Application of Parallel Decomposition for Creation of Reduced Feed-Forward Neural Networks

Jacek Lewandowski[1], Mariusz Rawski[2], and Henryk Rybinski[1]

[1] ICS, Warsaw University of Technology
[2] IT, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
j.lewandowski,hrb@ii.pw.edu.pl,rawski@tele.pw.edu.pl

Abstract. In this paper a method of creating layers of feed-forward neural network that does not need to be learned is presented. Described approach is based on algorithms used in synthesis of logic circuits. Experimental results presented in the paper prove that this method may significantly decrease the time of learning process, increase generalization ability and decrease a probability of sticking in a local minimum. Further work and goals to achieve are also discussed.

Keywords: feed-forward neural network, logical circuit, parallel decomposition, argument reduction, pattern recognition, learning algorithm.

1 Introduction

Feed-forward neural network is a widely used tool in the area of artificial intelligence. It is mainly used in various applications of pattern recognition and prediction. However, feed-forward neural network has to be learned before it is used, and the learning process seems to be the most important step during the development of the network. There are few essential problems which can be met during the learning process. Most of algorithms use optimization methods to adjust weights of neurons - the network acts a parameterized function. The function used in the optimization process - most often based on mean square error – has a large number of local minimums and regions with very low slope. Therefore the algorithms can stuck in such local minima, and the learning process has to be restarted. Moreover, the larger the network is, the more neurons it contains, and with each single neuron n weights are associated, where n is the number of neurons in the previous layer. So, the number of weights grows much faster than the number of neurons and the learning algorithm has to perform significantly more calculations. On the other hand the structure of the neural network is also very important, because if there are too many neurons in the layer, the generalization abilities may decrease.

Since the neural network may also act as a logic circuit, functional decomposition based methods can be used to solve the problems mentioned above. The

M. Kryszkiewicz et al. (Eds.): RSEISP 2007, LNAI 4585, pp. 564–573, 2007.

methods similar to those used in logic synthesis have already been used in knowledge discovery [1]. The first attempts in application of functional decomposition to neural networks were presented in [2,3] and then in [4]. In this approach the serial decomposition was used to shatter a large neural network into a couple of small networks connected with each other. The parallel decomposition and argument reduction were used to decrease the initial number of inputs. In [4] it has been shown that this approach has some major disadvantages – the generalization ability may be lower, or may decrease faster with the increase of noise in data. Another disadvantage is that this method can be used if the neural network is created for the binary or just quantized input and output data. It is useless for data that consists of real numbers.

In [4], another application of functional decomposition in feed-forward neural networks has been mentioned, but it was neither explored, nor experimentally verified. In this paper we attempt to fill this gap. In the presented approach, actually the created neurons are already learned, and the parallel decomposition or argument reduction is used to reduce number of them to the necessary minimum. The experimental results show very interesting capabilities of the method, in particular the learning time has been reduced for up to 30 times, without loss of the generalization abilities.

In Section 2 we present some basic notions. Then in Section 3 the considered approach is described. Section 4 presents experimental results. Finally conclusions are presented in Section 5.

2 Basic Notions

2.1 Feed-Forward Neural Network

A typical feed-forward neural network consists of few layers. Each layer contains a number of neurons which are not connected to any other neuron in the same layer, but each one is connected to all neurons in the previous and next layers. The input connector has a parameter, called weight, which is multiplied by an input signal value and the result is passed to the neuron. The neuron sums up all the signals and passes the result to an activation function (linear, binary, sigmoid or other). Fig. 1 shows the structure of a feed-forward neural network [5,6,7].

For example, the output signal for the neuron j in the layer b can be calculated as follows:

$$f(n(jb)) = f\left(B_{jb} + \sum_{i=1}^{k_a} W_{ia-jb} \cdot f(n(ia))\right) \tag{1}$$

where W_{x-y} is the weight assigned to a connection between neurons x and y, k_a is the number of neurons in the layer a, B is the parameter called bias (sometimes it is treated as a weight connected to the virtual fixed input signal equal to 1), and $f(x)$ is an activation function.

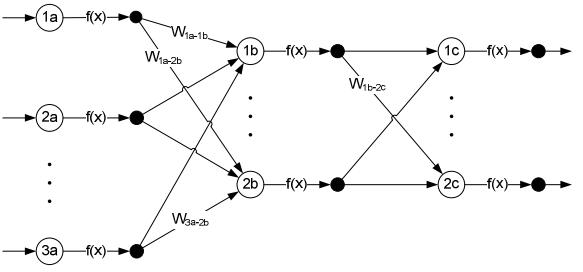

Fig. 1. A sample feed-forward neural network with 3 layers

In this paper the binary and sigmoid activation functions will be used.

$$f_{binary}(x) \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases} \tag{2}$$

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-ax}} \tag{3}$$

Let us note that formula (1) represents a hyper-plane that splits the hyper-space of the input signals into two parts. Especially when the binary activation function is used, the output value will be equal to 0 for the data points of the first part, and it will be equal to 1 for the data points of the other part. So, the aim of learning is to find such hyper-planes that can split the data points, for which the output signal is different. Fig. 2 shows a sample set of the data points for which the output signal is 0 (the black points) and 1 (the white points). The lines n1 - n7 show the partitions made by the corresponding neurons.

There are several learning algorithms that are based on calculating derivative of some error function in respect to all parameters. Usually, the most frequently used error function is the mean square error (MSE), calculated for the output of

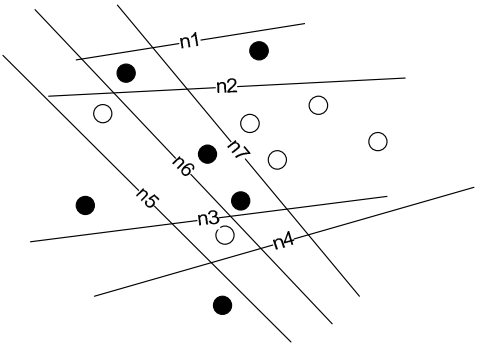

Fig. 2. A sample set of the data points

the neural network. Such algorithms require that the activation function is differentiable, therefore the usage of the binary functions for this case is impossible, and instead the sigmoid function is used [5,6,7]. In the presented approach we create few leading layers that contain neurons which have their weights already set, thus we do not need to apply the learning algorithm to them. Thus, in these layers we use the binary activation function. However end layers constitute a normal neural network, which has to be learned in a typical way, so we use the sigmoid activation function there.

2.2 Parallel Decomposition and Argument Reduction

In order to reduce a complexity of neural networks we create a number of leading layers with an extra number of neurons. Then we have to reduce them to the necessary minimum. To do this, a parallel decomposition and an argument reduction are used. These methods are commonly used in the logic synthesis of Boolean functions and we briefly present them below [8,9,10].

A problem occurs, when we want to implement a large Boolean function using components with a limited number of outputs. Note that such a parallel decomposition can also alleviate the problem of an excessive number of inputs of the function. This is because for typical functions most outputs do not depend on all input variables.

As an example let us consider a multiple-output function F (Table 1). Assume that F has to be decomposed into two components, G and H, with disjoint sets Y_G and Y_H of the output variables. Let us note that the set X_G of the input variables, on which the output variables from Y_G depend, may be smaller than X. Similarly, for the set X_H of input variables on which the outputs from Y_H depend, may be smaller than X. As a result, the components G and H have not only less outputs, but also less inputs than F. The exact formulation of the parallel decomposition problem depends on the constraints imposed by the implementation style. One possibility is to find the sets Y_G and Y_H, such that

Table 1. Function F

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	y_1	y_2	y_3	y_4	y_5	y_6
1	0	0	0	1	1	1	0	0	0	0	0	0	0	–	0
2	1	0	1	0	0	0	0	0	0	0	0	–	1	0	1
3	1	0	1	1	1	0	0	0	0	0	1	1	0	1	1
4	1	1	1	1	0	1	0	0	0	0	1	1	1	1	0
5	1	0	1	0	1	0	0	0	0	0	0	0	–	0	1
6	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0
7	1	1	1	0	0	0	0	0	0	1	0	–	0	1	0
8	1	0	1	1	0	1	0	0	0	1	1	0	0	–	1
9	1	0	1	1	0	1	1	0	0	–	1	0	1	–	1
10	1	1	1	0	0	0	0	1	0	1	0	1	0	1	–
11	0	0	0	1	1	1	0	0	1	0	0	1	0	–	1
12	0	0	0	1	1	0	0	0	1	–	–	1	0	0	0

$card(X_G + card(X_H$ is minimal. Partitioning the set of outputs into only two disjoint subsets is not important here, because the procedure can be applied iteratively for the resulting components G and H.

The minimal sets of input variables, on which each output of F depends, are:

$y_1 : \{x_1, x_2, x_6\}$
$y_2 : \{x_3, x_4\}$
$y_3 : \{x_1, x_2, x_4, x_5, x_9\}, \{x_1, x_2, x_4, x_6, x_9\}$
$y_4 : \{x_1, x_2, x_3, x_4, x_7\}$
$y_5 : \{x_1, x_2, x_4\}$
$y_6 : \{x_1, x_2, x_6, x_9\}$

An optimal two-block decomposition, minimizing $card(X_G)+card(X_H)$, is $Y_G = \{y_2, y_4, y_5\}$ and $Y_H = \{y_1, y_3, y_6\}$, with $X_G = \{x_1, x_2, x_3, x_4, x_7\}$ and $X_H = \{x_1, x_2, x_4, x_6, x_9\}$. The truth tables for the components G and $H$are shown in the tables 2 and 3.

Table 2. Function G of parallel decomposition

	X_1	x_2	x_3	x_4	x_7	y_2	y_4	y_5
1	0	0	0	1	0	0	1	0
2	1	0	1	0	0	0	1	0
3	1	0	1	1	0	1	0	1
4	1	1	1	1	0	1	1	1
5	0	0	1	1	0	1	1	0
6	1	1	1	0	0	0	0	1
7	1	0	1	1	1	1	1	–

Table 3. Function H of parallel decomposition

	x_1	x_2	x_4	x_6	x_9	y_1	y_3	y_6
1	0	0	1	1	0	0	0	0
2	1	0	0	0	0	0	0	1
3	1	0	1	0	0	0	1	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	1	1	0	0	0	1	1	0
7	1	0	1	1	0	1	0	1
8	0	0	1	1	1	0	1	1
9	0	0	1	0	1	-	1	0

The algorithm itself is general in the sense that the function to be parallel decomposed can be specified in a compact cube notation. The calculation of the minimal sets of input variables for each individual output can be a complex task. Thus in the practical implementations, heuristic algorithms are used which support calculations with the help of the so called indiscernible variables. But the simplest way to obtain the quasi-minimal set of input variables for an output of a Boolean function is just trying to eliminate an input variable and then checking the consistency of the function - if the function is still consistent, the input variable can be safely removed.

3 Application of the Decomposition to Reduce the Neural Network

The idea is based on the fact that each neuron represents a hyper-plane. We can manually create neuron by neuron, so that each represents a hyper-plane

splitting the data space. However it is very difficult to decide how to split the space, as there is unlimited number of solutions, the more that we decide it only for the training set without knowing the whole space. On the other hand we should build the network with possibly least number of neurons in order to achieve the network with the sufficient abilities to generalize. To this end, our algorithm should create a surplus set of hyper-planes and then to reduce it using a parallel decomposition and an argument reduction. In that way we create a layer with the reduced set of neurons. This step is repeated till the cardinality of the unique output signals of the layer is lower than of the previous layer. Finally, for the last layer we create a typical neural network to process the output data of the last but one layer. The steps are described in the following subsections in more details.

3.1 Creating Initial Set of Neurons

As stated above, at the beginning we devise an initial set of neurons, which may be superfluous. Let us assume that there are k data points in some n dimensional hyper-space. Each point has to be mapped to one of m possible output signals. The hyper-planes have to split the training data points that are mapped to the different output signals. A naive approach would be based on calculating the centroid of the data points for which the output signal is the same. Then a hyper-plane would be created between two randomly selected centroids, and the set of data points is shattered into two parts. The algorithm is repeated for each part till there is only a single centroid in each set. We have tested this approach, but it turned out to be inefficient and in some cases inconsistent.

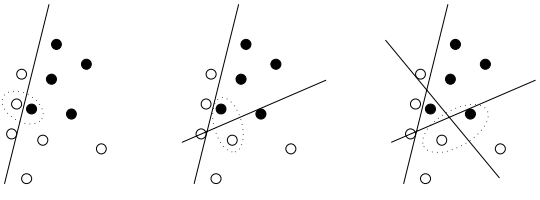

Fig. 3. Steps of the algorithm that creates hyper-planes

We base our approach on the observation that by splitting the closest distinguishable points we achieve the network with higher ability to distinct input data. Therefore we select such data points from varying groups that are closest to one other and create a hyper-plane between them. We repeat the process till the groups are homogenous. The algorithm is illustrated on Fig. 3.

3.2 Reducing the Set of Neurons

Each hyper-plane splits a hyper-space into two parts. Let us assume that for each data point, the value 0 is assigned if the data point is located in the first

Table 4. A layer of neurons presented as the truth table

Point number	Assignments for each hyper-plane					Output signal
	neuron 1	neuron 2	neuron 3	...	neuron n	
1	1	1	1	...	0	0 ...
2	0	0	1	...	0	0 ...
3	0	0	1	...	1	0 ...
...	...		...	...	...	... ...
k	0	1	0	...	1	1 ...

part of the hyper-space and the value 1 if one is located in the other part. This corresponds to assigning the value of the binary activation function (Table 4).

A logic function (similar to the one shown in Table 1) is created, so an argument reduction can be applied to it. The argument reduction removes unnecessary input arguments so the corresponding hyper-planes can be safely removed. Thus we will have a quasi-minimal set of neurons in a layer. We can also use a parallel decomposition and create the set with the minimum number of neurons for each output. Then we may group these sets to get the best results. We have performed a number of experiments, which have shown that the argument reduction used for the hyper-planes created by the algorithm sketched in p. 3.1 reduces the number of the neurons dozens times.

3.3 Normal Neural Network as the Output

The presented algorithms are not perfect. Although they significantly reduce the number of dimensions in the hyper-space for the layers, there is rather a slim chance that the cardinality of different output signals is equal to the cardinality of unique output signals of the system. To this end, we can (1) map the output signals of the last layer to the desired output signals or (2) create a simple neural network which can process them. We have selected the second solution for our experiments.

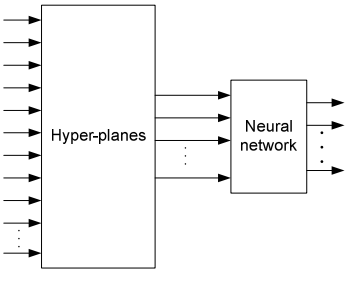

Fig. 4. A model of reduced neural network

4 Experimental Results

The experiments have been performed with the RPROP [6] learning algorithm for two cases: (1) a normal neural network and (2) a reduced neural network. The learning process was stopped when the error measure had fallen behind 0.01. If the algorithm had stuck in a local minimum, the learning process was started again, and a time was reset. In the experiments the neural network simulator and the decomposition tool created by authors were used. The measured variables were:

- a total learning time, i.e. the time of creating the hyper-planes and reducing the neural network (in case of testing the reduced network) + a time of learning),
- a learning result (a part of learning vectors that were correctly classified),
- a test result (a part of testing vectors that were correctly classified) and
- a number of local minima (an average number of events that the learning algorithm got stuck in a local minimum).

Each test was repeated for 5 times and an average value was calculated for each variable.

4.1 Comparison of Learning Time, Generalization Ability

In the first experiment 10 inputs and 4 outputs neural network was used. It analyzed the input vector, translating it to a code. The number of learning vectors was 492. We have divided the initial network into 4 parts for both normal and reduced neural network tests, so that each output was evaluated independently (one part per single output).

The second experiment was performed for the pattern classifier. Input data were matrices of 36 characters, each character in 3 variants – so the total number of learning vectors was equal to 108. There were 360 inputs and 36 outputs – each output is activated for the single character.

Table 5. The results of learning and testing the ones counter

Neural network	Total learning time	Learning result	Test result	Number of local minimums
Normal	38.6	0.9455	0.5779	3.6
Reduced	23.7	0.9565	0.6335	1.0

Table 6. The results of learning and testing the pattern classifier

Neural network	Total learning time	Learning result	Test result	Number of local minimums
Normal	33,9	0.9889	0.7630	0.0
Reduced	0,9	1.0000	0.7778	0.0

The results mentioned above show that the learning time was significantly shortened and the generalization abilities were slightly increased for the reduced network. Moreover, Table 5 shows that for the normal neural network the learning process stuck in the local minimum some 3 times more than for the reduced one.

4.2 Comparison of Abilities to Generalize for Different Noise Level in the Test Data

In the third experiment an input data were matrices of 93 characters, each character in 3 variants – so the total number of learning vectors was 279. The size of the input matrix was 1200. 7 outputs were binary representation of the character code. The tests were performed for 4 sets of the test vectors with a different noise level. The initial neural network was divided into 7 networks - each network for the single output.

Table 7 shows that the learning time was over 30 times shorter in the case of the reduced neural network. Furthermore the generalization abilities were better

Table 7. The learning time comparison

Neural network	Total learning time
Normal	345.5
Reduced	10.4

Table 8. The generalization ability for the different noise levels

Neural network	Test result - Generalization ability			
	Noise 25	Noise 50	Noise 100	Noise 200
Normal	0.515	0.389	0.278	0.121
Reduced	0.722	0.647	0.449	0.371

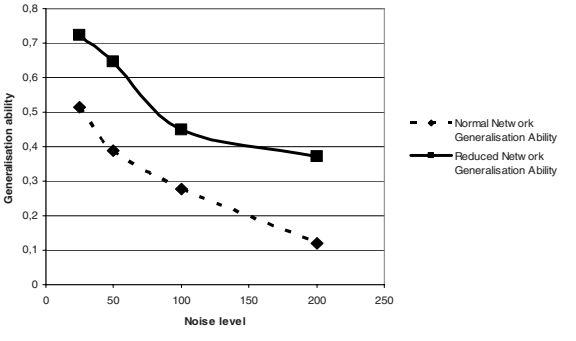

Fig. 5. The abilities to generalize for the different noise levels

for each examined noise level (Table 8). Fig. 5 depicts that they decrease slower for the reduced neural network than for the normal one.

5 Conclusions

The presented method allows simplifying a learning process of a feed-forward neural network. It is useful especially to solve various pattern recognition problems. The experimental results presented in the paper prove the effectiveness and efficiency of the proposed method. The time of learning, as well as, the probability of sticking in a local minimum were significantly reduced. The method allows also increasing the generalization abilities. The technique we put forward can be improved in the future works. The goal is to develop an algorithm for creating hyper-planes such that it would eliminate the need of using a normal neural network in the last layer.

References

1. Pawlak, Z.: Rough Sets: Theoretical Aspects of Reasoning About Data. Kluwer Academic Publishing, Dordrecht (1991)
2. Niewiadomski, H.: Serial decomposition of feedforward neural networks. Warsaw University of Technology. M. Sc. Thesis (2003)
3. Lewandowski, J., Rawski, M., Migacz, M., Rybiński, H.: Analysis of decomposition of feed-forward neural network and it's impact on ability to generalization. Multimedia and network information systems. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, pp. 269–278 (2006)
4. Lewandowski, J.: Dekompozycja w sieciach neuronowych. Warsaw University of Technology, M. Sc Thesis (2006)
5. Haykin, S.: Neural Networks. A Comprehensive Foundation, 2nd edn. Pearson Education, Inc. Delhi, India (2005)
6. Osowski, S.: Sieci neuronowe do przetwarzania informacji. Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa (2000)
7. Kasabov, N.K.: Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering. Massachusetts Institute of Technology (1996)
8. Łuba, T., Selvaraj, H., Nowicka, M., Kraśniewski, A.: Balanced multilevel decomposition and its applications in FPGA-based synthesis. In: Saucier, G., Mingnotte, A. (eds.) Logic and Architecture Synthesis, Chapman & Hall, Sydney (1995)
9. Łuba, T., Lasocki, R., Rybnik, J.: An Implementation of Decomposition Algorithm and its Application in Information Systems Analysis and Logic Synthesis. In: Ziarko, W. (ed.) Rough Sets, Fuzzy Sets and Knowledge Discovery. Workshops in Computing Series, pp. 458–465. Springer, Heidelberg (1994)
10. Brzozowski, J.A., Łuba, T.: Decomposition of Boolean Functions Specified by Cubes. Journal of Multi-Valued Logic & Soft. Computing 9, 377–417 (2003)