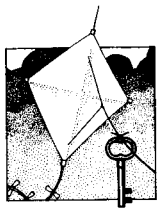




# Neurons for Computers

Artificial neural networks are computer models inspired by the structure and behavior of real neurons. Like the brain, they can recognize patterns, reorganize data and, most interestingly, learn. The artificial networks are made up of objects called units, which represent the bodies of the neurons. The units are connected by links, which act as the axons and dendrites. The link multiplies the output from a unit by a weighting factor, a value analogous to the connection strength at a synapse. The link then passes the weighted output value to another unit, which sums up the values passed to it by all other incoming links. If the total input value exceeds some threshold value, the unit fires.

Modifications in the firing pattern constitute the learning. In real neurons, learning is thought to occur in the synapses: when the connection strengths between synapses change, the firing behavior of the network changes. In artificial networks the learning occurs when the weighting factors on the links change [see "How Neural Networks Learn from Experience," by Geoffrey E. Hinton, page 144].

Artificial neural networks are made up of three types of units. The input units take in information from the outside world. The output units send out signals that are visible to the external world. The hidden units act as go-betweens from the input to the output units; they neither receive input directly from the outside nor produce a visible output.

An illustration of a simple network appears below. The numbers in the units indicate threshold values. The values along the connecting lines are the link weights. Note that there can be shortcuts: some input connections can bypass the hidden unit.

If one input unit is presented with 1 and the other with 0, then the input to the hidden unit will be $(1 \times 1) + (0 \times 1) = 1$. But because this value is less than the threshold value, the hidden unit will not fire (that is, it will have an output of 0). The values to the output unit will be $(1 \times 1) + (0 \times (-2)) + (0 \times 1) = 1$, which is greater than the threshold value of 0.5. The output unit will then fire.

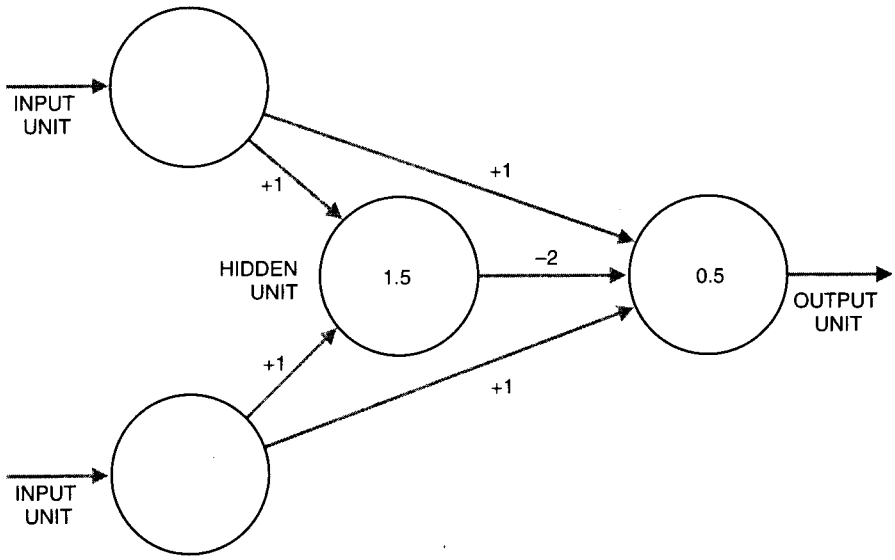
You may recognize that the logic involved mimics an "exclusive or" (XOR) gate. Specifically, if just one input unit is given the value of 1, then the network will produce an output value of 1. Otherwise, it will produce 0—that is, it will not fire. You will find that the other three possible input patterns lead to the appropriate output results for an XOR gate as well.

That you can model an XOR gate with an artificial neural network is not particularly exciting. By carefully selecting the weights and thresholds of a suitably complex network, you can model any logic function. What is exciting about such networks is that you do not have to choose the weights and thresholds. For our XOR gate, we could have started with any weight and threshold. Then, by repeatedly being shown the patterns of inputs and outputs, the network would learn the weights necessary to implement the XOR gate. Even more interesting, the network can generalize what it has learned. For large data sets, it can recognize patterns that it has not seen before.

To create such a network, we need to make a few modifications to our artificial neurons. First, the mathematics used to train the network becomes much easier if we do not have to change both weights and thresholds during the learning. This is actually simple to do. Any unit with a positive threshold, $T$, and a certain number, $n$, of incoming links can be replaced by a unit with a 0 threshold and $n + 1$ links, where the extra link has the weight of $-T$ and comes from a unit that always fires (that is, produces an output of 1). This trick is called biasing. Artificial neural networks often introduce the bias unit, which is usually connected to every unit in the network, to change the threshold into a weight. Representing the output of a unit as a sigmoid function also simplifies the math. The sigmoid is just a smooth approximation of a threshold function.

Although there are several ways to train a network, I chose a method from a particular class of algorithms called supervised learning (other types include unsupervised learning and reinforcement learning). In supervised learning the weights of the network are adjusted in a manner that causes the actual outputs of the net to move closer to the desired outputs. One of the most successful of such training methods is the back-propagation algorithm.

Hinton's article gives a more detailed




***ARTIFICIAL NEURAL NETWORK shown here represents an "exclusive or" (XOR) gate: the output unit fires only if an input unit is presented with a 1. The numbers along the connections are weights, and those inside the units are threshold values.***

DREW VAN CAMP is a computer programmer and researcher at the University of Toronto. He specializes in creating neural-network simulators for various applications.

description of the technique. Briefly, for those who have a tolerance for elementary calculus, you are taking the derivative of a function in order to find the direction that minimizes the network's error. The function most commonly used for the error is the sum of the squared errors of the output units. The box to the right lists the equations and the steps needed to implement the back-propagation algorithm.

My first implementation was an auto-encoder network. Such devices compress patterns. For example, think of a network that has four input units, two hidden units and four output units (a 4-2-4 encoder). When any input pattern is presented to the network, all four input values must be combined into the two hidden units. These two values must then be able to reproduce the original pattern of the four values in the output units. The entire four-valued input pattern thus must be encoded in the two hidden units.

In the 4-2-4 encoder, note that every input is connected to every hidden unit. Similarly, every hidden unit is connected to every output unit. The bias unit is connected to all the hidden and output units (there is no need to connect the bias unit to the input units, as they will just reproduce the inputs). Each of the four patterns I wanted to train the network to recognize has one unit on and the three remaining units off.

Before I started training, I assigned random values between -1 and 1 for the initial weights. When I presented the four patterns to this untrained network, the total error (the value of $E$) varied between three and five (depending on the initial weights).

Training the network consisted of repeatedly presenting the patterns to the network and updating the weights after each presentation. I decided that the network reached an acceptable performance level when the total error of the four patterns was less than 0.1. Using this criterion, I found it took between 800 and 2,000 presentations before the network learned the task, depending on the step size, $\delta$, for changing the weights. Indeed, one of the trickiest parts in training the network was choosing $\delta$. If the value chosen is too low, the network will take a long time to converge. If the value is too high, the network behavior will be unstable and may never converge. Through trial and error, I came to use a value of 0.5. Although the network did not train quickly, it never blew up.

Once the network was trained, I decided to see what kind of internal patterns it used to encode the presented patterns. The hidden units contained

## Training an Artificial Neural Network

A network learns by successive repetitions of a problem, making smaller errors with each iteration. The most commonly used function for the error is the sum of the squared errors of the output units:

$$E = {}^1/_2 \Sigma ( y_i - d_i )^2$$

The variable $d_i$ is the desired output of unit $i$, and $y_i$ is its actual output, where $y_i$ is the sigmoid function $1/(1 + e^{-x})$. To minimize the error, take the derivative of the error with respect to $w_{ij}$, the weight between units $i$ and $j$:

$$\frac{\partial E}{\partial w_{ij}} = y_i \, y_j (1 - y_j) \beta_j$$

where $\beta_j = ( y_j - d_j )$ for output units and $\beta_j = \Sigma_k w_{jk} y_k (1 - y_k) \beta_k$ for hidden units ($k$ represents the number of units in the next layer that unit $j$ is connected to). Note that $y_j (1 - y_j)$ is the derivative of the sigmoid function [*see illustration on next page*].

As you can see, the error can be calculated directly for the links going into the output units. For hidden units, however, the derivative depends on values calculated at all the layers that come after it. That is, the value $\beta$ must be back-propagated through the network to calculate the derivatives.

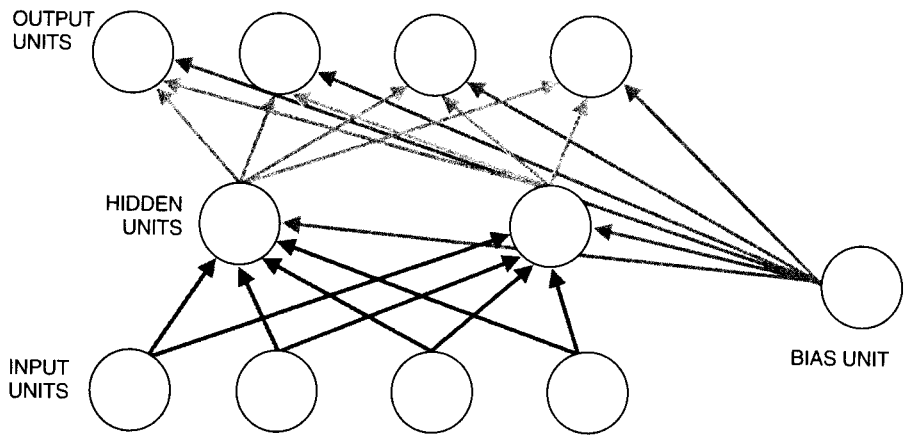
Using these equations, we can state the back-propagation algorithm as follows:

- Choose a step size, $\delta$ (used to update the weights).
- Until the network is trained,
  - For each sample pattern,
    - Do a forward pass through the net, producing an output pattern.
    - For all output units, calculate $\beta_j = (y_j - d_j)$.
    - For all other units (from last layer to first), calculate $\beta$ using the calculation from the layer after it:

$$\beta_j = \Sigma_k \, w_{jk} y_k (1 - y_k) \beta_k .$$

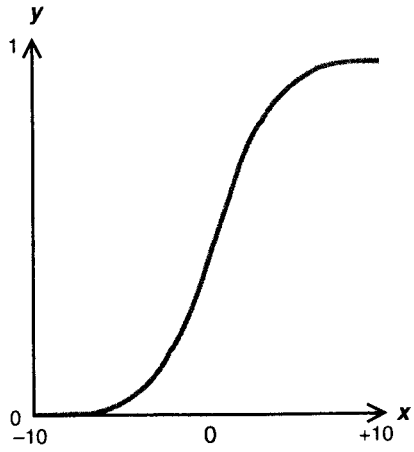
  - For all weights in the network, change the weight by

$$\Delta w_{ij} = -\delta y_i \, y_j (1 - y_j) \beta_j .$$




| INPUT PATTERNS | | | | HIDDEN UNIT OUTPUTS | | ACTUAL OUTPUTS | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0.03 | 0.97 | 0.91 | 0.10 | 0.00 | 0.07 |
| 0 | 1 | 0 | 0 | 0.98 | 0.96 | 0.07 | 0.88 | 0.06 | 0.00 |
| 0 | 0 | 1 | 0 | 0.91 | 0.02 | 0.00 | 0.10 | 0.91 | 0.06 |
| 0 | 0 | 0 | 1 | 0.03 | 0.07 | 0.07 | 0.00 | 0.09 | 0.90 |

***4-2-4 ENCODER NETWORK compresses patterns. After the training period, the network accurately reproduced the input by representing the patterns essentially as binary code (as revealed by the hidden units).***




*SIGMOID FUNCTION, defined by the equation* $y = 1/(1 + e^{-x})$*, produces almost the same output as an ordinary threshold (a step function) but is mathematically simpler. Its derivative is* $dy/dx = y(1 - y)$.

values such as 0.03, 0.97, 0.98 and 0.07. Essentially, the network had developed a binary code for the patterns.

Finally, I decided to build a network that could learn arithmetic—specifically, the addition of two three-bit binary numbers. The input to the network would be the six binary digits specifying the two numbers to add. The output would be the four digits to which they summed (four digits are necessary in case a number carries over).

Before building this network, I generated a training set of 64 patterns. To be sure that the representations of the numbers in three-bit form were correct, I wrote a short program that did the calculations and output all the training patterns. Next, I had to determine the architecture of the network. Once again, I decided to use a single hidden layer, but I was not sure how many units it should contain. That is because of a common problem that arises during the training of artificial neural networks: their performance on the training data will always continue to improve (assuming a stable procedure to update the weights). But if you give patterns that are not from the training set, you will notice that the network's performance on these patterns will first improve, then get worse.

The process is called overfitting. It occurs after the network has learned some general rules about the data. As you train the network, it learns more and more about the anomalies in the training set. It then tries to generalize these anomalies to other data and, as a result, produces a large error.

To avoid serious overfitting, the number of weights in a network should be much less than the number of bits required to specify the desired output for all the training examples. For the binary addition task, there should be far fewer than 256 weights (64 patterns times four digits of output). Using this rule of thumb, I decided to try 15 hidden units with six input units, four output units and a bias unit, which gave 169 connections. This network took significantly longer to train than the previous networks. In fact, it took more than 30,000 iterations.

Once I knew the network could be trained, I tried something different. I removed four of the patterns from the training data, randomized the weights and retrained the network on the remaining 60 patterns. Once it was trained, I tested it on the four patterns it had never seen. It produced the correct answers for these patterns. The network had learned to do binary addition.

Many modifications would greatly enhance the network's learning rate. In particular, you could use more complex methods for choosing the direction to move the weights or use line searches to determine how large a step to take. There are also many more complex algorithms to try, including constructive algorithms that add hidden units as they train.

*For a copy of the encoder program (written in ANSI C language), send a formatted double-density disk (3 1/2 or 5 1/4 inches) with a stamped, self-addressed mailer to: The Amateur Scientist, Scientific American, 415 Madison Avenue, New York, NY 10017-1111.*

## Network Parameters and Data Structures

```
/*Network parameters*/
#define NUM_INPUT       4       /*number of input units*/
#define NUM_HIDDEN      2       /*number of hidden units*/
#define NUM_OUTPUT      4       /*number of output units*/
#define NUM_BIAS        1       /*number of bias units*/
#define STEP_SIZE       0.5     /*step size for updating weights*/

/* Network data structures: Unit, Net */
typedef struct Unit {
        double  input;          /*total input*/
        double  output;         /*total output (input through
                                   the sigmoid function)*/
        double  target;         /*target output (for output units)*/
        double  beta;           /*error derivative (backpropagated)*/

} Unit;

typedef struct Net {
        Unit    input [NUM_INPUT];      /*layer of input units*/
        Unit    hidden [NUM_HIDDEN];    /*layer of hidden units*/
        Unit    output [NUM_OUTPUT];    /*layer of output units*/
        Unit    bias [NUM_BIAS];        /*layer of bias units*/

/*Connections between layers (note order of indices)*/
        double  i2h[NUM_HIDDEN][NUM_INPUT];     /*input to hidden*/
        double  h2o[NUM_OUTPUT][NUM_HIDDEN];    /*hidden to output*/
        double  b2h[NUM_HIDDEN][NUM_BIAS];      /*bias to hidden*/
        double  b2o[NUM_OUTPUT][NUM_BIAS];      /*bias to output*/

        FILE    *fpPatterns;    /*circular file of patterns*/
        double  error;          /*network error*/

} Net;

/*Useful macros: sigmoid (derivative), random, square*/
#define sigmoid (x)             (1.0 / (1.0 + exp(-double) (x)))
#define sigmoidDerivative (x)   ((double) (x) * (1.0 - (x)))
#define random (x)              ((double)rand()/(RAND_MAX))
#define square (x)              ((x)*(x))
```

FURTHER READING

EXPLORATIONS IN PARALLEL DISTRIBUTED PROCESSING: A HANDBOOK OF MODELS, PROGRAMS, AND EXERCISES. David E. Rumelhart and James L. McClelland. The MIT Press, 1988.