Generalisation.

The subtext of the last section can be stated simply: simply recalling the correct output pattern for each input in the supplied training set (including the test set, and the validation set) is not enough.

(This is why making NNs learn simple logical predicates by showing them all the cases, though instructive, must be treated with caution)

What we require is for the system to *generalise*: that is, to process unseen data appropriately.

Unfortunately, *appropriately* depends on the application, and is hard to predefine.

We can identify two forms of generalisation appropriate to the two forms of problem we discussed, *classification* problems and *mapping* problems.

Classification:
Typically there are T output units, and in the training data, exactly one of these outputs is 1 and all the rest are 0. In other words, the training input vectors are classified as belonging to exactly one of the T classes.

Conceptually, the input training data set, I, has been chosen from some set S, and we want the network to produce a classification which extends the classification provided so that it gives the desired classification on the set S–I.

Clearly:
(i) this is not predefinable without reference to the problem at hand.
(ii) good generalisation is not directly testable, at least not without gathering more information from the system being modelled.

Note that we usually have to introduce some form of *interpretation procedure* for unseen data: for example, the classification is taken to be the class associated with the output unit with the largest output, even if this output is relatively small. One can introduce measures of the certainty of a classification based on the actual output values.

Good performance means that the percentage of correct generalisation is adequate for the task at hand.

Mapping: in mapping problems, the range of output values in the training data is larger than in classification problems, typically some interval of values inside the possible range of each output unit.

For example, there might be two output units, each with a logistic output function (so that their possible range of output is (0,1)). The range of actual outputs in the training data might be [0.14, 0.83] for one, and [0.2, 0.86] for the other.

Unseen inputs will generally have a different input vector from any supplied in the training data, and will, generally, require to be mapped to an output different from any of the outputs in the training data. What the network is doing is modelling some function whose output covers some interval: new input vectors have their output calculated by the function modelled by the network.

There are two possible cases here: the network may be performing *interpolation*, that is the input vector is "inside" the set of vectors supplied in training (mathematically, inside the convex hull defined by the training input vectors), or it may be performing *extrapolation* (i.e. the unseen input vector is outside the convex hull defined by the training input vectors.).

Networks generally perform interpolation reasonably well: extrapolation is generally much more difficult.

Note that this discussion of generalisation is necessarily not general. Some problems are a mixture of of classification and mapping: others don’t really fall into either type.

Hints on applying BP.

(i) ***Data-massaging***

* The input data and the output data need to be in an appropriate range.

* Logistic units which are to be sensitive to small changes on some input line around 0 will be swamped by large input values. If some input line has input which has a large range, but sometimes the output is sensitive to small changes around (say) 0, then better performance may be achieved by recoding the input using more than one input unit.

For example, one might replace a value X by

$$X_1 = \log_{10}(|\ X\ | + 1)$$

and

$$X_2 = \frac{1}{1 + \exp(-10X)}$$

so that X_1 will compress X into a small range, and X_2 will be more sensitive to small changes in X around 0.

* Highly compact representations are frequently inappropriate. Such representations are often optimal for total storage requirements, but achieve this at the expense of similarity in representation of similar entities. A good example is binary representation of integers:

15 = 01111
16 = 10000

These two representations are a different as possible, yet represent integers only 1 different!

It is generally better to use a representation in which there is some redundancy. Such representations are inevitably longer (when counted in bits of information), and this can lead to different problems.

*Systems with large numbers of input units (i.e. high-dimensional input spaces) need more training data. There are more weights to be specified by the training data. In general, one should choose representations to suit both the problem at hand, and the net method of solution chosen.

*Where possible, choose the training set so that as much as possible of the problem input and output spaces are used. (Clearly, this may not be possible if the training data is supplied).

(ii) ***Choosing the net topology***.

The network is intended to form a model of the system from which the training data was extracted. We want the model to both conform to the training data, and to generalise appropriately. To this end one should use as few hidden units as possible because

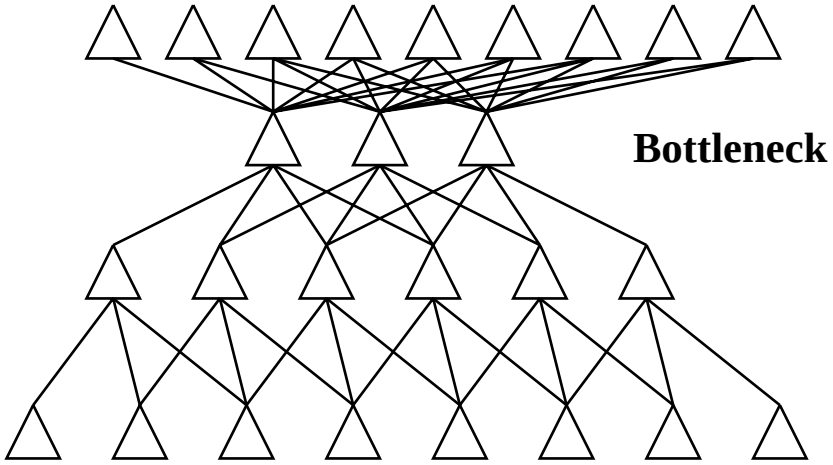
* all the weights need to be set by the training data. The more weights there are, the more training data will be required to set them.

* if you use too many hidden units, the net may well converge to the training data, but be *overspecialised*.

Additionally, one should use as few hidden layers as possible. Using more than a small number of hidden layers will result in the net being slow to train. Unfortunately, there are occasions when more than two hidden layers is best: these tend to occur when very similar inputs require very different outputs, so that one hidden layer is used to recode the data prior to the second layer which is used to perform the mapping required.

We usually are best to choose the simplest topology which can be trained to a reasonable approximation of the training data. By using a simple topology, we are *constraining the nature of the mapping* producible by the network: the simpler the topology, the simpler the range of possible mappings producible.

Often a "bottleneck" in the hidden layer can help this, since it forces the final output to be based on a small number of variables.



However, if too narrow a bottleneck is used, the net will fail to converge.

* generally, use a completely connected feedforward network, unless you have good reason to believe that certain inputs should be processed independently of others.

(iii) Choosing the initial weights.

The initial weights cannot be 0, as this would lead to none of the error being back-propagated.

It is best to choose weights which are small. Generally, they are chosen from a uniform distribution on [-0.5, 0.5], for example.

In this way, one starts off in an area of the output function where the derivative is not too small.

Where a unit has a large number of inputs, it may be advantageous to use smaller weights.

(iv) ***Choosing the network operating parameters***.

One needs to select

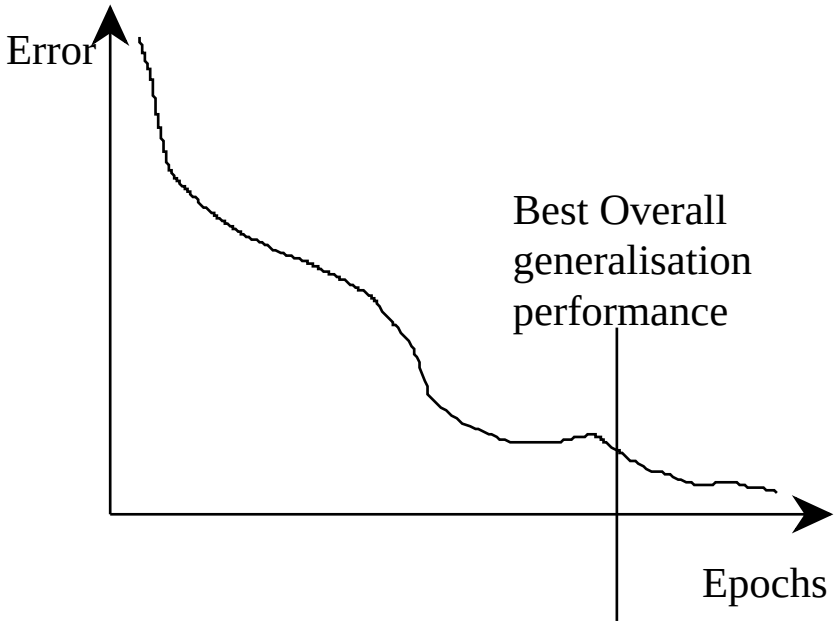
η the learning rate
α the momentum parameter
ε the acceptable error level,
and the number of epochs of training to be attempted before giving up.

We have already discussed ε. η and α interact: additionally, the best values to use depend on the problem. This is because they decide the weight change size as a function of the gradient of the error at this particular point in W-space. The nature of the error as a function of the weights is completely problem-dependent.

It can be useful to start with η high, and gradually reduce it. Often the best value to use requires some experimentation.

Overspecialisation.

Training the network for too long on some training data can result in poor generalisation performance. This can be a particular problem when rather too large a hidden layer has been used.



Using too many hidden units tends to give the same effect. What happens is that the network develops a transfer function which attempts to explain the input/output pairs precisely.

Function example (blackboard!).