(Efficient Optimization of the Number of Hidden Neurons in Multilayer Neural Networks)

(Byoung-Tak Zhang)

Abstract This paper addresses the problem of optimizing the number of hidden neurons in multilayer neural networks which have been one of the most popular neural network models for practical applications so far. The presented method starts learning with a small network and increase the number of hidden units on demand. It also starts with a small training set and expand the set during further training. This incremental learning has the advantage that the number of hidden units can be efficiently determined by using only a subset of the given data. In addition, this method does not need an additional training phase after size optimization since the optimization is constructively performed during a normal learning process. The efficiency and optimality characteristics of the presented algorithm are examined by experiments.

1. Introduction

The goal of every data driven learning system, including neural networks, is to minimize the expected error rate on unknown data (generalization error) as well as to reduce the error for the given data (training error). In general, too small a neural network may not be able to learn the known data, resulting in poor training accuracy. On the other hand, an oversized network may overfit the data, resulting in poor generalization performance [12]. Therefore, the network size has to be optimized to satisfy both the necessary and sufficient conditions for good performance.

Recently, several learning algorithms have been proposed to construct optimal size feedforward networks. These constructive methods begin with a small network and introduce new hidden units and/or connections on demand. Some of them try to find a compact distributed representation, where the optimization is done with respect to the number of units in the hidden layer [3, 7]. Others construct more or less localized representations by building a deep or flat-but-wide network [5, 6, 10]. Each of these methods uses all of the given data for network construction and training.

The training time generally increases in proportion to the number of examples, while there is no guarantee that the generalization performance is improved by increasing the training set size [1].
Early studies have shown that a network trained on carefully selected examples generalizes much better than a network trained on the same number of randomly chosen examples [2]. Thus, various methods for selecting a concise training set have been proposed and tested, including [17, 9]. In the previous work the present author has shown that a representative training set could be found by alternating the network training and the example selection and that the learning could be accelerated enormously by the reduction of effective training data [14, 15, 16].

In this paper we extend our previous work by further exploiting the above findings to optimize network size. The basic idea is that we try to find an optimal number of hidden units by a carefully selected subset of the given data, instead of using all the known examples. In Section 2 we illustrate the principles on which the algorithm is based. Section 3 describes the algorithm in detail. Section 4 is devoted to empirical studies on the convergence and optimality properties of the algorithm. Sections 5 and 6 discuss some heuristics for applying the algorithm and summarize the results and implications of this work.

2. Three Principles

Our approach to network size optimization is based on three simple principles which were originally introduced in [13]. The first principle is to start learning with a small size network and expand it only if necessary. Let \( A_h \) denote a fully connected feedforward structure with \( h \) hidden units. We sequentially consider the structures of increasing complexity:

\[
A_1 \subset A_2 \subset A_3 \subset \ldots \subset A_h.
\]

This kind of bottom-up search ensures that the number of hidden units found are minimal for learning the given training set up to the specified performance level. Implementing the first principle is relatively straightforward in the case of multilayer perceptrons: add a new unit in the hidden layer.

The second principle is to train each network with a small data set and expand data incrementally. We use a data set of monotonically increasing size:

\[
D_0 \subset D_1 \subset D_2 \subset \ldots \subset D_N.
\]

That is, rather than attempting to directly minimize the total error \( E(D_N | w, A) \) of the current network \( A \) for the given data \( D_N \), we try to minimize a series of objective functions

\[
E(D_N | w, A) = E(D_0 | w, A) + E(D_1 | w, A) + \ldots + E(D_N | w, A)
\]

from left to right sequentially. Here \( E(D_n | w, A) \) is the error measure for the training set \( D_n \) given the architecture \( A \) and weight vector \( w \). The measure we have used is the total sum of squared errors, thus

\[
E(D_n | w, A) = \sum (y - f(x; w, A))^2
\]

where \( y \) is the desired output and \( f(x; w, A) \) is the actual output of the current network. To implement the second principle, we partition the given data set \( B \) into the training set \( D_N \) and the candidate set \( C \). Then the examples are chosen from \( C \) to \( D \) at each selection. Previous work has shown that learning can be significantly accelerated by training on the examples selected in this way [14].

Fig. 1 Control flow in network size optimization.

The third principle is to combine the incremental data expansion with the network expansion, which process we call self-development [13]. Starting with a small network and a small data set, the data set \( D \)
is expanded and learned until the network fails to learn $D$, in which case the network is expanded and learning resumes. This process is repeated until a network is found that shows a desired performance level (see Figure 1).

The basic idea behind this approach is that the self-development process has a synergetic effect: a small network leads to a good generalization performance for the current training set, which in turn guides the next selection of good examples. On the other hand, the use of a small set of training examples may result in fast optimization of the complexity for learning the given data.

Figure 2 schematically illustrates the change of the data set and the network size during the whole process. As indicated by the size of ellipses in the figure, the training set $D$ expands while the candidate set $C$ shrinks. As learning goes on and thus $D$ increases, the network $A$ grows as well. The purpose of this paper is to show that the self-development learning can be used to efficiently optimize the size of multilayer neural networks.

Fig. 2 The change of network size, training set, and candidate set.

3. Algorithm

The network size optimization is interleaved with the normal training process. The algorithm is, in essence, an iteration of three main steps: training the network, selecting examples and expanding the network, as outlined in the previous section. At each training step, the weights of the network with architecture $A_i$, $1 \leq h \leq H$, are adapted by a training set $D_i$, $1 \leq n \leq N$. If the training satisfies some success criterion, new examples are selected and added to the current training set. If the training fails to achieve the desired performance level after a sufficiently long period, the network grows to increase learning capacity. Then training continues until the next step of example selection or network expansion takes place. A more detailed description of the algorithm follows.

1. Initialize the training set $D^{(0)}$ and the candidate set $C^{(0)}$ by splitting all known data $B = D^{(0)} \cup C^{(0)}$. Initialize the network architecture $A^{(0)}$ and weights $w^{(s,g,t)}$. Set $s \leftarrow 0$, $g \leftarrow 0$, and $t \leftarrow 0$, where $s$, $g$, and $t$ are the indices for example selection steps, network growing steps, and weight training steps, respectively.

2. Train the network $A^{(s)}$ to reduce the error $E(D^{(s)}|w^{(s,g,t)}, A^{(s)})$ by using $D^{(s)}$ and adjusting $w^{(s,g,t)}$. Then set $t \leftarrow t + 1$. The weights are modified by the following rule:

$$ w^{(s+1,g,t)} = w^{(s,g,t)} + \Delta w^{(s,g,t)} , \quad (5) $$

$$ \Delta w^{(s,g,t)} = \xi \nabla E_{i} \big|_{w^{(s,g,t)}} + \eta \Delta w^{(s,g,t-1)} , \quad (6) $$

where $E_i$ is the total sum of the errors for $D^{(s)}$

$$ E_{i} = E(D^{(s)}|w^{(s,g,t)}, A^{(s)}) = \sum (y_i - f(x_i, w^{(s,g,t)}, A^{(s)}))^2 \quad (7) $$

and the error gradient $\nabla E_{i} \big|_{w^{(s,g,t)}}$ is approximated by the back-propagation procedure [11]. $\xi$ and $\eta$ are the step size and the momentum factor, respectively.

3. If the error is smaller than the error tolerance $\varepsilon$, go to step 6. Otherwise continue with the next step. The error tolerance value is defined as a function of the total number $K$ of free parameters of the network:

$$ \varepsilon_i = \varepsilon^{-1} K + \varepsilon^{-1} (1 + l) \cdot H_s + (H_s + 1) \cdot O_i , \quad (8) $$

where $I$, $O$, and $H_s$ are number of input, output, and
hidden units of network $A^{(s)}$. Here $K_x$ is used as a measure for determining the learning capacity of the network and $\tau$ is the parameter that determines the error tolerance for each adjustable weight [1, 4].

4. Trace the error reduction rate and check if got stuck at local minimum. If yes, go to next step, otherwise return to step 2.

5. Increase the network size by $n$ hidden units to create $A^{(s+1)}$ and set weight values $w_x^{(s+1)}$. Set $g \leftarrow g + 1$, and $t \leftarrow 0$. Return to step 2 to train the new network. The network grows by introducing $n$ new units to the hidden layer:

$$\sum_{h_j \in H_s} H' = H_s + n,$$

(9)

$$H_{s+1} = H_s + n,$$

(10)

where $H'$ denotes the index set of hidden units in $A^{(s)}$.

6. Test the generalization $G_c$ of the network on the original data $D^{(s)} \cup C^{(s)}$. If $G_c$ exceeds the desired performance level $l$, halt the algorithm.

7. Generate $D^{(s+1)}$ and $C^{(s+1)}$ by extracting $\lambda$ candidates from $C^{(s)}$ and appending them to $D^{(s)}$:

$$\sum_{h_j \in H_s} D^{(s+1)} = D^{(s)} \cup \{ x_i, y_i \},$$

(11)

$$\sum_{h_j \in H_s} C^{(s+1)} = C^{(s)} \cup \{ x_i, y_i \}.$$

(12)

Set $s \leftarrow s + 1$. Return to step 2 to learn the increased training set. Examples are selected that have the largest error for the current network:

$$E(x_i, y_i; x_i, \{ x_j, y_j \}, A^{(s)}; A^{(s)}).$$

(13)

where $c$ is the index for the candidate examples.

In step 5, the inserted hidden units have full connectivity with all input and output units. The values of new connections can be initialized in several ways. Two strategies are studied in the simulations. The first one is to reinitialize all the weights, including the existing ones. This strategy guarantees an escape from a local minimum and hence leads to a minimal network size. We will use this strategy for discrete problems and highly noisy continuous function approximations.

An alternative strategy is to keep the trained weights unchanged and to initialize new connections with values proportional to the average of the weights in the existing connections of the same weight layer. In this paper we suggest to use the following method: the weights from the new hidden units $j'$ to the output units $i$ have

$$w_{ij}^{(s+1)} = \gamma \cdot \frac{\sum_{h_j \in H_s} w_{ij}^{(s)}}{v_{ij}},$$

(14)

where $0 < \gamma < 1$ is a discount factor and the $v_{ij}$ terms are random values from the interval $[-v, +v]$, used to break the symmetries [11]. Likewise, the weights of the connections from the input $k$ to the new hidden units $j'$ are initialized by

$$w_{ij}^{(s+1)} = \gamma \cdot \frac{\sum_{h_j \in H_s} w_{ij}^{(s)}}{H'_{j'k}}.$$

(15)

The biases of the output and hidden units are initialized by the averages of existing bias values:

$$w_{ij}^{(s+1)} = \gamma \cdot \frac{\sum_{h_j \in H_s} w_{ij}^{(s)}}{v_{ij}}.$$

(16)

The latter strategy will ensure an effective escape from the local minimum without loss of information learned up to the expansion point. This is useful for continuous-valued problems with clean data or small noise.

4. Efficiency and Optimality Characteristics

To empirically examine the optimality of the network size found by the algorithm, we need a task for which the optimal number of hidden units can be calculated. One problem that serves this purpose is the autoassociation problem between $n$ dimensional binary vectors:

$$x = F(x), F:[0,1]^n \rightarrow [0,1]^n.$$

(17)

Note that the problem has a total number of $2^n$ examples and thus differs from the usual encoder/decoder problem [11] for which only $n$ examples are possible. Using a fully connected feedforward network of $H$ hidden units and $I = n$ inputs and outputs ($I$-$H$-$I$ architecture), the minimal
number of hidden units for solving this problem is $H = I$; between each pair of $k$th ($k = 1, \ldots, I$) input/output units a unique hidden unit must be allocated. We experimented with $I = 15$.

The optimality of the method was tested by varying the hidden layer growth size $u$. For each $u$, the network was initialized with $u$ hidden units and in each network expansion step the hidden layer was extended by $u$ new units. A total of 300 randomly generated examples are given ($N = 300$), and the learning was started with a training set consisting of two seed examples ($N_0 = 2$). In each selection phase the training set was expanded by 10 new examples ($\lambda = 10$).

The results for $u = 1, 2, 3, 4, 5$ are shown in Figure 3. For $u = 1, 3, 5$ the minimal network size $H = 15$ was always found. For $u = 2, 4$ the network converged to $H = 16$. In the latter case, the algorithm could not find the minimal size because the algorithm searches only the integer multiples of $u$:

$$H_g = H_0 + g \cdot u.$$

However the algorithm always finds the smallest size greater than the minimum, which is optimal in terms of the specified $u$. Notice that the network size grows rapidly at the early stage and the optimal network size was found, in the worst case, using only 72 examples out of 300 given examples.

The efficiency of the network size optimization is affected by the network growth parameter. Figure 4 depicts the total learning time $T$ as a function of the network growth size $u$, averaged over 30 runs. For the autoassociation problem the learning time decreases exponentially as $u$ increases. Increasing $u$ means, however, as was shown above, increasing difference of the optimized network size from the minimal size unless mod ($H_{\text{min}}, u$) = 0.

The efficiency of optimizing networks (denoted self) are compared with those of back-propagation networks (denoted bp), varying the initially given learning set size $N = 100, 200, 300, 400$. Table 1 summarizes the average results of 30 learning trials. Each self net was initialized with $H_0 = 3$ which was converged to $H = 15$ using expansion step size $u = 3$. The bp networks were initialized $H = 15$, the optimal size, which was fixed during learning. Weight modification step sizes were the same for both algorithms. For $N = 100$, i.e., given 100 examples, the self networks converged three times slower than the bp networks. In this case, both networks could not
achieve 100% generalization, but the generalization performance of the self networks was better than that of the bp networks. For \( N = 400 \), both networks achieved 100% generalization but the self networks converged faster than the bp networks. Between \( N = 100 \) and 400 as the given data size increases the convergence speed of self becomes increasingly superior to back-propagation networks.

Table 1 Comparison of performance between SELF networks and BP networks. The results are for the 15-H-15 autoassociation problem with \( u = 3 \), averaged over 30 runs. The symbol \( N \) denotes the size of initially given data set. The generalization performance \( G \) was measured on an independent test set of 1000 examples. \( N_s \) denotes the size of the training set actually used. The learning time \( T \) is in millions of weight modifications

<table>
<thead>
<tr>
<th>( N )</th>
<th>Method</th>
<th>( G )</th>
<th>( N_s )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>SELF</td>
<td>0.892</td>
<td>100.0</td>
<td>8.6</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td>0.860</td>
<td>100.0</td>
<td>3.0</td>
</tr>
<tr>
<td>200</td>
<td>SELF</td>
<td>0.977</td>
<td>186.6</td>
<td>11.0</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td>0.994</td>
<td>200.0</td>
<td>4.7</td>
</tr>
<tr>
<td>300</td>
<td>SELF</td>
<td>1.000</td>
<td>189.3</td>
<td>11.6</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td>1.000</td>
<td>300.0</td>
<td>9.4</td>
</tr>
<tr>
<td>400</td>
<td>SELF</td>
<td>1.000</td>
<td>216.8</td>
<td>12.5</td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td>1.000</td>
<td>400.0</td>
<td>14.8</td>
</tr>
</tbody>
</table>

The total learning time has a close relationship to the actually used training set size, as shown in Figure 5. As the given data size increases, the relative efficiency of the proposed algorithm gets improved. When a large amount of data is known, even with the additional cost for network size optimization, the presented algorithm can learn much faster than the backpropagation network with a ready-optimized number of hidden units.

The effectiveness of the method was further investigated on a more realistic data: 6800 handwritten digit patterns of 15x10 bitmap each. One half of them were used for training and the other half for generalization test. Since the main purpose of these experiments is to see the behavior of the proposed algorithm on a more realistic domain, we just compare its performance with the simple backpropagation algorithm using the optimized number of hidden units. We varied the parameter values \( u = 2, 5 \) and \( \lambda = 50, 100 \). This results in three variations of self(2,50), self(5,50), and self(5,100).

Figure 5 Relative efficiency of the proposed algorithm, averaged over 30 runs. The relative learning time \( T_{rel} = T_{SELF} / T_{BP} \) is measured by the ratio of the total learning time of the SELF network to that of the BP network. Similarly, \( G_{rel} \) and \( N_{rel} \) represents the relative performance in generalization performance and the training size reduction. It is observed that the efficiency of SELF is closely related with the reduction rate of training sets.

Figure 6 depicts the network size as a function of the training set size. All the three converged to a fixed number of hidden units: self(2,50) converged to \( H = 28 \) and the other two variations, self(5,50) and self(5,100), to \( H = 30 \). Note that the latter two algorithms cannot converge to \( H = 28 \) because they used \( u = 5 \) and the number 28 is not a multiple of 5. The algorithm self(5,50) was the fastest of the three, but there was no significant difference in generalization performance among them. The goodness of \( H = 28 \) and \( H = 30 \) was tested by training networks of fixed hidden units of \( H = 15, 20, 25, 30 \). We found that \( H = 30 \) showed the best performance, while \( H = 25 \) was the second best. This
implies that all the networks constructed can be said to be optimal with respect to the specified $u$.

To test the reasonableness of the network size optimized by the self-developmental process, we observed the performance of a multilayer neural network with 30 hidden units trained by the standard backpropagation procedure (no growing, no example selection). Figure 7 shows the results. For comparison, we also trained a network with 25 hidden units using the backpropagation, whose results are shown in Figure 8. As can be observed in the strong oscillations in training and generalization performance in Figure 8, a hidden layer consisting of 25 units seems insufficient in its capacity for solving this problem. In contrast, the smooth learning curves in Figure 7 show that 30 hidden units for this problem is a reasonable size.

Network size optimization capability of the method was tested further on the two-spirals problem [8]. The task is to learn to discriminate between two sets of training points which lie on two distinct spirals in the $x$-$y$ plane. These spirals coil three times around the origin and around one another. This problem is known to be a very difficult task for standard backpropagation networks. Thus, several researchers have used different architectures.

Lang and Witbrock [8] report results obtained on this problem using a 2-5-5-5-1 backpropagation network with short-cut connections. Stephen A. Frostrom of SAIC reports (unpublished) that a 2-20-10-1 network with no short-cut connections was used to solve this problem. Fahlman and Lebiere [5] report that the Cascade-Correlation algorithm is able to solve this problem. In 100 trials, the networks constructed ranged from 12 to 19 hidden units.

Difficulty of solving the two-spirals problem for standard backpropagation networks was confirmed by our own experiments as shown in Figure 9. The backpropagation networks consisting of one hidden layer, irrespective of how many hidden units, were not able to solve this problem using all the training data from the outset. But, we could solve this problem using the presented incremental learning algorithm.
The learning curves are shown in Figure 10. The algorithm found a solution using approximately 35% of given data. More important is the fact that the final network structure contains 20 hidden units which is the nearest size (for our parameter setting) to the optimal value reported by other researchers.

A better way is to think in terms of the maximum number of expansion steps \(gm\), instead of \(u\), since \(gm\) is the major factor determining the optimization cost. We found setting \(gm\) to be around 10 is a good starting point:

\[
\frac{gm}{10} = 10. \tag{19}
\]

The actual value of \(gm\) is determined by the optimal number of hidden units, \(H^*\), and the parameter \(u\) in the following way:

\[
\frac{gm}{u} = \left\lfloor \frac{H^*}{u} \right\rfloor \tag{20}
\]

Normally \(H^*\) is unknown but we can estimate its upper bound from the training data. A loose overestimation \(\hat{H}^*\) is sufficient for our purposes. By substituting the estimate \(\hat{H}^*\) in (20) and equating (19) and (20), we have the parameter value of \(u\):

\[
u = \left\lfloor \frac{\hat{H}^*}{gm} \right\rfloor = \left\lfloor \frac{\hat{H}^*}{10} \right\rfloor \tag{21}\]

Here the ceiling operator is used to assure \(u = 1\) if \(\hat{H}^* < \frac{gm}{10}\). Using this rule is equivalent to requiring the network size to be optimized within 10% of the minimum number of hidden units for learning the given data.

When one-shot optimization is not compelling, we may use a two-pass optimization, that is, a large \(u\) is applied first to find a rough size and then a smaller \(u\) to do a more refined optimization. Note that two (or more) pass optimization by self-development is different from the usual trial-and-error methods, where one trial does not give much insight into the minimal network size for learning the given examples. In contrast, after the first run of the self algorithm one can be statistically sure that the minimum lies within the interval \([H_{-\infty} - u + 1, H_{-\infty} - u]\), where \(H_{-\infty}\) is the number of hidden units for the final network expansion step in the run.

6. Summary and Conclusion

We presented a method for network size opti-
mization in multilayer neural networks. The method is based on three simple principles of incremental learning: data expansion, network expansion, and integration of both in a constructive way.

We have shown by experiments that the method can find a reasonable number of hidden units. For the autoassociation problem, the method could find always the minimal network size very efficiently. The efficiency comes from the fact that the number of hidden units are determined by using only a subset of the given data. Given sufficient amount of time to optimize, the method guarantees to find a minimal network size when the parameter value $\nu = 1$ is used.

The presented method does not need an additional training phase after the hidden unit optimization; the optimization is performed constructively during the normal learning process. In cases that a large amount of data is known, even with the additional cost for network size optimization, the presented algorithm can learn much faster than the backpropagation network with a ready-optimized number of hidden units.

References


