Available online at www.sciencedirect.com

SCIENCE DIRECT®

Comput. Methods Appl. Mech. Engrg. 194 (2005) 4481–4505

Computer methods in applied mechanics and engineering

www.elsevier.com/locate/cma

Mapping polynomial fitting into feedforward neural networks for modeling nonlinear dynamic systems and beyond

Jin-Song Pei [a,*], Joseph P. Wright [b], Andrew W. Smyth [c]

[a] School of Civil Engineering and Environmental Science, University of Oklahoma, Norman, OK 73019-1024, United States
[b] Division of Applied Science, Weidlinger Associates Inc., New York, NY 10014-3656, United States
[c] School of Engineering and Applied Science, Columbia University, New York, NY 10027-6699, United States

Received 13 January 2004

Abstract

This study presents an explicit demonstration on constructing a multilayer feedforward neural network to approximate polynomials and conduct polynomial fitting. Built on an algebraic analysis of sigmoidal activation functions rather than incremental training, this work reveals the capability of the "universal approximator" by relating the "soft computing tool" to an important class of conventional computing tools widely used in modeling nonlinear dynamic systems and many other scientific computing applications. The authors strive to enable physical interpretations and afford full control when applying the highly adaptive, powerful yet subjective neural network approach. This work is a part of the effort of bridging the gap between the black-box and mechanics-based parametric modeling.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Feedforward neural networks; Polynomial fitting; Nonlinear dynamic systems

1. Introduction

1.1. Motivation

Polynomial functions are widely used in analytical models and fitting experimental results. There is no exception in applied mechanics, where polynomials are widely adopted as basis functions in modeling nonlinearities (e.g., [25]). The authors aim to bridge the gap between the model-based interpretive and neural network-based black-box models for nonlinear dynamic systems. Thus exploring how exactly to map

* Corresponding author.

0045-7825/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.cma.2004.12.010

polynomials and polynomial fitting into neural networks has important significance and is the focus of this paper.

Presented here is a part of an ongoing study of introducing meaning into neural network inner working in approximating nonlinear functions. Such an interest is prompted by the need in the health monitoring of civil and mechanical systems (e.g., [2,1]) where variations in system identification results are often used to infer possible damages (e.g., [23]). The authors have demonstrated some preliminary results in [19–21]. If designed properly, a single-degree-of-freedom (SDOF) memoryless system can be simulated and identified more efficiently using a multilayer feedforward neural network (see Section 1.2) than using the polynomial based conventional methods [14,18]. For example, it has been shown that a neural network with very few hidden nodes can be used to approximate softening, Coulomb and clearance nonlinearities, the superior performance of which originates from the sigmoidal basis functions (see Section 1.2) and can hardly be matched by the fixed polynomial basis. More importantly, the authors have demonstrated that weights and biases of the neural network can be related to physically, mathematically or geometrically interpretable meaning through an engineered neural network initialization process. This approach can be applied in a future study to a chain-like multi-degree-of-freedom (MDOF) system based on the methodology in [13]. For a coupled MDOF system with memory, it was demonstrated in a separate study [22] how neural network based system identification could be made transparent while remaining adaptive. Nonlinear restoring forces approximated using the proposed method include both polynomial and nonpolynomial types such as Duffing, hardening, Van der Pol, softening, saturation, Coulomb, and clearance [19–21]. The strategy of mapping polynomials into neural networks, for all these cases, serves as a cornerstone for future work.

1.2. Capabilities of multilayer feedforward neural networks

Function approximation is the core of many engineering research areas such as identification, modeling and simulation. The task of function approximation is to construct a function that reproduces a given input–output relationship. The use of artificial neural networks for function approximation entails the construction of a neural network that approximates the desired functional mapping from numerical input vectors to numerical output vectors. Since each component of the output vector can be approximated separately, the problem becomes approximation of an output scalar using an input vector. Based on such an input(vector)–output(scalar) relation, this study explores how to execute "polynomial fitting" (i.e., the functional approximation with a polynomial series function using some best fit criterion) using a neural network whose architecture design is entirely guided by the nature of the function approximation task rather than some ambiguous empirical guess.

Multilayer feedforward neural networks with sigmoidal activation functions are powerful for function approximation. In fact, Cybenko [5] and Hornik et al. [8] proved that a finite linear sum of continuous sigmoidal functions can approximate any function to any desired degree of accuracy. Consequently, a feedforward neural network with one hidden layer and an arbitrary number (n_h) of hidden nodes is often called a universal approximator. Fig. 1 shows a neural network architecture consistent with the theorem, which says that for a given continuous scalar goal function $g(\mathbf{p})$ of vector $\mathbf{p}$, there is a scalar output function $z(\mathbf{p})$ in a linear summation form:

$$z(\mathbf{p}) = \sum_{j=1}^{n_h} w_{2,j} S(\mathbf{w}_{\mathbf{1,j}}\mathbf{p} - b_j) \tag{1}$$

such that $|z(\mathbf{p}) - g(\mathbf{p})| < \varepsilon$ for all $\mathbf{p}$, where ε is an arbitrarily small number. Note that S may be any continuous sigmoidal function with the property

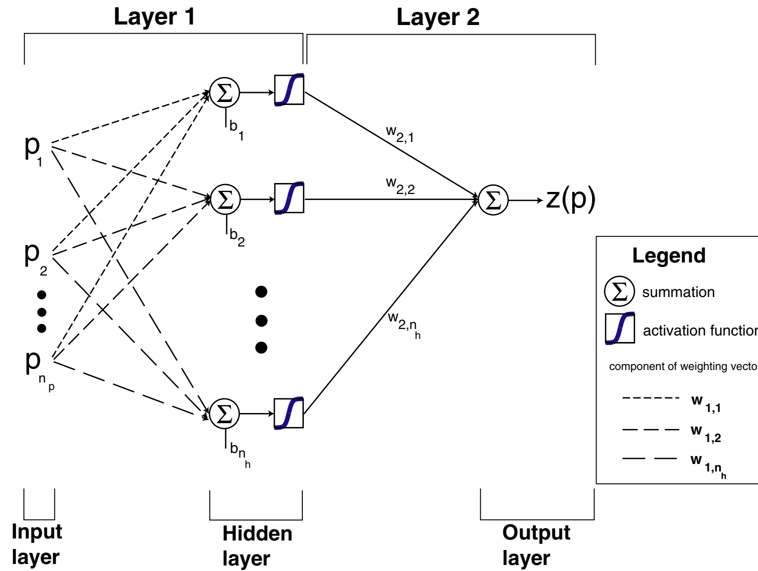

Fig. 1. Neural network architecture with one hidden layer, which is a universal approximator.

$$S(\text{variable}) \rightarrow \begin{cases} 1 & \text{variable} \rightarrow +\infty, \\ 0 & \text{variable} \rightarrow -\infty. \end{cases} \tag{2}$$

The logistic sigmoidal function, used in this study because of its popularity, is defined by

$$S(x) = \frac{1}{1+\mathrm{e}^{-x}}, \tag{3}$$

where $x = \mathbf{w_{1,j}p} - b_j$ and $j = 1, \ldots, n_h$ as in Eq. (1) and Fig. 1.

As long as the number of hidden nodes n_h can be decided upon, all the weights and biases, $\mathbf{w_{1,j}}$, $w_{2,j}$ and b_j, can be trained using a training data set, i.e., $\mathbf{p}$ and $g(\mathbf{p})$ pairs, starting with a set of initial values. However since the theorem proves existence and therefore is not ''constructive'', it does not indicate what the value of n_h should be, thus imposing a significant practical challenge. Moreover, training is often based on random initialization procedures that produce nonunique trained weights and biases, leading to difficulty when trying to interpret these quantities in physically meaningful terms. This neural network initial design problem, together with the initialization of the weights and biases, nonuniqueness and resultant interpretation are listed as the least researched issues in neural network applications [19,20]. It is exactly these understudied practical issues that have made neural networks seem like ''black boxes'' to many researchers and practicing engineers.

Contributions to illustrate the ''meaning'' of neural networks can be found in [24,12], however, they are not in the area of function approximation but pattern classification using multilayer perceptrons. Constructive approaches to function approximation are proposed in [11,10], however both of these are difficult to implement in realistic applications. As outlined in Section 1.1, the authors have attempted to give physical or mathematical meaning to neural network parameters, thereby overcoming initial design problems and avoiding nonuniqueness [19,22,20,21]. Such an effort will lead to adaptive, efficient yet transparent black-box techniques in the identification of nonlinear dynamic systems.

1.3. Mapping polynomials into neural networks for engineering applications

To relate the neural network based function approximation to conventional approaches, it is important to understand precisely how a multilayer feedforward neural network with one-hidden layer can be constructed to approximate a known function. Mhaskar [17] has studied optimal approximation of smooth and analytic functions using multilayer feedforward neural networks. This work, however, with an emphasis on the feasibility of neural networks and mathematical rigor is notationally dense and may not be readily accessible to those outside the neural network mathematics research community. Meade et al. [16,15] presented a method on how to map polynomials into a neural network architecture in a study of mechanical systems using the Duffing oscillator, where both multilayer feedforward and recurrent neural networks are considered.

Using the force-state mapping problem in engineering mechanics [14,18,3,4,25], which was originally based on either Chebyshev or ordinary polynomials, the authors pursued questions of how many hidden nodes and what the initial values of weights and biases should be in approximating typical nonlinear functions of two variables, especially displacement and velocity of single-degree-of-freedom (SDOF) memoryless nonlinear systems [19,22,20,21]. As a starting point of that study, mimicking polynomials and conducting polynomial fitting were studied and certain results will be presented in this paper. Unlike the previous work in [16,15], here the authors' approach goes beyond polynomial type nonlinearities [19,22,20,21]. To map polynomial fitting into neural networks, the work presented here relies on fixed-weight training procedures (discussed in Sections 4 and 5), which are simpler to apply than Tikhonov regularization [16].

1.4. Objectives

This paper will demonstrate constructively how a multilayer feedforward neural network can be built to approximate polynomials (both single variable terms and cross terms) and how a neural network can be designed to perform function approximation in the same way as conventional polynomial fitting.

Given the motivation stated above, this work serves the following purposes:

- This work showcases that neural networks need not be treated as "black boxes". Approximating polynomials and conducting polynomial fitting are merely examples that clarify how a "soft computing tool" (i.e., neural networks) can be made equivalent to a conventional computing tool.
- In the process of demystifying neural networks, an engineering context of nonlinear dynamics is provided with detailed derivations based on Taylor series expansion. Rigorous analysis based on function analysis is available in the literature, but the focus of this study is to give explicit demonstrations of how neural networks can be operated in an engineering context.
- This work lays a foundation for an approach where the inner workings of neural networks and interpretation of their parameters can be pursued [19–21].

2. Approximating polynomials: one-variable case

The neural network architecture shown in Fig. 1 is the center in this study because [5,8] indicates that such a neural network is a universal approximator as long as the value of n_h (number of hidden nodes) is not restricted. To approximate a scalar function $g(p)$ of a single variable p, one has the following:

$$\begin{aligned} g(p) &\approx z(p) = w_{2,1}h_1(p) + w_{2,2}h_2(p) + \cdots + w_{2,n_h}h_{n_h}(p), \\ h_j(p) &= \frac{1}{1+\mathrm{e}^{-(w_{1,j}p-b_j)}}, \quad j = 1,\ldots,n_h, \end{aligned} \tag{4}$$

and the weights and biases can be denoted as

$$\mathbf{w}_1 = [w_{1,1}, w_{1,2}, \ldots, w_{1,n_h}]^{\mathrm{T}}, \quad \mathbf{b} = [b_1, b_2, \ldots, b_{n_h}]^{\mathrm{T}}, \quad \mathbf{w}_2 = [w_{2,1}, w_{2,2}, \ldots, w_{2,n_h}].$$

When $g(p)$ is a polynomial term of p, it will be shown that all weights and biases as well as the value of n_h can be determined *by derivations*. The cases of zeroth, first, second and third power approximation will be considered, respectively, in the following subsections.

The generic format of the basis functions inside Eq. (4) is

$$h(p) = \frac{1}{1 + \mathrm{e}^{-(wp-b)}}, \tag{5}$$

where w is the weight (inside the vector $\mathbf{w}_1$) and b bias (inside the vector $\mathbf{b}$) to be derived. "Coefficients" inside the vector $\mathbf{w}_2$ are also to be derived. The Taylor series with remainder of the basis function in Eq. (5) at the origin $p = 0$ to the first, second, third and fourth power, respectively, are listed as follows:

$$h = h(p) = h_0 + h'_\xi p,$$

$$h = h(p) = h_0 + h'_0 p + \frac{1}{2!} h''_\xi p^2,$$

$$h = h(p) = h_0 + h'_0 p + \frac{1}{2!} h''_0 p^2 + \frac{1}{3!} h'''_\xi p^3,$$

$$h = h(p) = h_0 + h'_0 p + \frac{1}{2!} h''_0 p^2 + \frac{1}{3!} h'''_0 p^3 + \frac{1}{4!} h^{(4)}_\xi p^4,$$

where the uncertain number $\xi \in [0, p]$ for $p > 0$, and all the derivatives as well as their shorthand notations are listed in Table 1. Also, in the following derivation, the notation used is

$$h_j^{[i]} = \frac{1}{1 + \mathrm{e}^{-(w_{1,j}^{[i]} p - b_j^{[i]})}}, \tag{6}$$

where the superscript $[i]$ denotes the ith power of polynomial term being approximated ($i = 0, 1, 2, 3$ will be considered later), while the subscript j stands for the jth node ($j = 1, \ldots, n_h$ as before). Other subscripts follow the notation defined in Eq. (1) and Fig. 1 should be referred to.

Table 1
Derivatives of sigmoidal function

Derivative at the origin	Derivative at the uncertain number ξ	Shorthand notation
$h_0 = \frac{1}{1+\mathrm{e}^b}$		$Q_0(q) = \frac{1}{1+\mathrm{e}^q}$
$h'_0 = \frac{w\mathrm{e}^b}{(1+\mathrm{e}^b)^2}$	$h'_\xi = \frac{w\mathrm{e}^{-(w\xi-b)}}{[1+\mathrm{e}^{-(w\xi-b)}]^2}$	$Q_1(q) = \frac{\mathrm{e}^q}{(1+\mathrm{e}^q)^2}$
$h''_0 = \frac{w^2\mathrm{e}^b[-1+\mathrm{e}^b]}{[1+\mathrm{e}^b)^2}$	$h''_\xi = \frac{w^2\mathrm{e}^{-(w\xi-b)}[-1+\mathrm{e}^{-(w\xi-b)}]}{[1+\mathrm{e}^{-(w\xi-b)}]^3}$	$Q_2(q) = \frac{\mathrm{e}^q[-1+\mathrm{e}^q]}{[1+\mathrm{e}^q]^3}$
$h'''_0 = \frac{w^3\mathrm{e}^b[1-4\mathrm{e}^b+\mathrm{e}^{2b}]}{[1+\mathrm{e}^b]^4}$	$h'''_\xi = \frac{w^3\mathrm{e}^{-(w\xi-b)}[1-4\mathrm{e}^{-(w\xi-b)}+\mathrm{e}^{-2(w\xi-b)}]}{[1+\mathrm{e}^{-(w\xi-b)}]^4}$	$Q_3(q) = \frac{\mathrm{e}^q[1-4\mathrm{e}^q+\mathrm{e}^{2q}]}{[1+\mathrm{e}^q]^4}$
	$h^{(4)}_\xi = \frac{w^4\mathrm{e}^{-(w\xi-b)}[-1+11\mathrm{e}^{-(w\xi-b)}-11\mathrm{e}^{-2(w\xi-b)}+\mathrm{e}^{-3(w\xi-b)}]}{[1+\mathrm{e}^{-(w\xi-b)}]^5}$	$Q_4(q) = \frac{\mathrm{e}^q[-1+11\mathrm{e}^q-11\mathrm{e}^{2q}+\mathrm{e}^{3q}]}{[1+\mathrm{e}^q]^5}$

Before proceeding to detailed derivations, an important issue must be considered. The question is whether the input (or even the output) of the network should be normalized with respect to its maximum value before being passed through the network. The idea of normalization of input variables can be found in [11,10,3]. This data pre-processing procedure is supposed to reduce numerical problems such as ill-conditioning [3]. Also, the range of input values is not considered critical [11]. Lapedes and Farber [11] also suggested to scale the output to [0, 1] in addition to the inputs and then to re-scale the trained weights back to fit the original problem. The output may or may not need scaling, but to simplify error analysis involving a Taylor series expansion, it is useful to scale inputs to $[-1, 1]$. The following procedure will be used here when normalization is involved. New input data will be scaled according to the manner in which the training input data were scaled, and the weights obtained from training the scaled input–output pairs will be used to predict new output. Finally the predicted output data will be scaled back in the same manner in which the training output data were scaled. Note that the following derivations are carried out for the cases where the input is not normalized, while the discussion on error bounds is mainly for the normalized input case.

2.1. Forming constant term, $z \approx p^0 = 1$, using a one-hidden layer feedforward neural network

Two sigmoidal functions are chosen, $h_1^{[0]}$ and $h_2^{[0]}$, where $w_{1,1}^{[0]} = -w_{1,2}^{[0]}$ and $b_1^{[0]} = b_2^{[0]} = 0$. Then one has the following:

$$z = h_1^{[0]} + h_2^{[0]} = \frac{1}{1+\mathrm{e}^{-w_{1,1}^{[0]}p}} + \frac{1}{1+\mathrm{e}^{w_{1,1}^{[0]}p}} = \frac{\mathrm{e}^{w_{1,1}^{[0]}p}}{\left(1+\mathrm{e}^{-w_{1,1}^{[0]}p}\right)\mathrm{e}^{w_{1,1}^{[0]}p}} + \frac{1}{1+\mathrm{e}^{w_{1,1}^{[0]}p}} = 1. \tag{7}$$

This equation holds true for any value of $w_{1,1}^{[0]}$. So with zero error, one may let z mimic the zeroth power of input p where scalar $w_{1,1}^{[0]}$ can be any number. Converting the above algebraic sum into a one-hidden layer neural network, $n_h = 2$ and the weights and biases can be obtained as shown in [19] for a general case. A set of numerical values is presented in Table 2 while the estimation error is zero. This table is used throughout this derivation work for many cases.

2.2. Forming first power, $z \approx p^1$, using a one-hidden layer feedforward neural network

Again, two sigmoidal functions are chosen, $h_1^{[1]}$ and $h_2^{[1]}$, where $w_{1,1}^{[1]} = -w_{1,2}^{[1]}$ and $b_1^{[1]} = b_2^{[1]} = 0$. The Taylor series expansion of both functions at the origin $p = 0$ to the second power can be written as

$$h_1^{[1]} = \frac{1}{2} + \frac{w_{1,1}^{[1]}}{4}p + \frac{1}{2!}\left(w_{1,1}^{[1]}\right)^2 Q_2\left(-w_{1,1}^{[1]}\xi_1^{[1]}\right)p^2,$$

$$h_2^{[1]} = \frac{1}{2} - \frac{w_{1,1}^{[1]}}{4}p + \frac{1}{2!}\left(w_{1,1}^{[1]}\right)^2 Q_2\left(+w_{1,1}^{[1]}\xi_2^{[1]}\right)p^2,$$

where the shorthand notation Q_2 is defined in Table 1.

The constant term can be eliminated by taking the difference of the above two functions. For a nonzero $w_{1,1}^{[1]}$, one then has

$$z = \frac{2}{w_{1,1}^{[1]}}\left(h_1^{[1]} - h_2^{[1]}\right) = p + w_{1,1}^{[1]}\left\{Q_2\left(-w_{1,1}^{[1]}\xi_1^{[1]}\right) - Q_2\left(+w_{1,1}^{[1]}\xi_2^{[1]}\right)\right\}p^2. \tag{8}$$

One may choose z to mimic the first power of input p. Converting this algebraic sum into a one-hidden layer neural network, $n_h = 2$ and the weights and biases can be obtained as shown in [19] for a general case where

Table 2
Derived weights and biases in approximating polynomials from the zeroth to third power—specific results satisfying the error bounds in Section 2 for normalized inputs

Term	Weights in Layer 1	Biases in Layer 1	Weights in Layer 2
p^0	$\mathbf{w}_1^{[0]} = \begin{bmatrix} w_{1,1}^{[0]} \\ -w_{1,1}^{[0]} \end{bmatrix}$	$\mathbf{b}^{[0]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[0]} = [1 \quad 1]$
p^1	$\mathbf{w}_1^{[1]} = \begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix}$	$\mathbf{b}^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[1]} = [+20 \quad -20]$
p^2	$\mathbf{w}_1^{[2]} = \begin{bmatrix} 0.1 \\ -0.1 \\ 1 \\ -1 \end{bmatrix}$	$\mathbf{b}^{[2]} = \begin{bmatrix} 10 \\ 10 \\ 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[2]} = \begin{bmatrix} +2.2030466 \times 10^6 \\ +2.2030466 \times 10^6 \\ -2.0002724 \times 10^2 \\ -2.0002724 \times 10^2 \end{bmatrix}^T$
p^3	$\mathbf{w}_1^{[3]} = \begin{bmatrix} 0.01 \\ 0.01 \\ 1 \\ -1 \\ 0.001 \\ -0.001 \end{bmatrix}$	$\mathbf{b}^{[3]} = \begin{bmatrix} 10 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[3]} = \begin{bmatrix} +6.6103402 \times 10^{10} \\ +6.6103402 \times 10^{10} \\ -6.6103402 \times 10^{10} \\ -6.6103402 \times 10^{10} \\ -1.2003269 \times 10^{08} \\ +1.2003269 \times 10^{08} \end{bmatrix}^T$
p_1p_2	$\mathbf{W}_1^{[1,1]} = \begin{bmatrix} +0.1 & +0.1 \\ -0.1 & -0.1 \\ +1 & +1 \\ -1 & -1 \\ +0.1 & -0.1 \\ -0.1 & +0.1 \\ +1 & -1 \\ -1 & +1 \end{bmatrix}$	$\mathbf{b}^{[1,1]} = \begin{bmatrix} 10 \\ 10 \\ 0 \\ 0 \\ 10 \\ 10 \\ 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[1,1]} = \begin{bmatrix} 5.5076165 \times 10^5 \\ 5.5076165 \times 10^5 \\ -50.006810 \\ -50.006810 \\ -5.5076165 \times 10^5 \\ -5.5076165 \times 10^5 \\ 50.006810 \\ 50.006810 \end{bmatrix}^T$
$p_1^2p_2$	$\mathbf{W}_1^{[2,1]} = \begin{bmatrix} +0.01 & +0.01 \\ +0.01 & +0.01 \\ +1 & +1 \\ -1 & -1 \\ +0.001 & +0.001 \\ -0.001 & -0.001 \\ +0.01 & -0.01 \\ +0.01 & -0.01 \\ +1 & -1 \\ -1 & +1 \\ +0.001 & -0.001 \\ -0.001 & +0.001 \\ 0 & +0.01 \\ 0 & +0.01 \\ 0 & +1 \\ 0 & -1 \\ 0 & +0.001 \\ 0 & -0.001 \end{bmatrix}$	$\mathbf{b}^{[2,1]} = \begin{bmatrix} +10 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \\ +10 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \\ +10 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\mathbf{w}_2^{[2,1]} = \begin{bmatrix} +1.1017234 \times 10^{10} \\ +1.1017234 \times 10^{10} \\ -1.1017234 \times 10^{10} \\ -1.1017234 \times 10^{10} \\ -2.0005449 \times 10^{08} \\ +2.0005449 \times 10^{08} \\ -1.1017234 \times 10^{10} \\ -1.1017234 \times 10^{10} \\ +1.1017234 \times 10^{10} \\ +1.1017234 \times 10^{10} \\ +2.0005449 \times 10^{08} \\ -2.0005449 \times 10^{08} \\ -2.2034467 \times 10^{10} \\ -2.2034467 \times 10^{10} \\ +2.2034467 \times 10^{10} \\ +2.2034467 \times 10^{10} \\ +4.0010898 \times 10^{08} \\ -4.0010898 \times 10^{08} \end{bmatrix}^T$

the scalar $w_{1,1}^{[1]}$ can be determined based on the required accuracy. As shown in [19], the estimation error E_1 is bounded by $U_1(w_{1,1}^{[1]}p)w_{1,1}^{[1]}p^2$ assuming $w_{1,1}^{[1]} > 0$, which means U_1 is a function of $w_{1,1}^{[1]}p$. It can be seen that the error bound is determined by values of $w_{1,1}^{[1]}$ and p. In other words, for a given error bound, the value of $w_{1,1}^{[1]}$ can be calculated. For the case in Table 2, the upper bound of the error is 0.25% [19]. Note that this

approximation scheme for the first power of input p differs from what is in [16,15], where the partial derivatives of the sigmoidal function with respect to the weights (rather than the inputs) are utilized. These partial derivatives are then evaluated at the weights of zero values, and then the derivatives are approximated using a finite difference scheme.

2.3. Forming second power, $z \approx p^2$, using a one-hidden layer feedforward neural network

Two sigmoidal functions are chosen first, $h_1^{[2]}$ and $h_2^{[2]}$, where $w_{1,1}^{[2]} = -w_{1,2}^{[2]}$ and $b_1^{[2]} = b_2^{[2]} \neq 0$. The Taylor series expansion of both functions at the origin $p = 0$ to the third power can be written as

$$h_1^{[2]} = \frac{1}{1+e^{b_1^{[2]}}} + \frac{w_{1,1}^{[2]} e^{b_1^{[2]}}}{\left(1+e^{b_1^{[2]}}\right)^2} p + \frac{1}{2!} \frac{\left(w_{1,1}^{[2]}\right)^2 e^{b_1^{[2]}} \left[-1+e^{b_1^{[2]}}\right]}{\left[1+e^{b_1^{[2]}}\right]^3} p^2 + \frac{1}{3!}\left(w_{1,1}^{[2]}\right)^3 Q_3\left(-w_{1,1}^{[2]}\xi_1^{[2]} + b_1^{[2]}\right) p^3,$$

$$h_2^{[2]} = \frac{1}{1+e^{b_1^{[2]}}} - \frac{w_{1,1}^{[2]} e^{b_1^{[2]}}}{\left(1+e^{b_1^{[2]}}\right)^2} p + \frac{1}{2!} \frac{\left(w_{1,1}^{[2]}\right)^2 e^{b_1^{[2]}} [-1+e^{b_1^{[2]}}]}{\left[1+e^{b_1^{[2]}}\right]^3} p^2 - \frac{1}{3!}\left(-w_{1,1}^{[2]}\right)^3 Q_3\left(+w_{1,1}^{[2]}\xi_2^{[2]} + b_1^{[2]}\right) p^3,$$

where Q_3 are defined as before.

The first power term can be eliminated by taking the sum of the above two functions:

$$\begin{aligned} h_1^{[2]} + h_2^{[2]} &= \frac{2}{1+e^{b_1^{[2]}}} + \frac{\left(w_{1,1}^{[2]}\right)^2 e^{b_1^{[2]}} \left[-1+e^{b_1^{[2]}}\right]}{\left[1+e^{b_1^{[2]}}\right]^3} p^2 \\ &\quad + \frac{1}{3!}\left(w_{1,1}^{[2]}\right)^3 \left\{Q_3\left(-w_{1,1}^{[2]}\xi_1^{[2]} + b_1^{[2]}\right) + Q_3\left(+w_{1,1}^{[2]}\xi_2^{[2]} + b_1^{[2]}\right)\right\} p^3. \end{aligned} \tag{9}$$

Two extra sigmoidal functions are chosen, $h_3^{[2]}$, $h_4^{[2]}$, where $w_{1,3}^{[2]} = -w_{1,4}^{[2]}$ and $b_3^{[2]} = b_4^{[2]} = 0$. Based on the result of approximating term p^0, $\frac{2}{1+e^{b_1^{[2]}}}(h_3^{[2]} + h_4^{[2]}) = \frac{2}{1+e^{b_1^{[2]}}}$. Then the constant term in Eq. (9) can be eliminated. For a nonzero $w_{1,1}^{[2]}$ and $b_1^{[2]}$, one has the following:

$$\begin{aligned} z &= \frac{\left[1+e^{b_1^{[2]}}\right]^3}{\left(w_{1,1}^{[2]}\right)^2 e^{b_1^{[2]}} \left[-1+e^{b_1^{[2]}}\right]} \left[h_1^{[2]} + h_2^{[2]} - \frac{2}{1+e^{b_1^{[2]}}}\left(h_3^{[2]} + h_4^{[2]}\right)\right] \\ &= p^2 + \frac{1}{3!} w_{1,1}^{[2]} \frac{1}{Q_2\left(b_1^{[2]}\right)} \left\{Q_3\left(-w_{1,1}^{[2]}\xi_1^{[2]} + b_1^{[2]}\right) + Q_3\left(+w_{1,1}^{[2]}\xi_2^{[2]} + b_1^{[2]}\right)\right\} p^3. \end{aligned} \tag{10}$$

One may choose z to mimic the second power of input p. Converting this algebraic sum into a one-hidden layer neural network, $n_h = 4$ and the weights and biases can be obtained as shown in [19] for a general case where the scalar $w_{1,3}^{[2]}$ is an arbitrary number, scalars $w_{1,1}^{[2]}$ and $b_1^{[2]}$ can be determined based on the required accuracy. As shown in [19], the estimation error E_2 is bounded by $U_2(w_{1,1}^{[2]}p, b_1^{[2]})w_{1,1}^{[2]}p^3$, which means U_2 is a function of $w_{1,1}^{[2]}p$ and $b_1^{[2]}$. It can be seen that the error bound is determined by values of $w_{1,1}^{[2]}$, $b_1^{[2]}$ and p. In other words, for a given error bound, the value of $w_{1,1}^{[2]}$ and $b_1^{[2]}$ can be calculated. For the case in Table 2, the upper bound of the error is 3.35% [19].

2.4. Forming third power, $z \approx p^3$, using a one-hidden layer feedforward neural network

Two sigmoidal functions are chosen, $h_1^{[3]}$ and $h_2^{[3]}$, where $w_{1,1}^{[3]} = +w_{1,2}^{[3]}$ and $b_1^{[3]} = -b_2^{[3]} \neq 0$. The Taylor series expansion of both functions at the origin $p = 0$ to the fourth power can be written as

$$h_1^{[3]} = \frac{1}{1+e^{b_1^{[3]}}} + \frac{w_{1,1}^{[3]} e^{b_1^{[3]}}}{\left(1+e^{b_1^{[3]}}\right)^2} p + \frac{1}{2!} \frac{\left(w_{1,1}^{[3]}\right)^2 e^{b_1^{[3]}} \left[-1+e^{b_1^{[3]}}\right]}{\left[1+e^{b_1^{[3]}}\right]^3} p^2 + \frac{1}{3!} \frac{\left(w_{1,1}^{[3]}\right)^3 e^{b_1^{[3]}} \left[1-4e^{b_1^{[3]}}+e^{2b_1^{[3]}}\right]}{\left[1+e^{b_1^{[3]}}\right]^4} p^3 + \frac{1}{4!}\left(w_{1,1}^{[3]}\right)^4 Q_4\left(-w_{1,1}^{[3]}\xi_1^{[3]} + b_1^{[3]}\right) p^4,$$

$$h_2^{[3]} = \frac{1}{1+e^{-b_1^{[3]}}} + \frac{w_{1,1}^{[3]} e^{-b_1^{[3]}}}{\left(1+e^{-b_1^{[3]}}\right)^2} p + \frac{1}{2!} \frac{\left(w_{1,1}^{[3]}\right)^2 e^{-b_1^{[3]}} \left[-1+e^{-b_1^{[3]}}\right]}{\left[1+e^{-b_1^{[3]}}\right]^3} p^2 + \frac{1}{3!} \frac{\left(w_{1,1}^{[3]}\right)^3 e^{-b_1^{[3]}} \left[1-4e^{-b_1^{[3]}}+e^{-2b_1^{[3]}}\right]}{\left[1+e^{-b_1^{[3]}}\right]^4} p^3 + \frac{1}{4!}\left(w_{1,1}^{[3]}\right)^4 Q_4\left(-w_{1,1}^{[3]}\xi_2^{[3]} - b_1^{[3]}\right) p^4.$$

The second power term can be eliminated by taking the sum of the above two functions:

$$h_1^{[3]} + h_2^{[3]} = 1 + \frac{2w_{1,1}^{[3]} e^{b_1^{[3]}}}{\left(1+e^{b_1^{[3]}}\right)^2} p + \frac{1}{3} \frac{\left(w_{1,1}^{[3]}\right)^3 e^{b_1^{[3]}} \left[1-4e^{b_1^{[3]}}+e^{2b_1^{[3]}}\right]}{\left[1+e^{b_1^{[3]}}\right]^4} p^3 + \frac{1}{4!}\left(w_{1,1}^{[3]}\right)^4 \left\{Q_4\left(-w_{1,1}^{[3]}\xi_1^{[3]} + b_1^{[3]}\right) + Q_4\left(-w_{1,1}^{[3]}\xi_2^{[3]} - b_1^{[3]}\right)\right\} p^4. \tag{11}$$

Another two sigmoidal functions are chosen, $h_3^{[3]}$ and $h_4^{[3]}$, where $w_{1,3}^{[3]} = -w_{1,4}^{[3]}$ and $b_3^{[3]} = b_4^{[3]} = 0$. Based on the result of approximating term p^0, $h_3^{[3]} + h_4^{[3]} = 1$.

Two extra sigmoidal functions are chosen, $h_5^{[3]}$ and $h_6^{[3]}$, where $w_{1,5}^{[3]} = -w_{1,6}^{[3]}$ and $b_5^{[3]} = b_6^{[3]} = 0$. As derived in Eq. (8) when approximating term p^1, the difference of the two terms can be used to approximate p^1.

Thus, the constant and first power terms can be eliminated from Eq. (11). For a nonzero $w_{1,1}^{[3]}$ and $b_1^{[3]}$, the following can be obtained:

$$z = \frac{3\left[1+e^{b_1^{[3]}}\right]^{[4]}}{\left(w_{1,1}^{[3]}\right)^3 e^{b_1^{[3]}} \left[1-4e^{b_1^{[3]}}+e^{2b_1^{[3]}}\right]} \left[h_1^{[3]} + h_2^{[3]} - \left(h_3^{[3]} + h_4^{[3]}\right) - \frac{4w_{1,1}^{[3]} e^{b_1^{[3]}}}{w_{1,5}^{[3]}\left(1+e^{b_1^{[3]}}\right)^2}\left(h_5^{[3]} - h_6^{[3]}\right)\right]$$
$$= p^3 + \frac{3\left[1+e^{b_1^{[3]}}\right]^4}{\left(w_{1,1}^{[3]}\right)^3 e^{b_1^{[3]}} \left[1-4e^{b_1^{[3]}}+e^{2b_1^{[3]}}\right]} \left\{\frac{1}{4!}\left(w_{1,1}^{[3]}\right)^4 \left\{Q_4\left(-w_{1,1}^{[3]}\xi_1^{[3]} + b_1^{[3]}\right) + Q_4\left(-w_{1,1}^{[3]}\xi_2^{[3]} - b_1^{[3]}\right)\right\} p^4 - \frac{2w_{1,5}^{[3]} w_{1,1}^{[3]} e^{b_1^{[3]}}}{(1+e^{b_1^{[3]}})^2}\left\{Q_2\left(-w_{1,5}^{[3]}\xi_5^{[3]}\right) - Q_2\left(+w_{1,5}^{[3]}\xi_6^{[3]}\right)\right\} p^2\right\}. \tag{12}$$

One may choose z to mimic the third power of input p. Converting this algebraic sum into a one-hidden layer neural network, $n_h = 6$ and the weights and biases are obtained as shown in [19] for a general case where scalar $w_{1,3}^{[3]}$ is an arbitrary number, scalars $w_{1,1}^{[3]}$, $w_{1,5}^{[3]}$ and $b_1^{[3]}$ can be determined based on the required

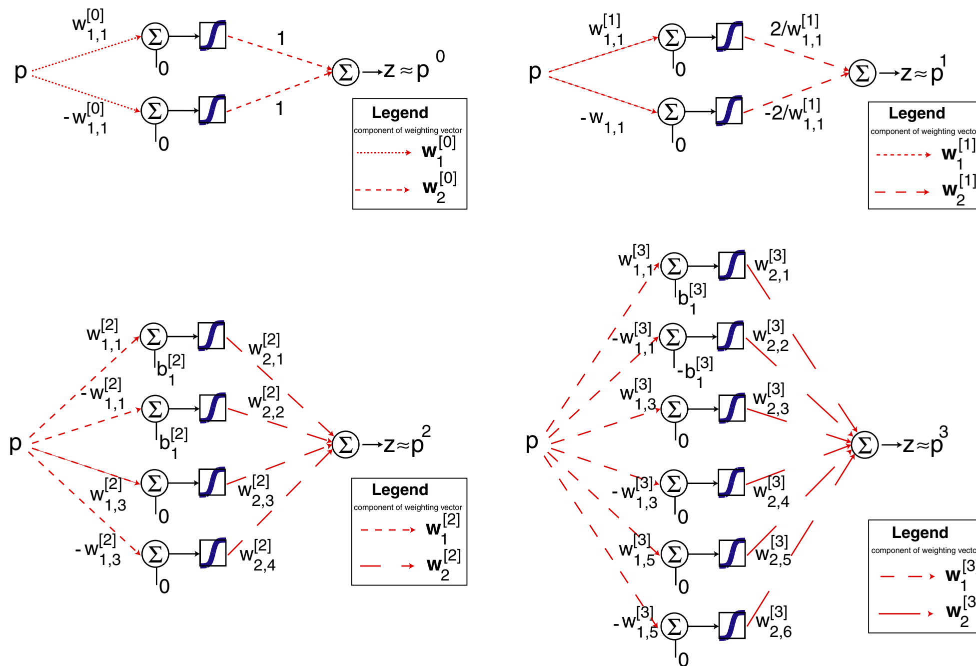

Fig. 2. Neural network architectures used in approximating zeroth (p^0), first (p^1), second (p^2), third (p^3) and powers of polynomial terms, $p \in [-1, 1]$.

accuracy. As shown in [19], the estimation error E_3 is bounded by $U_3(w^{[3]}_{1,1}, w^{[3]}_{1,5}, b^{[3]}_1, p)$, which means U_3 is a function of $w^{[3]}_{1,1}$, $w^{[3]}_{1,5}$ and $b^{[3]}_1$ and p. It can be seen that the error bound is determined by values of $w^{[3]}_{1,1}$, $w^{[3]}_{1,5}$, $b^{[3]}_1$ and p. In other words, for a given error bound, the value of $w^{[3]}_{1,1}$, $w^{[3]}_{1,5}$ and $b^{[3]}_1$ can be calculated. For the case in Table 2, the upper bound of the error is 1.75% [19]. The neural network architectures and the corresponding approximation performances are summarized in Figs. 2 and 3, respectively.

For integer powers with an order higher than 3, a similar philosophy may be applied to form the neural network architecture. Since quadratic and cubic powers cover the most commonly seen hardening-type nonlinearities (e.g., Van der Pol and Duffing oscillators), higher powers will not be pursued here. The quantitative analysis presented above has revealed the efficiency of sigmoidal functions in approximating polynomials for one-variable case. The derivations for the two-variable case as well as the philosophy for other multi-variable cases will be presented in Section 3.

3. Approximating polynomials: two-variable case

There is a practical need to fit polynomials of multi-variables. For example for force-state mapping, it is necessary to study how neural networks can approximate polynomials of two variables. As an extension of the work in Section 2, a neural network with one hidden layer will be under study, where the output of the network equals the product of two normalized inputs to the network with a specified accuracy. Power terms of one of the inputs only are easily obtained as demonstrated from the results in Section 2 because the latter can be adopted directly. The product of two inputs and the product of the first power of one input and second power of another input will be considered in this section.

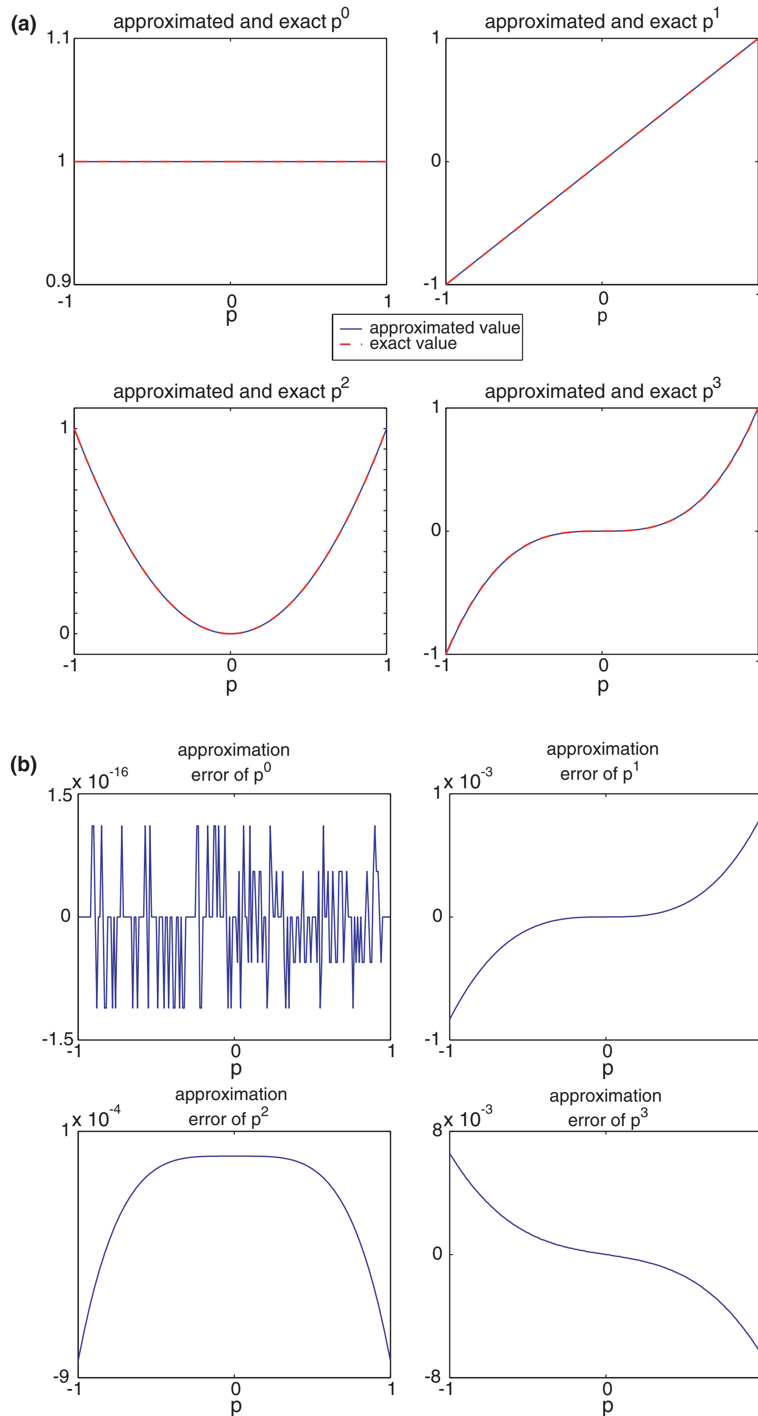

Fig. 3. (a) Comparison between the approximated and exact zeroth (p^0), first (p^1), second (p^2) and third (p^3) powers of polynomial terms for $p \in [-1, 1]$ and (b) Approximation errors of the zeroth (p^0), first (p^1), second (p^2) and third (p^3) powers of polynomial terms. Note that the orders of magnitude in Part (b) are 10^{-16}, 10^{-3}, 10^{-4} and 10^{-3}, respectively.

In the case of two inputs, any hidden node output, h, can be viewed as a function of input, p_1 and p_2. This function is expressed by

$$h(p_1, p_2) = \frac{1}{1 + e^{-(w_1 p_1 + w_2 p_2 - b)}}. \tag{13}$$

The Taylor series expansion of this bi-variable function could be used to extend the method used in Section 2, however, there is a shortcut by directly utilizing the results in the single-variable case and this will be adopted here. The key step is to use algebraic identities. Note that this methodology differs from that in [15,16]. What is presented here can be readily applied to other multi-variable case as long as the corresponding algebraic identities are adopted.

3.1. Forming $z \approx p_1 p_2$ using a one-hidden layer feedforward neural network

The algebraic identity used here is

$$p_1 p_2 = \tfrac{1}{4}[(p_1 + p_2)^2 - (p_1 - p_2)^2]. \tag{14}$$

Two intermediate variables are defined as follows:

$$\begin{aligned} u_1 &= p_1 + p_2 = [\,1 \quad 1\,] \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \bar{\mathbf{w}}_{1,1}^{[1,1]} \mathbf{p}, \\ u_2 &= p_1 - p_2 = [\,1 \quad -1\,] \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \bar{\mathbf{w}}_{1,2}^{[1,1]} \mathbf{p}, \end{aligned} \tag{15}$$

where the numbers 1 and 1 in the superscript [1, 1] denote the powers of p_1 and p_2 to be approximated, respectively. The linear transform from $\mathbf{p} = [\,p_1 \quad p_2\,]^{\mathrm{T}}$ to $\mathbf{u} = [\,u_1 \quad u_2\,]^{\mathrm{T}}$ is equivalent to a neural network operation of passing a weighted sum of input vector $\mathbf{p}$ through a linear activation function to get the output vector $\mathbf{u}$, as shown in Fig. 4(a).

Substituting Eq. (15) into (14) results in the following:

$$p_1 p_2 = \begin{bmatrix} \frac{1}{4} & -\frac{1}{4} \end{bmatrix} \begin{bmatrix} (u_1)^2 \\ (u_2)^2 \end{bmatrix} = \bar{\mathbf{w}}_2^{[1,1]} \begin{bmatrix} (u_1)^2 \\ (u_2)^2 \end{bmatrix} = \begin{bmatrix} \bar{w}_{2,1}^{[1,1]} & \bar{w}_{2,2}^{[1,1]} \end{bmatrix} \begin{bmatrix} (u_1)^2 \\ (u_2)^2 \end{bmatrix} \tag{16}$$

which means the target function, $p_1 p_2$, is a linear sum of the second power of the components of vector $\mathbf{u}$. Its equivalent neural network operation is shown in Fig. 4(c). Again, a linear activation function is adopted here.

The two neural networks shown in Fig. 4(a) and (c) can be connected if there is a certain neural network to which sending each component of vector $\mathbf{u}$ as an input, will obtain its second power as an output. The exact solution may not be provided by a neural network, however, a neural network already derived in approximating the second power can be adopted as reproduced in Fig. 4(b).

The entire neural network in Fig. 4(a)–(c) has three hidden layers. However, it can be condensed into a network with only one hidden layer as shown in Fig. 4(d). This simplification is based on the existence of two linear activation functions. Hidden nodes with sigmoidal activation functions remain unchanged while nodes with linear activation function are eliminated after their adjacent layers have been combined. New weighting matrices are the products of old weighting matrices as shown in [19]. A set of numerical values is presented in Table 2 following the derived weights and biases for the second power in the same table.

As indicated by Eq. (16), the error bound is entirely determined by the neural network second power estimator.

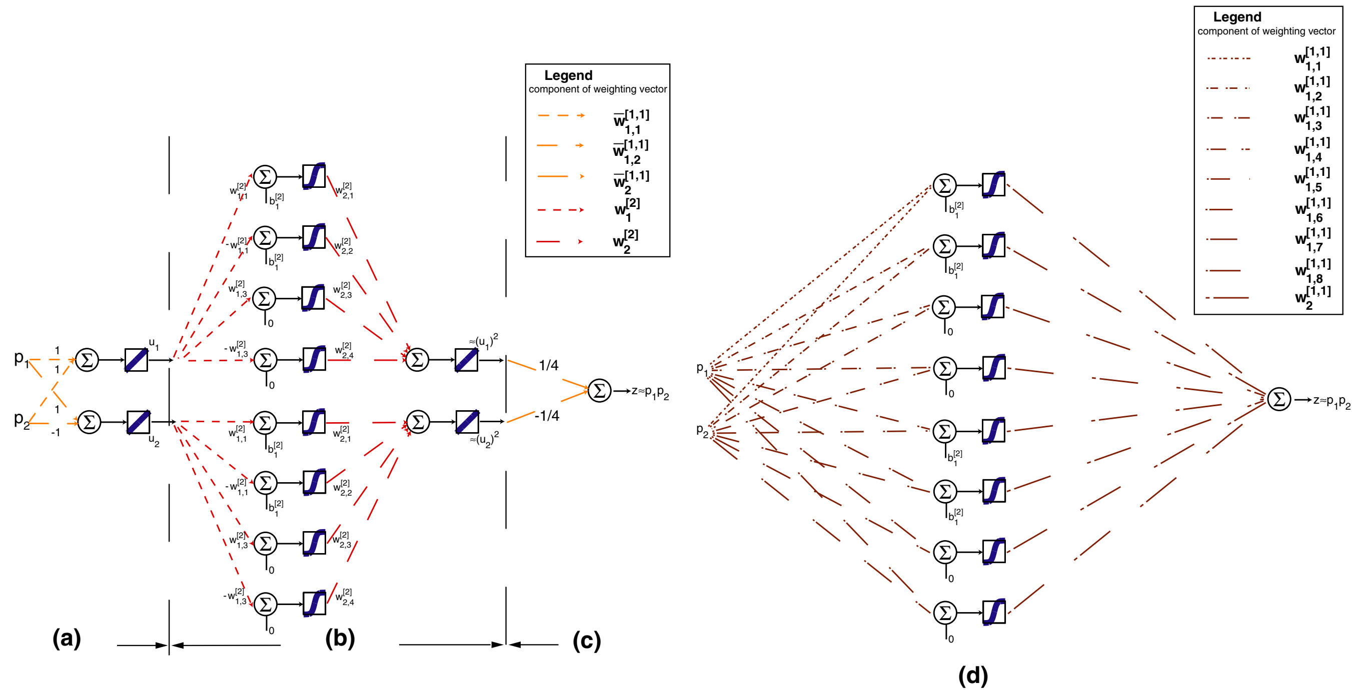

Fig. 4. Neural network architecture used in approximating term p_1p_2 with (a), (b) and (c) showing detailed substructures, and (d) showing condensed neural network architecture used in approximating term p_1p_2. $p_1 \in [-1, 1]$ and $p_2 \in [-1, 1]$.

3.2. Forming $z \approx (p_1)^2 p_2$ using one-hidden layer feedforward neural network

The algebraic identity used here is

$$(p_1)^2 p_2 = \tfrac{1}{6}(p_1+p_2)^3 - \tfrac{1}{6}(p_1-p_2)^3 - \tfrac{1}{3}(p_2)^3. \tag{17}$$

Three intermediate variables are defined as follows:

$$\begin{aligned} u_1 &= p_1 + p_2 = [1 \quad 1]\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \bar{\mathbf{w}}_{1,1}^{[2,1]}\mathbf{p}, \\ u_2 &= p_1 - p_2 = [1 \quad -1]\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \bar{\mathbf{w}}_{1,2}^{[2,1]}\mathbf{p}, \\ u_3 &= p_2 = [0 \quad 1]\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \bar{\mathbf{w}}_{1,3}^{[2,1]}\mathbf{p}, \end{aligned} \tag{18}$$

where the numbers 2 and 1 in the superscript [2, 1] denote the powers of p_1 and p_2 to be approximated, respectively. The linear transform from $\mathbf{p} = [p_1 \quad p_2]^{\mathrm{T}}$ to $\mathbf{u} = [u_1 \quad u_2 \quad u_3]^{\mathrm{T}}$ is equivalent to a neural network operation of passing the weighted sum vector of input vector $\mathbf{p}$ through a linear activation function to get the output vector $\mathbf{u}$, as shown in Fig. 5(a).

Substituting Eq. (18) into (17) results in the following:

$$(p_1)^2 p_2 = \begin{bmatrix} \frac{1}{6} & -\frac{1}{6} & -\frac{1}{3} \end{bmatrix} \begin{bmatrix} (u_1)^3 \\ (u_2)^3 \\ (u_3)^3 \end{bmatrix} = \bar{\mathbf{w}}_2^{[2,1]} \begin{bmatrix} (u_1)^3 \\ (u_2)^3 \\ (u_3)^3 \end{bmatrix} = \begin{bmatrix} \bar{w}_{2,1}^{[2,1]} & \bar{w}_{2,2}^{[2,1]} & \bar{w}_{2,3}^{[2,1]} \end{bmatrix} \begin{bmatrix} (u_1)^3 \\ (u_2)^3 \\ (u_3)^3 \end{bmatrix} \tag{19}$$

which means the target function, $(p_1)^2 p_2$, is a linear sum of the third power of the components of vector $\mathbf{u}$. Its equivalent neural network operation is shown in Fig. 5(c). Again, a linear activation function is used here.

The two neural networks shown in Fig. 5(a) and (c) can be connected if there is a certain neural network to which sending each component of vector $\mathbf{u}$, as an input, will obtain its third power as an output. The exact solution may not be provided by a neural network, however, a neural network as derived in approximating term p^3 can be used to approximate the third power, and it is reproduced here in Fig. 5(b).

The entire neural network shown in Fig. 5(a)–(c) has three hidden layers. However, it can be condensed into a network with only one hidden layer as shown in Fig. 5(d). This simplification is based on the existence of two linear activation functions. Hidden nodes with sigmoidal activation function remain unchanged while nodes with linear activation function are eliminated after their adjacent layers have been combined. New weighting matrices are the products of old weighting matrices as shown in [19]. A set of numerical values is presented in Table 2 following the derived weights and biases for the second power in the same table.

As indicated by Eq. (19), the error bound is entirely determined by the neural network third power estimator. Fig. 6 shows the comparison between the target and output function of the derived neural network, as well as the approximation error. It can be seen that the approximation is very satisfactory for both cases. The above procedure can be generalized to multi-variable cases as long as an algebraic identity can be found for the desired powers of cross terms. The efficiency of sigmoidal functions in approximating polynomials can be seen from the study on both one- and two-variable cases.

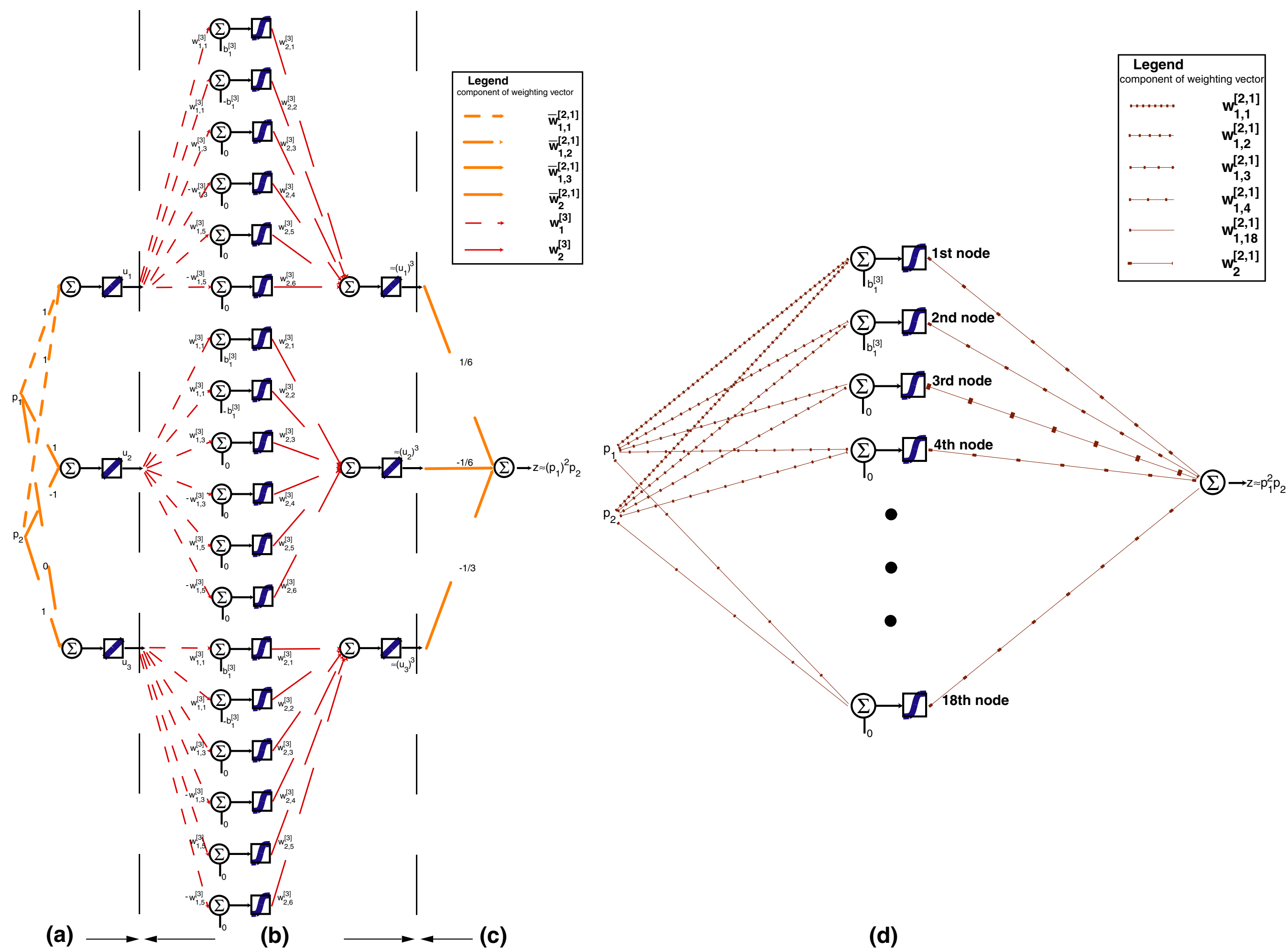

Fig. 5. Neural network architecture used in approximating term $(p_1)^2p_2$ with (a), (b) and (c) showing detailed substructures, and (d) showing condensed neural network architecture used in approximating term $(p_1)^2p_2 \cdot p_1 \in [-1, 1]$ and $p_2 \in [-1, 1]$.

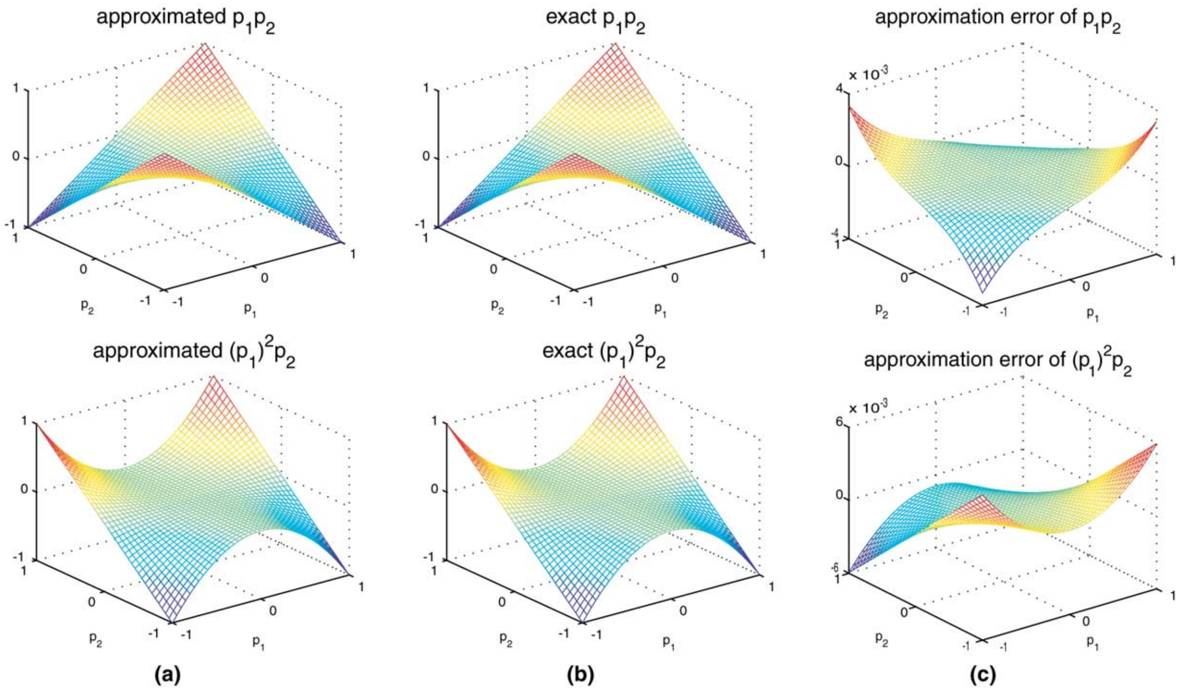

Fig. 6. Comparison between (a) the approximated and (b) the exact p_1p_2 and $(p_1)^2p_2$, and (c) the estimation errors of the both terms. Note the order of magnitude of the errors in Panel (c) is 10^{-3}. $p_1 \in [-1, 1]$ and $p_2 \in [-1, 1]$.

4. Fixed-weight training method

Sections 2 and 3 demonstrate how to approximate individual polynomial terms of input using networks with one hidden layer and the sigmoidal activation function. These individual networks with specified weights and biases can be assembled in parallel with another layer built on top to form a new neural network. This new network can be used to perform polynomial fitting of its input–target pairs. When this network is trained, not all the weights and biases need to be updated. Those weights and biases inherited from fitting each individual polynomial term are fixed throughout the training; the only weights to be trained are those related to the coefficients of polynomials. This feature distinguishes this training method from others and also explains why it is called *fixed-weight training method*, which originated from neural networks in pattern classification [9]. The fixed-weight training method within the specified architecture will lead to unique solutions as will be discussed later in Section 5.

4.1. Basic network architecture for fixed-weight training approach

To carry out polynomial fitting using the fixed-weight training method, a neural network with two hidden layers is designed. A sigmoidal function with bias is used in the first hidden layer, while a linear activation function with no bias is used for second hidden and output layer. The number of nodes in hidden layers are determined by the desired accuracy. Weights in Layers 1 (from input to the first hidden layer) and 2 (from the first to second hidden layer) are directly adopted from individual polynomial fitting and remain unchanged throughout the training. Weights in Layer 3 (from the second hidden to output layer), however, are the coefficients to be identified; these weights are the only parameters to be trained.

For the single-input–single-output case (i.e., one-variable polynomial fitting), the architecture shown in Fig. 7(a) is appropriate for polynomial fitting from the zeroth up to third power. It can be seen that the four nodes in the second hidden layer correspond to zeroth to third power of the input, respectively, while the four trained weights of Layer 3 are simply the approximations of the coefficients of the zeroth to third powers. Under this design, trained weights have very specific and unique meaningful "interpretations". The fourteen nodes in the first hidden layer and their corresponding weights and biases are carried over from the examples of approximating individual integer powers in Section 2 provided that the accuracy is acceptable. If a different degree of accuracy is required, the number of nodes as well as the values of weights and biases should be re-derived.

The neural network shown in Fig. 7(a) is not fully connected but can be made so as shown in Fig. 7(b). The weights are set to zero along the imaginary connections represented by the thin lines. With this fully connected network architecture, a weighting matrix can be used for Layer 2, which then enables the application of matrix and vector based training algorithms.

After training, the architecture shown in Fig. 7(b) can be further converted into a network with one hidden layer. This is because the second hidden layer uses a linear activation functions with no biases. The equivalent network with a total of 14 hidden nodes is shown in Fig. 7(c). Note that the weighting vector of Layer 1 remains the same while that of the new Layer 2 is the matrix product of the weights in former Layers 2 and 3. This network serves as an example of how to approximate an arbitrary polynomial function up to third power using a network with only one hidden layer. The 14 nodes adopted here are not necessarily the least number of nodes; this required number is inherited from the way individual polynomial terms are approximated and therefore, related to approximation accuracy.

The same philosophy applies to the case of double-input–single-output (i.e., two-variable polynomial fitting). Fig. 8(a) shows an architecture that is able to fit all the combinations from the zeroth up to third power, while Fig. 8(b) shows the corresponding fully connected network. The equivalent architecture after training with one hidden layer is shown in Fig. 8(c). In total, 70 nodes are used in the condensed architecture above. Note that all of these three architectures can only meet a certain approximation accuracy requirement. For higher accuracy, network approximating individual polynomial terms needs to be re-derived to meet the higher accuracy and then assembled in parallel following the manner presented here.

4.2. Training algorithm

Training neural networks is a numerical procedure. Training algorithms need to be numerically stable and converge rapidly within the length of a given input–target data set. The row weighting vector at Layer 3 is initialized as zero at the beginning of the training, i.e., $\mathbf{w}(0) = \mathbf{0}$. It will be shown in the Discussion that this set of initial values will not affect the final trained values. The fixed-weight training procedure adopted in this study can be described in the following steps and Figs. 7(b) and 8(b) can be referred to for the referred neural network architecture.

Step 1. Pass the input column vector $\mathbf{p}(t)$ through Layer 1 to obtain $\boldsymbol{\phi}_1(t)$, the column output vector of Layer 1. The dimension of $\boldsymbol{\phi}_1(t)$ equals the number of hidden nodes in the first hidden layer, n_{h_1}, while the jth component of $\boldsymbol{\phi}_1(t)$ is an individual sigmoidal result of $\mathbf{p}(t)$. Symbolically,

$$\phi_{1,j}(t) = S(\mathbf{w}_{1,j}\mathbf{p}(t) + b_j), \quad j = 1, \ldots, n_{h_1}, \tag{20}$$

where $\mathbf{w}_{i,j}$ is the row weighting vector related to jth hidden node, and b_j the bias scalar.

Step 2. Pass the column vector $\boldsymbol{\phi}_1(t)$ through Layer 2 to obtain $\boldsymbol{\phi}_2(t)$, the column output vector of Layer 2. A matrix–vector product is involved here:

$$\boldsymbol{\phi}_2(t) = \mathbf{W}_2\boldsymbol{\phi}_1(t). \tag{21}$$

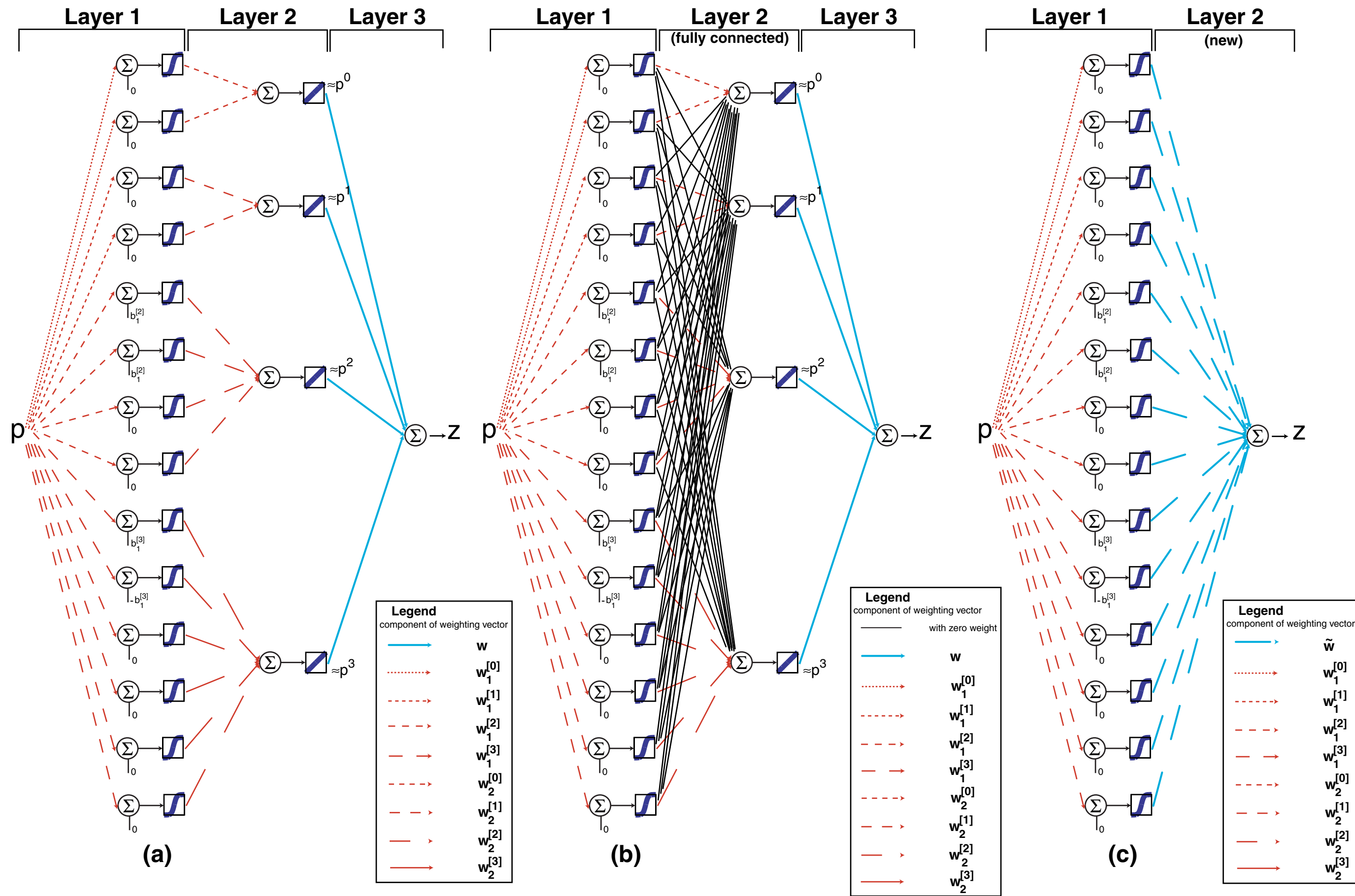

Fig. 7. A schematic neural network architecture used to demonstrate the idea of conducting one-variable polynomial fitting up to third power, where the input is normalized.

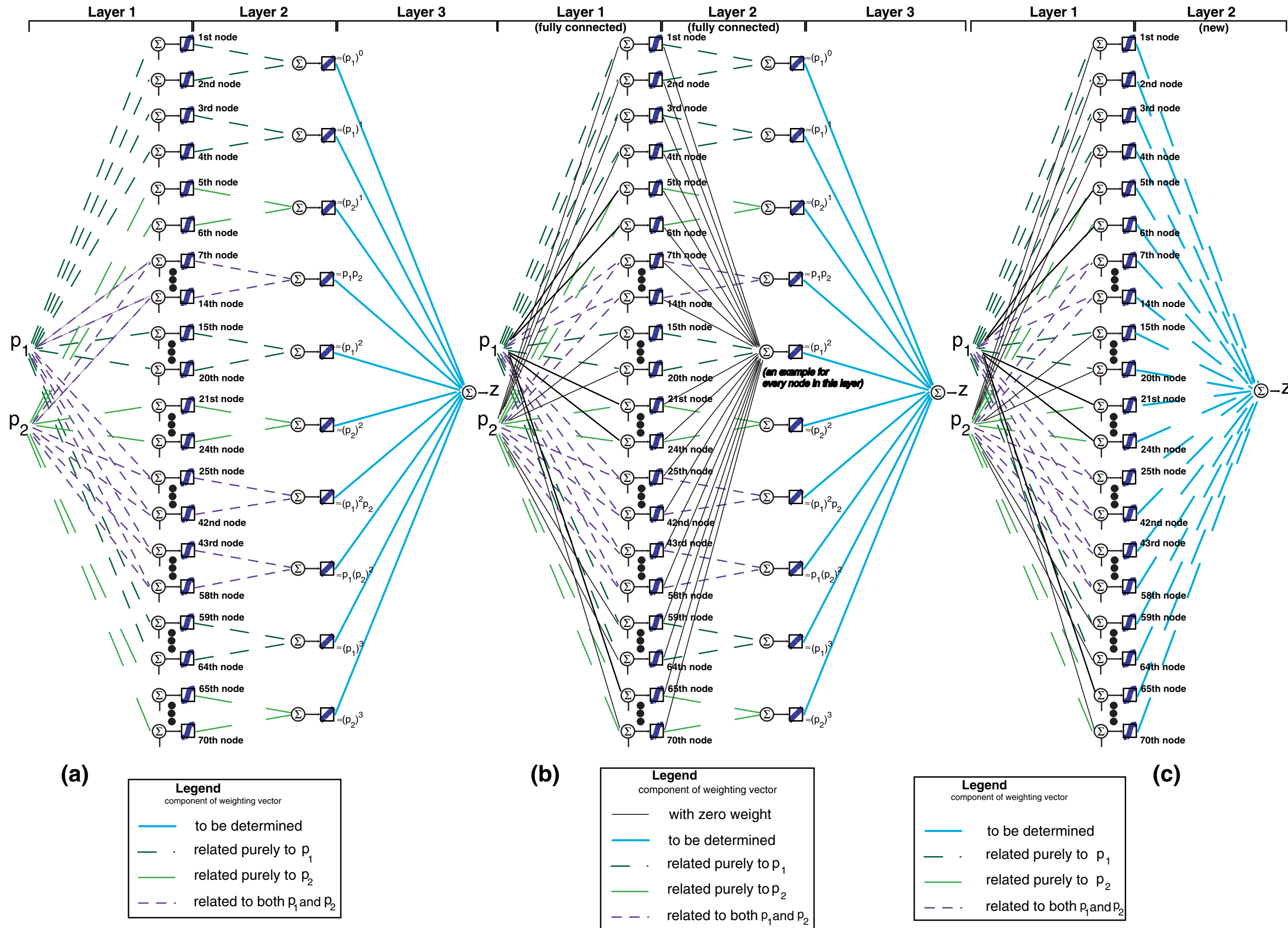

Fig. 8. A schematic neural network architecture used to demonstrate the idea of conducting two-variable polynomial fitting up to third power, where the inputs are normalized.

The dimension of $\boldsymbol{\phi}_2(t)$ equals the number of hidden nodes in the second hidden layer, n_{h_2}. Therefore, weighting matrix $\mathbf{W}_2$ has a dimension of $n_{h_2} \times n_{h_1}$. Note that the weights and biases in Layers 1 and 2 are fixed throughout the training.

Step 3. Pass $\boldsymbol{\phi}_2(t)$ through Layer 3 to obtain $\mathbf{z}(t)$, the output of the network. Another matrix–vector product is involved here:

$$\mathbf{z}(t) = \mathbf{w}(t)\boldsymbol{\phi}_2(t), \tag{22}$$

where $\mathbf{w}(t)$ is the row weighting vector to be trained in Layer 3, when the output $\mathbf{z}(t)$ is a scalar.

Step 4. Update weights $\mathbf{w}(t)$,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t), \tag{23}$$

where $\Delta\mathbf{w}(t)$ is determined based on current estimate error vector $\mathbf{g}(t) - \mathbf{z}(t)$, i.e. the difference between target and output of the network at the current time step. In this study, the LMS algorithm (see, e.g., [7]) can be used, where the design constant, learning rate γ, is a small constant that needs to be specified beforehand. The weight updating rule can be expressed by

$$\Delta\mathbf{w}(t) = \gamma[\mathbf{g}(t) - \mathbf{z}(t)]\boldsymbol{\phi}_2(t)^{\mathrm{T}}. \tag{24}$$

Step 5. Return to Step 1 and repeat. Training stops when all input–target pairs are utilized or when a certain error criterion is met.

In the above procedure, function nonlinearity is only introduced in Step 1 due to the use of sigmoidal functions. Note that the weights are updated after each input–target data pair is presented. This procedure can be classified as the *incremental training mode* [6]. In terms of the application in system identification, this training approach is for *online* identification (i.e., real-time).

4.3. Training examples

Figs. 9 and 10 present two training examples using the fixed-weight training procedure combined with the architecture shown in Fig. 7(b) for one input, or Fig. 8(b) for two uncorrelated inputs, respectively. These examples are directly related to approximating a nonlinear restoring force as a function of displacement only, or both displacement and velocity, respectively. In both cases, the inputs to the network have random values between −1 and +1, while the target value is a polynomial of the input(s) with all the coefficients equal to one for all the terms from the zeroth to third power. For the convenience of demonstration, they are arbitrarily assigned to be unity. There are 1000 pairs in the input–target data set.

Selecting the learning rate γ in Eq. (24) is important because it governs the convergence. In Fig. 9, the learning rate is chosen to be 0.40 because, based on trial and error, it gives the smallest output fitting error. This example shows the convergence of all the learned coefficients. Compared with their actual values, all the learned coefficients have satisfactory accuracy. In Fig. 10, the learning rate is 0.27. Similarly, it gives the smallest output fitting error based on trial and error. This example also shows the convergence of all the learned coefficients. Compared with their actual values, almost all of the learned coefficients have satisfactory accuracy; the terms of $(p_1)^2 p_2$ and $p_1(p_2)^2$ have less satisfactory accuracies with about 5% error. These errors may be reduced by improving the approximation accuracy of the individual polynomials especially that of the cubic term. Based on the number of nodes used in the approximation, the results are considered acceptable and further improvement can be achieved.

Table 3 summarizes the estimation errors for both the polynomial terms and learned coefficients. The error in approximating polynomials has been analyzed in Section 2, while the error in the learned coefficients may come from three sources:

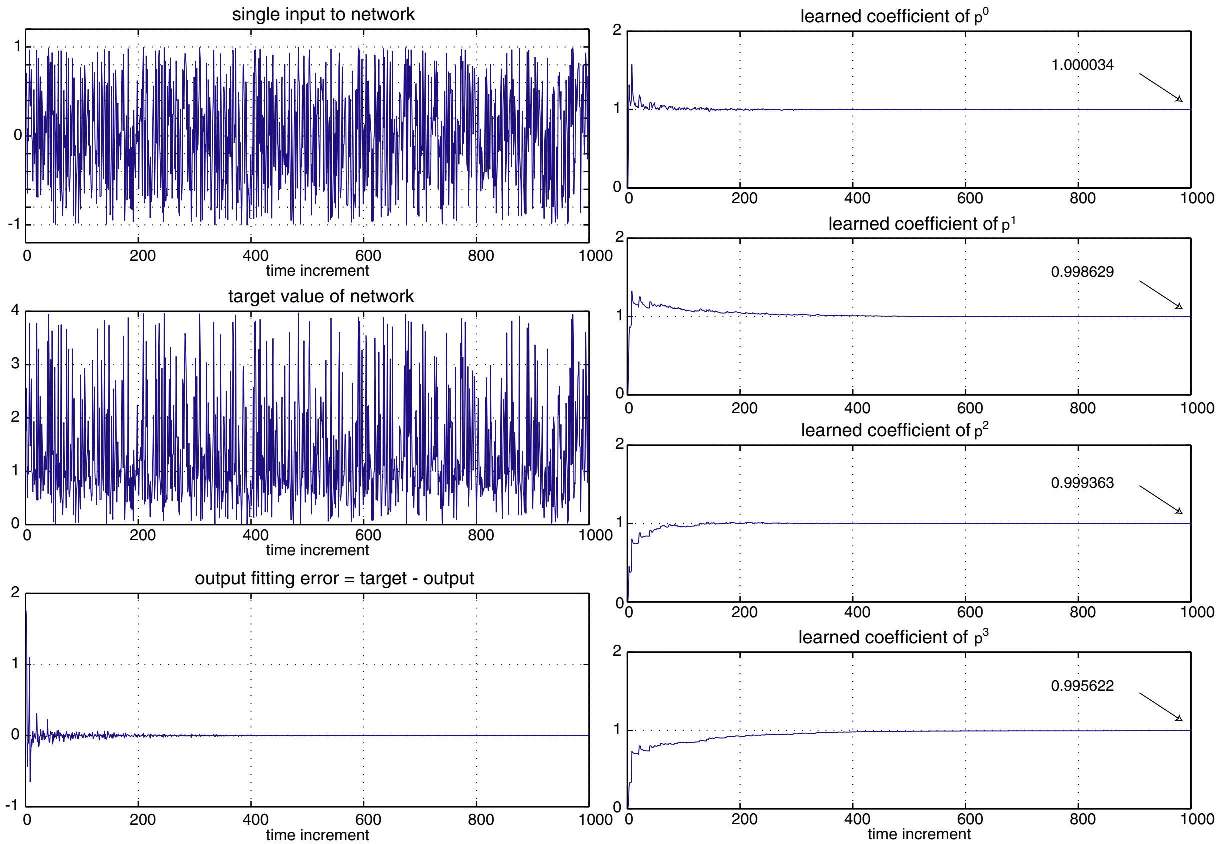

Fig. 9. A numerical application example of the neural network used for conducting one-variable polynomial fitting. The normalized input, target and output fitting error are shown in the figure.

1. the number of nodes included in the second hidden layer;
2. estimation error of individual polynomial terms, as analyzed in Section 2; and
3. truncation error of the derived weights for individual polynomial terms.

5. Discussion and conclusion

5.1. Significance

Normally when a neural network approach is adopted, a problem will immediately arise regarding the initial network setup. For a multilayer feedforward neural network, the number of hidden layers and hidden nodes in each layer need to be determined, however there is a lack of clearly defined procedures as guidance or at least as a point of reference. In this sense, this study presents an analytic approach to address the basic question on the number of needed hidden nodes for an important class of function approximation tools—polynomials and polynomial fitting. For example, if polynomial fitting is conducted up to the third power, the numbers of hidden nodes needed are 14 and 70 for one- and two-variable cases, respectively, provided that the required accuracy is comparable with that used in this study. For example, the results

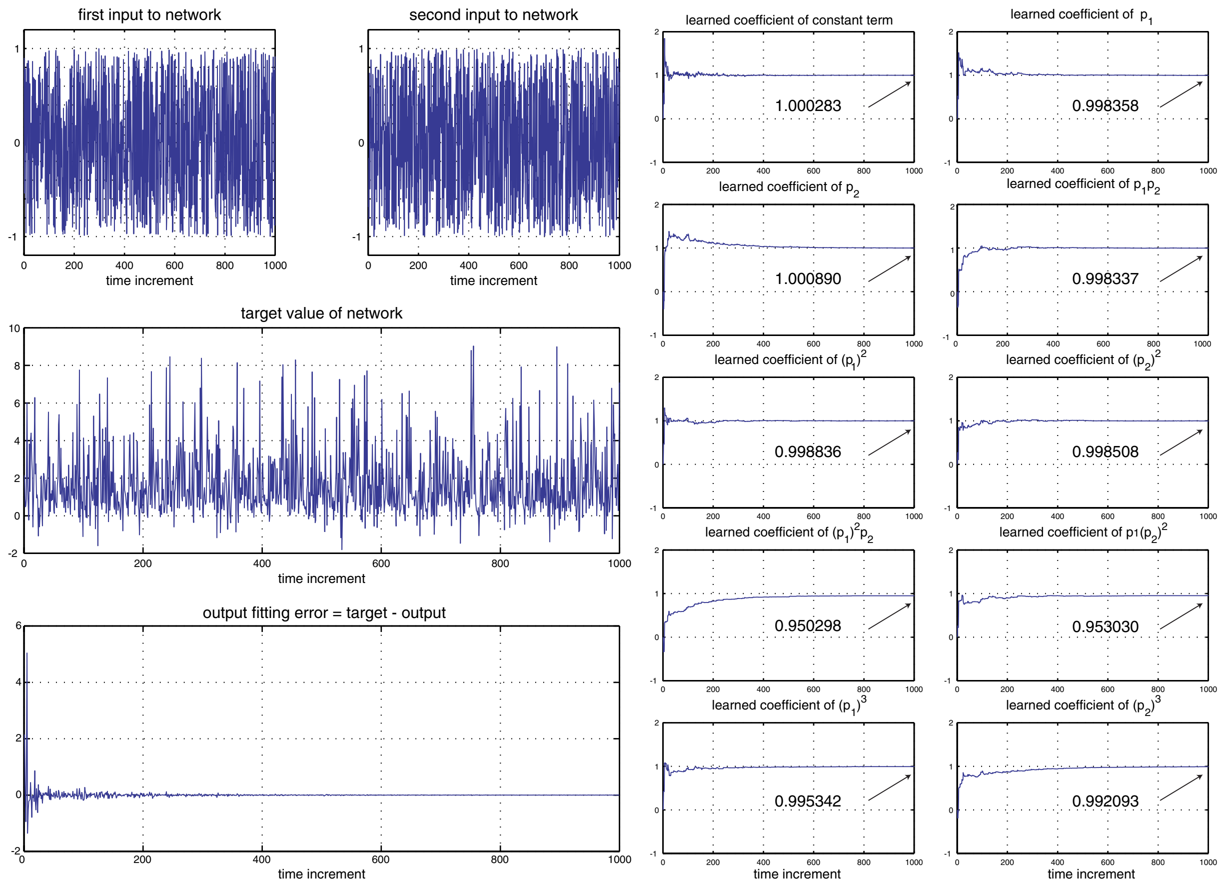

Fig. 10. A numerical application example of the neural network used for conducting two-variable polynomial fitting. The normalized inputs, target and output fitting error are shown in this figure.

Table 3
Summary of the estimated error for the normalized input cases as in Figs. 3, 6, 9 and 10

Term	Relative error in estimated polynomial term cf. Figs. 3 and 6	Relative error in learned coefficient cf. Figs. 9 and 10 (%)
p^0	$<1.5\times10^{-16}$	−0.0034
p^1	$<1.0\times10^{-3}$	0.1371
p^2	$<9.0\times10^{-4}$	0.0637
p^3	$<7.0\times10^{-3}$	0.4378
Constant term	$<1.5\times10^{-16}$	−0.0283
$(p_1)^1$	$<1.0\times10^{-3}$	0.1642
$(p_2)^1$	$<1.0\times10^{-3}$	−0.0890
p_1p_2	$<4.0\times10^{-3}$	0.1663
$(p_1)^2$	$<9.0\times10^{-4}$	0.1164
$(p_2)^2$	$<9.0\times10^{-4}$	0.1492
$(p_1)^2p_2$	$<6.0\times10^{-3}$	4.9702
$p_1(p_2)^2$	$<6.0\times10^{-3}$	4.6970
$(p_1)^3$	$<7.0\times10^{-3}$	0.4658
$(p_2)^3$	$<7.0\times10^{-3}$	0.7907

of this study have been directly adopted in [19,22,20,21] to model nonlinear hysteresis restoring forces of mechanical and structural systems.

Note again that these numbers of hidden nodes are not necessarily the least numbers, as the derivation of the *least* number of hidden nodes is not a goal of this work. This study aims to *demonstrate* quantitatively the equivalence between a neural network and a polynomial fitting scheme so as to demystify the "black-box" type approach, therefore, the authors' interest is on *how* to derive *a* set of hidden nodes for a given polynomial.

5.2. Uniqueness of trained/learned weights from fixed-weight training method

The way in which a multilayer feedforward neural network is normally used is compared with the fixed-weight training method especially in terms of the uniqueness of the trained results.

To start to train a neural network in the way that it is normally used, the neural network architecture shown in Figs. 7(b) and 8(b) will be "borrowed" to overcome the uncertainty in the initial setup because there is no other solid rationale on how to choose a proper architecture to conduct a polynomial fitting.

Since backpropagation is normally associated with slow training, a fast backpropagation training algorithm as well as batch training mode will be adopted to reflect a common situation in practice. The *batch training mode* refers to having the weights and biases updated only after the entire training set has been presented to the neural network [6]. The Levenberg–Marquardt training algorithm will be selected as a fast backpropagation algorithm. Using the analytically derived architecture shown in Figs. 7(b) and 8(b) and the training data sets in Figs. 9 and 10, respectively, the training will run for 1000 epochs in each training case where the term *epoch* is defined as the presentation of the training data set to a neural network all together in a batch [6]. Note that the term epoch as defined is not the same as the time increment used in the fixed-weight training. With regard to system identification, the batch training mode presented in terms of the epoch can be thought of as *off-line* identification, while the incremental training mode presented in terms of the time increment can be considered to be *online* identification.

The nonuniqueness issue of neural network modeling approaches, where trained network parameter results vary depending on their initial values, is well known. Take a one-variable function as an example, Table 4 is used to illustrate this phenomenon. For the purpose of demonstration, the weights and biases of Layers 1 and 2 in Fig. 7(b) are initialized using the Nguyen-Widrow layer initialization function [6] and kept the same for all the training cases 1–3. The initial weighting vector in Layer 3 is the only quantity that will vary. In Case 1, the initial weighting vector is set to be a zero vector, while in Cases 2 and 3 small numbers generated randomly with zero means are used for the initial values. These random numbers have the standard deviations (denoted by σ) of 0.1 and 0.5 in Cases 2 and 3, respectively. All the weights and biases are trained, but only the trained weighting vectors in Layer 3 are presented in Table 4 to demonstrate the non-uniqueness. The difference between the trained weights at epoch 1000 can be clearly seen for these three cases as they are the result of different sets of initial values.

Table 4
Values of the trained weights in Layer 3 in Fig. 7(b) when all the weights and biases are trained with different sets of initial values

Identified coefficient of	Case 1		Case 2		Case3	
	Initial	Learned	Initial	Learned	Initial	Learned
p^0	0	2.9549	0.0110	−3.2148	−0.1320	4.0645
p^1	0	−0.1632	0.2732	0.0488	1.2477	3.9741
p^2	0	−2.6762	0.0411	2.5752	0.4280	0.7398
p^3	0	1.7003	−0.1307	1.1950	−0.4255	2.6825

By contrast, the weights in Layer 3 obtained from the fixed-weight training method are insensitive to the variation of their initial values. In an exercise in [19], the input–target data and learning rate used are the same with those in Figs. 9 and 10. The ideal values of the weights are all one. The initial values tested include several cases. One case is all zero, the corresponding training results can be found in Table 3. In the other cases, the initial values are the random numbers of zero mean. The standard deviations considered are 0.1, 0.2, 0.3, 0.4 and 0.5, respectively. Pei [19] illustrated that all the trained weights are only slightly different from one another for both one- and two-variable cases, and these differences are negligible compared to the differences in trained weights in Table 4. It may be proper to state that the fixed-weight training method is able to yield a unique solution when the weights in only one layer are to be trained. For a given learning rate, the variation in the trained weights is nominal when their initial values vary within a reasonable range. This is an advantage of adopting the fixed-weight training method for polynomial fitting. As pointed out previously, these trained weights in Layer 3 have real "meanings"; they are the identified coefficients of polynomial terms. This is another feature of the fixed-weight training method that the commonly used training approach cannot match.

5.3. Limitations

Though the fixed-weight training method could lead to unique trained results and meaningful interpretations of weights, it should be noted that the adaptivity of the neural network is sacrificed when many layers are fixed during the training. The architecture in Figs. 7(b) and 8(b) could be used to fit a wider range of functions, however, in their partially constrained state they are merely for polynomial fitting. Recall the key focus of this study is to bridge the gap between the soft and hard computing through the example of polynomial fitting, it can be seen that the challenges of dealing with multilayer feedforward neural networks (and even neural networks in general) in a rational and transparent fashion are tremendous. The efforts made in [19,22,20,21] and this study mark the initial efforts. In particular, in this study, it has been demonstrated how neural networks can be designed to approximate a *known* type of mathematical functions. This is the first step to understanding how neural networks can be designed to solve *unknown* types of problems as are commonly encountered in engineering practice.

In terms of the details in this study, note first that the derivations of the number of hidden nodes and values of weights and biases are for normalized inputs only. If the inputs are not normalized, the approximation error is proportional to the power of the input (see Section 2), which could be significant. More nodes are needed in such a case to directly approximate the inputs that are not normalized. In Table 2, the significant digits are quite high for the weights in Layer 2 for a majority of cases. This is due to the requirement on the approximation accuracy. More nodes could be added if adopting lower significant digits to retain the degree of accuracy.

It should also be noted that this study focuses on the logistic sigmoidal function as in Eq. (3) as basis function. The results here can be readily applied to the hyperbolic tangent sigmoidal function based on the similarity of its definition to that of the logistic sigmoidal function. In either case, differentiability of the basis function is required to enable the Taylor series expansion.

Given the motivation in Section 1.1 and objectives in Section 1.4, it is important to note that this analytic study leads to injecting features of parameterization to nonparametric approaches such as neural networks for a wide range of engineering applications such as simulation, identification, health monitoring and damage detection. This is achieved by matching the capability of sigmoidal basis functions with one of the most popular basis functions, polynomials, in a constructive neural network design. Built on this work, another study by the authors [20,21] has shown the superiority of neural networks based on a linear sum of sigmoidal basis functions versus the traditional function approximation using polynomials and signum basis for a range of typical nonlinearities in engineering mechanics. Engineering judgement is gained through this line of study to guide neural network initial design and aid interpretation of training results, which directly

facilitates proper and efficient uses of this highly adaptive and powerful computational tool in engineering practice.

Acknowledgment

This study was supported in part by the National Science Foundation under SGER CMS-0332350 for the first author and CAREER Award CMS-0134333 for the third author.

References

[1] A review of structural health monitoring literature: 1996–2001, Technical Report, Los Alamos National Laboratory, LA-13976-MS, 2003.
[2] Special section: Phase i of the iasc-asce structural health monitoring benchmark, ASCE J. Engrg. Mech. 130 (1) (2004).
[3] M. Al-Hadid, J. Wright, Developments in the force-state mapping technique for non-linear systems and the extension to the location of non-linear elements in a lumped-parameter system, Mech. Systems Signal Process. 3 (3) (1989) 269–290.
[4] F. Benedettini, D. Capecchi, F. Vestroni, Identification of hysteretic oscillators under earthquake loading by nonparametric models, ASCE J. Engrg. Mech. 121 (5) (1995) 606–612.
[5] G. Cybenko, Approximation by superpositions of sigmoidal function, Math. Control Signals Systems 2 (1989) 303–314.
[6] H. Demuth, M. Beale, Neural Network Toolbox for Use with MATLAB, The Math Works Inc., 1998.
[7] M. Hagan, H. Demuth, M. Beale, Neural Network Design, PWS Publishing Company, 1995.
[8] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (1989) 359–366.
[9] W. Huang, R. Lippmann, Neural net and traditional classifiers, in: D. Anderson (Ed.), Neural Information Processing Systems, American Institute of Physics, New York, 1988, pp. 387–396.
[10] L. Jones, Constructive approximations for neural networks by sigmoidal functions, Proc. IEEE 78 (10) (1990) 1586–1589.
[11] A. Lapedes, R. Farber, in: D. Anderson (Ed.), Neural Information Processing Systems, American Institute of Physics, New York, 1988, pp. 442–456.
[12] R. Lippmann, Pattern classification using neural networks, IEEE Commun. Mag. (November) (1989) 47–64.
[13] S. Masri, G. Bekey, H. Sassi, T. Caughey, Non-parametric identification of a class of nonlinear multidegree dynamic systems, Earthquake Engrg. Struct. Dynamics 10 (1982) 1–30.
[14] S. Masri, T. Caughey, A nonparametric identification technique for nonlinear dynamic problems, J. Appl. Mech. 46 (June) (1979) 433–447.
[15] A. Meade, Regularization of a programmed recurrent artificial neural network, J. Guidance Control Dynamics (2003).
[16] A. Meade, G. Lind, B. Zeldin, Feedforward artificial neural network initialization by mathematical models, Internat. J. Neural Systems (1996).
[17] H. Mhaskar, Neural networks for optimal approximation of smooth and analytic functions, Neural Comput. 8 (1995) 164–177.
[18] K. O'Donnell, E. Crawley, Identification of nonlinear system parameters in space structure joints using the force-state mapping technique, pp. 170, Technical Report, MIT Space Systems Lab., SSL#16-85, July 1985.
[19] J. Pei, Parametric and nonparametric identification of nonlinear systems, Ph.D. Dissertation, Columbia University, 2001.
[20] J. Pei, A. Smyth, A new approach to design multilayer feedforward neural network architecture in modeling nonlinear restoring forces: Part i—formulation. ASCE J. Engrg. Mech., in press.
[21] J. Pei, A. Smyth, A new approach to design multilayer feedforward neural network architecture in modeling nonlinear restoring forces: Part ii—applications. ASCE J. Engrg. Mech., in press.
[22] J. Pei, A. Smyth, E. Kosmatopoulos, Analysis and modification of volterra/wiener neural networks for identification of nonlinear hysteretic dynamic systems, J. Sound Vib. 275 (3–5) (2004) 693–718.
[23] A. Smyth, J. Pei, S. Masri, System identification of the Vincent Thomas suspension bridge using earthquake inputs, Earthquake Engrg. Struct. Dynamics 32 (2003) 339–367.
[24] A. Wieland, R. Leighton, Geometric analysis of neural network capabilities, in: IEEE First Internal Conference on Neural Networks, vol. III, June 1987, pp. 385–392.
[25] K. Worden, G. Tomlinson, Nonlinearity in Structural Dynamics: Detection, Identification and Modelling, Institute of Physics Pub., 2001, p. 680.