Variations on the BP algorithm.

Since BP was introduced in 1985, there have been many variations attempted, with the aim of

* making it run faster
* making it less likely to become stuck in local minima.

These variations extend from simple "improvements" to completely new versions.

Simple "Improvements".

1. Modifying the derivative of the output function.

Since one of the main reasons for the slowness of BP is the low value of the derivative

$$\frac{dy}{dx}$$

for large modulus values of x, one modification which has been tried is simply to add a constant (such as 0.05) to the derivative.

This has the effect of increasing the effective amount of error propagated back through hidden units whose outputs are near to 0 or 1, and so causing larger weight change steps to be taken.

In addition, completely different cost functions have been used (most notably the relative entropy measure, see Haykin, section 6.20).

2. Modifying η *to* η_{ij}.

As the basic algorithm stands, the learning rate parameter is fixed for all the variables. However, the value of the error derivative for different weights will vary.

One way of trying to avoid any problems this may produce is to

(i) alter w_{ij}

(ii) check whether this alteration really has decreased the error, and keep the alteration only if the error has decreased.

However, this is computationally very expensive. However, if used, it can allow η to be selected for each weight, so that $\eta = \eta_{ij}$. (i.e. η is a function of i and j)

This leads us to

New Versions.

BP is a 1st order gradient descent technique: that is, each weight change results in a step in weight space so that the error decreases. It may well be that the effect of changing a complete set of weights is to increase the error, or, at least, not to decreases the error in an optimal way.

Second order methods can change the weight vector (**W**) all at once, giving steepest descent instead.

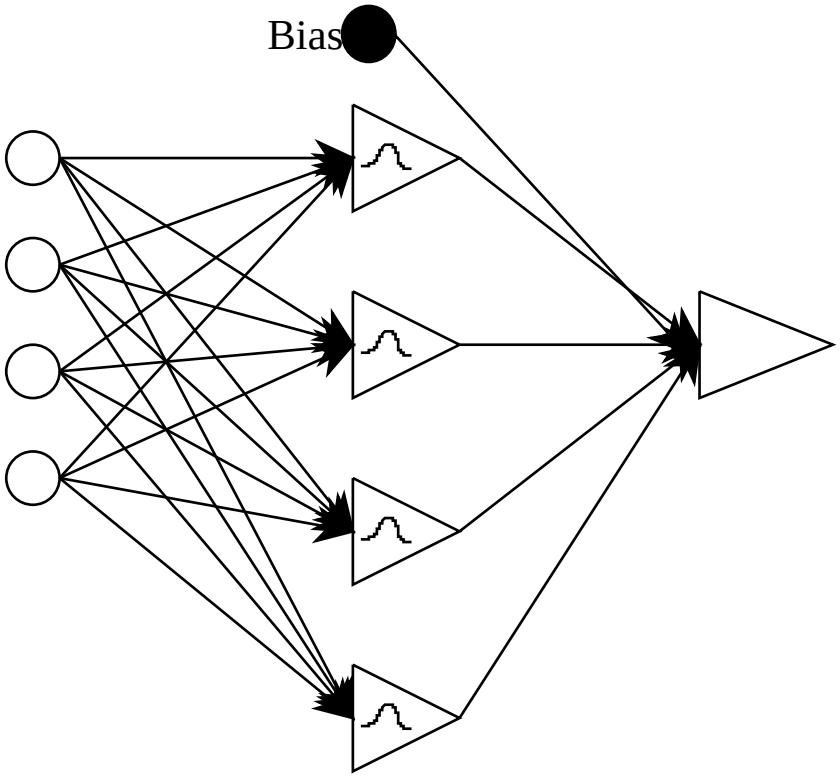
Such algorithms are described in detail in Hertz, Krogh, and Palmer, pp 124-129.

In essence, these second order methods require much more computation, as they require the 2nd derivative matrix (the Hessian) of the error.

This results in the computation becoming non-local, since the actual weight change to be made at a weight will depend on the error derivative at all the other weights. Any pretence at a straightforward parallel implementation is lost. However, as simply numerical methods for solving problems, these methods can be attractive.

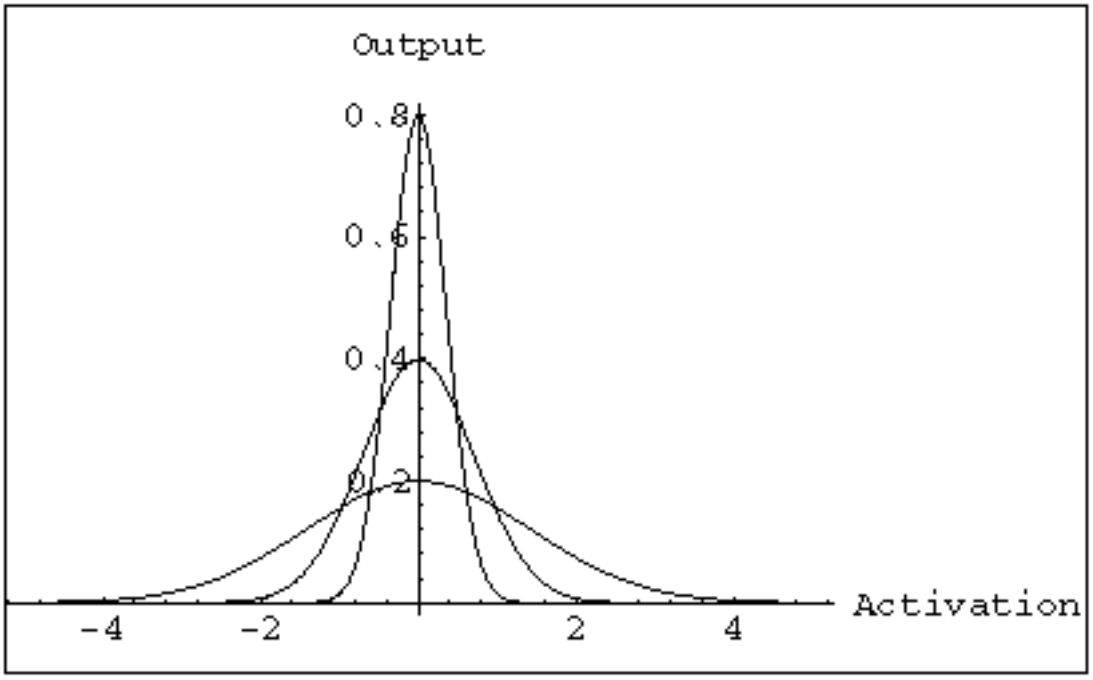
Radial Basis Functions.

A different approach to the use of feedforward neural networks. Though the architecture looks like a standard 3-layer BP network, there is a crucial difference.



The difference is that the hidden units embody a different type of neuron altogether. These units have a *receptive field*: that is, they have their maximal response (generally 1) for some particular point in the input space, and this response tails off as the input value moves away from this point.

One can show such a response for a 1-dimensional input space:



In general, Gaussian units are used: that is the unit has a Gaussian response function.

Each hidden unit, h, computes

Gauss((**I** - $\mathbf{I}_h$,$\mathbf{r}_h$), where

I is the input vector
$\mathbf{I}_h$ is the centre of the receptive field of this hidden unit
$\mathbf{r}_h$ is the radius of the Gaussian.

Gauss(**0**, r) =1, and its value tails away to 0 as the first argument increases in distance away from 0. The speed at which it tends to 0 will depend on **r**.

The output unit (there may be many: only one is shown here, for clarity), may be a simple linear unit, or it may be a logistic unit or a tanh unit.

Notes:

1: Each RBF in the hidden layer responds to input only in some subspace of the input space. When the input is far away from it own centre, $\mathbf{I}_h$, (many radii away), then the output of that unit will be so small as to be ignorable.

2: The hidden layer recodes the input space using the RBF outputs. Each RBF has a receptive field: that is, an area of the input space to which it responds. Not only is this more biologically plausible (since many sensory neurons respond only to some small subspace of the input space, and are silent in response to all other inputs), but it also allows us to have large numbers of units in the hidden layer without adding (much) to the complexity of the total mapping.

Training the Radial Basis Function system.

To train the system, we need to select:

$\mathbf{I}_h$ for each hidden unit

$\mathbf{r}_h$ for each hidden unit

W (the weight vector from the RBF layer to the output unit) for each output unit.

There are a number of techniques used:

1: Simplest technique.

Choose the $\mathbf{I}_h$ to coincide with the different training inputs.

Choose $\mathbf{r}_h$ = r (i.e. fixed for all the units, and the same in all directions) so that

$$r = \frac{d}{\sqrt{2M}}$$

where d is the maximal distance between the $\mathbf{I}_h$, and M is the number of RBF units.

1a: Cluster the inputs first:

If there is a great deal of training data, using 1 per RBF unit may be impossible. Instead, one can choose M points in the input space, selecting them so that they cover the same region as the input.

After this, one simply trains the W using the delta rule.

This type of training clearly takes place in two distinct steps: choosing the centres and radii of the RBF layer, then training up the weights. The way in which the centres are chosen may involve the use of a *clustering technique,* of which more later.

2: Supervised technique.

In this technique, the RBF centres and radii are initially selected in a similar way to the above (clustered), and the output layer of weights trained.

Then, one attempts to alter the centers and the radii using a gradient descent technique to "fine-tune" the centres, radii, and weights.

This is still an area of development: however, initial results suggest that RBF's do offer an effective alternative to BP: with the more sophisticated training system, their results are certainly comparable to those of BP.

Notes:

RBF's are usable for both classification and for mapping, in the same way as BP systems are.

The RBF technique has a good mathematical background based in the theory of functional approximation. For more information, see Haykin, chapter 7.

Note that if $\mathbf{r}_h$ is small, then one is really training up the system just to use each RBF unit for just one input in the training set. Thus, one can get a lack of generalisation, just as one can in BP. If $\mathbf{r}_h$ is large, one can get overgeneralisation.

Using the supervised learning algorithm to choose the centres and the radii can optimise performance.

One can see how the RBF has selected the centres for the RBFs. In this way, it is easier to get an explanation for the behaviour of an RBF system than it is for a BP system.

If one knows that a problem is particularly sensitive to some small area of the input space, one can choose to place an RBF unit with centre there: thus domain knowledge can be used to guide network construction.

RBFs and Fuzzy Logic: RBF based networks are one way of including fuzzy-set theoretic information into neural networks.