SOLVING DIFFENTIAL EQUATIONS WITH NEURAL NETWORKS: IMPLEMENTATION ON A DSP PLATFORM

K. Valasoulis, D.I. Fotiadis, I.E. Lagaris and A. Likas

10– 2002

Preprint, no 10 – 02 / 2002

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece

SOLVING DIFFERENTIAL EQUATIONS WITH NEURAL NETWORKS: IMPLEMENTATION ON A DSP PLATFORM

K. Valasoulis, D. I. Fotiadis, I. E. Lagaris and A. Likas

Department of Computer Science, University of Ioannina
and
Biomedical Research Institute, FORTH
P.O. Box 1186 - GR 45110 Ioannina, Greece
{csst9731,fotiadis,lagaris,arly}@cs.uoi.gr

Abstract:
Artificial neural networks have been successfully employed for the solution of ordinary and partial differential equations. According to this methodology, the solution to a differential equation is written as a sum of two parts. The first part satisfies the initial/boundary conditions and contains no adjustable parameters. The second part involves a feedforward neural network (MLP) whose weights must be adjusted in order to solve the equation. A significant advantage of the above methodology is the ability of of direct hardware implementation of both the solution and the training procedure. In this work we describe the implementation of the method on a hardware platform with two digital signal processors. We address several implementation and performance issues and provide comparative results against a PC-based implementation of the method.

1. INTRODUCTION

Artificial neural networks have been recently shown that they can be successfully integrated into solution methods for differential equations, both ordinary (ODEs) and partial (PDEs) [3, 4, 6]. These solution methods rely on the function approximation capabilities of feedforward neural networks and result in the construction of a solution written in a differentiable, closed analytic form. This form (model function) employs a feedforward neural network as the basic approximation element, whose parameters (weights and biases) are adjusted to minimize an appropriate error function. The model function is expressed as the sum of two terms: the first term satisfies the initial/boundary conditions and contains no adjustable parameters. The second term involves a feedforward neural network to be trained so as to satisfy the differential equation.

The employment of a neural architecture adds many attractive features to the method:

- The solution via ANN's is a *differentiable, closed analytic form* easily used in any subsequent calculation. Most other techniques offer a discrete solution (for example predictor-corrector, or Runge-Kutta methods) or a solution of limited differentiability (for example finite elements).

- The employment of neural networks provides a solution with very good generalization properties. Comparative results with the finite element method presented in this work illustrate this point clearly.

- The required number of model parameters is far less than any other solution technique and, therefore, compact solution models are obtained with very low demand on memory space.

- The method can be realized in hardware and hence offer the opportunity to tackle in real time difficult differential equation problems arising in many engineering applications.

In this work we focus on the last issue and present details and results from the implementation of the method on a hardware platform with two digital signal processors.

2. THE SOLUTION METHOD

The proposed approach for solving differential equations using neural networks [3, 4, 6] can be described in terms of the following general differential equation definition:

$$G(\vec{x}, \Psi(\vec{x}), \nabla\Psi(\vec{x}), \nabla^2\Psi(\vec{x})) = 0, \vec{x} \in D \qquad (1)$$

subject to certain boundary conditions (B.Cs) (for instance Dirichlet and/or Neumann), where $\vec{x} = (x_1, \ldots, x_n) \in R^n$, $D \subset R^n$ denotes the definition domain and $\Psi(\vec{x})$ is the solution to be computed.

To obtain a solution to the above differential equation, the collocation method is adopted, which assumes a discretization of the domain D into a set points $\hat{D}$. The problem is then transformed into the following system of equations:

$$G(\vec{x_i}, \Psi(\vec{x_i}), \nabla\Psi(\vec{x_i}), \nabla^2\Psi(\vec{x_i})) = 0, \forall \vec{x_i} \in \hat{D} \qquad (2)$$

subject to the constraints imposed by the B.Cs.

If $\Psi_t(\vec{x}, \vec{p})$ denotes a *trial solution* with adjustable parameters $\vec{p}$, the problem is transformed to:

$$min_{\vec{p}} \sum_{\vec{x_i} \in \hat{D}} (G(\vec{x_i}, \Psi_t(\vec{x_i}, \vec{p}), \nabla\Psi(\vec{x_i}, \vec{p}), \nabla^2\Psi(\vec{x_i}, \vec{p})))^2 \qquad (3)$$

subject to the constraints imposed by the B.Cs.

In the proposed approach, the trial solution Ψ_t employs a neural network (which is a multilayer perceptron (MLP) in our implementation) and the parameters $\vec{p}$ correspond to the weights and biases of the neural architecture. The trial function $\Psi_t(\vec{x})$ can be cast in a form that by construction satisfies the BCs. This is achieved by writing it as a sum of two terms:

$$\Psi_t(\vec{x}) = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p})) \qquad (4)$$

where $N(\vec{x}, \vec{p})$ is a single-output feedforward neural network with parameters $\vec{p}$ and n input units fed with the input vector $\vec{x}$. The term $A(\vec{x})$ contains no adjustable parameters and satisfies the boundary conditions, while the second term F employs the neural network whose weights and biases are to be adjusted in order to deal with the minimization problem. In this way the problem has been reduced from the original constrained optimization problem to an unconstrained one which is much easier to handle.

In [3] a systematic way is presented to construct the trial solution, i.e. to define the functional forms of both A and F. As indicated by the experiments, the above solution method is very effective and provides in reasonable computing time accurate solutions with impressive generalization (interpolation) properties.

It must be noted that the efficient minimization of equation (3) can be considered as a procedure of training the neural network where the error corresponding to each input vector $\vec{x}_i$ is the value $G(\vec{x}_i)$ which has to become zero. Computation of this error value involves not only the network output (as is the case in conventional training) but also the derivatives of the output with respect to any of the inputs. Therefore, in computing the gradient of the error function with respect to the network weights, we need to compute not only the gradient of the network but also the gradient of the network derivatives with respect to its inputs. It has been shown [3] that derivative of a multilayer perceptron (with one hidden layer) with respect to any of its inputs is also a multilayer perceptron, thus it can be efficiently implemented in hardware.

3. DSP IMPLEMENTATION

We implemented the solution methods on the following two architectures:

1. Daytona67 Board

   This is a DSP Board with *two* processing nodes. Each node consists of a TMS320C6701 floating point DSP (167 MHz) and several memory chips [1].

2. PC Ordinary PC based on a *single* Intel PIII processor (733 MHz).

We implemented the NNs on the TMS320C6701 EVM entirely in software using the C language and the features available in the CCS Ver 1.2 [CCS]. Moreover, in order to better exploit the capabilities of the Daytona67 Board, we used some hand-optimized assembly routines for fast computation of the dot product of two vectors, the matrix-vector multiplication and the sum of two vectors.

For the time-consuming computation of scalar division, we implemented a special function instead of using the standard division operator "/". Our function is based on the combined use of the reciprocal *"RCPSP()"* call [2] with the *Newton-Raphson* method and provides a sufficiently accurate reciprocal value of the denominator. Then the outcome is multiplied with the dividend to provide the final division result. The whole procedure requires no more than 62 cycles, while the use of the standard division operator requires at least 172 cycles.

Another time-consuming procedure is the computation of the log-sigmoid activation function of each hidden layer unit:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Apart from the previously mentioned division operation, the sigmoid computation also involves computing the exponential which in its standard form "exp()" or "expf()" requires 1030 and 390 cycles respectively. We implemented a special exponential function (which consumes less cycles) based on the following formula:

$$e^{-x} = e^{-integer(x)} e^{-float(x)}$$

where *integer(x)* and *float(x)* are the integer and the decimal part of *x* respectively. In case that *integer(x)* is a non-negative number, only multiplications are required for the computation of the first term of the product. Otherwise, an additional call to the division function mentioned above is required. In order to compute $e^{-float(x)}$ the dot product of the following vectors must be computed:

$$[1, y, y^2, y^3, ..., y^{29}] \qquad y = -float(x)$$

and

$$[1, 1, \frac{1}{2!}, \frac{1}{3!}, ..., \frac{1}{29!}]$$

The first vector is constructed each time the exponential function is called and the second vector is constructed once during program initialization. The dot product above actually corresponds to the Taylor series approximation of $e^{-float(x)}$ that involves thirty terms. The above implementation of the log-sigmoid function are required 175-230 cycles.

Before implementing the neural system that solves differential equations, we considered an implementation on the Daytona67 board of a conventional MLP training method for a classification dataset with 20-dimensional patterns and four classes. The selected MLP architecture contains 20 input units, one hidden layer with ten hidden units and four output units. We used used off-line back-propagation as the training algorithm and the average number of cycles per epoch for the two implemenations (Daytona67 Board and PC) is presented in Table 1.

It is clear, from Table 1, that the training process on the Daytona67 Board requires less cycles per epoch. In addition we have also observed (using other datasets) that as the dimensionality of the input patterns increases, the advantage of the DSP board becomes more clear. This is

	Daytona67 Board	Intel PIII
Number of patterns	Required cycles per epoch	Required cycles per epoch
10	35.900	≈110.000
100	340.730	≈1.100.000
1000	3.389.030	≈11.000.000

Table 1. Cycles per epoch of MLP training for a classification problem.

mainly due to the employement of the previously mentioned hand-optimized assembly routines (e.g. matrix-vector multiplication) of the Daytona67 Board.

3.1. ODE example

As a first implementation example we assumed the following ODE problem:

$$\frac{\partial \Psi}{\partial x} + \frac{1}{5}\Psi = e^{-\frac{x}{5}} cos(x)$$

and

$$\Psi(0) = 0, \qquad \frac{\partial}{\partial x}\Psi(0) = 0, \qquad x \in [0,2]$$

The form of the trial solution Ψ_t can be found in [3]. We used an MLP which contains *one* input unit, *one* hidden layer with *ten* hidden units and *one* output unit. We implemented the off-line back-propagation as a training algorithm and the number of execution cycles per epoch using the Daytona67 Board and the PC are presented in Table 2.

ODE Problem	Intel PIII	Daytona 67
Number of training points	Number of cycles per epoch	Number of cycles per epoch
10	≈36.650	44.702
50	≈183.250	221.502
100	≈359.170	442.502

Table 2. ODE example: Cycles per epoch for different numbers of training points.

3.2. PDE example

As a second implemetation example we assumed the following PDE problem:

$$\nabla^2\Psi(x,y) = e^{-x}(x - 2 + y^3 + 6y)$$

with

$$x, y \in [0,1],$$

and bounday conditions

$$\Psi(0,y) = y^3, \qquad \Psi(1,y) = (1+y^3)e^{-1},$$

$$\Psi(x,0) = xe^{-x}, \qquad \Psi(x,1) = e^{-x}(x+1).$$

REQUIRED CYCLES PER EPOCH		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	172.225	689.020	4.284.385
	20	333.515	1.319.400	8.231.590
	30	488.911	1.957.110	12.167.800
	40	648.705	2.580.160	16.150.922
	50	817.295	3.225.200	20.069.540

Table 3. PDE example: PC with Intel PIII using back-propagation with off-line update of network parameters as the training algorithm.

REQUIRED CYCLES PER EPOCH		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	127.279	506.688	3.181.501
	20	230.835	914.590	5.697.200
	30	331.558	1.323.960	8.299.210
	40	433.582	1.731.887	10.800.102
	50	536.047	2.140.260	13.400.854

Table 4. PDE example: Daytona67 Board using back-propagation with off-line update of network parameters as the training algorithm.

The form of the trial solution is described in [3]. We used an MLP with *two* input units, *one* hidden layer with varying number of hidden units and *one* output unit. We implemented both the off-line back-propagation and the Polak-Ribiere conjugate gradient method [5] as training algorithms. The number of execution cycles per epoch are presented in Tables 3, 4, 5 and 6.

In order to provide a clearer view of the way that the number of hidden units and the number of training points affect the performance of the method, we present in Tables 7 and 8 the value of the parameter

$$\alpha = \frac{cycles(Intel)}{cycles(Daytona67)},$$

It is clear from Tables 7 and 8 that the number of training points almost does not affect the value of α. On the other hand, the number of hidden units significantly affects the value of α. More specifically, when the hidden units increase, the Daytona67 Board becomes more effective. This is due to the increase in the input dimension of the assembly hand-optimized routines, which makes

REQUIRED CYCLES PER EPOCH		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	432.470	1.627.993	10.045.765
	20	813.696	3.093.260	21.223.282
	30	1.194.790	4.568.056	32.393.469
	40	1.577.228	6.052.381	43.260.389
	50	1.960.388	7.521.810	53.905.122

Table 5. PDE example: PC with Intel PIII using Polak-Ribiere conjugate gradient method as the training algorithm.

REQUIRED CYCLES PER EPOCH		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	397.976	1.568.006	9.758.006
	20	748.571	2.951.696	18.373.571
	30	1.099.116	4.335.416	26.989.166
	40	1.449.659	5.719.109	35.605.259
	50	1.800.173	7.102.748	44.220.773

Table 6. PDE example: Daytona67 Board using Polak-Ribiere conjugate gradient method as the training algorithm.

$\alpha = \frac{cycles(Intel)}{cycles(Daytona67)}$		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	1.353	1.359	1.346
	20	1.444	1.442	1.444
	30	1.474	1.478	1.466
	40	1.496	1.489	1.495
	50	1.524	1.506	1.497

Table 7. Training using off-line back-propagation.

them much more efficient compared to the corresponding PC routines.

It can also be observed that the performance of the Daytona67 Board decreases more (compared to the Pentium performance) when the Polak-Ribiere conjugate gradient method is used. This is because this training method contains a line search procedure [5] which uses some computations that require much more cycles at the Daytona67 Board than at the PC. Nevertheless, it must be mentioned that most of those time-consuming computations can be implemented using hand-optimized assembly routines which are commercially available. In the current implementation, we did not make use of these routines. Consequently, the Daytona67 implementation has the potential to exhibit much better performance in the case where these optimized routines will become available.

Except for this immmediate future research plan, we are also examining the possibility of implementing the method in alternative platforms which may be more effective for neural computations such as boards employing neurochips. Another issue that needs to be examined is the possibility of performing the computation of the sigmoid activation function using tables and interpolation methods. Finally, we are aiming at using the platform for the solution of differential equations appearing in real-world problems and especially in biomedical engineering applications.

$\alpha = \frac{cycles(Intel)}{cycles(Daytona67)}$		NUMBER OF TRAINING POINTS		
		25	100	625
NUMBER OF HIDDEN NEURONS	10	1.086	1.036	1.029
	20	1.087	1.047	1.155
	30	1.087	1.053	1.200
	40	1.088	1.058	1.215
	50	1.089	1.059	1.219

Table 8. Training using the Polak-Ribiere conjugate gradient method.

Acknowledgement This work is partially funded by the European Commission: PLACEBO IST-2001-35270, An Innovative Computational Platform for Solving Differential Equations of Modelling Biomedical Processes.

REFERENCES

[1] *Daytona Technical Reference*, www.spectrumsignal.com, 2000.

[2] *TMS320C600 Code Composer Studio Tutorial*, www.ti.com, 2000.

[3] I. E. Lagaris, A. Likas and D. I. Fotiadis, Artificial Neural Networks for Solving Ordinary and Partial Differential Equations, *IEEE Trans. on Neural Networks*, vol. 9, no. 5, pp. 987-1000, 1998.

[4] I. E. Lagaris, A. Likas and D. G. Papageorgiou, Neural Networks Methods for Boundary Value Problems with Irregular Boundaries, *IEEE Trans. on Neural Networks*, vol. 11, no. 5, pp. 1041-1049, 2000.

[5] R. Fletcher, *Practical methods of optimization*, second edition, John Wiley 1987.

[6] I.E. Lagaris, A. Likas and D.I. Fotiadis, Artificial Neural Network Methods in Quantum Mechanics, *Computer Physics Communications*, vol. 104, pp. 1-14, 1997.