Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 7 (19): 2812-2817, 2007
ISSN 1812-5654
© 2007 Asian Network for Scientific Information

Feedforward Neural Network for Solving Partial Differential Equations

Mohsen Hayati and Behnam Karami
Department of Electrical Engineering, Razi University, Kermanshah-67149, Iran

Abstract: In this study a new method based on neural network has been developed for solution of differential equations. A modified neural network is used to solve the Burger's equation in one-dimensional quasilinear partial differential equation. This method is generally applicable to nth order partial differential equations on a finite domain with boundary conditions. The results obtained by this method, have been compared with the exact solution and found to be in good agreement with each other and because of superior properties of neural network i.e., parallel processing thereby less computational cost, this method has advantages over conventional methods.

Key words: Feedforward neural network, partial differential equation, Burger's equation

INTRODUCTION

The dynamic of a physical system is usually determined by solving a differential equation. It normally proceeds by reducing the Differential Equation (DE) to a familiar form with known (compact) solutions using suitable coordinate transformations (Joes and Freeman, 1986; Arfken and Weber, 1995). In nontrivial cases however, finding the appropriate transformations is difficult and numerical methods are used. A number of algorithms (e.g., Runge-Kutta, finite difference, etc.) are available for calculating the solution accurately and efficiently (Press *et al*., 1986; Kunz and Lubbers, 1993). However, these methods are sequential and require much memory space and time as a result have great computational cost. Hemmessy and Patterson (Saif *et al*., 2006) have estimated that most computational complex application spent 90% of their execution time in only 10% of their code, hence a parallel approach to run an algorithm can be nontrivial. Here, we demonstrate a new method to find solutions of differential equations. Clearly we use an unsupervised feedforward Neural Network to solve Burger's equation that is the one-dimensional quasilinear parabolic partial differential equation. The Neural Network methods have been applied successfully to linear (Monterda and Saloma, 1998) and coupled DE's (Cruito *et al*., 2001), but not yet been tested with this class of differential equations. An interesting theorem that sheds some light on the capabilities of multilayer networks was proven by Kolmogorov and is described in (Lorentz, 1976). This theorem states that only continuous function of N variables can be computed using only linear summations and nonlinear but continuously increasing function of only one variable. Solving a DE implies the training of an Neural Network that calculates the solution values at any point in the solution space, including those not considered during the Neural Network training. In this sense, the unsupervised Neural Network learns to solve the differential equation analytically. Because the correct form of the differential equation solution is unknown beforehand, the convenient supervised algorithm for training the neural network can not be applied here, so we propose a new modified algorithm to train the neural network. The Neural Network is trained in an unsupervised manner using an energy (error) function that is derived from the differential equation itself and the governing boundary conditions. The proper choice of the energy function is crucial in training the neural network to learn the analytical properties of the solution. The solution of the differential equations in terms of neural network possesses several interesting features as neural networks learns to solve the differential equation analytically, its computational complexity does not increase quickly with number of sampling points, its rapid calculation of the solution values, its superior interpolation properties as compared to well-established methods such as finite elements (Joes and Freeman, 1986; Arfken and Weber, 1995) and finally implementation ability on existing specialized hardware (neuroprocessor), a fact that will result in a significant computational speedup.

MATERIALS AND METHODS

Mathematical definition: As a prelude to our method, some mathematical definitions and results are given. For a detailed exposition, the reader can refer to the books of Cichocki and Unbehauen (1993).

Corresponding Author: Mohsen Hayati, Department of Electrical Engineering, Islamic Azad University, Kermanshah Branch, Iran

1-Let $f(x) = f(x_1, x_2, ..., x_n)$ be a real scalar function of a real vector $x \in \Re^n$ ($f: \Re^n \to \Re^1$).

The derivative of function f (x) with respect to the vector x is an $1 \times n$ vector,

$$\frac{\partial f(x)}{\partial x} = \left[\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \cdots \frac{\partial f}{\partial x_n} \right]$$

Its gradient is an $n \times 1$ vector, $\nabla_x f(x) = \left(\frac{\partial f(x)}{\partial x} \right)^T$

2-For a real vector function

$$f(x) = [f_1(x) \; f_2(x) \; ... \; f_m(x)]^T$$

where, $x \in \Re^n$ ($f: \Re^n \to \Re^m$), its gradient is defined as

$\nabla F(x) = [\nabla f_1(x) \; \nabla f_2(x) \; ... \; \nabla f_m(x)]$ with

$$\nabla f_i(x) = \left[\frac{\partial f_i}{\partial x_1} \quad \frac{\partial f_i}{\partial x_2} \cdots \frac{\partial f_i}{\partial x_n} \right]^T_{(i=1,...,m)}$$

The Jacobian matrix is $J_f(x) = \frac{\partial f(x)}{\partial x} = \nabla f^T(x)$.

3-Let $f(x) = h(g(x))$, where $x \in \Re^n$, $h: \Re^r \to \Re^m$, $g: \Re^n \to \Re^r$ and $f: \Re^n \to \Re^m$. The chain rule of differentiation is $J_f(x) = J_h(g(x)) J_g(x)$ and

$\nabla f(x) = \nabla f(x) \nabla h(g(x))$.

4-Let $F = h(f(w))$ and $f(w) = [f_1(w) \; f_2(w) \; ... \; f_m(w)]^T$ Where $w \in \Re^{n \times m}$, $f: \Re^{n \times m} \to \Re^m$ and $h: \Re^m \to \Re^1$. The chain rule for the differentiation of the scalar function F with respect to the matrix w yields

$$\frac{\partial F(w)}{\partial w} = \sum_{i=1}^{m} \frac{\partial h(f(w))}{\partial f_i(w)} \frac{\partial f_i(w)}{\partial w}$$

Illustration of the method: A general structure of feedforward neural network is shown in Fig. 1. The feedforward neural networks are the most popular architectures due to their structural flexibility, good representational capabilities and availability of a large number of training algorithm (Haykin, 1999). This network consists of neurons arranged in layers which every neuron in a layer is connected to all neurons of next layer (a fully connected network). The neurons in the input layer simply store the input values. The hidden layers and output layer neurons each carry out two calculations. First they multiply all inputs and a bias (equal to a constant value of 1) by a weight and then they sum the result representing input net of a neuron 'O' and second the output of neurons 'a' calculated as a function (named activation function) of 'O'. The choice of activation function depends on application of the network but the sigmoid function is normally being used. The mathematical formulas for a neuron are:

$$o^k = W^k a^{k-1} + b^k, \; a^k = f^k(o^k), \; k = 1, 2, ..., n \quad (1)$$

Where, $a^0 = x$ is an n_0-dimensional input vector, a^k and f^k are the n_k -dimensional output and the activation function vector in the kth layer, respectively. The ω^k and b^k for $k = 1, 2, ..., n$ are:

$$w^k = \begin{pmatrix} w_{11}^k & \cdots & w_{1n_{k-1}}^k \\ \vdots & \ddots & \vdots \\ w_{n_k 1}^k & \cdots & w_{n_k n_{k-1}}^k \end{pmatrix}, \; b^k = \begin{pmatrix} b_1^k \\ \vdots \\ b_{n_k}^k \end{pmatrix}$$

We feed the network with input data that propagating through layers from input layer to output layer. This state of the network is called relax state. In the train state of the network the adjustable parameters of the networks (weights and biases) are tuned by a proper learning algorithm to minimize the energy function of the

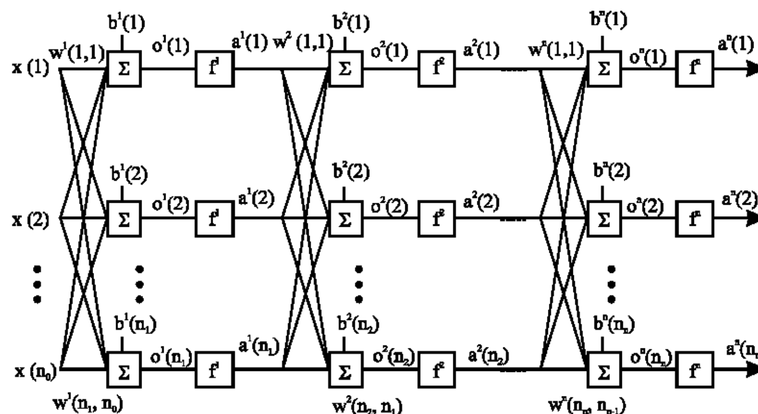

Fig. 1: An N-layer Feedforward Neural Network

network. We train the network via the Gradient-descend backpropagation method (Haykin, 1999) with the nonnegative energy function.

The Gradient-descend has a number of advantages over other optimization techniques such as Conjugate Gradient Method (CGM) and Limited-Memory quasi-Newton Algorithm (LMNA) which consider the Jacobin (JM) and the Hessian (HM) Matrix, respectively. HM contains second-order partial derivatives of the error surface with respect to the weights, while JM (or Gradient vector) contains the first-order partial derivatives. Empirical studies have shown that both HM and JM are almost rank-deficient when used in multilayer perceptron Neural Network due to the intrinsically ill-conditioned nature of the Neural Network training problem (Saarineen *et al*., 1991). Higher-order methods may not converge faster and the computations of HM and JM and their corresponding inverse are also computationally expensive. Both LMNA and CGM also involves calculation of the inverse of HM and JM, respectively. HM and JM are not always guaranteed to be non-singular and therefore their inverse may be uncomputable. According to the Gradient descend algorithm the changes of the weights and bias, Δw^k Δb^k through the minimization of the energy function with respect to the weights w^k and bias, b^k, are:

$$\Delta w^k = -\eta \frac{\partial E}{\partial w^k} \quad , \quad \Delta b^k = -\eta \frac{\partial E}{\partial b^k} \tag{2}$$

where, η is learning rate. The value of the learning rate is adaptively varied from 0 to 1 according to the following rule (Haykin, 1999):

- Increases when the gradient of E with respect to a particular weight does not change algebraic sign after several (in our case, four) consecutive iterations that enhances the search for the minimum error surface.
- Increases when the gradient of E change algebraic sign after four consecutive iteration in the search space and that prevents unnecessary oscillations.
- Otherwise, the η value remains the same. The rate is adaptively varied to account for the change of the error surface along different regions of single-weight dimension (Yam Chow, 2000; Lou, 1991). When the gradient of the error (energy), E, with respect to the linear output O^k in the kth Layer is defined as (Hagan and Menhaj, 1994).

$$\delta^k = \nabla_{o^k} E = [\delta_1^k \quad \delta_2^k \quad \ldots \; \delta_{nk}^k \quad]^T, \quad k = 1,2,\ldots,n \tag{3}$$

then

$$\frac{\partial E}{\partial w^k} = \sum_{j=1}^{n_k} \frac{\partial E}{\partial o_j^k} \frac{\partial o_j^k}{\partial w^k} = \begin{bmatrix} \delta_1^k a^{(k-1)T} \\ \delta_2^k a^{(k-1)T} \\ \vdots \\ \delta_{nk}^k a^{(k-1)T} \end{bmatrix} = \delta^k a^{(k-1)T} \tag{4}$$

$$\frac{\partial E}{\partial b^k} = (\nabla_{b^k} o^k \nabla_{o^k} E)^T = \delta^{kT}$$

where, $o_j^k = w_j^k a^{k-1} + b_j^k \quad (j = 1,2,\ldots,n_k)$ and $w^k = [w_1^{kT} w_2^{kT} \ldots w_{nk}^{kT}]^T$.

The recurrence relation of the gradient can be obtained by:

$$\begin{aligned} \delta^k &= \nabla_{o^k} E \\ \delta^k &= \nabla_{o^k} a^k \nabla_{a^k} o^{k+1} \nabla_{o^{k+1}} E \\ &= \nabla_{o^k} a^k \nabla_{a^k} o^{k+1} \delta^{k+1} \\ &= \dot{F}^k(o^k) w^{(k+1)T} \delta^{k+1} , \quad k = 1,2,\ldots,n\text{-}1 \end{aligned} \tag{5}$$

Where

$$\nabla_{a^k} o^{k+1} = \left(\frac{\partial o^{k+1}}{\partial a^k} \right)^T = w^{(k+1)T}$$

$$\dot{F}^k(o^k) = \begin{bmatrix} \dot{f}^k(o_1^k) & 0 & \ldots & 0 \\ 0 & \dot{f}^k(o_2^k) & \ldots & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \ldots & \dot{f}^k(o_{nk}^k) \end{bmatrix} = \nabla_{o^k} a^k = \left(\frac{\partial a^k}{\partial o^k} \right)^T$$

$$\dot{f}^k(o_j^k) = \frac{df^k(o_j^k)}{do_j^k} \quad , \quad j = 1,2,\ldots,n_k.$$

This recurrence relation is initialized at the final layer

$$\delta^n = \nabla_{o^n} E = \nabla_{o^n} a^n \nabla_{a^n} E \tag{6}$$

The error back propagated into the network must be reformulated as the parameters of the neural network. The exact form of the equations depends on the differential equation that is considered. In brief, we follow these steps to train the network :

Step 1: Discretizing the Domain and generate training data set.
Step 2: Initializing weights and bias.
Step 3: Presenting randomly sampled data.
Step 4: Calculating energy function with respect to present values of parameters.

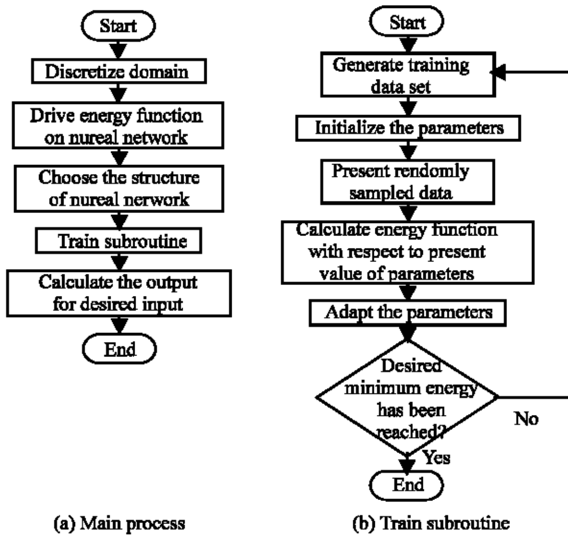

Fig. 2: The flowchart for main process and its train subroutine

Step 5: Adapting parameters.
Step 6: Repeating by going to step 3 until the desired minimum energy has been reached.
Step 7: Calculating the outputs of the network for a given point from input domain as an approximation of the solution.

The flowchart for main process and its train subroutine is shown in Fig. 2. Once the network is trained for a specific equation with its boundary and initial conditions, the solution of the equation for a desired input is obtained at just one cycle of relax state of the network.

A case study: To show the procedure, we will examine Burger's equation of the form

$$u_t + uu_x = vu_{xx} \qquad a< x <b, t > 0 \tag{7}$$

which is the one-dimensional quasilinear parabolic partial differential equation and v>0 is the coefficient of the Kinematics Viscosity of the fluid. This equation was intended as an approach to the study of turbulence, shock wave and gas dynamics. The equation is considered with the following initial and boundary conditions:

$$u(x, 0) = G(x) \qquad a < x < b$$
$$u(a, t) = g_1(t), \; u(b, t) = g_2(t), \qquad t > 0$$

We rewrite Eq. 7 as the form

$$F(x, t) = vu_{xx} - uu_x - u_t = 0 \tag{8}$$

For this PDE, a three-layer feedforward neural network is chosen, consisting of 2 inputs x_m where m = 1,2, H hidden units and one output (u). The 2 input consist of 2 independent variables $x_1 = x$ and $x_2 = n$. We use sigmoid functions for hidden layer neurons and linear function for output neuron. The accuracy of neural network solutions, depends on how close the energy function is reduces to zero during training. We choose an energy function of the form

$$E = \sum_{r=1}^{4} E_r \tag{9}$$

Where:

$$E_1 = |F(x, t)|^2, \qquad E_2 = |u(x, 0 - G(x)|^2$$
$$E_3 = |u(a, t) - g_1(t)|^2, E_4 = |u(b, t) - g_2(t)|^2$$

The energy function is chosen such that E is zero if and only if the terms in the right-hand side of the equation are all identically equal to zero. To achieve an accurate solution for u (x, t), training must reduce

$$E = \sum_{r=1}^{4} Er,$$

to a value that is as close as possible to zero. The vanishing of E_1 ensures that u (x, t) exactly satisfies equation while the reduction of each E_2, E_3 and E_4 terms to zero guarantees a unique solution for u (x, t). To reformulate this PDE with neural network, we derive equations as below:

$$a^2 = u = u(x_1, x_2) = f^2(o^2)$$

$$o^2 = \sum_{i=1}^{H} a^1(i)\, w^2(1,i) + b^2$$
$$a^1(i) = f^1(o^1(i))$$
$$o^1(i) = \sum_{j=1}^{H} x_j w^1(i,j) + b^1(i)$$

$$u = o^2 = \sum_{i=1}^{H} a^1(i) w^2(1,i) + b^2$$

$$u = \sum_{i=1}^{H} f^1(x_1 w^1(i,1) + x_2 w^1(i,2) + b^1) w^2(1,i) + b^2$$

where $f^1 = \sigma(\xi) = \dfrac{1}{1+e^{-\xi}}$

$$\frac{\partial u}{\partial x_1} = \sum_{i=1}^{H} f^{1'}(x_1 w^1(i,1) + x_2 w^1(i,2) + b^1)\, w^1(i,1) w^2(1,i) \tag{10}$$

$$\frac{\partial u}{\partial x_2} = \sum_{i=1}^{H} f^{1'}(x_1 w^1(i,1) + x_2 w^1(i,2) + b^1)\, w^1(i,2) w^2(1,i) \quad (11)$$

$$\frac{\partial^2 u}{\partial x_1^2} = \sum_{i=1}^{H} f^{1''}(x_1 w^1(i,1) + x_2 w^1(i,2) + b^1)\, w^1(i,2) w^1(i,2) w^2(1,i) \quad (12)$$

Where the w (i, j) is the weight between jth neuron of a layer and ith neuron of next layer. Our training set consists of 30×10^5 unique combination of x and t in the following range:

$$0 < x < 1, \quad 0 < t < 1$$

The trained network generalizes solution values including points those not considered during training state to the solution curves with accuracy of 10^{-12} for energy function. The accuracy can be controlled efficiently by varying the number of hidden layer neurons. By trial and error we found that the most trainable three-layer architecture is one with 40 hidden nodes (H = 40).

RESULTS AND DISCUSSION

Let consider the Burger's equation with following problem:

Problem 1: The first problem is Burger's equation with initial condition

$$u(x, 0) = \sin \pi x, \quad 0 < x < 1 \quad (13)$$

and homogeneous boundary conditions

$$u(0, t) = u(1, t) = 0, \quad t > 0 \quad (14)$$

The exact solution of the Burger's equation with initial condition, Eq. 13 and boundary conditions, Eq. 14, is obtained as:

$$u(x,t) = 2\pi v \frac{\sum_{n=1}^{\infty} a_n e^{-n^2\pi^2 vt} n \sin(n\pi x)}{a_0 + \sum_{n=1}^{\infty} a_n e^{-n^2\pi^2 vt} \cos(n\pi x)} \quad (15)$$

where a_0 and a_n for n = 1, 2,... are Fourier Coefficients and they are defined by the following equations, respectively.

$$a_0 = \int_0^1 \exp\{-(2\pi v)^{-1}[1 - \cos \pi x]\} dx$$

$$a_n = 2\int_0^1 \exp(-(2\pi v)^{-1}(1 - \cos \pi x)) \cos n\pi x dx$$

where $n \geq 1$

Problem 2: The second problem is the Burger's equation with the initial condition

$$u(x, 0) = 4x(1-x), \qquad 0 < x < 1$$

and homogeneous boundary conditions, Eq. 14 and the exact solution, Eq. 15, with

$$a_0 = \int_0^1 \exp\{-x^2(3v)^{-1}(3 - 2x)\} dx$$

$$a_n = 2\int_0^1 \exp\{-x^2(3v)^{-1}(3 - 2x)\} \cos n\pi x \, dx \quad \text{where, } n \geq 1$$

Numerical results: Table 1 and Table 2 shows the ANN solutions of problem-1 and problem-2, respectively for different values of viscosity coefficient. It is observed

Table 1: Comparison of results at different times for v =1, v = 0.1 and v = 0.01

		v = 1			v = 0.1			v = 0.01		
x	t	ANN	Numerical	Exact	ANN	Numerical	Exact	ANN	Numerical	Exact
0.25	0.4	0.01358	0.01359	0.01357	0.30898	0.31215	0.30889	0.34421	0.34819	0.34191
	0.6	0.00188	0.00189	0.00189	0.24117	0.2436	0.24074	0.27175	0.27536	0.26896
	0.8	0.00026	0.00026	0.00026	0.19605	0.19815	0.19568	0.22358	0.22752	0.22148
	1	0.00004	0.00004	0.00004	0.16297	0.16473	0.16256	0.19122	0.19375	0.18819
	3	0.00000	0.00000	0.00000	0.02745	0.02771	0.02720	0.07683	0.07754	0.07511
0.5	0.4	0.01925	0.01927	0.01924	0.57012	0.57293	0.56963	0.66214	0.66543	0.66071
	0.6	0.00265	0.00268	0.00267	0.44745	0.45088	0.44721	0.53174	0.53525	0.52942
	0.8	0.00037	0.00037	0.00037	0.35974	0.36286	0.35924	0.44137	0.44526	0.43914
	1	0.00005	0.00005	0.00005	0.29384	0.29532	0.29192	0.37876	0.38047	0.37442
	3	0.00000	0.00000	0.00000	0.04063	0.04097	0.04021	0.15194	0.15362	0.15018
0.75	0.4	0.01364	0.01365	0.01363	0.62541	0.63038	0.62544	0.91102	0.91201	0.91026
	0.6	0.00189	0.00189	0.00189	0.48802	0.49268	0.48721	0.76952	0.77132	0.76724
	0.8	0.00026	0.00026	0.00026	0.37386	0.37912	0.37392	0.65016	0.65254	0.64740
	1	0.00004	0.00004	0.00004	0.28888	0.29204	0.28747	0.55989	0.56157	0.55605
	3	0.00000	0.00000	0.00000	0.02994	0.03038	0.02977	0.22617	0.22874	0.22481

Table 2: Comparision of results at different times for v = 1, v = 0.1, v = 0.01

		v = 1			v = 0.1			v = 0.01		
x	t	ANN	Numerical	Exact	ANN	Numerical	Exact	ANN	Numerical	Exact
0.25	0.4	0.01401	0.01403	0.01400	0.31876	0.32091	0.31752	0.36415	0.36911	0.36226
	0.6	0.00196	0.00195	0.00195	0.24597	0.24910	0.24614	0.28503	0.28905	0.28204
	0.8	0.00027	0.00027	0.00027	0.19986	0.20211	0.19956	0.23101	0.23703	0.23045
	1	0.00004	0.00004	0.00004	0.16671	0.16782	0.16560	0.19782	0.20069	0.19469
	3	0.00000	0.00000	0.00000	0.02802	0.02828	0.02776	0.07756	0.07865	0.07613
0.5	0.4	0.01986	0.01988	0.01985	0.58527	0.58788	0.58454	0.68513	0.68818	0.68368
	0.6	0.00275	0.00276	0.00276	0.46036	0.46174	0.45798	0.55121	0.55425	0.54832
	0.8	0.00038	0.00038	0.00038	0.36982	0.37111	0.36740	0.45812	0.46011	0.45371
	1	0.00004	0.00005	0.00005	0.29883	0.30183	0.29834	0.38926	0.39206	0.38568
	3	0.00000	0.00000	0.00000	0.04157	0.04185	0.04107	0.15421	0.15576	0.15218
0.75	0.4	0.01408	0.01409	0.01407	0.62673	0.65054	0.62562	0.92103	0.92194	0.92050
	0.6	0.00195	0.00195	0.00195	0.50527	0.50825	0.50268	0.78472	0.78676	0.78299
	0.8	0.00027	0.00027	0.00027	0.38584	0.39068	0.38534	0.66501	0.66777	0.66272
	1	0.00004	0.00004	0.00004	0.29667	0.30057	0.29586	0.57024	0.57491	0.56932
	3	0.00000	0.00000	0.00000	0.03083	0.03106	0.03044	0.22978	0.23183	0.22774

that ANN solutions are seen to be satisfactory in agreement with the exact solutions. The ANN solutions are more accurate as compared to solution obtained with numerical method by Kutluay *et al.* (2004).

CONCLUSIONS

In this study a new method based on ANN has been applied to find solutions for Burger's equation, but this method is straightforwardly applicable to nth order partial differential equations. It is observed that ANN solutions are seen to be satisfactory in agreement with the exact solutions. The ANN solutions are more accurate as compare to solution obtained with numerical method. The neural network here allows us to obtain fast solutions of differential equations starting from randomly sampled data sets. The algorithm allows us to achieve good approximation results without wasting memory space and computational time and therefore reducing the complexity of the problem, due to parallel structure of the network. A comparison to the exact solutions shows that proposed method is able to achieve good results both in accuracy and computational complexity.

REFERENCES

Arfken, G. and H. Weber, 1995. Mathematical Methods for Physicists. 4th Edn., Academic Press, New York.

Cichocki, A. and R. Unbehauen, 1993. Neural Networks for Optimization and Signal Processing. Willy, New York.

Cruito, M., C. Monterola and C. Saloma, 2001. Solving N-body Problems with Neural Networks. Phys. Rev. Lett., pp: 4741-4744.

Hagan, M.T. and M.B. Menhaj, 1994. Training feedforward Neural Network with the Marquardt algorithm. IEEE Trans. Neural Networks, 5: 989-993.

Haykin, S., 1999. Neural Networks: A Comprehensive Foundation. 2nd Edn., Prentice-Hall, New York.

Joes, G. and I. Freeman, 1986. Theoretical Physics. Dover Publication, New York.

Kunz, K. and R. Luebbers, 1993. Finite Difference Time Domain Method for Electromagnetic. CRC Press, Boca Raton.

Kutluay, S., A. Esen and I. Dag, 2004. Numerical solution of the Burger's equation by the least-squares quadratic B-spline finite element method. J. Comput. Applied Math., 167: 21-33.

Lorentz, G.G., 1976. The 13th Problem of Hilbert, Proceeding of the symposium in pure mathemtics of the American Mathematical Society. Northern Illinois University, pp: 419-430.

Luo, Z., 1991. On the Convergence of the LMS algorithm with adaptive learning rate for linear feedforward neural networks. Neural Comput., 3: 213-225.

Monterda, C. and C. Saloma, 1998. Charactering the dynamic of constrained physical systems with unsupervised neural network. Phys. Rev., E57: 1247-1250.

Press, W., S. Flannery, S. Teakolesky and W. Vetterling, 1986. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, New York.

Saarineen, S., R. Bramley and G. Cybenko, 1991. The numerical solution of the neural network training problems. CRSD Report 1089, Center for Supercomputing Research and Development, University of Illinois, Urbana.

Saif, S., H.M. Abbas, S.M. Nassar and A.A. Wahdan, 2006. An FPGA implementation of neural optimization of block truncation coding for image/video compression. Microprocessor Microsystems, (In Press).

Yam, J. and T. Chow, 2000. A weight initialization method for improving training speed in feedforward neural network. Neurocomputing, 30: 219-232.