Scale Equalized Higher-order Neural Networks

Chien-Ming Lin, Keng-Hsuan Wu, Jung-Hua Wang†
Electrical Engineering Department
National Taiwan Ocean University
2 Peining Rd. Keelung, Taiwan
E-mail: jhwang@mail.ntou.edu.tw

Abstract – *This paper presents a novel network, called Scale Equalized Higher-order Neural Network (SEHNN) based on concept of Scale Equalization (SE). We show that SE is particularly useful in alleviating the scale divergence problem that plagues higher-order networks. SE comprises two main processes: setting the initial weight vector and conducting the matrix transformation. An illustrative embodiment of SEHNN is built on the Sigma-Pi Network (SPN) applied to task of function approximation. Empirical results verify that SEHNN outperforms other higher-order networks in terms of computation efficiency. Compared to SPN, and Pi-Sigma Network (PSN), SEHNN requires less number of epochs to complete the training process.*

Keywords: SEHNN, Scale Equalization, Higher-order Neural Network, function approximation

1. Introduction

Recently, higher-order networks have drawn great attention from researchers due to their superior performance in nonlinear input-output mapping, function approximation, and memory storage capacity. Just to name a few, Sigma-Pi network (SPN) [1], Pi-Sigma network (PSN) [2], and Ridge Polynomial networks (RPN) [3]. Among these networks, SPN is the first network paradigm presented to the field. Training SPN is straightforward as it employs the simple error correction algorithm. However, it suffers a major drawback: proliferation in input combinatorial terms (a.k.a. product terms). That is, adding a network input or order rapidly increases the total number of higher-order correlations which in turn increases the number of connection weights. On the other hand, PSN requires smaller number of connection weights than SPN when performing the same task. Unlike SPN, the number of connection weights in PSN increases rather mildly with the input dimension. But PSN has some pitfalls too. Particularly, we note that PSN has to use either random or asynchronous scheme to update connection weights, inevitably decreasing computation efficiency [2]. In comparison, SPN with fully synchronous update scheme can be trained as fast as a Single Layer Perceptron (SLP) network [4]. Nevertheless, SPN are likely to have great scale differences among connection weights owing to the vast number of product terms, which impose great difficulty in training, namely excessive training epochs are often needed in order to achieve satisfactory results [2]. Aiming to solve aforementioned problems, the concept of Scale Equalization (SE) is presented in this paper. An exemplar network based on SE concept, called Scale Equalized Higher-order Neural Network (SEHNN), is proposed. SEHNN can alleviate the problem of scale divergence by applying an optimal initial weight vector set to obtain a transformation matrix. With the matrix, the original input training data is cast into a different feature space wherein the training process can be reformulated. By doing so, the network can advantageously converge more quickly and achieve better performance. Although there are many ways to optimally determine the initial weight vector, for illustration purpose we simply adopt the Support Vector Machine (SVM). The SVM is well known for its capability of assuring convergence to the global optimal solution [5] using small number of training data. More desirably, SVM can be extended to allow for non-linear problems by projecting the original inputs in a higher dimensional feature space and by formulating the linear separable problems in the feature space.

The rest of this paper is organized as follows. Some related background concerning higher order neural networks is given in Section 2, in particular, SPN, PSN, and SVM are briefly described in order to reveal their distinguishing characteristics and differences in the context of training efficiency. In section 3, we elaborate on implementing SEHNN that incorporates SE and SVM. Experimental results are provided in section 4 to verify the effectiveness of the SEHNN. Finally, conclusions and discussions are given in section 5.

2. Higher-order networks

2.1 Sigma-Pi network (SPN)

The basic architecture of SPN contains a set of Higher-order processing units (HPUs) [6], as shown in Fig. 1, where d, $\sigma(\cdot)$, and K denote the input dimension, the activation function (e.g. Sigmoid) and K the network order, respectively. Thus, Fig. 1 shown a SPN having HPUs with order k, k=1…K. In general, the greater the order K is, the more computationally capable the network can be. Eq. (1) prescribes how a HPU is generated in SPN, and the final output of SPN is determined by using Eq. (2)

†The correspondent author

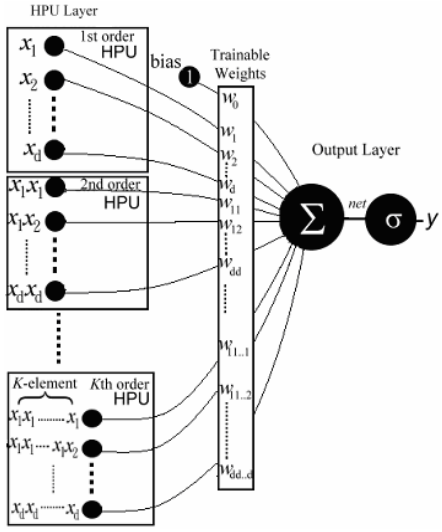

Fig. 1 Architecture of SPN

$$term_k(x_1,...x_d) = \sum_{i_1=1}^{d}\sum_{i_2=i_1}^{d}...\sum_{i_k=i_{k-1}}^{d} w_{i_1 i_2 ... i_k} x_{i_1} x_{i_2} ... x_{i_k} \quad (1)$$

$$\begin{aligned} y(x_1,...x_d) &= \sigma\left(w_0 + \sum_{k=1}^{K} term_k \right) \\ &= \sigma\left(w_0 + \sum_{k=1}^{K}\sum_{i_1=1}^{d}\sum_{i_2=i_1}^{d} ... \sum_{i_k=i_{k-1}}^{d} \left[w_{i_1 i_2 ... i_k} \left(\prod_{j=1}^{K} x_{i_j} \right) \right] \right) \end{aligned} \quad (2)$$

The undesirable effect of Eq. (1) is that the total number of weights (denoted by n) required in SPN proliferates rapidly with the growing of d and K. To see this more clearly, the dependency of n on K and d is given in Eq.(3),

$$n(K,d) = \binom{K+d}{K} \quad (3)$$

Boost in the number of weights inevitably demands the increase in the computation load required for completing the training process.

2.2 Pi-Sigma network (PSN)

The drawback of proliferation in SPN naturally leads to a simple conclusion that reducing the number of connection weights should improve the computation performance of training process. In light of this, PSN was proposed [2] by building a higher-order network capable of performing similar computational tasks but with fewer trainable weights than SPN. Eq. (4) prescribes the output y of PSN with order K and input dimension d.

$$y(x_1,...x_d) = \sigma\left(\prod_{j=1}^{K} S_j \right) = \sigma\left(\prod_{j=1}^{K} (w_{0j} + \sum_{i=1}^{d} w_{ij} x_i) \right), \quad (4)$$

$$S_j(x_1,...x_d) = w_{0j} + \sum_{i=1}^{d} w_{ij} x_i \quad (5)$$

The number of connection weight of PSN is $(d+1)K$. Compared to SPN, the proliferation rate with respect to K obviously is milder. Although training individual linear sum term (i.e., Eq. (5)) can employ the error correction learning algorithm, but as noted in [2] training the whole set of summing units with a fully synchronous scheme could get unstable. Using asynchronous and random update scheme inevitably deteriorates the computation performance. Despite the reduction in the number of trainable weights, computation time required for each training epochs in PSN expands more rapidly as K increases than in SPN. Table 1 and Fig. 2 show that a training epoch in PSN needs almost twice the computation load required for training the SPN. Despite HPU-based SPN requires less number of arithmetic *multiply* operations than PSN per epoch, the former needs much more epochs than the latter.

Table 1 Comparisons of computation load

Computation load / Training scheme	Number of Multiplications (in each training epoch)
K^{th} order PSN (Asynchronous)	$(d+1)K^2+dK$
K^{th} order PSN (Random)	$(d+1)K+d$
K^{th} order SPN (full synchronous)	$2\binom{K+d}{K}+3$

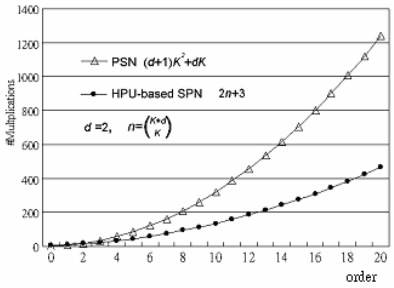

Fig. 2 Number of multiplications vs. order

Due to the fact that combinatorial terms in SPN are obtained from multiplying the input signals to generate the vast number of product terms, it is likely to have great scale differences among the weights, which impose great difficulty in training, particularly excessive training epochs are often needed in order to achieve satisfactory results. To see this, a perceptron output prescribed in Eq.(6) is used. We multiply some of x_1w_1, x_2w_2, …, x_dw_d in Eq.(6) for obtaining higher-order correlation terms in Eq.(1), and rewrite them as Eq.(7).

$$y(x_1,...x_d) = \sigma\left(\sum_{i=1}^{d} x_i w_i \right) = \sigma(x_1w_1 + x_2w_2 ... x_dw_d) \quad (6)$$

$$term_k(x_1,...x_d)=\sum_{i_1=1}^{d}\sum_{i_2=i_1}^{d}...\sum_{i_k=i_{k-1}}^{d}\left[\left(w_{i_1}x_{i_1}\right)\left(w_{i_2}x_{i_2}\right)..\left(w_{i_k}x_{i_k}\right)\right]$$
$$=\sum_{i_1=1}^{d}\sum_{i_2=i_1}^{d}...\sum_{i_k=i_{k-1}}^{d}\left[\left(w_{i_1}w_{i_2}...w_{i_k}\right)\left(x_{i_1}x_{i_2}...x_{i_k}\right)\right] \quad (7)$$

Comparing Eq.(1) and Eq.(7) readily yields $w_{i_1i_2...i_k}=w_{i_1}w_{i_2}...w_{i_k}$. If many of the trained weights $w_{i_1},w_{i_2},...,w_{i_k}$ are less than 1.0, the multiplied result $w_{i_1i_2...i_k}$ is very likely a very small value. In contrast, if all the trained weights are greater than one, then $w_{i_1i_2...i_k}>>1$. That is, some connection weights being nearly zeros, while the rest being excessively large. To summarize, the inefficient training in SPN is a direct consequence of the great scale differences among the weights. The greater the scale difference is, the more difficult it is to specify a proper learning rate for the training algorithm employed.

2.3 Support Vector Machine

In last decade, there has been a surge of interest in SVM [7]. It has been shown that SVM gives good performance of generalization on a wide variety of problems such as function approximation problems [8], handwritten character recognition [9], face detection [10]. Extensive use of SVM is hampered by the following reasons: (1) the training algorithms for SVM are too slow, especially for large problems. (2) SVM is too complex, subtle, and difficult for an average engineer to implement. The training time of SVM exponentially increases [5]. Still, SVM sets up one of the most powerful methods for constructing a mathematical model on the basis of a given number of training examples. The basic idea of SVM is to construct a hyperplane as the decision surface in such a way that the margin of separation between positive and negative examples is maximized. Fig.3 shows the geometric construction of an optimal hyperplane for a two-dimensional input space.

In simplest, linear form, the discriminant function for the hyperplane is $u=w\cdot x+b$, where x is an input vector, w is an adjustable weight vector, and b is a bias. We can find out the maximum margin m by Eq. (8).

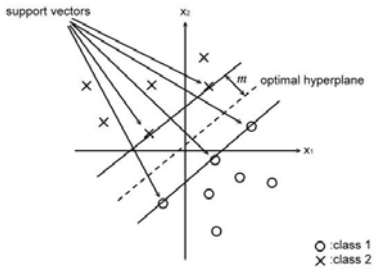

Fig.3 Illustration of an optimal hyperplane in SVM

$$m=\frac{2}{\|w\|} \quad (8)$$

Maximizing margin can be expressed via the following optimization problem [11]. Given a training set of labeled pairs (x_i, d_i), i=1,…,N where $x_i\in R^n$ and $d_i\in\{1,-1\}$,

$$\min_{w,b,\xi}\ \frac{1}{2}w^Tw+C\left(\sum_{i=1}^{l}\xi_i^2\right)$$
$$\text{s.t.}\ \ d_i\left(w^T\phi(x_i)+b\right)\ge 1-\xi_i \quad (9)$$

where w denotes the optimal initial weight vector. The slack variables ξ_i measure the deviation of a data point from the ideal condition of pattern separability. The function $\phi(x)$ transforms x into a higher dimensional feature space. Instead of tackling Eq.(9) directly, in practice its dual problem is solved as follows.

$$\text{maximize}\ \ \sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_jQ_{ij}$$
$$\text{s.t.}\ \ \sum_{i=1}^{N}\alpha_id_i=0\ \ ,0\le\alpha\le C\ \ \text{for}\ i=1...N \quad (10)$$

where C is a user-specified positive parameter. Q is a $N\times N$ positive semi-definite matrix, $Q_{ij}\equiv d_id_jk(x_i,x_j)$, and $k(x_i,x_j)\equiv\phi(x_i)^T\phi(x_j)$ is the kernel function that depends on choosing the features and representing the scalar product in the feature space. The kernel function measures the similarity or distance between the input vector and the stored training vector. Well-known models of the kernel function include Gaussians, polynomial, and neural network non-linearities [11]. Because that the objective function in Eq. (10) is bounded, SVM must converge to the *global optimal solution* in a finite number of iterations. Solving Eq.(10) yields the values of α_i. Plugging them into Eq. (11) gives us the optimal initial weight vector.

$$w=\sum_{i=1}^{N}\alpha_id_i\mathbf{ker} \quad (11)$$

where $\mathbf{ker}=\phi(x_i)$. In the following section, we will show how this optimization technique SVM can be useful in implementing SEHNN.

3. Implementing SEHNN

3.1 Setting the initial weight vector

To solve the problem of scale difference, we first transform the original weight space so as to reduce the search space during the training process by rewriting Eq. (2) as Eq. (12),

$$y(m_1,...m_{n-1}) = \sigma\left(w_0 + \sum_{p=1}^{n-1}\left(w_p m_p\right)\right) = \sigma\left(\sum_{p=0}^{n-1}\left(w_p m_p\right)\right) \quad (12)$$

where m_p denotes the product terms in Eq.(2), m_0 the bias, w_p the weights, and n the total number of weights. Assume L is the number of training patterns, from which n samples are chosen. If $L \geqq n$, we randomly pick from the block of training patterns as the sample patterns. If $L < n$, input patterns near the L training patterns are generated as the *augmented* samples. The closer the input pattern to the sample pattern, the more similar their corresponding outputs are. Fig. 4 shows the selection process. We can use Vector Quantization [11] technique, such as LBG algorithm [12] or Self-Organization Map (SOM) [13] for choosing samples. After sampling, we will obtain n input features $\vec{x}_s = (\vec{x}_1,....,\vec{x}_d)$ s=1…n.

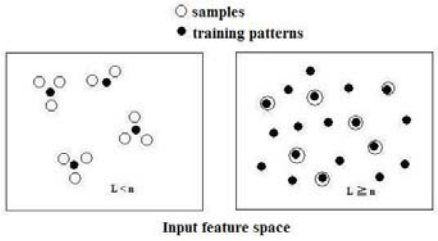

Fig.4 The sampling process

Rather than setting the initial weights by zero or random numbers as in most neural network models, here we employ SVMs to determine the initial weight vector that allows *exact* input-output mapping for the selected n training patterns. As will be shown later, such initialization scheme offers the benefit of fast terminating at a minimum during the training process. This scheme, in fact, can also be applied to other single layer networks, e.g. SLP.

According to Eq.(11),

$$\vec{W} = \vec{\alpha} \otimes (\vec{T}_n \cdot \mathbf{ker}) \quad (13)$$

where the symbol $\otimes$ is defined as an vector multiply operator, i.e. $[a \quad b] \otimes [c \quad d] = [ac \quad bd]$. The essence of our approach is to apply the initial weight vector to deduce a matrix that transforms the original input space into another. Formulated in vector form, the product results of all L training patterns are calculated and arranged as $Z_t = [\vec{z}_1 \vec{z}_2 ... \vec{z}_L]$ and $\vec{T}_t = [t_1 t_2 ... t_L]$. Consequently, Eq.(12) is rewritten as Eq.(14).

$$\vec{T}_t = \sigma[\vec{W} \cdot Z_t] \quad (14)$$

By plugging $\vec{W}$ into Eq. (14), we have

$$\vec{T}_t = \sigma[(\vec{\alpha} \otimes (\vec{T}_n \cdot \mathbf{ker})) \cdot Z_t] \quad (15)$$

Following that, we rewrite Eq.(15) as Eq.(16):

$$\vec{T}_t = \sigma[(\vec{\alpha} \otimes \vec{T}_n) \cdot (\mathbf{ker} \cdot Z_t)] \quad (16)$$

$$= \sigma[\vec{V} \cdot U]$$

where $\vec{V}$ and U represent the vector of $(\vec{\alpha} \otimes \vec{T}_n)$ and the results of $(\mathbf{ker} \cdot Z_t)$, respectively. Decomposing the vector $\vec{V}$ and the matrix U yields the network structure of SEHNN shown in Fig. 5.

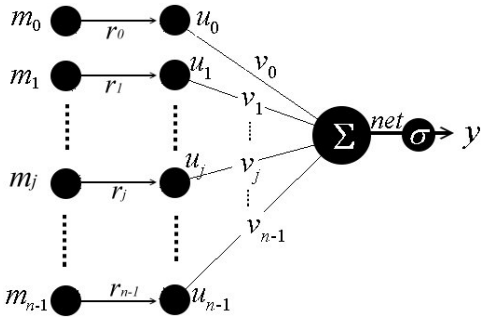

Fig.5 Network architecture of SEHNN

And the output y is given in Eq. (17).

$$y(m_1, m_2 ... m_{n-1}) = \sigma\left(\sum_{j=0}^{n-1} v_j \cdot u_j\right) \quad (17)$$

$$u_j = r_j \times m_j$$

where r_i denotes the j-th element of the vector **ker**. Note that the vector $[u_0 \ u_1 \ \ldots \ u_{n-1}]$ can be viewed as a transformed input by the vector **ker**. With this transformation, the original input training data is cast into a feature space wherein the training process can be reformulated, and the network training can be conducted that space. More significantly, the scale difference among connection weights is alleviated due to the use of **ker**. The beauty of this transformation is that SEHNN not only advantageously converges more quickly, but also achieves better performance.

3.2 Training Process

After obtaining the matrix U, training the resulting SEHNN will be the same as in training a SLP. Eq. (18) is the update equation derived based on the gradient descent rule. Solving the partial differential equation yields the modification amount in weights.

$$\Delta v_i \propto -\frac{\partial \varepsilon}{\partial v_i}, \quad \varepsilon = \frac{1}{L}\sum_{j=1}^{L}(t_j - y_j)^2 \tag{18}$$

$$\Delta v_i = \frac{\eta}{L}\sum_{j=1}^{L}\left[(t_j - y_j)\sigma'(net_j)\left(r_j \cdot \vec{z}_j\right)\right] \tag{19}$$

Substituting r_j with **ker** yields

$$\Delta V = \frac{\eta}{L}\sum_{j=1}^{L}\left[(t_j - y_j)\sigma'(net_j)\left(\mathbf{ker} \cdot \vec{z}_j\right)\right] \tag{20}$$

To further simplify, one can rewrite Eqs. (19) and (20) in vector forms, and the results are Eq. (21) and Eq. (22).

$$\Delta v_i = \frac{\eta}{L} \cdot \left[(\vec{T}_t - \vec{Y}) \otimes \vec{\Gamma}\right] \cdot H_i \tag{21}$$

$$\Delta \vec{V} = \frac{\eta}{L} \cdot \left[(\vec{T}_t - \vec{Y}) \otimes \vec{\Gamma}\right] \cdot U^T \tag{22}$$

where $H_i=(r_j \cdot Z_t)^T$, $\vec{\Gamma} = [\sigma`(net_1)\ \sigma`(net_2) \ldots \sigma`(net_L)]$, and $\vec{Y} = [\, y_1 \ \ y_2 \ldots y_L \,]$. In training process, the element of H_i and U^T are constant, and they can be pre-determined. In order to compare, the error correction formulate in SPN in vector form is derived as follows

$$\begin{aligned} \Delta \vec{W} &= \frac{\eta}{L}\sum_{j=1}^{L}\left[(t_j - y_j) \cdot \sigma'(net_j) \cdot \vec{z}_j^T\right] \\ &= \frac{\eta}{L} \cdot \left[(\vec{T}_t - \vec{Y}) \otimes \vec{\Gamma}\right] \cdot Z_t^T \end{aligned} \tag{23}$$

Comparing Eq. (21) and Eq. (23) easily reveals that they are actually the same in terms of computational complexity, implying the same computation load per epoch. But the major difference is that our proposed update equations have removed the problem of great scale differences that plagues SPN.

4. Experimental Results

The Gabor function shown in Fig. 6 is used as the test input to show the function approximation capability of SEHNN. The formula of Gabor function is given below.

$$h(x,y) = \frac{1}{2\pi\lambda\sigma^2} \cdot e^{-\frac{(x/\lambda)^2 + y^2}{2\sigma^2}} \cdot e^{2j\pi(u_0 x + v_0 y)} \tag{24}$$

where λ is the aspect ratio, σ is the scale factor and (u_0, v_0) the modulation parameters. The function becomes circularly symmetric if λ=1. In simulations, the parameters are set as λ =1, σ=5, and (u_0, v_0) = (1, 1). 256 points were picked from 16×16 grid on -0.5≦x≦0.5 and -0.5≦y≦0.5, from which we selected 28 points out of 256 points as the training patterns, the rest as test patterns. The output activation function used the hyperbolic tangent function. The network order K and learning rate η used 6 and 0.1, respectively. Here the sampling patterns were chosen using uniformly random scheme, as shown in Fig.7. Fig.8 illustrates the learning curve of the simulation.

The SEHNN achieved MSE=0.005 after 128 epochs, and the computation time for initial weight in Pentium IV is 0.0753 sec. (0.062 sec. for training and 0.0133 sec. for preparation). To compare SEHNN and the original counterpart PSN, we also tested the PSN using the same Gabor function. Table 2 compares SEHNN, PSN and HPU-based SPN.

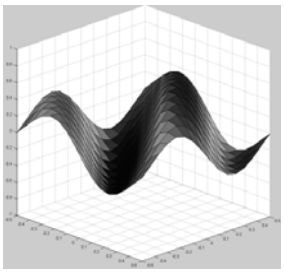

Fig.6 The Gabor Function.

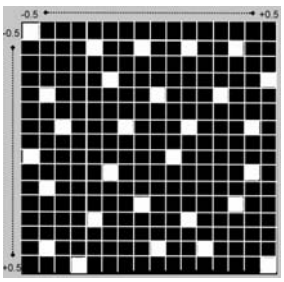

Fig.7 Samples selected uniformly

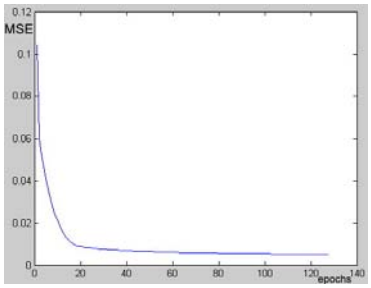

Fig.8 Learning curve of SEHNN

Table 2 Comparison of Performance

Network	MSE (training)	MSE (testing)	Epochs	Time (sec.)
SEHNN	0.005	0.0067	128	0.0753

PSN	0.005	0.0075	142	0.2300
HPU-based SPN	0.028	0.0370	5000	1.9865

Fig. 9 shows the result of scale equalization and compares the resulting connection weights $\{v_i: i=1\ldots28\}$ of SEHNN and that of HPU-based SPN $\{w_i: i=1\ldots28\}$, where $S(w_i) = \log_{10}(|w_i|)$ and $S(v_i)= \log_{10}(|v_i|)$. Results in Fig.9 clearly show that the scale technique presented in this paper indeed can effectively equalize the scales of connection weight.

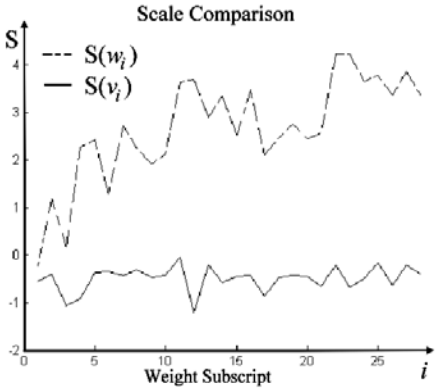

Fig. 9 Scale difference of connection weights

5. Conclusions

We have presented a novel concept of Scale Equalization (SE) useful for implementing higher-order networks. We have shown that SE is effective in alleviating the scale divergence problem that plagues higher-order networks. An illustrative embodiment built on the Sigma-Pi Network (SPN) applied to task of function approximation has been conducted, and empirical results verify that the resulting network SEHNN outperforms other higher-order networks in terms of computation efficiency. Finally, the future work will focus on the use of incremental techniques [14] to farther improve SEHNN in minimizing the network order.

Acknowledgment

This research was supported by the National Science Council of Taiwan under grant: NSC 93-2611-E-019-007

References

[1] D. E. Rumelhart, J. L. McClelland, *Parallel Distributed Processing*, Vol. 1, MA: MIT Press, Cambridge,1987.

[2] Y. Shin and J. Ghosh, "The pi-sigma network: An efficient higher-order neural network for pattern classification and function approximation," *Proc. Int. Joint Conf. Neural Networks*, Vol. I, Seattle, WA, pp. 13-18, July 1991.

[3] J. Ghosh and Y. Shin, "Ridge polynomial networks," *IEEE Trans. on Neural Networks*, Vol. 6, No. 3, pp. 610-622, 1995.

[4] B. Widrow and M.E. Hoff, Jr., "Adaptive switching circuits." 1960 IRE Western Electric Show and Convention Record, Part 4, Aug. 23, 1960, pp. 96-104.

[5] Osuna, E., Freund, R., Girosi, F., "An improved training algorithm for support vector machines," *Neural Networks for Signal Processing [1997] VII. Proc. of the 1997 IEEE Workshop,* pp. 276-285, Sep. 1997.

[6] C. L. Giles and T. Maxwall, "Learning, invariance, and generalization in a higher-order neural network," *Applied Optics*, Vol. 26, No. 23, pp 4972-4978, 1987.

[7] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[8] Shu-Xia Lu, Xi-Zhao Wang, "A comparison among four SVM classification methods: LSVM, NLSVM, SSVM and NSVM," *Machine Learning and Cybernetics, 2004. Proceedings of 2004 Intnl Conf. on*, Vol. 7, pp. 4277-4282, Aug. 2004.

[9] Cheng-Lin Liu, Sako, H., Fujisawa, H., "Effects of classifier structures and training regimes on integrated segmentation and recognition of handwritten numeral strings," *Pattern Analysis and Machine Intelligence, IEEE Trans on*, Vol. 26, Issue: 11 , pp. 1395-1407, Nov. 2004.

[10] Kepenekci, B., Akar, G.B., "Face classification with support vector machine," *Signal Processing and Communications Applications Conference*, *2004. Proceedings of the IEEE 12th* , 28-30, pp. 583-586, April 2004

[11] B. Fritzke, "Growing Cell Structures-A self-organizing network for unsupervised and supervised learning," *Neural Networks*, Vol. 7, No. 9, pp. 1441-1460, 1994.

[12] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Transactions on Communications*, Vol. COM-28, pp. 84-95, 1980.

[13] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, Vol. 43, pp. 59, 1982.

[14] A. G. Ivakhnenko, "Polynomial theory of complex systems," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-1, no. 4, Oct. 1971