

ARTIFICIAL NEURAL NETWORKS

Colin Fahey's Guide

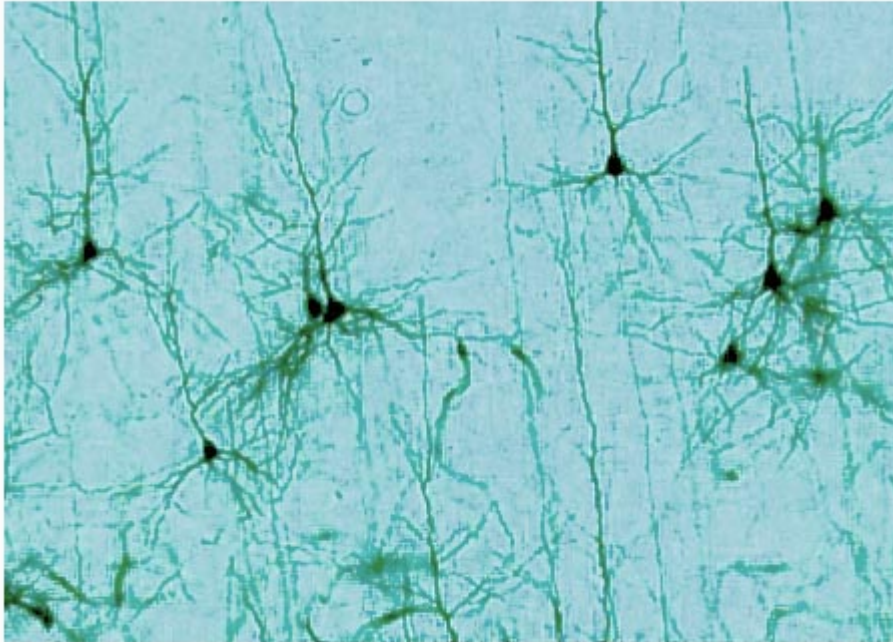


FIGURE: Network of biological neurons.

INTRODUCTION

This article describes how to implement an artificial neural network that is capable of being trained to recognize patterns.

This article discusses a "feed-forward network" and a training algorithm known as "back-propagation". The formulae and algorithms can be used to form a network with an arbitrary number of layers.

This article offers the complete C++ source code for download, featuring the demonstrations discussed in the article. The source code is public domain, platform-independent, and carefully written.

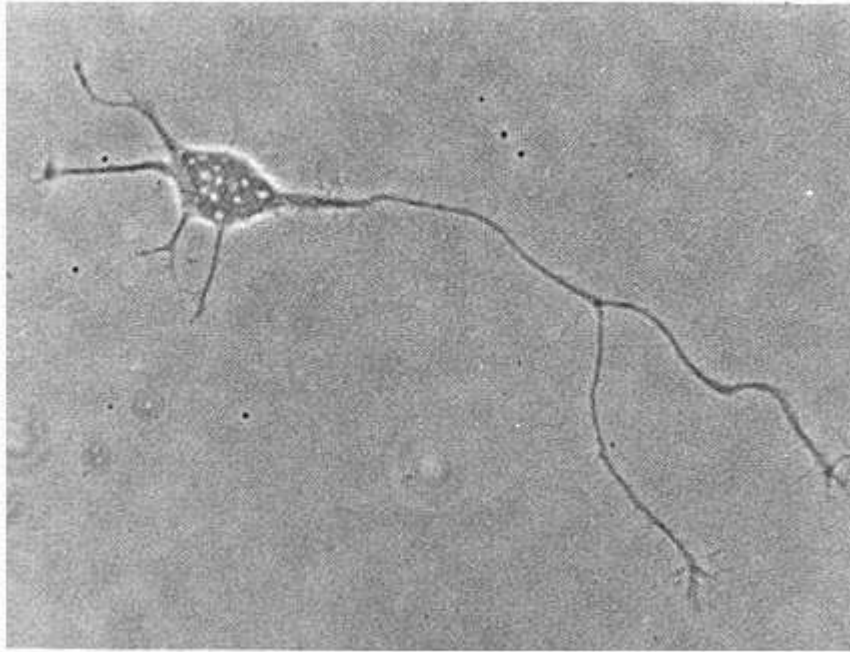


FIGURE: A biological neuron. ("multipolar" type, ~4 um cell body)

TABLE OF CONTENTS

- [1] [Introduction](#)
- [2] [Table of Contents](#)
- [3] [Biological Neuron](#)
- [4] [Artificial Neuron](#)
- [5] [Layer of Neurons](#)
- [6] [Network of Neurons](#)
- [7] [Network Simulation](#)
- [8] [Back-Propagation](#)
- [9] [Example: XOR \(Exclusive-OR\)](#)
- [10] [Example: Tic-Tac-Toe Game](#)
- [11] [Discussion about Training](#)
- [12] [DOWNLOAD Source Code](#)
- [13] [References](#)
- [14] [Disclaimer](#)
- [15] [Reader Feedback](#)
- [16] [Contact Information](#)

BIOLOGICAL NEURON

A neuron is type of cell that has the ability to receive and transmit nerve signals. Neurons are the basis of nerve systems, found in animals, birds, fish, and insects. Brains with memory and logic, and simple reflex systems, are both based on arrangements of neurons. Neurons are also used to convey signals over long distances in a creature's body, such as from sensors to the brain, or from the brain to muscles.

The behavior of a biological neuron is very complex, but the following simplified description captures the basic principle: The neuron accumulates signals received from other neurons, and if the total signal accumulation exceeds a threshold, the neuron transmits its own signals to other neurons.

The following diagram shows the basic parts of a neuron:

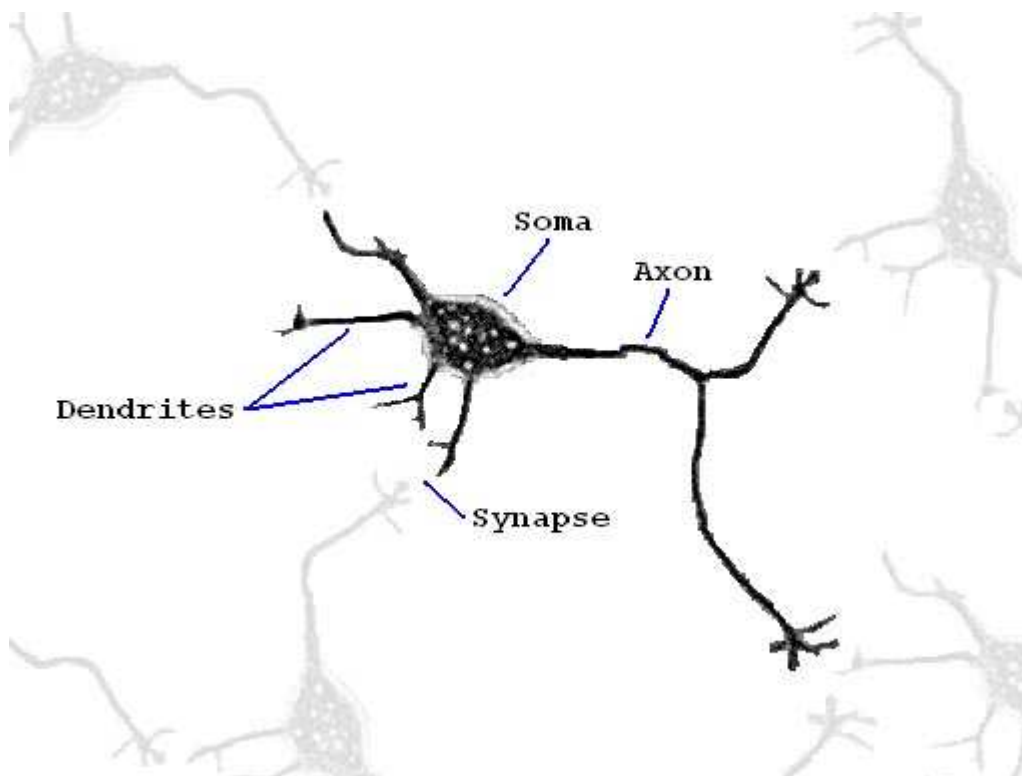


FIGURE: Basic parts of a biological neuron.

Soma	The cell body of a neuron.
Dendrites	Fibers with chemical receptors (inputs) that extend from the cell body of a neuron. A neuron typically has many dendrites, and dendrites often have many branches.
Axon	A fiber with chemical emitters (outputs) at its endpoint that extends from the cell body of the neuron. A neuron has a single axon, and the axon usually has very few branches.
Synapse	A configuration such that the axon of one neuron and the dendrites of another neuron are separated by a very small gap. In such a configuration, chemicals emitted by an axon of a neuron cross the synapse and are received by the dendrites of the other neuron. This is how neurons influence

other neurons.

FIRING:

A neuron accumulates chemical signals from its dendrites, and if the total chemical accumulation exceeds a threshold within a period of time, the neuron "fires", sending its own signal through its axon. Some neurons are capable of firing pulses on the order of 100 Hz. The signals passing through neurons involve accumulations of sodium (Na), potassium (K), and chloride (Cl) ions, and a resulting electrochemical potential (i.e., voltage). The resting voltage (-70mV) and firing voltage (+30mV) can be measured or even influenced by conventional electrical circuitry.

BRAIN NETWORK:

The human brain has roughly 10^{11} (100 billion) neurons. Each neuron in the cerebellum receives input from as many as 10^4 (10,000) synapses. Although the axon and dendrites of a neuron often extend only a few micrometers away from the cell body, some axons are on the order of a meter in length. A brain has neurons with relatively short axons grouped in areas or clusters. A brain also has bundles of neurons with relatively long axons to link areas separated by centimeters. Thus a hierarchical network of processing elements is formed.

BRAIN STATE:

The state of a network of neurons is both the way the neurons are connected and the signals at all of the synapses. It is unclear how much state information would be lost if a brain was tranquilized in to total inactivity for any amount of time. One can imagine information sustained only by signals moving through the network, and not by network connectivity itself, like cellular automata simulations like Conway's "Game of Life", simple Dynamic Random Access Memory (DRAM) chips, and echos in a chamber.

LEARNING:

Conventional learning occurs when the properties of dendrites change at a synapse to become more or less efficient at receiving chemical signals from an axon. The reasons for such changes are complicated, but the result is that a neuron requires a different combination of synapse inputs to trigger an output signal.

EXAMPLE:

The following is a voltage recording of a rat neuron firing at a rate of roughly 100 Hz when a single whisker is touched and held out of its resting position:

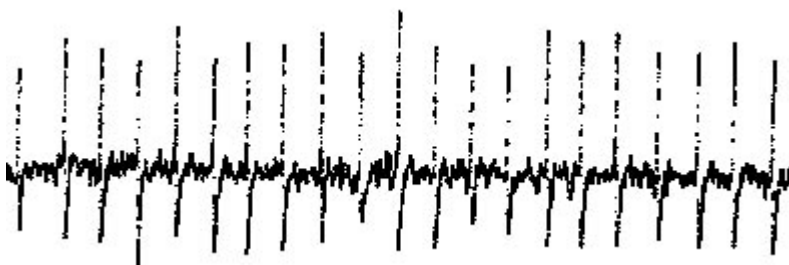


FIGURE: Rat neuron firing (100Hz) due to holding a whisker.

The following is the same signal manifested as audio:

[neuron_spikes_whisker01.wav \[16 KB\]](#)

Although the stimulus is constant, the neuron signal is rapid pulsing.

ARTIFICIAL NEURON

DEFINITION

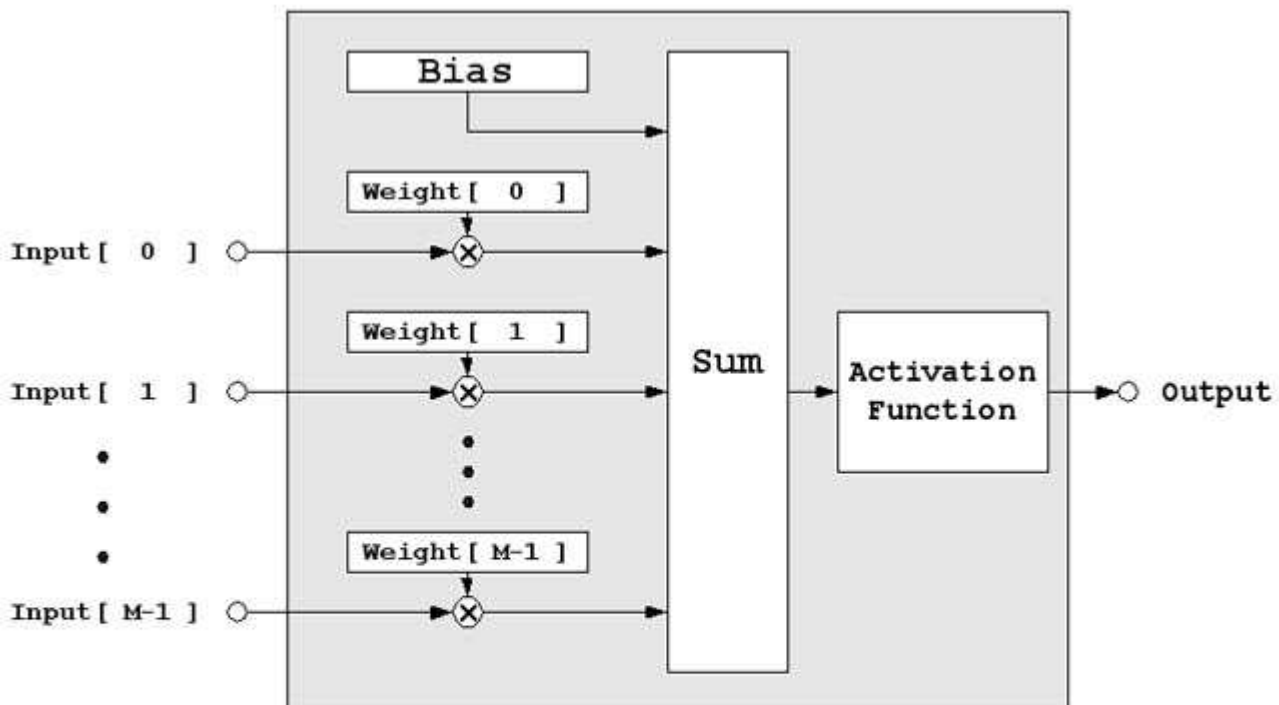
An "artificial neuron" is an algorithm or physical device that implements a mathematical model inspired by the basic behavior of a biological neuron.

Recall the simplified description of a biological neuron: The neuron accumulates signals received from other neurons, and if the total signal accumulation exceeds a threshold, the neuron transmits its own signals to other neurons.

Any mathematical model that incorporates the idea of taking multiple inputs and yielding a single output that accentuates the relative intensity of the input relative to some nominal level can be exploited for pattern recognition. Such models can be the basis of an artificial neuron. If the influence of each input can be modified, then the model can support learning.

TYPICAL MODEL

The following is a typical mathematical model implemented by an artificial neuron:



$$\text{Sum} = \text{Bias} + \sum_{i=0}^{M-1} \left(\text{Weight}[i] * \text{Input}[i] \right)$$

$$\text{Output} = \text{ActivationFunction}(\text{Sum})$$

FIGURE: Typical mathematical model implemented by an artificial neuron

ACTIVATION FUNCTION

The Activation Function accepts a value that is the weighted sum of neuron inputs and returns a value that represents the output of the neuron.

In a sense, the Activation Function determines if a neuron should "fire", and does the actual "firing" if appropriate.

Although it is possible to devise a time-dependent Activation Function, to mimic the pulsing behavior of biological neurons, it is very common for artificial neurons to have an Activation Function whose output depends only on the neuron inputs. Inputs in an artificial neuron simulation can be held constant for an arbitrarily long (or short) period of time, and the Activation Function would yield a constant output for that same period of time.

The output of the Activation Function is limited to a specific range, such as [0.0,+1.0], or [-1.0,+1.0]. This imitates biological neurons, whose outputs are limited to the the [-70,+30] mV range due to electrochemistry.

There are several candidates for the Activation Function:

(a) Unit Step Function:

=====

```
float F( float x )
{
    if (x < 0.0f) return(0.0f);
    return(1.0f);
}
```

This function should be used for regular operation of a neural network if the output is strictly Boolean. This function should NOT be used during the training of a neural network, because the only error indication we get is 0% or 100%.

(b) Clamped Linear Ramp Function:

=====

```
float F( float x )
{
    if (x < 0.0f) return(0.0f);
    if (x > 1.0f) return(1.0f);
    return( x );
}
```

This function should be used for regular operation of a neural network when the outputs should basically be a linear function of the inputs. As long as the inputs are limited to a certain range, the outputs will be a linear combination of the inputs, otherwise the outputs are clamped.

(c) Sigmoid Function:

=====

A Sigmoid Function is actually a general class of smooth functions that asymptotically approach a lower limit for input values approaching negative infinity, and asymptotically approach an upper limit for input values approaching positive infinity. One specific Sigmoid function is the "Sigmoid Logistic" function:

```
float F( float x )
{
    return( 1.0f / (1.0f + exp(-x)) );
}
```

This function should be used for training neural networks because a continuous function like this gives better feedback about the degree of error in a network. This function should also be used if the system being simulated obeys a continuous, smooth, non-linear function. However, it should be noted that it is okay to switch from a Sigmoid function to a Unit Step function or a Clamped Linear function once training is completed, if Boolean or linear outputs are desired.

The following graph shows the Sigmoid Logistic function, which is a popular choice for the Activation Function for both training and regular simulation:

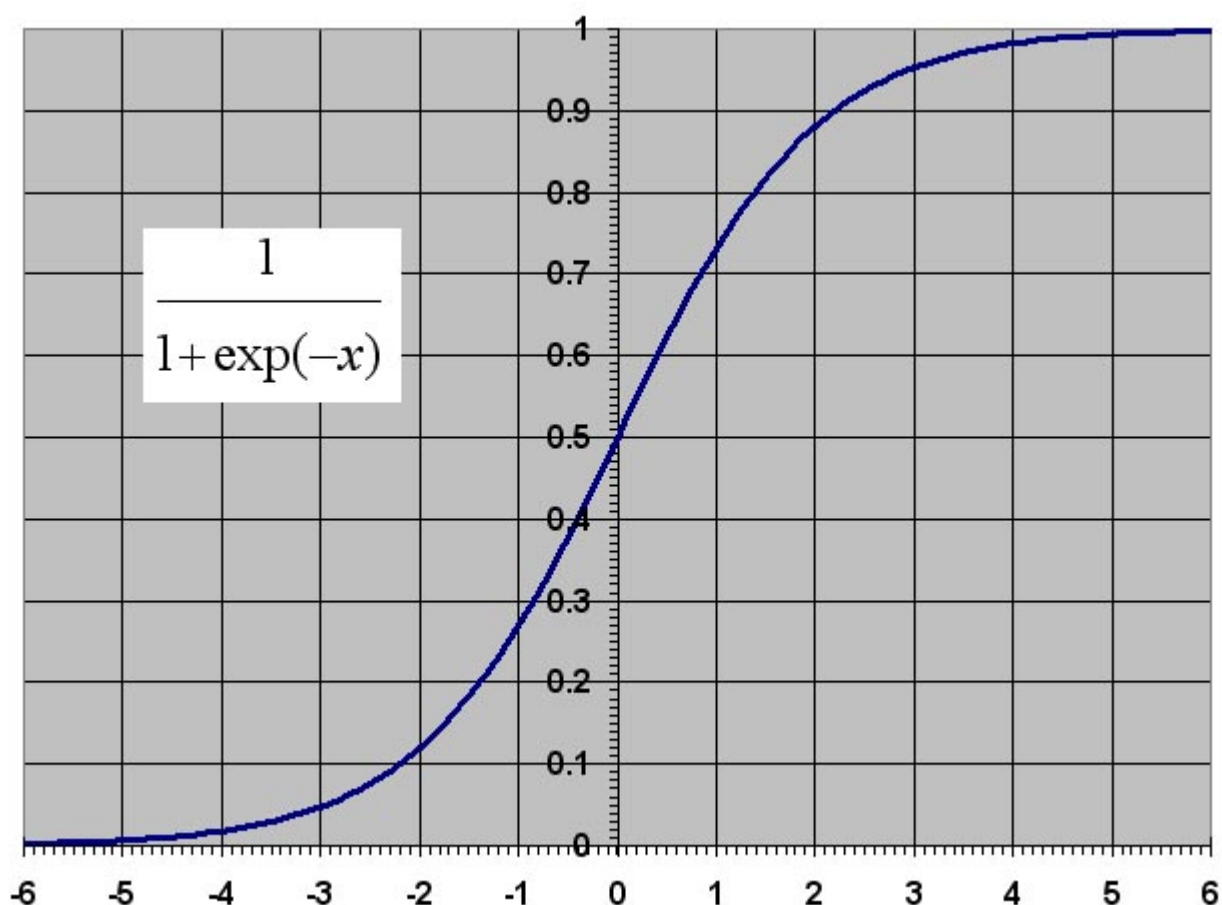


FIGURE: Sigmoid Logistic function.

ARTIFICIAL NEURON LAYER

DEFINITION

An "artificial neuron layer" is simply a set of artificial neurons with access to a single set of inputs. For simplicity, if a layer has M inputs and N outputs, we require N neurons, and each neuron in the layer must have exactly M inputs.

The following figure shows a neuron layer as a processing element with M inputs and N outputs:

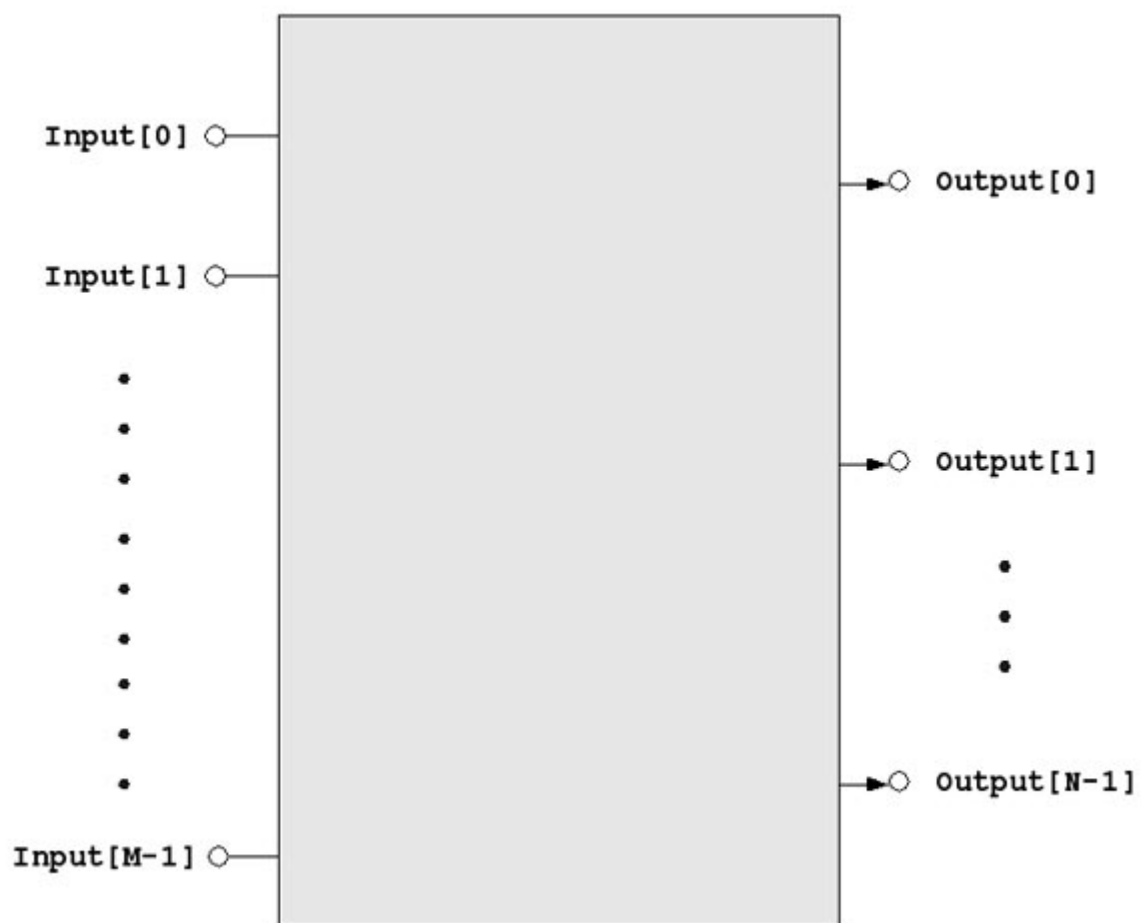


FIGURE: Neuron layer regarded as a processing element with M inputs and N outputs.

The following figure shows how a neuron layer is simply a container for neuron objects, with access to a common set of inputs:

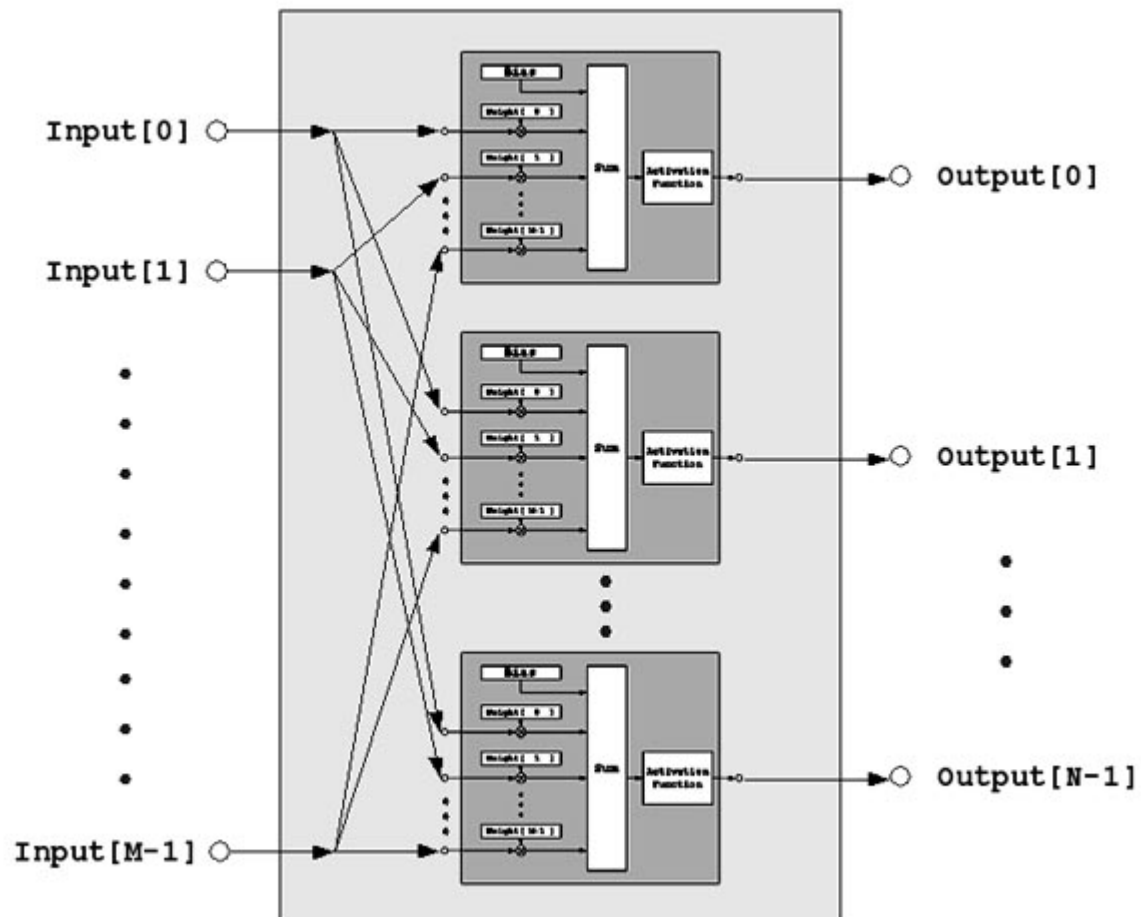


FIGURE: A neuron layer is simply a container for neuron objects.

ARTIFICIAL NEURON NETWORK

DEFINITION

Although "artificial neuron network" refers to any collection of connected neurons, let us focus on a network made up of layers. In a network made up of layers, external stimuli are accepted as input to the first layer, and the outputs of the first layer are accepted as inputs to a possible second layer. Outputs of any given layer are accepted as inputs at the next layer. The outputs of the final layer in the sequence is considered the output of the entire network.

The following figure shows a neuron network as a processing element with M inputs and N outputs:

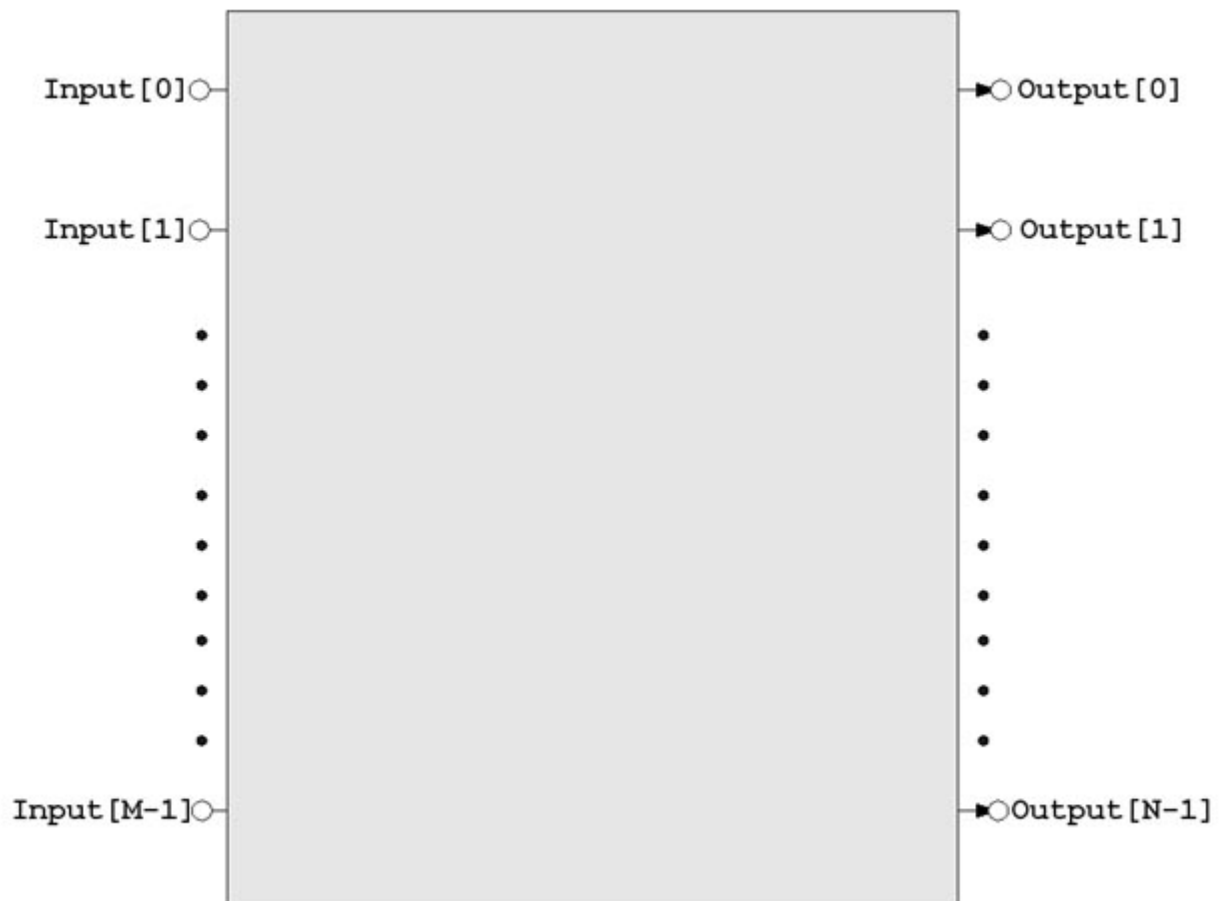


FIGURE: Neuron network regarded as a processing element with M inputs and N outputs.

The following figure shows how a neuron network is simply a container for neuron layer objects, with layers connected to their immediate neighbors in a sequence of layers:

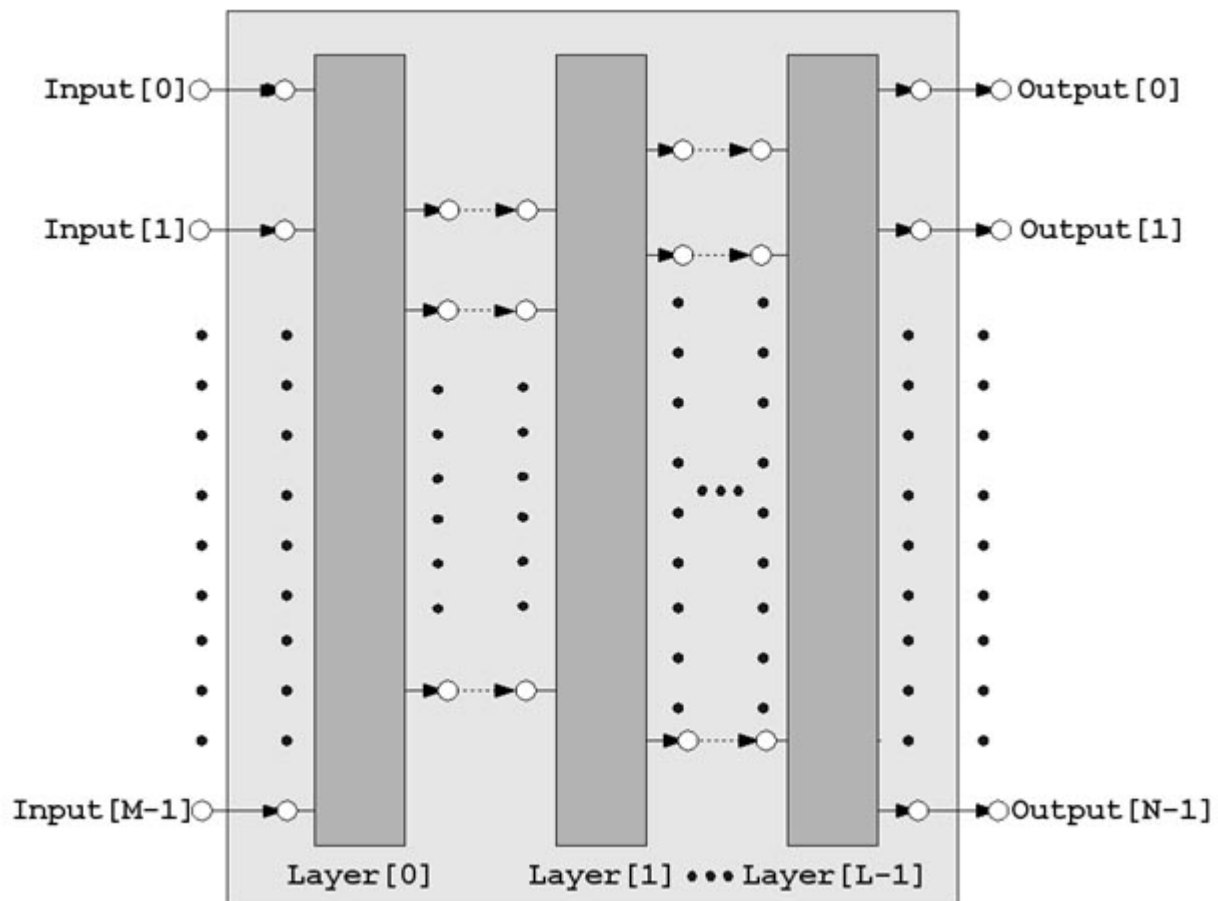


FIGURE: A neuron network is simply a container for neuron layer objects.

NETWORK SIMULATION

DEFINITION

"Network simulation" is the procedure used to process network inputs and ultimately yield network outputs.

Network simulation involves the simulation of all of its constituent neurons.

SIMULATIONS WITHOUT LOOPS OR TIME

There are many possible network configurations involving loops, and many neuron models that depend on time, but some of the most common applications of artificial neurons involve neither loops nor time.

The examples of networks presented in this article are of this simple variety. However, it should be easy for you to modify the neuron model in the source code to depend on time, and you can create networks that have some loops.

Here is the mathematical model of a single neuron as presented in the artificial neuron section:

$$\text{Sum} = \text{Bias} + \sum_{i=0}^{(M-1)} \left(\text{Weight}[i] * \text{Input}[i] \right)$$

$$\text{Output} = \text{ActivationFunction}(\text{Sum})$$

FIGURE: Formulae for simulating a single neuron.

With this neuron model, and a network without "loops", we simply start from the external inputs, compute outputs of the first layer of neurons, and supply those outputs as inputs to the next layer, compute outputs for that layer, and continue through layers of neurons until the final outputs are computed.

LOOPS

A network can have connections in the form of loops, sometimes called "cycles". For example, the output of a neuron can be connected directly to an input of that same neuron, causing "feedback". Another example is the output of neuron #1 being connected to the input of neuron #2, and the output of neuron #2 being connected to the input of neuron #1. If you can start from some point in a network and trace a path through neurons and connections, obeying the one-way flow of the signals, and eventually arrive at that same starting point, the path is a loop.

Loops introduce the interesting possibility of signals flowing around the network for indefinite periods of time. Some simple models assume that it takes a specific amount of time for signals to pass through individual neurons. In such models, signals circulate through loops with few neurons faster than signals circulate through loops with many neurons. A neuron connected to itself will have the fastest signal circulation rate. If a neuron has an input X , a weight W , a bias B , and a non-negative output Y (e.g., $0.0 \rightarrow 1.0$), then we can form an oscillator just by setting $W = (-8)$ and $B = +4$ and connecting Y to X ; each time we simulate the neuron, the signal will be toggled to the opposite state.

Networks with loops can be busy with activity even when it does not accept external signals (stimuli) as inputs. Conway's "Game of Life" cellular automata rules could be implemented in a neural network, which gives you a small hint of the kind of rich diversity of activity that can transpire in a neural network with loops. Finite state machines (FSM), oscillators, volatile memory (as opposed to learning patterns via changing weights), are made possible with looping.

If a network has loops, we cannot update any outputs until we compute ALL outputs; thus, we require a temporary buffer to store computed outputs until we compute all outputs, and then we can commit the new output values to the neurons in the network. Any scheme that updates outputs in the actual network in a progressive way, instead of an all-at-once way, introduces an arbitrary ordering in time that leads to chaos. Physics simulations involving coupled entities, such as planets orbiting a star with mutual gravitational forces between all

bodies, require the same kind of approach: compute the net forces on all bodies before updating any velocity and position.

TIME-DEPENDENCE

A simple network simulation typically involves inputs causing the desired outputs after a single simulation time step. In such a simulation, we think in terms of "number of iterations" rather than "time in seconds". There need not be any correspondance between iterations and a time scale. A system might be designed to do a network simulation (iteration) only when new input is available, which might occur at irregular intervals of time.

However, consider a mathematical model of a neuron that attempts to simulate the pulsing output aspect of a biological neuron. The pulsing might be characterized in terms of time, such as pulsing at a particular frequency or having pulses whose curve extends for a particular amount of time. We can have other time-dependent elements in a mathematical model of a neuron, such as an input accumulator whose value gets contributions from inputs but has a leak proportional to its current value. In general, we can find an electrical circuit analogy for elements that obey certain mathematical equations, and so one can regard a neuron as a circuit with resistors, capacitors, and a non-linear amplifier. Just as a circuit can exhibit complex time-dependent behavior, the output of a neuron can be regarded as a function that depends on its inputs and time in a complicated way.

BACK-PROPAGATION

DEFINITION

"Back-Propagation" is a mathematical procedure that starts with the error at the output of a neural network and propagates this error backwards through the network to yield output error values for all neurons in the network.

A common form of learning is "trial and error". A "trial" is the output of a system in response to particular stimuli. An "error" is the external reaction to the output of the system that is supplied to the system as some other kind of stimulus. A system capable of "trial and error" learning relies on receiving feedback that describes the nature and severity of mistakes. The system can use the error information to make corrections in the way it responds to that particular combination of stimuli in the future.

Back-Propagation yields neuron error values throughout a neural network. Learning occurs when neuron input weights and bias values are adjusted in an attempt to reduce the output error for the same stimuli.

It should be noted that defining a mechanism for learning implicitly defines the nature of phenomena that will frustrate learning.

BACK-PROPAGATION FORMULAE

The goal is to compute output errors for every neuron in a network. The output errors for neurons at the output layer of a network is particularly easy to compute:

$$\text{Error}[i] = (\text{Output}[i] - \text{Desired}[i])$$

FIGURE: Output error for neurons at the output layer of a network.

CAUTION: Many textbooks compute this error as "(Desired - Output)", which affects the sign of other formulae. You should be careful when comparing formulae across texts. I am strongly in favor of a convention such that: $\text{Output} = (\text{Ideal} + \text{Error})$. For example, a small positive error value means that the output will be HIGHER than the ideal or desired value.

The error value for a neuron in an arbitrary layer that is not an output layer is computed using the following formula:

$$\text{Error}[i] = \text{Output}[i] * (1 - \text{Output}[i]) * \left(\sum_{k=0}^{(\text{NextLayerNeurons}-1)} \left(\text{NextNeuron}[k].\text{Weight}[i] * \text{NextError}[k] \right) \right)$$

FIGURE: Output error for neurons that are NOT at the output layer of a network.

The following diagram illustrates concept expressed by the formula:

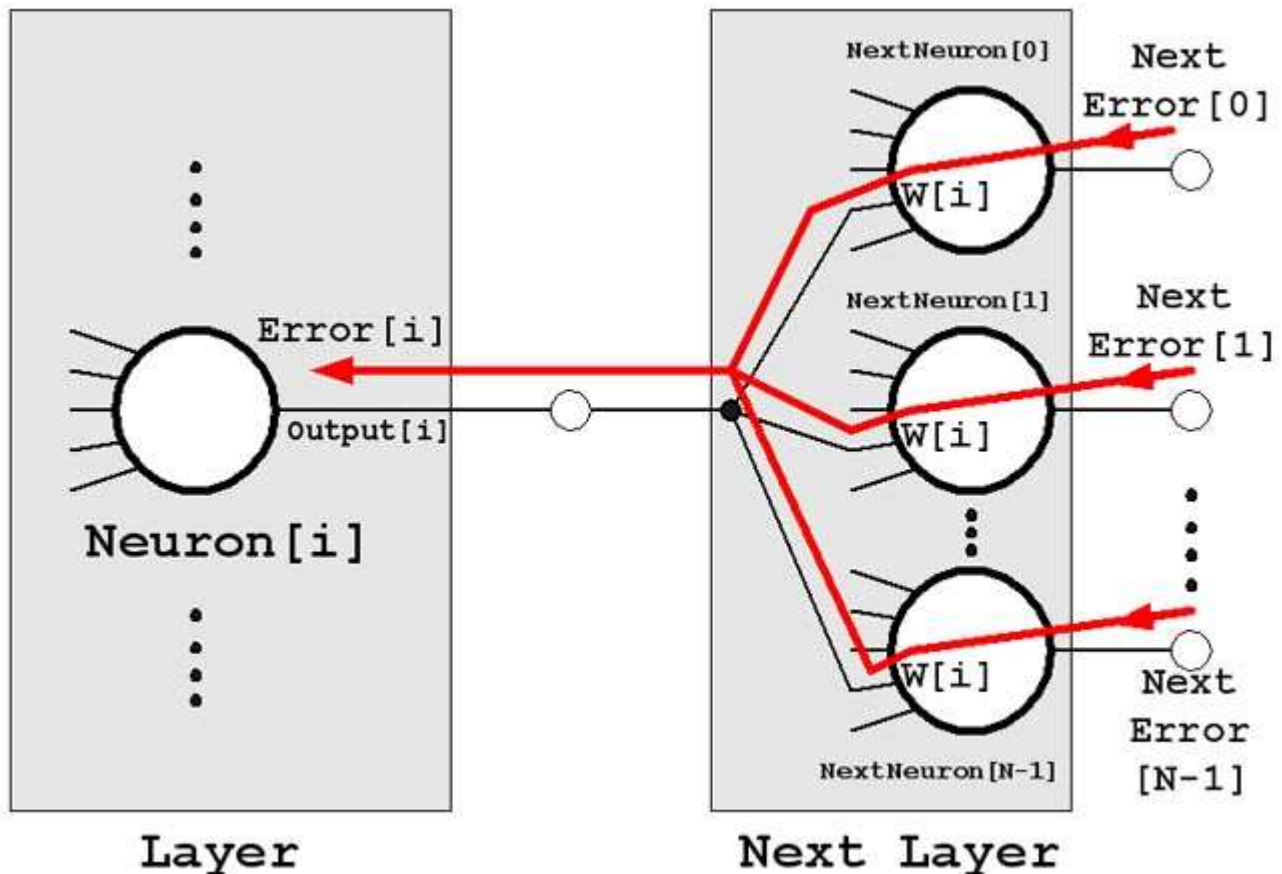


FIGURE: Output error for neurons that are NOT at the output layer of a network.

In a sense, all of the output errors at the next layer leak backwards through the input weights and accumulate at the output of a neuron in a previous layer. This accumulated value is multiplied by a value that is greatest when the current output of the neuron is most neutral (most "undecided") and is least when the output of the neuron is most extreme (very "certain").

WEIGHT CHANGE AND BIAS CHANGE FORMULAE

The basis of learning is the adjustment of weights and bias values in an attempt to reduce future output errors.

"Learning Rate" is a numerical value that essentially indicates how quickly a neuron adjusts weight and bias values according to error values.

The following formula indicates how to change the weights of a neuron with a particular set of input values and its output error value:

$$\text{WeightChange}[i] = (- (\text{Rate} * \text{Input}[i] * \text{Error}))$$

FIGURE: Weight changes for a neuron.

CAUTION: The minus sign is a consequence of the error convention adopted by this article. Many texts do not have this minus sign.

The following formula indicates how to change the bias of a neuron given the current output error for the neuron:

$$\text{BiasChange} = (- (\text{Rate} * \text{Error}))$$

FIGURE: Bias change for a neuron.

CAUTION: The minus sign is a consequence of the error convention adopted by this article. Many texts do not have this minus sign.

OVERALL TRAINING ERROR

The overall error of a network can be characterized by the square-root of the average of squared errors, or "root-mean-square" (RMS).

For a given output vector `Output[N]` and a desired output vector `Desired[N]`, we have:

```
Error[i] = (Output[i] - Desired[i]);
```

The sum of squared errors is:

```
TrainingItemSumSquaredOutputErrors = 0.0f;
for ( i = 0; i < N; i++ )
    TrainingItemSumSquaredOutputErrors += (Error[i] * Error[i]);
```

We compute `TrainingItemSumSquaredOutputErrors` for each item in the training set, adding the value to a master sum:

```
TrainingSetSumOfSquaredErrors +=
    TrainingItemSumSquaredOutputErrors;
```

We find the average:

```
AverageTrainingSetSumOfSquaredErrors =
    (TrainingSetSumOfSquaredErrors / (float)TotalTrainingItems);
```

Finally, we take the square-root to get the overall RMS error value:

```
TrainingSetRMSError =  
    sqrt( AverageTrainingSetSumOfSquaredErrors );
```

This value is one way to characterize the overall error of a network considering all training cases.

TRAINING PROCEDURE

One can start with a trained network and continue to reduce output error with further training, but one often starts with an untrained network.

Before training, choose random values for all weights of all neurons in the network. I observed problems when I randomly selected values in the interval $[-1.0, +1.0]$, and I did not have problems when I selected random values from the interval $[+0.1, +1.0]$. This is undoubtedly some kind of fluke, but I mention it anyhow.

The purpose of random weights is to mitigate the possibility of any pathological situations in a network. If all neurons in a network started with the same weights, the network would have no basis for increasing differentiation between neurons.

I have observed that setting all bias values to zero (0.0) is acceptable.

A training session involves going through a training set many times, perhaps hundreds or thousands of times.

For each pass through the training set, we consider each item in the training set.

A training set item has a set of inputs, and a set of desired outputs. We simulate the network, using the set of inputs specified by the training item. The simulation yields output values. We do the back-propagation procedure to compute the output errors for all neurons. We update all weights and biases.

CAUTION: One academic text that discussed neural networks advocated going through the entire training set and only summing up weight changes and biases. After going through the entire training set we have a set of sums of weight changes and bias changes. We take these sums and update all weights and biases.

Such sums could be HUGE for large training sets -- and the resulting jump in weight-space would be unreasonably large. So I think dividing by the number of training items, to get average weight change values and average bias change values, would be reasonable.

There is something appealing about computing a single weight change vector that somehow takes the entire training set in to consideration.

I don't know if I just made a mistake in implementing the idea, but I nearly gave up entirely on neural networks because of how poorly things were turning out. Then, when I tried the

naive alternative, namely making updates upon every training item, things worked perfectly.

Considering the entire training set before doing an update has some advantages and disadvantages:

ADVANTAGE:

=====

Single training items in the training set with extreme error (i.e., bad training item) will not make a big contribution to the update, since it will be overwhelmed by the influence of the "good" data;

DISADVANTAGE:

=====

If N is the number of items in your training set, your rate of progress to the optimal weight vector will be divided by N -- OR, for a given distance you will have only a fraction of direction hints along the way compared to the naive approach;

Perhaps this technique will work for you, but try out the naive approach before you give up on neural networks in utter frustration!

One critical aspect of training involves detecting the failure to reduce error, which is discussed in the following section.

FAILURE TO REDUCE ERROR

Training may fail to reduce the overall error for the training set. This may be due to one of several reasons:

- (1) The weight combination has reached a RELATIVE minimum of the error surface, and is "stuck";

SOLUTION: Start a fresh simulation with new random weights.

- (2) The network has too few neurons or layers to encode all of the patterns in your training set;

SOLUTION: Cautiously entertain the possibility of adding layers or neurons.

- (3) One or more items in your training set contradicts or is grossly inconsistent with your other training items;

SOLUTION: Check your data set for irregularities. Find the test items that yield the most error for your trained network. Look in to techniques to average weight changes over the entire data set to reduce the influence of any bad cases.

- (4) The learning rate is too high (anything over 1.0 is probably excessive), and the updates always overshoot the goal;

SOLUTION: Reduce learning rate.

- (5) The learning rate is too low (anything below 0.01 might be too small), and the network really IS converging on

the ideal weight combination -- but is too slow;

SOLUTION: Increase learning rate.

These items are listed in approximate order of probability, with the first item being the most probable.

When I was training a two-layer, three-neuron network to match the Exclusive-OR (XOR) function, I was frustrated and depressed by the failure to converge -- until I discovered that it was just a bad combination of initial weights. For that situation it turned out that random positive weights worked fine every time, but certain combinations of positive and negative weights sometimes caused the training to fail in a big way.

I find the notion of picking new random initial weights to recover from a failure to converge to be really lame, but from my limited experience I have the sense that this extreme measure is rarely necessary.

DISCUSSION ABOUT LEARNING

In this section we explore ways to visualize the learning process.

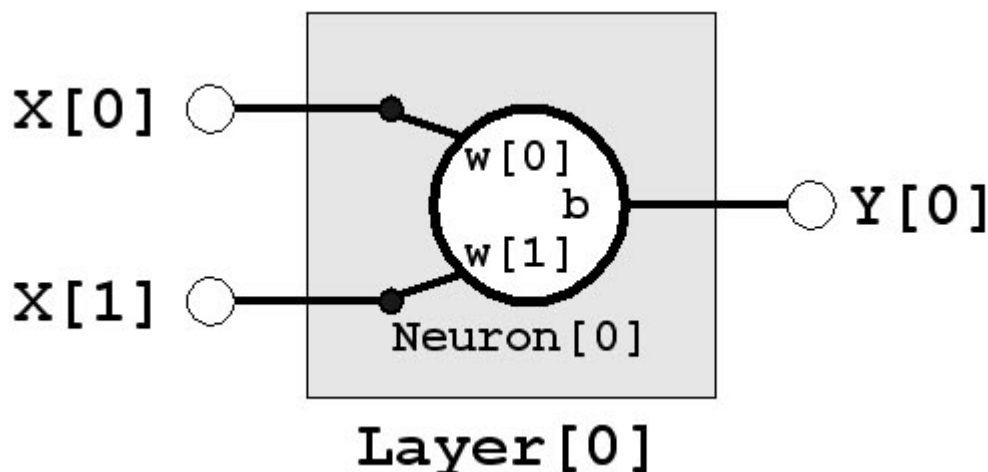


FIGURE: Single neuron with two inputs.

Imagine a single neuron with two inputs, (x_0, x_1) , two weights (w_0, w_1) , a bias, b , and an output, y .

We train this neuron by supplying inputs, computing the output, computing the error, computing weight and bias changes, and updating the weights and the bias, arriving at new weights (w_0', w_1') .

There is a very interesting way to visualize this process.

We can regard the set of weights as a vector in a multi-dimensional space. For example, for two weights we have the vector $W = (w_0, w_1)$ in a two-dimensional "weight space". When weights are adjusted, we have a new weight vector $W' = (w_0', w_1')$. We can visualize this as a point W moving to a new point W' as part of a process to minimize output error.

Normally one wouldn't compute the output error for all possible weight combinations, since the hope is that the weight adjustment process will efficiently head toward the best combination. However, let us plot the surface that essentially shows how well a neuron satisfies all items in a training set as a function of its two

weights:

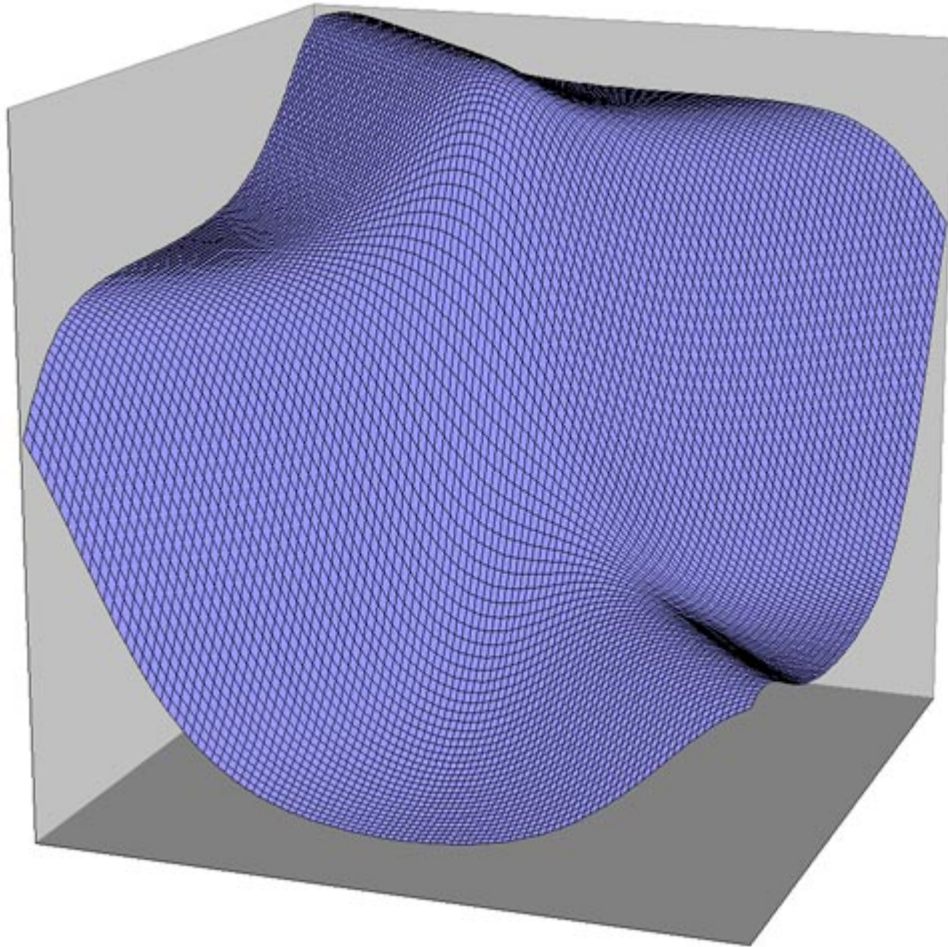


FIGURE: Negative of sum of squared errors for a specified training set as a function of two weights (w_0 , w_1).

Basically, the goal of learning is to climb to the highest peak of this surface, where error is minimized. Once we find the point $W = (w_0, w_1)$ that yields the peak value on this surface, learning is finished and we can simply use the trained neuron. (Of course, another way to look at this is to find the deepest valley in a surface that represents the positive sum of all squared errors.)

The following graph shows the output of a trained neuron as a function of all possible inputs $X = (x_0, x_1)$:

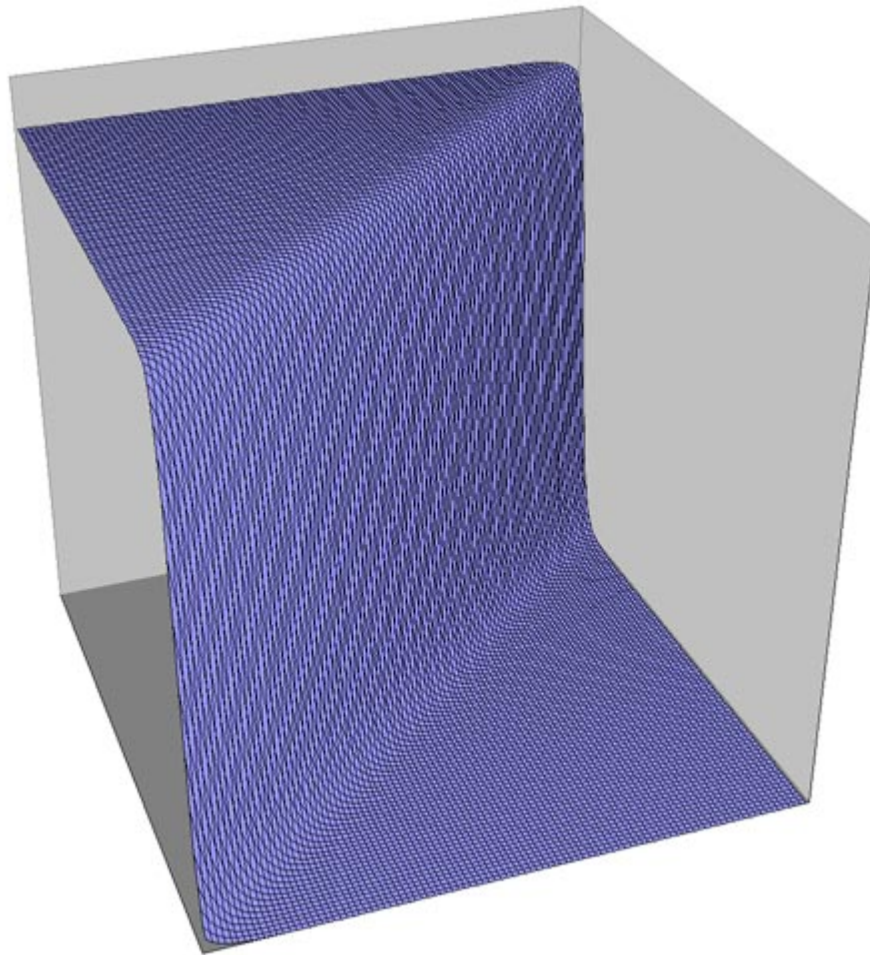


FIGURE: Neuron output as a function of two inputs (x_0 , x_1) for a weight combination that minimized squared error (i.e., maximized negative error).

Even though the weighted sum for this two-input neuron is simply $(w_0 \cdot x_0 + w_1 \cdot x_1)$, the activation function turns a simple rotated plane into a cliff. As it happens, this surface has the correct output values for all input combinations (x_0, x_1) specified by our training set. But you can imagine how input vectors $X=(x_0, x_1)$ similar to training values would also lead to the proper output values; this feature of neural networks is called "generalization" and is the main value of neural networks.

As we attempt to "climb" the surface of negative squared error, we must "leap before we look"! We update the weight vector and bias, and then we evaluate the "height" of the surface at our new location. One consequence of this is that we might leap off of a cliff to a place of more extreme error. Another consequence is that it might take a while to ascend back to the altitude of our previous location.

The possibility of "leaping" to more extreme peaks and valleys of the error surface is directly related to the "learning rate", since the learning rate determines how much influence error values have on our weight and bias changes.

The following graph shows how increasing the learning rate hastens our arrival at the peak of the negative squared error surface, where error is minimized, but introduces the possibility of bad steps and recovery:

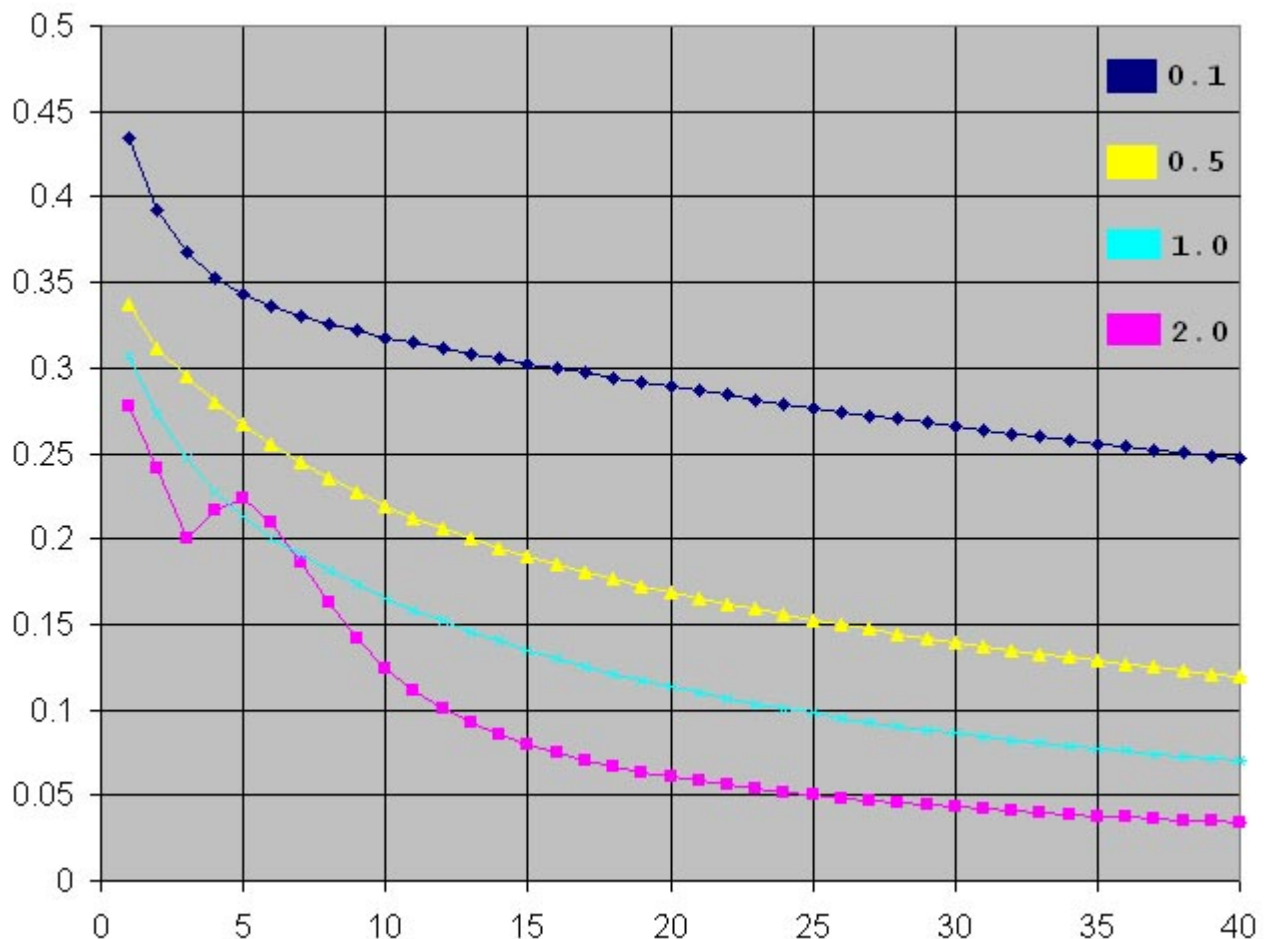


FIGURE: Short term trend of root-mean-squared (RMS) error for the entire training set over several training iterations -- for learning rates 0.1, 0.5, 1.0, 2.0.

Here is a graph of the root-mean-squared output error of a multi-layer network with a training set with 19386 items that experienced many bad steps on the path to the best weight vectors:

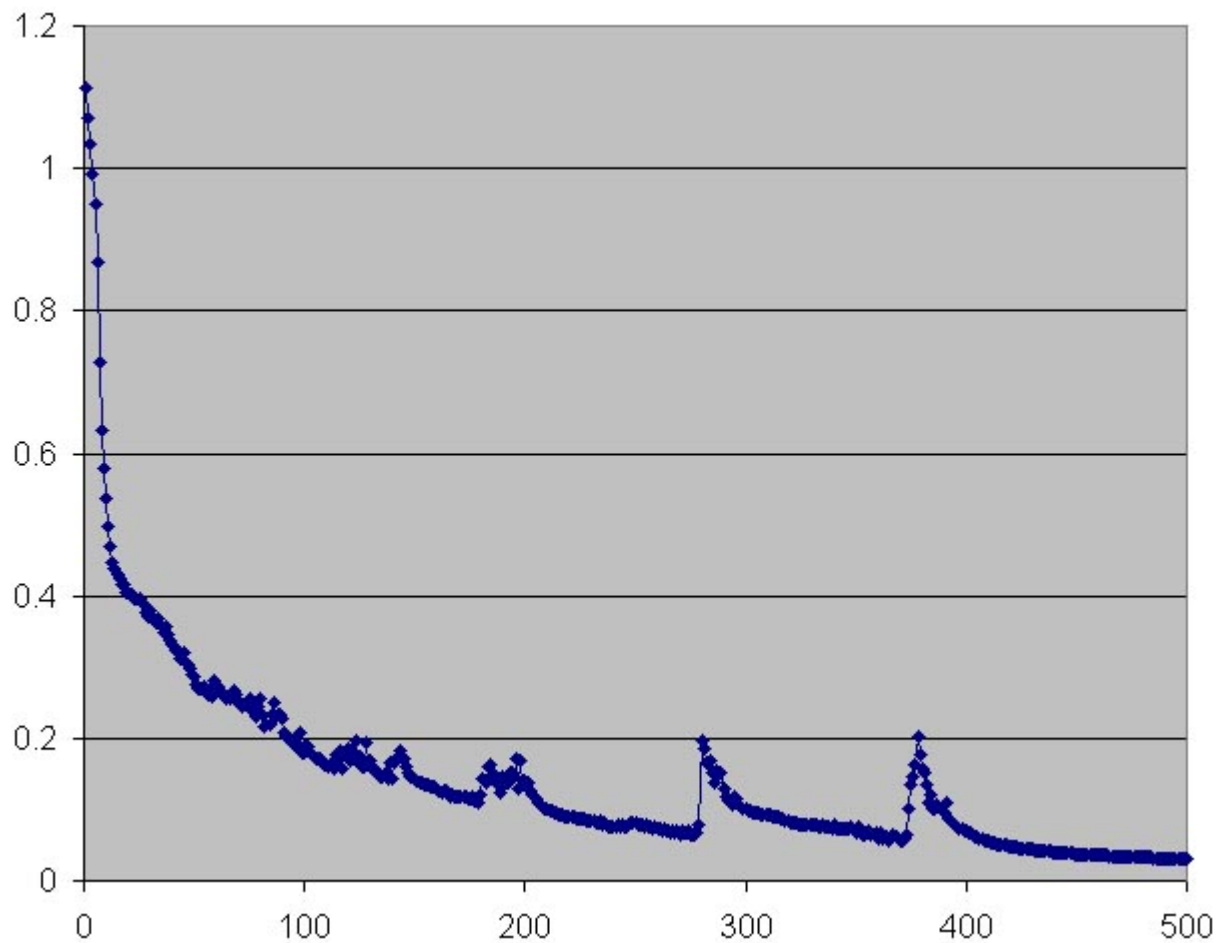


FIGURE: Training sometimes encounters spikes in the root-mean-squared (RMS) error, when error increases for some iterations before resuming a decreasing trend.

Sometimes the long term trend is simply smooth convergence to the desired set of weights:

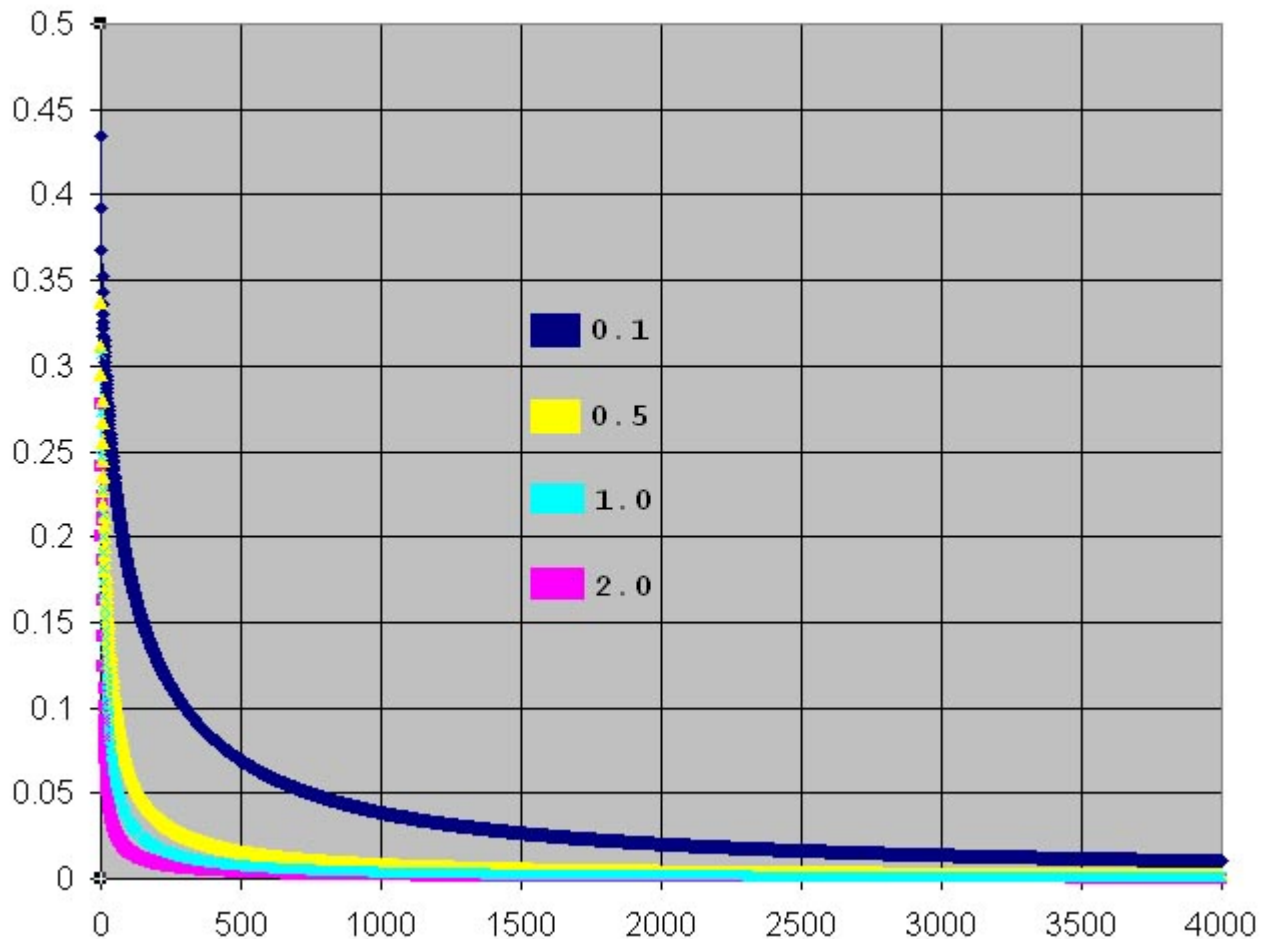


FIGURE: Long term trend of root-mean-squared (RMS) error for the entire training set over several training iterations -- for learning rates 0.1, 0.5, 1.0, 2.0.

EXAMPLE: EXCLUSIVE-OR (XOR)

DEFINITION

"Exclusive-OR" (XOR) is a function that accepts two Boolean inputs and yields a single Boolean output according to the following table:

X0	X1	Y
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE: Truth table for Exclusive-OR (XOR) function.

Consider a single neuron with two inputs:

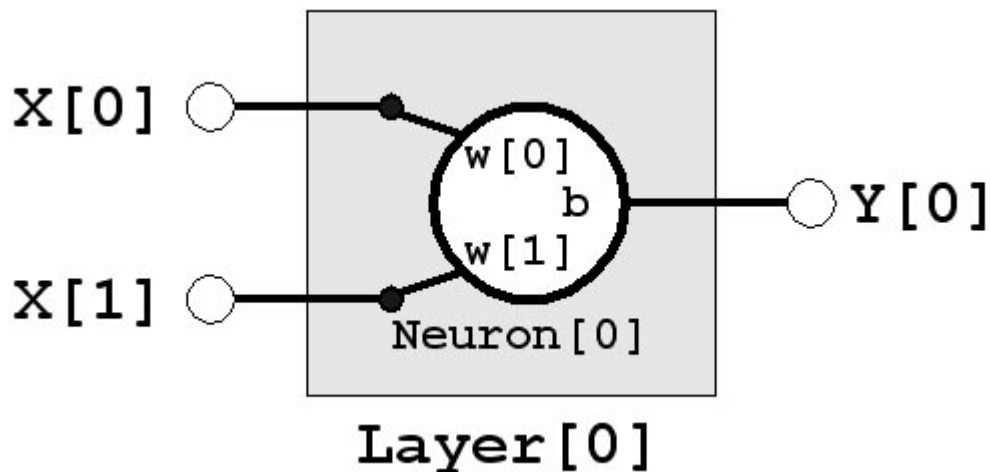


FIGURE: Single neuron with two inputs.

It is impossible to implement Exclusive-OR (XOR) with a single neuron. No weight and bias combination can yield the correct output for all four input pairs.

Texts on neural networks go in to the details of how each neuron is like a "separating plane" that divides the space of input vectors in to regions that will have potentially different outputs. For Exclusive-OR (XOR) a single neuron cannot make the TWO separations necessary to sufficiently distinguish the input combinations that yield distinct output combinations.

Two neurons can separate the input combinations sufficiently to form regions of different output values, but we require a single output signal, thus we use two neurons to do the separation of input combinations in to regions of distinct output values, and then we use a single neuron to summarize the outputs of the two other neurons. The following diagram shows the two layer neural network that can be trained to implement the Exclusive-OR (XOR) function:

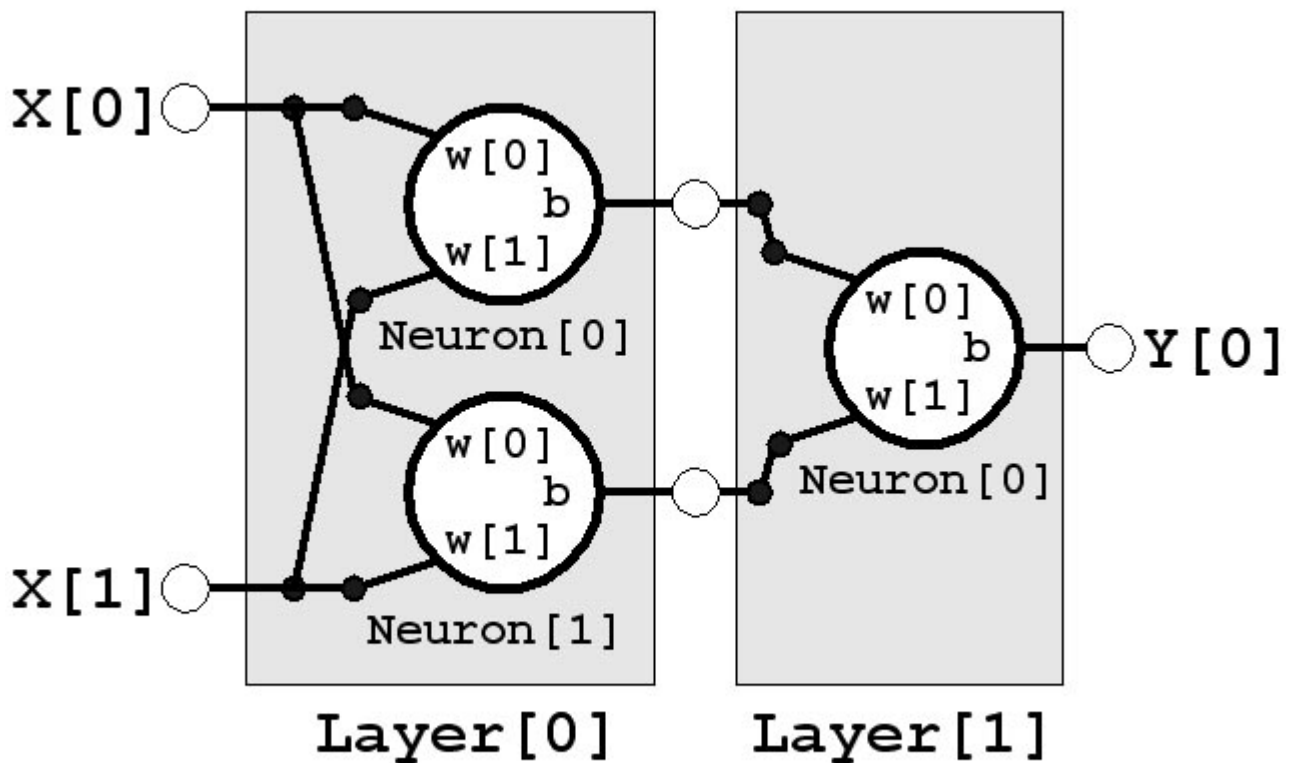


FIGURE: Two-layer network capable of implementing Exclusive-OR (XOR).

The source code demonstrates training a network to implement this function. When I trained the network shown above, the training yielded the following weights and bias values:

```
// ===== BEGIN LAYER 0 =====
// ----- BEGIN NEURON 0 -----
bias = -8.0892534
total_weights = 2
5.2864676 // 0
5.2856183 // 1
// ----- BEGIN NEURON 1 -----
bias = -3.1887615
total_weights = 2
7.0408645 // 0
7.0457759 // 1
// ===== BEGIN LAYER 1 =====
// ----- BEGIN NEURON 0 -----
bias = -5.3604155
total_weights = 2
-12.2911196 // 0
11.5286865 // 1
```

We will now analyze the weights and biases to identify the significance of the "rules" that spontaneously evolved during the training process.

The first neuron has a large negative bias, and the weights are such

that of the four possible Boolean input combinations only (1,1) will cause this neuron to fire.

The second neuron has a large negative bias, and the weights are such that if EITHER of the two Boolean inputs has a non-zero value the neuron will fire. Thus, this neuron fires for (0,1), (1,0), (1,1).

The single neuron in the second layer has a large negative bias, and has a weight combination such that the neuron will fire only if the first input is low (such as 0.0) and the second input is high (such as 1.0).

Thus, the only Boolean input value combinations that will cause this neural network to yield non-zero output are: (0,1), (1,0).

Here are the results of simulating the trained network for all four possible Boolean input combinations:

[0]	IN	0.000	0.000	;	OUT	0.007	;	DESIRED	0.000	;	ERR	0.007
[1]	IN	0.000	1.000	;	OUT	0.995	;	DESIRED	1.000	;	ERR	0.005
[2]	IN	1.000	0.000	;	OUT	0.995	;	DESIRED	1.000	;	ERR	0.005
[3]	IN	1.000	1.000	;	OUT	0.006	;	DESIRED	0.000	;	ERR	0.006

Of course the inputs to the neurons can be any floating-point values, not simply Boolean values 0.0 and 1.0, so let us see how this trained network generalized the XOR concept to arbitrary floating-point value combinations (x0,x1):

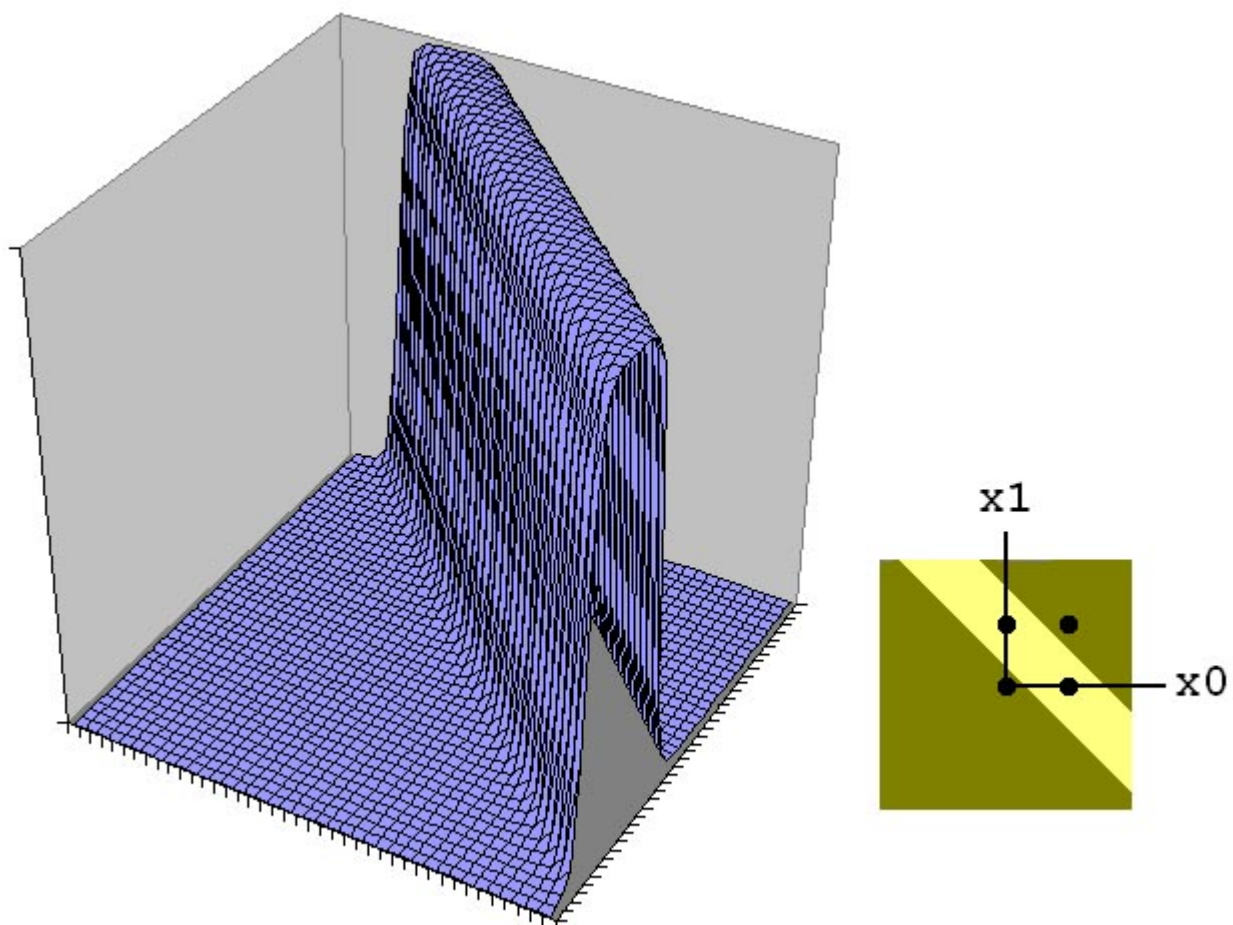


FIGURE: Output of network for all possible input combinations (x0,x1)

in the ranges $(-2.0, +2.0)$. The output ranges from 0.0 at the lowest parts of the surface to 1.0 at the peak of the surface. The inset top-view map show the locations of the four Boolean value combinations used to train the network for Exclusive-OR (XOR): (0,0), (1,0), (0,1), (1,1).

The light colored strip in the inset top-view map is where the surface is approximately 1.0, whereas the dark area is where the surface is very close to 0.0.

CAUTION

If you randomize the initial weights to the range $(-1.0, +1.0)$ before training, there is a significant chance that the training will get trapped at a relatively high overall error value.

I have found that random weights in an all-positive range such as $(+0.1, +1.0)$ consistently leads to uniform convergence to zero error.

However, the lesson is that if the error value converges to some non-zero value, then new random initial weights should be selected, and a new training session should be started. It is possible that the network does not have the capacity to represent the training set, and will never converge, but that is another matter.

EXAMPLE: Tic-Tac-Toe Game

INTRODUCTION

0	1	2
3	4	5
6	7	8

FIGURE: The nine cells of a Tic-Tac-Toe board.

Tic-Tac-Toe, also called "Naughts and Crosses", is a simple game played on a 3 x 3 grid of cells that can be marked with "O" or "X". Players alternately place "O" and "X" marks in unoccupied cells until one of the players completes a row, column, or diagonal. Since there are 3 rows, 3 columns, and 2 diagonals, there are eight winning patterns.

It is trivial to write a recursive function that explores all possible Tic-Tac-Toe games, since the maximum duration of the game is nine moves. At each point in the game we simply examine the results of moving in each of the remaining unoccupied cells. Such a function can confirm that a Tic-Tac-Toe game played with "perfect players" will end with no winner.

TRAINING A NEURAL NET TO PLAY TIC-TAC-TOE

In this example I discuss training a neural network to make the best possible move at any point in a conventional Tic-Tac-Toe game.

NOTE: This example does NOT involve a system that learns to play Tic-Tac-Toe by playing many games and reacting to wins and losses. Such a system would be more interesting than the example we describe here, but it is too complicated for this article.

This example involves training a neural network to indicate the best move for a given board configuration.

We build up a training set of unique board configurations and the corresponding optimal move for each configuration. A recursive function can explore all possible games and find optimal moves for each board configuration. We add boards and moves to the training set if we haven't encountered them before.

The network will have 9 inputs corresponding to each cell of the grid, and the input values will follow a convention:

0 : Unoccupied cell
+1 : Protagonist player
-1 : Opponent player

The network will have 9 outputs corresponding to each cell of the grid, and the output values will follow a convention:

0 : Don't move here
1 : Move here

We train the network with outputs vectors with eight zero's and a single one value. For any board configuration that can occur in conventional Tic-Tac-Toe games, the neural network should yield outputs that are all very close to 0.0 except for a single value that is close to 1.0, indicating the cell that is the best move.

CHOOSING A NETWORK SIZE

I read somewhere that any Boolean function can be implemented using only two layers of a neural network. Although the inputs of my Tic-Tac-Toe network have three possible states { -1, 0, +1 }, which makes it something different than a simple Boolean logic function, I decided to try training a network with only two layers.

This network requires 9 inputs and 9 outputs, so a two layer network would have a first layer with 9 inputs and N outputs, and the second layer would have N inputs and 9 outputs. How many neurons, N, should the first layer have?

To answer this question I trained 48 different networks,

corresponding to $N = 1, 2, \dots, 48$ total neurons in the first layer. The following graph shows the evolution of the total error over successive training iterations for all 48 networks:

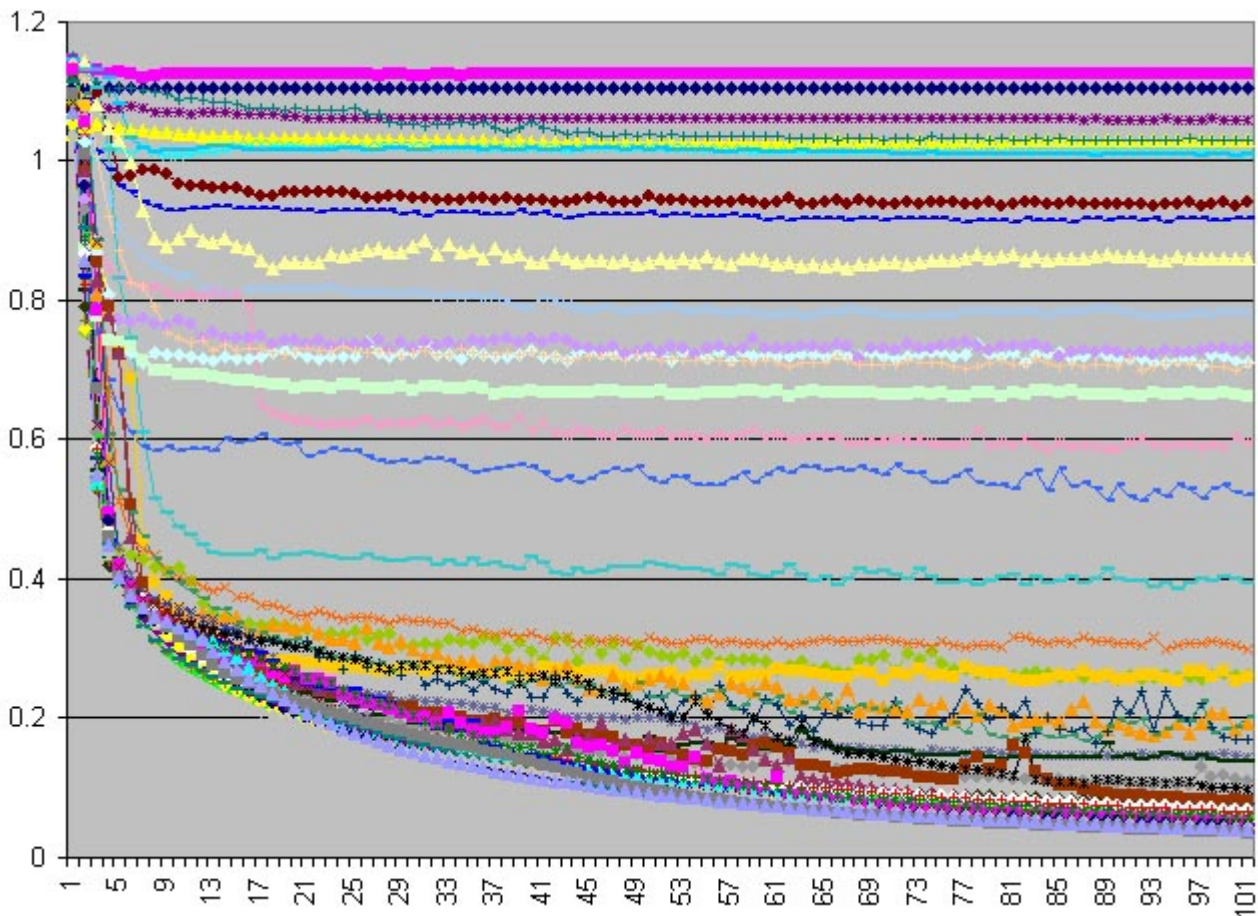


FIGURE: Evolution of overall training set error for each of 48 different two-layer neural networks, with $N = 1, 2, \dots, 48$ neurons in the first layer. NOTE: $N = 1$ neuron is the top curve, and $N = 48$ neurons is the bottom curve. Most intermediate curves are in the order of increasing neurons, from top to bottom.

Another way to visualize this trend is to take the family of curves and form a surface:

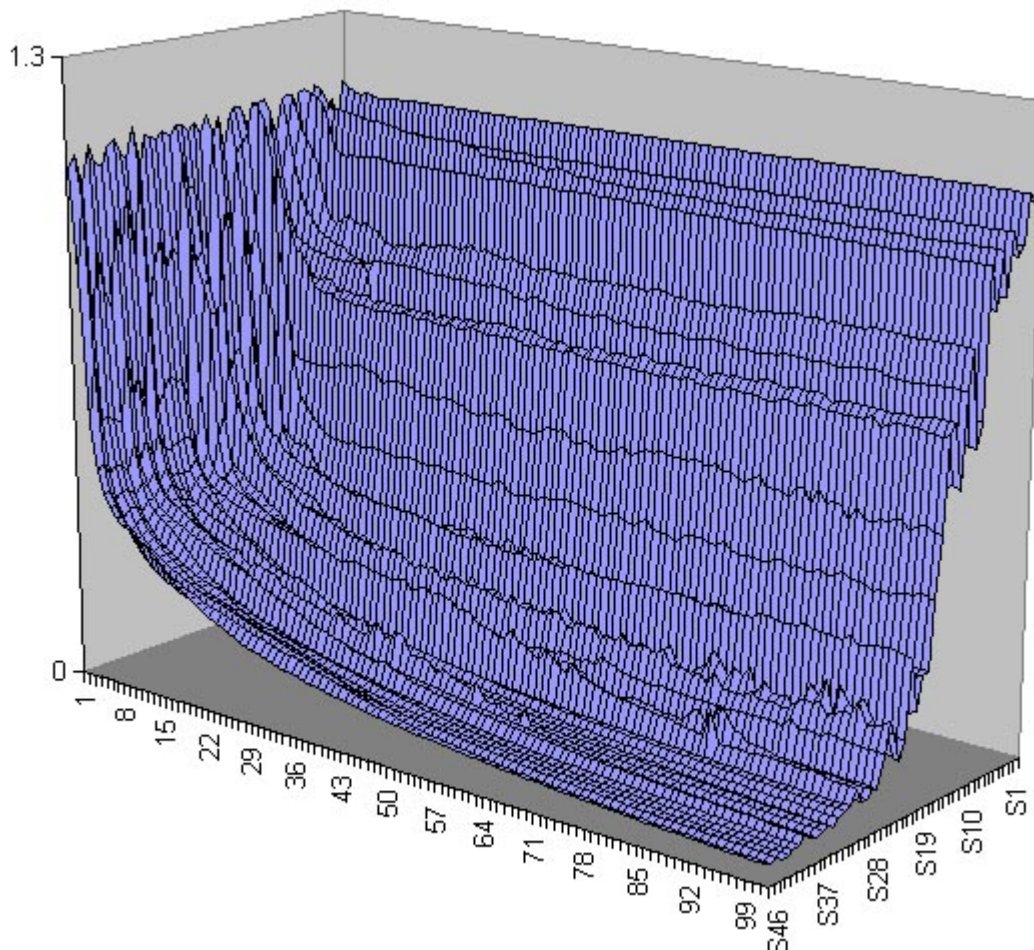


FIGURE: Evolution of overall training set error for each of 48 different two-layer neural networks, with $N = 1, 2, \dots, 48$ neurons in the first layer. NOTE: The number of neurons increases from back to front.

Thus, we see that as we approach $N = 48$ neurons, the network seems to be able to accept all training cases. Anything fewer than 48 neurons levels off at distinct error values.

My interpretation of this observation is that the nature of the Tic-Tac-Toe problem requires $N = 48$ neurons to recognize board patterns that imply the optimal move. Anything less than $N = 48$ neurons is insufficient to represent all necessary patterns, so the network performance levels off to some non-zero error value, which is the best the network can do under the circumstances.

Looking at the cross-section of the surface is also interesting. The error values at iteration 100 form a decreasing curve for increasing N values, which indicates the incremental benefit of each additional neuron. Clearly the marginal benefit of each additional neuron quickly decays to a very small amount.

RESULTS AND COMMENTS

I created a two-layer network with 9 inputs and 9 outputs, and 48 neurons in the first layer. I used a learning rate of 0.3 for all neurons. My training set had 4520 training items. In 100 training iterations (a total of $100 \times 4,520 = 452,000$ updates to the entire network), the RMS error for the entire training set was 0.03, compared to an initial error around 1.1 before any training.

Here are results from two simulations of the trained network:

```

INPUT: Tic-Tac-Toe Board State:
 1.00 -1.00  0.00
 0.00  1.00 -1.00
-1.00  0.00  0.00
OUTPUT: Best Move:
 0.00  0.00  0.00
 0.00  0.00  0.00
 0.00  0.00  1.00

INPUT: Tic-Tac-Toe Board State:
-1.00 -1.00  0.00
 1.00  1.00  0.00
 0.00  0.00  0.00
OUTPUT: Best Move:
 0.00  0.00  0.01
 0.00  0.00  1.00
 0.00  0.00  0.00

```

Note that the network was trained for the player whose mark is "+1". If you want to determine the best move for the opponent player, whose mark is "-1", invert the sign of all inputs to the network and do the simulation.

DISCUSSION: TRAINING NETWORKS

Quote from Patrick Henry Winston's "Artificial Intelligence" (3rd ed), Chapter 22, Learning by Training Neural Nets, p. 468:
(Addison Wesley; 1993)

Neural-Net Training is an Art
=====

You now know that you face many choices after you decide to work on a problem by training a neural net using back propagation:

- * How can you represent information in neural net terms?
How can you use neural net inputs to express what you know?
How can you use neural net outputs to determine what you want to know?
- * How many neurons should you have in your neural net?
How many inputs? How many outputs? How many weights?
How many hidden layers?
- * What rate parameter should you use in the back-propagation formula?
- * Should you train your neural net in stages or simultaneously?

The wrong choices lead to poor performance. A small neural net

may not learn what you want it to learn. A big net will learn slowly, may get stuck on local maxima, and may exhibit overfitting. A small rate parameter may promote instability or provide poor predictions.

Unfortunately, the proper choices depend on the nature of the samples. Mathematically, you can view the samples as representative glimpses of a hidden function, with one dimension for each input. If there are many inputs, the function's multidimensional character makes the function hard to think about and impossible to visualize.

Accordingly, the best guide to your choices is trial and error, buttressed, if possible, by reference to the choices that have worked well in similar problems. Thus, the successful deployment of neural-net technology requires time and experience. Neural-net experts are artists; they are not mere handbook users.

SOURCE CODE

INTRODUCTION

This section describes how to download, compile, and use my neural net source code.

DOWNLOAD

Click on the following link to download the latest version of my neural network source code. It is stored in a ZIP archive file. Extract contents to an arbitrary folder, and the "neuron_project" folder containing all source code files should appear.

2003 April 20th : Neural Network Source Code

[neuron_project.zip \[24 KB; ZIP Archive\]](#)

PROJECT FILES

The "neuron_project" folder, once extracted from the ZIP archive, contains the following files:

```
[ 1] neuron_float_vector.h      : FloatVector   class definition
[ 2] neuron_float_vector.cpp    : FloatVector   class implementation
[ 3] neuron_neuron.h           : Neuron        class definition
[ 4] neuron_neuron.cpp          : Neuron        class implementation
[ 5] neuron_layer.h            : NeuronLayer   class definition
[ 6] neuron_layer.cpp           : NeuronLayer   class implementation
```

[7]	neuron_network.h	: NeuronNetwork class definition
[8]	neuron_network.cpp	: NeuronNetwork class implementation
[9]	neuron_training_set.h	: TrainingSet class definition
[10]	neuron_training_set.cpp	: TrainingSet class implementation
[11]	neuron_main.h	: NeuronMain class definition
[12]	neuron_main.cpp	: main() function
[13]	neuron_project.dsw	: Microsoft Visual C++ 6.0 Workspace
[14]	neuron_project.dsp	: Microsoft Visual C++ 6.0 Project
[15]	neuron_project.opt	: Microsoft Visual C++ 6.0 Options

COMPILING

The "neuron_project" folder, once extracted from the ZIP archive having the same name, contains all source code files.

The "neuron_project" folder also contains a Microsoft Visual C++ 6.0 workspace file (*.dsw) and project file (*.dsp).

If you have Microsoft Visual Studio .NET or Microsoft Visual Studio 6.0, then you can simply open the workspace, hit Control-F5 to compile and execute, and see the demonstrations. Be sure to compile the code in a RELEASE configuration instead of a DEBUG configuration, otherwise the program will take FOUR TIMES AS LONG to compute the same results -- and you will incorrectly conclude that neural networks aren't practical for real-time applications, or that my code is inefficient. I included the *.OPT file with the project so that VC++ 6.0 will be set to use the RELEASE configuration, but with Visual Studio .NET you will need to set the Active Configuration to RELEASE after you are prompted to convert the project to the Visual Studio .NET project file format (*.vcproj).

I have not tested the code under Linux, but I believe that my code is generic enough to compile just fine under any recent distribution of Linux. I use the Standard Template Library (STL), and basic parts of the C-Runtime Library (printf(), sqrt(), rand(), etc) -- nothing platform-specific.

WHEN YOU FIRST COMPILE AND EXECUTE

When you first download, compile, and execute the code, you will see text output for the following two demonstrations:

(1) Exclusive-OR (XOR) Demonstration:

=====

In this demonstration a two-layer neural network, with two neurons in the first layer and a single neuron in the second layer, is trained to respond to Boolean inputs with an output that is the Exclusive-OR of the two inputs.

The demonstration shows a decreasing error value at various iterations during the training process. After training the network for a specific number of iterations, the network is tested for each item in the training set. Then a description of the trained network is printed, so that the weights and bias values can be examined.

(2) Tic-Tac-Toe Demonstration:

=====

In this demonstration, a two-layer neural network, with

48 neurons in the first layer and 9 neurons in the second layer, is trained to indicate the optimal move on a Tic-Tac-Toe board for the board configuration presented to the network inputs.

This demonstration has a training set with 4,520 items. The training session goes through the training set 100 times. The overall training set error is printed after each of the 100 training cycles.

After training, the network is tested with two example Tic-Tac-Toe board scenarios.

Please refer to the sections of this article corresponding to these demonstrations for more information.

THIS CODE IS IN THE PUBLIC DOMAIN

I have placed this code in to the public domain. I assume no responsibility for the code. Thus, you have complete freedom, and I have no liability. For starters, you can remove my name from all source code files.

IDEAS FOR USING THE CODE

The demonstrations should give you some idea of how the code can be used. You can easily create neural networks with arbitrary numbers of layers and neurons. You can easily generate "throw-away" networks according to the temporary needs of a procedure. You can easily study various network configurations just by using a loop variable to control the number of neurons or layers in a temporary network, which you can train and test -- and this is exactly what I did in the Tic-Tac-Toe section of this article to generate 48 different curves for 48 different network configurations.

Here are some highlights of the code:

```
[1] FloatVector    : Essentially an "array" of floating-point values.
[2] Neuron         : Properties and methods of a single neuron.
[3] NeuronLayer    : Collection of neurons with access to input set.
[4] NeuronNetwork  : Ordered collection of neuron layers; a network.
[5] TrainingSet    : Set of training items, cases, to train a network.
[6] NeuronMain     : Demonstration code.
[7] main()         : Entry point of application.
```

You can do a lot simply using instances of the NeuronNetwork object, and it is easy to fill in a TrainingSet object to submit to a network for a training iteration. Instances of the FloatVector object can be used to populate a TrainingSet object, or to submit inputs to a network and retrieve cached outputs.

The following code, from the demonstration code, is an example of the use of a NeuronNetwork object:

```
NeuronNetwork N;                // Instantiate a NeuronNetwork
```

```

N.SetTotalInputs( 2 );           // Set network to use two inputs
N.AddLayer( 2 );                 // Add a layer with two neurons
N.AddLayer( 1 );                 // Add a layer with one neuron
N.RandomizeWeights( 0.1f, 1.0f ); // Randomize all weights to range
N.SetLearningRates( 0.2f );      // Set "learning rate"

```

The following code, from the demonstration code, is an example of supplying input to a network, simulating, and getting outputs:

```

FloatVector In;                 // Instantiate FloatVector for inputs
In.SetTotalElements( 2 );       // Reserve room for two input values
FloatVector Out;                // Instantiate FloatVector for outputs
Out.SetTotalElements( 1 );      // Reserve room for one output value

In[0] = x1; In[1] = x2;         // Set inputs from variables
N.Simulate( In );               // Simulate network using inputs
N.GetOutputs( Out );            // Get cached outputs from network
y = Out[0];                     // Copy output value to variable

```

Examples of building training sets, training networks, and computing training errors are also easily spotted in the demonstration code.

REFERENCES

I found the following books to be very helpful while trying to understand neural networks, but there are probably far better books out there:

- [1] C++ Neural Networks & Fuzzy Logic (2nd ed)
Valluru Rao & Hayagriva Rao; MIS:Press; 1995;

This book is a fairly comprehensive and practical introduction to neural networks. The real significance of the source code is not that it is useful for building your own neural network applications, but that the authors encountered and addressed many of the practical problems in implementing and training neural networks. Overall, the book is useful and interesting.

- [2] Artificial Intelligence (3rd ed)
Patrick Henry Winston; Addison-Wesley; 1993;

This book is terrific! If you have any interest in Artificial Intelligence, you should acquire this book. The author covers many fascinating topics in sufficient detail, with examples, to enable you to actually apply the concepts in practical situations. The author's discussion of neural networks and back-propagation is superior to reference [1] above -- but the value of reference [1] derives from its detail regarding actual implementation.

- [3] Artificial Intelligence: A New Synthesis
Nils J Nilsson; Morgan Kaufmann; 1998

This book, like reference [2], is a fascinating and practical introduction to Artificial Intelligence. However, this book is slightly more concise than reference [2], and I really think I benefitted from having both [2] & [3] to compare presentations and identify the common ideas. I would only recommend this book if you have a strong interest in all of AI, whereas I recommend reference [2] even to those who are specifically interested in learning about neural networks.

After writing this article I saw several books in a local store (Irvine Sci-Tech Bookstore) that looked interesting:

[A] Neural Engineering (MIT Press)

Intermediate level coverage of neural networks, with many fascinating observations and illustrations. If you are mathematically inclined, and you have some interest in the theories behind neural networks, you might like this book. I'd buy it myself if I wasn't on a bit of a budget crunch right now.

[B] Focus on AI (Prima Tech / Prima Publishing)

(Part of the game development series edited by Andre LaMothe, and part of the sub-series "Focus on...", which are handy, thin, soft-cover references)
This book is geared towards game programmers, and is filled with source code examples that demonstrate neural networks, genetic algorithms, and fuzzy logic, all applied to game character behavior. Although I still believe that my article is useful, and the source code very clean and flexible, the "Focus on AI" book has fun applications that are worth trying out.

[C] AI Programming Wisdom (Charles River Media)

Like the "Game Programming Gems" series, the "AI Programming Wisdom" series is a fantastic collection of articles featuring source code and very practical and interesting demonstrations. I highly recommend the books in this series for AI projects, especially in a game programming context.

DISCLAIMER

My experience in the field of neural networks is really only two weeks of research and experimentation, and this article completely documents this experience. I welcome feedback about this article, especially corrections, but I do not have the knowledge or experience to answer questions beyond the scope of the material in this article.

I wrote this article to share my learning experience with others. My motivation for learning about neural networks comes from a personal project that happens to require classification of data. I was skeptical about neural networks being efficient and accurate enough to compete with more traditional searching techniques, but the experiences described in this article have encouraged me.

READER QUESTIONS AND COMMENTS

This article was published online on 2003 April 20th (Sunday).
Some e-mail feedback will result in corrections or augmentation of
the article, but some noteworthy e-mail messages may be summarized
here for the benefit of future readers of this article.

CONTACT INFORMATION

--- Colin P. Fahey
cpfahey@earthlink.net
<http://www.colinfahey.com>