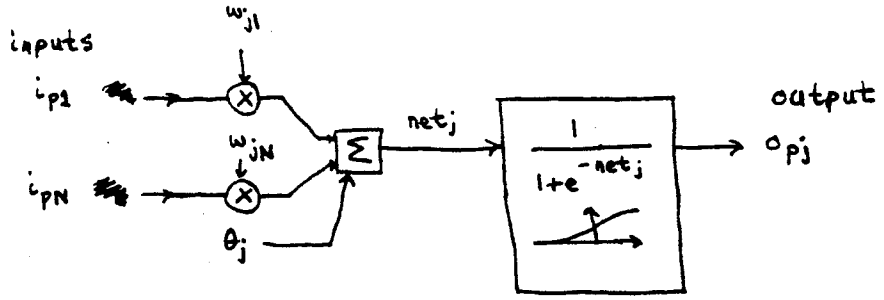
Gradient Descent —

10 May 1989
Neil E Cotter

Backward Error Propagation (BEP) - Problem Statement

BEP is the most popular learning algorithm for neural networks. BEP is associated with a particular network architecture in most people's minds but BEP is applicable to many kinds of networks. Our discussion will be confined to the architecture in common use. This network is <u>not</u> described by differential eqns. Rather, the network is a black box into which input information flows and out of which output information flows — instantly. We will see how the network is constructed, starting with a neuron and working our way up to a network. Just to entice you with what is ahead, we are going to be designing a box which can compute virtually any function of N variables. Such a box can be used in control systems that are far superior to those in use today. We will consider the specific case of controlling a robot.

Neuron Model



sigmoidal logistic squashing function $\frac{1}{1+e^{-x}} \equiv f(x)$

- A network is created from many neurons connected in layers.
- The neuron is almost identical to a Hopfield neuron but has no RC low-pass filter.
- The network utilizes a logistic function, $\frac{1}{1+e^{-x}}$, instead of a tanh x function.

Gradient Descent – BEP –

10 May 1989 Equations for the neuron

Neil E Cotter

notation: $i_{p1}, \ldots, i_{pN}$ — input pattern (vector)

p denotes which input pattern is being considered. Since the values of $i_1, \ldots, i_N$ tell us what the input pattern is, p is superfluous. The authors of the original papers on BEP used it, however, and I will too.

$w_{j1}, \ldots, w_{jN}$ — synaptic weight (vector) for neuron j

θ_j — bias or threshold for neuron

net_{pj} — sum of input activity

$f(\,)$ — (sigmoidal) logistic squashing function

o_{pj} — output of neuron j when input pattern is $i_{p1}, \ldots, i_{pN}$, i.e. p^{th} input pattern.

$$net_{pj} = \sum_{i=1}^{N} w_{ji}\, i_{pi} + \theta_j$$

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

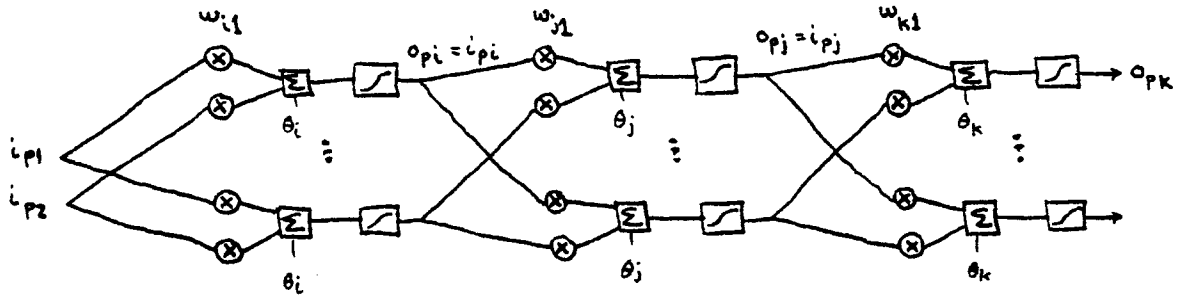
$$o_{pj} = f(net_{pj}) = \frac{1}{1+e^{-net_{pj}}} = \frac{1}{1+e^{-\left[\sum_{i=1}^{N} w_{ji}\, i_{pi} + \theta_j\right]}}$$

Output is squashed version of weighted sum of inputs.

Gradient Descent - BEP -

2 May 1989 Layered Network

Neil E Cotter



- We can have as many neurons on each layer as we want.
- All of the outputs from one layer go to all the inputs on the following layer.
- The network can have multiple outputs and can compute several different functions of the inputs that way.
- When the input changes, the outputs change instantly.
- We could have more than three layers, but three layers is enough for any computation.
- The network computes functions of the input variables. We can write $o_{pk} = g_k(i_{p1}, \ldots, i_{pN})$. The network can be trained to compute $g_k()$.

Backward Error Propagation

Suppose we want the network to learn a set of functions $o_{pk} = g_k(i_{p1}, \ldots, i_{pN})$. We have to know what we want the network to learn, so we have to know what value o_{pk} should have for inputs $i_{p1}, \ldots, i_{pN}$.

Q. If we know what $g()$ is, why do we need a neural network?

A. There are several answers to this. First, we might know what $g()$ is, but storing it in a machine other than a neural network might require excessive memory or computations.

12 May 1989

Neil E Cotter

Gradient Descent – BEP – Training

- Second, the function, g(), might arise from observations of some physical system such as weather. The function cannot be written down on paper perhaps.
- Third, the function, g(), might be known only for certain input values. Neural networks are good at interpolating between data points in a sensible way. Thus, the network is able to generalize from what it has learned.
- Fourth, neural networks process information in parallel and have a uniform structure. They can process information more rapidly than digital computers.

To learn, we adjust synaptic weights:

1) Specify a training pattern $i_{p1}, \ldots, i_{pN}$ and t_{pk} for each output neuron. i.e. both the input for the network and the correct output of the network for that input. (or target t_{pk})

2) Present $i_{p1}, \ldots, i_{pN}$ to the network and see what values of o_{pk} it computes.

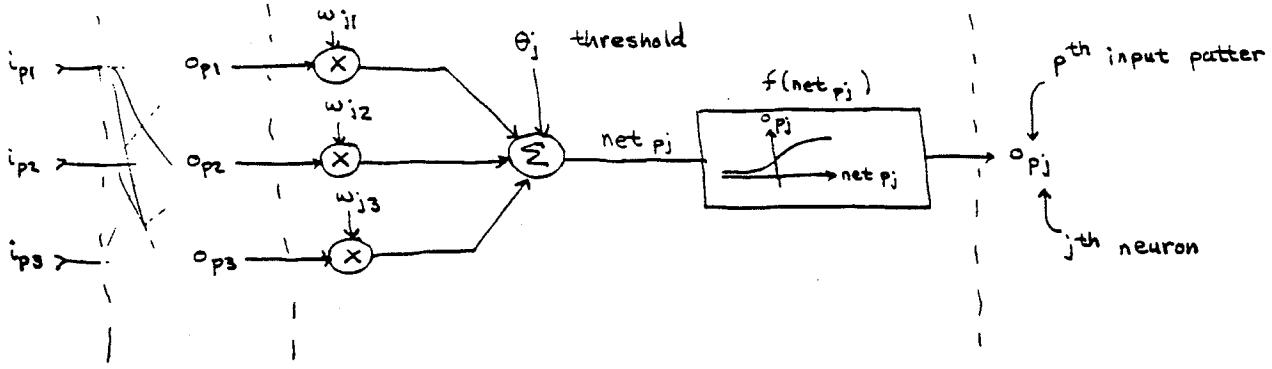
3) Compute the error $E_{pk} = \frac{(t_{pk} - o_{pk})^2}{2}$ at each output.

4) Use gradient descent to change the synaptic weights throughout the network so as to reduce the total error, $E_p = \sum_k E_{pk}$, at the output of the network.

5) Pick another training pattern, (hopefully $i_{p1}, \ldots, i_{pN}$ can be chosen at random), and go to 1). Repeat until the total error is as small as desired or stops changing.

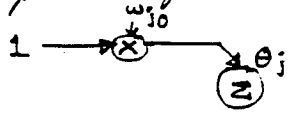
6 May 1988
Neil E Cotter

Gradient Descent — Backward Error Propagation — Sigmoids



- All neurons have the same form as the one shown for layer 2.
- The model is the same as for the perceptron with two exceptions: 1) The notation is different, and 2) the sigmoidal f() function replaces the step function ⎍ used in perceptions.

- Threshold θ_j can be replaced by a fixed input and a synaptic weight as in the perceptron:



- The sigmoidal function $f(net_{pj})$ is usually

Sigmoid: $$f(net_{pj}) = \frac{1}{1+e^{-net_{pj}}}$$

properties: $$f(0) = \frac{1}{1+e^{0}} = \frac{1}{2}$$

$$f(-\infty) = \frac{1}{1+e^{--\infty}} = 0$$

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

$$\partial f / \partial net_{pi} = f(1-f)$$