International Journal of Computer Applications (0975 – 8887)
Volume 85 – No 3, January 2014

Parallel Implementation of a Neural Network Learning Algorithm

S. Volokitin
Dept. of Computer Science
Southwest State University
Russia

ABSTRACT

This paper describes parallel implementation of an artificial neural network training algorithm and its effectiveness when applied to performing cryptographic functions. As a cryptographic function a permutations have been used because of its prevalence in complex cryptographic functions such as block ciphers. In order to enhance performance of artificial neural network training algorithm a method of backward propagation of errors has been parallelized.

General Terms

Computer Science, Algorithms

Keywords

Neural network; training algorithm; parallelism; cryptography

1. INTRODUCTION

As a result of widespread demand to transmit confidential data safely via computer networks, cryptography began to play key role in modern information technology, presenting an opportunity to protect sensitive data.

Therefore research and development of cryptographic algorithms and its optimization is impotent. In this work, artificial neural networks (ANNs) have been used to perform cryptographic functions, because ANNs are able to implement different operations after being trained [1].

Functionality of artificial neural networks is determined by its structure, connections between nodes, training algorithm and characteristics of neurons making up a neural network. Artificial neural networks possess a processing power because of their distributed structure and ability to learn.

2. CRYPTOGRAPHIC FUNCTION

The goal of this research is to find an efficient, parallel ANN training algorithm which could be used to train ANN to perform permutations of an input block of bits according to a training program. Transformation implementing by this network could be described by following expression:

$$(i_1,\ i_2,\ ..i_n) \tag{1}$$

In expression (1), i_1 is a place of a first input bit in an output vector of bits, while i_2 is a place of second input bit, etc. Depending on the number of input bits, the overall number of different permutations is $n!$.

The benefit of using ANN to perform cryptographic functions is related to the capability of changing implemented algorithms without changing network itself, just applying different training set. ANN could be trained to perform a reliable algorithm instead of a vulnerable one, in the same way as the SSL protocol allows to change compromised cryptographic algorithm to another reliable algorithm [2]. Such an approach is especially useful for cryptographic hardware, which cannot be modified after releasing. In addition to the above, there are papers which describe implementing cryptographic algorithms using programmable logic devices, emphasizing the significance of using flexible methods for developing hardware cryptographic systems [3 - 5].

3. ANN DESIGN

3.1 Structure of ANN

On abstraction level ANN performing permutation of bits is presented in Figure 1.

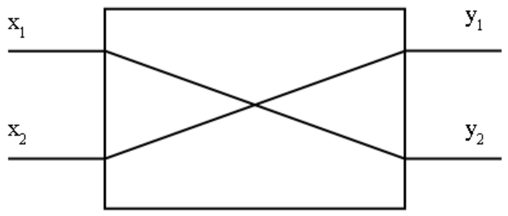

Fig. 1: ANN implementing primitive permutation

ANN depicted in Figure 1 has two inputs and two outputs, and it's capable to perform 2! different permutations. To train this artificial neural network, 2^n training pairs is required, which are presented in Table 1.

Table 1. Training set for primitive permutation

Input vector	Target vector
(0,0)	(0,0)
(0,1)	(1,0)
(1,0)	(0,1)
(1,1)	(1,1)

Since an amount of training pairs depends exponentially on a number of ANN inputs, there are a limited number of input bits. In this research, to estimate efficiency of parallelization of an ANN training algorithm, neural network consisting of 3 neuron layers with 16 neurons in each layer has been trained. The designed ANN is a fully connected feedforward neural network; its topology is presented in Figure 2.

As an activation function for all neurons in ANN, a sigmoid function has been used. It's presented in Formula 2.

$$y = \tanh(s * x) = \frac{2}{1+e^{-2\cdot s\cdot x}} - 1 \tag{2}$$

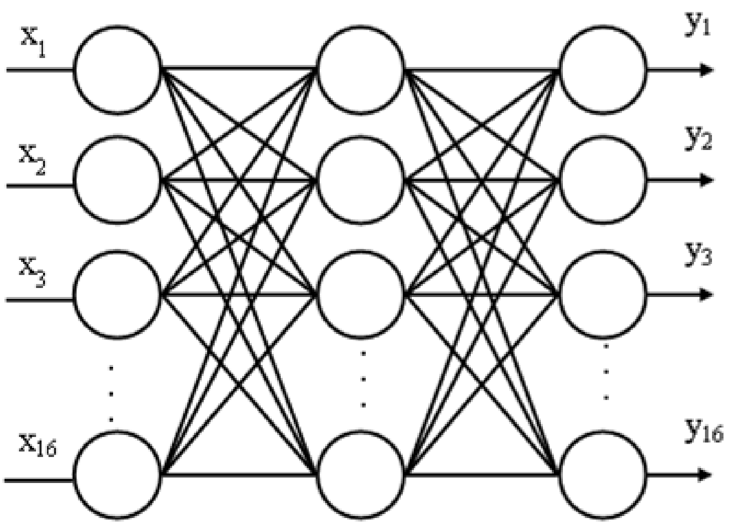

Fig. 2: Topology of designed ANN.

The activation function (Formula 2) is a hyperbolic tangent, where x is a weighted total of all input signals and s is a coefficient of steepness of activation function. A derivative of the activation function is presented in Formula 3.

$$y' = s \cdot (1 - y \cdot y) \tag{3}$$

3.2 Training algorithm

As a training algorithm, a method of backward propagation of errors has been chosen [6]. This method is an iterative gradient descent algorithm often used for training ANN. The key idea of this algorithm is a backward propagation of errors from output neurons to input ones.

To train ANN for each vector of input set, the following steps are implemented:

1. The vector of signals is given to artificial neurons of input.
2. The results of all outputs of neurons in layer L_i are calculated layer by layer.
3. The signal values of output neurons are compared with objective vector.
4. In case an error is more than an allowable error, value δ_{ij} is calculated from Formula 4 for each neuron in layer L_i starting from output layer.
5. For each neuron connection, a weight change Δw_{ij} is calculated from Formula 6.
6. The weights of all neuron connections are changed by value Δw_{ij}.

Calculating of weight change of neurons in output layer is computed from Formula 4, where r_i is an output value of neuron and t_i is a desired value.

$$\delta_i = -r_i(1 - r_i)(t_i - r_i) \tag{4}$$

Amending of neuron connections weights in layers other than the output layer is presented in Formula 5, where $w_{i,k}$ is the weight of connection between ith and kth neurons and *K* is the set of all neurons connected to ith neuron.

$$\delta_i = -r_i(1 - r_i)\sum\nolimits_{j \in K} \delta_k \cdot w_{i,k} \tag{5}$$

A change in weight is computed according to Formula 6.

$$\Delta w_{i,j} = -\eta \delta_j r_i \tag{6}$$

The above described algorithm is executed until ANN outputs for all 65536 training pairs does not be differ from a desired output of more than an allowable error. An output error is computed as a Euclidean distance in n-dimensional space according to Formula 7, where y_i is an output value of neuron and t_i is a desired output.

$$R^2 = \sum_{i=1}^{16} (y_i - t_i) \tag{7}$$

To avoid an endless training of an ANN, there are a limited number of epochs of training a network, and after executing this number of epochs the algorithm finishes.

In Figure 3, a parallel implementation of ANN training algorithm is depicted. After initializing connection weights with a small random numbers, the outer loop is executed until the ANN is trained or a number *k* is achieved, where *k* is a maximum number of epochs.

4. CONCLUSION

There are a number of papers describing performance enhancing of ANN training algorithm as applied to different architectures (CPU [7], GPU [8, 9] and FPGA[10]) and various types of training algorithms. Acceleration of parallel implementation may range within fairly broad limits because of differences in architecture and specification of hardware.

A developed parallel application ran on 2 cores has 25% better performance compared to sequential application. Increasing the number of cores does not increase performance due to rising overhead expenses.

Rising of overheads is caused by the fact that ANN training algorithm is not parallel and requires synchronizations after each ANN layer is trained. Increasing the number of neurons in each layer should decrease the influence of overheads and lead to achieving a better performance of parallel application.

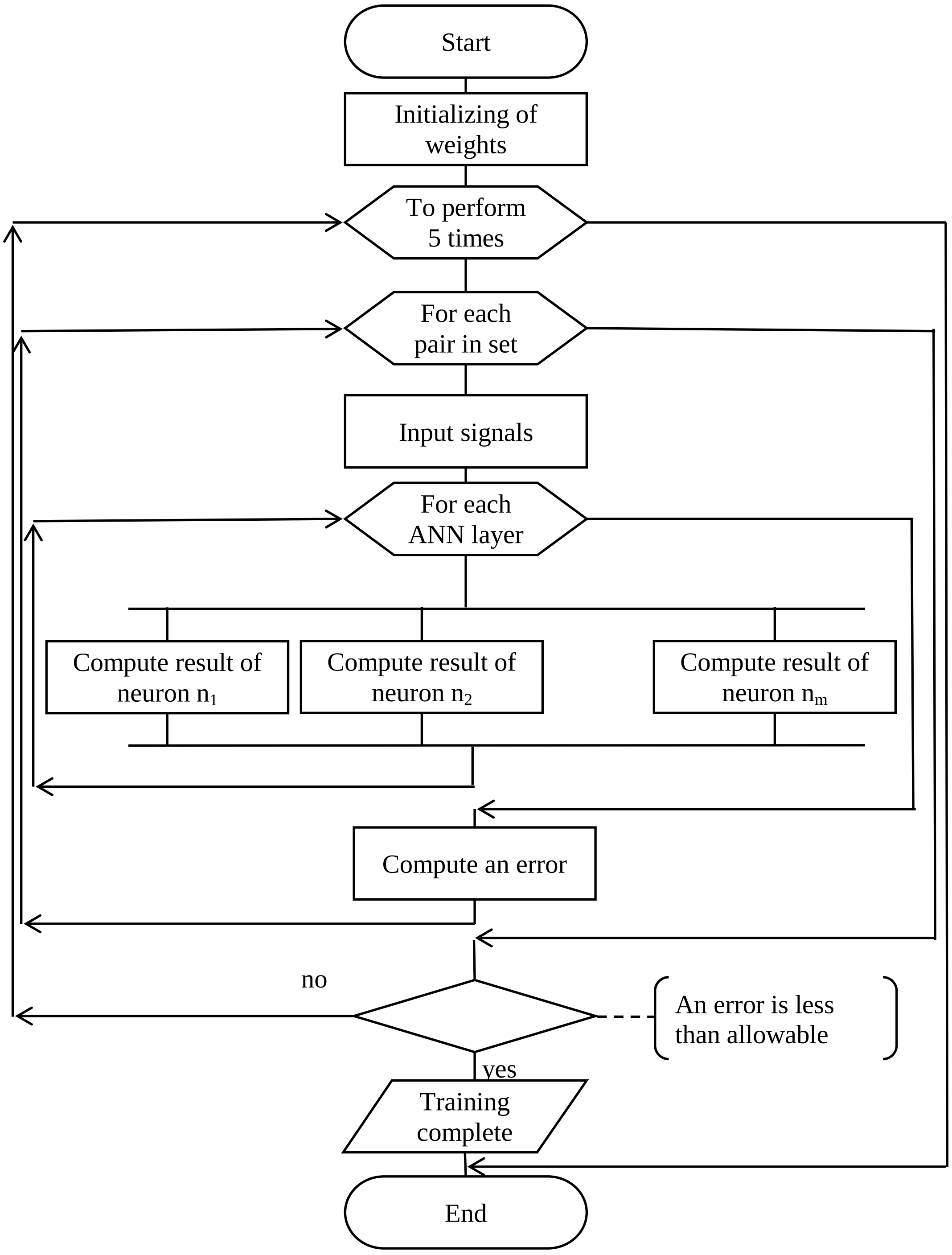

Fig. 3: Parallel implementation of ANN training algorithm

5. REFERENCES

[1] Kotlars P., Kotulski Z. On application of neural networks for S-box design, in: P.S. Szczepaniak, J.Kacprzyk, A.Niewiadomski, ed.Advances in Web Intelligence, AWIC 2005, LNCS 3528. P. 243-248. Berlin 2005.

[2] John Viega Network Security with OpenSSL. — 1-st. — O'Reilly Media, USA, June 15, 2002.

[3] L. Bossuet, G. Gogniat, and W. Burleson. Dynamically configurable Security for SRAM FPGA Bitstreams. International Journal of Embedded Systems, 2(1-2):73–85, 2006.

[4] T. Blum and C. Paar. High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. IEEE Transactions on Computers, 50(7):759–764, 2001.

[5] P. Bulens, F.X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In S. Vaudenay, editor, Proceedings of First International Conference on Cryptology in Africa – AFRICACRYPT 2008, volume 5023 of LNCS Series, pages 16–26. Springer-Verlag, 2008.

[6] Rumelhart D.E., Hinton G.E., Williams R.J., Learning Internal Representations by Error Propagation. In:

Parallel Distributed Processing, vol. 1, pp. 318—362. Cambridge, MA, MIT Press. 1986.

[7] Veselý, Karel, Burget, Lukas and Grézl, Frantisek. Parallel training of neural networks for speech recognition. ISCA, page 2934-2937, 2010.

[8] Jang, H., Park, A. & Jung, K.. Neural Network Implementation Using CUDA and OpenMP. DICTA, page 155-161. IEEE Computer Society, 2008.

[9] X. Sierra-Canto, F. Madera-Ramirez, V. Uc-Cetina. Parallel Training of a Back-Propagation Neural Network Using CUDA. ICMLA, page 307-312. IEEE Computer Society, 2010.

[10] S.T. Brassai, L. Bako, G. Pana, S. Dan. Neural control based on RBF network implemented on FPGA. OPTIM 2008, page 41-46, 2008.