Parallel Implementation of a Recursive Least Squares Neural Network Training Method on the Intel IPSC/2

James Edward Steck

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

K. Krishnamurthy
Missouri University of Science and Technology, kkrishna@mst.edu

M. Reza Ashouri

et. al. For a complete list of authors, see https://scholarsmine.mst.edu/comsci_facwork/203

PARALLEL IMPLEMENTATION OF A RECURSIVE LEAST SQUARES NEURAL NETWORK TRAINING METHOD ON THE INTEL IPSC/2

James Edward Steck, Bruce M. McMillin, K. Krishnamurthy,

M. Reza Ashouri and **Gary G. Leininger**

Intelligent Systems Center
University of Missouri-Rolla
Rolla, MO 65401

ABSTRACT

An algorithm based on the Marquardt-Levenberg least squares optimization method has been shown by S. Kollias and D. Anastasiou to be a much more efficient training method than gradient descent, when applied to some small feedforward neural networks. Yet, for many applications, the increase in computational complexity of the method outweighs any gain in learning rate obtained over current training methods. However, the least squares methods lends itself to a more efficient implementation on parallel architectures than do standard methods. This is demonstrated by comparing computation times and learning rates for the least squares method implemented on 1, 2, 4, 8, and 16 processors on an Intel iPSC/2 multicomputer. Two applications are given which demonstrate the faster real time learning rate of the least squares method over that of gradient descent.

INTRODUCTION

Much interest has been focused on the learning behavior of a type of simple connectionist model called a feedforward artificial neural network. This network is composed of several layers of neurons, the first layer bringing input from outside the network, followed by several hidden layers which receive inputs from previous layers, and an output layer. Inputs, interior signals, and outputs are real numbers, and the interior connections are weighted with real, adjustable weights. Each neuron performs a summing operation over its inputs and calculates its activation level by applying a real valued function to the sum. This activation level is output along weighted connections, becoming inputs to neurons in the following layer. The activity propagates through the network to yield an output, or a network response at the final layer. Learning involves applying some type of paradigm to adjust the connection weights in such a way that the network responds with desired outputs when presented with certain inputs.

Rumelhart, Hinton and Williams [1] present a training method for feedforward networks, originated by Werbos and based on a gradient descent minimization of the sum of the squared error at the output neurons. Basically, the network is allowed to respond to an input, and the weights are adjusted to reduce the sum of the squared error between the desired and actual output response of the network. This method is generally referred to as backpropagation, and has become a reliable and widely used training method for many artificial neural network applications. Although backpropagation can train a network to produce desired responses to given inputs, the number of times the input/output pairs (training pairs) must be presented to the network is large, and the computation time to learn a reasonable number of training pairs is lengthy.

Much effort has been spent in an attempt to improve the learning rate of the backpropagation algorithm, with notable success in several directions. Parker [2] presents a 'second order' method, as does Becker and le Cun [3]. These methods are shown to improve the first order behavior of backpropagation learning by adding second order terms to the equation which modifies the connection weights. Kollias and Anastasiou [4] present a higher order method based on a modification of the Marquardt-Levenberg least squares optimization method. This is a nonlinear least squares estimation technique which solves the least squares problem in a recursive manner to determine the network weights. This method has been shown

This work was supported in part by the National Science Foundation under Grant Numbers MIP-8909749 and CDA-8820714, by a grant from Intel Scientific Computers, and by the Missouri Department of Economic Development through the Manufacturing Research and Training Center.

to train with significantly fewer presentations of the training data. Both of these improved methods are more computationally complex than standard backpropagation. They achieve quicker learning by increasing training complexity and computation time. When implemented on a serial computer this trade-off, for most applications, means that learning using any of the methods (backpropagation, 2nd order B.P., least squares) requires approximately the same computation time. Yet neural networks are intended to be implemented in a massively parallel computing environment, with each neuron being a physical processor. In this parallel environment, which at present cannot be constructed, the complexity of the higher order methods may in fact yield faster learning.

Until massivly parallel environments are available, neural network architectures must be simulated on currently available computers. As a step in the direction of massive parallelism, the natural parallel nature of the neural network architecture can be taken to advantage by simulating it on a multiple processor machine, such as the Intel iPSC/2 multicomputer. This machine has up to 128 processors, each with local memory, operating in parallel and communicating over a cube connection architecture. Ideally, the more processors available to assign to simulating the neural network, the less time required to calculate the response and learning cycles. In reality, communication costs between processors considerably limit the ability to parallelize the simulation. Ceci, Lynn, and Gardner [5] discuss three methods for distributing feedforward network response and backpropagation learning tasks onto available processors on the iPSC/2. Two of the methods involve assigning a processor to each layer of the network, and pipelining the training pairs through the network, by processing more than one training pair simultaneously. The other method distributes the set of training pairs over several processors, with complete network simulations running on each processor. The best method of the three depends on the number of training pairs, and the number of neurons in the largest layer. In fact, for small networks, with less than 40 neurons per layer and fewer than 50 training pairs, the serial implementation is fastest.

An alternative to distributing neurons, layers, or training pairs over parallel processors is to choose a more complex training method, such as a second order method or the least squares method, which learns with fewer presentations of the training pairs. Since inter-processor communication occurs with each training cycle, and communication costs are high (time for one 8 byte communication equals the time for 53 floating point calculations on the iPSC/2), fewer training presentations may imply faster real time learning. The following sections present an implementation of the recursive least squares training algorithm on the iPSC/2. The implementation distributes the additional matrix calculations and storage of the method over available processors, giving an effective computation speed which approaches that of simple backpropagation. Since the least squares method learns with much fewer training cycles, it effectively learns at a much faster real time rate. This increase in performance is demonstrated with two examples in which a neural network is trained to simulate a physical process, an application similar to parameter estimation in adaptive control theory.

LEARNING WITH RECURSIVE LEAST SQUARES

The Marquardt-Levenberg least squares optimization technique presented by Kollias and Anastassiou is an extension of linear least squares estimation theory. It is nonlinear estimation applied to the specific estimation form of a feedforward neural network. The learning algorithm is very similar to linear recursive least squares estimation. As such, it can more easily be understood as an extension of the standard linear least squares problem.

Consider the estimation problem of finding an approximation $\hat{\phi}$ to a real valued function ϕ of n real variables $(x_1, x_2, \ldots x_n) = X$. ϕ is known only at L points in its domain, giving the "training pairs" $(\phi(X^l), X^l)$ for $l = 1, L$. The least squares problem is to find $\hat{\phi}$ such that

$$E_L = \sum_{l=1,L} \beta^{L-l} \left| \phi(X^l) - \hat{\phi}(X^l) \right|^2 \tag{1}$$

is minimized. β is a weight factor or forgetting factor allowing a higher weighting of the last training pairs. To proceed, the form of the approximation $\hat{\phi}$ is chosen to be $\hat{\phi}(X) = g(\sum w_j x_j)$, where the w_j are n parameters to be determined, and g is a fixed differentiable real valued function. The calculation of the approximate function, $\hat{\phi}$ can also be thought of as a simple neural network with one output neuron, having activation function g, and input connection weights w_j.

Minimization of equation (1) can be performed by several methods. A gradient descent minimization would give the standard delta-rule, or backpropagation learning for single layer networks. This is an iterative process and, as mentioned earlier, can take a large amount of computational time. Alternately, the set of nonlinear equations

$$0 = \frac{\partial E_L}{\partial w_j} = 2\sum_{l=1,L} \beta^{L-l}[\phi(X^l) - \hat{\phi}(X^l)]g'(Net^l)x_j^l \quad \text{for } j = 1,n \tag{2}$$

can be solved for the weights w_j, which minimize the squared error E_L for L training pairs. If the activation function g is linear, then its derivative is a constant, and the set of equations becomes a set of algebraic equations which are easily solved. The solution, however, of the general nonlinear equations would involve some type of iterative scheme which, again would be computationally lengthy.

A recursive (least squares) method of solving equations (2) can be formulated in the following manner. First assume weights w_j^{L-1} are known which minimize E_{L-1}, the error of equation (1) for L-1 training pairs. The function g is approximated by a first order Taylor series $g(Net^L) = g(Net^{L-1}) + g'(Net^{L-1})[Net^L - Net^{L-1}]$ where $Net^L = \sum_{j=1,n} w_j^L x_j^L$ and $Net^{L-1} = \sum_{j=1,n} w_j^{L-1} x_j^L$. Using the assumption above, and this approximation for g, gives

$$S_{kj}^{L-1}\,[w_j^L - w_j^{L-1}] = F_{Lk}^{L-1}\,[\phi(X^L) - g(Net^{L-1})] \quad \text{(for k = 1,n, sum over j = 1,n),} \tag{3}$$

$$\text{where} \quad S_{kj}^{L-1} = \sum_{l=1,L} (\beta^{L-l})F_{lk}^{L-1}F_{lj}^{L-1} \doteq \beta S_{kj}^{L-2} + F_{Lk}^{L-1}F_{Lj}^{L-1} \quad \text{and} \quad F_{lk}^{L-1} = g'(Net^{L-1})x_k^l. \tag{4}$$

Equation (3) can be solved by inverting the matrix S using a matrix inversion lemma, as detailed by Clarke in [6] or by Plackett in [7] for standard recursive least squares parameter estimation, giving a final weight update rule;

$$w_j^L - w_j^{L-1} = \eta K_j^L \delta^L \tag{5}$$

where η is a small learning rate, as in gradient descent, and where the vector K and the recursive update rule for the matrix P are given by

$$K_j^L = \frac{P_{jk}^{L-1}x_k^L}{\beta + (\gamma^L)^2\, x_\alpha^L P_{\alpha i}^{L-1} x_i^L}, \qquad P_{\alpha k}^L = \frac{1}{\beta}\,[P_{\alpha k}^{L-1} - (\gamma^L)^2 K_\alpha^L x_i^L P_{ik}^{L-1}] \tag{6}$$

with implied summation over α, k, i, and where $\delta^L = g'(Net^{L-1})[\phi(X^L) - g(Net^{L-1})]$ and $\gamma^L = g'(Net^{L-1})$ This recursion is initiallised by setting P equal to the identity matrix multiplied by a large constant (10^3).

This recursive least squares training algorithm can be extended to train the weights of a neural network with hidden layers and multiple outputs, as detailed by Kollias and Anastassiou [4]. The update equations for the input weights to a neuron are those of equations (5) and (6), where δ^L becomes the back-propagated error for the neuron, and γ^L is a squared backpropagation of an error of unity at the output layer. Each neuron has associated with it a P matrix and a K vector with dimensions equal to the number of inputs coming into the neuron. The summations in the expressions for P and K above are taken over the number of inputs, with the X vector being the activation values of the neurons connected as inputs to the neuron being trained. Training in begun by initializing the P matrix, the training pairs are presented, and the weights updated with the presentation of each pair. After one pass through the training set, called a training cycle, another pass is begun, using the P matricies from the first pass. This is repeated until the error at the output is within desirable bounds.

PARALLEL IMPLEMENTATION

Recursive least squares, while able to train with considerably fewer training cycles than gradient descent, is much more computationally intense. For a single update of network weights, RLS (Recursive Least Squares) requires the exact same calculation of response and back-propagated errors as gradient descent. In addition, RLS requires the calculation and storage of a P matrix for each neuron in the network. These matrix calculations increase the compution time for RLS by a factor of 10 to 20, over that of Gradient Descent, as shown by the CPU times for a single processor in Figure 1. Yet, RLS reduces training presentations by a factor of 10 to 20; a tradeoff which yields approximately the same real time learning rate as Gradient Descent, for serial implementations of both algorithms.

Since the calculation of the P matrices, one for each neuron, are computationally intense and can be done independently, this suggests distributing the computational load over several parallel processors. The response and error backpropagation are performed concurrently on one processor. As more processors are applied to the matrix calculation task, the computation time of the RLS algorithm approaches that of simple gradient descent. Communication costs are kept to a minimum since only one communication per neuron per training pair is required, that being the information required to perform the required matrix calculations. A comparison of a serial implementation of Gradient Descent with a parallel implementation of the RLS training algorthm is shown in Figure 1. Shown are times for RLS implemented on 1 (essentially a serial version) through 16 processors on the iPSC/2, and the time for Gradient Descent implemented serially on one processor of the iPSC/2. The computation times are the elapsed response and weight update time for processing one training pair, for a network with 14 input neurons, two hidden layers of 53 and 40 neurons, and 4 output neurons, for a total of 111 neurons. Each layer is fully interconnected with the layer above and below it, except for the final neuron in each layer which is a bias neuron. The activation function used is, $g(x) = -1 + 2/(1 + e^{-x})$, a sigmoid function from the interval (-1,1) to the real line. As the matrix calculations of the RLS are distributed over an increasing number of processors, the training computation time approaches that of gradient descent. With 16 concurrent processors, the time for RLS training is less than twice that of simple gradient Descent. Since RLS trains with a factor of 10 to 20 fewer training pairs, this yields an effective decrease in real training time by a factor of 5 to 10. This decrease in real training time is demonstrated for two examples, in the following section.

SYSTEM ESTIMATION APPLICATIONS

The recursive least squares training method is applied to two problems in system identification, or estimation, from the theory of digital control. A general physical system can be represented as a functional relationship between time varying inputs and outputs. The approximate system output at time t is some function of the input at time t, as well as delayed samples of the input and output at times t-1, ... t-n. Estimation involves determining an approximation to the unknown relationship between the delayed samples and the current system output. Feedforward neural nets, trained with the RLS algorithm, are well suited to this estimation task, as demonstrated by two examples. The first is the estimation of the forces on a rapidly pitching airfoil, or aircraft wing section. The second is the representation of the rotary motion of a single link robotic arm.

The forces due to air flowing past an airfoil are lift, in the vertical direction, drag in the horizontal direction, and a pitching moment. These forces are a function of the angle between the airfoil and the direction of the air velocity, called the angle of attack. This system is modeled by a neural estimator with inputs as angle of attack at times t, t-1, t-2, and t-3 and drag, lift, and pitching moment at times t-1,t-2, and t-3. The outputs are the forces and moments at time t. The network has 14 input neurons, (one is a bias neuron) two hidden layers with 53 and 40 neurons, respectively, and 3 output neurons. The hidden layers use the sigmoid activation, the outputs use a linear activation function. Training pairs were generated for a sinusoidal pitching motion of the airfoil from -6 degrees to + 14 degrees, from experimental data presented by McCroskey [8]. Eighteen training pairs, constituting one training cycle, were used.

A comparison of RLS and gradient descent training for this example is shown in Figure 2. Gradient descent is shown for learning rates of $\eta = .001$, and, $\eta = .0025$. For the smaller learning rate, gradient descent trained to an acceptable error level in 600 training cycles, but for the larger learning rate, an acceptable error was never achieved; this was also true for learning rates greater that .0025 . The recursive least squares algorithm is shown for learning rates of .01, and .05, with training to acceptable error levels in less than 30 training cycles. The same results are shown in Figure 3 plotted as a function of elapsed processing time on the iPSC/2 hypercube. The RLS algorithm was run using 16 processors concurrently. The gradient descent algorithm was measured as elapsed time on a single processor. This is a valid comparison since any decrease in computation time of the gradient descent algorithm due to a parallel implementation would yield a similar decrease for the RLS algorithm, since the RLS algorithm must perform the gradient descent calculations.

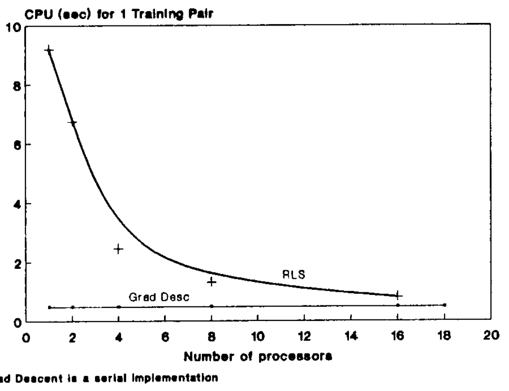


Figure 1. Execution time as a function of number of iPSC/2 processors for 111 neurons, 2778 interconnects.

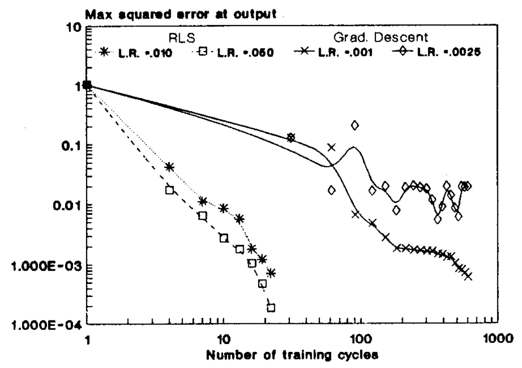


Figure 2. Comparison of RLS and gradient descent with respect to number of training cycles, aerodynamics application, RLS beta=.975 .

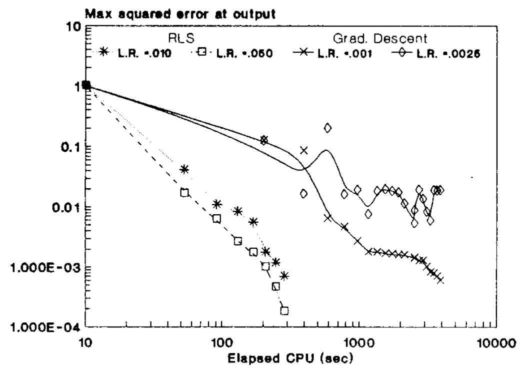
The second application uses a neural network to estimate the response of a rotary robot arm to an input drive motor signal. Joint friction effects are included and the arm drive motor has upper and lower limits on the torque which it can apply. Training pairs are generated by simulating the robot arm by solving the equation

$$J\ddot{\theta} + B(\theta,\dot{\theta}) = T, \tag{7}$$

where J is the mass moment of inertia of the arm and payload, B is a friction model, T is the limited applied torque, and θ, the angle of the arm, is the 'output' of the system. The inputs to the network are unlimited motor torque signal at t, t-1, t-2, and θ, and, $\dot{\theta}$ at t-1,t-2. The outputs are estimites of θ, and, $\dot{\theta}$ at time t. Since the input is the unlimited torque signal, and the output is trained to be the system response with a motor torque limit, the network must model the limit on the motor torque, as well as the friction and system dynamics. The network has 8 input neurons (one is a bias neuron), two hidden layers with 43 and 36 neurons, respectively, and 2 output neurons, with sigmoid activation functions in the hidden neurons. Training pairs were generated for motion of the arm from rest at 0 degrees, to 60 degrees, slowing to a stop and reversing direction, for a total of 53 training pairs. The applied torque signal exceeded the maximum motor torque for the first 9 training pairs. A comparison of RLS and gradient descent training for this example is shown in Figure 4. Gradient descent is shown for learning rates of $\eta = .001$, and, $\eta = .0025$. Higher learning rates than this produce nonconvergent results. Recursive least squares is shown for learning rates of $\eta = .025$, and, $\eta = .05$.

CONCLUSIONS

The recursive least squares method, when implemented on a parallel architecture, trains at a significantly faster rate than gradient descent for the applications shown. Faster learning yet could be accomplished using current methods to implement gradient descent on a parallel machine, since the recursive least squares method is essentially gradient descent with additional matrix calculations. A major factor in obtaining the rapid learning shown in the examples was determining the optimal learning rate and



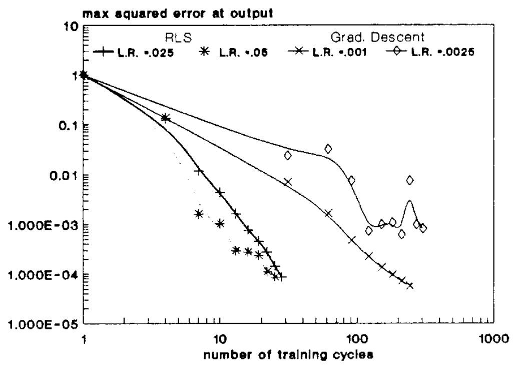


Figure 3. Comparison of RLS and gradient descent with respect to elapsed CPU time for 16 processors, aerodynamics application.

Figure 4. Comparison of RLS and gradient descent with respect to number of training cycles, robotics application, RLS beta = .997 .

forgetting factor from a series of runs in which these parameters were varied. This can be time consuming yet, considering the significant increase in learning speed, well worth the effort. Kollias and Anastassiou discussed a method for varying the learning rate dynamically during training to obtain optimal learning, and included a momentum term similar to the idea of momentum for gradient descent. Neither of these were implemented in the recursive least squares method explored here, but are the next obvious step in applying the method to further applications. With regard to the forgetting factor, much theory and experience exists for linear recursive least squares which gives various methods to choose this parameter and to vary it in an optimal manner during learning. These methods could be investigated to improve upon the results presented here.

BIBLIOGRAPHY

1. Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation", in Parallel Distributed Processing, Vol.1, eds. D. Rumelhart J. McCleland, MIT Press, Cambridge, Ma, 1986.

2. Becker, S. and le Cun, Yann, "Improving The Convergence of Back-Propagatoin Learning With Second Order Methods", Proceeding of the 1988 Connectionist Models Summer School.

3. Parker, D., "Second Order back propagation: Implementing an Optimal O(n) Approximation to Newton's Method as an Artificial Neural Network", presented at the IEEE Conf. on Neural Information Processing Systems, Denver, CO, Nov. 1987.

4. Kollias, S. and Anastassiou, D., "An Adaptive Least Squares Algorithm for the Efficient Training of Artificial Neural Networks", IEEE Transactions on Circuits and Systems, Vol 36, Aug. 1989.

5. Ceci, L. G., Lynn, P. and Gardner P. E., "Efficient Distribution of Back-Propagation Models on Parallel Architectures", University of Colorado, Boulder Technical Report, Sept. 1988.

6. Clarke, D. W., "Introduction to Self-tuning Controllers", in Self-Tuning and Adaptive Control, eds. Harris, C. J. and Billings, S. A., p.48, Peter Peregrinus Ltd., New York, 1981.

7. Plackett, R. L., Principles of Regression Analysis, Oxford University Press, 1960.

8. McCroskey, W. J., McAlister, K. W., Carr, L. W. and Pucci, S. L., "An Experimental Study of Dynamic Stall on Advanced Airfoil Sections", Vols. 1,2,and3, NASA TM-34245, 1982.