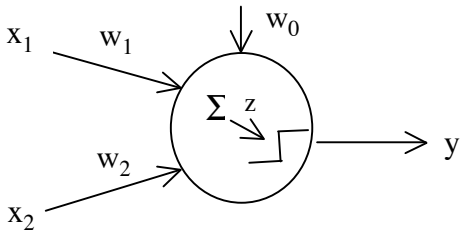
6.034 Recitation October 30: Neural Nets

Bob Berwick

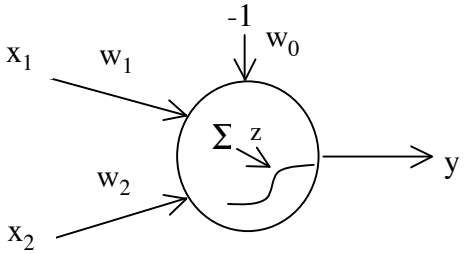
Neural nets are networks of simulated neurons, each of which has weighted inputs, an adder that sums the weighted inputs, a threshold function that checks the sum against a threshold and returns an output. By means of *forward propagation*, inputs are run through the network to produce outputs: At each level of the network, weighted inputs are summed, then run through either a step function or a sigmoid to produce an output. As with other machine learning techniques, the goal in using a neural net is to classify unknown inputs, i.e., assign a known class to an unknown input. (Output that is continuous rather than discrete implements regression rather than classification.)

A simulated neuron that employs a *step function* returns 1 or 0, depending on whether the input is above or below a specified threshold value, respectively. Neural nets of these simulated neurons, sometimes called perceptrons, can be thought of as linearly dividing a space of input vectors into regions, thereby creating decision boundaries. The weights in multi-layer perceptron nets cannot be automatically computed, i.e., learned, because of the discontinuity of the threshold function.

The more usual kind of neuron in a neural net employs a *sigmoid function.* Such sigmoid neural nets are extremely difficult to design by hand; weights are learned via a *backpropagation* algorithm. Two modifications to the step function idea enabled this automatic computation (and "nice" math): (1) representing a non-zero threshold as a weight with a –1 input value, so that the threshold becomes zero and can be computed automatically just as any other weight; (2) replacing the step function with a continuous (sigmoid) function, thus enabling gradient ascent (or descent) methods to be used in computing weights.

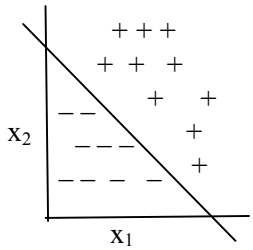


step function neuron

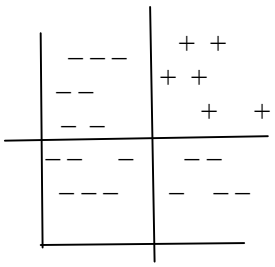


sigmoid neuron

Step function neurons: linear decision boundaries



(a) one decision boundary



(b) two decision boundaries needed for an AND

We can think of a step function net as defining a linear decision boundary, $ax + by - c = 0$. The sum of the weights of a step function neuron can represent the equation of the line: $w_1x_1 + w_2x_2 - w_0 = 0$. In example (a) above, to the left of the line can be represented by a 0 output; to the right a 1 output. Each decision line is represented by one neuron in the lowest layer of the net. Each class is represented by a neuron in the final layer,

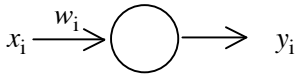
with two classes only requiring one neuron. Interior neurons implement Boolean combinations, which allow for classifying data sets such as that shown in (b) above.

Backpropagation Notes

New weights depend on a learning rate and the derivative of a performance function with respect to weights:

(1) $w_i' = w_i + r\frac{\partial P}{\partial w_i}$

Using the chain rule, where y_i designates a neuron's output:



(2) $\frac{\partial P}{\partial w_i} = \frac{\partial P}{\partial y_i}\frac{\partial y_i}{\partial w_i}$

For the standard performance function, where y^* is the final desired output and y is the actual final output, $P = -\frac{1}{2}\sum(y^* - y)^2$:

(3) $\frac{\partial P}{\partial y_i} = \frac{\partial}{\partial y_i}(-\frac{1}{2}(y^* - y)^2) = (y^* - y)$

For a neural net, the total input to a neuron is $z = \sum w_i x_i$

(Note that x_i is sometimes written y_i to indicate that in a multilayer network, the input for one node is the output for a previous layer node)

For a sigmoid neural net, the output of a neuron, where z is the total input, is $y = s(z) = \frac{1}{1+e^{-z}}$.

Recall that the derivative $\frac{\partial y}{\partial z} = y(1-y)$.

So for the <u>output</u> layer of a sigmoid neural net:

(4) $\frac{\partial y_i}{\partial w_i} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i} = y(1-y)x_i$

Substituting (3) and (4) into (2):

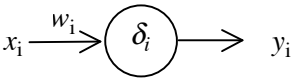
(5) $\frac{\partial P}{\partial w_i} = \frac{\partial P}{\partial y_i}\frac{\partial y_i}{\partial w_i} = (y^* - y)y(1-y)x_i$

Substituting (5) into the weight equation (1):

$w_i' = w_i + r(y^* - y)y(1-y)x_i$

... which can be written in terms of δ, which is the derivative of P with respect to total input, $\frac{\partial P}{\partial z}$:

$w_i' = w_i + r\delta_i x_i$, where $\delta_i = \frac{\partial P}{\partial y}\frac{\partial y}{\partial z} = (y^* - y)y(1-y) = y(1-y)(y^* - y)$

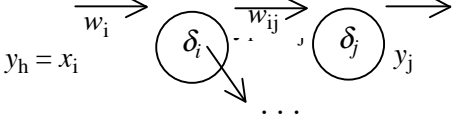


(Note that to change P, one only needs to substitute the new $\frac{\partial P}{\partial y}$ into this equation.)

For an <u>inner</u> node, we use a similar equation that takes into account the output y_i for that node, and the δ and w values for the <u>next</u> layer, where "next layer" means the neighboring nodes one layer <u>closer</u> to the output layer:

$w_i' = w_i + r\delta_i x_i$, where $\delta_i = y_i(1-y_i)\sum\delta_j w_{ij}$

and w_{ij} is the weight between layers i and j.



Summary: For each layer, $w_i' = w_i + r\delta_i x_i$, where for <u>outer</u> layer $\delta_i = y(1-y)(y^* - y)$

<u>inner</u> layer $\delta_i = y_i(1-y_i)\sum\delta_j w_{ij}$