.2.

Although the simulated plant is nonlinear, all obtained controllers were able to balance the pole in a $\pm 38^\circ$ range, even though training was only for a $\pm 20^\circ$ range. It is remarkable, that in additional experiments, strategy 4a and 4b controllers generalized out to 48° .

Figure 7: Accumulated cost {C(4), C(5) and C(6)} for test and generalization runs for the different strategies

4.2 Steering Control of a 4-Wheel Vehicle

In the previous Sections, the efficacy of the proposed new training strategy was demonstrated. Since this training strategy was developed on an empirical basis,

it was deemed appropriate to test the methodology on a substantially different dynamical system. To this end we turn to a 4-wheel vehicle with an independent electric drive motor on each wheel. This system is approximated here by what is known as a (nonlinear) bicycle steering model. The equations and documentation for the bicycle model were extracted from [9] which in turn were based on equations developed in [10].

We begin with the classical constant velocity bicycle model [10] which was used by Ackermann to develop his front/rear wheel steering decoupling system [11], though we use a full nonlinear model. The model includes a separate unknown tire side-slip function on each wheel and assumes that the vehicle mass is unknown. The front wheel steering model is obtained by restricting the rear wheel steering angle to zero.

4.2.1 Bicycle Model

The geometry assumed for the bicycle model is shown in Fig. (8). The mass of the vehicle is assumed to be concentrated at the front and back wheels with the total mass, m , while the wheelbase of the vehicle is $l = l_f + l_b$. The vehicle chassis is assumed to be pointed in the x -direction relative to the x - y coordinate system which is translated by an angle ψ , $\rho = \psi$, from an absolute x_0 - y_0 coordinate system. The vehicle velocity at the center-of-gravity has magnitude, v , at an angle β from the x -direction. We assume that v is constant, and of course, β varies through a turn.

Fig. (8) also illustrates the local geometry at each wheel. Here, δ_i , $i = \{f, b\}$, represents the steering angle for the front and back wheels, respectively. The velocity

of the chassis at the i th axle, $i = \{f, b\}$, is assumed to have magnitude v_i , at an angle β_i from the x axis. As such, the wheel side slip angle is $\alpha_i = \delta_i - \beta_i$, $i = \{f, b\}$. I.e., α_i , $i = \{f, b\}$, represents the angular difference between the direction in which the wheel is pointed and the direction in which it is actually moving.

Figure 8: Bicycle Model Geometry

This, in turn, produces a side force on the wheel with magnitude $f_i = f(\alpha_i)$ in the direction perpendicular to that in which the wheel is steered (i.e., $\delta_i + 90^\circ$) where f_i is a monotonic function of α_i .

The development of the bicycle steering model follows that in Mitschke [1990] and Ackermann [1994]. We start with Newton's laws in the x, y, and z-directions where z is the rotational direction about the center-of-gravity yielding.

$$\begin{bmatrix} mv(\dot{\beta} + \rho) \\ m\dot{v} \\ ml_f l_b \dot{\rho} \end{bmatrix} = \begin{bmatrix} -\sin(\beta) & \cos(\beta) & 0 \\ \cos(\beta) & \sin(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_x \\ f_y \\ m_z \end{bmatrix}$$

where f_x is the total force applied to the chassis in the x-direction, f_y is the total force in the y-direction, and m_z is the moment about the center-of-gravity. The term, $ml_f l_b$ in the above equation, is the moment of inertia of the chassis about the center-of-gravity. Now, assuming that the vehicle velocity about the center-of-gravity is constant through the turn we may set $v = 0$ in the above equation yielding

$$f_x = -\frac{\sin(\beta)}{\cos(\beta)} f_y$$

and upon substitution into the above equation yields the

Dynamic Equations

$$\begin{bmatrix} mv(\dot{\beta} + \rho)\cos(\beta) \\ ml_f l_b \dot{\rho} \end{bmatrix} = \begin{bmatrix} f_y \\ m_z \end{bmatrix}$$

Recalling that the only external forces applied to the chassis are those produced by the tire

side forces, $f = f(\alpha_b)$, the right hand side of the above equation may be evaluated yielding

the

Force Equations

$$\begin{bmatrix} f_y \\ m_z \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ l_f & -l_b \end{bmatrix} \begin{bmatrix} f_f(\alpha_f)\cos(\delta_f) \\ f_b(\alpha_b)\cos(\delta_b) \end{bmatrix}$$

Here, the 1st equation is simply the sum of the two tire side forces projected into the y-direction while the 2nd equation is the sum of the moments generated by the two tire side forces around the center-of-gravity.

We now derive some kinematics equalities for the system. In particular, the longitudinal velocity of the chassis in the x-direction is the same at every point on the chassis, hence

$$v_b \cos(\beta_b) = v \cos(\beta) = v_f \cos(\beta_f)$$

while the velocity of the chassis in the y-direction at the front and back axles can be computed in terms of the velocity at the center-of-gravity via

$$v_f \sin(\beta_f) = v \sin(\beta) + l_f \rho$$

$$v_b \sin(\beta_b) = v \sin(\beta) - l_b \rho$$

Upon dividing the above y-velocity equation by the appropriate terms in the x-velocity equation we obtain

$$\tan(\beta_f) = \frac{v \sin(\beta) + l_f \rho}{v \cos(\beta)} = \tan \beta + \frac{l_f \rho}{v \cos(\beta)}$$

$$\tan(\beta_b) = \frac{v \sin(\beta) - l_b \rho}{v \cos(\beta)} = \tan(\beta) - \frac{l_b \rho}{v \cos(\beta)}$$

This implies that

$$\beta_f = a \tan \left(\tan(\beta) + \frac{l_f \rho}{v \cos(\beta)} \right)$$

$$\beta_b = a \tan \left(\tan(\beta) - \frac{l_f \rho}{v \cos(\beta)} \right)$$

from which we obtain the desired

Kinematics Equations

$$\alpha_f = \delta_f - \beta_f = \delta_f - a \tan \left(\tan(\beta) + \frac{l_f \rho}{v \cos(\beta)} \right)$$

$$\alpha_b = \delta_b - \beta_b = \delta_b - a \tan \left(\tan(\beta) - \frac{l_f \rho}{v \cos(\beta)} \right)$$

The above derived Dynamic Equations, Force Equations, and Kinematics Equations combine to define a set of second order nonlinear state equations for our bicycle steering model. Here, the state variables are β and ρ while the inputs are the front and back wheel steering commands, δ_f and δ_b .

In addition to the state variables, which determine the position of the chassis on the road and the chassis yaw, in a hierarchical control structure the steering control system is required to generate velocity commands for the wheel controllers. The required outputs for the steering control system are therefore β , ρ , u_f and u_b , where u_f and u_b are the velocities of the front and back wheels in the direction in the plane of the wheels (i.e., the δ_i , $i = \{f, b\}$), direction. Here,

$$u_f = v_f \cos(\alpha_f)$$

$$u_b = v_b \cos(\alpha_b)$$

are simply the projections of the wheel velocity onto the direction in which the wheel is pointed.

Now, from the longitudinal velocity equations $v_f = v \frac{\cos(\beta)}{\cos(\beta_f)}$ and $v_b = v \frac{\cos(\beta)}{\cos(\beta_b)}$.

Upon substituting these equalities into the above equations and applying the cosine of the difference formula we obtain

$$u_f = v \frac{\cos(\beta)}{\cos(\beta_f)} \cos(\alpha_f) = v \frac{\cos(\beta)}{\cos(\beta_f)} \cos(\delta_f - \beta_f)$$

$$u_b = v \frac{\cos(\beta)}{\cos(\beta_b)} \cos(\alpha_b) = v \frac{\cos(\beta)}{\cos(\beta_b)} \cos(\delta_b - \beta_b)$$

$$= v \frac{\cos(\beta)}{\cos(\beta_f)} \left[\cos(\delta_f) \cos(\beta_f) + \sin(\delta_f) \sin(\beta_f) \right]$$

$$= v \frac{\cos(\beta)}{\cos(\beta_b)} \left[\cos(\delta_b) \cos(\beta_b) + \sin(\delta_b) \sin(\beta_b) \right]$$

Now, expanding this expression and substituting the above derived formulae for $\tan(\beta_f)$

and $\tan(\beta_b)$ obtained from the y-direction velocity formulae yields the

Output Equations

$$u_f = v \cos(\beta) \cos(\delta_f) + v \cos(\beta) \sin(\delta_f) \tan(\beta_f)$$

$$u_b = v \cos(\beta) \cos(\delta_b) + v \cos(\beta) \sin(\delta_b) \tan(\beta_b)$$

$$\begin{aligned}
&= v \cos(\beta) \cos(\delta_f) + v \cos(\beta) \sin(\delta_f) \left[\tan \beta + \frac{l_f \rho}{v \cos \beta} \right] \\
&= v \cos(\beta) \cos(\delta_b) + v \cos(\beta) \sin(\delta_b) \left[\tan \beta + \frac{l_b \rho}{v \cos \beta} \right] \\
&= v \cos(\beta) \cos(\delta_f) + v \sin(\beta) \sin(\delta_f) + l_f \rho \sin(\delta_f) \\
&= v \cos(\beta) \cos(\delta_b) + v \sin(\beta) \sin(\delta_b) + l_b \rho \sin(\delta_b) \\
&= v \cos(\delta_f - \beta) + l_f \rho \sin(\delta_f) \\
&= v \cos(\delta_b - \beta) + l_b \rho \sin(\delta_b)
\end{aligned}$$

which, together with the state variables, β and ρ , define the **output equations** for our steering model.

Although the dynamic equations, force equations, kinematics equations and output equations derived above could be combined into a single set of equations in the classical state space format, it is better here to retain four separate sets of equations formulating the steering model as shown in Fig. (9). Here, the kinematics equations are used to compute the tire side slip angles, α_f and α_b . These angles, in turn, drive the force equations which are used to compute f_y and m_z . Finally, this force vector drives the dynamic equations which are used to evaluate the derivative of the state vector, β and ρ . This is integrated to evaluate the state vector, β and ρ , which, in turn, drives the output equations. Of course, the input, δ_f and δ_b , and the state vector, β and ρ , are available to all modules.

Note, the above model represents a full nonlinear model of the vehicle's steering dynamics. The only assumptions are the use of a bicycle model and the constant velocity assumption.

Figure 9: Block Diagram for Bicycle Steering Model

4.2.2 Implementing DHP/Strategy 4 a for the Bicycle Steering Model

4.2.2.1 Preliminary Comments

The number of processing elements and the transfer functions for the actionNN and criticNN are shown in Table IV.

TABLE IV:

NUMBER OF ELEMENTS AND TRANSFER FUNCTIONS FOR ACTIONNN
AND CRITICNN

Net	Input layer	Hidden layer	Output layer
action	7 PE linear transfer	4 PE tanh transfer	1 PE tanh transfer
critic	7 PE linear transfer	5 PE tanh transfer	4 PE linear transfer

The inputs to the nets were scaled into a (-1; 1) range.

The inputs to criticNN and actionNN are:

1. desired position-y
2. desired velocity-y
3. desired front axle acceleration
4. position-y
5. velocity-y
6. vehicle rotation
7. front axle acceleration

The output of the actionNN is the steering signal. The criticNN's output is λ_1 to λ_4 ,

where:

$$\lambda_1 = \partial J / \partial \text{position } y$$

$$\lambda_2 = \partial J / \partial \text{velocity } y$$

$$\lambda_3 = \partial J / \partial \text{vehicle rotation}$$

$$\lambda_4 = \partial J / \partial \text{front axle acceleration}$$

The primary utility function was defined as:

$$U(t) = - (\text{desired position } y - \text{position } y)^2 - 0.5 * (\text{desired velocity } y - \text{velocity } y)^2$$

4.2.2.2 Preconditioning the ActionNN

It was found useful to manually precondition the actionNN before the actual training took place. The preconditioning consisted simply of setting the initial weight

connections from inputs 1 and 4 to the hidden layer to a value of one (i.e., to implement a $(y - y_{des})$ term) and setting the initial weight connection from hidden element one to the output to 1, with the remaining weights initialized to a small random value.

Figure 10: Preconditioned ActionNN

4.2.2.3 Dividing the State Space into Units of Training Time

In order to achieve convergence in the DHP learning process, it was found useful to adapt a training method proposed in [13]. This method divides the state space into portions. The neural nets start to train on the smallest fraction, here defined as unit of time, along the trajectory to be learned. Then, after successful learning to control the plant in that unit, the neural nets are trained on two units of time, the fraction already learned and another one of the same size. After learning to control the plant in this

bigger fraction of state space, a third one is added to the training process. In general, after successful learning of the given portion, another unit of training time is added., and training starts over with the previously designed controller. The controller design is accomplished after successfully training the neural nets over the expected range in state space.

4.2.2.4 Selection Module

It was observed that a more accurate controller could be designed by training two controllers: one to accomplish left turns and the other one to accomplish right turns. The decision on which to use is made by a selection module, with a hysteresis built in the decision process to smooth out the transition. The selection module selected the left or right actionNN according to:

- Initialize: If $daf > 0$ then right-turn controller, else left turn-controller.
- If $(daf_der < 0)$ and $(daf \leq -0.4434)$ then switch to right turn-controller
- If $(daf_der \geq 0)$ and $(daf > 0.4434)$ then switch to left-turn controller,

where daf is desired acceleration front, and daf_der is the derivative of desired acceleration front with respect to time.

4.2.2.5 Performance

The DHP methodology described in the earlier sections was applied to this steering control problem. Various experiments were run to determine appropriate values for learning coefficients, epoch size and sampling rates. A successful controller design was accomplished. Fig. (11) shows the trajectory of the car using the controller while

executing a lane change (traveling from left to right along the x axis; $y=0$ represents the center line of lane 1, and $y=3$ represents the center line of lane 2). The learning coefficient for the criticNN was 0.3 and the learning coefficient for the actionNN was equal 0.03. Strategy 4a was employed to train actionNN and criticNN's. The epoch size was equal to 50 and sampling rate was 0.01 seconds. Analytical equations were used to calculate plant terms in the neural net update equations. The friction coefficient was 1, what represents dry road conditions and "new tires". The unit of training time used in the training process was one second.

Figure 11: Trajectory of Car on Road (friction coefficient = 1)

Chapter 5

CONCLUSION

The main contribution of this work has been development of a faster training strategy for the Dual Heuristic Programming method of designing a plant controller employing neural networks. The improved strategy was developed based on experiments with an inverted pendulum as the plant to be controlled. The equations used to represent this (nonlinear) system have six state variables, all of which are known by the controller. The next step was to apply the strategy to a dynamical system with a higher-dimensional representation. The control context in this case was to steer a four-wheel vehicle which has independent electric motors on the front two wheels. The equations used to represent this (nonlinear) system involve 16 state variables, only four of which are known by the controller. Successful design of a controller for this context was also accomplished with the proposed training Strategy 4a.

5.1 Future Research

The next step of the envisioned research program will be to apply this methodology in adapting the controller design in the face of changes to the plant. Some first steps toward this objective are described below, and suggestions for further research are given.

For the inverted pendulum problem, the simulated plant change explored was to increase the length of the pole from 1 meter to 2.4 meters long. The DHP/Strategy 4a method successfully developed a new controller design, as expected. After the pole-

length change, an experiment, which used a single pass through the (5, -10, 20, -5, -20, 10) [degrees from vertical] training sequence reported earlier was applied, where the starting controller design was the one already developed for the original 1 meter length pole. A time history of the pole angles during the first 4 seconds after the step change in length is shown in Fig. (12). We note that the new design was accomplished from the original design without dropping the pole. Plots A.7 and A.8 in Appendix A show the adaptation time history during the -20° disturbance; superimposed is the response of the original controller with $l=1\text{m}$ for the same disturbance.

For the automobile steering problem, the important parameter to which adaptation is required is the changing coefficient of friction between the tire(s) and the ground -- e.g., due to tire wear and/or road conditions such as loose gravel, water, or ice. The experiment reported in the previous section was based on a coefficient of friction of 1.0 (good tire condition, dry road). We next conducted experiments with the coefficient modified to .85 and then to .70. The results of these controller design efforts by the DHP/Strategy 4a method are shown in Figs. (13) and (14). We note that again the resulting controller yields "very nice" results.

Figure 12: Pole angles while strategy 4a adapts 1m controller design to 2.4m during first 30 sec. of adaptation. (5° deflection)

Figure 13: Trajectory of Car on Road (friction coefficient = 0.85)

Figure 14: Trajectory of Car on Road (friction coefficient = 0.7)

For purposes of useful application in the automobile steering context, a method for real-time adaptation to changes in the coefficient of friction will need to be developed. Future work on this problem will benefit from exploring the capability of the DHP method to design a successful controller over the full range of values of the coefficient (ice condition is represented by a value of .3). Selecting values of neural network training parameters is a tedious process. For example, reducing the friction coefficient from, 1.0 down to .85 and to .70 required reducing the size of the unit of training time from the 1 second period reported in the previous section down to .5 seconds in this case, the learning rate was increased by a factor of five for the actionNN, and the epoch size was cut in half. A quick experiment of the friction coefficient all the way down to .3 with these modified training parameter values did not converge. It is predicted that the unit of training time will have to be reduced significantly (perhaps geometrically), and collateral changes will be required of the other training parameters. This is left for future work.

A possible strategy for developing an adaptive controller in this context is to train a collection of different actionNN modules, one each for a selected set of values for the friction coefficient. Then, a hierarchical control approach might be pursued in which another neural network is developed to select the appropriate actionNN module to be used for a given situation, determination of which will be based on sensor data from the car platform, e.g., from a set of appropriate accelerometers. Of course, the issues of system stability will have to be considered in any such approach. This is

proposed as a major future research project.

5.2 Synopsis of Contribution

In DHP and associated methods, a "desired output" (target) is needed for training the *critic*NN, and this is typically calculated by running the *critic*NN one more computational cycle to provide its next-in-time output, and then use this value to compute the target for the present-time cycle. The error term is calculated and the *critic*NN update is performed in the usual way. These approaches typically use a "copy" of the *critic*NN to perform (or at least explain) the calculation of the target, and both copies are updated at the same time. Since the *critic*NN that calculates the target is changing with each update, it provides a "moving target" for the *critic*NN training. In this thesis, a real copy of the *critic*NN (*critic*NN#2) for making the target calculations is introduced, and very importantly, this *critic*NN#2 is trained differently than is *critic*NN#1. The idea is to provide the target from a stable platform during an epoch while adapting *critic*NN#1. Then at the end of the epoch, *critic*NN#2 is made identical to the then-current adapted state of *critic*NN#1, and a new epoch starts. In this way, both the *critic*NN#1 and the *action*NN can be trained on-line during each epoch, with a faster overall convergence than the older approach. Further, the lower relative values of C(4), C(5) and C(6) suggest a "better" design is developed for the controller (*action*NN).

References

- [1] Barto, A., Sutton, R. and Anderson, C. "Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems" in IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-13, No.5, Sep/Oct 1983.
- [2] Prokhorov, D. and Wunsch, D. "Advanced Adaptive Critic Designs", PROCEEDINGS WCNN'96, pp. 83-87, San Diego, Erlbaum, Sept. 1996
- [3] Santiago, R., presentation at the First Joint Mexico-US International Workshop on Neural Networks and Neurocontrol, Playacar, Mexico, Sept. 1995.
- [4] Santiago, R. and Werbos, P. "New Progress Towards Truly Brain-Like Intelligent Control", PROCEEDINGS WCNN '94, pp. I-2tol-33, San Diego, Erlbaum, 1994.
- [5] Visnevski, N. and Prokhorov, D. "Control of a Nonlinear Multivariable System with Adaptive Critic Designs", in Intelligent Engineering Systems through Artificial Neural Networks 6 (PROC. ANNIE '96), Dagli, et.al., Eds., ASME Press, pp.559- 565, 1996.
- [6] Werbos, P. "Approximate Dynamic Programming for Real-Time Control and Neural Modeling", Ch. 13 in Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches, (White, D.A. and Sofge, D.A., eds.), Van Nostrand Reinhold, New York, NY, 1992
- [7] Prokhorov, D., Santiago, R., Wunsch, D. "Adaptive Critic Designs: A Case

- Study for Neurocontrol" in Neural Networks, Vol. 8, No. 9, pp 1367-1372, 1995
- [8] Lendaris, G. G. and Paintz, C. "Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method" to appear, PROCEEDINGS ICNN'97, Houston, TX
 - [9] AAC "Advanced Intelligent Control of next generation vehicles", National Science Foundation SBIR Phase I Final Report, August 1995
 - [10] Mitschke, M. Dynamik der Kraftfahrzeuge, Vol. C, Springer-Verlag, Berlin.
 - [11] Ackermann, J., "Robust Decoupling, Ideal Steering Dynamics and Yaw Stabilization of 4WS Cars", Automatica, Vol. 60, pp. 1761-1768.
 - [12] "Neural Computing" A Technology Handbook, NeuralWare, Inc., 1995
 - [13] Nguyen, D. and B. Widrow: "The Truck Backer-Upper: An Example of Self-Learning in Neural Networks", Chapter 12 in Neural Networks for Control, Miller, W. T., Sutton, S. and P.J. Werbos (Eds.) MIT Press 1991
 - [14] Bellman, R. E. Dynamic programming, Princeton, NJ: Princeton University Press (1957)

APPENDIX A

Figure A.1: Algorithm 1

Figure A.2: Algorithm 2a

Figure A.3: Algorithm 2b

Figure A.4: Algorithm 3a

Figure A.5: Algorithm 4a

Figure A.6: Algorithm 4b

Figure A.7: Pole angle after -20° disturbance for pole length 1m and 2.4m

Figure A.8: Force applied by controller after -20° disturbance for pole length 1m and 2.4m

APPENDIX B

Pascal Code for Inverted Pendulum simulation, using strategy 4a.

```
{ last change: May/4/97; 4a: concurrent action and critic learning }
{important functions:
    target : calculates desired output for criticNN
    F_fu: calculates "error" for actionNN"
    execute: sets criticNN#2=criticNN#1
    control: calculates next time state of plant
    critic: forward pass through criticNN
    action: forward pass through actionNN
    upaction: updates actionNN weights
    upcritic: updates criticNN}

program invpend; { plant is inverted pendulum }
const
    sampling=0.05;
    maxpe=8;
    g=9.8; moc=1; mc=0.0005; mp=0.000002;
    duration=100000;

    fo=0.1;deltax=0.05;
    a=0.1;b=0.5;c=1;d=1;
    ic=6; hc=6; oc=6;
    ia=6; ha=3; oa=1;
    im=7; hm=6; om=6;
var
    t,nf,fallen,count: integer;
    w, dw: array [1..3, 2..3, 1..maxpe, 0..maxpe] of real;
    out,e : array [1..3, 1..3, 0..maxpe] of real;
    rr: array [1..om, 0..500] of real;
    lr: array [1..oc] of real;
    ls: array [1..oc,0..500] of real;
    u: array [1..oa,0..500] of real;
    maxr: array [1..om] of real;
    gh: 0..om;
    pumpe,experiment,k:longint;
    fall1,num:integer;
    mop,length,power,desired,sum,range:real;
    f: text;
    s: string;
```

```

yes:boolean;
function sgn(w:real):real;
begin
if w=0 then sgn:=0;
if w>0 then sgn:=1;
if w<0 then sgn:=-1;
end;
function eq1(tet:integer):real;
var q10,q11,teta:real;
begin
q10:=(-u[1,tet]-
mop*length*sqr(rr[5,tet])*sin(rr[4,tet])+mc*sgn(rr[2,tet]))/(moc+mop);
q11:=cos(rr[4,tet])*q10+g*sin(rr[4,tet])-mp*rr[5,tet]/(mop*length);
teta:=q11/(length*(1.333-mop*sqr(cos(rr[4,tet]))/(moc+mop)));
eq1:=teta;
end;
function eq2(te:integer):real;
var q22,q20:real;
begin
q20:=u[1,te]+mop*length*(sqr(rr[5,te])*sin(rr[4,te])-cos(rr[4,te])*rr[6,te]);
q22:=(q20-mc*sgn(rr[2,te]))/(moc+mop);
eq2:=q22;
end;
procedure control(time:integer); {simulats real plant}
begin
rr[1,time+1]:=rr[1,time]+sampling*rr[2,time];
rr[2,time+1]:=rr[2,time]+sampling*rr[3,time];
rr[4,time+1]:=rr[4,time]+sampling*rr[5,time];
rr[5,time+1]:=rr[5,time]+sampling*rr[6,time];
rr[6,time+1]:=eq1(time);
rr[3,time+1]:=eq2(time);
end;
function sig(imm:real):real;
var g:real;
begin
if abs(imm)<20 then g:=1/(1+exp(-imm)) else
begin
if imm<0 then g:=0 else g:=1;
end;
sig:=g;
end;
function tanh(z:real):real;
var ez,emz,v,th:real;
begin

```

```

th:=0;
if abs(z)<20 then
begin
  ez:=exp(z);
  emz:=exp(-z);
  v:=(ez-emz)/(ez+emz);
  th:=1;
end;
if th=0 then begin
  if z<0 then v:=-1 else v:=1;
end;
tanh:=v;
end;
procedure critic(time:integer); {computes lr[..] }
var
a,b:integer;
net:real;
begin { critic }
  for a:=1 to ic do
    begin {input to critic}
      out[1,1,a]:=rr[a,time]/maxr[a];
    end;
  for a:=1 to hc do begin {output hidden critic}
    net:=0;
    for b:=0 to ic do begin
      net:=net+w[1,2,a,b]*out[1,1,b];
    end;
    out[2,1,a]:=tanh(net);
  end;
  for a:=1 to oc do begin {output output critic}
    net:=0;
    for b:=0 to hc do begin
      net:=net+w[1,3,a,b]*out[2,1,b];
    end;
    out[3,1,a]:=net;
    lr[a]:=out[3,1,a];
  end;
end; { critic }
procedure critic2(time:integer); {computes lr[..] }
var
a,b:integer;
net:real;
begin { critic }
  for a:=1 to ic do

```

```

begin    {input to critic}
  out[1,3,a]:=rr[a,time]/maxr[a];
end;
for a:=1 to hc do begin    {output hidden critic}
net:=0;
  for b:=0 to ic do begin
    net:=net+w[3,2,a,b]*out[1,3,b];
  end;
  out[2,3,a]:=tanh(net);
end;
for a:=1 to oc do begin    {output output critic}
net:=0;
  for b:=0 to hc do begin
    net:=net+w[3,3,a,b]*out[2,3,b];
  end;
  out[3,3,a]:=net;
  lr[a]:=out[3,3,a];
end;
end;    { critic2 }
procedure action(time:integer); {computes u(t) through action }
var
a,b:integer;
net:real;
begin
  for a:=1 to ia do begin    {input action}
    out[1,2,a]:=rr[a,time]/maxr[a];
  end;
  for a:=1 to ha do begin    {output hidden action}
net:=0;
    for b:=0 to ia do begin
      net:=net+w[2,2,a,b]*out[1,2,b];
    end;
    out[2,2,a]:=tanh(net);
  end;
  for a:=1 to oa do begin    {output u(t)}
net:=0;
    for b:=0 to ha do begin
      net:=net+w[2,3,a,b]*out[2,2,b];
    end;
    out[3,2,a]:=tanh(net);
    u[a,time]:=power*out[3,2,a];
  end;
end;
{ functions used for F_fu(t) }

```



```

function F_nu(gg:integer):real;
begin
  F_nu:=0;
end;
function drt1_dut(k,u,tt:integer):real;      { drm[k,t+1]/du[k,t] }
var po,pop:real;
begin
  po:=0;
  pop:=cos(rr[4,t])/(moc+mop);
  if k=6 then po:=-pop/(length*1.333-length*mop*sqr(cos(rr[4,tt]))/(moc+mop));
  if k=3 then po:=1/(moc+mop);
  drt1_dut:=po;
end;      { F_Ru }
procedure F_fu(mtime:integer);      { computes e[3,2,i] }
var mi,ms:integer;
su,ma,mb,mc:real;
begin
  for mi:=1 to oa do begin
    ma:=F_nu(mi);
    mb:=0;
    for ms:=1 to om do begin
      mc:=drt1_dut(ms,mi,mtime);
      mb:=mb+lr[ms]*mc;
    end;
    e[3,2,mi]:=ma+mb;
  end; { i }
end;      { F_u }
{ functions used in target }
function dn_dr(ii:integer):real;      { derivative of utility w/rt r(t) }
var no,nenner:real;
begin
  no:=0;
  if ii=4 then no:=-b*(rr[4,t]-desired);
  dn_dr:=no;
end;
function dout21m3_dout11i(m2,m1:integer):real;
var r:integer;
r1,rr:real;
begin
  r1:=0;
  for r:=1 to ia do begin
    if r=m1 then rr:=1/maxr[m1] else rr:=0;
    r1:=r1+w[2,2,m2,r]*rr;
  end;

```

```

    dout21m3_dout11i:=(1-out[2,2,m2])*(1+out[2,2,m2])*r1;
end;
    { dout21m3_dout11i }
function dujdrit(j,i:integer):real;
var m3:integer;
    mm3,mmm3:real;
begin
    mm3:=0;
    for m3:=1 to ha do begin
        mmm3:=dout21m3_dout11i(m3,i);
        mm3:=mm3+w[2,3,j,m3]*mmm3;
    end;
    { m3 }
    dujdrit:=(power-power*out[3,2,j])*(1+out[3,2,j])*mm3;
end;
    { computes part of h1, dujdrit }
function e1(tet,tim:integer):real;
    { used for drkt1_drit }
var q0,q1,eta,mcc:real;
    r: array [1..6] of real;
    bu:integer;
begin
    for bu:=1 to 6 do begin
        r[bu]:=rr[bu,tim];
        if bu=tet then r[bu]:=r[bu]+deltax;
    end;
    mcc:=mc*sgn(r[2]);
    q0:=(-u[1,tim]-mop*length*sqr(r[5])*sin(r[4])+mcc)/(moc+mop);
    q1:=cos(r[4])*q0+g*sin(r[4])-mp*r[5]/(mop*length);
    eta:=q1/(length*(1.333-mop*sqr(cos(r[4]))/(moc+mop)));
    e1:=eta;
end;
function e2(te,im:integer):real;
var p22,p20,lp,sn:real;
    ra: array [1..6] of real;
    lauf:integer;
begin
    for lauf:=1 to om do
        begin
            if lauf=te then ra[lauf]:=(rr[lauf,im]+deltax) else ra[lauf]:=rr[lauf,im];
        end;
        p20:=u[1,im]+mop*length*(sqr(ra[5])*sin(ra[4])-cos(ra[4])*ra[6]);
        sn:=sgn(ra[2]);
        p22:=(p20-mc*sn)/(moc+mop);
        e2:=p22;
    end;
end;
function drkt1_drit(k,i,time:integer):real;
var la1,la2,la3,la4,la5:real;

```

```

begin
la1:=0;la5:=0;
  la3:=drt1_dut(k,1,time);
  la4:=dujdrit(1,i);
  if k=1 then begin
    if i=1 then la5:=1;
    if i=2 then la5:=sampling;
    la1:=la5+la3*la4;
    end;
  if k=2 then begin

    if i=2 then la5:=1;
    if i=3 then la5:=sampling;
    la1:=la5+la3*la4;
    end;
  if k=4 then begin

    if i=4 then la5:=1;
    if i=5 then la5:=sampling;
    la1:=la5+la3*la4;
    end;
  if k=5 then begin

    if i=5 then la5:=1;
    if i=6 then la5:=sampling;
    la1:=la5+la3*la4;
    end;
  if k=6 then begin
    la2:=e1(i,time);
    la1:=(la2-rr[6,time+1])/deltax+la3*la4;
    end;
  if k=3 then begin
    la2:=e2(i,time);
    la1:=(la2-rr[3,time+1])/deltax+la3*la4;
    end;
  drkt1_drit:=la1;
end;
procedure target(time:integer);      { computes ls[...t] }
var h2,i:integer;
    h5,hh2,hhh2:real;
begin
  for i:=1 to oc do
    begin
      hh2:=0;

```

```

    for h2:=1 to om do
        begin
            hhh2:=drkt1_drit(h2,i,t);
            hh2:=hh2+lr[h2]*hhh2;
        end; { h2 }
        h5:=dn_dr(i);
        ls[i,time]:=h5+hh2;
    end; { i }
end; {target}
procedure upcritic(tu:integer); {updates critic}
const momc=0.0;
var i,j,n,o,ll9:integer;
    su,tp,sp,lcoefco,lcoefch:real;
begin {upcritic}
    lcoefco:=0.15;
    lcoefch:=0.15;
    for i:=1 to oc do begin {each output pe}
        for j:=0 to hc do begin {from each hidden pe}
            dw[1,3,i,j]:=momc*dw[1,3,i,j]+lcoefco*(ls[i,tu]-out[3,1,i])*out[2,1,j];
        end;
    end; {dw for output pe done}
    for n:=1 to hc do begin
        for o:=0 to ic do begin
            su:=0;
            for ll9:=1 to oc do begin
                sp:=fo+(1+out[2,1,n])*(1-out[2,1,n]);
                su:=su+(ls[ll9,tu]-out[3,1,ll9])*w[1,3,ll9,n]*out[1,1,o]*sp;
            end; { ll9 }
            dw[1,2,n,o]:=momc*dw[1,2,n,o]+lcoefch*sus;
            w[1,2,n,o]:=w[1,2,n,o]+dw[1,2,n,o];
        end; { o }
    end; { n : hidden weights adjusted }
    for i:=1 to oc do begin
        for j:=0 to hc do begin
            w[1,3,i,j]:=w[1,3,i,j]+dw[1,3,i,j];
        end;
    end; {output weights done}
end; {upcritic}
procedure upaction1(ty:integer);
const moma=0.0;
var i,j,l,f,g:integer;
    su,ee:real;
    lim,lcoefao,lcoefah,foa:real;
begin {upaction}

```

```

lcoefao:=0.6;lcoefah:=0.6;
for i:=1 to oa do begin
ee:=e[3,2,i];
e[3,2,i]:=((power+power*out[3,2,i])*(1-out[3,2,i]))*e[3,2,i];
foa:=0.0;
ee:=e[3,2,i]+foa*ee;
for j:=0 to ha do begin
dw[2,3,i,j]:=moma*dw[2,3,i,j]+lcoefao*e[3,2,i]*out[2,2,j];
end;
end;          { output weight changes calculated }
for i:=1 to ha do begin
su:=0;
for j:=1 to oa do begin
su:=su+w[2,3,j,i]*e[3,2,j];
end;  { j }
e[2,2,i]:=su;
for j:=0 to ia do begin
e[2,2,i]:=e[2,2,i]*(foa+(1+out[2,2,i])*(1-out[2,2,i]));
dw[2,2,i,j]:=e[2,2,i]*out[1,2,j]*lcoefah+moma*dw[2,2,i,j];
w[2,2,i,j]:=w[2,2,i,j]+dw[2,2,i,j];
end;  { j }
end;          { i, all hidden weights adjusted }
for i:=1 to oa do begin
for j:=0 to ha do begin
w[2,3,i,j]:=w[2,3,i,j]+dw[2,3,i,j];
end;
end;          { out action weights adjusted }
end;          { upaction }
procedure initweights;          {sets all weights randomly}
var i,j:integer;
begin
randomize;
for i:=1 to hc do begin          {hidden critic }
for j:=0 to ic do begin
w[1,2,i,j]:=random(100)/5000-0.01;
dw[1,2,i,j]:=0;
end;
end;
for i:=1 to oc do begin          { out critic }
for j:=0 to hc do begin
w[1,3,i,j]:=random(100)/5000-0.01;
dw[1,3,i,j]:=0;
end;
end;          {critic }

```

```

for i:=1 to ha do begin
  for j:=0 to ia do begin
    w[2,2,i,j]:=random(100)/5000-0.01;
    dw[2,2,i,j]:=0;
  end;
end;
for i:=1 to oa do begin
  for j:=0 to ha do begin
    w[2,3,i,j]:=random(100)/5000-0.01;
    dw[2,3,i,j]:=0;
  end;
end;
for i:=1 to hm do begin
  for j:=0 to im do begin
    w[3,2,i,j]:=random(100)/5000-0.01;
    dw[3,2,i,j]:=0;
  end;
end;
for i:=1 to om do begin
  for j:=0 to hm do begin
    w[3,3,i,j]:=random(100)/5000-0.01;
    dw[3,3,i,j]:=0;
  end;
end;
end;          { all weights initialized, end initweights }
procedure bias(time:integer;ran:real);          { sets bias output, initial states }
  var i,j:integer;
      ii:real;
  begin
    randomize;
    for i:=1 to 3 do begin
      for j:=1 to 2 do begin
        out[j,i,0]:=0.01;
      end;
    end;          { bias set }
    for i:=1 to om do
      begin
        rr[i,time]:=0;
      end;
      rr[4,time]:=ran*0.087266;
      writeln('initial angle ',rr[4,time]*180/3.1415927:6:0);
    end;          { bias }
procedure weightcheck;
  var s,l:integer;

```

```

begin
  writeln('to hidden critic');
  for l:=1 to hc do begin
    for s:=0 to ic do begin
      write(w[1,2,l,s]:7:2, ' ');
    end;
    writeln;
  end;
  writeln('to output critic');
  for l:=1 to oc do begin
    for s:=0 to hc do begin
      write(w[1,3,l,s]:7:2, ' ');
    end;
    writeln;
  end;
  writeln('to hidden action');
  for l:=1 to ha do begin
    for s:=0 to ia do begin
      write(w[2,2,l,s]:7:2, ' ');
    end;
    writeln;
  end;
  writeln('to output action');
  for l:=1 to oa do begin
    for s:=0 to ha do begin
      write(w[2,3,l,s]:7:2, ' ');
    end;
    writeln;
  end;

  end;          { weightcheck }
procedure execute;
var i,j:integer;
begin
  for i:=1 to hc do begin          {hidden critic }
    for j:=0 to ic do begin
      w[3,2,i,j]:=w[1,2,i,j];
    end;
  end;
  for i:=1 to oc do begin          { out critic }
    for j:=0 to hc do begin
      w[3,3,i,j]:=w[1,3,i,j];
    end;
  end;
end;

```

```

end;                                { critic }
procedure flip;
begin
    for t:=0 to pompe do begin

        action(t);
        control(t);
        critic2(t+1);
        target(t);
        F_fu(t);
        upaction1(t);
        critic(t);
        upcritic(t);
        if abs(rr[4,t])>1.4 then
            begin
                fall1:=fall1+1;
                bias(t+1,range);
            end;
        end;
    execute;
end;    { flip }
procedure test;
begin
    for t:=0 to pompe do begin

        action(t);
        control(t);

        if abs(rr[4,t])>1.4 then
            begin
                fall1:=fall1+1;
                bias(t+1,range);
            end;
        end;

    end;    { flip }
begin    { simulation dhp }
initweights;execute;
maxr[1]:=4;maxr[2]:=4;maxr[3]:=12;maxr[4]:=1.6;maxr[5]:=3.5;maxr[6]:=15;
mop:=0.1; length:=0.5;power:=15;count:=120;
pumpe:=4;fallen:=0;desired:=0.0;
sum:=0;
Assign(f,'sf_4a_1.asc');
rewrite(f);

```



```

for nf:=1 to 3 do begin

for experiment:=1 to 6 do begin
  case experiment of
    1: range:=1;
    2: range:=-2;
    3: range:=4;
    4: range:=-1;
    5: range:=-4;
    6: range:=2;
  end;
  bias(0,range);
  for num:=1 to count do
    begin
      fall1:=0;
      flip;
      for t:=0 to pumpe do
        begin
          if nf=1 then
            begin
              str(180*rr[4,t]/3.1415927:6:3,s);
              write(f,s+' ');
              str(u[1,t]/30:6:2,s);
              write(f,s);writeln(f);
            end;
            sum:=sum+sqr(rr[4,t]-desired);
          end;
          fallen:=fallen+fall1;
          for gh:=1 to om do
            begin
              rr[gh,0]:=rr[gh,pumpe+1];
            end;
          end;
        end;
      end;
      { experiment }
      str(fallen:6,s);
      write(f,s+' ');
      str(sum:6:2,s);
      write(f,s+' ');
      writeln(f);
      writeln(sum:6:1);
      fallen:=0;sum:=0;
    end; { nf }
  for experiment:=1 to 6 do begin
    case experiment of

```

```

1: range:=1;
2: range:=-2;
3: range:=4;
4: range:=-1;
5: range:=-4;
6: range:=2;
end;
bias(0,range);
for num:=1 to count do
begin
    fall1:=0;
    test;
    for t:=0 to pumpe do
        begin
            sum:=sum+sqr(rr[4,t]-desired);
        end;
    fallen:=fallen+fall1;
    for gh:=1 to om do
        begin
            rr[gh,0]:=rr[gh,pumpe+1];
        end;
    end;
end;
{ experiment }
str(fallen:6,s);
write(f,s+' ');
str(sum:6:2,s);
write(f,s+' ');
writeln(f);
writeln(sum:6:1);
for experiment:=1 to 4 do begin
case experiment of
1: range:=-7.6;
2: range:=-6.6;
3: range:=4.6;
4: range:=7.6;
end;
bias(0,range);
for num:=1 to count do
begin
    fall1:=0;
    test;
    for t:=0 to pumpe do
        begin
            str(180*rr[4,t]/3.1415927:6:3,s);

```

```

        write(f,s+' ');
        str(u[1,t]/30:6:2,s);
        write(f,s);writeln(f);
        sum:=sum+sqr(rr[4,t]-desired);
    end;
    fallen:=fallen+fall1;
    for gh:=1 to om do
        begin
            rr[gh,0]:=rr[gh,pumpe+1];
        end;
    end;
end;          { experiment }
str(fallen:6,s);
write(f,s+' ');
str(sum:6:2,s);
write(f,s+' ');
writeln(f);
writeln(sum:6:1);
Close(f);      readln;
end.          { invpend }

```