Neural Networks for Self-Learning Control Systems

Derrick H. Nguyen and Bernard Widrow

ABSTRACT: Neural networks can be used to solve highly nonlinear control problems. This paper shows how a neural network can learn of its own accord to control a nonlinear dynamic system. An emulator, a multilayered neural network, learns to identify the system's dynamic characteristics. The controller, another multilayered neural network, next learns to control the emulator. The self-trained controller is then used to control the actual dynamic system. The learning process continues as the emulator and controller improve and track the physical process. An example is given to illustrate these ideas. The "truck backer-upper," a neural network controller steering a trailer truck while backing up to a loading dock, is demonstrated. The controller is able to guide the truck to the dock from almost any initial position. The technique explored here should be applicable to a wide variety of nonlinear control problems.

Introduction

This paper addresses the problem of controlling severely nonlinear systems from the standpoint of utilizing neural networks to achieve nonlinear controller design. The methodology shows promise for application to control problems that are so complex that analytical design techniques do not exist and may not exist for sometime to come. Neural networks can be used to implement highly nonlinear controllers with weights or internal parameters that can be determined by a self-learning process.

Neural Networks

A *neural network* is a system with inputs and outputs and is composed of many simple and similar processing elements. The processing elements each have a number of internal parameters called *weights*. Changing the weights of an element will alter the behavior of the element and, therefore, will also alter the behavior of the whole network. The goal here is to choose the weights of the network to achieve a desired input/output relationship. This process is known as *training the network*. The network can be considered memoryless in the sense that, if one keeps the weights constant, the output vector depends only on the current input vector and is independent of past inputs.

The authors are with Information Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

Adalines

The processing element used in the networks in this paper, the Adaline [1], is shown in Fig. 1. It has an input vector $X = \{x_i\}$, which contains n components, a single output y, and a weight vector $W = \{w_i\}$, which also contains n components. The weights are variable coefficients indicated by circles with arrows. The output y equals the sum of inputs multiplied by the weights and then passed through a nonlinear function. (Note: In the early 1960s, Adaline elements utilized sharp quantizers in the form of signum functions. Today both signum and the differentiable sigmoid functions are used.)

$$s(X) = \sum_{i=0}^{n-1} w_i x_i \tag{1}$$

$$y(X) = f(s(X)) \tag{2}$$

The nonlinear function $f(s)$ used here is the sigmoid function

$$f(s) = [1 - \exp(-2s)]/[1 + \exp(-2s)] = \tanh(s) \tag{3}$$

With this nonlinearity, the Adaline behaves similar to a linear filter when its output is small, but saturates to $+1$ or -1 as the output magnitude increases. It should be noted that one of the Adaline's inputs is usually set to $+1$. This provides the Adaline with a way of adding a constant bias to the weighted sum.

The goal here is to train the Adaline to achieve a desired form of behavior. During the training process, the Adaline is presented with an input X, which causes its output to be $y(X)$. We would like the Adaline to output a desired value $d(X)$ instead, and so we adjust the weights to cause the output to be something closer to $d(X)$ the next time X is presented. The value $d(X)$ is called the *desired response.* Many input, desired-response pairs are used in the training of the weights.

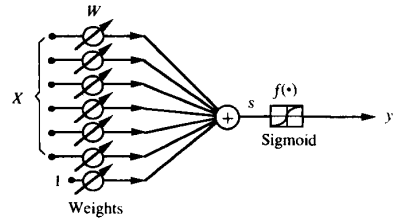


Fig. 1. Adaline with sigmoid.

A good measure of the Adaline's performance is the mean-squared error **J**, where $E(\cdot)$ denotes an expectation over all available $(X, d(X))$ pairs.

$$\mathbf{J} = E(\text{error}^2) \tag{4}$$

$$= E(d(X) - y(X))^2 \tag{5}$$

$$= E\left(d(x) - f\left(\sum_{i=0}^{n-1} w_i x_i\right)\right)^2 \tag{6}$$

By applying gradient descent [1]–[3], the algorithm to adjust W to minimize **J** turns out to be the following, where $f'(s)$ is the derivative of the function $f(s)$.

$$w_{i,\text{new}} = w_{i,\text{old}} + 2\mu\delta x_i \tag{7}$$

$$\delta = (d(X) - y(X))\, f'(s(X)) \tag{8}$$

The designer chooses μ, which affects the speed of convergence and stability of the weights during training. The value δ can be thought of as an "equivalent error" and would be equal to the error $d(X) - y(X)$ if $f(s)$ were the identity function. In this case, Eqs. (7) and (8) would be the same as the 1959 least-mean-squares (LMS) algorithm of Widrow and Hoff [1] and Widrow and Stearns [3].

The preceding algorithm is applied many times with many different $(X, d(X))$ pairs until the weights converge to a minimum of the objective function **J**.

Back-Propagation Algorithm

In this paper, Adalines are connected together to form what is known as a *layered feedforward neural network*, shown in Fig. 2. A layer of Adalines is created by connecting a number of Adalines to the same input vector. Many layers can then be cascaded, with outputs of one layer connected to the inputs of the next layer, to form a

0272-1708/90/0400-0018 $01.00 © 1990 IEEE

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.

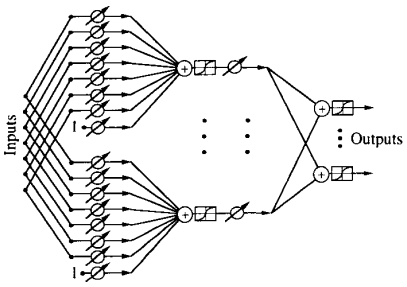


Fig. 2. Two-layer feedforward neural network.

network. It has been proven that a network consisting of only two layers of Adalines can implement any nonlinear function X, $d(X)$ given enough Adalines in the first layer (the layer closest to the input). The idea is that each Adaline in the first layer can take a small piece of the function relating X to $d(X)$ and make a linear approximation to that piece. The second layer then adds the pieces together to form the complete approximation to the desired function. A proof of this is given in [4]. [Note that $d(X)$ can be vector valued since the network can have more than one output.] Despite this theoretical result, networks of many more layers than two are being used. They offer a variety of convergence properties, robustness, and generalization characteristics (an ability to respond correctly to inputs that were not trained in) that can be quite different from those obtainable with a two-layer network.

The algorithm used to train layered neural networks is known as *back-propagation* [2], [5], [6]. This algorithm converges to a set of weights that minimizes the mean-square error

$$\mathbf{J} = E(\|d(X) - y(X)\|^2) \qquad (9)$$

where $y(X)$ is the output vector of the last layer of the network. Just as in the case of the single Adaline, it is convenient to define "equivalent error" for each Adaline in the network. For Adaline m in the output layer, the equivalent error is the following, where y_m is the output of Adaline m, δ_j is the equivalent error of the jth Adaline, j indexes the set of all Adalines that have inputs connected to Adaline m's output, and w_{jm} is the weight of the connection from Adaline m's output to Adaline j's input.

$$\delta_m = (d_m(X) - y_m(X))\, f'(s_m(X)) \qquad (10)$$

For Adaline m in one of the other layers, the equivalent error is

$$\delta_m = f'(s_m(X)) \sum_j \delta_j w_{jm} \qquad (11)$$

Each weight is updated using the same equation as for the single Adaline case, where i ranges over the inputs of Adaline m.

$$w_{mi,\text{new}} = w_{mi,\text{old}} + 2\mu\delta_m x_k \qquad (12)$$

Note that this is called the back-propagation algorithm because the equivalent error is computed for the output layer using Eq. (10), and then propagated backward through the layers toward the input layer using Eq. (11). As the equivalent error is computed during the backward propagation, the weights are updated using Eq. (12).

Layered neural networks adapted by means of the back-propagation algorithm are powerful tools for pattern recognition, associative memory, and adaptive filtering. In this paper, adaptive neural networks will be used to solve nonlinear adaptive control problems that are very difficult to solve with conventional methods.

Control Problem

The standard representation of a finite-dimensional discrete-time plant is shown in Fig. 3. The vector u_k represents the inputs to the plant at time k and the vector z_k represents the state of the plant at time k. The function $A(z_k, u_k)$ maps the current inputs and state into the next state. When the plant is linear, the usual state equation holds, where F and G are matrices.

$$z_{k+1} = A(z_k, u_k) = Fz_k + Gu_k \qquad (13)$$

The function $A(z_k, u_k)$ would be nonlinear for a nonlinear plant.

A common problem in control is to provide the correct input vector to drive a nonlinear plant from an initial state to a subsequent desired state z_d. The typical approach used in solving this problem involves linearizing the plant around a number of operating points, building linear state-space models of the plant at these operating points, and then building a controller. For nonlinear plants, this approach is usually computationally intensive and requires considerable design effort.

In this paper, the objective is to train a controller—in this case, a neural network—to produce the correct signal u_k to drive the plant to the desired state z_d given the current state of the plant z_k (Fig. 3). Each value of u_k over time plays a part in determining the state of the plant. Knowing the desired state, however, does not easily yield information about the values of u_k that would be required to achieve it.

A number of different approaches for training a controller have been described in the literature. They include reinforcement learning [7]–[9], inverse control [10], [11], and optimal control [11]. The architecture and training algorithm presented in this paper are novel in that they require little guidance from the designer to solve the control problem. This approach uses neural networks in optimal control by training the controller to maximize a performance function. The approach is different from [11] in that the plant can be an unknown plant and plant identification is a part of the algorithm. A similar approach has been used by Widrow and Stearns [3], Widrow [10], and Jordan [12].

Training Algorithm

Plant Identification—Training the Plant Emulator

Before training the neural net controller, a separate neural net is trained to behave like the plant. Specifically, the neural net is trained to emulate $A(z_k, u_k)$. Training the emulator is similar to plant identification in control theory, except that the plant identification here (Fig. 4) is done automatically by a neural network capable of modeling nonlinear plants.

In this paper, we assume that the states of the plant are directly observable without noise. A neural net with as many outputs as there are states, and as many inputs as there

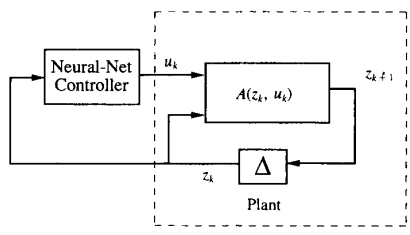


Fig. 3. Plant and controller.

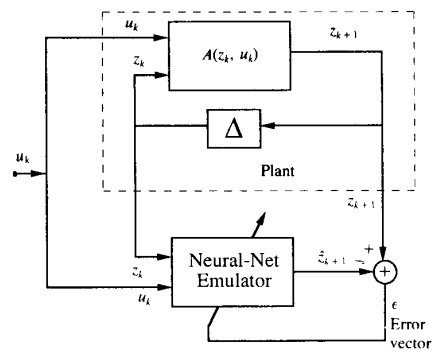


Fig. 4. Training the neural net plant emulator.

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.

are states plus plant inputs, is created. The number of layers in the neural net and the number of nodes in each layer presently are determined empirically since they depend on the degree of nonlinearity of the plant.

In Fig. 4, the training process begins with the plant in an initial state. The plant inputs are generated randomly. At time k, the input of the neural net is set equal to the current state of the plant z_k and the plant input u_k. The neural net is trained by back-propagation [Eqs. (10)–(12)] to predict the next state of the plant, with the value of the next state of the plant z_{k+1} used as the desired response during training. This process is roughly analogous to the steps that would be taken by a human designer to identify the plant. In this case, however, the plant identification is done automatically by a neural network.

Training the Neural Network Controller

Given that the emulator now closely matches the plant dynamics, we use it for the purpose of training the controller. The controller learns to drive the plant emulator from an initial state z_0 to the desired state z_d in K time steps. Learning takes place during many trials or runs, each starting from an initial state and terminating at a final state z_K. The objective of the learning process is to find a set of controller weights that minimizes the error function $\mathbf{J}$, where $\mathbf{J}$ is averaged over the set of initial states z_0.

$$\mathbf{J} = E(\|z_d - z_K\|^2) \tag{14}$$

The training process for the controller is illustrated in Fig. 5. The training process starts with the neural net plant emulator set in a random initial state z_0. Because the neural net controller initially is untrained, it will output an erroneous control signal u_0 to the plant emulator and to the plant itself. The plant emulator will then move to the next state z_1, and this process continues for K time steps. At this point, the plant is at the state z_K. (Note that the number of time steps K needs to be determined by the designer.)

We now would like to modify the weights in the controller network so that the square error $(z_d - z_K)^2$ will be less at the end of the next run. To train the controller, we need to know the error in the controller output u_k for each time step k. Unfortunately, only the error in the final plant state, $(z_d - z_K)$, is available. However, because the plant emulator is a neural network, we can back-propagate the final plant error $(z_d - z_K)$ through the plant emulator using Eqs. (10) and (11) to get an equivalent error for the controller in the Kth stage. This error then can be used to train the controller by using Eqs. (11) and (12). The emulator in a sense translates the error in the final plant state to the error in the controller output. The real plant cannot be used here because the error cannot be propagated through it. This is why the neural network emulator is needed. The error continues to be back-propagated through all K stages of the run using Eq. (11), and the controller's weight change is computed for each stage. The weight changes from all the stages obtained from the back-propagation algorithm are added together and then added to the controller's weights. This completes the training for one run.

The algorithm described would require saving all the weight changes so that they can be added to the original weights at the end of the run. In practice, for simplicity's sake, the weight changes are added immediately to the weights as they are computed. This does not significantly affect the final result since the weight changes are small and do not affect the controller's weights very much after one run. It is their accumulated effects over a large number of runs that improve the controller's performance.

Figure 5 represents the controller training process. For clarity, the details of error back-propagation are not illustrated there, but are described above and are represented algebraically by Eqs. (10)–(12). Because the training algorithm is essentially an implementation of gradient descent, local minima in the error function may yield suboptimal results. In practice, however, a good solution is almost always achieved by using a large number of Adalines in the hidden layers of the neural networks.

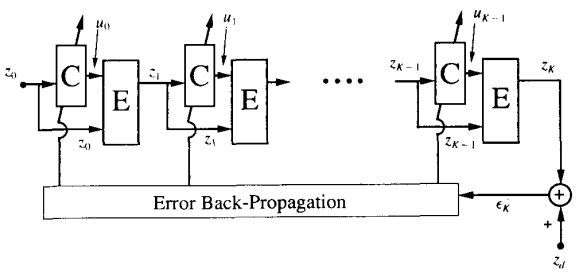


Fig. 5. Training the controller with back-propagation (C = controller, E = emulator).

An Example: Truck Backer-Upper

Backing a trailer truck to a loading dock is a difficult exercise for all but the most skilled truck drivers. Anyone who has tried to back up a house trailer or a boat trailer will realize this. Normal driving instincts lead to erroneous movements, and a great deal of practice is required to develop the requisite skills.

When watching a truck driver backing toward a loading dock, one often observes the driver backing, going forward, backing again, going forward, etc., and finally backing up to the desired position along the dock. The forward and backward movements help to position the trailer for successful backing up to the dock. A more difficult backing up sequence would only allow backing, with no forward movements permitted. The specific problem treated in this example is that of the design by self-learning of a nonlinear controller to control the steering of a trailer truck while backing up to a loading dock from any initial position. Only backing up is allowed. Computer simulation of the truck and its controller has demonstrated that the algorithm described earlier can train a controller to control the truck very well. An experimental two-layer neural controller containing 25 adaptive neural units in the first layer and one unit in the second layer has exhibited exquisite backing up control. The trailer truck can be straight or initially "jackknifed" and aimed in many different directions, toward and away from the dock, but as long as there is sufficient clearance, the controller appears to be capable of finding a solution.

Figure 6 shows a computer-screen image of the truck, the trailer, and the loading dock. The critical state variables representing the position of the truck and that of the loading dock are θ_{cab}, the angle of the cab, θ_{trailer}, the angle of the trailer, and x_{trailer} and y_{trailer}, the Cartesian position of the rear of the center of the trailer. The definition of the state variables is illustrated in Fig. 6.

The truck is placed at some initial position and is backed up while being steered by the controller. The run ends when the truck comes to the dock. The goal is to cause the back of the trailer to be parallel to the loading dock, i.e., to make θ_{trailer} go to zero and to have the point $(x_{\text{trailer}}, y_{\text{trailer}})$ be aligned as closely as possible with the point $(x_{\text{dock}}, y_{\text{dock}})$. The final cab angle is unimportant.

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.

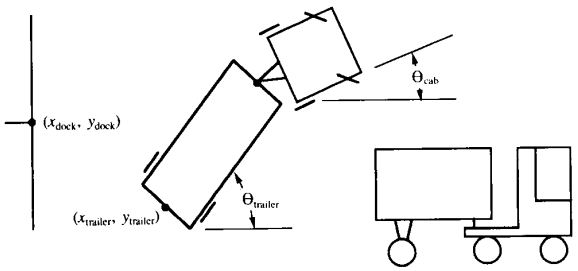


Fig. 6. Truck, trailer, and loading dock.

The controller will learn to achieve these objectives by adapting its weights to minimize the objective function **J**, where **J** is averaged over all training runs.

$$\mathbf{J} = E(\alpha_1(x_{dock} - x_{trailer})^2 + \alpha_2(y_{dock} - y_{trailer})^2 + \alpha_3(0 - \theta_{trailer})^2) \quad (15)$$

The constants α_1, α_2, and α_3 are chosen by the designer to weigh the importance of each error component.

Training

As described in the previous section, the learning process for the truck backer-upper controller involves two stages. The first stage trains a neural network to be an emulator of the truck and trailer kinematics. The second stage enables the neural-network controller to learn to control the truck by using the emulator as a guide. The control process consists of feeding the state vector z_k to the controller, which, in turn, provides a steering signal u_k between -1 (hard right) and $+1$ (hard left) to the truck (k is the time index). At each time step, the truck backs up by a fixed small distance. The next state is determined by the present state and the steering signal, which is fixed during the time step.

The process used to train the emulator is shown in Fig. 4. The emulator used in this example is a two-layer network with 25 Adalines in the first layer and four Adalines in the second layer. A suitable architecture for this network was determined by experiment. There is no theory for this yet. Experience shows that the choice of network architecture is important but a range of variation is permissible. The emulator network has five inputs corresponding to the four state variables x_k and the steering signal u_k, and four outputs corresponding to the next four state variables z_{k+1}.

During training, the truck backs up randomly, going through many cycles with randomly selected steering signals. The emulator learns to generate the next positional state vector when given the present state vector and the steering signal. This is done for a wide variety of positional states and steering angles. The two-layer emulator is adapted by means of the back-propagation algorithm. By this process, the emulator "gets the feel" of how the trailer and truck behave. Once the emulator is trained, then it can be used to train the controller.

Refer to Fig. 7. The identical blocks labeled C represent the controller net. The identical blocks labeled T represent the truck and trailer emulator. Let the weights of C be chosen at random initially. Let the truck back up. The initial state vector z_0 is fed to C, whose output sets the steering angle of the truck. The backing up cycle proceeds with the truck backing a small fixed distance so that the truck and trailer soon arrive at the next state z_1. With C remaining fixed, a new steering angle is computed for state z_1, and the truck backs up a small fixed distance once again. The backing up sequence continues until the truck hits something and stops. The final state z_K is compared with the desired final state (the rear of the trailer parallel to the dock with proper positional alignment) to obtain the final state error vector ϵ_K. (Note that, in reality, there is only one controller C. Figure 7 shows multiple copies of C for the purpose of explanation.) The error vector contains four elements, which are the errors of interest in $z_{trailer}$, $y_{trailer}$, $\theta_{trailer}$, and θ_{cab} and are used to adapt the controller C. The final angle of the cab, θ_{cab}, does not matter and so the element of the error vector due to θ_{cab} is set to zero. Each element of the error vector is also weighted by the corresponding α_i of Eq. (15).

The method of adapting the controller C is illustrated in Fig. 7. The final state error vector ϵ_K is used to adapt the blocks labeled C, which are maintained identical to each other throughout the adaptive process. The controller C is a two-layer neural network. The first layer has the six state variables as inputs, and this layer contains 25 adaptive Adaline units. The second, or output, layer has one adaptive Adaline unit and produces the steering signal as its output. All of the Adaline units have sigmoidal activation functions.

The controller C is adapted as described in the previous section. The weights of C are chosen initially at random. The initial position of the truck is chosen at random. The truck backs up, undergoing many individual back-up moves, until it comes to the dock. The final error is then computed and used by back-propagation to adapt the controller. The error is used to update the weights as it is back-propagated through the network. This way, the controller is adapted to minimize the sum of the squares of the components of the error vector using the method of steepest descent. The entire process is repeated by placing the truck and trailer in another initial position and allowing it to back up until it stops. Once again, the controller weights are adapted. And so on.

The controller and the emulator are two-layered neural networks each containing 25 hidden units. Thus, each stage of Fig. 7 amounts to a four-layer neural network. The entire process of going from an initial state to the final state can be seen from Fig. 7 to be analogous to a neural network having a number of layers equal to four times the

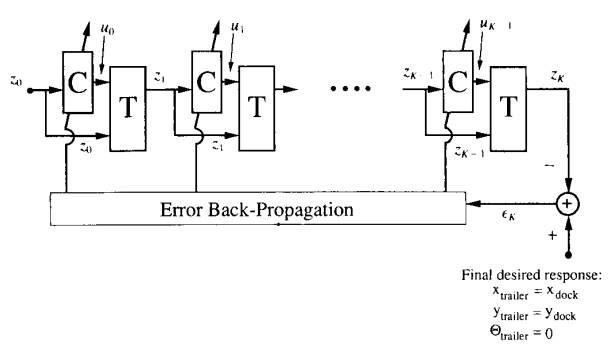


Fig. 7. Training the truck controller with back-propagation (C = controller, T = truck emulator).

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.

number of backing up steps when going from the initial state to the final state. The number of steps K varies, of course, with the initial position of the truck and trailer relative to the position of the target, the loading dock.

The diagram of Fig. 7 was simplified for clarity of presentation. The output error actually back-propagates through the T-blocks and C-blocks. Thus, the error used to adapt each of the C-blocks does originate from the output error ϵ_K, but travels through the proper back-propagation paths. For purposes of back-propagation of the error, the T-blocks are the truck emulator. However, the actual truck kinematics are used when sensing the error ϵ_K itself.

The training of the controller was divided into several "lessons." In the beginning, the controller was trained with the truck initially set to points very near the dock and the trailer pointing at the dock. Once the controller was proficient at working with these initial positions, the problem was made harder by starting the truck farther away from the dock and at increasingly difficult angles. This way, the controller learned to do easy problems first and more difficult problems after it mastered the easy ones. There were 16 lessons in all. In the easiest lesson, the trailer was set about half a truck length from the dock in the x direction pointing at the dock, and the cab at a random angle of ± 30 deg. In the last and most difficult lesson, the rear of the trailer was set randomly between one and two truck lengths from the dock in the x direction and ± 1 truck length from the dock in the y direction. The cab and trailer angles were set to be the same, at a random angle of ± 90 deg. (Note that uniform distributions were used to generate the random parameters.) The controller was trained for about 1000 truck backups per lesson during the early lessons, and 2000 truck backups per lesson during the last few. It took about 20,000 backups to train the controller.

Results

The controller learned to control the truck very well with the preceding training process. Near the end of the last lesson, the root-mean-square (rms) error of y_{trailer} was about 3 percent of a truck length. The rms error of θ_{trailer} was about 7 deg. There is no error in x_{trailer} since a truck backup is stopped when $x_{\text{trailer}} = x_{\text{dock}}$. One may, of course, trade off the error in y_{trailer} with the error in θ_{trailer} by giving them different weights in the objective function during training.

After training, the controller's weights were fixed. The truck and trailer were placed in a variety of initial positions, and backing up was done successfully in each case. A back-up run when using the trained controller is demonstrated in Fig. 8. Initial and final states are shown on the computer screen displays, and the backing up trajectory is illustrated by the time-lapse plot. The trained controller was capable of controlling the truck from initial positions it had never seen. For example, the controller was trained with the cab and trailer placed at angles of ± 90 deg, but was capable of backing up the truck with the cab and trailer placed at any angle provided that there was enough distance between the truck and the dock.

A More Sophisticated Objective Function

The above-described truck controller was trained to minimize only the final state error. One can also train it to minimize total path length or control energy in addition to the final state error. For example, the objective function to minimize control energy is the following, with $\mathbf{J}$ averaged over all training trials.

$$\begin{aligned}\mathbf{J} = E\Big[&\alpha_1(x_{\text{dock}} - x_{\text{trailer}})^2 \\ &+ \alpha_2(y_{\text{dock}} - y_{\text{trailer}})^2 \\ &+ \alpha_3(\theta_{\text{dock}} - \theta_{\text{trailer}})^2 \\ &+ \alpha_4 \sum_{k=0}^{K-1} u_k^2\Big] \qquad (16)\end{aligned}$$

A simple change is made to the algorithm to minimize this objective function. In the original algorithm, the equivalent error for the controller at each time step k is computed during the backward pass of the back-propagation algorithm. It is easy to show that control energy can be minimized by adding $-\alpha_4 u_k$ to the equivalent error of the controller at each time step. The modified equivalent error is then back-propagated through the controller to update the controller's weights as earlier. This change makes sense, since using $-\alpha_4 u_k$ as an error in u_k causes the controller to learn to make u_k smaller in magnitude.

Training the controller to minimize control energy would cause it to drive the truck to the dock with as little steering as possible. An example with the controller trained in this manner is shown in Fig. 9. This example uses the same truck and trailer initial position as with the example of Fig. 8. Note that the path of the truck controlled by the new controller contains fewer sharp turns. Of course, the final state error increases somewhat because of the new control objective.

Summary

The truck emulator in the form of a two-layer neural network was able to represent the trailer and truck when jackknifed, in line, or in any condition in between. Nonlinearity in the emulator was essential for accurate modeling of the kinematics. The angle between the truck and the trailer was not small and thus $\sin\theta$ could not be represented approximately as θ. Controlling the nonlinear kinematics of the truck and trailer required a nonlinear controller, implemented by another two-layer neural network. Self-learning processes were used to determine the parameters of both the emulator and the controller. Thousands of backups were required to train these networks, requiring several hours on a workstation. Without the learning process, however, substantial amounts of human effort and design time would have been required to devise the controller.

The truck "backer-upper" learns to solve sequential decision problems. The control decisions made early in the backing up pro-

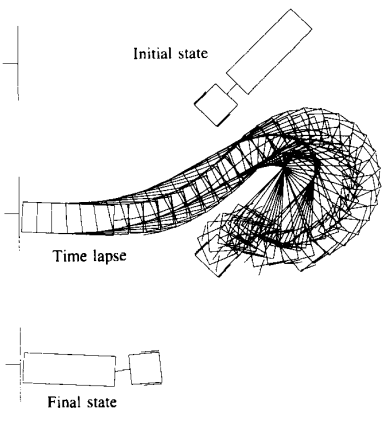


Fig. 8. A backing up example.

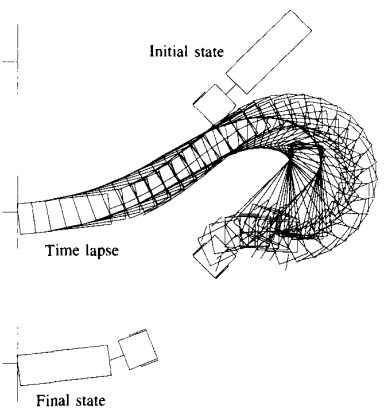


Fig. 9. Backing up while minimizing control energy.

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.

cess have substantial effects on final results. Early moves may not always be in a direction to reduce error, but they position the truck and trailer for ultimate success. In many respects, the truck backer-upper learns a control strategy similar to a dynamic programming problem solution. The learning is done in a layered neural network. Connecting signals from one layer to another corresponds to the idea that the final state of a given backing up cycle is the same as the initial state of the next backing up cycle.

Future research will be concerned with

- Determination of complexity of the emulator as related to the complexity of the system being controlled.
- Determination of the complexity of the controller as related to the complexity of the emulator.
- Determination of the convergence and rate of learning for the emulator and controller.
- Proof of robustness of the control scheme.
- Analytic derivation of the nonlinear controller for the truck backer-upper, and comparison with the self-learned controller.
- Relearning in the presence of movable obstacles.
- Exploration of other areas of application for self-learning neural networks.

Acknowledgments

This research was sponsored by the SDIO Innovative Science and Technology Office and managed by ONR under Contract N00014-86-K-0718, by the Department of the Army Belvoir RD&E Center under Contract DAAK70-89-K-0001, by NASA Ames under Contract NCA2-389, by the Rome Air Development Center under Subcontract E-21-T22-S1 with Georgia Institute of Technology, and by grants from the Thomson CSF Company and the Lockheed Missiles and Space Company.

This material is based on work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] B. Widrow and M. E. Hoff, Jr., "Adaptive Switching Circuits," in *1960 IRE WESTCON Conv. Record, Part 4*, pp. 96–104, 1960.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," in D. E. Rumelhart and J. L. McClelland, Eds., *Parallel Distributed Processing*, vol. 1, chap. 8, Cambridge, MA: MIT Press, 1986.

[3] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[4] B. Irie and S. Miyake, "Capabilities of Three-Layered Perceptrons," *Proc. IEEE Intl. Conf. Neural Networks*, pp. I–641, 1988.

[5] D. B. Parker, "Learning Logic," Tech. Rept. TR-47, Center for Comput. Res. Econ. and Manage., Massachusetts Institute of Technology, Cambridge, MA, 1985.

[6] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. Thesis, Harvard Univ., Cambridge, MA, Aug. 1974.

[7] B. Widrow, N. K. Gupta, and S. Maitra, "Punish/Reward: Learning with a Critic in Adaptive Threshold Systems," *IEEE Trans. Syst., Man, Cybern.*, Sept. 1973.

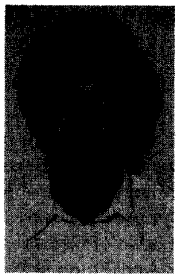
[8] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Trans. Syst., Man, Cybern.*, Sept./Oct. 1983.

[9] C. W. Anderson, "Learning to Control an Inverted Pendulum Using Neural Networks," *IEEE Contr. Syst. Mag.*, vol. 9, no. 3, Apr. 1989.

[10] B. Widrow, "Adaptive Inverse Control," in *Adaptive Systems in Control and Signal Processing 1986*, International Federation of Automatic Control, July 1986.

[11] D. Psaltis, A. Sideris, and A. A. Yamamura, "A Multilayered Neural Network Controller," *IEEE Contr. Syst. Mag.*, vol. 8, Apr. 1988.

[12] M. I. Jordan, "Supervised Learning and Systems with Excess Degrees of Freedom," COINS 88-27, Massachusetts Institute of Technology, Cambridge, MA, 1988.



Derrick H. Nguyen is a Ph.D. candidate in the Electrical Engineering Department at Stanford University. He received the B.S.E.E. degree from the California Institute of Technology in 1986 and the M.S.E.E. degree from Stanford University in 1987. His research interests include adaptive signal processing and adaptive control with neural networks. He is a Student Member of IEEE and a member of Tau Beta Pi. He received a National Science Foundation Graduate Fellowship in 1986.

Bernard Widrow is Professor of Electrical Engineering at Stanford University. Before joining the Stanford faculty in 1959, he was with the Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts. He is presently engaged in research and teaching in neural networks, pattern recognition, adaptive filtering, and adaptive control systems. He is Associate Editor of the journals *Adaptive Control and Signal Processing, Neural Networks, Information Sciences*, and *Pattern Recognition*, and coauthor with S. D. Stearns of *Adaptive Signal Processing* (Prentice-Hall). Dr. Widrow received the S.B., S.M., and Sc.D. degrees from MIT in 1951, 1953, and 1956. He is a member of the American Association of University Professors, the Pattern Recognition Society, Sigma Xi, and Tau Beta Pi. He is a Fellow of the IEEE and of the American Association for the Advancement of Science. He is President of the International Neural Network Society. Dr. Widrow received the IEEE Alexander Graham Bell Medal in 1986 for exceptional contributions to the advancement of telecommunications.

Authorized licensed use limited to: UNIVERSIDADE ESTADUAL DE CAMPINAS. Downloaded on August 5, 2009 at 10:01 from IEEE Xplore. Restrictions apply.