Archive-name: ai-faq/neural-nets/part2
Last-modified: 2002-10-11
URL: ftp://ftp.sas.com/pub/neural/FAQ2.html
Maintainer: saswss@unx.sas.com (Warren S. Sarle)

This is part 2 (of 7) of a monthly posting to the Usenet newsgroup comp.ai.neural-nets. See the part 1 of this posting for full information what it is all about.

========== Questions ==========

Subject: What are combination, activation, error, and objective functions?

Most neural networks involve combination, activation, error, and objective functions.

Combination functions

Each non-input unit in a neural network combines values that are fed into it via synaptic connections from other units, producing a single value called the "net input". There is no standard term in the NN literature for the function that combines values. In this FAQ, it will be called the "combination function". The combination function is a vector-to scalar function. Most NNs use either a linear combination function (as in MLPs) or a Euclidean distance combination function (as in RBF networks). There is a detailed discussion of networks using these two kinds of combination function under "How do MLPs compare with RBFs?"

Activation functions

Most units in neural networks transform their net input by using a scalar-to-scalar function called an "activation function", yielding a value called the unit's "activation". Except possibly for output units, the activation value is fed via synaptic connections to one or more other units. The activation function is sometimes called a "transfer", and activation functions with a bounded range are often called "squashing" functions, such as the commonly used tanh (hyperbolic tangent) and logistic (1/(1+exp(-x)))) functions. If a unit does not transform its net input, it is said to have an "identity" or "linear" activation function. The reason for using non-identity activation functions is explained under "Why use activation functions?"

Error functions

Most methods for training supervised networks require a measure of the discrepancy between the networks output value and the target (desired output) value (even unsupervised networks may require such a measure of discrepancy--see "What does unsupervised learning learn?").

Let:

- `j` be an index for cases

- `X` or `X_j` be an input vector
- `W` be a collection (vector, matrix, or some more complicated structure) of weights and possibly other parameter estimates
- `y` or `y_j` be a target scalar
- `M(X,W)` be the output function computed by the network (the letter `M` is used to suggest "mean", "median", or "mode")
- `p` or `p_j = M(X_j,W)` be an output (the letter `p` is used to suggest "predicted value" or "posterior probability")
- `r` or `r_j = y_j - p_j` be a residual
- `Q(y,X,W)` be the case-wise error function written to show the dependence on the weights explicitly
- `L(y,p)` be the case-wise error function in simpler form where the weights are implicit (the letter `L` is used to suggest "loss" function)
- `D` be a list of indices designating a data set, including inputs and target values
  - `DL` designate the training (learning) set
  - `DV` designate the validation set
  - `DT` designate the test set
- `#(D)` be the number of elements (cases) in `D`
  - `NL` be the number of cases in the training (learning) set
  - `NV` be the number of cases in the validation set
  - `NT` be the number of cases in the test set
- `TQ(D,W)` be the total error function
- `AQ(D,W)` be the average error function

The difference between the target and output values for case `j, r_j = y_j - p_j,` is called the "residual" or "error". This is *NOT* the "error function"! Note that the residual can be either positive or negative, and negative residuals with large absolute values are typically considered just as bad as large positive residuals. Error functions, on the other hand, are defined so that bigger is worse.

Usually, an error function `Q(y,X,W)` is applied to each case and is defined in terms of the target and output values `Q(y,X,W) = L(y,M(X,W)) = L(y,p)`. Error functions are also called "loss" functions, especially when the two usages of the term "error" would sound silly when used together. For example, instead of the awkward phrase "squared-error error", you would typically use "squared-error loss" to mean an error function equal to the squared residual, `L(y,p) = (y - p)^2`. Another common error function is the classification loss for a binary target `y` in {0, 1}:

```
L(y,p) = 0 if |y-p| < 0.5
         1 otherwise
```

The error function for an entire data set is usually defined as the sum of the case-wise error functions for all the cases in a data set:

```
TQ(D,W) =  sum    Q(y_j,X_j,W)
          j in D
```

Thus, for squared-error loss, the total error is the sum of squared errors (i.e., residuals), abbreviated SSE. For classification loss, the total error is the number of misclassified cases.

It is often more convenient to work with the average (i.e, arithmetic mean) error:

```
AQ(D,W) = TQ(D,W)/#(D)
```

For squared-error loss, the average error is the mean or average of squared errors (i.e., residuals), abbreviated MSE or ASE (statisticians have a slightly different meaning for MSE in linear models, `TQ(D,W)/[#(D)-#(W)]`). For classification loss, the average error is the proportion of misclassified cases.

The average error is also called the "empirical risk."

Using the average error instead of the total error is especially convenient when using batch backprop-type training methods where the user must supply a learning rate to multiply by the negative gradient to compute the change in the weights. If you use the gradient of the average error, the choice of learning rate will be relatively insensitive to the number of training cases. But if you use the gradient of the total error, you must use smaller learning rates for larger training sets. For example, consider any training set `DL_1` and a second training set `DL_2` created by duplicating every case in `DL_1`. For any set of weights, `DL_1` and `DL_2` have the same average error, but the total error of `DL_2` is twice that of `DL_1`. Hence the gradient of the total error of `DL_2` is twice the gradient for `DL_1`. So if you use the gradient of the total error, the learning rate for `DL_2` should be half the learning rate for `DL_1`. But if you use the gradient of the average error, you can use the same learning rate for both training sets, and you will get exactly the same results from batch training.

The term "error function" is commonly used to mean any of the functions, `Q(y,X,W)`, `L(y,p)`, `TQ(D,W)`, or `AQ(D,W)`. You can usually tell from the context which function is the intended meaning. The term "error surface" refers to `TQ(D,W)` or `AQ(D,W)` as a function of `W`.

Objective functions

The objective function is what you directly try to minimize during training.

Neural network training is often performed by trying to minimize the total error `TQ(DL,W)` or, equivalently, the average error `AQ(DL,W)` for the training set, as a function of the weights `W`. However, as discussed in Part 3 of the FAQ, minimizing training error can lead to overfitting and poor generalization if the number of training cases is small relative to the complexity of the network. A common approach to improving generalization error is regularization, i.e., trying to minimize an objective function that is the sum of the total error function and a regularization function. The regularization function is a function of the weights `W` or of the output function `M(X,W)`. For example, in weight decay, the regularization function is the sum of squared weights. A crude form of Bayesian learning can be done using a regularization function that is the log of the prior density of the weights (weight decay is a special case of this). For more information on regularization, see Part 3 of the FAQ.

If no regularization function is used, the objective function is equal to the total or average error function (or perhaps some other monotone function thereof).

Subject: What are batch, incremental, on-line, off-line, deterministic, stochastic, adaptive, instantaneous, pattern, constructive, and sequential learning?

There are many ways to categorize learning methods. The distinctions are overlapping and can be confusing, and the terminology is used very inconsistently. This answer attempts to impose some order on the chaos, probably in vain.

Batch vs. Incremental Learning (also Instantaneous, Pattern, and Epoch)

Batch learning proceeds as follows:

```
Initialize the weights.
Repeat the following steps:
```

```
   Process all the training data.
   Update the weights.
```

Incremental learning proceeds as follows:

```
Initialize the weights.
Repeat the following steps:
   Process one training case.
   Update the weights.
```

In the above sketches, the exact meaning of "Process" and "Update" depends on the particular training algorithm and can be quite complicated for methods such as Levenberg-Marquardt (see "What are conjugate gradients, Levenberg-Marquardt, etc.?"). Standard backprop (see What is backprop?) is quite simple, though. Batch standard backprop (without momentum) proceeds as follows:

```
Initialize the weights W.
Repeat the following steps:
   Process all the training data DL to compute the gradient
      of the average error function AQ(DL,W).
   Update the weights by subtracting the gradient times the
      learning rate.
```

Incremental standard backprop (without momentum) can be done as follows:

```
Initialize the weights W.
Repeat the following steps for j = 1 to NL:
   Process one training case (y_j,X_j) to compute the gradient
      of the error (loss) function Q(y_j,X_j,W).
   Update the weights by subtracting the gradient times the
      learning rate.
```

Alternatively, the index `j` can be chosen randomly each type the loop is executed, or `j` can be chosen from a random permutation.

The question of when to stop training is very complicated. Some of the possibilities are:

- Stop when the average error function for the training set becomes small.
- Stop when the gradient of the average error function for the training set becomes small.
- Stop when the average error function for the validation set starts to go up, and use the weights from the step that yielded the smallest validation error. For details, see "What is early stopping?"
- Stop when your boredom level is no longer tolerable.

It is very important *NOT* to use the following method--*which does not work*--but is often mistakenly used by beginners:

```
Initialize the weights W.
Repeat the following steps for j = 1 to NL:
   Repeat the following steps until Q(y_j,X_j,W) is small:
      Compute the gradient of Q(y_j,X_j,W).
      Update the weights by subtracting the gradient times the
         learning rate.
```

The reason this method does not work is that by the time you have finished processing the second case, the network will have forgotten what it learned about the first case, and when you are finished with the third case, the network will have forgotten what it learned about the first two cases, and so on. If you really need to use a method like this, see the section below on "Sequential Learning, Catastrophic Interference, and the Stability-Plasticity Dilemma".

The term "batch learning" is used quite consistently in the NN literature, but "incremental learning" is often used for on-line, constructive, or sequential learning. The usage of "batch" and "incremental" in the

NN FAQ follows Bertsekas and Tsitsiklis (1996), one of the few references that keeps the relevant concepts straight.

"Epoch learning" is synonymous with "batch learning."

"Instantaneous learning" and "pattern learning" are usually synonyms for incremental learning. "Instantaneous learning" is a misleading term because people often think it means learning instantaneously. "Pattern learning" is easily confused with "pattern recognition". Hence these terms are not used in the FAQ.

There are also intermediate methods, sometimes called mini-batch:

```
Initialize the weights.
Repeat the following steps:
   Process two or more, but not all training cases.
   Update the weights.
```

Conventional numerical optimization techniques (see "What are conjugate gradients, Levenberg-Marquardt, etc.?") are batch algorithms. Conventional stochastic approximation techniques (see below) are incremental algorithms. For a theoretical discussion comparing batch and incremental learning, see Bertsekas and Tsitsiklis (1996, Chapter 3).

For more information on incremental learning, see Saad (1998), but note that the authors in that volume do not reliably distinguish between on-line and incremental learning.

On-line vs. Off-line Learning

In off-line learning, all the data are stored and can be accessed repeatedly. Batch learning is always off-line.

In on-line learning, each case is discarded after it is processed and the weights are updated. On-line training is always incremental.

Incremental learning can be done either on-line or off-line.

With off-line learning, you can compute the objective function for any fixed set of weights, so you can see whether you are making progess in training. You can compute a minimum of the objective function to any desired precision. You can use a variety of algorithms for avoiding bad local minima, such as multiple random initializations or global optimization algorithms. You can compute the error function on a validation set and use early stopping or choose from different networks to improve generalization. You can use cross-validation and bootstrapping to estimate generalization error. You can compute prediction and confidence intervals (error bars). With on-line learning you can do none of these things because you cannot compute the objective function on the training set or the error function on the validation set for a fixed set of weights, since these data sets are not stored. Hence, on-line learning is generally more difficult and unreliable than off-line learning. Off-line incremental learning does not have all these problems of on-line learning, which is why it is important to distinguish between the concepts of on-line and incremental learning.

Some of the theoretical difficulties of on-line learning are alleviated when the assumptions of stochastic learning hold (see below) and training is assumed to proceed indefinitely.

For more information on on-line learning, see Saad (1998), but note that the authors in that volume do not reliably distinguish between on-line and incremental learning.

Deterministic, Stochastic, and Adaptive Learning

Deterministic learning is based on optimization of an objective function that can be recomputed many times and always produces the same value given the same weights. Deterministic learning is always off-line.

Stochastic methods are used when computation of the objective function is corrupted by noise. In particular, basic stochastic approximation is a form of on-line gradient descent learning in which the training cases are obtained by a stationary random process:

```
Initialize the weights.
Initialize the learning rate.
Repeat the following steps:
   Randomly select one (or possibly more) case(s)
      from the population.
   Update the weights by subtracting the gradient
      times the learning rate.
   Reduce the learning rate according to an
      appropriate schedule.
```

In stochastic on-line learning, the noise-corrupted objective function is the error function for any given case, assuming that the case-wise error function has some stationary random distribution. The learning rate must gradually decrease towards zero during training to guarantee convergence of the weights to a local minimum of the noiseless objective function. This gradual reduction of the learning rate is often called "annealing."

If the function that the network is trying to learn changes over time, the case-wise error function does not have a stationary random distribution. To allow the network to track changes over time, the learning rate must be kept strictly away from zero. Learning methods that track a changing environment are often called "adaptive" (as in adaptive vector quantization, Gersho and Gray, 1992) or "continuous" rather than "stochastic". There is a trade-off between accuracy and speed of adaptation. Adaptive learning does not converge in a stationary environment. Hence the long-run properties of stochastic learning and adaptive learning are quite different, even though the algorithms may differ only in the sequence of learning rates.

The object of adaptive learning is to forget the past when it is no longer relevant. If you want to remember the past in a changing learning environment, then you would be more interested in sequential learning (see below).

In stochastic learning with a suitably annealed learning rate, overtraining does not occur because the more you train, the more data you have, and the network converges toward a local optimum of the objective function for the entire population, not a local optimum for a finite training set. Of course, this conclusion does not hold if you train by cycling through a finite training set instead of collecting new data on every step.

For a theoretical discussion of stochastic learning, see Bertsekas and Tsitsiklis (1996, Chapter 4). For further references on stochastic approximation, see "What is backprop?" For adaptive filtering, see Haykin (1996).

The term "adaptive learning" is sometimes used for gradient methods in which the learning rate is changed during training.

Constructive Learning (Growing networks)

Constructive learning adds units or connections to the network during training. Typically, constructive learning begins with a network with no hidden units, which is trained for a while. Then without altering

the existing weights, one or more new hidden units are added to the network, training resumes, and so on. Many variations are possible, involving different patterns of connections and schemes for freezing and thawing weights. The most popular constructive algorithm is cascade correlation (Fahlman and Lebiere, 1990), of which many variations are possible (e.g., Littmann and Ritter, 1996; Prechelt, 1997). Various other constructive algorithms are summarized in Smieja (1993), Kwok and Yeung (1997; also other papers at http://info.cs.ust.hk/faculty/dyyeung/paper/cnn.html), and Reed and Marks (1999). For theory, see Baum (1989), Jones (1992), and Meir and Maiorov (1999). Lutz Prechelt has a bibliography on constructive algorithms at http://wwwipd.ira.uka.de/~prechelt/NN/construct.bib

Constructive algorithms can be highly effective for escaping bad local minima of the objective function, and are often much faster than algorithms for global optimization such as simulated annealing and genetic algorithms. A well-known example is the two-spirals problem, which is very difficult to learn with standard backprop but relatively easy to learn with cascade correlation (Fahlman and Lebiere, 1990). Some of the Donoho-Johnstone benchmarks (especially "bumps") are almost impossible to learn with standard backprop but can be learned very accurately with constructive algorithms (see ftp://ftp.sas.com/pub/neural/dojo/dojo.html.)

Constructive learning is commonly used to train multilayer perceptrons in which the activation functions are step functions. Such networks are difficult to train nonconstructively because the objective function is discontinuous and gradient-descent methods cannot be used. Several clever constructive algorithms (such as Upstart, Tiling, Marchand, etc.) have been devised whereby a multilayer perceptron is constructed by training a sequence of perceptrons, each of which is trained by some standard method such as the well-known perceptron or pocket algorithms. Most constructive algorithms of this kind are designed so that the training error goes to zero when the network gets sufficiently large. Such guarantees do not apply to the generalization error; you should guard against overfitting when you are using constructive algorithms just as with nonconstructive algorithms (see part 3 of the FAQ, especially "What is overfitting and how can I avoid it?")

Logically, you would expect "destructive" learning to start with a large network and delete units during training, but I have never seen this term used. The process of deleting units or connections is usually called "pruning" (Reed, 1993; Reed and Marks 1999). The term "selectionism" has also been used as the opposite of "constructivism" in cognitive neuroscience (Quartz and Sejnowski, 1997).

Sequential Learning, Catastrophic Interference, and the Stability-Plasticity Dilemma

"Sequential learning" sometimes means incremental learning but also refers to a very important problem in neuroscience (e.g., McClelland, McNaughton, and O'Reilly 1995). To reduce confusion, the latter usage should be preferred.

Sequential learning in its purest form operates on a sequence of training sets as follows:

```
Initialize the weights.
Repeat the following steps:
   Collect one or more training cases.
   Train the network on the current training set
      using any method.
   Discard the current training set and all other
      information related to training except the
      weights.
```

Pure sequential learning differs from on-line learning in that most on-line algorithms require storage of information in addition to the weights, such as the learning rate or aproximations to the objective function or Hessian Matrix. Such additional storage is not allowed in pure sequential learning.

Pure sequential learning is important as a model of how learning occurs in real, biological brains. Humans and other animals are often observed to learn new things without forgetting old things. But pure sequential learning tends not to work well in artificial neural networks. With a fixed architecture, distributed (rather than local) representations, and a training algorithm based on minimizing an objective function, sequential learning results in "catastrophic interference", because the minima of the objective function for one training set may be totally diferent than the minima for subsequent training sets. Hence each successive training set may cause the network to forget completely all previous training sets. This problem is also called the "stability-plasticity dilemma."

Successful sequential learning usually requires one or more of the following:

- Noise-free data.
- Constructive architectures.
- Local representations.
- Storage of extra information besides the weights (this is impure sequential learning and is not biologically plausible unless a a biological mechanism for storing the extra information is provided)

The connectionist literature on catastrophic interference seems oblivious to statistical and numerical theory, and much of the research is based on the ludicrous idea of using an autoassociative backprop network to model recognition memory. Some of the problems with this approach are explained by Sharkey and Sharkey (1995). Current research of the PDP variety is reviewed by French (1999). PDP remedies for catastrophic forgetting generally require cheating, i.e., storing information outside the network. For example, pseudorehearsal (Robins, 1995) requires storing the distribution of the inputs, although this fact is often overlooked. It appears that the only way to avoid catastrophic interference in a PDP-style network is to combine two networks modularly: a fast-learning network to memorize data, and a slow-learning network to generalize from data memorized by the fast-learning network (McClelland, McNaughton, and O'Reilly 1995). The PDP literature virtually ignores the ART literature (See "What is ART?"), which provides a localist constructive solution to what the ART people call the "stability-plasticity dilemma." None of this research deals with sequential learning in a statistically sound manner, and many of the methods proposed for sequential learning require noise-free data. The statistical theory of sufficient statistics makes it obvious that efficient sequential learning requires the storage of additional information besides the weights in a standard feedforward network. I know of no references to this subject in the NN literature, but Bishop (1991) provided a mathematically sound treatment of a closely-related problem.

References:

Baum, E.B. (1989), "A proposal for more powerful learning algorithms," Neural Computation, 1, 201-207.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-10-8.

Bishop, C. (1991), "A fast procedure for retraining the multilayer perceptron," International Journal of Neural Systems, 2, 229-236.

Fahlman, S.E., and Lebiere, C. (1990), "The Cascade-Correlation Learning Architecture", in Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems 2,*, Los Altos, CA: Morgan Kaufmann Publishers, pp. 524-532, ftp://archive.cis.ohio-state.edu/pub/neuroprose/fahlman.cascor-tr.ps.Z, http://www.rafael-ni.hpg.com.br/arquivos/fahlman-cascor.pdf

French, R.M. (1999), "Catastrophic forgetting in connectionist networks: Causes, consequences and solutions," Trends in Cognitive Sciences, 3, 128-135, http://www.fapse.ulg.ac.be/Lab/Trav

/rfrench.html#TICS_cat_forget

Gersho, A. and Gray, R.M. (1992), *Vector Quantization and Signal Compression,* Boston: Kluwer Academic Publishers.

Haykin, S. (1996), *Adaptive Filter Theory,* Englewood Cliffs, NJ: Prentice-Hall.

Jones, L. (1992), "A simple lemma on greedy approximation in Hilbert space and convergence rate for projection pursuit regression and neural network training," Annals of Statistics, 20, 608-613.

Kwok, T.Y. and Yeung, D.Y. (1997), "Constructive algorithms for structure learning in feedforward neural networks for regression problems," IEEE Transactions on Neural Networks, volume 8, 630-645.

Littmann, E., and Ritter, H. (1996), "Learning and generalization in cascade network architectures," Neural Computation, 8, 1521-1539.

McClelland, J., McNaughton, B. and O'Reilly, R. (1995), "Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory," Psychological Review, 102, 419-457.

Meir, R., and Maiorov, V. (1999), "On the optimality of incremental neural network algorithms," in Kerans, M.S., Solla, S.A., amd Cohn, D.A. (eds.), *Advances in Neural Information Processing Systems 11,* Cambridge,MA: The MIT Press, pp. 295-301.

Prechelt, L. (1997), "Investigation of the CasCor Family of Learning Algorithms," Neural Networks, 10, 885-896, http://wwwipd.ira.uka.de/~prechelt/Biblio/#CasCor

Quartz, S.R., and Sejnowski, T.J. (1997), "The neural basis of cognitive development: A constructivist manifesto," Behavioral and Brain Sciences, 20, 537-596, ftp://ftp.princeton.edu/pub/harnad/BBS/bbs.quartz

Reed, R. (1993), "Pruning algorithms--A survey," IEEE Transactions on Neural Networks, 4, 740-747.

Reed, R.D., and Marks, R.J, II (1999), *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks,* Cambridge, MA: The MIT Press, ISBN 0-262-18190-8.

Robins, A. (1995), "Catastrophic Forgetting, Rehearsal, and Pseudorehearsal," Connection Science, 7, 123-146.

Saad, D., ed. (1998), *On-Line Learning in Neural Networks,* Cambridge: Cambridge University Press.

Sharkey, N.E., and Sharkey, A.J.C. (1995), "An analysis of catastrophic interference," Connection Science, 7, 301-329.

Smieja, F.J. (1993), "Neural Network Constructive Algorithms: Trading Generalization for Learning Efficiency?" Circuits, Systems and Signal Processing, 12, 331-374, ftp://borneo.gmd.de/pub/as/janus/pre_6.ps

---

Subject: What is backprop?

"Backprop" is short for "backpropagation of error". The term *backpropagation* causes much confusion. Strictly speaking, *backpropagation* refers to the method for computing the gradient of the case-wise error function with respect to the weights for a feedforward network, a straightforward but elegant application of the chain rule of elementary calculus (Werbos 1974/1994). By extension, *backpropagation* or *backprop* refers to a training method that uses backpropagation to compute the gradient. By further extension, a *backprop* network is a feedforward network trained by backpropagation.

"Standard backprop" is a euphemism for the *generalized delta rule*, the training algorithm that was popularized by Rumelhart, Hinton, and Williams in chapter 8 of Rumelhart and McClelland (1986), which remains the most widely used supervised training method for neural nets. The generalized delta rule (including momentum) is called the "heavy ball method" in the numerical analysis literature (Polyak 1964, 1987; Bertsekas 1995, 78-79).

Standard backprop can be used for both batch training (in which the weights are updated after processing the entire training set) and incremental training (in which the weights are updated after processing each case). For batch training, standard backprop usually converges (eventually) to a local minimum, if one exists. For incremental training, standard backprop does not converge to a stationary point of the error surface. To obtain convergence, the learning rate must be slowly reduced. This methodology is called "stochastic approximation" or "annealing".

The convergence properties of standard backprop, stochastic approximation, and related methods, including both batch and incremental algorithms, are discussed clearly and thoroughly by Bertsekas and Tsitsiklis (1996). For a practical discussion of backprop training in MLPs, Reed and Marks (1999) is the best reference I've seen.

For batch processing, there is no reason to suffer through the slow convergence and the tedious tuning of learning rates and momenta of standard backprop. Much of the NN research literature is devoted to attempts to speed up backprop. Most of these methods are inconsequential; two that are effective are Quickprop (Fahlman 1989) and RPROP (Riedmiller and Braun 1993). Concise descriptions of these algorithms are given by Schiffmann, Joost, and Werner (1994) and Reed and Marks (1999). But conventional methods for nonlinear optimization are usually faster and more reliable than any of the "props". See "What are conjugate gradients, Levenberg-Marquardt, etc.?".

Incremental backprop can be highly efficient for some large data sets if you select a good learning rate, but that can be difficult to do (see "What learning rate should be used for backprop?"). Also, incremental backprop is very sensitive to ill-conditioning (see ftp://ftp.sas.com/pub/neural/illcond/illcond.html).

For more on-line info on backprop, see Donald Tveter's Backpropagator's Review at http://www.dontveter.com/bpr/bpr.html or http://gannoo.uce.ac.uk/bpr/bpr.html.

References on backprop:

Bertsekas, D. P. (1995), *Nonlinear Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-14-0.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-10-8.

Polyak, B.T. (1964), "Some methods of speeding up the convergence of iteration methods," Z. Vycisl. Mat. i Mat. Fiz., 4, 1-17.

Polyak, B.T. (1987), *Introduction to Optimization,* NY: Optimization Software, Inc.

Reed, R.D., and Marks, R.J, II (1999), *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks,* Cambridge, MA: The MIT Press, ISBN 0-262-18190-8.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986), "Learning internal representations by error propagation", in Rumelhart, D.E. and McClelland, J. L., eds. (1986), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* Volume 1, 318-362, Cambridge, MA: The MIT Press.

Werbos, P.J. (1974/1994), *The Roots of Backpropagation,* NY: John Wiley & Sons. Includes Werbos's 1974 Harvard Ph.D. thesis, *Beyond Regression.*

References on stochastic approximation:

Robbins, H. & Monro, S. (1951), "A Stochastic Approximation Method", Annals of Mathematical Statistics, 22, 400-407.

Kiefer, J. & Wolfowitz, J. (1952), "Stochastic Estimation of the Maximum of a Regression Function," Annals of Mathematical Statistics, 23, 462-466.

Kushner, H.J., and Yin, G. (1997), *Stochastic Approximation Algorithms and Applications,* NY: Springer-Verlag.

Kushner, H.J., and Clark, D. (1978), *Stochastic Approximation Methods for Constrained and Unconstrained Systems*, Springer-Verlag.

White, H. (1989), "Some Asymptotic Results for Learning in Single Hidden Layer Feedforward Network Models", J. of the American Statistical Assoc., 84, 1008-1013.

References on better props:

Fahlman, S.E. (1989), "Faster-Learning Variations on Back-Propagation: An Empirical Study", in Touretzky, D., Hinton, G, and Sejnowski, T., eds., *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, 38-51.

Reed, R.D., and Marks, R.J, II (1999), *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks,* Cambridge, MA: The MIT Press, ISBN 0-262-18190-8.

Riedmiller, M. (199?), "Advanced supervised learning in multi-layer perceptrons--from backpropagtion to adaptive learning algorithms," ftp://i11s16.ira.uka.de/pub/neuro/papers/riedml.csi94.ps.Z

Riedmiller, M. and Braun, H. (1993), "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", *Proceedings of the IEEE International Conference on Neural Networks* 1993, San Francisco: IEEE.

Schiffmann, W., Joost, M., and Werner, R. (1994), "Optimization of the Backpropagation Algorithm for Training Multilayer Perceptrons," ftp://archive.cis.ohio-state.edu/pub/neuroprose/schiff.bp_speedup.ps.Z

Subject: What learning rate should be used for backprop?

In standard backprop, too low a learning rate makes the network learn very slowly. Too high a learning

rate makes the weights and objective function diverge, so there is no learning at all. If the objective function is quadratic, as in linear models, good learning rates can be computed from the Hessian matrix (Bertsekas and Tsitsiklis, 1996). If the objective function has many local and global optima, as in typical feedforward NNs with hidden units, the optimal learning rate often changes dramatically during the training process, since the Hessian also changes dramatically. Trying to train a NN using a constant learning rate is usually a tedious process requiring much trial and error. For some examples of how the choice of learning rate and momentum interact with numerical condition in some very simple networks, see ftp://ftp.sas.com/pub/neural/illcond/illcond.html

With batch training, there is no need to use a constant learning rate. In fact, there is no reason to use standard backprop at all, since vastly more efficient, reliable, and convenient batch training algorithms exist (see Quickprop and RPROP under "What is backprop?" and the numerous training algorithms mentioned under "What are conjugate gradients, Levenberg-Marquardt, etc.?").

Many other variants of backprop have been invented. Most suffer from the same theoretical flaw as standard backprop: the magnitude of the change in the weights (the step size) should NOT be a function of the magnitude of the gradient. In some regions of the weight space, the gradient is small and you need a large step size; this happens when you initialize a network with small random weights. In other regions of the weight space, the gradient is small and you need a small step size; this happens when you are close to a local minimum. Likewise, a large gradient may call for either a small step or a large step. Many algorithms try to adapt the learning rate, but any algorithm that multiplies the learning rate by the gradient to compute the change in the weights is likely to produce erratic behavior when the gradient changes abruptly. The great advantage of Quickprop and RPROP is that they do not have this excessive dependence on the magnitude of the gradient. Conventional optimization algorithms use not only the gradient but also second-order derivatives or a line search (or some combination thereof) to obtain a good step size.

With incremental training, it is much more difficult to concoct an algorithm that automatically adjusts the learning rate during training. Various proposals have appeared in the NN literature, but most of them don't work. Problems with some of these proposals are illustrated by Darken and Moody (1992), who unfortunately do not offer a solution. Some promising results are provided by by LeCun, Simard, and Pearlmutter (1993), and by Orr and Leen (1997), who adapt the momentum rather than the learning rate. There is also a variant of stochastic approximation called "iterate averaging" or "Polyak averaging" (Kushner and Yin 1997), which theoretically provides optimal convergence rates by keeping a running average of the weight values. I have no personal experience with these methods; if you have any solid evidence that these or other methods of automatically setting the learning rate and/or momentum in incremental training actually work in a wide variety of NN applications, please inform the FAQ maintainer (saswss@unx.sas.com).

References:

Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-10-8.

Darken, C. and Moody, J. (1992), "Towards faster stochastic gradient search," in Moody, J.E., Hanson, S.J., and Lippmann, R.P., eds. *Advances in Neural Information Processing Systems 4,* San Mateo, CA: Morgan Kaufmann Publishers, pp. 1009-1016.

Kushner, H.J., and Yin, G. (1997), *Stochastic Approximation Algorithms and Applications,* NY: Springer-Verlag.

LeCun, Y., Simard, P.Y., and Pearlmetter, B. (1993), "Automatic learning rate maximization by

on-line estimation of the Hessian's eigenvectors," in Hanson, S.J., Cowan, J.D., and Giles, C.L. (eds.), *Advances in Neural Information Processing Systems 5,* San Mateo, CA: Morgan Kaufmann, pp. 156-163.

Orr, G.B. and Leen, T.K. (1997), "Using curvature information for fast stochastic search," in Mozer, M.C., Jordan, M.I., and Petsche, T., (eds.) *Advances in Neural Information Processing Systems 9,* Cambrideg, MA: The MIT Press, pp. 606-612.

Subject: What are conjugate gradients, Levenberg-Marquardt, etc.?

Training a neural network is, in most cases, an exercise in numerical optimization of a usually nonlinear objective function ("objective function" means whatever function you are trying to optimize and is a slightly more general term than "error function" in that it may include other quantities such as penalties for weight decay; see "What are combination, activation, error, and objective functions?" for more details).

Methods of nonlinear optimization have been studied for hundreds of years, and there is a huge literature on the subject in fields such as numerical analysis, operations research, and statistical computing, e.g., Bertsekas (1995), Bertsekas and Tsitsiklis (1996), Fletcher (1987), and Gill, Murray, and Wright (1981). Masters (1995) has a good elementary discussion of conjugate gradient and Levenberg-Marquardt algorithms in the context of NNs.

There is no single best method for nonlinear optimization. You need to choose a method based on the characteristics of the problem to be solved. For objective functions with continuous second derivatives (which would include feedforward nets with the most popular differentiable activation functions and error functions), three general types of algorithms have been found to be effective for most practical purposes:

- For a small number of weights, stabilized Newton and Gauss-Newton algorithms, including various Levenberg-Marquardt and trust-region algorithms, are efficient. The memory required by these algorithms is proportional to the square of the number of weights.
- For a moderate number of weights, various quasi-Newton algorithms are efficient. The memory required by these algorithms is proportional to the square of the number of weights.
- For a large number of weights, various conjugate-gradient algorithms are efficient. The memory required by these algorithms is proportional to the number of weights.

Additional variations on the above methods, such as limited-memory quasi-Newton and double dogleg, can be found in textbooks such as Bertsekas (1995). Objective functions that are not continuously differentiable are more difficult to optimize. For continuous objective functions that lack derivatives on certain manifolds, such as ramp activation functions (which lack derivatives at the top and bottom of the ramp) and the least-absolute-value error function (which lacks derivatives for cases with zero error), subgradient methods can be used. For objective functions with discontinuities, such as threshold activation functions and the misclassification-count error function, Nelder-Mead simplex algorithm and various secant methods can be used. However, these methods may be very slow for large networks, and it is better to use continuously differentiable objective functions when possible.

All of the above methods find local optima--they are not guaranteed to find a global optimum. In practice, Levenberg-Marquardt often finds better optima for a variety of problems than do the other usual methods. I know of no theoretical explanation for this empirical finding.

For global optimization, there are also a variety of approaches. You can simply run any of the local optimization methods from numerous random starting points. Or you can try more complicated methods designed for global optimization such as simulated annealing or genetic algorithms (see Reeves 1993 and "What about Genetic Algorithms and Evolutionary Computation?"). Global optimization for neural nets is especially difficult because the number of distinct local optima can be astronomical.

In most applications, it is advisable to train several networks with different numbers of hidden units. Rather than train each network beginning with completely random weights, it is usually more efficient to use constructive learning (see "Constructive Learning (Growing networks"), where the weights that result from training smaller networks are used to initialize larger networks. Constructive learning can be done with any of the conventional optimization techniques or with the various "prop" methods, and can be very effective at finding good local optima at less expense than full-blown global optimization methods.

Another important consideration in the choice of optimization algorithms is that neural nets are often ill-conditioned (Saarinen, Bramley, and Cybenko 1993), especially when there are many hidden units. Algorithms that use only first-order information, such as steepest descent and standard backprop, are notoriously slow for ill-conditioned problems. Generally speaking, the more use an algorithm makes of second-order information, the better it will behave under ill-conditioning. The following methods are listed in order of increasing use of second-order information: steepest descent, conjugate gradients, quasi-Newton, Gauss-Newton, Newton-Raphson. Unfortunately, the methods that are better for severe ill-conditioning are the methods that are preferable for a small number of weights, and the methods that are preferable for a large number of weights are not as good at handling severe ill-conditioning. Therefore for networks with many hidden units, it is advisable to try to alleviate ill-conditioning by standardizing input and target variables, choosing initial values from a reasonable range, and using weight decay or Bayesian regularization methods. For more discussion of ill-conditioning in neural nets, see ftp://ftp.sas.com/pub/neural/illcond/illcond.html

Writing programs for conventional optimization algorithms is considerably more difficult than writing programs for standard backprop. As "Jive Dadson" said in comp.ai.neural-nets:

> Writing a good conjugate gradient algorithm turned out to be a lot of work. It's not even easy to find all the technical info you need. The devil is in the details. There are a lot of details.

Indeed, some popular books on "numerical recipes" are notoriously bad (see http://math.jpl.nasa.gov/nr/ for details). If you are not experienced in both programming and numerical analysis, use software written by professionals instead of trying to write your own. For a survey of optimization software, see Moré and Wright (1993).

For more on-line information on numerical optimization see:

- The kangaroos, a nontechnical description of various optimization methods, at ftp://ftp.sas.com/pub/neural/kangaroos.
- Sam Roweis's paper on Levenberg-Marquardt at http://www.gatsby.ucl.ac.uk/~roweis/notes.html
- Jonathan Shewchuk's paper on conjugate gradients, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," at http://www.cs.cmu.edu/~jrs/jrspapers.html
- Lester Ingber's page on Adaptive Simulated Annealing (ASA), karate, etc. at http://www.ingber.com/ or http://www.alumni.caltech.edu/~ingber/
- The Netlib repository, http://www.netlib.org/, containing freely available software, documents, and databases of interest to the numerical and scientific computing community.
- The linear and nonlinear programming FAQs at http://www.mcs.anl.gov/home/otc/Guide/faq/.
- Arnold Neumaier's page on global optimization at http://solon.cma.univie.ac.at/~neum/glopt.html.
- Simon Streltsov's page on global optimization at http://cad.bu.edu/go.

References:

Bertsekas, D. P. (1995), *Nonlinear Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-14-0.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming,* Belmont, MA: Athena Scientific, ISBN 1-886529-10-8.

Fletcher, R. (1987) *Practical Methods of Optimization,* NY: Wiley.

Gill, P.E., Murray, W. and Wright, M.H. (1981) *Practical Optimization,* Academic Press: London.

Levenberg, K. (1944) "A method for the solution of certain problems in least squares," Quart. Appl. Math., 2, 164-168.

Marquardt, D. (1963) "An algorithm for least-squares estimation of nonlinear parameters," SIAM J. Appl. Math., 11, 431-441. This is the third most frequently cited paper in all the mathematical sciences.

Masters, T. (1995) *Advanced Algorithms for Neural Networks: A C++ Sourcebook,* NY: John Wiley and Sons, ISBN 0-471-10588-0

Moré, J.J. (1977) "The Levenberg-Marquardt algorithm: implementation and theory," in Watson, G.A., ed., *Numerical Analysis,* Lecture Notes in Mathematics 630, Springer-Verlag, Heidelberg, 105-116.

Moré, J.J. and Wright, S.J. (1993), *Optimization Software Guide,* Philadelphia: SIAM, ISBN 0-89871-322-6.

Reed, R.D., and Marks, R.J, II (1999), *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks,* Cambridge, MA: The MIT Press, ISBN 0-262-18190-8.

Reeves, C.R., ed. (1993) *Modern Heuristic Techniques for Combinatorial Problems,* NY: Wiley.

Rinnooy Kan, A.H.G., and Timmer, G.T., (1989) *Global Optimization: A Survey,* International Series of Numerical Mathematics, vol. 87, Basel: Birkhauser Verlag.

Saarinen, S., Bramley, R., and Cybenko, G. (1993), "Ill-conditioning in neural network training problems," Siam J. of Scientific Computing, 14, 693-714.

Subject: How does ill-conditioning affect NN training?

Numerical condition is one of the most fundamental and important concepts in numerical analysis. Numerical condition affects the speed and accuracy of most numerical algorithms. Numerical condition is especially important in the study of neural networks because ill-conditioning is a common cause of slow and inaccurate results from backprop-type algorithms. For more information, see:
ftp://ftp.sas.com/pub/neural/illcond/illcond.html

Subject: How should categories be encoded?

First, consider unordered categories. If you want to classify cases into one of C categories (i.e. you have a categorical target variable), use 1-of-C coding. That means that you code C binary (0/1) target variables corresponding to the C categories. Statisticians call these "dummy" variables. Each dummy variable is given the value zero except for the one corresponding to the correct category, which is given the value one. Then use a softmax output activation function (see "What is a softmax activation function?") so that the net, if properly trained, will produce valid posterior probability estimates (McCullagh and Nelder, 1989; Finke and Müller, 1994). If the categories are Red, Green, and Blue, then the data would look like this:

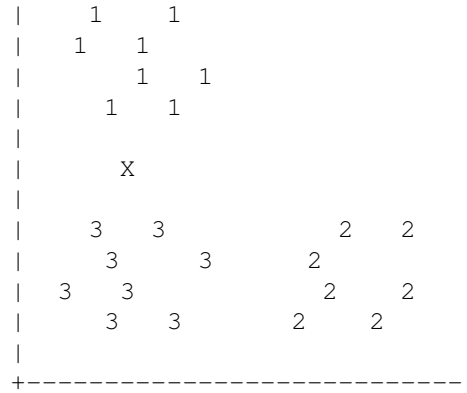
```
Category  Dummy variables
--------  ---------------
 Red        1   0   0
 Green      0   1   0
 Blue       0   0   1
```

When there are only two categories, it is simpler to use just one dummy variable with a logistic output activation function; this is equivalent to using softmax with two dummy variables.

The common practice of using target values of .1 and .9 instead of 0 and 1 prevents the outputs of the network from being directly interpretable as posterior probabilities, although it is easy to rescale the outputs to produce probabilities (Hampshire and Pearlmutter, 1991, figure 3). This practice has also been advocated on the grounds that infinite weights are required to obtain outputs of 0 or 1 from a logistic function, but in fact, weights of about 10 to 30 will produce outputs close enough to 0 and 1 for all practical purposes, assuming standardized inputs. Large weights will not cause overflow if the activation functions are coded properly; see "How to avoid overflow in the logistic function?"

Another common practice is to use a logistic activation function for each output. Thus, the outputs are not constrained to sum to one, so they are not admissible posterior probability estimates. The usual justification advanced for this procedure is that if a test case is not similar to any of the training cases, all of the outputs will be small, indicating that the case cannot be classified reliably. This claim is incorrect, since a test case that is not similar to any of the training cases will require the net to extrapolate, and extrapolation is thoroughly unreliable; such a test case may produce all small outputs, all large outputs, or any combination of large and small outputs. If you want a classification method that detects novel cases for which the classification may not be reliable, you need a method based on probability density estimation. For example, see "What is PNN?".

It is very important *not* to use a single variable for an unordered categorical target. Suppose you used a single variable with values 1, 2, and 3 for red, green, and blue, and the training data with two inputs looked like this:



Consider a test point located at the X. The correct output would be that X has about a 50-50 chance of being a 1 or a 3. But if you train with a single target variable with values of 1, 2, and 3, the output for X will be the average of 1 and 3, so the net will say that X is definitely a 2!

If you are willing to forego the simple posterior-probability interpretation of outputs, you can try more elaborate coding schemes, such as the error-correcting output codes suggested by Dietterich and Bakiri (1995).

For an input with categorical values, you can use 1-of-(C-1) coding if the network has a bias unit. This is just like 1-of-C coding, except that you omit one of the dummy variables (doesn't much matter which one). Using all C of the dummy variables creates a linear dependency on the bias unit, which is not advisable unless you are using weight decay or Bayesian learning or some such thing that requires all C weights to be treated on an equal basis. 1-of-(C-1) coding looks like this:

```
Category  Dummy variables
--------  ---------------
 Red        1   0
 Green      0   1
 Blue       0   0
```

If you use 1-of-C or 1-of-(C-1) coding, it is important to standardize the dummy inputs; see "Should I standardize the input variables?" "Why not code binary inputs as 0 and 1?" for details.

Another possible coding is called "effects" coding or "deviations from means" coding in statistics. It is like 1-of-(C-1) coding, except that when a case belongs to the category for the omitted dummy variable, all of the dummy variables are set to -1, like this:

```
Category  Dummy variables
--------  ---------------
 Red        1   0
 Green      0   1
 Blue      -1  -1
```

As long as a bias unit is used, any network with effects coding can be transformed into an equivalent network with 1-of-(C-1) coding by a linear transformation of the weights, so if you train to a global optimum, there will be no difference in the outputs for these two types of coding. One advantage of effects coding is that the dummy variables require no standardizing, since effects coding directly produces values that are approximately standardized.

If you are using weight decay, you want to make sure that shrinking the weights toward zero biases ('bias' in the statistical sense) the net in a sensible, usually smooth, way. If you use 1 of C-1 coding for an input, weight decay biases the output for the C-1 categories towards the output for the 1 omitted category, which is probably not what you want, although there might be special cases where it would make sense. If you use 1 of C coding for an input, weight decay biases the output for all C categories roughly towards the mean output for all the categories, which is smoother and usually a reasonable thing to do.

Now consider ordered categories. For inputs, some people recommend a "thermometer code" (Smith, 1996; Masters, 1993) like this:

```
Category  Dummy variables
--------  ---------------
 Red        1   1   1
 Green      0   1   1
 Blue       0   0   1
```

However, thermometer coding is equivalent to 1-of-C coding, in that for any network using 1-of-C coding, there exists a network with thermometer coding that produces identical outputs; the weights in the thermometer-encoded network are just the differences of successive weights in the 1-of-C-encoded network. To get a genuinely ordinal representation, you must constrain the weights connecting the dummy variables to the hidden units to be nonnegative (except for the first dummy variable). Another approach that makes some use of the order information is to use weight decay or Bayesian learning to encourage

the the weights for all but the first dummy variable to be small.

It is often effective to represent an ordinal input as a single variable like this:

```
Category  Input
--------  -----
 Red        1
 Green      2
 Blue       3
```

Although this representation involves only a single quantitative input, given enough hidden units, the net is capable of computing nonlinear transformations of that input that will produce results equivalent to any of the dummy coding schemes. But using a single quantitative input makes it easier for the net to use the order of the categories to generalize when that is appropriate.

B-splines provide a way of coding ordinal inputs into fewer than C variables while retaining information about the order of the categories. See Brown and Harris (1994) or Gifi (1990, 365-370).

Target variables with ordered categories require thermometer coding. The outputs are thus cumulative probabilities, so to obtain the posterior probability of any category except the first, you must take the difference between successive outputs. It is often useful to use a proportional-odds model, which ensures that these differences are positive. For more details on ordered categorical targets, see McCullagh and Nelder (1989, chapter 5).

References:

Brown, M., and Harris, C. (1994), *Neurofuzzy Adaptive Modelling and Control,* NY: Prentice Hall.

Dietterich, T.G. and Bakiri, G. (1995), "Error-correcting output codes: A general method for improving multiclass inductive learning programs," in Wolpert, D.H. (ed.), *The Mathematics of Generalization: The Proceedings of the SFI/CNLS Workshop on Formal Approaches to Supervised Learning,* Santa Fe Institute Studies in the Sciences of Complexity, Volume XX, Reading, MA: Addison-Wesley, pp. 395-407.

Finke, M. and Müller, K.-R. (1994), "Estimating a-posteriori probabilities using stochastic network models," in Mozer, M., Smolensky, P., Touretzky, D., Elman, J., and Weigend, A. (eds.), *Proceedings of the 1993 Connectionist Models Summer School,* Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 324-331.

Gifi, A. (1990), *Nonlinear Multivariate Analysis,* NY: John Wiley & Sons, ISBN 0-471-92620-5.

Hampshire II, J.B., and Pearlmutter, B. (1991), "Equivalence proofs for multi-layer perceptron classifiers and the Bayesian discriminant function," in Touretzky, D.S., Elman, J.L., Sejnowski, T.J., and Hinton, G.E. (eds.), *Connectionist Models: Proceedings of the 1990 Summer School,* San Mateo, CA: Morgan Kaufmann, pp.159-172.

Masters, T. (1993). *Practical Neural Network Recipes in C++*, San Diego: Academic Press.

McCullagh, P. and Nelder, J.A. (1989) *Generalized Linear Models,* 2nd ed., London: Chapman & Hall.

Smith, M. (1996). *Neural Networks for Statistical Modeling,* Boston: International Thomson Computer Press, ISBN 1-850-32842-0.

Subject: Why not code binary inputs as 0 and 1?

The importance of standardizing input variables is discussed in detail under "Should I standardize the input variables?" But for the benefit of those people who don't believe in theory, here is an example using the 5-bit parity problem. The unstandardized data are:

x1	x2	x3	x4	x5	target
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	0	1
1	1	0	0	0	0
0	0	1	0	0	1
1	0	1	0	0	0
0	1	1	0	0	0
1	1	1	0	0	1
0	0	0	1	0	1
1	0	0	1	0	0
0	1	0	1	0	0
1	1	0	1	0	1
0	0	1	1	0	0
1	0	1	1	0	1
0	1	1	1	0	1
1	1	1	1	0	0
0	0	0	0	1	1
1	0	0	0	1	0
0	1	0	0	1	0
1	1	0	0	1	1
0	0	1	0	1	0
1	0	1	0	1	1
0	1	1	0	1	1
1	1	1	0	1	0
0	0	0	1	1	0
1	0	0	1	1	1
0	1	0	1	1	1
1	1	0	1	1	0
0	0	1	1	1	1
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	1	1	1

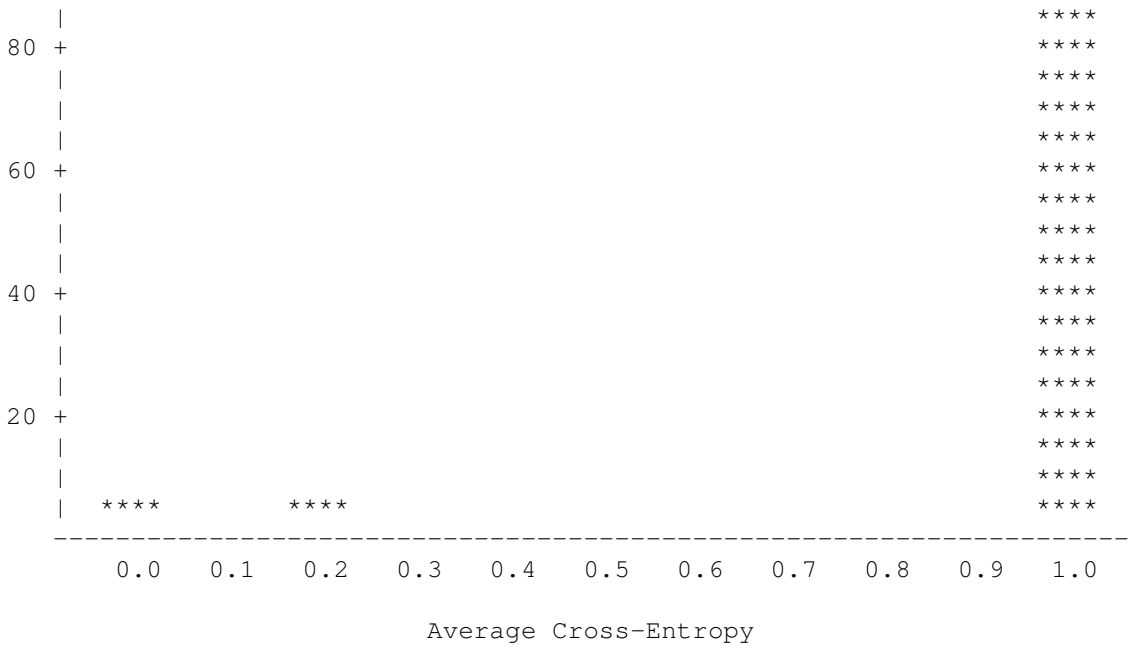
The network characteristics were:

Inputs:	5
Hidden units:	5
Outputs:	5
Activation for hidden units:	tanh
Activation for output units:	logistic
Error function:	cross-entropy
Initial weights:	random normal with mean=0, st.dev.=1/sqrt(5) for input-to-hidden =1 for hidden-to-output
Training method:	batch standard backprop
Learning rate:	0.1
Momentum:	0.9
Minimum training iterations:	100
Maximum training iterations:	10000

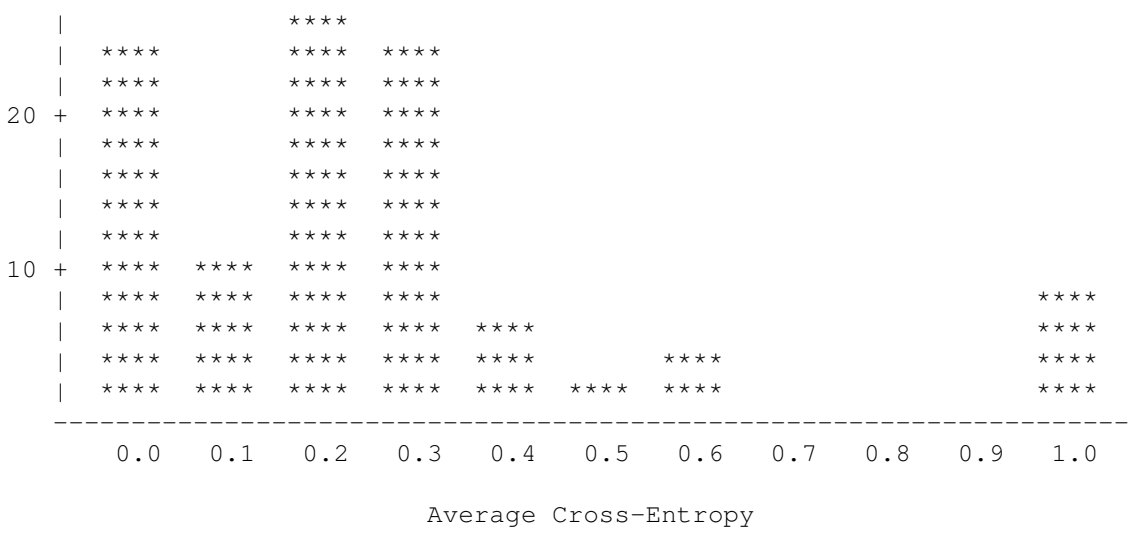
One hundred networks were trained from different random initial weights. The following bar chart shows the distribution of the average cross-entropy after training:

```
Frequency

  |                                                          ****
```



As you can see, very few networks found a good (near zero) local optimum. Recoding the inputs from {0,1} to {-1,1} produced the following distribution of the average cross-entropy after training:



The results are dramatically better. The difference is due to simple geometry. The initial hyperplanes pass fairly near the origin. If the data are centered near the origin (as with {-1,1} coding), the initial hyperplanes will cut through the data in a variety of directions. If the data are offset from the origin (as with {0,1} coding), many of the initial hyperplanes will miss the data entirely, and those that pass through the data will provide a only a limited range of directions, making it difficult to find local optima that use hyperplanes that go in different directions. If the data are far from the origin (as with {9,10} coding), most of the initial hyperplanes will miss the data entirely, which will cause most of the hidden units to saturate and make any learning difficult. See "Should I standardize the input variables?" for more information.

Subject: Why use a bias/threshold?

Sigmoid hidden and output units usually use a "bias" or "threshold" term in computing the net input to the unit. For a linear output unit, a bias term is equivalent to an intercept in a regression model.

A bias term can be treated as a connection weight from a special unit with a constant, nonzero activation value. The term "bias" is usually used with respect to a "bias unit" with a constant value of one. The term "threshold" is usually used with respect to a unit with a constant value of negative one. Not all authors follow this distinction. Regardless of the terminology, whether biases or thresholds are added or subtracted has no effect on the performance of the network.

The single bias unit is connected to every hidden or output unit that needs a bias term. Hence the bias terms can be learned just like other weights.

Consider a multilayer perceptron with any of the usual sigmoid activation functions. Choose any hidden unit or output unit. Let's say there are N inputs to that unit, which define an N-dimensional space. The given unit draws a hyperplane through that space, producing an "on" output on one side and an "off" output on the other. (With sigmoid units the plane will not be sharp -- there will be some gray area of intermediate values near the separating plane -- but ignore this for now.)

The weights determine where this hyperplane lies in the input space. Without a bias term, this separating hyperplane is constrained to pass through the origin of the space defined by the inputs. For some problems that's OK, but in many problems the hyperplane would be much more useful somewhere else. If you have many units in a layer, they share the same input space and without bias they would ALL be constrained to pass through the origin.

The "universal approximation" property of multilayer perceptrons with most commonly-used hidden-layer activation functions does not hold if you omit the bias terms. But Hornik (1993) shows that a sufficient condition for the universal approximation property without biases is that no derivative of the activation function vanishes at the origin, which implies that with the usual sigmoid activation functions, a fixed nonzero bias term can be used instead of a trainable bias.

Typically, every hidden and output unit has its own bias term. The main exception to this is when the activations of two or more units in one layer always sum to a nonzero constant. For example, you might scale the inputs to sum to one (see Should I standardize the input cases?), or you might use a normalized RBF function in the hidden layer (see How do MLPs compare with RBFs?). If there do exist units in one layer whose activations sum to a nonzero constant, then any subsequent layer does not need bias terms if it receives connections from the units that sum to a constant, since using bias terms in the subsequent layer would create linear dependencies.

If you have a large number of hidden units, it may happen that one or more hidden units "saturate" as a result of having large incoming weights, producing a constant activation. If this happens, then the saturated hidden units act like bias units, and the output bias terms are redundant. However, you should not rely on this phenomenon to avoid using output biases, since networks without output biases are usually ill-conditioned (see ftp://ftp.sas.com/pub/neural/illcond/illcond.html) and harder to train than networks that use output biases.

Regarding bias-like terms in RBF networks, see "How do MLPs compare with RBFs?"

Reference:

Hornik, K. (1993), "Some new results on neural network approximation," Neural Networks, 6, 1069-1072.

Subject: Why use activation functions?

Activation functions for the hidden units are needed to introduce nonlinearity into the network. Without nonlinearity, hidden units would not make nets more powerful than just plain perceptrons (which do not have any hidden units, just input and output units). The reason is that a linear function of linear functions is again a linear function. However, it is the nonlinearity (i.e, the capability to represent nonlinear functions) that makes multilayer networks so powerful. Almost any nonlinear function does the job, except for polynomials. For backpropagation learning, the activation function must be differentiable, and it helps if the function is bounded; the sigmoidal functions such as logistic and tanh and the Gaussian function are the most common choices. Functions such as tanh or arctan that produce both positive and negative values tend to yield faster training than functions that produce only positive values such as logistic, because of better numerical conditioning (see ftp://ftp.sas.com/pub/neural/illcond/illcond.html).

For hidden units, sigmoid activation functions are usually preferable to threshold activation functions. Networks with threshold units are difficult to train because the error function is stepwise constant, hence the gradient either does not exist or is zero, making it impossible to use backprop or more efficient gradient-based training methods. Even for training methods that do not use gradients--such as simulated annealing and genetic algorithms--sigmoid units are easier to train than threshold units. With sigmoid units, a small change in the weights will usually produce a change in the outputs, which makes it possible to tell whether that change in the weights is good or bad. With threshold units, a small change in the weights will often produce no change in the outputs.

For the output units, you should choose an activation function suited to the distribution of the target values:

- For binary (0/1) targets, the logistic function is an excellent choice (Jordan, 1995).
- For categorical targets using 1-of-C coding, the softmax activation function is the logical extension of the logistic function.
- For continuous-valued targets with a bounded range, the logistic and tanh functions can be used, provided you either scale the outputs to the range of the targets or scale the targets to the range of the output activation function ("scaling" means multiplying by and adding appropriate constants).
- If the target values are positive but have no known upper bound, you can use an exponential output activation function, but beware of overflow.
- For continuous-valued targets with no known bounds, use the identity or "linear" activation function (which amounts to no activation function) unless you have a very good reason to do otherwise.

There are certain natural associations between output activation functions and various noise distributions which have been studied by statisticians in the context of generalized linear models. The output activation function is the inverse of what statisticians call the "link function". See:

McCullagh, P. and Nelder, J.A. (1989) *Generalized Linear Models,* 2nd ed., London: Chapman & Hall.

Jordan, M.I. (1995), "Why the logistic function? A tutorial discussion on probabilities and neural networks", MIT Computational Cognitive Science Report 9503, http://www.cs.berkeley.edu/~jordan/papers/uai.ps.Z.

For more information on activation functions, see Donald Tveter's Backpropagator's Review.

Subject: How to avoid overflow in the logistic function?

The formula for the logistic activation function is often written as:

```
netoutput = 1 / (1+exp(-netinput));
```

But this formula can produce floating-point overflow in the exponential function if you program it in this simple form. To avoid overflow, you can do this:

```
if (netinput < -45) netoutput = 0;
else if (netinput > 45) netoutput = 1;
else netoutput = 1 / (1+exp(-netinput));
```

The constant 45 will work for double precision on all machines that I know of, but there may be some bizarre machines where it will require some adjustment. Other activation functions can be handled similarly.

Subject: What is a softmax activation function?

If you want the outputs of a network to be interpretable as posterior probabilities for a categorical target variable, it is highly desirable for those outputs to lie between zero and one and to sum to one. The purpose of the softmax activation function is to enforce these constraints on the outputs. Let the net input to each output unit be `q_i, i=1,...,c,` where `c` is the number of categories. Then the softmax output `p_i` is:

```
         exp(q_i)
p_i = ------------
       c
      sum exp(q_j)
      j=1
```

Unless you are using weight decay or Bayesian estimation or some such thing that requires the weights to be treated on an equal basis, you can choose any one of the output units and leave it completely unconnected--just set the net input to 0. Connecting all of the output units will just give you redundant weights and will slow down training. To see this, add an arbitrary constant z to each net input and you get:

```
         exp(q_i+z)        exp(q_i) exp(z)        exp(q_i)
p_i = ------------   = -------------------- = ------------
       c                  c                    c
      sum exp(q_j+z)    sum exp(q_j) exp(z)    sum exp(q_j)
      j=1               j=1                    j=1
```

so nothing changes. Hence you can always pick one of the output units, and add an appropriate constant to each net input to produce any desired net input for the selected output unit, which you can choose to be zero or whatever is convenient. You can use the same trick to make sure that none of the exponentials overflows.

Statisticians usually call softmax a "multiple logistic" function. It reduces to the simple logistic function when there are only two categories. Suppose you choose to set `q_2` to 0. Then

```
         exp(q_1)          exp(q_1)               1
p_1 = ------------ = ----------------- = -------------
       c             exp(q_1) + exp(0)   1 + exp(-q_1)
      sum exp(q_j)
      j=1
```

and `p_2,` of course, is `1-p_1`.

The softmax function derives naturally from log-linear models and leads to convenient interpretations of

the weights in terms of odds ratios. You could, however, use a variety of other nonnegative functions on the real line in place of the exp function. Or you could constrain the net inputs to the output units to be nonnegative, and just divide by the sum--that's called the Bradley-Terry-Luce model.

The softmax function is also used in the hidden layer of normalized radial-basis-function networks; see "How do MLPs compare with RBFs?"

References:

Bridle, J.S. (1990a). Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In: F.Fogleman Soulie and J.Herault (eds.), Neurocomputing: Algorithms, Architectures and Applications, Berlin: Springer-Verlag, pp. 227-236.

Bridle, J.S. (1990b). Training Stochastic Model Recognition Algorithms as Networks can lead to Maximum Mutual Information Estimation of Parameters. In: D.S.Touretzky (ed.), Advances in Neural Information Processing Systems 2, San Mateo: Morgan Kaufmann, pp. 211-217.

McCullagh, P. and Nelder, J.A. (1989) *Generalized Linear Models,* 2nd ed., London: Chapman & Hall. See Chapter 5.

Subject: What is the curse of dimensionality?

Answer by Janne Sinkkonen.

Curse of dimensionality (Bellman 1961) refers to the exponential growth of hypervolume as a function of dimensionality. In the field of NNs, curse of dimensionality expresses itself in two related problems:

1. Many NNs can be thought of mappings from an input space to an output space. Thus, loosely speaking, an NN needs to somehow "monitor", cover or represent every part of its input space in order to know how that part of the space should be mapped. Covering the input space takes resources, and, in the most general case, the amount of resources needed is proportional to the hypervolume of the input space. The exact formulation of "resources" and "part of the input space" depends on the type of the network and should probably be based on the concepts of information theory and differential geometry.

   As an example, think of a vector quantization (VQ). In VQ, a set of units competitively learns to represents an input space (this is like Kohonen's Self-Organizing Map but without topography for the units). Imagine a VQ trying to share its units (resources) more or less equally over hyperspherical input space. One could argue that the average distance from a random point of the space to the nearest network unit measures the goodness of the representation: the shorter the distance, the better is the represention of the data in the sphere. It is intuitively clear (and can be experimentally verified) that the total number of units required to keep the average distance constant increases exponentially with the dimensionality of the sphere (if the radius of the sphere is fixed).

   The curse of dimensionality causes networks with lots of irrelevant inputs to be behave relatively badly: the dimension of the input space is high, and the network uses almost all its resources to represent irrelevant portions of the space.

   Unsupervised learning algorithms are typically prone to this problem - as well as conventional

RBFs. A partial remedy is to preprocess the input in the right way, for example by scaling the components according to their "importance".

2. Even if we have a network algorithm which is able to focus on important portions of the input space, the higher the dimensionality of the input space, the more data may be needed to find out what is important and what is not.

A priori information can help with the curse of dimensionality. Careful feature selection and scaling of the inputs fundamentally affects the severity of the problem, as well as the selection of the neural network model. For classification purposes, only the borders of the classes are important to represent accurately.

References:

Bellman, R. (1961), *Adaptive Control Processes: A Guided Tour*, Princeton University Press.

Bishop, C.M. (1995), *Neural Networks for Pattern Recognition,* Oxford: Oxford University Press, section 1.4.

Scott, D.W. (1992), *Multivariate Density Estimation*, NY: Wiley.

Subject: How do MLPs compare with RBFs?

Multilayer perceptrons (MLPs) and radial basis function (RBF) networks are the two most commonly-used types of feedforward network. They have much more in common than most of the NN literature would suggest. The only fundamental difference is the way in which hidden units combine values coming from preceding layers in the network--MLPs use inner products, while RBFs use Euclidean distance. There are also differences in the customary methods for training MLPs and RBF networks, although most methods for training MLPs can also be applied to RBF networks. Furthermore, there are crucial differences between two broad types of RBF network--ordinary RBF networks and normalized RBF networks--that are ignored in most of the NN literature. These differences have important consequences for the generalization ability of the networks, especially when the number of inputs is large.

Notation:

```
a_j     is the altitude or height of the jth hidden unit
b_j     is the bias of the jth hidden unit
f       is the fan-in of the jth hidden unit
h_j     is the activation of the jth hidden unit
s       is a common width shared by all hidden units in the layer
s_j     is the width of the jth hidden unit
w_ij    is the weight connecting the ith input to
          the jth hidden unit
w_i     is the common weight for the ith input shared by
          all hidden units in the layer
x_i     is the ith input
```

The inputs to each hidden or output unit must be combined with the weights to yield a single value called the "net input" to which the activation function is applied. There does not seem to be a standard term for the function that combines the inputs and weights; I will use the term "combination function". Thus, each hidden or output unit in a feedforward network first computes a combination function to produce the net input, and then applies an activation function to the net input yielding the activation of the unit.

A multilayer perceptron has one or more hidden layers for which the combination function is the inner

product of the inputs and weights, plus a bias. The activation function is usually a `logistic` or `tanh` function. Hence the formula for the activation is typically:

```
h_j = tanh( b_j + sum[w_ij*x_i] )
```

The MLP architecture is the most popular one in practical applications. Each layer uses a linear combination function. The inputs are fully connected to the first hidden layer, each hidden layer is fully connected to the next, and the last hidden layer is fully connected to the outputs. You can also have "skip-layer" connections; direct connections from inputs to outputs are especially useful.

Consider the multidimensional space of inputs to a given hidden unit. Since an MLP uses linear combination functions, the set of all points in the space having a given value of the activation function is a hyperplane. The hyperplanes corresponding to different activation levels are parallel to each other (the hyperplanes for different units are not parallel in general). These parallel hyperplanes are the *isoactivation contours* of the hidden unit.

Radial basis function (RBF) networks usually have only one hidden layer for which the combination function is based on the Euclidean distance between the input vector and the weight vector. RBF networks do not have anything that's exactly the same as the bias term in an MLP. But some types of RBFs have a "width" associated with each hidden unit or with the the entire hidden layer; instead of adding it in the combination function like a bias, you divide the Euclidean distance by the width.

To see the similarity between RBF networks and MLPs, it is convenient to treat the combination function as the square of distance/width. Then the familiar `exp` or `softmax` activation functions produce members of the popular class of Gaussian RBF networks. It can also be useful to add another term to the combination function that determines what I will call the "altitude" of the unit. The altitude is the maximum height of the Gaussian curve above the horizontal axis. I have not seen altitudes used in the NN literature; if you know of a reference, please tell me (saswss@unx.sas.com).

The output activation function in RBF networks is usually the identity. The identity output activation function is a computational convenience in training (see Hybrid training and the curse of dimensionality) but it is possible and often desirable to use other output activation functions just as you would in an MLP.

There are many types of radial basis functions. Gaussian RBFs seem to be the most popular by far in the NN literature. In the statistical literature, thin plate splines are also used (Green and Silverman 1994). This FAQ will concentrate on Gaussian RBFs.

There are two distinct types of Gaussian RBF architectures. The first type uses the `exp` activation function, so the activation of the unit is a Gaussian "bump" as a function of the inputs. There seems to be no specific term for this type of Gaussian RBF network; I will use the term "ordinary RBF", or ORBF, network.

The second type of Gaussian RBF architecture uses the softmax activation function, so the activations of all the hidden units are normalized to sum to one. This type of network is often called a "normalized RBF", or NRBF, network. In a NRBF network, the output units should not have a bias, since the constant bias term would be linearly dependent on the constant sum of the hidden units.

While the distinction between these two types of Gaussian RBF architectures is sometimes mentioned in the NN literature, its importance has rarely been appreciated except by Tao (1993) and Werntges (1993). Shorten and Murray-Smith (1996) also compare ordinary and normalized Gaussian RBF networks.

There are several subtypes of both ORBF and NRBF architectures. Descriptions and formulas are as follows:

ORBFUN
: Ordinary radial basis function (RBF) network with unequal widths
`h_j = exp( - s_j^-2 * sum[(w_ij-x_i)^2] )`

ORBFEQ
: Ordinary radial basis function (RBF) network with equal widths
`h_j = exp( - s^-2 * sum[(w_ij-x_i)^2] )`

NRBFUN
: Normalized RBF network with unequal widths and heights
`h_j = softmax(f*log(a_j) - s_j^-2 * sum[(w_ij-x_i)^2] )`

NRBFEV
: Normalized RBF network with equal volumes
`h_j = softmax( f*log(s_j) - s_j^-2 * sum[(w_ij-x_i)^2] )`

NRBFEH
: Normalized RBF network with equal heights (and unequal widths)
`h_j = softmax( - s_j^-2 * sum[(w_ij-x_i)^2] )`

NRBFEW
: Normalized RBF network with equal widths (and unequal heights)
`h_j = softmax( f*log(a_j) - s^-2 * sum[(w_ij-x_i)^2] )`

NRBFEQ
: Normalized RBF network with equal widths and heights
`h_j = softmax( - s^-2 * sum[(w_ij-x_i)^2] )`

To illustrate various architectures, an example with two inputs and one output will be used so that the results can be shown graphically. The function being learned resembles a landscape with a Gaussian hill and a logistic plateau as shown in ftp://ftp.sas.com/pub/neural/hillplat.gif. There are 441 training cases on a regular 21-by-21 grid. The table below shows the root mean square error (RMSE) for a test data set. The test set has 1681 cases on a regular 41-by-41 grid over the same domain as the training set. If you are reading the HTML version of this document via a web browser, click on any number in the table to see a surface plot of the corresponding network output (each plot is a gif file, approximately 9K).

The MLP networks in the table have one hidden layer with a tanh activation function. All of the networks use an identity activation function for the outputs.

Hill and Plateau Data: RMSE for the Test Set

HUs	MLP	ORBFEQ	ORBFUN	NRBFEQ	NRBFEW	NRBFEV	NRBFEH	NRBFUN
2	0.218	0.247	0.247	0.230	0.230	0.230	0.230	0.230
3	0.192	0.244	0.143	0.218	0.218	0.036	0.012	0.001
4	0.174	0.216	0.096	0.193	0.193	0.036	0.007	
5	0.160	0.188	0.083	0.086	0.051	0.003		
6	0.123	0.142	0.058	0.053	0.030			
7	0.107	0.123	0.051	0.025	0.019			
8	0.093	0.105	0.043	0.020	0.008			
9	0.084	0.085	0.038	0.017				
10	0.077	0.082	0.033	0.016				
12	0.059	0.074	0.024	0.005				
15	0.042	0.060	0.019					
20	0.023	0.046	0.010					
30	0.019	0.024						
40	0.016	0.022						
50	0.010	0.014						

The ORBF architectures use radial combination functions and the `exp` activation function. Only two of the radial combination functions are useful with ORBF architectures. For radial combination functions including an altitude, the altitude would be redundant with the hidden-to-output weights.

Radial combination functions are based on the Euclidean distance between the vector of inputs to the unit

and the vector of corresponding weights. Thus, the isoactivation contours for ORBF networks are concentric hyperspheres. A variety of activation functions can be used with the radial combination function, but the `exp` activation function, yielding a Gaussian surface, is the most useful. Radial networks typically have only one hidden layer, but it can be useful to include a linear layer for dimensionality reduction or oblique rotation before the RBF layer.

The output of an ORBF network consists of a number of superimposed bumps, hence the output is quite bumpy unless many hidden units are used. Thus an ORBF network with only a few hidden units is incapable of fitting a wide variety of simple, smooth functions, and should rarely be used.

The NRBF architectures also use radial combination functions but the activation function is softmax, which forces the sum of the activations for the hidden layer to equal one. Thus, each output unit computes a weighted average of the hidden-to-output weights, and the output values must lie within the range of the hidden-to-output weights. Therefore, if the hidden-to-output weights are within a reasonable range (such as the range of the target values), you can be sure that the outputs will be within that same range for all possible inputs, even when the net is extrapolating. No comparably useful bound exists for the output of an ORBF network.

If you extrapolate far enough in a Gaussian ORBF network with an identity output activation function, the activation of every hidden unit will approach zero, hence the extrapolated output of the network will equal the output bias. If you extrapolate far enough in an NRBF network, one hidden unit will come to dominate the output. Hence if you want the network to extrapolate different values in a different directions, an NRBF should be used instead of an ORBF.

Radial combination functions incorporating altitudes are useful with NRBF architectures. The NRBF architectures combine some of the virtues of both the RBF and MLP architectures, as explained below. However, the isoactivation contours are considerably more complicated than for ORBF architectures.

Consider the case of an NRBF network with only two hidden units. If the hidden units have equal widths, the isoactivation contours are parallel hyperplanes; in fact, this network is equivalent to an MLP with one logistic hidden unit. If the hidden units have unequal widths, the isoactivation contours are concentric hyperspheres; such a network is almost equivalent to an ORBF network with one Gaussian hidden unit.

If there are more than two hidden units in an NRBF network, the isoactivation contours have no such simple characterization. If the RBF widths are very small, the isoactivation contours are approximately piecewise linear for RBF units with equal widths, and approximately piecewise spherical for RBF units with unequal widths. The larger the widths, the smoother the isoactivation contours where the pieces join. As Shorten and Murray-Smith (1996) point out, the activation is not necessarily a monotone function of distance from the center when unequal widths are used.

The NRBFEQ architecture is a smoothed variant of the learning vector quantization (Kohonen 1988, Ripley 1996) and counterpropagation (Hecht-Nielsen 1990), architectures. In LVQ and counterprop, the hidden units are often called "codebook vectors". LVQ amounts to nearest-neighbor classification on the codebook vectors, while counterprop is nearest-neighbor regression on the codebook vectors. The NRBFEQ architecture uses not just the single nearest neighbor, but a weighted average of near neighbors. As the width of the NRBFEQ functions approaches zero, the weights approach one for the nearest neighbor and zero for all other codebook vectors. LVQ and counterprop use ad hoc algorithms of uncertain reliability, but standard numerical optimization algorithms (not to mention backprop) can be applied with the NRBFEQ architecture.

In a NRBFEQ architecture, if each observation is taken as an RBF center, and if the weights are taken to be the target values, the outputs are simply weighted averages of the target values, and the network is

identical to the well-known Nadaraya-Watson kernel regression estimator, which has been reinvented at least twice in the neural net literature (see "What is GRNN?"). A similar NRBFEQ network used for classification is equivalent to kernel discriminant analysis (see "What is PNN?").

Kernels with variable widths are also used for regression in the statistical literature. Such kernel estimators correspond to the the NRBFEV architecture, in which the kernel functions have equal volumes but different altitudes. In the neural net literature, variable-width kernels appear always to be of the NRBFEH variety, with equal altitudes but unequal volumes. The analogy with kernel regression would make the NRBFEV architecture the obvious choice, but which of the two architectures works better in practice is an open question.

Hybrid training and the curse of dimensionality

A comparison of the various architectures must separate training issues from architectural issues to avoid common sources of confusion. RBF networks are often trained by "hybrid" methods, in which the hidden weights (*centers*) are first obtained by unsupervised learning, after which the output weights are obtained by supervised learning. Unsupervised methods for choosing the centers include:

1. Distribute the centers in a regular grid over the input space.
2. Choose a random subset of the training cases to serve as centers.
3. Cluster the training cases based on the input variables, and use the mean of each cluster as a center.

Various heuristic methods are also available for choosing the RBF widths (e.g., Moody and Darken 1989; Sarle 1994b). Once the centers and widths are fixed, the output weights can be learned very efficiently, since the computation reduces to a linear or generalized linear model. The hybrid training approach can thus be much faster than the nonlinear optimization that would be required for supervised training of all of the weights in the network.

Hybrid training is not often applied to MLPs because no effective methods are known for unsupervised training of the hidden units (except when there is only one input).

Hybrid training will usually require more hidden units than supervised training. Since supervised training optimizes the locations of the centers, while hybrid training does not, supervised training will provide a better approximation to the function to be learned for a given number of hidden units. Thus, the better fit provided by supervised training will often let you use fewer hidden units for a given accuracy of approximation than you would need with hybrid training. And if the hidden-to-output weights are learned by linear least-squares, the fact that hybrid training requires more hidden units implies that hybrid training will also require more training cases for the same accuracy of generalization (Tarassenko and Roberts 1994).

The number of hidden units required by hybrid methods becomes an increasingly serious problem as the number of inputs increases. In fact, the required number of hidden units tends to increase exponentially with the number of inputs. This drawback of hybrid methods is discussed by Minsky and Papert (1969). For example, with method (1) for RBF networks, you would need at least five elements in the grid along each dimension to detect a moderate degree of nonlinearity; so if you have `Nx` inputs, you would need at least `5^Nx` hidden units. For methods (2) and (3), the number of hidden units increases exponentially with the effective dimensionality of the input distribution. If the inputs are linearly related, the effective dimensionality is the number of nonnegligible (a deliberately vague term) eigenvalues of the covariance matrix, so the inputs must be highly correlated if the effective dimensionality is to be much less than the number of inputs.

The exponential increase in the number of hidden units required for hybrid learning is one aspect of the

curse of dimensionality. The number of training cases required also increases exponentially in general. No neural network architecture--in fact no method of learning or statistical estimation--can escape the curse of dimensionality in general, hence there is no practical method of learning general functions in more than a few dimensions.

Fortunately, in many practical applications of neural networks with a large number of inputs, most of those inputs are additive, redundant, or irrelevant, and some architectures can take advantage of these properties to yield useful results. But escape from the curse of dimensionality requires fully supervised training as well as special types of data. Supervised training for RBF networks can be done by "backprop" (see "What is backprop?") or other optimization methods (see "What are conjugate gradients, Levenberg-Marquardt, etc.?"), or by subset regression "What are OLS and subset/stepwise regression?").

Additive inputs

An additive model is one in which the output is a sum of linear or nonlinear transformations of the inputs. If an additive model is appropriate, the number of weights increases linearly with the number of inputs, so high dimensionality is not a curse. Various methods of training additive models are available in the statistical literature (e.g. Hastie and Tibshirani 1990). You can also create a feedforward neural network, called a "generalized additive network" (GAN), to fit additive models (Sarle 1994a). Additive models have been proposed in the neural net literature under the name "topologically distributed encoding" (Geiger 1990).

Projection pursuit regression (PPR) provides both universal approximation and the ability to avoid the curse of dimensionality for certain common types of target functions (Friedman and Stuetzle 1981). Like MLPs, PPR computes the output as a sum of nonlinear transformations of linear combinations of the inputs. Each term in the sum is analogous to a hidden unit in an MLP. But unlike MLPs, PPR allows general, smooth nonlinear transformations rather than a specific nonlinear activation function, and allows a different transformation for each term. The nonlinear transformations in PPR are usually estimated by nonparametric regression, but you can set up a *projection pursuit network* (PPN), in which each nonlinear transformation is performed by a subnetwork. If a PPN provides an adequate fit with few terms, then the curse of dimensionality can be avoided, and the results may even be interpretable.

If the target function can be accurately approximated by projection pursuit, then it can also be accurately approximated by an MLP with a single hidden layer. The disadvantage of the MLP is that there is little hope of interpretability. An MLP with two or more hidden layers can provide a parsimonious fit to a wider variety of target functions than can projection pursuit, but no simple characterization of these functions is known.

Redundant inputs

With proper training, all of the RBF architectures listed above, as well as MLPs, can process redundant inputs effectively. When there are redundant inputs, the training cases lie close to some (possibly nonlinear) subspace. If the same degree of redundancy applies to the test cases, the network need produce accurate outputs only near the subspace occupied by the data. Adding redundant inputs has little effect on the effective dimensionality of the data; hence the curse of dimensionality does not apply, and even hybrid methods (2) and (3) can be used. However, if the test cases do not follow the same pattern of redundancy as the training cases, generalization will require extrapolation and will rarely work well.

Irrelevant inputs

MLP architectures are good at ignoring irrelevant inputs. MLPs can also select linear subspaces of

reduced dimensionality. Since the first hidden layer forms linear combinations of the inputs, it confines the networks attention to the linear subspace spanned by the weight vectors. Hence, adding irrelevant inputs to the training data does not increase the number of hidden units required, although it increases the amount of training data required.

ORBF architectures are not good at ignoring irrelevant inputs. The number of hidden units required grows exponentially with the number of inputs, regardless of how many inputs are relevant. This exponential growth is related to the fact that ORBFs have *local receptive fields*, meaning that changing the hidden-to-output weights of a given unit will affect the output of the network only in a neighborhood of the center of the hidden unit, where the size of the neighborhood is determined by the width of the hidden unit. (Of course, if the width of the unit is learned, the receptive field could grow to cover the entire training set.)

Local receptive fields are often an advantage compared to the *distributed* architecture of MLPs, since local units can adapt to local patterns in the data without having unwanted side effects in other regions. In a distributed architecture such as an MLP, adapting the network to fit a local pattern in the data can cause spurious side effects in other parts of the input space.

However, ORBF architectures often must be used with relatively small neighborhoods, so that several hidden units are required to cover the range of an input. When there are many nonredundant inputs, the hidden units must cover the entire input space, and the number of units required is essentially the same as in the hybrid case (1) where the centers are in a regular grid; hence the exponential growth in the number of hidden units with the number of inputs, regardless of whether the inputs are relevant.

You can enable an ORBF architecture to ignore irrelevant inputs by using an extra, linear hidden layer before the radial hidden layer. This type of network is sometimes called an "elliptical basis function" network. If the number of units in the linear hidden layer equals the number of inputs, the linear hidden layer performs an oblique rotation of the input space that can suppress irrelevant directions and differentally weight relevant directions according to their importance. If you think that the presence of irrelevant inputs is highly likely, you can force a reduction of dimensionality by using fewer units in the linear hidden layer than the number of inputs.

Note that the linear and radial hidden layers must be connected in series, not in parallel, to ignore irrelevant inputs. In some applications it is useful to have linear and radial hidden layers connected in parallel, but in such cases the radial hidden layer will be sensitive to all inputs.

For even greater flexibility (at the cost of more weights to be learned), you can have a separate linear hidden layer for each RBF unit, allowing a different oblique rotation for each RBF unit.

NRBF architectures with equal widths (NRBFEW and NRBFEQ) combine the advantage of local receptive fields with the ability to ignore irrelevant inputs. The receptive field of one hidden unit extends from the center in all directions until it encounters the receptive field of another hidden unit. It is convenient to think of a "boundary" between the two receptive fields, defined as the hyperplane where the two units have equal activations, even though the effect of each unit will extend somewhat beyond the boundary. The location of the boundary depends on the heights of the hidden units. If the two units have equal heights, the boundary lies midway between the two centers. If the units have unequal heights, the boundary is farther from the higher unit.

If a hidden unit is surrounded by other hidden units, its receptive field is indeed local, curtailed by the field boundaries with other units. But if a hidden unit is not completely surrounded, its receptive field can extend infinitely in certain directions. If there are irrelevant inputs, or more generally, irrelevant directions that are linear combinations of the inputs, the centers need only be distributed in a subspace orthogonal to the irrelevant directions. In this case, the hidden units can have local receptive fields in relevant directions

but infinite receptive fields in irrelevant directions.

For NRBF architectures allowing unequal widths (NRBFUN, NRBFEV, and NRBFEH), the boundaries between receptive fields are generally hyperspheres rather than hyperplanes. In order to ignore irrelevant inputs, such networks must be trained to have equal widths. Hence, if you think there is a strong possibility that some of the inputs are irrelevant, it is usually better to use an architecture with equal widths.

References:

There are few good references on RBF networks. Bishop (1995) gives one of the better surveys, but also see Tao (1993) and Werntges (1993) for the importance of normalization. Orr (1996) provides a useful introduction.

Bishop, C.M. (1995), *Neural Networks for Pattern Recognition,* Oxford: Oxford University Press.

Friedman, J.H. and Stuetzle, W. (1981), "Projection pursuit regression," J. of the American Statistical Association, 76, 817-823.

Geiger, H. (1990), "Storing and Processing Information in Connectionist Systems," in Eckmiller, R., ed., *Advanced Neural Computers,* 271-277, Amsterdam: North-Holland.

Green, P.J. and Silverman, B.W. (1994), *Nonparametric Regression and Generalized Linear Models: A roughness penalty approach,*, London: Chapman & Hall.

Hastie, T.J. and Tibshirani, R.J. (1990) *Generalized Additive Models,* London: Chapman & Hall.

Hecht-Nielsen, R. (1990), *Neurocomputing*, Reading, MA: Addison-Wesley.

Kohonen, T (1988), "Learning Vector Quantization," Neural Networks, 1 (suppl 1), 303.

Minsky, M.L. and Papert, S.A. (1969), *Perceptrons*, Cambridge, MA: MIT Press.

Moody, J. and Darken, C.J. (1989), "Fast learning in networks of locally-tuned processing units," Neural Computation, 1, 281-294.

Orr, M.J.L. (1996), "Introduction to radial basis function networks," http://www.anc.ed.ac.uk/~mjo/papers/intro.ps or http://www.anc.ed.ac.uk/~mjo/papers/intro.ps.gz

Ripley, B.D. (1996), *Pattern Recognition and Neural Networks*, Cambridge: Cambridge University Press.

Sarle, W.S. (1994a), "Neural Networks and Statistical Models," in SAS Institute Inc., *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc., pp 1538-1550, ftp://ftp.sas.com/pub/neural/neural1.ps.

Sarle, W.S. (1994b), "Neural Network Implementation in SAS Software," in SAS Institute Inc., *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc., pp 1551-1573, ftp://ftp.sas.com/pub/neural/neural2.ps.

Shorten, R., and Murray-Smith, R. (1996), "Side effects of normalising radial basis function networks" International Journal of Neural Systems, 7, 167-179.

Tao, K.M. (1993), "A closer look at the radial basis function (RBF) networks," *Conference Record*

of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers (Singh, A., ed.), vol 1, 401-405, Los Alamitos, CA: IEEE Comput. Soc. Press.

Tarassenko, L. and Roberts, S. (1994), "Supervised and unsupervised learning in radial basis function classifiers," IEE Proceedings-- Vis. Image Signal Processing, 141, 210-216.

Werntges, H.W. (1993), "Partitions of unity improve neural function approximation," Proceedings of the IEEE International Conference on Neural Networks, San Francisco, CA, vol 2, 914-918.

Subject: What are OLS and subset/stepwise regression?

If you are statistician, "OLS" means "ordinary least squares" (as opposed to weighted or generalized least squares), which is what the NN literature often calls "LMS" (least mean squares).

If you are a neural networker, "OLS" means "orthogonal least squares", which is an algorithm for forward stepwise regression proposed by Chen et al. (1991) for training RBF networks.

OLS is a variety of supervised training. But whereas backprop and other commonly-used supervised methods are forms of continuous optimization, OLS is a form of combinatorial optimization. Rather than treating the RBF centers as continuous values to be adjusted to reduce the training error, OLS starts with a large set of candidate centers and selects a subset that usually provides good training error. For small training sets, the candidates can include all of the training cases. For large training sets, it is more efficient to use a random subset of the training cases or to do a cluster analysis and use the cluster means as candidates.

Each center corresponds to a predictor variable in a linear regression model. The values of these predictor variables are computed from the RBF applied to each center. There are numerous methods for selecting a subset of predictor variables in regression (Myers 1986; Miller 1990). The ones most often used are:

- Forward selection begins with no centers in the network. At each step the center is added that most decreases the objective function.
- Backward elimination begins with all candidate centers in the network. At each step the center is removed that least increases the objective function.
- Stepwise selection begins like forward selection with no centers in the network. At each step, a center is added or removed. If there are any centers in the network, the one that contributes least to reducing the objective function is subjected to a statistical test (usually based on the F statistic) to see if it is worth retaining in the network; if the center fails the test, it is removed. If no centers are removed, then the centers that are not currently in the network are examined; the one that would contribute most to reducing the objective function is subjected to a statistical test to see if it is worth adding to the network; if the center passes the test, it is added. When all centers in the network pass the test for staying in the network, and all other centers fail the test for being added to the network, the stepwise method terminates.
- Leaps and bounds (Furnival and Wilson 1974) is an algorithm for determining the subset of centers that minimizes the objective function; this optimal subset can be found without examining all possible subsets, but the algorithm is practical only up to 30 to 50 candidate centers.

OLS is a particular algorithm for forward selection using modified Gram-Schmidt (MGS) orthogonalization. While MGS is not a bad algorithm, it is not the best algorithm for linear least-squares (Lawson and Hanson 1974). For ill-conditioned data (see ftp://ftp.sas.com/pub/neural/illcond/illcond.html), Householder and Givens methods are generally preferred, while for large, well-conditioned

data sets, methods based on the normal equations require about one-third as many floating point operations and much less disk I/O than OLS. Normal equation methods based on sweeping (Goodnight 1979) or Gaussian elimination (Furnival and Wilson 1974) are especially simple to program.

While the theory of linear models is the most thoroughly developed area of statistical inference, subset selection invalidates most of the standard theory (Miller 1990; Roecker 1991; Derksen and Keselman 1992; Freedman, Pee, and Midthune 1992).

Subset selection methods usually do not generalize as well as regularization methods in linear models (Frank and Friedman 1993). Orr (1995) has proposed combining regularization with subset selection for RBF training (see also Orr 1996).

References:

Chen, S., Cowan, C.F.N., and Grant, P.M. (1991), "Orthogonal least squares learning for radial basis function networks," IEEE Transactions on Neural Networks, 2, 302-309.

Derksen, S. and Keselman, H. J. (1992) "Backward, forward and stepwise automated subset selection algorithms: Frequency of obtaining authentic and noise variables," British Journal of Mathematical and Statistical Psychology, 45, 265-282,

Frank, I.E. and Friedman, J.H. (1993) "A statistical view of some chemometrics regression tools," Technometrics, 35, 109-148.

Freedman, L.S. , Pee, D. and Midthune, D.N. (1992) "The problem of underestimating the residual error variance in forward stepwise regression", The Statistician, 41, 405-412.

Furnival, G.M. and Wilson, R.W. (1974), "Regression by Leaps and Bounds," Technometrics, 16, 499-511.

Goodnight, J.H. (1979), "A Tutorial on the SWEEP Operator," The American Statistician, 33, 149-158.

Lawson, C. L. and Hanson, R. J. (1974), *Solving Least Squares Problems,* Englewood Cliffs, NJ: Prentice-Hall, Inc. (2nd edition: 1995, Philadelphia: SIAM)

Miller, A.J. (1990), Subset Selection in Regression, Chapman & Hall.

Myers, R.H. (1986), *Classical and Modern Regression with Applications,* Boston: Duxbury Press.

Orr, M.J.L. (1995), "Regularisation in the selection of radial basis function centres," Neural Computation, 7, 606-623.

Orr, M.J.L. (1996), "Introduction to radial basis function networks," http://www.cns.ed.ac.uk/people/mark/intro.ps or http://www.cns.ed.ac.uk/people/mark/intro/intro.html .

Roecker, E.B. (1991) "Prediction error and its estimation for subset-selected models," Technometrics, 33, 459-468.

Subject: Should I normalize/standardize/rescale the data?

First, some definitions. "Rescaling" a vector means to add or subtract a constant and then multiply or

divide by a constant, as you would do to change the units of measurement of the data, for example, to convert a temperature from Celsius to Fahrenheit.

"Normalizing" a vector most often means dividing by a norm of the vector, for example, to make the Euclidean length of the vector equal to one. In the NN literature, "normalizing" also often refers to rescaling by the minimum and range of the vector, to make all the elements lie between 0 and 1.

"Standardizing" a vector most often means subtracting a measure of location and dividing by a measure of scale. For example, if the vector contains random values with a Gaussian distribution, you might subtract the mean and divide by the standard deviation, thereby obtaining a "standard normal" random variable with mean 0 and standard deviation 1.

However, all of the above terms are used more or less interchangeably depending on the customs within various fields. Since the FAQ maintainer is a statistician, he is going to use the term "standardize" because that is what he is accustomed to.

Now the question is, should you do any of these things to your data? The answer is, it depends.

There is a common misconception that the inputs to a multilayer perceptron must be in the interval [0,1]. There is in fact no such requirement, although there often are benefits to standardizing the inputs as discussed below. But it is better to have the input values centered around zero, so scaling the inputs to the interval [0,1] is usually a bad choice.

If your output activation function has a range of [0,1], then obviously you must ensure that the target values lie within that range. But it is generally better to choose an output activation function suited to the distribution of the targets than to force your data to conform to the output activation function. See "Why use activation functions?"

When using an output activation with a range of [0,1], some people prefer to rescale the targets to a range of [.1,.9]. I suspect that the popularity of this gimmick is due to the slowness of standard backprop. But using a target range of [.1,.9] for a classification task gives you incorrect posterior probability estimates. This gimmick is unnecessary if you use an efficient training algorithm (see "What are conjugate gradients, Levenberg-Marquardt, etc.?"), and it is also unnecessary to avoid overflow (see "How to avoid overflow in the logistic function?").

Now for some of the gory details: note that the training data form a matrix. Let's set up this matrix so that each case forms a row, and the inputs and target variables form columns. You could conceivably standardize the rows or the columns or both or various other things, and these different ways of choosing vectors to standardize will have quite different effects on training.

Standardizing either input or target variables tends to make the training process better behaved by improving the numerical condition (see ftp://ftp.sas.com/pub/neural/illcond/illcond.html) of the optimization problem and ensuring that various default values involved in initialization and termination are appropriate. Standardizing targets can also affect the objective function.

Standardization of cases should be approached with caution because it discards information. If that information is irrelevant, then standardizing cases can be quite helpful. If that information is important, then standardizing cases can be disastrous.

Should I standardize the input variables (column vectors)?

That depends primarily on how the network combines input variables to compute the net input to the next (hidden or output) layer. If the input variables are combined via a distance function (such as Euclidean

distance) in an RBF network, standardizing inputs can be crucial. The contribution of an input will depend heavily on its variability relative to other inputs. If one input has a range of 0 to 1, while another input has a range of 0 to 1,000,000, then the contribution of the first input to the distance will be swamped by the second input. So it is essential to rescale the inputs so that their variability reflects their importance, or at least is not in inverse relation to their importance. For lack of better prior information, it is common to standardize each input to the same range or the same standard deviation. If you know that some inputs are more important than others, it may help to scale the inputs such that the more important ones have larger variances and/or ranges.

If the input variables are combined linearly, as in an MLP, then it is rarely strictly necessary to standardize the inputs, at least in theory. The reason is that any rescaling of an input vector can be effectively undone by changing the corresponding weights and biases, leaving you with the exact same outputs as you had before. However, there are a variety of practical reasons why standardizing the inputs can make training faster and reduce the chances of getting stuck in local optima. Also, weight decay and Bayesian estimation can be done more conveniently with standardized inputs.

The main emphasis in the NN literature on initial values has been on the avoidance of saturation, hence the desire to use small random values. How small these random values should be depends on the scale of the inputs as well as the number of inputs and their correlations. Standardizing inputs removes the problem of scale dependence of the initial weights.

But standardizing input variables can have far more important effects on initialization of the weights than simply avoiding saturation. Assume we have an MLP with one hidden layer applied to a classification problem and are therefore interested in the hyperplanes defined by each hidden unit. Each hyperplane is the locus of points where the net-input to the hidden unit is zero and is thus the classification boundary generated by that hidden unit considered in isolation. The connection weights from the inputs to a hidden unit determine the orientation of the hyperplane. The bias determines the distance of the hyperplane from the origin. If the bias terms are all small random numbers, then all the hyperplanes will pass close to the origin. Hence, if the data are not centered at the origin, the hyperplane may fail to pass through the data cloud. If all the inputs have a small coefficient of variation, it is quite possible that all the initial hyperplanes will miss the data entirely. With such a poor initialization, local minima are very likely to occur. It is therefore important to center the inputs to get good random initializations. In particular, scaling the inputs to [-1,1] will work better than [0,1], although any scaling that sets to zero the mean or median or other measure of central tendency is likely to be as good, and robust estimators of location and scale (Iglewicz, 1983) will be even better for input variables with extreme outliers.

For example, consider an MLP with two inputs (X and Y) and 100 hidden units. The graph at lines-10to10.gif shows the initial hyperplanes (which are lines, of course, in two dimensions) using initial weights and biases drawn from a normal distribution with a mean of zero and a standard deviation of one. The inputs X and Y are both shown over the interval [-10,10]. As you can see, most of the hyperplanes pass near the origin, and relatively few hyperplanes go near the corners. Furthermore, most of the hyperplanes that go near any given corner are at roughly the same angle. That is, the hyperplanes that pass near the upper right corner go predominantly in the direction from lower left to upper right. Hardly any hyperplanes near this corner go from upper left to lower right. If the network needs to learn such a hyperplane, it may take many random initializations before training finds a local optimum with such a hyperplane.

Now suppose the input data are distributed over a range of [-2,2] or [-1,1]. Graphs showing these regions can be seen at lines-2to2.gif and lines-1to1.gif. The initial hyperplanes cover these regions rather thoroughly, and few initial hyperplanes miss these regions entirely. It will be easy to learn a hyperplane passing through any part of these regions at any angle.

But if the input data are distributed over a range of [0,1] as shown at lines0to1.gif, the initial hyperplanes are concentrated in the lower left corner, with fewer passing near the upper right corner. Furthermore, many initial hyperplanes miss this region entirely, and since these hyperplanes will be close to saturation over most of the input space, learning will be slow. For an example using the 5-bit parity problem, see "Why not code binary inputs as 0 and 1?"

If the input data are distributed over a range of [1,2] as shown at lines1to2.gif, the situation is even worse. If the input data are distributed over a range of [9,10] as shown at lines9to10.gif, very few of the initial hyperplanes pass through the region at all, and it will be difficult to learn any but the simplest classifications or functions.

It is also bad to have the data confined to a very narraw range such as [-0.1,0.1], as shown at lines-0.1to0.1.gif, since most of the initial hyperplanes will miss such a small region.

Thus it is easy to see that you will get better initializations if the data are centered near zero and if most of the data are distributed over an interval of roughly [-1,1] or [-2,2]. If you are firmly opposed to the idea of standardizing the input variables, you can compensate by transforming the initial weights, but this is much more complicated than standardizing the input variables.

Standardizing input variables has different effects on different training algorithms for MLPs. For example:

- Steepest descent is very sensitive to scaling. The more ill-conditioned the Hessian is, the slower the convergence. Hence, scaling is an important consideration for gradient descent methods such as standard backprop.
- Quasi-Newton and conjugate gradient methods begin with a steepest descent step and therefore are scale sensitive. However, they accumulate second-order information as training proceeds and hence are less scale sensitive than pure gradient descent.
- Newton-Raphson and Gauss-Newton, if implemented correctly, are theoretically invariant under scale changes as long as none of the scaling is so extreme as to produce underflow or overflow.
- Levenberg-Marquardt is scale invariant as long as no ridging is required. There are several different ways to implement ridging; some are scale invariant and some are not. Performance under bad scaling will depend on details of the implementation.

For more information on ill-conditioning, see ftp://ftp.sas.com/pub/neural/illcond/illcond.html

Two of the most useful ways to standardize inputs are:

- Mean 0 and standard deviation 1
- Midrange 0 and range 2 (i.e., minimum -1 and maximum 1)

Note that statistics such as the mean and standard deviation are computed from the training data, not from the validation or test data. The validation and test data must be standardized using the statistics computed from the training data.

Formulas are as follows:

```
Notation:

   X = value of the raw input variable X for the ith training case
    i

   S = standardized value corresponding to X
    i                                      i
```

```
   N = number of training cases


Standardize X  to mean 0 and standard deviation 1:
             i

          sum X
           i   i
   mean = ------
             N

                                 2
                  sum( X - mean)
                    i    i
   std  = sqrt( ---------------- )
                        N - 1


       X  - mean
        i
   S = ---------
    i     std


Standardize X  to midrange 0 and range 2:
             i

               max X  +  min X
                i   i     i   i
   midrange = ----------------
                      2


   range = max X  -  min X
            i   i     i   i


       X  - midrange
        i
   S = -------------
    i     range / 2
```

Various other pairs of location and scale estimators can be used besides the mean and standard deviation, or midrange and range. Robust estimates of location and scale are desirable if the inputs contain outliers. For example, see:

> Iglewicz, B. (1983), "Robust scale estimators and confidence intervals for location", in Hoaglin, D.C., Mosteller, M. and Tukey, J.W., eds., *Understanding Robust and Exploratory Data Analysis,* NY: Wiley.

Should I standardize the target variables (column vectors)?

Standardizing target variables is typically more a convenience for getting good initial weights than a necessity. However, if you have two or more target variables and your error function is scale-sensitive like the usual least (mean) squares error function, then the variability of each target relative to the others can effect how well the net learns that target. If one target has a range of 0 to 1, while another target has a range of 0 to 1,000,000, the net will expend most of its effort learning the second target to the possible exclusion of the first. So it is essential to rescale the targets so that their variability reflects their importance, or at least is not in inverse relation to their importance. If the targets are of equal importance, they should typically be standardized to the same range or the same standard deviation.

The scaling of the targets does not affect their importance in training if you use maximum likelihood estimation and estimate a separate scale parameter (such as a standard deviation) for each target variable. In this case, the importance of each target is inversely related to its estimated scale parameter. In other words, noisier targets will be given less importance.

For weight decay and Bayesian estimation, the scaling of the targets affects the decay values and prior distributions. Hence it is usually most convenient to work with standardized targets.

If you are standardizing targets to equalize their importance, then you should probably standardize to mean 0 and standard deviation 1, or use related robust estimators, as discussed under Should I standardize the input variables (column vectors)? If you are standardizing targets to force the values into the range of the output activation function, it is important to use lower and upper bounds for the values, rather than the minimum and maximum values in the training set. For example, if the output activation function has range [-1,1], you can use the following formulas:

```
Y = value of the raw target variable Y for the ith training case
 i

Z = standardized value corresponding to Y
 i                                      i

            upper bound of Y  +  lower bound of Y
midrange = ---------------------------------------
                              2


range = upper bound of Y  -  lower bound of Y


    Y  - midrange
     i
Z = -------------
 i    range / 2
```

For a range of [0,1], you can use the following formula:

```
             Y  - lower bound of Y
              i
Z = -------------------------------------
 i  upper bound of Y  -  lower bound of Y
```

And of course, you apply the inverse of the standardization formula to the network outputs to restore them to the scale of the original target values.

If the target variable does not have known upper and lower bounds, it is not advisable to use an output activation function with a bounded range. You can use an identity output activation function or other unbounded output activation function instead; see Why use activation functions?

Should I standardize the variables (column vectors) for unsupervised learning?

The most commonly used methods of unsupervised learning, including various kinds of vector quantization, Kohonen networks, Hebbian learning, etc., depend on Euclidean distances or scalar-product similarity measures. The considerations are therefore the same as for standardizing inputs in RBF networks--see Should I standardize the input variables (column vectors)? above. In particular, if one input has a large variance and another a small variance, the latter will have little or no influence on the results.

If you are using unsupervised competitive learning to try to discover natural clusters in the data, rather than for data compression, simply standardizing the variables may be inadequate. For more sophisticated

methods of preprocessing, see:

Art, D., Gnanadesikan, R., and Kettenring, R. (1982), "Data-based Metrics for Cluster Analysis," Utilitas Mathematica, 21A, 75-99.

Jannsen, P., Marron, J.S., Veraverbeke, N, and Sarle, W.S. (1995), "Scale measures for bandwidth selection", J. of Nonparametric Statistics, 5, 359-380.

Better yet for finding natural clusters, try mixture models or nonparametric density estimation. For example::

Girman, C.J. (1994), "Cluster Analysis and Classification Tree Methodology as an Aid to Improve Understanding of Benign Prostatic Hyperplasia," Ph.D. thesis, Chapel Hill, NC: Department of Biostatistics, University of North Carolina.

McLachlan, G.J. and Basford, K.E. (1988), Mixture Models, New York: Marcel Dekker, Inc.

SAS Institute Inc. (1993), SAS/STAT Software: The MODECLUS Procedure, SAS Technical Report P-256, Cary, NC: SAS Institute Inc.

Titterington, D.M., Smith, A.F.M., and Makov, U.E. (1985), Statistical Analysis of Finite Mixture Distributions, New York: John Wiley & Sons, Inc.

Wong, M.A. and Lane, T. (1983), "A kth Nearest Neighbor Clustering Procedure," Journal of the Royal Statistical Society, Series B, 45, 362-368.

Should I standardize the input cases (row vectors)?

Whereas standardizing variables is usually beneficial, the effect of standardizing cases (row vectors) depends on the particular data. Cases are typically standardized only across the input variables, since including the target variable(s) in the standardization would make prediction impossible.

There are some kinds of networks, such as simple Kohonen nets, where it is necessary to standardize the input cases to a common Euclidean length; this is a side effect of the use of the inner product as a similarity measure. If the network is modified to operate on Euclidean distances instead of inner products, it is no longer necessary to standardize the input cases.

Standardization of cases should be approached with caution because it discards information. If that information is irrelevant, then standardizing cases can be quite helpful. If that information is important, then standardizing cases can be disastrous. Issues regarding the standardization of cases must be carefully evaluated in every application. There are no rules of thumb that apply to all applications.

You may want to standardize each case if there is extraneous variability between cases. Consider the common situation in which each input variable represents a pixel in an image. If the images vary in exposure, and exposure is irrelevant to the target values, then it would usually help to subtract the mean of each case to equate the exposures of different cases. If the images vary in contrast, and contrast is irrelevant to the target values, then it would usually help to divide each case by its standard deviation to equate the contrasts of different cases. Given sufficient data, a NN could learn to ignore exposure and contrast. However, training will be easier and generalization better if you can remove the extraneous exposure and contrast information before training the network.

As another example, suppose you want to classify plant specimens according to species but the specimens are at different stages of growth. You have measurements such as stem length, leaf length, and leaf width.

However, the over-all size of the specimen is determined by age or growing conditions, not by species. Given sufficient data, a NN could learn to ignore the size of the specimens and classify them by shape instead. However, training will be easier and generalization better if you can remove the extraneous size information before training the network. Size in the plant example corresponds to exposure in the image example.

If the input data are measured on an interval scale (for information on scales of measurement, see "Measurement theory: Frequently asked questions", at ftp://ftp.sas.com/pub/neural/measurement.html) you can control for size by subtracting a measure of the over-all size of each case from each datum. For example, if no other direct measure of size is available, you could subtract the mean of each row of the input matrix, producing a row-centered input matrix.

If the data are measured on a ratio scale, you can control for size by dividing each datum by a measure of over-all size. It is common to divide by the sum or by the arithmetic mean. For positive ratio data, however, the geometric mean is often a more natural measure of size than the arithmetic mean. It may also be more meaningful to analyze the logarithms of positive ratio-scaled data, in which case you can subtract the arithmetic mean after taking logarithms. You must also consider the dimensions of measurement. For example, if you have measures of both length and weight, you may need to cube the measures of length or take the cube root of the weights.

In NN aplications with ratio-level data, it is common to divide by the Euclidean length of each row. If the data are positive, dividing by the Euclidean length has properties similar to dividing by the sum or arithmetic mean, since the former projects the data points onto the surface of a hypersphere while the latter projects the points onto a hyperplane. If the dimensionality is not too high, the resulting configurations of points on the hypersphere and hyperplane are usually quite similar. If the data contain negative values, then the hypersphere and hyperplane can diverge widely.

Subject: Should I nonlinearly transform the data?

Most importantly, nonlinear transformations of the targets are important with noisy data, via their effect on the error function. Many commonly used error functions are functions solely of the difference abs(target-output). Nonlinear transformations (unlike linear transformations) change the relative sizes of these differences. With most error functions, the net will expend more effort, so to speak, trying to learn target values for which abs(target-output) is large.

For example, suppose you are trying to predict the price of a stock. If the price of the stock is 10 (in whatever currency unit) and the output of the net is 5 or 15, yielding a difference of 5, that is a huge error. If the price of the stock is 1000 and the output of the net is 995 or 1005, yielding the same difference of 5, that is a tiny error. You don't want the net to treat those two differences as equally important. By taking logarithms, you are effectively measuring errors in terms of ratios rather than differences, since a difference between two logs corresponds to the ratio of the original values. This has approximately the same effect as looking at percentage differences, abs(target-output)/target or abs(target-output)/output, rather than simple differences.

Less importantly, smooth functions are usually easier to learn than rough functions. Generalization is also usually better for smooth functions. So nonlinear transformations (of either inputs or targets) that make the input-output function smoother are usually beneficial. For classification problems, you want the class boundaries to be smooth. When there are only a few inputs, it is often possible to transform the data to a linear relationship, in which case you can use a linear model instead of a more complex neural net, and many things (such as estimating generalization error and error bars) will become much simpler. A variety

of NN architectures (RBF networks, B-spline networks, etc.) amount to using many nonlinear transformations, possibly involving multiple variables simultaneously, to try to make the input-output function approximately linear (Ripley 1996, chapter 4). There are particular applications, such as signal and image processing, in which very elaborate transformations are useful (Masters 1994).

It is usually advisable to choose an error function appropriate for the distribution of noise in your target variables (McCullagh and Nelder 1989). But if your software does not provide a sufficient variety of error functions, then you may need to transform the target so that the noise distribution conforms to whatever error function you are using. For example, if you have to use least-(mean-)squares training, you will get the best results if the noise distribution is approximately Gaussian with constant variance, since least-(mean-)squares is maximum likelihood in that case. Heavy-tailed distributions (those in which extreme values occur more often than in a Gaussian distribution, often as indicated by high kurtosis) are especially of concern, due to the loss of statistical efficiency of least-(mean-)square estimates (Huber 1981). Note that what is important is the distribution of the noise, not the distribution of the target values.

The distribution of inputs may suggest transformations, but this is by far the least important consideration among those listed here. If an input is strongly skewed, a logarithmic, square root, or other power (between -1 and 1) transformation may be worth trying. If an input has high kurtosis but low skewness, an arctan transform can reduce the influence of extreme values:

$$\arctan\left(c \, \frac{\text{input} - \text{mean}}{\text{stand. dev.}} \right)$$

where c is a constant that controls how far the extreme values are brought in towards the mean. Arctan usually works better than tanh, which squashes the extreme values too much. Using robust estimates of location and scale (Iglewicz 1983) instead of the mean and standard deviation will work even better for pathological distributions.

References:

Atkinson, A.C. (1985) *Plots, Transformations and Regression,* Oxford: Clarendon Press.

Carrol, R.J. and Ruppert, D. (1988) *Transformation and Weighting in Regression,* London: Chapman and Hall.

Huber, P.J. (1981), *Robust Statistics,* NY: Wiley.

Iglewicz, B. (1983), "Robust scale estimators and confidence intervals for location", in Hoaglin, D.C., Mosteller, M. and Tukey, J.W., eds., *Understanding Robust and Exploratory Data Analysis,* NY: Wiley.

McCullagh, P. and Nelder, J.A. (1989) *Generalized Linear Models,* 2nd ed., London: Chapman and Hall.

Masters, T. (1994), *Signal and Image Processing with Neural Networks: A C++ Sourcebook*, NY: Wiley.

Ripley, B.D. (1996), *Pattern Recognition and Neural Networks*, Cambridge: Cambridge University Press.

Subject: How to measure importance of inputs?

The answer to this question is rather long and so is not included directly in the posted FAQ. See ftp://ftp.sas.com/pub/neural/importance.html.

Also see Pierre van de Laar's bibliography at ftp://ftp.mbfys.kun.nl/snn/pub/pierre/connectionists.html, but don't believe everything you read in those papers.

Subject: What is ART?

ART stands for "Adaptive Resonance Theory", invented by Stephen Grossberg in 1976. ART encompasses a wide variety of neural networks based explicitly on neurophysiology. ART networks are defined algorithmically in terms of detailed differential equations intended as plausible models of biological neurons. In practice, ART networks are implemented using analytical solutions or approximations to these differential equations.

ART comes in several flavors, both supervised and unsupervised. As discussed by Moore (1988), the unsupervised ARTs are basically similar to many iterative clustering algorithms in which each case is processed by:

1. finding the "nearest" cluster seed (AKA prototype or template) to that case
2. updating that cluster seed to be "closer" to the case

where "nearest" and "closer" can be defined in hundreds of different ways. In ART, the framework is modified slightly by introducing the concept of "resonance" so that each case is processed by:

1. finding the "nearest" cluster seed that "resonates" with the case
2. updating that cluster seed to be "closer" to the case

"Resonance" is just a matter of being within a certain threshold of a second similarity measure. A crucial feature of ART is that if no seed resonates with the case, a new cluster is created as in Hartigan's (1975) leader algorithm. This feature is said to solve the "stability-plasticity dilemma" (See "Sequential Learning, Catastrophic Interference, and the Stability-Plasticity Dilemma"

ART has its own jargon. For example, data are called an "arbitrary sequence of input patterns". The current training case is stored in "short term memory" and cluster seeds are "long term memory". A cluster is a "maximally compressed pattern recognition code". The two stages of finding the nearest seed to the input are performed by an "Attentional Subsystem" and an "Orienting Subsystem", the latter of which performs "hypothesis testing", which simply refers to the comparison with the vigilance threshhold, not to hypothesis testing in the statistical sense. "Stable learning" means that the algorithm converges. So the often-repeated claim that ART algorithms are "capable of rapid stable learning of recognition codes in response to arbitrary sequences of input patterns" merely means that ART algorithms are clustering algorithms that converge; it does *not* mean, as one might naively assume, that the clusters are insensitive to the sequence in which the training patterns are presented--quite the opposite is true.

There are various supervised ART algorithms that are named with the suffix "MAP", as in Fuzzy ARTMAP. These algorithms cluster both the inputs and targets and associate the two sets of clusters. The effect is somewhat similar to counterpropagation. The main disadvantage of most ARTMAP algorithms is that they have no mechanism to avoid overfitting and hence should not be used with noisy data (Williamson, 1995).

For more information, see the ART FAQ at http://www.wi.leidenuniv.nl/art/ and the "ART Headquarters" at Boston University, http://cns-web.bu.edu/. For a statistical view of ART, see Sarle (1995).

For C software, see the ART Gallery at http://cns-web.bu.edu/pub/laliden/WWW/nnet.frame.html

References:

Carpenter, G.A., Grossberg, S. (1996), "Learning, Categorization, Rule Formation, and Prediction by Fuzzy Neural Networks," in Chen, C.H., ed. (1996) *Fuzzy Logic and Neural Network Handbook,* NY: McGraw-Hill, pp. 1.3-1.45.

Hartigan, J.A. (1975), *Clustering Algorithms,* NY: Wiley.

Kasuba, T. (1993), "Simplified Fuzzy ARTMAP," AI Expert, 8, 18-25.

Moore, B. (1988), "ART 1 and Pattern Clustering," in Touretzky, D., Hinton, G. and Sejnowski, T., eds., *Proceedings of the 1988 Connectionist Models Summer School,* 174-185, San Mateo, CA: Morgan Kaufmann.

Sarle, W.S. (1995), "Why Statisticians Should Not FART," ftp://ftp.sas.com/pub/neural/fart.txt

Williamson, J.R. (1995), "Gaussian ARTMAP: A Neural Network for Fast Incremental Learning of Noisy Multidimensional Maps," Technical Report CAS/CNS-95-003, Boston University, Center of Adaptive Systems and Department of Cognitive and Neural Systems.

Subject: What is PNN?

PNN or "Probabilistic Neural Network" is Donald Specht's term for kernel discriminant analysis. (Kernels are also called "Parzen windows".) You can think of it as a normalized RBF network in which there is a hidden unit centered at every training case. These RBF units are called "kernels" and are usually probability density functions such as the Gaussian. The hidden-to-output weights are usually 1 or 0; for each hidden unit, a weight of 1 is used for the connection going to the output that the case belongs to, while all other connections are given weights of 0. Alternatively, you can adjust these weights for the prior probabilities of each class. So the only weights that need to be learned are the widths of the RBF units. These widths (often a single width is used) are called "smoothing parameters" or "bandwidths" and are usually chosen by cross-validation or by more esoteric methods that are not well-known in the neural net literature; gradient descent is *not* used.

Specht's claim that a PNN trains 100,000 times faster than backprop is at best misleading. While they are not iterative in the same sense as backprop, kernel methods require that you estimate the kernel bandwidth, and this requires accessing the data many times. Furthermore, computing a single output value with kernel methods requires either accessing the entire training data or clever programming, and either way is much slower than computing an output with a feedforward net. And there are a variety of methods for training feedforward nets that are much faster than standard backprop. So depending on what you are doing and how you do it, PNN may be either faster or slower than a feedforward net.

PNN is a universal approximator for smooth class-conditional densities, so it should be able to solve any smooth classification problem given enough data. The main drawback of PNN is that, like kernel methods in general, it suffers badly from the curse of dimensionality. PNN cannot ignore irrelevant inputs without major modifications to the basic algorithm. So PNN is not likely to be the top choice if you have more than 5 or 6 nonredundant inputs. For modified algorithms that deal with irrelevant inputs, see Masters (1995) and Lowe (1995).

But if all your inputs are relevant, PNN has the very useful ability to tell you whether a test case is similar

(i.e. has a high density) to any of the training data; if not, you are extrapolating and should view the output classification with skepticism. This ability is of limited use when you have irrelevant inputs, since the similarity is measured with respect to all of the inputs, not just the relevant ones.

References:

Hand, D.J. (1982) *Kernel Discriminant Analysis,* Research Studies Press.

Lowe, D.G. (1995), "Similarity metric learning for a variable-kernel classifier," Neural Computation, 7, 72-85, http://www.cs.ubc.ca/spider/lowe/pubs.html

McLachlan, G.J. (1992) *Discriminant Analysis and Statistical Pattern Recognition,* Wiley.

Masters, T. (1993). *Practical Neural Network Recipes in C++*, San Diego: Academic Press.

Masters, T. (1995) *Advanced Algorithms for Neural Networks: A C++ Sourcebook,* NY: John Wiley and Sons, ISBN 0-471-10588-0

Michie, D., Spiegelhalter, D.J. and Taylor, C.C. (1994) *Machine Learning, Neural and Statistical Classification,* Ellis Horwood; this book is out of print but available online at http://www.amsta.leeds.ac.uk/~charles/statlog/

Scott, D.W. (1992) *Multivariate Density Estimation,* Wiley.

Specht, D.F. (1990) "Probabilistic neural networks," Neural Networks, 3, 110-118.

Subject: What is GRNN?

GRNN or "General Regression Neural Network" is Donald Specht's term for Nadaraya-Watson kernel regression, also reinvented in the NN literature by Schi\oler and Hartmann. (Kernels are also called "Parzen windows".) You can think of it as a normalized RBF network in which there is a hidden unit centered at every training case. These RBF units are called "kernels" and are usually probability density functions such as the Gaussian. The hidden-to-output weights are just the target values, so the output is simply a weighted average of the target values of training cases close to the given input case. The only weights that need to be learned are the widths of the RBF units. These widths (often a single width is used) are called "smoothing parameters" or "bandwidths" and are usually chosen by cross-validation or by more esoteric methods that are not well-known in the neural net literature; gradient descent is *not* used.

GRNN is a universal approximator for smooth functions, so it should be able to solve any smooth function-approximation problem given enough data. The main drawback of GRNN is that, like kernel methods in general, it suffers badly from the curse of dimensionality. GRNN cannot ignore irrelevant inputs without major modifications to the basic algorithm. So GRNN is not likely to be the top choice if you have more than 5 or 6 nonredundant inputs.

References:

Caudill, M. (1993), "GRNN and Bear It," AI Expert, Vol. 8, No. 5 (May), 28-33.

Haerdle, W. (1990), *Applied Nonparametric Regression,* Cambridge Univ. Press.

Masters, T. (1995) *Advanced Algorithms for Neural Networks: A C++ Sourcebook,* NY: John Wiley and Sons, ISBN 0-471-10588-0

Nadaraya, E.A. (1964) "On estimating regression", Theory Probab. Applic. 10, 186-90.

Schi\oler, H. and Hartmann, U. (1992) "Mapping Neural Network Derived from the Parzen Window Estimator", Neural Networks, 5, 903-909.

Specht, D.F. (1968) "A practical technique for estimating general regression surfaces," Lockheed report LMSC 6-79-68-6, Defense Technical Information Center AD-672505.

Specht, D.F. (1991) "A Generalized Regression Neural Network", IEEE Transactions on Neural Networks, 2, Nov. 1991, 568-576.

Wand, M.P., and Jones, M.C. (1995), *Kernel Smoothing,* London: Chapman & Hall.

Watson, G.S. (1964) "Smooth regression analysis", Sankhy\=a, Series A, 26, 359-72.

Subject: What does unsupervised learning learn?

Unsupervised learning allegedly involves no target values. In fact, for most varieties of unsupervised learning, the targets are the same as the inputs (Sarle 1994). In other words, unsupervised learning usually performs the same task as an auto-associative network, compressing the information from the inputs (Deco and Obradovic 1996). Unsupervised learning is very useful for data visualization (Ripley 1996), although the NN literature generally ignores this application.

Unsupervised competitive learning is used in a wide variety of fields under a wide variety of names, the most common of which is "cluster analysis" (see the Classification Society of North America's web site for more information on cluster analysis, including software, at http://www.pitt.edu/~csna/.) The main form of competitive learning in the NN literature is vector quantization (VQ, also called a "Kohonen network", although Kohonen invented several other types of networks as well--see "How many kinds of Kohonen networks exist?" which provides more reference on VQ). Kosko (1992) and Hecht-Nielsen (1990) review neural approaches to VQ, while the textbook by Gersho and Gray (1992) covers the area from the perspective of signal processing. In statistics, VQ has been called "principal point analysis" (Flury, 1990, 1993; Tarpey et al., 1994) but is more frequently encountered in the guise of k-means clustering. In VQ, each of the competitive units corresponds to a cluster center (also called a codebook vector), and the error function is the sum of squared Euclidean distances between each training case and the nearest center. Often, each training case is normalized to a Euclidean length of one, which allows distances to be simplified to inner products. The more general error function based on distances is the same error function used in k-means clustering, one of the most common types of cluster analysis (Max 1960; MacQueen 1967; Anderberg 1973; Hartigan 1975; Hartigan and Wong 1979; Linde, Buzo, and Gray 1980; Lloyd 1982). The k-means model is an approximation to the normal mixture model (McLachlan and Basford 1988) assuming that the mixture components (clusters) all have spherical covariance matrices and equal sampling probabilities. Normal mixtures have found a variety of uses in neural networks (e.g., Bishop 1995). Balakrishnan, Cooper, Jacob, and Lewis (1994) found that k-means algorithms used as normal-mixture approximations recover cluster membership more accurately than Kohonen algorithms.

Hebbian learning is the other most common variety of unsupervised learning (Hertz, Krogh, and Palmer 1991). Hebbian learning minimizes the same error function as an auto-associative network with a linear hidden layer, trained by least squares, and is therefore a form of dimensionality reduction. This error function is equivalent to the sum of squared distances between each training case and a linear subspace of the input space (with distances measured perpendicularly), and is minimized by the leading principal

components (Pearson 1901; Hotelling 1933; Rao 1964; Joliffe 1986; Jackson 1991; Diamantaras and Kung 1996). There are variations of Hebbian learning that explicitly produce the principal components (Hertz, Krogh, and Palmer 1991; Karhunen 1994; Deco and Obradovic 1996; Diamantaras and Kung 1996).

Perhaps the most novel form of unsupervised learning in the NN literature is Kohonen's self-organizing (feature) map (SOM, Kohonen 1995). SOMs combine competitive learning with dimensionality reduction by smoothing the clusters with respect to an a priori grid (see "How many kinds of Kohonen networks exist?") for more explanation). But Kohonen's original SOM algorithm does not optimize an "energy" function (Erwin et al., 1992; Kohonen 1995, pp. 126, 237). The SOM algorithm involves a trade-off between the accuracy of the quantization and the smoothness of the topological mapping, but there is no explicit combination of these two properties into an energy function. Hence Kohonen's SOM is not simply an information-compression method like most other unsupervised learning networks. Neither does Kohonen's SOM have a clear interpretation as a density estimation method. Convergence of Kohonen's SOM algorithm is allegedly demonstrated by Yin and Allinson (1995), but their "proof" assumes the neighborhood size becomes zero, in which case the algorithm reduces to VQ and no longer has topological ordering properties (Kohonen 1995, p. 111). The best explanation of what a Kohonen SOM learns seems to be provided by the connection between SOMs and principal curves and surfaces explained by Mulier and Cherkassky (1995) and Ritter, Martinetz, and Schulten (1992). For further explanation, see "How many kinds of Kohonen networks exist?"

A variety of energy functions for SOMs have been proposed (e.g., Luttrell, 1994), some of which show a connection between SOMs and multidimensional scaling (Goodhill and Sejnowski 1997). There are also other approaches to SOMs that have clearer theoretical justification using mixture models with Bayesian priors or constraints (Utsugi, 1996, 1997; Bishop, Svensén, and Williams, 1997).

For additional references on cluster analysis, see ftp://ftp.sas.com/pub/neural/clus_bib.txt.

References:

Anderberg, M.R. (1973), *Cluster Analysis for Applications,* New York: Academic Press, Inc.

Balakrishnan, P.V., Cooper, M.C., Jacob, V.S., and Lewis, P.A. (1994) "A study of the classification capabilities of neural networks using unsupervised learning: A comparison with k-means clustering", Psychometrika, 59, 509-525.

Bishop, C.M. (1995), *Neural Networks for Pattern Recognition,* Oxford: Oxford University Press.

Bishop, C.M., Svensén, M., and Williams, C.K.I (1997), "GTM: A principled alternative to the self-organizing map," in Mozer, M.C., Jordan, M.I., and Petsche, T., (eds.) *Advances in Neural Information Processing Systems 9,* Cambrideg, MA: The MIT Press, pp. 354-360. Also see http://www.ncrg.aston.ac.uk/GTM/

Deco, G. and Obradovic, D. (1996), *An Information-Theoretic Approach to Neural Computing,* NY: Springer-Verlag.

Diamantaras, K.I., and Kung, S.Y. (1996) *Principal Component Neural Networks: Theory and Applications,* NY: Wiley.

Erwin, E., Obermayer, K., and Schulten, K. (1992), "Self-organizing maps: Ordering, convergence properties and energy functions," Biological Cybernetics, 67, 47-55.

Flury, B. (1990), "Principal points," Biometrika, 77, 33-41.

Flury, B. (1993), "Estimation of principal points," Applied Statistics, 42, 139-151.

Gersho, A. and Gray, R.M. (1992), *Vector Quantization and Signal Compression,* Boston: Kluwer Academic Publishers.

Goodhill, G.J., and Sejnowski, T.J. (1997), "A unifying objective function for topographic mappings," Neural Computation, 9, 1291-1303.

Hartigan, J.A. (1975), *Clustering Algorithms,* NY: Wiley.

Hartigan, J.A., and Wong, M.A. (1979), "Algorithm AS136: A k-means clustering algorithm," Applied Statistics, 28-100-108.

Hecht-Nielsen, R. (1990), *Neurocomputing,* Reading, MA: Addison-Wesley.

Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley: Redwood City, California.

Hotelling, H. (1933), "Analysis of a Complex of Statistical Variables into Principal Components," Journal of Educational Psychology, 24, 417-441, 498-520.

Ismail, M.A., and Kamel, M.S. (1989), "Multidimensional data clustering utilizing hybrid search strategies," Pattern Recognition, 22, 75-89.

Jackson, J.E. (1991), *A User's Guide to Principal Components,* NY: Wiley.

Jolliffe, I.T. (1986), *Principal Component Analysis,* Springer-Verlag.

Karhunen, J. (1994), "Stability of Oja's PCA subspace rule," Neural Computation, 6, 739-747.

Kohonen, T. (1995/1997), *Self-Organizing Maps,* Berlin: Springer-Verlag.

Kosko, B.(1992), *Neural Networks and Fuzzy Systems,* Englewood Cliffs, N.J.: Prentice-Hall.

Linde, Y., Buzo, A., and Gray, R. (1980), "An algorithm for vector quantizer design," IEEE Transactions on Communications, 28, 84-95.

Lloyd, S. (1982), "Least squares quantization in PCM," IEEE Transactions on Information Theory, 28, 129-137.

Luttrell, S.P. (1994), "A Bayesian analysis of self-organizing maps," Neural Computation, 6, 767-794.

McLachlan, G.J. and Basford, K.E. (1988), *Mixture Models,* NY: Marcel Dekker, Inc.

MacQueen, J.B. (1967), "Some Methods for Classification and Analysis of Multivariate Observations,"Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1, 281-297.

Max, J. (1960), "Quantizing for minimum distortion," IEEE Transactions on Information Theory, 6, 7-12.

Mulier, F. and Cherkassky, V. (1995), "Self-Organization as an Iterative Kernel Smoothing Process," Neural Computation, 7, 1165-1177.

Pearson, K. (1901) "On Lines and Planes of Closest Fit to Systems of Points in Space," Phil. Mag., 2(6), 559-572.

Rao, C.R. (1964), "The Use and Interpretation of Principal Component Analysis in Applied Research," Sankya A, 26, 329-358.

Ripley, B.D. (1996) *Pattern Recognition and Neural Networks,* Cambridge: Cambridge University Press.

Ritter, H., Martinetz, T., and Schulten, K. (1992), *Neural Computation and Self-Organizing Maps: An Introduction,* Reading, MA: Addison-Wesley.

Sarle, W.S. (1994), "Neural Networks and Statistical Models," in SAS Institute Inc., *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc., pp 1538-1550, ftp://ftp.sas.com/pub/neural/neural1.ps.

Tarpey, T., Luning, L>, and Flury, B. (1994), "Principal points and self-consistent points of elliptical distributions," Annals of Statistics, ?.

Utsugi, A. (1996), "Topology selection for self-organizing maps," Network: Computation in Neural Systems, 7, 727-740, http://www.aist.go.jp/NIBH/~b0616/Lab/index-e.html

Utsugi, A. (1997), "Hyperparameter selection for self-organizing maps," Neural Computation, 9, 623-635, available on-line at http://www.aist.go.jp/NIBH/~b0616/Lab/index-e.html

Yin, H. and Allinson, N.M. (1995), "On the Distribution and Convergence of Feature Space in Self-Organizing Maps," Neural Computation, 7, 1178-1187.

Zeger, K., Vaisey, J., and Gersho, A. (1992), "Globally optimal vector quantizer design by stochastic relaxation," IEEE Transactions on Signal Procesing, 40, 310-322.

Subject: Help! My NN won't learn! What should I do?

The following advice is intended for inexperienced users. Experts may try more daring methods.

If you are using a multilayer perceptron (MLP):

- Check data for outliers. Transform variables or delete bad cases as appropriate to the purpose of the analysis.
- Standardize quantitative inputs as described in "Should I standardize the input variables?"
- Encode categorical inputs as described in "How should categories be encoded?"
- Make sure you have more training cases than the total number of input units. The number of training cases required depends on the amount of noise in the targets and the complexity of the function you are trying to learn, but as a starting point, it's a good idea to have at least 10 times as many training cases as input units. This may not be enough for highly complex functions. For classification problems, the number of cases in the smallest class should be at least several times the number of input units.
- If the target is:

    - quantitative, then it is usually a good idea to standardize the target variable as described in "Should I standardize the target variables?" Use an identity (usually called "linear") output activation function.
    - binary, then use 0/1 coding and a logistic output activation function.
    - categorical with 3 or more categories, then use 1-of-C encoding as described in "How should categories be encoded?" and use a softmax output activation function as described in "What is a softmax activation function?"

- Use a tanh (hyperbolic tangent) activation function for the hidden units. See "Why use activation functions?" for more information.

- Use a bias term (sometimes called a "threshold") in every hidden and output unit. See "Why use a bias/threshold?" for an explanation of why biases are important.

- When the network has hidden units, the results of training may depend critically on the random initial weights. You can set each initial weight (including biases) to a random number such as any of the following:
    - A uniform random variable between -2 and 2.
    - A uniform random variable between -0.2 and -0.2.
    - A normal random variable with a mean of 0 and a standard deviation of 1.
    - A normal random variable with a mean of 0 and a standard deviation of 0.1.

  If any layer in the network has a large number of units, you will need to adjust the initial weights (not including biases) of the connections from the large layer to subsequent layers. Generate random initial weights as described above, but then divide each of these random weights by the square root of the number of units in the large layer. More sophisticated methods are described by Bishop (1995).

  Train the network using several (anywhere from 10 to 1000) different sets of random initial weights. For the operational network, you can either use the weights that produce the smallest training error, or combine several trained networks as described in "How to combine networks?"

- If possible, use conventional numerical optimization techniques as described in "What are conjugate gradients, Levenberg-Marquardt, etc.?" If those techniques are unavailable in the software you are using, get better software. If you can't get better software, use RPROP or Quickprop as described in "What is backprop?" Only as a last resort should you use standard backprop.

- Use batch training, because there are fewer mistakes that can be made with batch training than with incremental (sometimes called "online") training. If you insist on using incremental training, present the training cases to the network in random order. For more details, see "What are batch, incremental, on-line, off-line, deterministic, stochastic, adaptive, instantaneous, pattern, epoch, constructive, and sequential learning?"

- If you have to use standard backprop, you must set the learning rate by trial and error. Experiment with different learning rates. If the weights and errors change very slowly, try higher learning rates. If the weights fluctuate wildly and the error increases during training, try lower learning rates. If you follow all the instructions given above, you could start with a learning rate of .1 for batch training or .01 for incremental training.

  Momentum is not as critical as learning rate, but to be safe, set the momentum to zero. A larger momentum requires a smaller learning rate.

For more details, see What learning rate should be used for backprop?"

- Use a separate test set to estimate generalization error. If the test error is much higher than the training error, the network is probably overfitting. Read Part 3: Generalization of the FAQ and use one of the methods described there to improve generalization, such as early stopping, weight decay, or Bayesian learning.

- Start with one hidden layer.

  For a classification problem with many categories, start with one unit in the hidden layer; otherwise, start with zero hidden units. Train the network, add one or few hidden units, retrain the network, and repeat. When you get overfitting, stop adding hidden units. For more information on the number of hidden layers and hidden units, see "How many hidden layers should I use?" and "How many hidden units should I use?" in Part 3 of the FAQ.

  If the generalization error is still not satisfactory, you can try:

  - adding a second hidden layer
  - using an RBF network
  - transforming the input variables
  - deleting inputs that are not useful
  - adding new input variables
  - getting more training cases
  - etc.

If you are writing your own software, the opportunities for mistakes are limitless. Perhaps the most critical thing for gradient-based algorithms such as backprop is that you compute the gradient (partial derivatives) correctly. The usual backpropagation algorithm will give you the partial derivatives of the objective function with respect to each weight in the network. You can check these partial derivatives by using finite-difference approximations (Gill, Murray, and Wright, 1981) as follows:

1. Be sure to standardize the variables as described above.

2. Initialize the weights `W` as described above. For convenience of notation, let's arrange all the weights in one long vector so we can use a single subsbcript `i` to refer to different weights `W_i`. Call the entire set of values of the initial weights `w0`. So `W` is a vector of variables, and `w0` is a vector of values of those variables.

3. Let's use the symbol `F(W)` to indicate the objective function you are trying to optimize with respect to the weights. If you are using batch training, `F(W)` is computed over the entire training set. If you are using incremental training, choose any one training case and compute `F(W)` for that single training case; use this same training case for all the following steps.

4. Pick any one weight `W_i`. Initially, `W_i = w0_i`.

5. Choose a constant called `h` with a value anywhere from .0001 to .00000001.

6. Change the value of `W_i` from `w0_i` to `w0_i + h`. Do not change any of the other weights. Compute the value of the objective function `f1 = F(W)` using this modified value of `W_i`.

7. Change the value of `W_i` to `w0_i - h`. Do not change any of the other weights. Compute another new value of the objective function `f2 = F(W)`.

8. The central finite difference approximation to the partial derivative for `W_i` is `(f2-f1)/(2h)`. This

value should usually be within about 10% of the partial derivative computed by backpropagation, except for derivatives close to zero. If the finite difference approximation is very different from the partial derivative computed by backpropagation, try a different value of `h`. If no value of `h` provides close agreement between the finite difference approximation and the partial derivative computed by backpropagation, you probably have a bug.

9. Repeat the above computations for each weight `W_i` for `i=1, 2, 3, ...` up to the total number of weights.

References:

Bishop, C.M. (1995), *Neural Networks for Pattern Recognition,* Oxford: Oxford University Press.

Gill, P.E., Murray, W. and Wright, M.H. (1981) *Practical Optimization,* Academic Press: London.

Next part is part 3 (of 7). Previous part is part 1.