Table of Contents

The Generalized Delta Rule

A generalized form of the delta rule, developed by D.E. Rumelhart, G.E. Hinton, and R.J. Williams, is needed for networks with hidden layers. They showed that this method works for the class of semi linear activation functions (non-decreasing and differentiable).

Generalizing the ideas of the delta rule, consider a hierarchical network with an input layer, an output layer and a number of hidden layers. We will consider only the case where there is one hidden layer. The network is presented with input signals which produce output signals that act as input to the middle layer. Output signals from the middle layer in turn act as input to the output layer to produce the final output vector. This vector is compared to the desired output vector. Since both the output and the desired output vectors are known, the delta rule can be used to adjust the weights in the output layer. Can the delta rule be applied to the middle layer? Both the input signal to each unit of the middle layer and the output signal are known. What is not known is the error generated from the output of the middle layer since we do not know the desired output. To get this error, backpropagate through the middle layer to the units that are responsible for generating that output. The error genrated from the middle layer could be used with the delta rule to adjust the weights.

Definition of a Neural Network

Neural networks have a large appeal to many researchers due to their great closeness to the structure of the brain, a characteristic not shared by more traditional systems.

In an analogy to the brain, an entity made up of interconnected neurons, neural networks are made up of interconnected processing elements called units, which respond in parallel to a set of input signals given to each. The unit is the equivalent of its brain counterpart, the neuron.

A neural network consists of four main parts:

1. Processing units{uj}, where each uj has a certain activation level aj(t) at any point in time.

2. Weighted interconnections between the various processing units which determine how the activation of one unit leads to input for another unit.

3. An activation rule which acts on the set of input signals at a unit to produce a new output signal, or activation.

4. Optionally, a learning rule that specifies how to adjust the weights for a given input/output pair.

A processing unit uj takes a number of input signals, say a1j, a2j,...,anj with corresponding weights w1j, w2j,...,wnj, respectively. The net input to uj given by:

netj = SUM (wij * aij)

The new state of activation of uj given by:

aj(t+1) = F(aj(t),netj),

where F is the activation rule and aj(t) is the activation of uj at time t. The output signal oj of unit uj is a function of the new state of activation of uj:

oj(t+1) = fj(aj(t+1)).

One of the most important features of a neural network is its ability to adapt to new environments. Therefore, learning algorithms are critical to the study of neural networks.

Brief History of Neural Networks

The earliest work in neural computing goes back to the 1940's when McCulloch and Pitts introduced the first neural network computing model. In the 1950's, Rosenblatt's work resulted in a two-layer network, the perceptron, which was capable of learning certain classifications by adjusting connection weights. Although the perceptron was successful in classifying certain patterns, it had a number of limitations. The perceptron was not able to solve the classic XOR (exclusive or) problem. Such limitations led to the decline of the field of neural networks. However, the perceptron had laid foundations for later work in neural computing.

In the early 1980's, researchers showed renewed interest in neural networks. Recent work includes Boltzmann machines, Hopfield nets, competitive learning models, multilayer networks, and adaptive resonance theory models.

Learning in a Neural Network

Learning is essential to most of these neural network architectures and hence the choice of a learning algorithm is a central issue in network development. What is really meant by saying that a processing element learns? Learning implies that a processing unit is capable of changing its input/output behavior as a result of changes in the environment. Since the activation rule is usually fixed when the network is constructed and since the input/output vector cannot be changed, to change the input/output behavior the weights corresponding to that input vector need to be adjusted. A method is thus needed by which, at least during a training stage, weights can be modified in response to the input/output process. A number of such learning rules are available for neural network models. In a neural network, learning can be supervised, in which the network is provided with the correct answer for the output during training, or unsupervised, in which no external teacher is present.

The Pattern Associator

A pattern associator learns associations between input patterns and output patterns. One of the most appealing characteristics of such a network is the fact that it can generate what it learns about one pattern to other similar input patterns. Pattern associators have been widely used in distributed memory modeling.

The pattern associator is one of the more basic two-layer networks. Its architecture consists of two sets of units, the input units and the output units. Each input unit connects to each output unit via weighted connections. Connections are only allowed from input units to output units. The effect of a unit ui in the input layer on a unit uj in the output layer is determined by the product of the activation ai of ui and the weight of the connection from ui to uj. The activation of a unit uj in the output layer is given by: SUM(wij * ai).

A pattern associator can be trained to respond with a certain output pattern when presented with an input pattern. The connection weights can be adjusted in order to change the input/output behavior. However, one of the most interesting properties of these models is their ability to self-modify and learn. The learning rule is what specifies how a network changes it weights for a given input/output association. The most commonly used learning

rules with pattern associators are the Hebb rule and the Delta rule.

The Hebb Rule

The Hebb rule determines the change in the weight connection from ui to uj by Dwij = r * ai * aj, where r is the learning rate and ai, aj represent the activations of ui and uj respectively. Thus, if both ui and uj are activated the weight of the connection from ui to uj should be increased.

Examples can be given of input/output associations which can be learned by a two-layer Hebb rule pattern associator. In fact, it can be proved that if the set of input patterns used in training are mutually orthogonal, the association can be learned by a two-layer pattern associator using Hebbian learning. However, if the set of input patterns are not mutually orthogonal, interference may occur and the network may not be able to learn associations. This limitation of Hebbian learning can be overcome by using the delta rule.

The Delta Rule

Developed by Widrow and Hoff, the delta rule, also called the Least Mean Square (LMS) method, is one of the most commonly used learning rules. For a given input vector, the output vector is compared to the correct answer. If the difference is zero, no learning takes place; otherwise, the weights are adjusted to reduce this difference. The change in weight from ui to uj is given by: dwij = r* ai * ej, where r is the learning rate, ai represents the activation of ui and ej is the difference between the expected output and the actual output of uj. If the set of input patterns form a linearly independent set then arbitrary associations can be learned using the delta rule.

It has been shown that for networks with linear activation functions and with no hidden units (hidden units are found in networks with more than two layers), the error squared vs. the weight graph is a paraboloid in n-space. Since the proportionality constant is negative, the graph of such a function is concave upward and has a minimum value. The vertex of this paraboloid represents the point where the error is minimized. The weight vector corresponding to this point is then the ideal weight vector.

This learning rule not only moves the weight vector nearer to the ideal weight vector, it does so in the most efficient way. The delta rule implements a gradient descent by moving the weight vector from the point on the surface of the paraboloid down toward the lowest point,

the vertex. Minsky and Papert raised good questions. Is there a simple learning rule that is guaranteed to work for all kinds of problems? Does the delta rule work in all cases?

As stated previously, it has been shown that in the case of linear activation functions where the network has no hidden units, the delta rule will always find the best set of weight vectors. On the other hand, that is not the case for hidden units. The error surface is not a paraboloid and so does not have a unique minimum point. There is no such powerful rule as the delat rule for networks with hidden units. There have been a number of theories in response to this problem. These include the generalized delta rule and the unsupervised competitive learning model.

The Generalized Delta Rule

A generalized form of the delta rule, developed by D.E. Rumelhart, G.E. Hinton, and R.J. Williams, is needed for networks with hidden layers. They showed that this method works for the class of semilinear activation functions (non-decreasing and differentiable).

Generalizing the ideas of the delta rule, consider a hierarchical network with an input layer, an output layer and a number of hidden layers. We will consider only the case where there is one hidden layer. The network is presented with input signals which produce output signals that act as input to the middle layer. Output signals from the middle layer in turn act as input to the output layer to produce the final output vector. This vector is compared to the desired output vector. Since both the output and the desired output vectors are known, the delta rule can be used to adjust the weights in the output layer. Can the delta rule be applied to the middle layer? Both the input signal to each unit of the middle layer and the output signal are known. What is not known is the error generated from the output of the middle layer since we do not know the desired output. To get this error, backpropagate through the middle layer to the units that are responsible for generating that output. The error genrated from the middle layer could be used with the delta rule to adjust the weights.

What is the difference between biological and artificial neural networks?

Artificial neural networks (ANNs) are mathematical constructs, originally designed to approximate biological neurons. Each "neuron" is a relatively simple element --- for example, summing its inputs and applying a threshold to the result, to determine the output of that "neuron".

Several decades of research went into discovering how to build network architectures using these mathematical constructs, and how to automatically set the weighting on each of the connections between the neurons to perform a wide range of tasks. For example, ANNs can do things like recognition of hand-written digits.
A "biological neural network" would refer to any group of connected biological nerve cells.

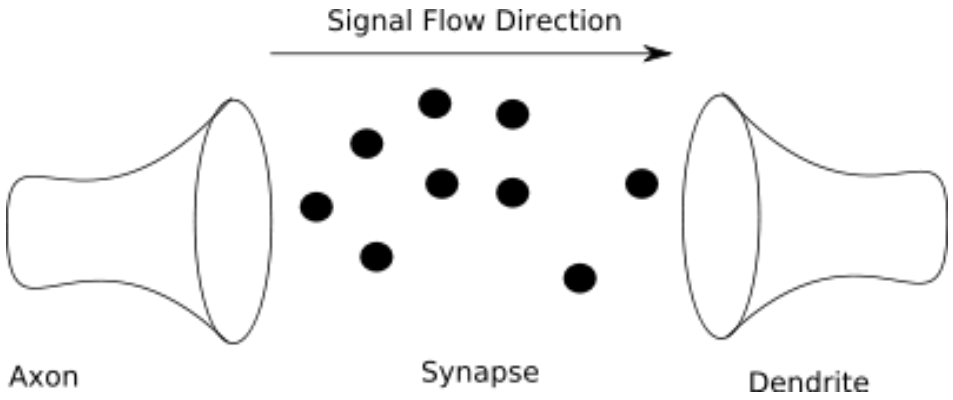
Your brain is a biological neural network, so is a number of neurons grown together in a dish so that they form synaptic connections. The term "biological neural network" is not very precise; it doesn't define a particular biological structure.
In the same way, an ANN can mean any of a large number of mathematical "neuron"-like constructs, connected in any of a large number of ways, to perform any of a large number of tasks.

Biological Neural Nets

In the case of a biological neural net, neurons are living cells with axons and dendrites that form interconnections through electro-chemical synapses. Signals are transmitted through the cell body (soma), from the dendrite to the axon as an electrical impulse. In the pre-synaptic membrane of the axon, the electrical signal is converted into a chemical signal in the form of various neurotransmitters. These neurotransmitters, along with other chemicals present in the synapse form the message that is received by the post-synaptic membrane of the dendrite of the next cell, which in turn is converted to an electrical signal.

This page is going to provide a brief overview of biological neural networks, but the reader will have to find a better source for a more in-depth coverage of the subject.

Synapses



The figure above shows a model of the synapse showing the chemical messages of the synapse moving from the axon to the dendrite. Synapses are not simply a transmission medium for chemical signals, however. A synapse is capable of modifying itself based on the signal traffic that it receives. In this way, a synapse is able to “learn” from its past activity. This learning happens through the strengthening or weakening of the connection. External factors can also affect the chemical properties of the synapse, including body chemistry and medication.

Neurons

Cells have multiple dendrites, each receives a weighted input. Inputs are weighted by the strength of the synapse that the signal travels through. The total input to the cell is the sum of all such synaptic weighted inputs. Neurons utilize a threshold mechanism, so that signals below a certain threshold are ignored, but signals above the threshold cause the neuron to fire. Neurons follow an “all or nothing” firing scheme, and are similar in this respect to a digital component. Once a neuron has fired, a certain refraction period must pass before it can fire again.

Biological Networks

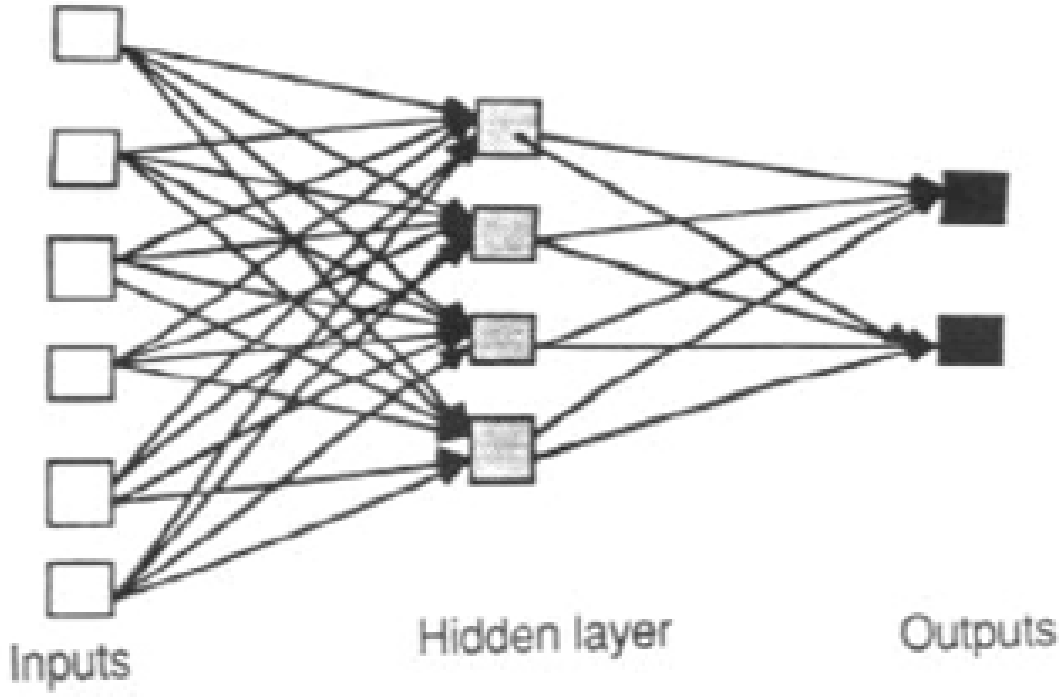
Biological neural systems are heterogeneous, in that there are many different types of cells with different characteristics. Biological systems are also characterized by macroscopic order, but nearly random interconnection on the microscopic layer. The random interconnection at the cellular level is rendered into a computational tool by the learning process of the synapse, and the formation of new synapses between nearby neurons.

ANN

Artificial neural networks are a computational tool, based on the properties of biological neural systems. Neural networks excel in a number of problem areas where conventional von Neumann computer systems have traditionally been slow and inefficient. This book is going to discuss the creation and use of artificial neural networks.

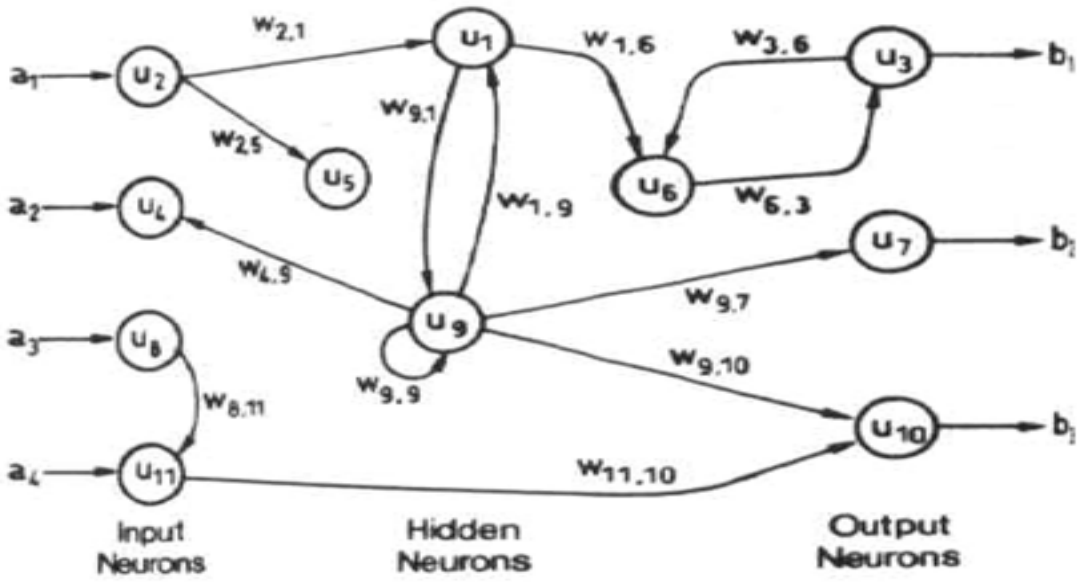
Feed-forward and recurrent neural networks?

Feed-Forward ANNs allow signals to travel one way only: from input to output. There are no feedback (loops); *i.e.*, the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straightforward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.



Feed-Back or recurrent or interactive) networks can have signals traveling in both directions by introducing loops in the network. Feedback networks are powerful and can get extremely complicated. Computations derived from earlier input are fed back into the network, which

gives them a kind of memory. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found.



Feedforward neural networks are ideally suitable for modeling relationships between a set of predictor or input variables and one or more response or output variables. In other words, they are appropriate for any functional mapping problem where we want to know how a number of input variables affect the output variable. The multilayer feedforward neural networks, also called multi-layer perceptrons (MLP), are the most widely studied and used neural network model in practice.

As an example of feedback network, I can recall Hopfield's network. The main use of Hopfield's network is as associative memory. An associative memory is a device which accepts an input pattern and generates an output as the stored pattern which is most closely associated with the input. The function of the associate memory is to recall the corresponding stored pattern, and then produce a clear version of the pattern at the output. Hopfield networks are typically used for those problems with binary pattern vectors and the input pattern may be a noisy version of one of the stored patterns. In the Hopfield network, the stored patterns are encoded as the weights of the network.

Kohonen's self-organizing maps (SOM) represent another neural network type that is markedly different from the feedforward multilayer networks. Unlike training in the feedforward MLP, the SOM training or learning is often called unsupervised because there are no known target outputs associated with each input pattern in SOM and during the training process, the SOM processes the input patterns and learns to cluster or segment the data through adjustment of weights (that makes it an important neural network model for dimension reduction and data clustering). A two-dimensional map is typically created in such a way that the orders of the interrelationships among inputs are preserved. The number and composition of clusters can be visually determined based on the output distribution generated by the training process. With only input variables in the training sample, SOM aims to learn or discover the underlying structure of the data.

Architecture (Multilayer Perceptron)

The Architecture tab is used to specify the structure of the network. The procedure can select the "best" architecture automatically, or you can specify a custom architecture.

Automatic architecture selection builds a network with one hidden layer. Specify the minimum and maximum number of units allowed in the hidden layer, and the automatic architecture selection computes the "best" number of units in the hidden layer. Automatic architecture selection uses the default activation functions for the hidden and output layers.

Custom architecture selection gives you expert control over the hidden and output layers and can be most useful when you know in advance what architecture you want or when you need to tweak the results of the Automatic architecture selection.

Hidden Layers

The hidden layer contains unobservable network nodes (units). Each hidden unit is a function of the weighted sum of the inputs. The function is the activation function, and the values of the weights are determined by the estimation algorithm. If the network contains a second hidden layer, each hidden unit in the second layer is a function of the weighted sum of the units in the first hidden layer. The same activation function is used in both layers.

Number of Hidden Layers. A multilayer perceptron can have one or two hidden layers.

Activation Function. The activation function "links" the weighted sums of units in a layer to the values of units in the succeeding layer.

- **Hyperbolic tangent.** This function has the form: $\gamma(c) = \tanh(c) = (ec - e - c)/(ec + e - c)$. It takes real-valued arguments and transforms them to the range (–1, 1). When automatic architecture selection is used, this is the activation function for all units in the hidden layers.

- **Sigmoid.** This function has the form: $\gamma(c) = 1/(1 + e - c)$. It takes real-valued arguments and transforms them to the range (0, 1).

Number of Units. The number of units in each hidden layer can be specified explicitly or determined automatically by the estimation algorithm.

Output Layer

The output layer contains the target (dependent) variables.

Activation Function. The activation function "links" the weighted sums of units in a layer to the values of units in the succeeding layer.

- **Identity.** This function has the form: $\gamma(c) = c$. It takes real-valued arguments and returns them unchanged. When automatic architecture selection is used, this is the activation function for units in the output layer if there are any scale-dependent variables.

- **Softmax.** This function has the form: $\gamma(ck) = \exp(ck)/\Sigma j\exp(cj)$. It takes a vector of real-valued arguments and transforms it to a vector whose elements fall in the range (0, 1) and sum to 1. Softmax is available only if all dependent variables are categorical. When automatic architecture selection is used, this is the activation function for units in the output layer if all dependent variables are categorical.

- **Hyperbolic tangent.** This function has the form: $\gamma(c) = \tanh(c) = (ec-e-c)/(ec+e-c)$. It takes real-valued arguments and transforms them to the range (–1, 1).

- **Sigmoid.** This function has the form: $\gamma(c) = 1/(1+e-c)$. It takes real-valued arguments and transforms them to the range (0, 1).

Rescaling of Scale Dependent Variables. These controls are available only if at least one scale-dependent variable has been selected.

- **Standardized.** Subtract the mean and divide by the standard deviation, $(x-\text{mean})/s$.

- **Normalized.** Subtract the minimum and divide by the range, $(x-\text{min})/(\text{max}-\text{min})$. Normalized values fall between 0 and 1. This is the required rescaling method for scale-dependent variables if the output layer uses the sigmoid activation function. The correction option specifies a small number ε that is applied as a correction to the rescaling formula; this correction ensures that all rescaled dependent variable values will be within the range of the activation function. In particular, the values 0 and 1, which occur in the uncorrected formula when x takes its minimum and maximum value, define the limits of the range of the sigmoid function but are not within that range. The corrected formula is $[x-(\text{min}-\varepsilon)]/[(\text{max}+\varepsilon)-(\text{min}-\varepsilon)]$. Specify a number greater than or equal to 0.

• **Adjusted Normalized.** Adjusted version of subtracting the minimum and dividing by the range, [2*(*x*−min)/(max−min)]−1. Adjusted normalized values fall between −1 and 1. This is the required rescaling method for scale-dependent variables if the output layer uses the hyperbolic tangent activation function. The correction option specifies a small number ε that is applied as a correction to the rescaling formula; this correction ensures that all rescaled dependent variable values will be within the range of the activation function. In particular, the values −1 and 1, which occur in the uncorrected formula when *x* takes its minimum and maximum value, define the limits of the range of the hyperbolic tangent function but are not within that range. The corrected formula is {2*[(*x*−(min−ε))/((max+ε)−(min−ε))]}−1. Specify a number greater than or equal to 0.

• **None.** No rescaling of scale-dependent variables.