8 Introduction to Neural Networks and the Backpropagation Algorithm

In This Chapter

- Neural Networks from the Biological Perspective
- Single Layer Perceptions
- Multiple-Layer Networks
- Backpropagation Learning
- Evolving Game AI Behaviors
- Source Discussion
- Neurocontroller Learning
- Neurocontroller Memory
- XOR Network Learning
- Other Topics
- Other Applications
- Summary
- References
- Resources

Although various neural network topologies and learning algorithms exist, this chapter will focus on feed-forward, multilayer networks using backpropagation learning. We'll begin with a simple introduction of neural networks and their components and then discuss the learning algorithm and some problems that can arise during backpropagation learning. We'll look at an example of a simple network and walk through the backpropagation algorithm to understand its properties. Finally, we'll look at simple neural networks as a way to give life to characters within game environments.

NEURAL NETWORKS FROM THE BIOLOGICAL PERSPECTIVE

Neural networks are simple implementations of local behavior observed in our own brains. The brain is composed of neurons, which are the individual processing elements. Neurons are connected by axons that end at the neuron in a synapse. The synapse is responsible for relaying a signal to the neuron. Synapses can be either inhibitory or excitory.

The human brain contains approximately 10^{11} neurons. Each neuron connects to approximately 1,000 other neurons, except in the cerebral cortex where the density of interneuron connectivity is much higher. The structure of the brain is highly cyclic (self-referential), but it can be thought of as having a layered architecture (see Figure 8.1). In a simplified model, the input layer to the network provides our sensory inputs from the environment; the middle layer, or cerebral cortex, processes the inputs; and the output layer provides motor control back to the environment.

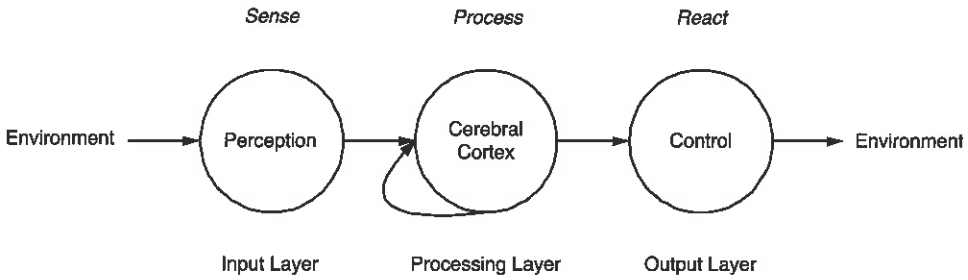


FIGURE 8.1 Layered architecture for a simple brain.

Artificial neural networks attempt to mimic the basic operation of the brain. Information is passed between the neurons, and based on the structure and synapse weights, a network behavior (or output mapping) is provided.

SINGLE LAYER PERCEPTRONS

A single layer perceptron (SLP) is a connectionist model that consists of a single processing unit. Each connection from an input to the cell includes a coefficient that represents a weighting factor. Each connection is specified by a weight w_i that specifies the influence of cell u_i on the cell. Positive weights indicate reinforcement, and negative weights indicate inhibition. These weights, along with the inputs to

the cell, determine the behavior of the network. See Figure 8.2 for a simple diagram of an SLP.

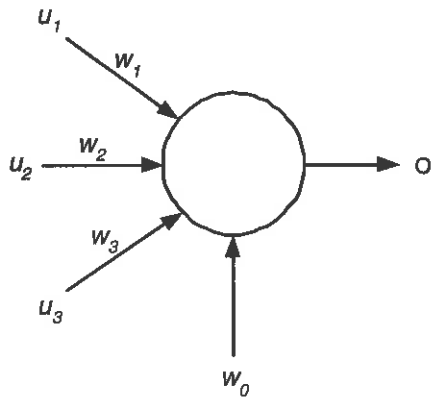


FIGURE 8.2 Single layer perceptron.

From Figure 8.2 we see that the cell includes three inputs (u_1, u_2, and u_3). A bias input (w_0) is also provided (this introduces a threshold to the neuron). Each input connection also includes a weight (w_1, w_2, and w_3). Finally, a single output, O, is provided. The cell that represents our function, defined as ($\gamma$is shown in Equation 8.1.

$$\gamma = w_0 + \sum_{i=1}^{3} u_i w_i \tag{8.1}$$

The equation shown in Equation 8.1 is simply a function that sums the products of the weights and inputs and adds in the bias. The output is then provided to an activation function, which can be defined as shown in Equation 8.2.

$$\begin{aligned} \gamma &= -1, if(\gamma <= 0) \\ \gamma &= 1, if(\gamma > 0) \end{aligned} \tag{8.2}$$

Or, simply, whenever the output is greater than 0, the output is thresholded at 1. If the output is less than or equal to 0, the output is thresholded at –1.

Modeling Boolean Expressions with SLP

Although simple, the SLP can be a powerful model. For example, the basic digital logic gates can easily be constructed, as shown in Figure 8.3.

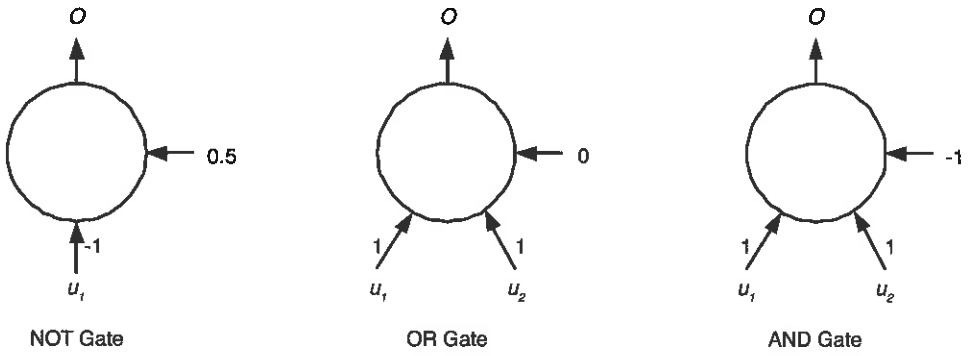


FIGURE 8.3 Logic gates built from single layer perceptrons.

Recall that an AND gate emits a 1 value if both inputs are 1; otherwise a 0 is emitted. Therefore, if both inputs are set (u vector of [1, 1]) and using the activation function from Equation 8.2 as *threshold*, we get the following result:

$$\gamma = bias + u_1w_1 + u_2w_2, \text{ or}$$
$$1 = \text{threshold}(-1 + (1 * 1) + (1 * 1))$$

Now let's try a u vector of [0, 1].

$$\gamma = bias + u_1w_1 + u_2w_2, \text{ or}$$
$$-1 = \text{threshold}(-1 + (0 * 1) + (1 * 1))$$

As both examples show, the simple perceptron model correctly implements the logical AND function (as well as the OR and NOT functions). A digital logic function that the SLP cannot model is the XOR function. The inability of the SLP to solve the XOR function is known as the *separability problem*. This particular problem was exploited by Marvin Minsky and Seymour Papert to all but destroy connectionist research in the 1960s (and support their own research in traditional symbolic AI approaches) [Minsky69].

The separability problem was resolved easily by adding one or more layers between the inputs and outputs of the neural network (see an example in Figure 8.4). This action led to the model known as multiple-layer perceptrons (MLP).

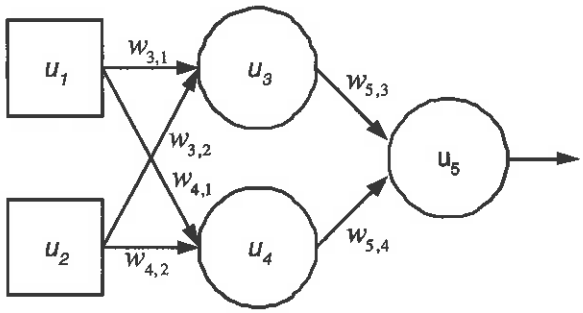


FIGURE 8.4 Multiple-layer perceptron (multilayer network).

MULTIPLE-LAYER NETWORKS

Multiple-layer networks permit more complex, nonlinear relationships of input data to output results. From Figure 8.4, we can see that the multiple-layer network consists of an input layer, an intermediate or hidden layer, and an output layer. The input layer represents the inputs to the network and is not composed of cells (neurons) in the traditional sense. The naming of cells in this example gives each cell a u_n identifier. The two input cells are named $[u_1, u_2]$, two hidden cells $[u_3, u_4]$, and one output cell $[u_5]$. Identifying connections within the network is $w_{3,1}$, which represents the weighted connection between u_3 and u_1.

Though input cells (u_1 and u_2) provide an input value to the network, hidden and output cells represent a function (recall Equation 8.1). The result of the sum of products is fed through a squashing function (typically a sigmoid), which results in the output of the cell. The sigmoid function is shown in Figure 8.5.

Now let's look at the complete picture of a neural network cell. In Figure 8.6, we see the output cell from the network shown in Figure 8.4. The output cell *u5* is fed by the two hidden cells u_3 and u_4 through weights $w_{5,3}$ and $w_{5,4}$, respectively.

One important note is that the sigmoid function should be applied to the hidden nodes in the sample network in Figure 8.6, but they are omitted in this case to illustrate the output layer processing only. The equation in Figure 8.6 illustrates the sum of products of the inputs from the hidden layer with the connection weights. The *f(x)* function represents the sigmoid applied to the result.

In a network with a hidden layer and an output layer, the hidden layer is computed first, and then the results of the hidden layer are used to compute the output layer.

Given a sufficiently large number of hidden neurons, an MLP can approximate any continuous function arbitrarily well.

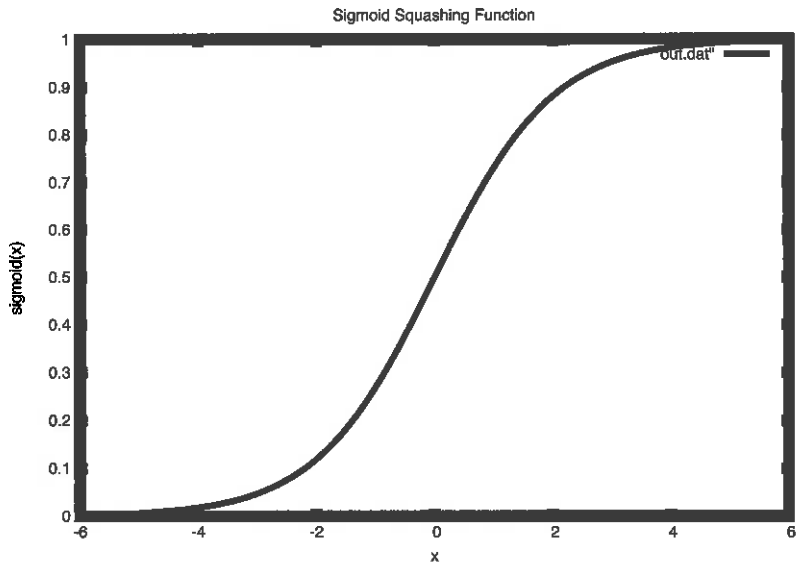


FIGURE 8.5 The sigmoid (squashing) activation function.

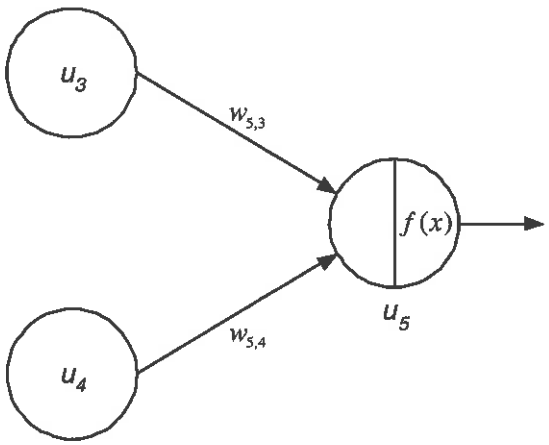


FIGURE 8.6 Hidden and output layers of a sample neural network.

BACKPROPAGATION LEARNING

Backward error propagation, or simply backpropagation, is the most popular learning algorithm for connectionist learning. As the name implies, an error in the output node is corrected by backpropagating this error by adjusting weights through

the hidden layer and back to the input layer. Although a relatively simple task, convergence can take some time, depending on the allowable error in the output.

Backpropagation Algorithm

The algorithm begins with the assignment of randomly generated weights for the feed-forward, multilayer network. The following process is then repeated until the mean-squared error (MSE) of the output is sufficiently small:

1. Take a training example E with its associated correct response C.
2. Compute the forward propagation of E through the network (compute the weighted sums of the network, S_i and the activations, u_i, for every cell).
3. Starting at the outputs, make a backward pass through the output and intermediate cells, computing the error values (Equations 8.3 and 8.4).

For the output cell $$\delta_o = (C_i - u_5)u_5(1-u_5) \tag{8.3}$$

For all hidden cells $$\delta_i = (\sum_{m:m>i} w_{m,j}\delta_o)u_i(1-u_i) \tag{8.4}$$

(Note that m denotes all cells connected to the hidden node, w is the given weight vector, and u is the activation).

4. Finally, the weights within the network are updated as follows (Equations 8.5 and 8.6):

For weights connecting hidden to output $$w^*_{i,j} = w_{i,j} + \rho\delta_o u_i \tag{8.5}$$

For weights connecting hidden to input $$w^*_{i,j} = w_{i,j} + \rho\delta_i u_i \tag{8.6}$$

where ρ represents the learning rate (or step size). This small value limits the change that may occur during each step.

The parameter ρ can be tuned to determine how quickly the backpropagation algorithm converges toward a solution. It's best to start with a small value (0.1) to test and then slowly increment.

The forward pass through the network computes the cell activations and an output. The backward pass computes the gradient (with respect to the given example). The weights are then updated so that the error is minimized for the given example. The learning rate minimizes the amount of change that may take place for

the weights. Although it may take longer for a smaller learning rate to converge, we minimize the chance of overshooting our target. If the learning rate is set too high, the network may never converge.

We'll see the actual code that's required to implement the functions described in this section, listed with the numerical steps for the backpropagation algorithm as shown.

Backpropagation Example

Now let's look at an example of backpropagation at work. Consider the network shown in Figure 8.7.

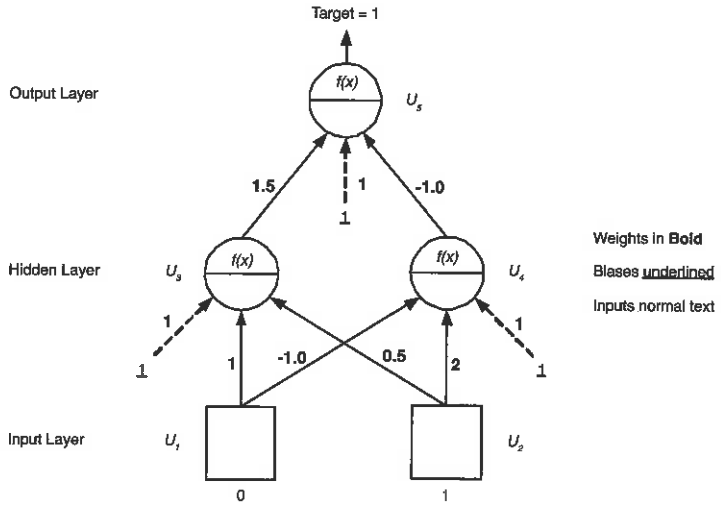


FIGURE 8.7 Numerical backpropagation example.

The Feed-Forward Pass

First, we feed forward the inputs through the network. Let's look at the values for the hidden layer.

$$u_3 = f(w_{3,1}u_1 + w_{3,2}u_2 + w_b{*}bias)$$
$$u_3 = f(1{*}0 + 0.5{*}1 + 1{*}1) = f(1.5)$$
$$u_3 = 0.81757$$

$$u_4 = f(w_{4,1}u_1 + w_{4,2}u_2 + w_b{*}bias)$$
$$u_4 = f(-1{*}0 + 2{*}1 + 1{*}1) = f(3)$$
$$u_4 = 0.952574$$

Recall that $f(x)$ is our activation function, the sigmoid function (Equation 8.7).

$$f(x) = \frac{1}{1+e^{-x}} \tag{8.7}$$

Our inputs have now been propagated to the hidden layer; the final step is to feed the hidden layer values forward to the output layer to calculate the output of the network.

$$u_5 = f(w_{5,3}u_3 + w_{5,4}u_4 + w_b{}^*{}_{\text{bias}})$$
$$u_5 = f(1.5{*}0.81757 + -1.0{*}0.952574 + 1{*}1) = f(1.2195)$$
$$u_5 = 0.78139$$

Our target for the neural network was 1.0; the actual value computed by the network was 0.78139. This isn't too bad, but by applying the backpropagation algorithm to the network, we can reduce the error.

The mean-squared error is typically used to quantify the error of the network. For a single node, this is defined as Equation 8.8.

$$err = 0.5 \times (0_{desired} - 0_{actual})^2 \tag{8.8}$$

Therefore, our error is:

$$\text{err} = 0.5 \times (1.0 - 0.78139)^2 = 0.023895$$

The Error Backward-Propagation Pass

Now let's apply backpropagation, starting with determining the error of the output node and the hidden nodes. Using Equation 8.3, we calculate the output node error.

$$\delta_o = (1.0 - 0.78139) * 0.78139 * (1.0 - 0.78139)$$
$$\delta_o = 0.0373$$

Now we calculate the error for both hidden nodes. We use the derivative of our sigmoidal equation (Equation 8.7), which is shown as Equation 8.9.

$$val = x * (1.0 - x) \tag{8.9}$$

Using Equation 8.4, we now calculate the errors for the hidden nodes.

$$\delta_{u4} = (\delta_o * w_{5,4}) * u_4 * (1.0 - u_4)$$

$$\delta_{u4} = (0.0373 * -1.0) * 0.952574 * (1.0 - 0.952574)$$
$$\delta_{u4} = -0.0016851$$

$$\delta_{u3} = (\delta_o * w_{5,3}) * u_3 * (1.0 - u_3)$$
$$\delta_{u3} = (0.0373 * 1.5) * 0.81757 * (1.0 - 0.81757)$$
$$\delta_{u3} = 0.0083449$$

Adjusting the Connection Weights

Now that we have the error values for the output and hidden layers, we can use Equations 8.5 and 8.6 to adjust the weights. We'll use a learning rate (ρ) of 0.5. First, we'll update the weights that connect our output layer to the hidden layer.

$$w^*_{i,j} = w_{i,j} + \rho\delta_0 u_i$$

$$w_{5,4} = w_{5,4} + (\rho * 0.0373 * u_4)$$
$$w_{5,4} = -1 + (0.5 * 0.0373 * 0.952574)$$
$$w_{5,4} = -0.9822$$

$$w_{5,3} = w_{5,3} + (\rho * 0.0373 * u_3)$$
$$w_{5,3} = 1.5 + (0.5 * 0.0373 * 0.81757)$$
$$w_{5,3} = 1.51525$$

Now let's update the output cell bias.

$$w_{5,b} = w_{5,b} + (\rho * 0.0373 * bias_5)$$
$$w_{5,b} = 1 + (0.5 * 0.0373 * 1)$$
$$w_{5,b} = 1.01865$$

In the case of $w_{5,4}$, the weight was decreased, whereas for $w_{5,3}$ the weight was increased. Our bias was updated for greater excitation. Now we'll show the adjustment of the hidden weights (for the input to hidden cells).

$$w_{4,2} = w_{4,2} + (\rho * -0.0016851 * u_2)$$
$$w_{4,2} = 2 + (0.5 * -0.0016851 * 1)$$
$$w_{4,2} = 1.99916$$

$$w_{4,1} = w_{4,1} + (\rho * -0.0016851 * u_1)$$
$$w_{4,1} = -1 + (0.5 * -0.0016851 * 0)$$
$$w_{4,1} = -1.0$$

$$w_{3,2} = w_{3,2} + (\rho * 0.0083449 * u_2)$$
$$w_{3,2} = 0.5 + (0.5 * 0.0083449 * 1)$$
$$w_{3,2} = 0.50417$$

$$w_{3,1} = w_{3,1} + (\rho * 0.0083449 * u_1)$$
$$w_{3,1} = 1.0 + (0.5 * 0.0083449 * 0)$$
$$w_{3,1} = 1.0$$

The final step is to update the cell biases.

$$w_{4,b} = w_{4,b} + (\rho * -0.0016851 * bias_4)$$
$$w_{4,b} = 1.0 + (0.5 * -0.0016851 * 1)$$
$$w_{4,b} = 0.99915$$

$$w_{3,b} = w_{3,b} + (\rho * 0.0083449 * bias_3)$$
$$w_{3,b} = 1.0 + (0.5 * 0.0083449 * 1)$$
$$w_{3,b} = 1.00417$$

That completes the updates of our weights for the current training example. To verify that the algorithm is genuinely reducing the error in the output, we'll run the feed-forward algorithm one more time.

$$u_3 = f(w_{3,1}u_1 + w_{3,2}u_2 + w_b * bias)$$
$$u_3 = f(1*0 + 0.50417*1 + 1.00417*1) = f(1.50834)$$
$$u_3 = 0.8188$$

$$u_4 = f(w_{4,1}u_1 + w_{4,2}u_2 + w_b * bias)$$
$$u_4 = f(-1*0 + 1.99916*1 + 0.99915*1) = f(2.99831)$$
$$u_4 = 0.952497$$

$$u_5 = f(w_{5,3}u_3 + w_{5,4}u_4 + w_b * bias)$$
$$u_5 = f(1.51525*0.8188 + -0.9822*0.952497 + 1.01865*1) = f(1.32379)$$
$$u_5 = 0.7898$$

$$err = 0.5 * (1.0 - 0.7898)^2 = 0.022$$

Recall that the initial error of this network was 0.023895. Our current error is 0.022, which means that this single iteration of the backpropagation algorithm reduced the mean-squared error by 0.001895.

EVOLVING GAME AI BEHAVIORS

The application that we'll use for backpropagation is the creation of neurocontrollers for game AI characters. A neurocontroller is the name commonly given to neural networks that are used in control applications. In this application, we'll use the neural network to select an action from an available set based on the current environment perceived by the character. The terms *character* and *agent* in the following discussion are synonyms.

The rationale for using neural networks as neurocontrollers is that practically, we can't give a game AI character a set of behaviors that covers all possible combinations of perceived environments. Therefore, we train the neural network with a limited number of examples (desired behaviors for a given environment) and then let it generalize all of the other situations to provide proper responses. The ability to generalize and provide proper responses to unknown situations is the key strength of the neurocontroller design.

Another advantage to the neurocontroller approach is that the neurocontroller is not a rigid function map from environment to action. Slight changes in the environment may elicit different responses from the neurocontroller, providing a more believable behavior from the character. Standard decision trees or finite automata lead to predictable behaviors, which are not gratifying in game play.

As shown in Figure 8.8, the environment is the character's source of information. Information is received from the environment and provided to the agent. This process of "seeing" the environment is called perception. The neurocontroller provides the capability for action selection. Finally, the character manipulates the environment through action. This action alters the environment, so the character cycles back to perception to continue the sense-react cycle.

Neurocontroller Architecture

In the prior example, a neural network with a single output was discussed in detail. For the game AI agent, we'll look at another architecture called a winner-take-all network. Winner-take-all architectures are useful where inputs must be segmented into one of several classes (see Figure 8.9).

In the winner-take-all network, the output cell with the highest weighted sum is the winner of the group and is allowed to fire. In our application, each cell represents a distinct behavior that is available to the character within the game. Example behaviors include fire-weapon, run-away, wander, and so on. The firing of a cell within the winner-take-all group causes the agent to perform the particular behavior. When the agent is allowed to perceive the environment again, the process is repeated.

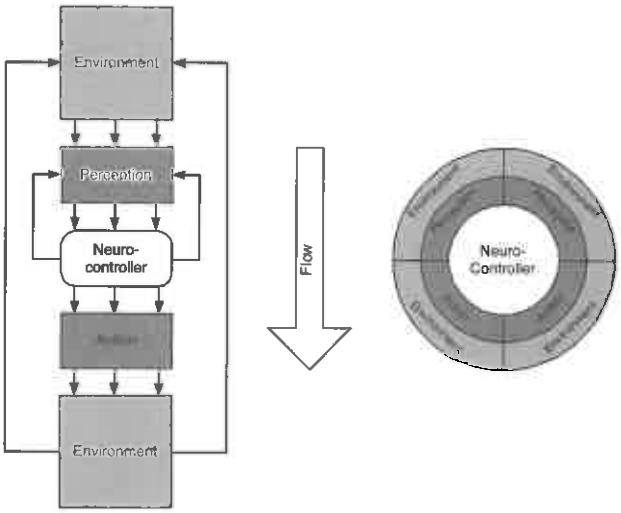


FIGURE 8.8 Example of a neurocontroller in an environment.

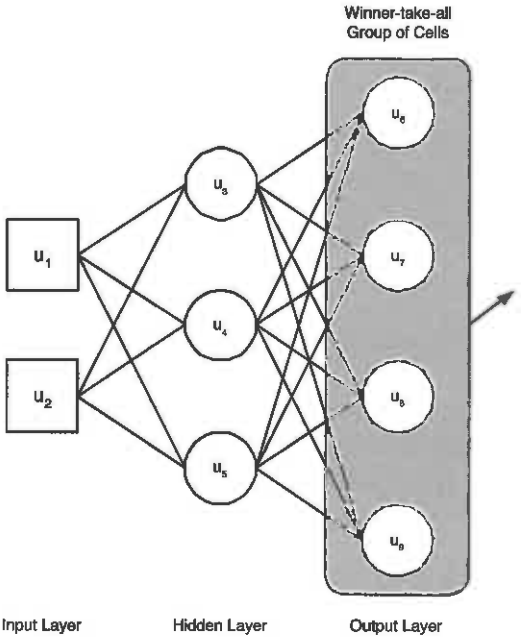


FIGURE 8.9 Winner-take-all group.

Figure 8.10 illustrates the network that was used to test the architecture and method for action-selection. The four inputs include the *health* of the character

(0-poor to 2-healthy), *has-knife* (1 if the character has a knife, 0 otherwise), *has-gun* (1 if in possession, 0 otherwise), and *enemy-present* (number of enemies in the field of view).

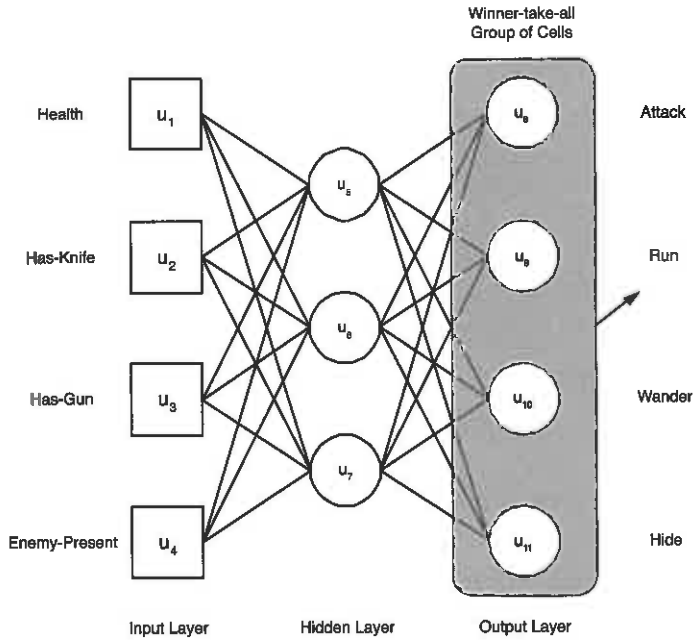


FIGURE 8.10 Game AI neurocontroller architecture for verification.

The outputs select the particular behavior that the character will take. Action *attack* causes the character to attack the peers in the field of view, *run* causes the character to flee its current position, *wander* represents random or nonrandom movement of the character within the environment, and *hide* causes the character to seek shelter. These behaviors are very high level, and it's anticipated that the behavior subsystem will take the selected action and follow through.

The architecture selected here (two hidden cells) was determined largely by trial and error. Three hidden cells could be trained for all presented examples with 100 percent accuracy. Decreasing the number of cells to two or one resulted in a network that did not successfully classify all examples.

Training the Neurocontroller

The neurocontroller within the game environment is a static element of the character's AI. The following section discusses online learning of the neurocontroller within the environment.

Training the neurocontroller consists of presenting training examples from a small set of desired behaviors and then performing the backpropagation on the network given the desired result and the actual result. For example, if the character is in possession of a gun, is healthy, and is in the presence of a single enemy, the desired behavior may be to attack. However, if the character is healthy and in possession of a knife but in the presence of two enemies, the correct behavior would be to hide.

Test Data Set

The test data set consists of a small number of desired perception-action scenarios. Because we want the neurocontroller to behave in a more lifelike fashion, we will not train the neurocontroller for every case. Therefore, the network will generalize the inputs and provide an action that should be similar to other scenarios for which it was trained. The examples used to train the network are presented in Table 8.1.

TABLE 8.1 Training Examples Used for the Neurocontroller

Health	*Has-Knife*	*Has-Gun*	*Enemies*	*Behavior*
2	0	0	0	Wander
2	0	0	1	Wander
2	0	1	1	Attack
2	0	1	2	Attack
2	1	0	2	Hide
2	1	0	1	Attack
1	0	0	0	Wander
1	0	0	1	Hide
1	0	1	1	Attack
1	0	1	2	Hide
1	1	0	2	Hide
1	1	0	1	Hide

0	0	0	0	Wander
0	0	0	1	Hide
0	0	1	1	Hide
0	0	1	2	Run
0	1	0	2	Run
0	1	0	1	Hide

The data set in Table 8.1 was presented randomly to the network in backpropagation training. The mean-squared error reduction is shown in Figure 8.11.

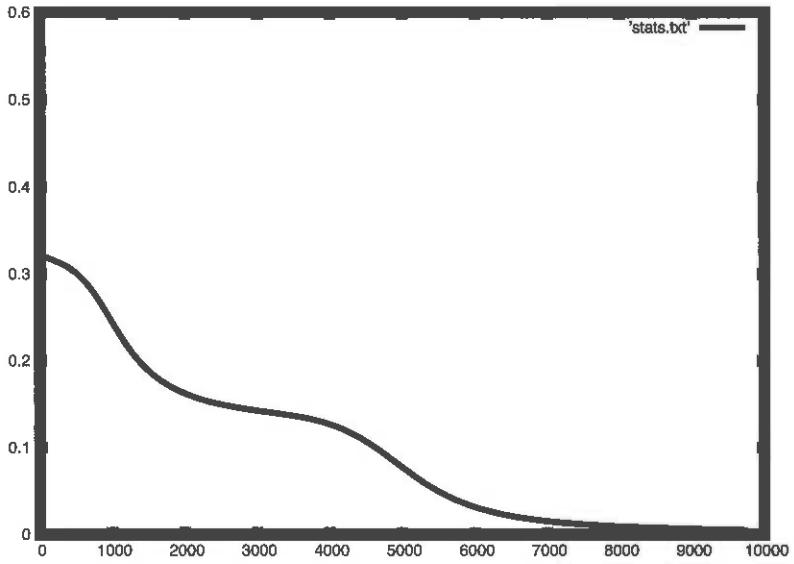


FIGURE 8.11 Sample run of the backpropagation algorithm on the neurocontroller.

In most cases, all examples shown in Table 8.1 are successfully trained. In some runs, up to two examples resulted in improper actions. Increasing the number of hidden cells resulted in perfect training. However, doing so would increase the computing resources needed to run the neurocontroller. Therefore, two hidden cells were chosen with multiple training cycles to ensure proper training. The

neurocontroller was trained offline for as many runs as were necessary for perfect training on the given set.

To test the neurocontroller, new examples were presented to the network to identify how it would react to scenarios for which it had no specific knowledge. These tests give an understanding of how well the neurocontroller can generalize and respond with desirable actions for unseen examples.

If we present the neurocontroller with a scenario in which it has full health, access to both weapons, and a single enemy present (2:1:1:1), the neurocontroller responds with an *attack* action. This response is reasonable given the situation. Now consider a scenario where the character has full health, access to a knife, and three enemies present (2:1:0:3). The neurocontroller in this scenario responds with the *hide* action. This is another reasonable scenario. See Table 8.2 for other examples.

TABLE 8.2 Test Examples Illustrating Successful Neurocontroller Generalization

Health	*Has-Knife*	*Has-Gun*	*Enemies*	*Behavior*
Good (2)	Yes	Yes	1	Attack
OK (1)	Yes	Yes	2	Hide
Poor (0)	No	No	0	Wander
Poor (0)	Yes	Yes	1	Hide
Good (2)	No	Yes	3	Hide
Good (2)	Yes	No	3	Hide
Poor (0)	Yes	No	3	Run

From Table 8.2, we can see that the neurocontroller successfully generalizes from the existing training set to provide reasonable actions for given environments. Though it wasn't trained for these examples directly, it's able to perform a desirable behavior in response to them.

The source code for the backpropagation algorithm can be found on the CD-ROM at /software/ch8/.

SOURCE DISCUSSION

Now let's look at the source code, which implements both the backpropagation algorithm for a configurable network topology as well as training and testing for the neurocontroller example.

The global variables are shown in Listing 8.1.

```
#define INPUT_NEURONS          4
#define HIDDEN_NEURONS         3
#define OUTPUT_NEURONS         4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

The weights are defined as input to hidden layer (wih) and hidden to output layer (who). The weight for the connection between u_5 and u_1 (from Figure 8.10) is an input to hidden layer weight represented by `wih[0][0]` (because u_1 is the first input cell and u_5 is the first hidden cell, based from 0). This weight is referred to as $w_{5,1}$. The weight $w_{11,7}$ (connection from u_{11} in the output layer to u_7 in the hidden layer) is `who[2][3]`. Bias weights occupy the last row in each of the tables, as identified by the +1 size for the `wih` and `who` arrays.

The activations are provided by four arrays. The `inputs` array defines the value of the input cells, the `hidden` array provides the output of the hidden cells, the `target` array represents what is desired of the network for the given inputs, and the `actual` array represents what the network actually provided.

The errors of the network are provided in two arrays. The `erro` array represents the error for each output cell. The `errh` array is the hidden cell errors.

To find random weights for initialization of the network, a number of macros are defined, shown in Listing 8.2.

LISTING 8.2 Macros and Symbolic Constants for Backpropagation

```
#define LEARN_RATE     0.2     /* Rho */

#define RAND_WEIGHT    ( ((float)rand() / (float)RAND_MAX) - 0.5)
```

```
#define getSRand()      ((float)rand() / (float)RAND_MAX)
#define getRand(x)      (int)((float)x*rand()/(RAND_MAX+1.0))

#define sqr(x)          ((x) * (x))
```

Weights are randomly selected in the range [-0.5 – 0.5]. The learning rate (rho) is defined as 0.2. The weight range and learning rate can both be adjusted depending on the problem and accuracy required.

Three support functions exist to assign random weights to the network and for algorithm support. These are shown in Listing 8.3.

LISTING 8.3 Support Functions for Backpropagation

```
void assignRandomWeights( void )
{
  int hid, inp, out;
    for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
          wih[inp][hid] = RAND_WEIGHT;
    }
  }

  for (hid = 0 ; hid < HIDDEN_NEURONS+1 ; hid++) {
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
      who[hid][out] = RAND_WEIGHT;
    }
  }

}

double sigmoid( double val )
{
  return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
  return ( val * (1.0 - val) );
}
```

Function `assignRandomWeights` randomly assigns a weight to each of the connections within the network (including all biases). The `sigmoid` function implements the squashing function used in the feed-forward phase (Equation 8.5). The

`sigmodDerivative` function implements the derivative of the `sigmoid` function and is used during error backpropagation.

It's important to choose small initial weights when creating a new neural network because it can help to avoid driving the neurons into saturation, which can limit learning.

The next function implements the feed-forward phase of the algorithm (see Listing 8.4).

LISTING 8.4 Feed-Forward Algorithm

```
void feedForward( )
{
  int inp, hid, out;
  double sum;

  /* Calculate input to hidden layer */
  for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {

    sum = 0.0;
    for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {
      sum += inputs[inp] * wih[inp][hid];
    }

    /* Add in Bias */
    sum += wih[INPUT_NEURONS][hid];

    hidden[hid] = sigmoid( sum );

  }

  /* Calculate the hidden to output layer */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {

    sum = 0.0;
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
      sum += hidden[hid] * who[hid][out];
    }

    /* Add in Bias */
    sum += who[HIDDEN_NEURONS][out];

    actual[out] = sigmoid( sum );
```

```
  }

}
```

As is illustrated in Listing 8.4, the feed-forward algorithm starts by calculating the activations of the hidden layers with the inputs from the input layer. The bias is added to the cell prior to the `sigmoid` function. The output layer is then calculated in the same manner. Note that the network may have one or more output cells. Therefore, the output cells are looped to perform all needed output activations.

The actual backpropagation algorithm is shown in Listing 8.5.

LISTING 8.5 Backpropagation Algorithm

```
void backPropagate( void )
{
  int inp, hid, out;

  /* Calculate the output layer error (step 3 for output cell)
  */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
    erro [out] = (target[out] - actual[out]) *
                 sigmoidDerivative( actual[out] );
  }

  /* Calculate the hidden layer error (step 3 for hidden cell)
  */
  for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {

    errh[hid] = 0.0;
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
      errh[hid] += erro[out] * who[hid][out];
    }

    errh[hid] *= sigmoidDerivative( hidden[hid] );

  }

  /* Update the weights for the output layer (step 4) */
  for (out = 0 ; out < OUTPUT_NEURONS ; out++) {

    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
      who[hid][out] += (LEARN_RATE * erro[out] * hidden[hid]);
```

```
        }

        /* Update the Bias */
        who[HIDDEN_NEURONS][out] += (LEARN_RATE * erro[out]);

      }

      /* Update the weights for the hidden layer (step 4) */
      for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {

        for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {
          wih[inp][hid] += (LEARN_RATE * errh[hid] * inputs[inp]);
        }

        /* Update the Bias */
        wih[INPUT_NEURONS][hid] += (LEARN_RATE * errh[hid]);

      }

    }
```

This function follows the algorithm outlined in the "Backpropagation Example" section. The error for the output cell (or cells) is calculated first using the actual output and the desired output. Next, the errors for the hidden cells are calculated. Finally, the weights for all connections are updated according to whether they're in the input-hidden layer or the hidden-output layer. An important note is that in the algorithm, we don't calculate a weight for the bias but instead simply apply the error to the bias itself. During the feed-forward algorithm, we add the bias in without applying any weight to it.

That's the algorithm for computing the output of a feed-forward, multilayer neural network and adjusting the network using the backpropagation algorithm. Now let's look at the example code that drives it for the neurocontroller example.

In Listing 8.6, a structure is defined to represent the examples for our training set. The structure defines the inputs (`health, knife, gun, enemy`) and an array for the desired output (`out` array). Listing 8.6 also contains the initialization of the training set.

LISTING 8.6 Representing the Neurocontroller Training Set

```
typedef struct {
   double health;
   double knife;
   double gun;
```

```
  double enemy;
  double out[OUTPUT_NEURONS];
} ELEMENT;

#define MAX_SAMPLES 18

/*  H    K    G    E     A    R    W    H  */
ELEMENT samples[MAX_SAMPLES] = {
  { 2.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 2.0, 0.0, 0.0, 1.0, {0.0, 0.0, 1.0, 0.0} },
  { 2.0, 0.0, 1.0, 1.0, {1.0, 0.0, 0.0, 0.0} },
  { 2.0, 0.0, 1.0, 2.0, {1.0, 0.0, 0.0, 0.0} },
  { 2.0, 1.0, 0.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 2.0, 1.0, 0.0, 1.0, {1.0, 0.0, 0.0, 0.0} },

  { 1.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 1.0, 0.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 0.0, 1.0, 1.0, {1.0, 0.0, 0.0, 0.0} },
  { 1.0, 0.0, 1.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 1.0, 0.0, 2.0, {0.0, 0.0, 0.0, 1.0} },
  { 1.0, 1.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },

  { 0.0, 0.0, 0.0, 0.0, {0.0, 0.0, 1.0, 0.0} },
  { 0.0, 0.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
  { 0.0, 0.0, 1.0, 1.0, {0.0, 0.0, 0.0, 1.0} },
  { 0.0, 0.0, 1.0, 2.0, {0.0, 1.0, 0.0, 0.0} },
  { 0.0, 1.0, 0.0, 2.0, {0.0, 1.0, 0.0, 0.0} },
  { 0.0, 1.0, 0.0, 1.0, {0.0, 0.0, 0.0, 1.0} }
};
```

Recall that the health input represents three values (2 for healthy, 1 for moderately healthy, and 0 for not healthy). Knife and gun inputs are Booleans (1 if the item is held, 0 if not), and `enemy` is the number of enemies that are seen. Actions are Booleans as well; a non-zero value represents the desired action to take.

Because we're implementing a winner-take-all network, we'll code a simple function to determine the output cell with the highest-weighted sum. This function searches the vector for the highest output and returns a string representing the action to take (see Listing 8.7). The return value can then be used as an offset into `strings` to emit the text response behavior.

LISTING 8.7 Winner-Take-All Determiner Function

```
char *strings[4]={"Attack", "Run", "Wander", "Hide"};

int action( double *vector )
{
  int index, sel;
  double max;

    sel = 0;
    max = vector[sel];

  for (index = 1 ; index < OUTPUT_NEURONS ; index++) {
    if (vector[index] > max) {
      max = vector[index]; sel = index;
    }
  }

  return( sel );
}
```

Finally, Listing 8.8 provides the `main` function for performing the training and testing of the neurocontroller.

LISTING 8.8 Sample `main` Function for Neurocontroller Training and Testing

```
int main()
{
  double err;
  int i, sample=0, iterations=0;
  int sum = 0;

  out = fopen("stats.txt", "w");

  /* Seed the random number generator */
  srand( time(NULL) );

  assignRandomWeights();

  /* Train the network */
  while (1) {

    if (++sample == MAX_SAMPLES) sample = 0;

    inputs[0] = samples[sample].health;
    inputs[1] = samples[sample].knife;
```

```
  inputs[2] = samples[sample].gun;
  inputs[3] = samples[sample].enemy;

  target[0] = samples[sample].out[0];
  target[1] = samples[sample].out[1];
  target[2] = samples[sample].out[2];
  target[3] = samples[sample].out[3];

  feedForward();

  err = 0.0;
  for (i = 0 ; i < OUTPUT_NEURONS ; i++) {
    err += sqr( (samples[sample].out[i] - actual[i]) );
  }
  err = 0.5 * err;

  fprintf(out, "%g\n", err);
  printf("mse = %g\n", err);

  if (iterations++ > 100000) break;

  backPropagate();

}

/* Test the network */
for (i = 0 ; i < MAX_SAMPLES ; i++) {

  inputs[0] = samples[i].health;
  inputs[1] = samples[i].knife;
  inputs[2] = samples[i].gun;
  inputs[3] = samples[i].enemy;

  target[0] = samples[i].out[0];
  target[1] = samples[i].out[1];
  target[2] = samples[i].out[2];
  target[3] = samples[i].out[3];

  feedForward();

  if (action(actual) != action(target)) {

    printf("%2.1g:%2.1g:%2.1g:%2.1g %s (%s)\n",
        inputs[0], inputs[1], inputs[2], inputs[3],
```

```
            strings[action(actual)], strings[action(target)]);

      } else {
        sum++;
      }

    }

    printf("Network is %g%% correct\n",
             ((float)sum / (float)MAX_SAMPLES) * 100.0);

    /* Run some tests */

    /*  Health            Knife            Gun
    Enemy */
    inputs[0] = 2.0; inputs[1] = 1.0; inputs[2] = 1.0;
    inputs[3] = 1.0;
    feedForward();
    printf("2111 Action %s\n", strings[action(actual)]);

    inputs[0] = 1.0; inputs[1] = 1.0; inputs[2] = 1.0;
    inputs[3] = 2.0;
    feedForward();
    printf("1112 Action %s\n", strings[action(actual)]);

    inputs[0] = 0.0; inputs[1] = 0.0; inputs[2] = 0.0;
    inputs[3] = 0.0;
    feedForward();
    printf("0000 Action %s\n", strings[action(actual)]);

   inputs[0] = 0.0; inputs[1] = 1.0; inputs[2] = 1.0;
    inputs[3] = 1.0;
    feedForward();
    printf("0111 Action %s\n", strings[action(actual)]);

    inputs[0] = 2.0; inputs[1] = 0.0; inputs[2] = 1.0;
    inputs[3] = 3.0;
    feedForward();
    printf("2013 Action %s\n", strings[action(actual)]);

    inputs[0] = 2.0; inputs[1] = 1.0; inputs[2] = 0.0;
    inputs[3] = 3.0;
    feedForward();
    printf("2103 Action %s\n", strings[action(actual)]);
```

```
    inputs[0] = 0.0; inputs[1] = 1.0; inputs[2] = 0.0;
    inputs[3] = 3.0;
    feedForward();
    printf("0103 Action %s\n", strings[action(actual)]);

    fclose(out);

    return 0;
}
```

After seeding the random-number generator with `srand`, the connection weights of the network are randomly generated. A `while` loop is then performed to train the network. Each of the samples is taken in order, rather than random selection of the examples. The feed-forward algorithm is performed, followed by a check of the mean-squared error. Finally, the backpropagation algorithm runs to adjust the weights in the network. After a number of iterations are performed, the loop exits and the network is tested with the training set to verify its accuracy. After the network is tested with the training set, a set of examples that were not part of the original training set are checked.

If the neural network were to be used within a game environment, the weights would be emitted to a file for later inclusion in source. The network could also be optimized at this stage to minimize the computing resources needed for its use within game environments.

NEUROCONTROLLER LEARNING

When the neurocontroller is embedded within a game environment, its training is complete and no further learning is possible. To include the ability for the character to learn, portions of backpropagation could be included with the game to adjust weights based on game play.

A simple mechanism could adjust the weights of the neurocontroller based on the last action made by the character. If the action led to negative consequences, such as the death of the character, the weights for this action given the current environment could be inhibited to make it less likely to occur in the future.

Further, all game AI characters could learn these same lessons, in a form of Lamarckian evolution (whereby children inherit the traits of their parents, which would include lessons learned). After numerous games played, the game AI characters would become slowly better at avoiding negative situations.

NEUROCONTROLLER MEMORY

The feature of memory could also be created within the neural network by creating tapped-delay lines for each of the inputs (extending the input vector from one to two dimensions). Therefore, the prior input from the environment doesn't disappear but becomes part of another input to the network. This action could be extended for a number of elements, giving the character quite a bit of history as well as many new intelligent abilities.

This method could also include feedback, so that the last actions could be fed back into the network providing the neurocontroller with action history. Additional internal feedback could include the health of the character as well as the simulated affective state (the character's emotional state). These mechanisms further the generation of rich and believable action selection.

XOR NETWORK LEARNING

As we discussed early in this chapter, an SLP cannot correctly implement an XOR gate, but an MLP can. Now let's look at one more example that breaks down backpropagation learning into its fundamental parts. We'll also use this opportunity to investigate another variant of backpropagation called batch updating, which is a way to decrease the training time for learning.

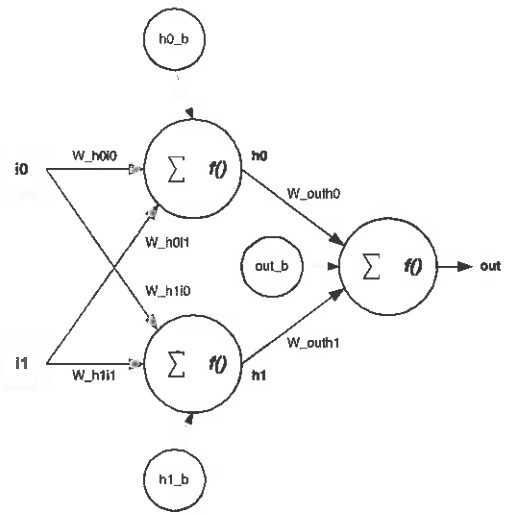
Recall that the result of the XOR function is 1 if only one input is set; if neither or both inputs are set, the output is 0. Consider the tutorial MLP shown in Figure 8.12. This MLP contains two input cells: a single hidden layer with two cells, and a single output cell. Each of the cells, their outputs, biases, and fundamental equations are shown in the figure.

Because we've already defined the equations that are central for learning in this network, we'll now review and discuss the C source code that implements backpropagation learning for this example.

Simple Backpropagation Learning

In Listing 8.9, we present a simple version of backpropogation for XOR network learning. This example is made up of five functions. The `sigmoid` and `sigmoidDerivative` functions provide their respective functions (as reviewed in the previous listings). A function to initialize the simple network is provided in `initializeNetwork`. This function assigns random weights to the network connections.

The `feedforward` function performs the feedforward pass over the network, calculating the output (`out`) given two inputs (`inp_0` and `inp_1`). The hidden cell values are also calculated, stored in global variables `hid_0` and `hid_1`.



Feedforward Pass

*h0 = f((i0 * W_h0i0) + (i1 * W_h0i1) + h0_b)*
*h1 = f((i0 * W_h1i0) + (i1 * W_h1i1) + h1_b)*
*out = f((h0 * W_outh0) + (h1 * W_out1) + out_b)*

Backpropagation Pass ó Error Determination

*out_err = (target - out) * f'(out)*
*h0_err = (out_err * W_outh0) * f'(h0)*
*h1_err = (out_err * W_outh1) * f'(h1)*

Backpropagation Pass ó Assign Error

*W_outh0 = W_outh0 + (p * out_err * h0)*
*W_outh1 = W_outh1 + (p * out_err * h1)*
*out_b = out_b + (p * out_err)*

*W_h0i0 = W_h0i0 + (p * h0_err * i0)*
*W_h0i1 = W_h0i1 + (p * h0_err * i1)*
*h0_b = h0_b + (p * h0_err)*

*W_h1i0 = W_h1i0 + (p * h1_err * i0)*
*W_h1i1 = W_h1i1 + (p * h1_err * i1)*
*h1_b = h1_b + (p * h1_err)*

f(x) = (1.0 / (1.0 + exp(-x)))
*f'(x) = (1.0 * (1.0 - x))*

FIGURE 8.12 MLP network for XOR learning.

The backpropagation pass is performed in the backpropagate function. Using the previously calculated values in the feed-forward pass, the error values are calculated and the internal weights adjusted.

Finally, the `main` function implements the training loop, randomly taking an item from the data set, testing it (via the `feedforward` function), and then adjusting the internal weights using the `backpropagate` function. At the end of the `main` function, each element of the data set is tested and the result emitted to identify how the MLP classified the dataset.

The source code for the XOR backpropagation example can be found on the CD-ROM at /software/ch8/mlp-bp.c.

LISTING 8.9 XOR Backpropogation Learning Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

```
double inp_0, inp_1;
double hid_0, hid_1;
double out;
double target;

double w_hid0inp0, w_hid0inp1, w_hid1inp0, w_hid1inp1;
double w_outh0, w_outh1;
double h0_bias, h1_bias, out_bias;

double rho = 3.0;
#define NUM_EPOCHS      10000

#define getRand(x)
           (int)((float)(x)*rand()/(RAND_MAX+1.0))
#define RAND_WEIGHT     (((double)rand() / (double)RAND_MAX))
#define sqr(x)          ((x) * (x))

#define HIGH    0.9
#define LOW     0.1
#define MID     (LOW + ((HIGH-LOW)/2.0))

double sigmoid( double val )
{
  return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
  return (val * (1.0 - val));
}

void initializeNetwork( void )
{
  w_hid0inp0 = RAND_WEIGHT; w_hid0inp1 = RAND_WEIGHT;
  w_hid1inp0 = RAND_WEIGHT; w_hid1inp1 = RAND_WEIGHT;
  w_outh0 = RAND_WEIGHT;    w_outh1 = RAND_WEIGHT;

  h0_bias = RAND_WEIGHT; h1_bias = RAND_WEIGHT;
  out_bias = RAND_WEIGHT;

  return;
}

void feedforward( void )
{
```

```
  /* First, calculate hidden node 0 */
  hid_0 = sigmoid( (inp_0 * w_hid0inp0) +
                   (inp_1 * w_hid0inp1) + h0_bias );

  /* Next, calculate hidden node 1 */
  hid_1 = sigmoid( (inp_0 * w_hid1inp0) +
                    (inp_1 * w_hid1inp1) + h1_bias );

  /* Finally, calculate the output node */
  out = sigmoid( ((hid_0 * w_outh0) + (hid_1 * w_outh1) +
                 out_bias) );

  return;
}

void backpropagate( void )
{
  double out_err, hid0_err, hid1_err;

  /* Calculate output layer error */
  out_err = (target - out) * sigmoidDerivative(out);

  /* Calculate the hidden layer error */
  hid0_err = (out_err * w_outh0) * sigmoidDerivative(hid_0);
  hid1_err = (out_err * w_outh1) * sigmoidDerivative(hid_1);

  /* Updated the output layer weights (and bias) */
  w_outh0  += (rho * out_err * hid_0);
  w_outh1  += (rho * out_err * hid_1);
  out_bias += (rho * out_err);

  /* Update weights for hidden node 0 and bias */
  w_hid0inp0 += ((rho * hid0_err) * inp_0);
  w_hid0inp1 += ((rho * hid0_err) * inp_1);
  h0_bias += (rho * hid0_err);

  /* Update weights for hidden node 1 and bias */
  w_hid1inp0 += ((rho * hid1_err) * inp_0);
  w_hid1inp1 += ((rho * hid1_err) * inp_1);
  h1_bias += (rho * hid1_err);

  return;
}
```

```
typedef struct {
  double inp0;
  double inp1;
  double out;
} dataset_t;

/* Xor dataset */
dataset_t dataset[4] = {
  {  LOW,  LOW,  LOW},
  {  LOW, HIGH, HIGH},
  { HIGH,  LOW, HIGH},
  { HIGH, HIGH,  LOW}
};

int main()
{
  double error;
  int epoch, i;

  srand(time(NULL));

  initializeNetwork();

  for (epoch = 0 ; epoch < NUM_EPOCHS ; epoch++) {

    i = getRand(4);
    inp_0 = dataset[i].inp0;
    inp_1 = dataset[i].inp1;
    target = dataset[i].out;

    feedforward();
    backpropagate();

    error = 0.5 * (sqr(target - out));
    printf("%lg\n", error);

  }

  printf("Testing...\n");

  /* Test the network */
  for (i = 0 ; i < 4 ; i++) {

    inp_0 = dataset[i].inp0;
```

```
    inp_1 = dataset[i].inp1;
    target = dataset[i].out;
    feedforward();

    printf("test %lg/%lg = %lg\n", inp_0, inp_1, out);

    out = (out >= MID) ? HIGH : LOW;

    if (out == target) printf("Success\n");
    else printf("Failed\n");

  }

  return 0;
}
```

Simple Backpropagation Learning with Batch Updating

Now let's take our example from Listing 8.9 and apply a technique called batch updating (or periodic updating). In standard backpropagation learning, we provide a test example, feed-forward to generate the result, identify the error and backpropagate, and repeat. Batch updating differs from the standard model in that it accumulates the weight updates for a number of backpropagation epochs and then applies them in a batch. This method has the effect of softening the effects of orthogonal changes to the network. For example, consider two training examples, one that affects weights in a negative way and another that affects them in a positive way. With batch updating, the overall effects of the weight changes are lessened due to their accumulation and batch update.

The significant change to the batch update version shown in Listing 8.10 (compared that shown in Listing 8.9) is the inclusion of a `batchUpdate` function to apply the weight updates to the network. The `backpropagate` function has also been altered to change the batch version of the network weights rather than affecting them directly. For example, instead of updating the `w_hidOinpO` weight in `backpropogate()`, the `bw_hidOinpO` variable accumulates `w_hidOinpO`'s changes, which are applied later through `batchUpdate()`.

The source code for the XOR backpropagation batch update example can be found on the CD-ROM at /software/ch8/mlp-btch.c.

LISTING 8.10 XOR Backpropogation Learning Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <math.h>

double inp_0, inp_1;
double hid_0, hid_1;
double out;
double target;

double w_hid0inp0, w_hid0inp1, w_hid1inp0, w_hid1inp1;
double w_outh0, w_outh1;
double h0_bias, h1_bias, out_bias;
double bw_hid0inp0=0.0, bw_hid0inp1=0.0, bw_hid1inp0=0.0,
bw_hid1inp1=0.0;
double bw_outh0=0.0, bw_outh1=0.0;
double bh0_bias=0.0, bh1_bias=0.0, bout_bias=0.0;

double rho = 3.0;
#define NUM_EPOCHS      10000

#define getRand(x)
         (int)((float)(x)*rand()/(RAND_MAX+1.0))
#define RAND_WEIGHT     (((double)rand() / (double)RAND_MAX))
#define sqr(x)          ((x) * (x))

#define HIGH    0.9
#define LOW     0.1
#define MID     (LOW + ((HIGH-LOW)/2.0))

double sigmoid( double val )
{
  return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
  return (val * (1.0 - val));
}

void initializeNetwork( void )
{
  w_hid0inp0 = RAND_WEIGHT; w_hid0inp1 = RAND_WEIGHT;
  w_hid1inp0 = RAND_WEIGHT; w_hid1inp1 = RAND_WEIGHT;
  w_outh0 = RAND_WEIGHT;    w_outh1 = RAND_WEIGHT;

  h0_bias = RAND_WEIGHT; h1_bias = RAND_WEIGHT;
```

```
  out_bias = RAND_WEIGHT;

  return;
}

void feedforward( void )
{
  /* First, calculate hidden node 0 */
  hid_0 = sigmoid( (inp_0 * w_hid0inp0) +
                   (inp_1 * w_hid0inp1) + h0_bias );

  /* Next, calculate hidden node 1 */
  hid_1 = sigmoid( (inp_0 * w_hid1inp0) +
                    (inp_1 * w_hid1inp1) + h1_bias );

  /* Finally, calculate the output node */
  out = sigmoid(((hid_0 * w_outh0) +
                 (hid_1 * w_outh1) + out_bias));

  return;
}

void backpropagate( void )
{
  double out_err, hid0_err, hid1_err;

  /* Calculate output layer error */
  out_err = (target - out) * sigmoidDerivative(out);

  /* Calculate the hidden layer error */
  hid0_err = (out_err * w_outh0) * sigmoidDerivative(hid_0);
  hid1_err = (out_err * w_outh1) * sigmoidDerivative(hid_1);

  /* Updated the output layer weights (and bias) */
  bw_outh0  += (rho * out_err * hid_0);
  bw_outh1  += (rho * out_err * hid_1);
  bout_bias += (rho * out_err);

  /* Update weights for hidden node 0 and bias */
  bw_hid0inp0 += ((rho * hid0_err) * inp_0);
  bw_hid0inp1 += ((rho * hid0_err) * inp_1);
  bh0_bias += (rho * hid0_err);

  /* Update weights for hidden node 1 and bias */
```

```
  bw_hid1inp0 += ((rho * hid1_err) * inp_0);
  bw_hid1inp1 += ((rho * hid1_err) * inp_1);
  bh1_bias += (rho * hid1_err);

  return;
}

void batchUpdate( void )
{
  w_hid0inp0 += bw_hid0inp0; w_hid0inp1 += bw_hid0inp1;
  w_hid1inp0 += bw_hid1inp0; w_hid1inp1 += bw_hid1inp1;
  w_outh0 += bw_outh0; w_outh1 += bw_outh1;
  h0_bias += bh0_bias; h1_bias += bh1_bias;
  out_bias += bout_bias;

  bw_hid0inp0 = bw_hid0inp1 = bw_hid1inp0 = bw_hid1inp1 = 0.0;
  bw_outh0 = bw_outh1 = 0.0;
  bh0_bias = bh1_bias = bout_bias = 0.0;

  return;
}

typedef struct {
  double inp0;
  double inp1;
  double out;
} dataset_t;

/* Xor dataset */
dataset_t dataset[4] = {
  {  LOW,  LOW,  LOW},
  {  LOW, HIGH, HIGH},
  { HIGH,  LOW, HIGH},
  { HIGH, HIGH,  LOW}
};

int main()
{
  double error=0.0;
  int epoch, i;
  int count = 0;

  srand(time(NULL));
```

```
  initializeNetwork();

  for (epoch = 0 ; epoch < NUM_EPOCHS ; epoch++) {

    i = count;
    inp_0 = dataset[i].inp0;
    inp_1 = dataset[i].inp1;
    target = dataset[i].out;

    feedforward();
    backpropagate();

    error += (sqr(target - out));

    if (++count == 4) {
      printf("%lg\n", (0.25*error));
      count = 0;
      error = 0.0;
      batchUpdate();
    }

  }

  printf("Testing...\n");

  /* Test the network */
  for (i = 0 ; i < 4 ; i++) {

    inp_0 = dataset[i].inp0;
    inp_1 = dataset[i].inp1;
    target = dataset[i].out;
    feedforward();

    printf("test %lg/%lg = %lg\n", inp_0, inp_1, out);

    out = (out >= MID) ? HIGH : LOW;

    if (out == target) printf("Success\n");
    else printf("Failed\n");

  }

  return 0;
}
```

Testing with the source, the XOR network converges on a correct network much faster using batch updating than with the standard backpropagation learning algorithm. A negative effect is that overtraining can now occur, but this result can be combated in other ways (such as limiting the number of training epochs).

OTHER TOPICS

Now let's look at a number of other neural network topics such as algorithm variations and problems and solutions for network training.

Variable Learning Rate

The Variable Learning Rate algorithm can be useful to avoid getting trapped in local minima in addition to converging on a solution in a shorter amount of time. The basic algorithm is defined as follows:

- Slowly increase the learning rate (ρ).
- If the error increases, decrease the learning rate.
- Continue decreasing until stable learning resumes.
- When the error decreases, increase the learning rate.

When the learning rate is too large, the network will never stabilize. But, if it's too small, it can take an inordinate amount of time to converge on a solution (potentially overfitting). Variable learning rate can identify the proper rate of learning given the feedback of the network error.

Momentum

Another method for avoiding the local minima is the addition of a momentum term to the backpropagation algorithm. The momentum term uses a fraction of the previous weight change (which results in pushing the ascent or descent a little further). Consider Equation 8.10.

$$w = w + \Delta w_t + m\Delta w_{t-1} \tag{8.10}$$

With a small momentum value (m, between 0 and 1), we can include in the weight update a portion of the last delta weight change, which can push our weight through small local minima troughs.

Weight Decay

A useful method to avoid in overtraining the network is the use of a decay term (which can penalize large weights in a network). The decay occurs by penalizing the error in each training epoch. Once the error (e) has been applied, we decay each weight as shown in Equation 8.11, where λ is a small number between 0 and 1.

$$\omega = \omega - \lambda\omega \tag{8.11}$$

It's a good idea to alter the decay constant (λ) when considering the layer in which backpropagation is occurring. For example, decay constant *wih* is used for the input-to-hidden layer, *whh* for hidden-to-hidden, and *who* for the hidden-to-output layer. It has been shown that good generalization requires different decay constants in each of the layers.

Training Data

The topic of training data is important because it can ultimately determine whether or how fast the network converges, and how well it can generalize to other inputs.

One useful method for dealing with training data is to partition the data into two independent sets. The first set consists of training data, which is used with the backpropagation algorithm. The second set consists of test data, which is used identify how well the neural network can generalize. Periodically, the training is interrupted and the testing data used to validate the network. If the error of the testing data is less than the last training period, the network is saved and training continues until some minimum error is achieved. This process can help the network avoid local minima.

It's also best to avoid a large number of training epochs, because they can cause the network to overtrain and subsequently not generalize at all. Noise can also be added to the training data to avoid approximating the training set and improving generalization.

OTHER APPLICATIONS

The backpropagation algorithm can be used to train neural networks for various applications, including the following:

- General pattern recognition
- Fault diagnosis
- Monitoring of patients in medical settings
- Character recognition

- Data filtering
- Odor/aroma analysis
- Fraud detection

In 1989, Carnegie Mellon University created ALVINN (Autonomous Land Vehicle in a Neural Network), which successfully drove from the East Coast to the West Coast of the United States, 98 percent under autonomous control. The system was trained by watching a human driver.

SUMMARY

In this chapter, an introduction to neural networks was provided with a discussion of the backpropagation algorithm. To illustrate the algorithm, the training of game AI neurocontrollers was presented and was found to provide proper responses to unseen situations. Although the approach presented here may not be possible in all game architectures due to computational requirements, it does provide a believable character architecture including nonlinear relationships between perceived environment and action selection.

REFERENCES

[Minsky69] Minsky, Marvin, and Seymour Papert, *Perceptrons: An Introduction to Computational Geometry*, Cambridge, Mass.: MIT Press, 1969.

RESOURCES

Gallant, Stephen L., *Neural Network Learning and Expert Systems*, Cambridge, Mass.: MIT Press, 1994.

Waggoner, Ben, and Brian Speer, "Jean-Baptiste Lamarck (1744–1829)," available online at *www.ucmp.berkeley.edu/history/lamarck.html*, accessed January 17, 2003.