Single-Layer Neural Network

What does Single-Layer Neural Network mean?

A single-layer neural network represents the simplest form of neural network, in which there is only one layer of input nodes that send weighted inputs to a subsequent layer of receiving nodes, or in some cases, one receiving node. This single-layer design was part of the foundation for systems which have now become much more complex.

One of the early examples of a single-layer neural network was called a “perceptron.” The perceptron would return a function based on inputs, again, based on single neurons in the physiology of the human brain. In some senses, perceptron models are much like “logic gates” fulfilling individual functions: A perceptron will either send a signal, or not, based on the weighted inputs. Another type of single-layer neural network is the single-layer binary linear classifier, which can isolate inputs into one of two categories.

Single-layer neural networks can also be thought of as part of a class of feedforward neural networks, where information only travels in one direction, through the inputs, to the output. Again, this defines these simple networks in contrast to immensely more complicated systems, such as those that use backpropagation or gradient descent to function. Neurons with this kind of activation function are also called *artificial neurons* or *linear threshold units*. A perceptron can be created using any values for the activated and deactivated states as long as the threshold value lies between the two. Perceptrons can be trained by a simple learning algorithm that is usually called the *delta rule*. It calculates the errors between calculated output and sample output data, and uses this to create an adjustment to the weights, thus implementing a form of gradient descent.

Single-layer perceptrons are only capable of learning linearly separable patterns; in 1969 in a famous monograph entitled *Perceptrons*, Marvin Minsky and Seymour Papert showed that it was impossible for a single-layer perceptron network to learn an XOR function (nonetheless, it was known that multi-layer perceptrons are capable of producing any possible boolean function). Although a single threshold unit is quite limited in its computational power, it has been shown that networks of parallel threshold units can approximate any continuous function from a compact interval of the real numbers into the interval [-1,1]. This result can be found in Peter Auer, Harald Burgsteiner and Wolfgang Maass "A learning rule for very simple universal approximators consisting of a single layer of perceptrons".

A single-layer neural network can compute a continuous output instead of a step function. A common choice is the so-called logistic function:

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

With this choice, the single-layer network is identical to the logistic regression model, widely used in statistical modeling. The logistic function is one of the family of functions called sigmoid functions because their S-shaped graphs resemble the final-letter lower case of the Greek letter Sigma. It has a continuous derivative, which allows it to be used in backpropagation. This function is also preferred because its derivative is easily calculated:

$$f'(x) = f(x)(1 - f(x)).$$

(The fact that f satisfies the differential equation above can easily be shown by applying the chain rule.) If single-layer neural network activation function is modulo 1, then this network can solve XOR problem with a single neuron.

$$f(x) = x \mod 1$$
$$f'(x) = 1$$

Multi-layer perceptron (briefly!)

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a *sigmoid function* as an activation function. The *universal approximation theorem* for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer. This result holds for a wide range of activation functions, e.g., for the sigmoidal functions.

Multi-layer networks use a variety of learning techniques, the most popular being *back-propagation*. Here, the output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has *learned* a certain target function. To adjust weights properly, one applies a general method for non-linear optimization that is called gradient descent. For this, the network calculates the derivative of the error function with respect to the network weights, and changes the weights such that the error decreases (thus going downhill on the surface of the error function).

For this reason, back-propagation can only be applied on networks with differentiable activation functions. In general, the problem of teaching a network to perform well, even on samples that were not used as training samples, is a quite subtle issue that requires additional techniques. This is especially important for cases where only very limited numbers of training samples are available. The danger is that the network overfits the training data and fails to capture the true statistical process generating the data. Computational learning theory is concerned with training classifiers on a limited amount of data. In the context of neural networks, a simple heuristic, called early stopping, often ensures that the network will generalize well to examples not in the training set. Other typical problems of the back-propagation algorithm are the speed of convergence and the possibility of ending up in a local minimum of the error function. Today, there are practical methods that make back-propagation in multi-layer perceptrons the tool of choice for many machine learning tasks. One also can use a series of independent neural networks moderated by some intermediary, a similar behavior that happens in brain. These neurons can perform separably and handle a large task, and the results can be finally combined.

ACTIVATION FUCTION

In computational networks, the activation function of a node defines the output of that node given an input or set of inputs. In a neural network, each neuron has an activation function which specifies the output of a neuron to a given input. Neurons are 'switches' that output a '1' when they are sufficiently activated and a '0' when not.

Types

1. Identity Function

$$f(x) = x \, for \, all \, x$$

2. Binary step function

$$f(x) = \begin{cases} 1, & x \geq \theta \\ 0, & x \leq \theta \end{cases}$$

Where is the threshold value for the activation function

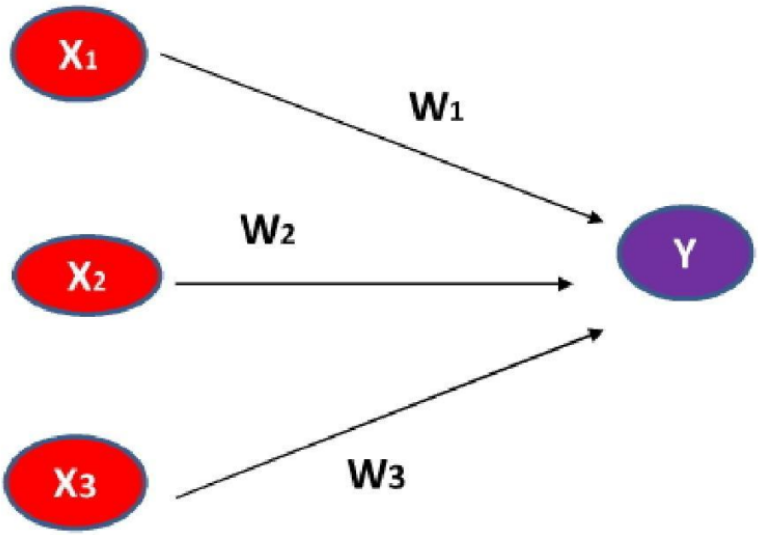
3. Binary Sigmoid Function

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

McCULLOCH PITTS NEURON

In 1943 Warren S. McCulloch, a neuroscientist, and Walter Pitts, a logician, published "A logical calculus of the ideas immanent in nervous activity" in the *Bulletin of*

Mathematical Biophysics 5:115-133. McCulloch and Pitts tried to understand how the brain could produce highly complex patterns by using many basic cells that are connected together. These basic brain cells are called neurons, and McCulloch and Pitts gave a highly simplified model of a neuron in their paper. The McCulloch and Pitts model of a neuron, called as **MCP neuron**, has been very important in computer science. Figure 4 shows the architecture of McCulloch Pitts network.



$$Y\_in = W_1X_1 + W_2X_2 + W_3X_3$$

$$Y = f(Y\_in)$$

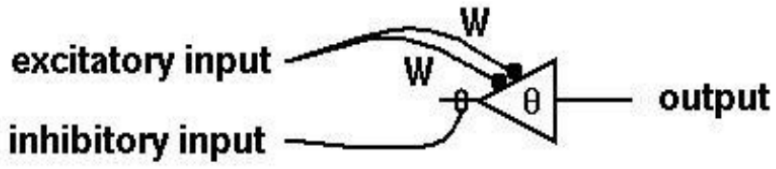
Figure 4. Architecture of McCulloch Pitts Model

The neurons operated under the following assumptions:

1. They are binary devices (Vi = [0,1])
2. Each neuron has a fixed threshold, theta
3. The neuron receives inputs from excitatory synapses, all having identical weights. (However it may receive multiple inputs from the same source, so the excitatory weights are effectively positive integers.)
4. Inhibitory inputs have an absolute veto power over any excitatory inputs.
5. At each time step the neurons are simultaneously (synchronously) updated by summing the weighted excitatory inputs and setting the output (Vi) to 1 if the sum is greater than or equal to the threshold AND if the neuron receives no inhibitory input.

the rules with the McCullough-Pitts output rule is,

$$V_i = \begin{cases} 1 & : \sum_j W V_j \geq \theta \text{ AND no inhibition} \\ 0 & : \text{otherwise} \end{cases}$$

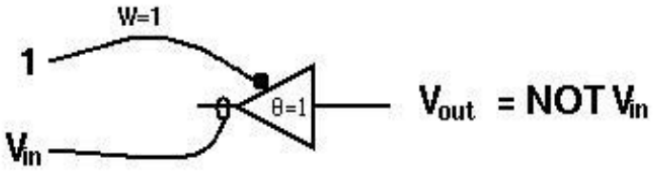


Using this scheme we can figure out how to implement any Boolean logic function. As you probably know, with a NOT function and either an OR or an AND, you can build up XOR's, adders, shift registers, and anything you need to perform computation.

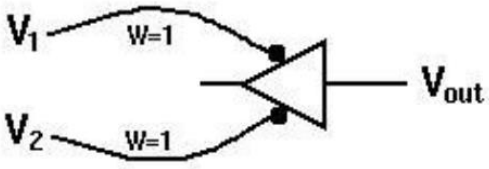
We represent the output for various inputs as a truth table, where 0 = FALSE, and 1 = TRUE. You should verify that when W = 1 and theta = 1, we get the truth table for the logical NOT,

Vin	Vout
1	0
0	1

by using this circuit:



With two excitatory inputs V1 and V2, and W =1, we can get either an OR or an AND, depending on the value of theta:



If θ=1 , Vout = V1 OR V2

if θ=2 , Vout = V1 AND V2

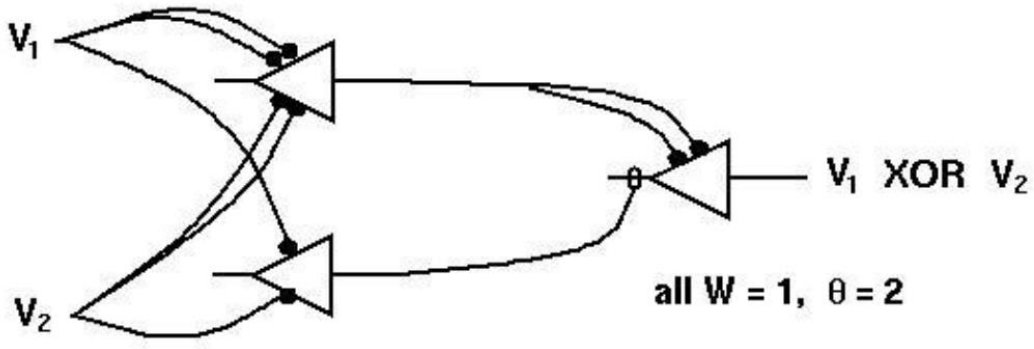
Can you verify that with these weights and thresholds, the various possible inputs for V1 and V2 result in this table?

V1	V2	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

The exclusive OR (XOR) has the truth table:

V1	V2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

It cannot be represented with a single neuron, but the relationship XOR = (V1 OR V2) AND NOT (V1 AND V2) suggests that it can be represented with the network



LINEAR SEPERABILITY

If two classes of patterns can be separated by a decision boundary, represented by the linear equation,

$$b + \sum_{i-1}^{n} x_i w_i = 0$$

then they are said to be linearly separable. The simple network can correctly classify the patterns. Where, b is bias value. Decision boundary (i.e,W,b or ɵ) of linearly separable classes can be determined either by some learning procedures or by solving linear equation systems based on representative patterns of each classes. If such a decision boundary does not exist, then the two classes are said to be linearly inseparable. Linearly inseparable problems cannot be solved by simple network, more sophisticated architecture is needed.

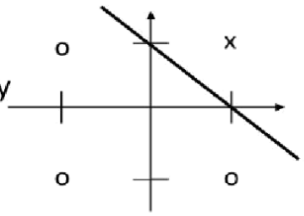
Examples of linearly separable classes

- Logical AND function

patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

w1 = 1
w2 = 1
b = -1
θ = 0
-1 + x1 + x2 = 0



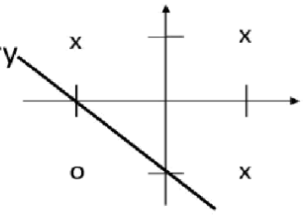
x: class I (y = 1)
o: class II (y = -1)

- Logical OR function

patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

w1 = 1
w2 = 1
b = 1
θ = 0
1 + x1 + x2 = 0



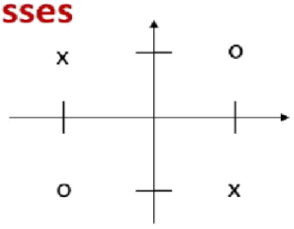
x: class I (y = 1)
o: class II (y = -1)

Examples of linearly inseparable classes

Logical XOR (exclusive OR) function

patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



x: class I (y = 1)
o: class II (y = -1)

No line can separate these two classes, as can be seen from the fact that the following linear inequality system has no solution

$$\begin{cases} b - w_1 - w_2 < 0 & (1) \\ b - w_1 + w_2 \geq 0 & (2) \\ b + w_1 - w_2 \geq 0 & (3) \\ b + w_1 + w_2 < 0 & (4) \end{cases}$$

because we have b < 0 from (1) + (4), and b >= 0 from (2) + (3), which is a contradiction

HEBB NET

In neural networks, learning is achieved mostly (but not exclusively) through changes in the strengths of the connections between neurons. Mechanisms of learning include: - changes in neural parameters (threshold, time constants) - creation of new synapses - elimination of synapses - changes in the synaptic weights or connection strengths Hebbian learning rule? One common way to calculate changes in connection strengths in a neural network is the so called “hebbian learning rule”, in which a change in the strength of a connection is a function of the pre – and postsynaptic neural activities. It is called the “hebbian learning rule” after D. Hebb (“When neuron A repeatedly participates in firing neuron B, the strength of the action of A onto B increases”).

If xj is the output of the pre-synaptic neuron,

xi the output of the postsynaptic neuron, and Wij the strength of the connection between them, and γ learning rate, the one form of a learning rule would be:

$\Delta Wij\ (t) = \gamma * xj * xi$

A more general form of a hebbian learning rule would be:

$\Delta Wij\ (t) = F(xj, xi, \gamma, t, \theta)$

in which time and learning thresholds can be taken into account.

ALG ORITHM

Step 0 Initialize weights Wi = 0(i = 0 to n)

Step 1 For each input training vector and target s:t do step 2 to 4

Step 2 Set activations for input units Xi = Si (i 1 to n)

Step 3 Set Activation for output unit y=t

Step 4 adjust weights and bias for

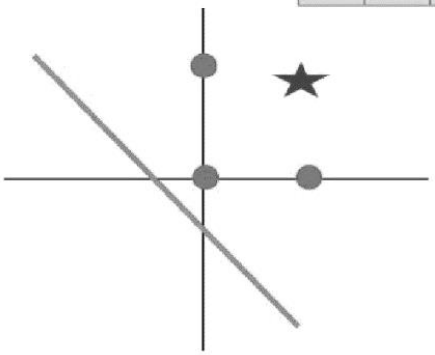
$$W_i(new) = W_i(old) + xt$$

$$b(new) = b(old) + t$$

LOGIC AND binary input and target

X1	X2	Bias	target
1	1	1	1
1	0	1	0
0	1	1	0
0	0	1	0

Input			Target	Wt changes			Weights		
X1	X2	1	1	Δw1	Δw2	Δb	W1	W2	b
1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1
0	0	1	0	0	0	0	1	1	1

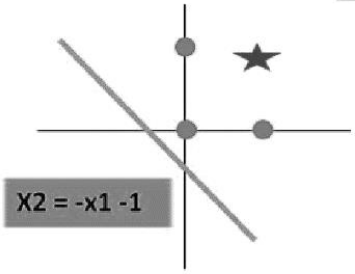


X2 = -x1 -1

LOGIC AND binary input and bipolar target

X1	X2	Bias	target
1	1	1	1
1	0	1	-1
0	1	1	-1
0	0	1	-1

Input			Target	Wt changes			Weights		
X1	X2	1	1	Δw1	Δw2	Δb	W1	W2	b
1	1	1	1	1	1	1	1	1	1
1	0	1	-1	-1	0	-1	0	1	0
0	1	1	-1	0	-1	-1	0	0	-1
0	0	1	-1	0	0	-1	0	0	-2

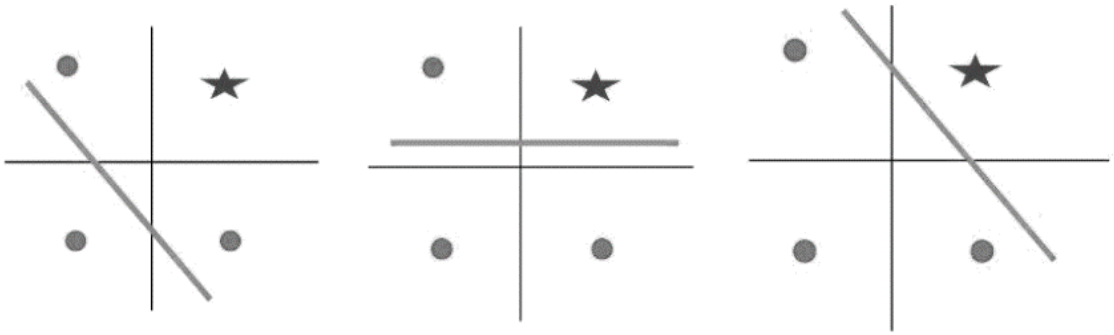


X2 = -x1 -1

LOGIC AND bipolar input and target

X1	X2	Bias	target
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

Input			Target	Wt changes			Weights		
X1	X2	1	1	Δw1	Δw2	Δb	W1	W2	b
1	1	1	1	1	1	1	1	1	1
1	-1	1	-1	-1	1	-1	0	2	0
-1	1	1	-1	1	1	-1	1	1	-1
-1	-1	1	-1	1	1	-1	2	2	-2

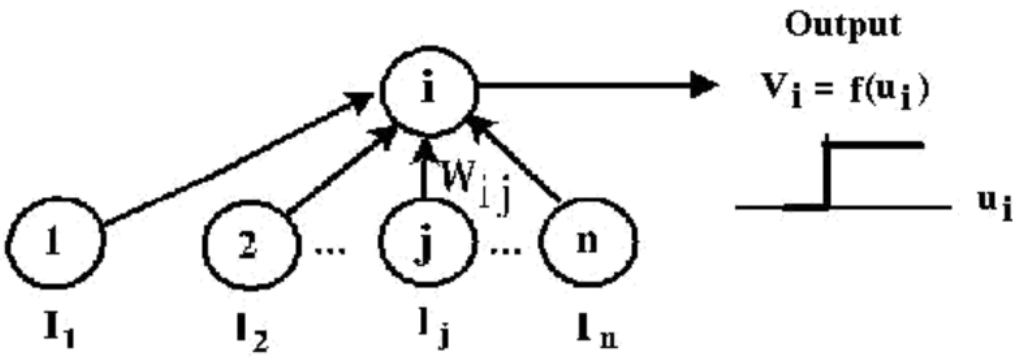


PERCEPTRON NETWORK

The perceptron network was introduced by Frank Rosenblatt in the year 1958. The perceptron had the following differences from the McCullough-Pitts neuron:

1. The weights and thresholds were not all identical.
2. Weights can be positive or negative.
3. There is no absolute inhibitory synapse.
4. Although the neurons were still two-state, the output function f(u) goes from [-1,1], not [0,1]. (This is no big deal, as a suitable change in the threshold lets you transform from one convention to the other.)
5. Most importantly, there was a learning rule.

Describing this in a slightly more modern and conventional notation (and with Vi = [0,1]) we could describe the perceptron like this:



This shows a perceptron unit, i, receiving various inputs Ij, weighted by a "synaptic weight" Wij.

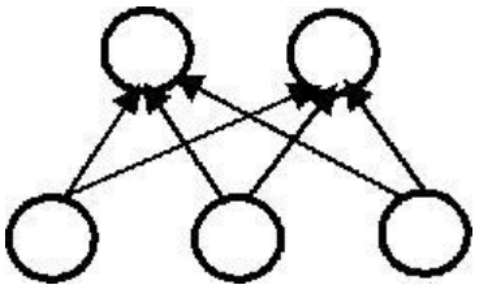
The ith perceptron receives its input from n input units, which do nothing but pass on the input from the outside world. The output of the perceptron is a step function:

$$V_i = f(u_i) = \begin{cases} 0 & : \quad u_i < 0 \\ 1 & : \quad u_i \geq 0 \end{cases}$$

$$u_i = \sum_j W_{ij} V_j + \theta_i$$

For the input units, Vj = Ij. There are various ways of implementing the threshold, or bias, thetai. Sometimes it is subtracted, instead of added to the input u, and sometimes it is included in the definition off (u).

A network of two perceptrons with three inputs would look like:



Note that they don't interact with each other - they receive inputs only from the outside. We call this a "single layer perceptron network" because the input units don't really count. They exist just to provide an output that is equal to the external input to the net.

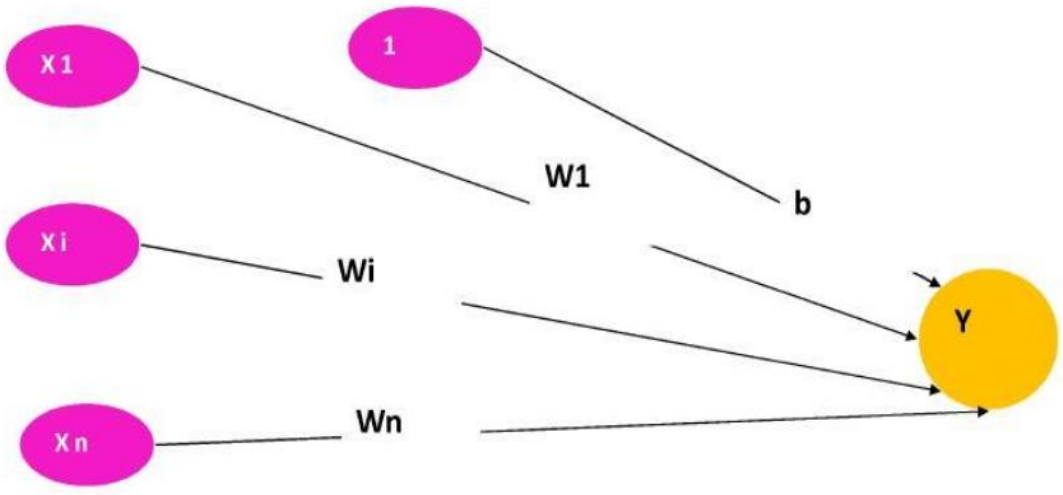
The learning scheme is very simple. Let ti be the desired "target" output for a given input pattern, and Vi be the actual output. The error (called "delta") is the difference between the desired and the actual output, and the change in the weight is chosen to be proportional to delta.

Specifically, $\delta_i = (t_i - V_i)$ and $\Delta W_{ij} = \epsilon \delta_i V_j$

where $0 \leq \epsilon < 1$. is the learning rate.

Note that if the output of the ith neuron is too small, the weights of all its inputs are changed to increase its total input. Likewise, if the output is too large, the weights are changed to decrease the total input.

Architecture



ALGORITHM

Step 0 Initialize weights and bias. Set learning rate α(0< α ≤ 1)

Step 1 While stopping condition is false do step 2-6 Step

2 For each training pair s:t do step 3-5

Step 3 Set activation s for input units Xi = Si Step

4 Compute response of the output unit

$$Y_{in} = b + \sum x_i\, w_i \qquad Y = \begin{cases} 1 & if\ Y_{in} > \theta \\ 0 & if\ -\theta \le Y_{in} \le \theta \\ -1 & if\ Y_{in} < \theta \end{cases}$$

Step 5 Update weights and bias if an error occurred in the pattern

$$\text{if} \quad Y \neq t \qquad W_i(new) = W_i(old) + \propto tX_i \qquad b(new) = b(old) + \propto t$$

$$\text{Else} \quad W_i(new) = W_i(old) \qquad b(new) = b(old)$$

Step 6 test stopping conditions. If no weights changed in step 2. Stop

else continue

Input			Y-in	Y	t	Wt changes			Weights		
Iteration 1											
X1	X2	B				Δw1	Δw2	Δb	W1	W2	b
1	1	1	0	0	1	1	1	1	1	1	1
1	0	1	2	1	-1	-1	0	-1	0	1	0
0	1	1	1	1	-1	0	-1	-1	0	0	-1
0	0	1	-1	-1	-1	0	0	0	0	0	-1

Iteration 2											
1	1	1	-1	-1	1	1	1	1	1	1	0
1	0	1	1	1	-1	-1	0	-1	0	1	-1
0	1	1	0	0	-1	0	-1	-1	0	0	-2
0	0	1	-2	-1	-1	0	0	0	0	0	-2

AND LOGIC
binary input
bipolar
output

α = 1 check for linear seperability

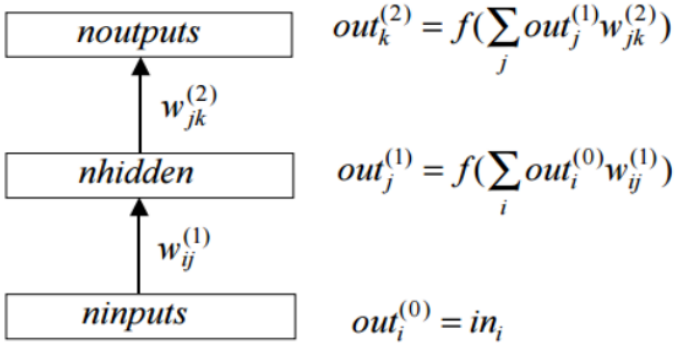
Iteration 10											
1	1	1	1	1	1	0	0	0	2	3	-4
1	0	1	-2	-1	-1	0	0	0	2	3	-4
0	1	1	-1	-1	-1	0	0	0	2	3	-4
0	0	1	-4	-1	-1	0	0	0	2	3	-4

AND LOGIC bipolar input bipolar output

Input			Y-in	Y	t	Wt changes			Weights		
Iteration 1											
X1	X2	B				Δw1	Δw2	Δb	W1	W2	b
1	1	1	0	0	1	1	1	1	1	1	1
1	-1	1	1	1	-1	-1	1	-1	0	2	0
-1	1	1	2	1	-1	1	-1	-1	1	1	-1
-1	-1	1	-3	-1	-1	0	0	0	1	1	-1
Iteration 2											
1	1	1	1	1	1	0	0	0	1	1	-1
1	-1	1	-1	-1	-1	0	0	0	1	1	-1
-1	1	1	-1	-1	-1	0	0	0	1	1	-1
-1	-1	1	-3	-1	-1	0	0	0	1	1	-1

α = 1 check for linear seperability

Multi-Layer Perceptrons (MLPs): To deal with non-linearly separable problems (such as XOR) we can use non-monotonic activation functions. More conveniently, we can instead extend the simple Perceptron to a Multi-Layer Perceptron, which includes a least one hidden layer of neurons with non-linear activations functions f(x) (such as sigmoids)



Training a Two-Layer MLP Network

The procedure for training a two layer MLP is now quite straight-forward:

1. Take the set of training (input – output) patterns the network is required to learn {ini p , outj p : i = 1 … ninputs, j = 1 … noutputs, p = 1 … npatterns} .

2. Set up a network with ninputs input units fully connected to nhidden hidden units via connections with weights wij (1) , which in turn are fully connected to noutputs output units via connections with weights wjk (2) .

3. Generate random initial connection weights, e.g. from the range

[–smwt, +smwt]

4. Select an appropriate error function E wjk n ( ) ( ) and learning rate η.

5. Apply the gradient descent weight update equation ∆wjk E w w n jk n jk ( ) ( ) (n) = −η∂ ( ) ∂ to each weight wjk (n) for each training pattern p. One set of updates of all the weights for all the training patterns is called one epoch of training.

6. Repeat step 5 until the network error function is 'small enough'.

The extension to networks with more hidden layers is straightforward.

Applications of Multi-Layer Perceptrons

Neural network applications fall into two basic types: Brain modelling The scientific goal of building models of how real brains work. This can potentially help us understand the nature of human intelligence, formulate better teaching strategies, or better remedial actions for brain damaged patients. Artificial System Building The

engineering goal of building efficient systems for real world applications. This may make machines more powerful, relieve humans of tedious tasks, and may even improve upon human performance. We often use exactly the same networks and techniques for both. Frequently progress is made when the two approaches are allowed to feed into each other. There are fundamental differences though, e.g. the need for biological plausibility in brain modelling, and the need for computational efficiency in artificial system building. Simple neural networks (MLPs) are surprisingly effective for both. Brain models need to cover Development, Adult Performance, and Brain Damage. Real world applications include: Data Compression, Time Series Prediction, Speech Recognition, Pattern Recognition and Computer Vision

DELTA RULE

In machine learning, the delta rule is a gradient descent learning rule for updating the weights of the inputs to artificial neurons in a single-layer neural network. It is a special

case of the more general backpropagation algorithm. For a neuron j with activation function $g(x)$, the delta rule for j's ith weight w_{ji} is given by

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i,$$

where

α is a small constant called *learning rate*

$g(x)$ is the neuron's activation function

t_j is the target output

h_j is the weighted sum of the neuron's inputs

y_j is the actual output

x_i is the ith input.

It holds that $h_j = \sum x_i w_{ji}$ and $y_j = g(h_j)$.

The delta rule is commonly stated in simplified form for a neuron with a linear activation function as

$$\Delta w_{ji} = \alpha(t_j - y_j)x_i$$

While the delta rule is similar to the perceptron's update rule, the derivation is different. The perceptron uses the Heaviside step function as the activation function $g(h)$, and

that means that $g'(h)$ does not exist at zero, and is equal to zero elsewhere, which makes the direct application of the delta rule impossible.

Derivation of the delta rule

The delta rule is derived by attempting to minimize the error in the output of the neural network through gradient descent. The error for a neural network with j outputs can be measured as

$$E = \sum_j \frac{1}{2}(t_j - y_j)^2$$

.

In this case, we wish to move through "weight space" of the neuron (the space of all possible values of all of the neuron's weights) in proportion to the gradient of the error

function with respect to each weight. In order to do that, we calculate the partial derivative of the error with respect to each weight. For the ith weight, this derivative can be written as

$$\frac{\partial E}{\partial w_{ji}}.$$

Because we are only concerning ourselves with the jth neuron, we can substitute the error formula above while omitting the summation:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \left(\frac{1}{2}\left(t_j - y_j\right)^2\right)}{\partial w_{ji}}$$

Next we use the chain rule to split this into two derivatives:

$$= \frac{\partial \left(\frac{1}{2}\left(t_j - y_j\right)^2\right)}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}}$$

$$= -\left(t_j - y_j\right) \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial w_{ji}}$$

To find the left derivative, we simply apply the general power rule:

$$= -\left(t_j - y_j\right) \frac{\partial y_j}{\partial w_{ji}}$$

To find the right derivative, we again apply the chain rule, this time differentiating with respect to the total input to j, h_j:

Note that the output of the jth neuron, y_j, is just the neuron's activation function g applied to the neuron's input h_j. We can therefore write the derivative of y_j with respect to h_j simply as g's first derivative:

$$= -\left(t_j - y_j\right) g'(h_j) \frac{\partial h_j}{\partial w_{ji}}$$

Next we rewrite h_j in the last term as the sum over all k weights of each weight w_{jk} times its corresponding input x_k:

$$= -\left(t_j - y_j\right) g'(h_j) \frac{\partial \left(\sum_k x_k w_{jk}\right)}{\partial w_{ji}}$$

Because we are only concerned with the ith weight, the only term of the summation that is relevant is $x_i w_{ji}$. Clearly,

$$\frac{\partial x_i w_{ji}}{\partial w_{ji}} = x_i$$

,

giving us our final equation for the gradient:

$$\frac{\partial E}{\partial w_{ji}} = -\left(t_j - y_j\right) g'(h_j) x_i$$

As noted above, gradient descent tells us that our change for each weight should be proportional to the gradient. Choosing a proportionality constant α and eliminating the

minus sign to enable us to move the weight in the negative direction of the gradient to minimize error, we arrive at our target equation:

$$\Delta w_{ji} = \alpha(t_j - y_j) g'(h_j) x_i.$$

Perceptron rule vs Delta rule

- Perceptron rule (target - thresholded output) guaranteed to converge to a separating hyperplane if the problem is linearly separable. Otherwise may not converge – could get in cycle
- Singe layer Delta rule guaranteed to have only one global minimum. Thus it will converge to the best SSE solution whether the problem is linearly separable or not.
    - Could have a higher misclassification rate than with the perceptron rule and a less intuitive decision surface – we will discuss with regression
- Stopping Criteria – For these models stop when no longer making progress

When you have gone a few epochs with no significant improvement/change between epochs (including oscillations

References

1. Zell, Andreas (1994). Simulation of Neural Networks (1st ed.). Addison-Wesley. p. 73. ISBN 3-89319-554-8.

2. Schmidhuber, Jürgen (2015-01-01). "Deep learning in neural networks: An overview". Neural Networks. 61: 85–117. arXiv:1404.7828. doi:10.1016/j.neunet.2014.09.003. ISSN 0893-6080.

3. Auer, Peter; Harald Burgsteiner; Wolfgang Maass (2008). "A learning rule for very simple universal approximators consisting of a single layer of perceptrons" (PDF). Neural Networks. 21 (5): 786–795.

4. Roman M. Balabin; Ravilya Z. Safieva; Ekaterina I. Lomakina (2007). "Comparison of linear and nonlinear calibration models based on near infrared (NIR) spectroscopy data for gasoline properties prediction".

5. Tahmasebi, Pejman; Hezarkhani, Ardeshir (21 January 2011). "Application of a Modular Feedforward Neural Network for Grade Estimation". Natural Resources Research. 20 (1): 25–32. doi:10.1007/s11053-011-9135-3. S2CID 45997840.