The Handbook of Brain Theory and Neural Networks

EDITED BY
Michael A. Arbib

A Bradford Book
THE MIT PRESS
Cambridge, Massachusetts
London, England

Neurosmithing: Improving Neural Network Learning

Russell Reed and Robert J. Marks II

Introduction

The goal of supervised training is to make the system output equal to a desired target function for any input. The standard approach is to (1) define an error function measuring the difference between the target and actual output functions, (2) determine how changes in parameters (network weights) affect the error, and (3) adjust parameters in a way that reduces the error. The backpropagation algorithm is the most commonly used technique for training multilayer perceptrons. Typically, the error function is the sum of squared differences between the desired targets $t(x_k)$ and the actual network outputs $y(x_k)$ summed over all training patterns k,

$$E(w) = \sum_k (t(x_k) - y(x_k))^2$$

Because the network output is a function of the weights, E is a function of w. If it could be plotted as a function of w, E might look like a rough landscape with hills and valleys, high where E is high and low where E is low. Backpropagation, as an approximation to gradient descent, could then be viewed as placing a marble at some random point on the landscape and

letting it roll to the lowest point. The core of the algorithm is a repeated loop in which (1) the derivative chain rule is applied to determine how weight changes affect the error and (2) the weights are adjusted by small increments in the direction that reduces the error. In "batch" mode, every training pattern is considered before each weight change, and the algorithm approximates gradient descent when the step size is small enough. In "on-line" mode, a random subset of patterns (usually just one) are considered before each weight change. When the step size is small enough, this approximates stochastic gradient descent since the accumulated weight changes tend to average to the true (negative) gradient. Since weight updates are much more frequent, however, the error may decrease faster when the training data are highly redundant. An added benefit is that the randomness of the individual weight changes may help jostle the "marble" out of small "potholes" and thus help prevent convergence to shallow local minima.

In spite of its apparent simplicity, the algorithm has proven remarkably effective, and there are many examples of networks trained to implement relatively complex functions. This is not to say that difficulties never occur, however. Backpropagation training is often very time consuming, for example, and may converge to suboptimal solutions. In the following, some practical techniques are described that may be helpful to accelerate learning and avoid potential problems. Some of the remarks are very basic and may be viewed as a checklist of standard procedures. Others are more specific. Many of the remarks apply to any learning system, but unless otherwise stated, the focus is on supervised learning in feedforward networks such as multilayered sigmoid perceptrons.

Data Preparation

Neural networks are often trained from examples of a desired input-output relationship. Aside from possible constraints built into the architecture or training algorithm, the examples are the only information provided about the target function, so it is important that they adequately describe the function.

Distribution of the data. In general, larger data sets are desirable from the standpoint of statistical accuracy since sparse data may contain spurious correlations and miss significant features of the function. Since the data distribution provides information about the relative importance of different regions of the function, it should generally match the distribution of patterns that will occur in normal operation.

Redundant and irrelevant information. Conversely, since neural networks are often applied to tasks where little is known about the appropriate choice of variables and their relationship to the target function (indeed, other techniques might be used if more were known), there is a temptation to provide as much information as possible and let the network sort it out. This might be feasible when data are abundant and training times unimportant, but it may lead to poor generalization otherwise.

As a rule, any external knowledge about which variables are important and how they relate to the target function should be used to reduce the amount of irrelevant information presented to the network. Although (ideally) the system should learn to ignore redundant and irrelevant inputs, these make its task harder and, when training sets are small, there may not be enough information to demonstrate that extra inputs are actually irrelevant. If the input dimension exceeds the sample size, for example, the data can be fitted exactly by a linear equation which will probably generalize poorly.

Dimensionality-reducing preprocessing, such as principal components analysis, is often used to avoid this problem. An alternative is to place a bottleneck (narrow hidden layer) in the network structure, thereby forcing the system to eliminate redundancies. Since the representation formed at a bottleneck is related to the principal components, weight initialization from principal components information has been suggested (Georgiou and Koutsougeras, 1992).

A case in which redundant input variables may be desirable is when the data are noisy (but abundant), since they can be averaged to reduce the effective noise if the noises are independent.

Variable centering and normalization. Centering and normalization put variables with different ranges on an equal footing. Without normalization, a system modeling an electronic device with voltages from 0 to 10,000 V and currents from 0 to 0.01 A would probably need very small weights from the voltage inputs and large weights from the currents. The system is very poorly conditioned, and training times will probably be long. Since backpropagation weight changes are proportional to the signal magnitudes, a single learning rate would probably not work for both. If these were output targets (in a network with linear outputs), the network would almost surely ignore errors in the currents as long as voltage errors remain. A commonly used normalization is

$$X' = (X - \mu)/\sigma$$

where μ is the mean value of X, and σ is its standard deviation. Normalization based on minimum and maximum values is also common.

Known nonlinearities. In general, it helps to eliminate known nonlinearities. Conversion from Cartesian to cylindrical coordinates, for example, may simplify a problem. Functions involving products or ratios of positive inputs can be made linear by taking logarithms. Of course, these sorts of transformations are completely problem dependent.

Problem decomposition and modularization. Learning is almost always easier if a task can be broken into smaller noninteracting parts. Separate networks then can be trained independently for each subproblem and combined. The result is (1) shorter training times because each subnetwork is smaller and (2) better generalization because each subnetwork is better constrained by available examples. Assuming the subtasks are truly independent, a system which does both together cannot do better, and may do worse, since its task is more complex.

Realistic engineering applications almost always require some sort of high-level partitioning. Systems like a postal zip code reader, for example, usually have separate segmentation and recognition subsystems. The tasks can be partitioned because digit identity is basically independent of size, location, etc.

Problem decomposition is completely task dependent, of course. A problem may be broken down in many ways, and knowing how to partition a task is a large part of knowing how to solve it. When high-level human knowledge is unavailable but data are abundant, an alternative is to divide the input space into pieces, assigning relatively simple subnetworks to learn each piece. Clustering or vector quantization techniques (possibly implemented as neural networks) can form the necessary partition. An example of partitioning by self-organization is the "mixture-of-experts" model (Jacobs et al., 1991; see MODULAR AND HIERARCHICAL LEARNING SYSTEMS).

Architecture Selection

One of the central tasks in network design is selection of an architecture. The goal is to find a network powerful enough to solve the problem, yet simple enough to train easily and generalize well. An advantage of local representation systems such as radial basis functions, self-organizing maps, Adaptive Resonance Theory (ART), and others is that they usually train much faster than layered networks (trained by backpropagation). They tend to generalize less well from an equivalent amount of data, however, so they are best used when data are abundant.

Although much work has been done on selecting appropriate structures and sizes, it is still basically an art. An approach that often works well in practice is to guess an approximate initial size and then use node creation and pruning algorithms (see TOPOLOGY-MODIFYING NEURAL NETWORK ALGORITHMS) to adjust the size during training, along with generalization-aiding techniques to suppress overfitting problems that may occur if the net is too large.

Pruning

Since the target function is unknown, it is often impossible to predict what size or configuration is appropriate. Although one can train a number of networks and choose the smallest/least complex one that learns the data, this can be inefficient if many networks have to be trained before an acceptable one is found. Even if the optimum size were known, the smallest adequate network might be difficult to train.

The pruning approach is to train a network that is larger than necessary and then remove unnecessary parts. The large initial size allows reasonably quick learning with less sensitivity to parameters, while the reduced complexity of the trimmed system favors improved generalization. In several studies, pruning techniques have produced small networks that generalize well where it was very difficult to obtain a solution by training the small network (obtained by pruning) from scratch with random weights (Sietsma and Dow, 1991).

Many pruning techniques have been suggested; a survey can be found in Reed (1993). Many of the algorithms fall into two broad groups. One group estimates the sensitivity of the error to removal of elements and removes those with the least effect. Another group adds terms to the error function that penalize unnecessarily complex solutions; many of these can be viewed as forms of regularization. In general, sensitivity methods modify a trained network; the network is trained, sensitivities are estimated, and then elements are removed. Penalty methods, however, modify the cost function so that optimization drives unnecessary weights to zero and, in effect, removes them during training. Even if the weights are not actually removed, the network acts like a smaller system. An advantage is that training and pruning are effectively done in parallel so the network can adapt to minimize errors introduced by pruning.

Although pruning and penalty term methods often may be faster than searching for and training a minimum-size network, they do not necessarily reduce training times; larger networks may take longer to train because of sheer size, and pruning takes some time itself. The goal, however, is improved generalization rather than faster training speed.

Constructive Methods

The opposite approach to pruning is to build the network incrementally by adding elements until a suitable configuration is found. The basic idea is to start with a small network, train until the error stops decreasing and then add a new node (or nodes) and resume training, repeating until an acceptable error is achieved. Algorithms differ in the network structures used, when new units are added, where they are placed, how they are initialized, etc.

In some cases, constructive methods can be faster than pruning methods since significant learning may occur while the network is still small. The approaches are not incompatible and are often used together. Since constructive methods, when used alone, sometimes create larger networks than necessary, a follow-up pruning phase can be useful to reduce the size.

It should be noted that pruning and constructive techniques are a means of adjusting network size rather than a way of deciding what size is appropriate. Other criteria are often useful to decide when to stop adding or removing elements.

Weight Initialization

The normal initialization procedure is to set weights to "small" random values. The randomness is intended to break symmetry, while "small" weights are chosen to avoid immediate saturation. Typically, weights are randomly selected from a range such as $(-A/\sqrt{N}, A/\sqrt{N})$, where N is the number of inputs to the node and A is between 2 and 3. More structured methods of initialization are discussed in Wessels and Barnard (1992) and Nguyen and Widrow (1990).

Initialization from a decision tree is considered in Sethi (1990). Since decision trees can be constructed very quickly, overall training time may be much shorter.

For problems in which the desired input-output relationship is well understood and expressible by a small set of rules, initialization based on a fuzzy logic implementation has been suggested.

Shortening Learning Times

Many heuristics have been developed in an attempt to shorten training times. The standard techniques of "on-line" training and momentum both tend to increase learning speed. "On-line" learning can be faster than batch learning since, with M training patterns, on-line learning will make M times as many weight updates in the same time. The effectiveness presumably implies redundancy in the data such that small samples give nearly as much information as the complete set. With *momentum*, a fraction of the previous weight change is added to the current weight change to give the system memory. This tends to stabilize the direction of movement by averaging opposing changes and often allows use of larger learning rates. In the analogy of the marble on the hilly surface, this gives the marble inertia, allowing it to coast over relatively flat areas and roll over small bumps.

Adaptive Learning Rates

The learning rate parameter has a direct effect on learning times. The "best" value, however, depends on the task to be solved and varies with local characteristics of the error surface, which change as the network learns. Different nodes in the net also may have different optimal rates, so there are no general rules for choosing a good fixed value a priori. With very small learning rates, learning is slower than necessary, and the system may settle in local minima, which it could easily escape otherwise. Very large rates, however, may send the system on wild jumps in essentially random directions or, in less extreme cases,

cause it to oscillate around a solution instead of settling to a minimum.

An alternative to setting a fixed rate a priori is to change it dynamically. A typical approach is to start with a moderate value, reduce it when the error starts to oscillate, and increase it when the error is decreasing very slowly. A moderate initial rate allows the system to find a rough initial solution quickly; reduction to a small value then allows the system to settle to the minimum. One of the most cited references for automatic learning rate adjustment is Jacobs (1988).

More sophisticated optimization methods such as conjugate-gradient or quasi-Newton methods often converge much faster than simple gradient descent, but these generally assume the error surface is well approximated by a quadratic function and may not work well when the assumption is not valid. The approximation is usually valid near a minimum, though, and these techniques can speed up final convergence to the endpoint after a rough solution is found by other methods.

Avoiding Paralysis

Paralysis occurs when nodes are driven into saturation. Since the tails of the sigmoid function are flat, the slope becomes very small when the node input is large. Consequently, weight updates are small, and learning is slow. One cause of saturation is large weights which amplify a normal activity pattern and create large signals that saturate nodes in following layers. Another cause is excessively high learning rates. The $E(w)$ graph often has a "stair-step" shape with large nearly flat regions separated by steep "cliffs" where E changes abruptly for small changes in w. (This is especially true for binary classification problems.) In using large learning rates to cross the plateaus quickly, there is a risk of taking a huge step in a wild direction on reaching a cliff. This may then create large weights and lead to saturation.

In simulations, code can be added to detect paralysis before it becomes serious and correct it by reducing the learning rate. Keeping a copy of the weights allows a step to be retracted and the step size reduced if saturation occurs suddenly.

Another guard against paralysis is the use of a nonsaturating node nonlinearity. Quickprop (Fahlman, 1988) uses a normal sigmoid nonlinearity but adds a small constant, e.g., 0.1, to the calculated derivative so it does not go to zero on saturation. This avoids paralysis but may make it more difficult for the network to settle to a solution.

Another technique is to reduce the sigmoid gain or, equivalently, to scale all node input weights by a factor less than 1 when saturation is detected in a node. This preserves the direction of the weight vector while reducing its magnitude. Weight decay tends to have a similar effect, since, when a node saturates, the decay term dominates other weight changes and reduces weights until the node comes out of saturation. Training with input noise sometimes has similar effects and also may help "jostle" the system out of saturation.

Hints

Another idea for accelerating learning and improving generalization is the use of "hint functions" (Suddarth, 1988; Yu and Simmons, 1990). Additional output nodes are appended to the network and trained to learn additional functions related to the function of interest but easier to express or learn. The hints may accelerate convergence by generating nonzero derivatives in regions where the original error function is flat and may aid generalization by providing additional constraints penalizing solutions which somehow match the original function on the training samples but do not include intermediate concepts embedded in the hints. After training, the extra nodes may be removed. Yu and Simmons (1990) demonstrate accelerated learning and improved generalization for a five-bit parity function by the use of hint nodes that count the number of ON bits using a thermometer representation.

Improving Generalization

Although neural networks are trained to minimize errors on the training patterns, we usually want the system to generalize from the examples and learn the underlying function so it will do well on new examples from the same function. A rule of thumb for generalization is that small simple systems are preferable to large complex systems if they give equal performance. Poor generalization usually results when the response does extreme things (like oscillating wildly) just to fit the data points. Simple systems tend to generalize better because they have less power to do things that are not "supported by the data." That is, they have fewer degrees of freedom and are better constrained by the available data. Pruning is one of the major ways of reducing network size to favor generalization.

Size is not the only factor affecting generalization, however. A network which is too small will have insufficient power to fit the desired function and will perform poorly. Also, large networks can mimic smaller networks. In most cases, it is not obvious what size is sufficient, and there is the risk of choosing an overly complex system. The techniques summarized below are designed to prevent an overly powerful network from overfitting the data.

Early Stopping

A simple estimate of the true generalization performance can be obtained by measuring the error on a separate testing data set which the network does not see while training. Since this reduces the size of the training set and is subject to statistical errors, more sophisticated methods are sometimes used.

Typically the training and test set errors decrease together in early learning stages as the network learns major features of the target function. With an unnecessarily complex system, the test set error usually reaches a minimum at some point and then begins to increase as the network exploits idiosyncrasies of the training data. A simple way of avoiding overfitting is to monitor the test set error and stop when the minimum is detected. This technique can be used with most of the other generalization-aiding techniques.

Regularization

Most techniques for improving generalization work by imposing additional constraints on the solution. The idea behind regularization methods (see GENERALIZATION AND REGULARIZATION IN NONLINEAR LEARNING SYSTEMS) is that one of the least restrictive assumptions is that the target function is smooth, i.e., that small changes in the input do not cause large changes in the output. This bias is embedded in the learning algorithm by adding terms to the cost function that penalize nonsmooth solutions. A generic cost function is

$$E = \sum_k (t(x_k) - y(x_k))^2 + \lambda E(\text{complexity})$$

where E(complexity) measures the complexity of the solution and λ balances the tradeoff between smoothing and error reduction. The complexity term is often a differential operator measuring how much the output changes over the region of interest.

Although this provides a way of biasing the learning algorithm, success depends on selection of an appropriate value for λ to determine the strength of the bias. Most other generalization heuristics have a similar parameter balancing error reduction and other constraints. This is often chosen by cross-validation.

Constraints to discourage overfitting are usually most helpful in the final learning stages and may be harmful in early stages if they bias the solution too much before the network has "seen sufficient evidence." When initial weights are small, saturation and overfitting usually do not become problems until later. Thus, it is often useful to change λ dynamically, starting at 0 and increasing gradually once an acceptable error is achieved or when cross-validation indicates overfitting.

Weight Decay

Large weights tend to cause sharp transitions in the sigmoid functions, and thus large changes in the output result from small changes in the inputs. A simple way to obtain some of the benefits of pruning without complicating the learning algorithm much is to add a decay term like $-\beta w$ to the weight update rule. Nonessential weights then decay to zero and can be removed. Even if not removed, they have no effect on the output, so the network acts like a smaller system. This can be viewed as a form of regularization, since a $\beta\Sigma(w_i^2)$ regularizing term yields a $(-\beta w_i)$ decay term in the weight update rule. Several methods are compared in Hergert, Finnoff, and Zimmerman (1992).

A drawback of the $\Sigma(w_i^2)$ penalty term is that it favors weight vectors with many small components over ones with a few large components, even when this is an effective choice. An alternative is (Weigend, Rumelhart, and Huberman, 1991)

$$\lambda \sum_i w_i^2/(w_i^2 + w_o^2)$$

where w_o is a constant. For $|w_i| \ll w_o$, the cost of a weight is small but grows like w_i^2. For $|w_i| \gg w_o$, the cost saturates and approaches a constant λ, so the weight does not incur additional penalties.

Weight Sharing

"Soft weight sharing" (Nowlan and Hinton, 1992) is another method that allows large weights when they are effective by giving "preferred status" to several other weight values besides zero. This reduces system complexity by increasing correlation among weight values.

"Hard" weight sharing is commonly used in image processing networks where the same kernel is applied repeatedly at different positions in the image (see CONVOLUTIONAL NETWORKS FOR IMAGES, SPEECH, AND TIME SERIES). In a neural network, separate nodes could be used to apply the kernel at different locations, and the number of weights could be huge. Constraining nodes that compute the same kernel to have equal weights greatly reduces the number of independent parameters and makes an otherwise unmanageable problem tractable (LeCun et al., 1990).

Adding Noise to the Data

Many studies have noted that adding small amounts of input noise during training often helps generalization and fault tolerance. Since the network never sees the same pattern twice, it cannot simply memorize the training patterns. This is equivalent to imposing a smoothness assumption, since we are effectively telling the network that slightly different inputs give about the same output. The effective target function is obtained by convolving the noise density with the original function. This is a smoother version of the original and helps prevent overfitting because, although the original function may be known only at discrete sample points, the effective function is continuous over the entire input space. The network is forced to use excess degrees of freedom to approximate the smoothed function instead of forming an arbitrarily complex surface that may match the target only at the sample points. Even though the network may be large, it models a simpler system. A drawback of training with noise is that it can be very slow, and there is the question of how much noise to use.

Multiple Networks

Another idea for improving generalization is to combine the outputs of several systems that classify novel examples differently because of differences in architecture, randomness of initialization, variations in parameters, differences in training data, etc., or because completely different types of classifiers are used. With a mean-square-error function, the best generalization is expected when the system produces the expected value of all possible functions consistent with the examples, weighted by their probability of occurrence

$$f'(x) = \int f(x)p_f(f)\,df$$

Averaging outputs of different systems is a very simple approximation to this expected value and tends to damp out extreme behaviors not justified by the data. Combination of subsystems is also an issue in the "mixture-of-experts" model (see MODULAR AND HIERARCHICAL LEARNING SYSTEMS).

Discussion

A number of commonly used techniques for improving learning in multilayer perceptrons have been mentioned. Many are quite simple and easily implemented but can have a significant effect on the speed and probability of successful learning. This list is by no means complete, of course. Some of the most important factors affecting learning, e.g., representation of input-output variables, have not been considered.

Although neural networks are often said to learn solely from the examples, at the most basic level it is biases built into the network structure and training algorithms that determine what the network can learn from the data. The network designer implicitly manipulates biases at each stage of the process, from data preparation to final tuning of the training parameters. Many of the techniques mentioned here can be thought of as biases that guide the network by making some functions easier to learn while excluding others.

Road Map: Learning in Artificial Neural Networks, Deterministic
Background: Backpropagation: Basics and New Developments; Perceptrons, Adalines, and Backpropagation

References

Fahlman, S. E., 1988. Faster-learning variations of back-propagation: An empirical study, in *Proceedings of the 1988 Connectionist Models Summer School* (D. Touretzky, G. Hinton, and T. Sejnowski, Eds.), San Mateo, CA: Morgan Kaufmann, pp. 38–51.

Georgiou, G. M., and Koutsougeras, C., 1992. Embedding domain information in backpropagation, in *Proceedings of the SPIE Conference on Adaptive and Learning Systems*, Bellingham, WA: SPIE.

Hergert, F., Finnoff, W., and Zimmermann, H. G., 1992. A comparison of weight elimination methods for reducing complexity in neural

networks, in *Proceedings of the International Joint Conference on Neural Networks*, vol. 3, Piscataway, NJ: IEEE, pp. 980–987.

Jacobs, R. A., 1988, Increased rates of convergence through learning rate adaptation, *Neural Netw.*, 1:295–307.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E., 1991, Adaptive mixtures of local experts, *Neural Computat.*, 3:79–87.

LeCun, Y., et al., 1990, Handwritten digit recognition with a back-propagation network, in *Advances in Neural Information Processing 2* (D. S. Touretzky, Ed.), San Mateo, CA: Morgan Kaufmann, pp. 396–404.

Nguyen, D. H., and Widrow, B., 1990, Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights, in *Proceedings of the International Joint Conference on Neural Networks*, Piscataway, NJ: IEEE, pp. 211–226.

Nowlan, S. J., and Hinton, G. E., 1992, Simplifying neural networks by soft weight-sharing, *Neural Computat.*, 4:473–493.

Reed, R. D., 1993, Pruning algorithms: A survey, *IEEE Trans. Neural Netw.*, 4:740–744. ◆

Sethi, I. K., 1990, Entropy nets: From decision trees to neural networks, *Proc. IEEE*, 78:1605–1613.

Sietsma, J., and Dow, R. J. F., 1991, Creating artificial neural networks that generalize, *Neural Netw.*, 4:67–79.

Suddarth, S. C., 1988, The symbolic-neural method for creating models and control behaviors from examples, PhD thesis, University of Washington.

Weigend, A. S., Rumelhart, D. E., and Huberman, B. A., 1991, Generalization by weight-elimination applied to currency exchange rate prediction, in *Proceedings of the International Joint Conference on Neural Networks*, vol. 1, Piscataway, NJ: IEEE, pp. 837–841.

Wessels, L. F. A., and Barnard, E., 1992, Avoiding false local minima by proper initialization of connections, *IEEE Trans. Neural Netw.*, 3:899–905.

Yu, Y.-H., and Simmons, R. F., 1990, Extra output biased learning, in *Proceedings of the International Joint Conference on Neural Networks*, vol. 3, Piscataway, NJ: IEEE, pp. 161–166.