Feed-forward neural networks

Why network size is so important

George Bebis and
Michael Georgiopoulos

One critical aspect neural network designers face today is choosing an appropriate network size for a given application. Network size involves in the case of layered neural network architectures, the number of layers in a network, the number of nodes per layer, and the number of connections. Roughly speaking, a neural network implements a nonlinear mapping of $u=G(x)$. The mapping function G is established during a training phase where the network learns to correctly associate input patterns x to output patterns u. Given a set of training examples (x, u), there is probably an infinite number of different size networks that can learn to map input patterns x into output patterns u.

The question is, which network size is more appropriate for a given problem? Unfortunately, the answer to this question is not always obvious. Many researchers agree that the quality of a solution found by a neural network depends strongly on the network size used. In general, network size affects network complexity, and learning time. It also affects the generalization capabilities of the network; that is, its ability to produce accurate results on patterns outside its training set.

Here is a very illustrative analogy between neural network learning and curve fitting that highlights the importance of network size. There are two problems in curve fitting: 1) finding out the order of the polynomial, and 2) finding out the coefficients of the polynomial. Thus, given a set of data points, first we decide on the order of the polynomial we will use and then we compute the coefficients of the polynomial. This way we minimize the sum of the squared differences between required and predicted values. Once the coefficients have been computed, we can evaluate any value of the polynomial given a data point, even for data points that were not in the initial data set.

Image Bank/Todd Davidson

If the order of the polynomial chosen is very low, the approximations obtained are not good, even for points contained in the initial data set. On the other hand, if the order of the polynomial chosen is very high, very bad values may be computed for points not included in the initial data set. Fig. 1 illustrates these concepts. Similarly, a network having a structure simpler than necessary cannot give good approximations even for patterns included in its training set. A more complicated than necessary structure, "overfits" the training data, that is, it performs nicely on patterns included in the training set but performs very poorly on unknown patterns.

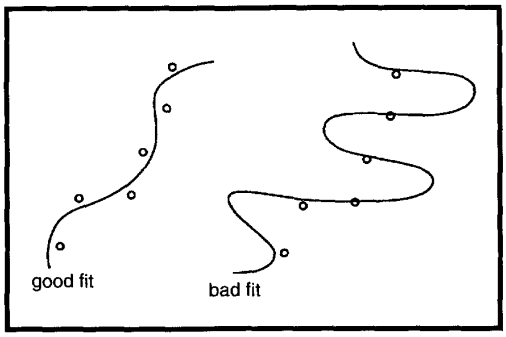

Fig. 1 Good and bad fits

What does the theory say?

Generally, the number of nodes in the input and output layers can be determined by the dimensionality of the problem. However, determining the number of hidden nodes is not straightforward. It requires first the determination of the number of hidden layers. There is a number of theoretical results concerning the number of hidden layers in a network. Specifically, Nielsen has shown that a network with two hidden layers can approximate any arbitrary nonlinear function and generate any complex decision region for classification problems.

Later Cybenko

0278-6648/94/$4.00 © 1994 IEEE

showed that a single layer is enough to form a close approximation to any nonlinear decision boundary. (Furthermore, it was shown that one hidden layer is enough to approximate any continuous function with arbitrary accuracy—when the accuracy is determined by the number of nodes in the hidden layer; also, one hidden layer is enough to represent any Boolean function).

Recently, Hornik and Stinchombe have come up with a more general theoretical result. They have shown that a single hidden layer feedforward network, with arbitrary sigmoid hidden layer activation functions, can approximate an arbitrary mapping from one finite dimensional space to another. This tells us that feed-forward networks can approximate virtually any function of interest to any desired degree of accuracy, provided enough hidden units are available.

Although the above theoretical results are of great importance, they don't give us an indication of how to choose the number of hidden units needed per hidden layer. Also, even if one hidden layer may be enough theoretically, in practice more than one hidden layer should be utilized for faster, more efficient problem solving. For example, we mentioned that one hidden layer is enough to approximate any continuous function. However, in some problems, a large number of hidden nodes may be required to achieve the desired accuracy. Thus, a network with two hidden layers and much fewer nodes should solve the same problem more efficiently. Hence, choosing an appropriate network size for a given problem is still something of an art.

Why are small and simple networks better?

In determining network size, one is only guided by intuition and some specific knowledge about the problem. For example, when the input is an image, it is more reasonable to define local receptive fields (that is, to use local connections), instead of full connectivity. This is because nearby pixels in the image are probably more correlated than pixels located far away from each other.

Unfortunately, when no a-priori knowledge about the problem is available, one has to determine the network size by trial and error. Usually, one has to train different size networks and if they don't yield an acceptable solution, then they are discarded. This procedure is repeated until an appropriate network is found.

Experience has shown that using the smallest network which can learn the task, is better for both practical and theoretical reasons. Smaller networks require less memory to store the connection weights and can be implemented in hardware more easily and economically. Training a smaller network usually requires less computations because each iteration is less computationally expensive. Smaller networks have also very short propagation delays from their inputs to their outputs. This is very important during the testing phase of the network, where fast responses are usually required.

Bigger networks generally need larger numbers of training examples to achieve good generalization performance. It has been shown that the training examples needed grows almost linearly with the number of hidden units. However, in many practical cases we have only a limited number of training data. This may lead to a very poor generalization. Finally, although bigger networks can perform more complicated mappings, when trained with limited training data they exhibit poor generalization.

Modifying the network architecture

We can see that solving a given problem using the smallest possible network provides a lot of advantages. However, choosing a smaller network over a larger one means that we actually restrict the number of free parameters in the network. Consequently, the error surface of a smaller network is more complicated and contains more local minima compared with the error surface of a larger network. Local minima in the error surface can seriously prevent a network from reaching a good solution. Thus, although smaller networks can prove to be very beneficial in terms of generalization, their training may require a lot of effort.

A number of techniques attempt to improve the generalization capabilities of a network by modifying not only the connection weights but also the architecture as training proceeds. These techniques can be divided into two categories. The first category includes methods that start with a big network and gradually eliminate the unnecessary nodes or connections. These methods are called *pruning* methods.

The second category includes methods that start with a small network and gradually add nodes or connections as needed. These methods are called *constructive* methods. Fig. 2 illustrates the classification of these methods.

Pruning methods

These methods attempt to find a quick solution by starting with a large network and reducing it. Considering the curve fitting problem, this approach implies that we start with a high order polynomial and gradually eliminate the higher order terms which do not contribute significantly to the fit. There are two main subcategories of pruning methods: (i) pruning based on modifying the error function and (ii) pruning based on sensitivity measures.

Modifying the error function

The basic idea is to modify the error function of the network in such a way that the unnecessary connections will have zero weight (or near zero) after training. Then, these connections can be removed without degrading the performance of the network. These approaches, which are also called weight decay approaches, actually encourage the learning algorithm to find solutions that use as few weights as possible. The simplest function can be formed by adding to the original error function a term proportional to the sum of squares

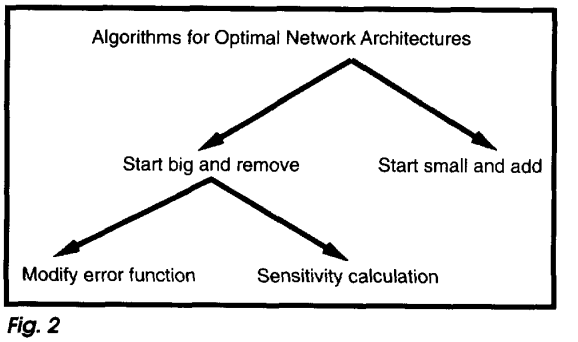

Fig. 2

of weights:

$$E = E_0 + \gamma \sum_i \sum_j w_{ij}^2$$

where E_0 is the original error function (sum of the squared differences between actual and desired output values), γ is a small positive constant which is used to control the contribution of the second term, and w_{ij} is the weight of the connection between node j of a layer, and node i of the immediately higher indexed layer. The above error function penalizes the use of more w_{ij}'s than necessary. To show this, lets see how the weight updating rule is changed. Assuming that we apply the gradient descent procedure to minimize the error, the modified weight update rule is given by:

$$\Delta w_{ij}(t) = -\alpha\left(\frac{\partial E}{\partial w_{ij}}\right)(t) = -\alpha\left(\frac{\partial E_0}{\partial w_{ij}}\right)(t) - 2\gamma\alpha w_{ij}(t)$$

where t denotes the t-th iteration and a denotes the learning rate. This expression can be written as:

$$w_{ij}(t+1) = -\alpha\left(\frac{\partial E_0}{w_{ij}}\right)(t) + (1-2\gamma\alpha)w_{ij}(t)$$

We can show that the magnitude of the weights decreases exponentially towards zero by computing the weight values after t weight adaptations:

$$w_{ij}(t) = \alpha \sum_{i=1}^{t} (1-2\gamma\alpha)^{t-i}\left(-\frac{\partial E_0}{2w_{ij}}\right)(t) + (1-2\gamma\alpha)^t w_{ij}(0)$$

(assuming $|1-2\gamma\alpha| < 1$). This approach has the disadvantage that all the weights of the network decrease at the same rate. However, it is more desirable to allow large weights to persist while small weights tend toward zero. This can be achieved by modifying the error function so that small weights are affected more significantly than large weights. This can be done, for example, by choosing the following modified function:

$$E = E_0 + \gamma \sum_i \sum_j \frac{w_{ij}^2}{1 + w_{ij}^2}$$

The weight updating rule then becomes:

$$w_{ij}(t+1) = -\alpha\left(\frac{\partial E_0}{\partial w_{ij}}\right)(t) + \left(1 - \frac{2\gamma\alpha}{(1+w_{ij}^2(t))^2}\right)w_{ij}(t)$$

In this case, small weights decrease more rapidly than large ones.

Sensitivity based methods

The general idea is to train a network in performing a given task and then compute how important the existence is of a connection or node. The least important connections or nodes are removed and the remaining network is retrained. In general, the sensitivity measurement does not interfere with training; however, it does require an extra amount of computational effort.

The key issue is finding a way to measure how a solution reacts to removing a connection or a node. Early approaches attempt to remove a connection by evaluating the change in the network's output error. If the error increases too much, then the weight must be restored.

More sophisticated approaches evaluate the change in error for all the connections and training data, and then remove the one connection which produces the least error increment. Obviously, both approaches are extremely time consuming, especially when large networks are considered.

A more heuristic approach is the "skeletonization" procedure proposed by Mozer and Smolensky. In their approach, the relevance of a connection is computed by measuring the error increase when the connection is removed. However, the relevance of a connection is computed by using information about the error surface's shape near the network's current minimum.

This is performed using the partial derivative of the error with respect to the connection to be removed. Connections with relevance below a certain threshold are then removed.

"Optimal brain damage" is another approach proposed by Le Cun and his co-workers. In this approach, the saliency of connections is measured using the second derivative of the error with respect to the connection. In particular, the saliency S_{ij} of a connection w_{ij} is given by

$$s_{ij} = \left(\frac{\partial^2 E}{\partial w_{ij}^2}\right)\frac{w_{ij}^2}{2}$$

where the second derivative measures the sensitivity of the error to small perturbations in w_{ij}. Thus, connections with small weight values having a significant influence on the solution are not removed.

Constructive methods

These start with a minimal network and add new nodes during training. Small networks get easily trapped to local minima, so new hidden nodes are added to change the shape of the weight space and to escape the local minima. Considering the curve fitting problem again, the constructive approach implies that we start with a very low order polynomial and we add higher order terms every time the current polynomial cannot provide a good fit.

There are a lot of interesting algorithms falling into this category and various heuristics are employed during the network growth process. To understand how these algorithms operate, we will focus on two of them: the *Upstart* algorithm which appears to be very successful for binary mappings, and the *Cascade Correlation* algorithm which appears to be very successful for real valued mappings. Both algorithms build a tree-like network by dividing the input space successively.

The *Upstart* algorithm builds a tree-like network in a top-down fashion. The nodes of the network are linear threshold units. The output of a node is either 0 or 1. The number of input-output nodes is determined by the nature of the problem. In the following description, we assume that the network consists of a single output node, that is, the network can assign an input pattern into two possible classes (one is represented by 0 and the other by 1). The steps can be summarized as follows:

Step 1. Initially, a single node is assumed which is connected to each input of the network. This node is trained to learn as many associations as possible.

Step 2. If that node creates wrong classifications, two "child" nodes are created to correct the erroneous "0" and "1" classifications of their 'parents'.

Step 3. The weights from the inputs to the parent node are frozen. The child nodes are connected to the inputs of the network. Each child node is trained to correct the erroneous "0" classifications and the erroneous "1" classifications.

Step 4. The child node which corrects erroneous "0" is connected to its parent node with a large positive weight. The child node that incorrectly fixes the 1's cases is connected to its parent node with a large negative weight.

Step 5. Two nodes are added for each child node in order to correct their wrong classifications. The old child nodes are treated as parent nodes now and the added nodes as new child

nodes. Training continues until all the data are classified correctly.

The *Upstart* algorithm is guaranteed to converge because each subnode is guaranteed to classify at least one of its targets correctly. This is true because for binary patterns, it is always possible to cut off a corner of the binary hypercube with a plane. The number of nodes grows linearly with the number of patterns. The resulting hierarchical architecture can be converted into an equivalent two layer network. This algorithm can easily be extended to be a classifier with more than one classes.

The *Cascade Correlation* algorithm also builds a tree-like network but in a bottom-up fashion. The number of input-output nodes is determined a-priori based on the problem's characteristics. The hidden units are added to the network one at a time and are connected in a cascaded way. The activation functions for the nodes may be sigmoidal functions or any mixture of non-linear activation functions. The main steps of the algorithm follow:

Step 1. Connect each input node to each output node and train the network over the entire training set to learn as many associations as possible.

Step 2. When no significant error reduction has occurred after a certain number of training iterations, run the network one last time over the entire training set to measure the error.

Step 3. If the error is less than a threshold then stop, otherwise add a new hidden node (candidate), and connect it with every input node and every pre-existing hidden node. Don't connect it to the output nodes yet.

Step 4. Freeze all the weights of the network. Adjust only the new hidden unit's input weights by trying to maximize the magnitude of the correlation between the new unit's output and the network's output error.

Step 5. If the new hidden unit stops improving (i.e., the error doesn't decrease), freeze its input weights and connect it to the output nodes.

Step 6. Train the network adjusting only the connections from the new hidden unit to the output nodes. Then, go back to step 3.

Step 4 of the algorithm actually maximizes the magnitude of the correlation between the candidate node's output and the network's error. Specifically, the function to be maximized has the form

$$S = \sum_{o,p} \left| (V_p - \bar{V})(E_{p,o} - \bar{E}_o) \right|$$

where V_p is the candidate node output when the p-th training pattern is presented to the network. The $E_{p,o}$ is the output error of the o output node when the p-th training pattern is presented to the network. Furthermore, V and E are the averages of V_p and $E_{o,p}$ over all the training patterns. Each new node added to the network actually learns a mapping which has the best possible correlation with the errors of the previous network.

The way the hidden output weights are modified is the following: if a hidden unit correlates positively with the error at a given output node, it will develop a negative connection weight to that node, attempting to cancel some of the error. Otherwise, it will develop a positive connection weight. The main advantages of the *Cascade Correlation* algorithm are: (i) it learns fast, (ii) it builds reasonably small networks, and (iii) it requires no back-propagation of error signals.

Just a few more words ...

So far we have discussed pruning and constructive approaches without comparing them. It is really difficult to say which approach performs better. Pruning has the disadvantage that often larger than the required size networks are chosen as starting points. Since a lot of time is spent training before pruning really starts, this may be computationally wasteful. In addition, since many medium-size networks can learn the same problem, the pruning procedure may not be able to find a small-size network because it may get stuck with one of these medium-size networks.

Constructive approaches tend to result with networks having long propagation delays from network inputs to network outputs. In addition, new nodes are usually assigned random weights which are likely to disrupt the approximate solution already found.

A probably superior approach would be a combination of constructive approaches with pruning. For example, the authors of the *Cascade Correlation* algorithm suggest that to keep the depth of the network small and to minimize the number of connections to the hidden and output nodes, simply use a weight decay approach by adding a penalty term to the error function. A general procedure for coupling constructive and pruning approaches would be the following: allow a small network to grow enough during training until a reasonable solution is found. Prune the network in order to achieve a smaller and faster network which provides the desired solution more efficiently and accurately.

Our discussion would be incomplete without mentioning two emerging approaches: weight sharing and Genetic algorithms.

Weight sharing tries to reduce the number of weights in a network by first assigning a local receptive field to each hidden node. Then the weights of hidden nodes, having receptive fields at different locations of their inputs, are given the same values. Thus, hidden nodes that have receptive fields with common weights actually try to detect the same kind of features but at different locations of their input. Weight sharing has been applied by Le Cun and his co-workers on a handwritten digit recognition task.

Genetic algorithms are a class of optimization procedures inspired by the biological mechanisms of reproduction. A genetic algorithm operates iteratively on a population of structures. Each one represents a candidate solution to the problem the algorithm is trying to solve. On each iteration, a new population is produced by first applying on the old population three fundamental operations: reproduction, crossover, and mutation. Then, each member of the population is evaluated through a fitness function. Members assigned a bad evaluation are discarded, while members assigned a good evaluation survive in future populations.

The key issue is how an architecture should be translated to be utilized by the genetic algorithm, and how much information about the architecture should be encoded into this representation. For example, Miller, Todd, and Hyde represent the network architecture as a connection matrix, mapped directly into a bit-string. Then, a number of different size networks are encoded in this way in order to form the initial population. The genetic operators act on this population. New populations are formed. A decoding procedure is applied on each member of the population in order to transform a bit-string into a legitimate network architecture. The fitness of

each network is evaluated by training it for a certain number of epochs, and recording the network's error. There has been some preliminary success associated with the problem of optimizing the network size but there is still a lot to be accomplished.

Read more about it

• J. Hertz, A. Krogh, and R. Palmer, *Introduction to the theory of Neural Computation,* Addison Wesley, 1991.

• D. Hush and B. Horne, "Progress in supervised Neural Networks," *IEEE Signal Processing Magazine,* Jan. 1993, pp. 8-39.

• R. Reed, "Pruning algorithms - a survey," *IEEE Transactions on Neural Networks,* vol. 4, no. 5, Sept. 1993, pp. 740-747.

• M. Wynne-Jones, "Constructive algorithms and pruning: improving the multilayer perceptron," in *Proc. of the 13th IMACS World Congress on Computation and Applied Mathematics,* R. Vichnevetsky and J. Miller, Eds., 1991, pp. 747-750.

• R. Hetcht-Nielsen, "Theory of the backpropagation neural networks," *Proc. Inter. Joint Conf Neural Networks,* vol. I, June 1989, pp. 593-611.

• G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems,* vol. 2, 1989, pp. 303-314.

• K Hornik and M. Stinchombe, "Multilayer feed-forward networks are universal approximators," in Artificial Neural Networks: Approximation and Learning Theory, H. White et. al., Eds., Blachvell press, Oxford, 1992.

• A. Blum and R. Rivest, "Training a 3-node neural network is NP-complete," *Proc. of the 1988 Workshop on Computational Learning,* 1988, pp. 9-18.

• M. Frean, "The Upstart algorithm: a method for constructing and training feed-forward networks," *Neural Computation,* vol. 2, 1990, pp. 198-209.

• S. Fahlman and C. Lebiere, "The Cascade-Correlation learning architecture," in *Advances in Neural Information Processing Systems II,* ed. D. Touretzky, 1990, pp. 524-532.

• Y. Le Cun, B. Boser, J. Denker, D. Henderson. R. Howard. W. Hubbard, and L. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation,* vol. 1, pp. 541-551,1989.

• G. Miller, P. Todd and S. Hegde, "Designing Neural Networks using Genetic Algorithms," *Third International Conference on Genetic Algorithms,* pp. 379-384, 1989.

About the authors

George Bebis is currently a Ph.D. student at the Electrical and Computer Engineering Department of the University of Central Florida. He has a B.S. in Mathematics, and a Masters in Computer Science, both from the University of Crete, Iraklion, Greece. His research interests lie in the areas of Computer Vision and Neural Networks.

Michael Georgiopoulos is an Associate Professor at the Electrical and Computer Engineering Department of the University of Central Florida. He has a Diploma in EE from the National Technical University of Athens, Greece, a Masters in EE and a Ph.D. in EE, both from the University of Connecticut, Storrs, CT. His research interests include communication systems and neural networks.