Scale Equalization Higher-order Neural Networks

Jung-Hua Wang, Keng-Hsuan Wu, and Fu-Chiang Chang
Department of Electrical Engineering
National Taiwan Ocean University
2 Peining Rd. Keelung, Taiwan
Email: jhwang@mail.ntou.edu.tw

Abstract

This paper presents a novel approach, called Scale Equalization (SE), to implement higher-order neural networks. SE is particularly useful in eliminating the scale divergence problem commonly encountered in higher order networks. Generally, the larger the scale divergence is, the more the number of training steps required to complete the training process. Effectiveness of SE is illustrated with an exemplar higher-order network built on the Sigma-Pi Network (SESPN) applied to function approximation. SESPN requires the same computation time as SPN per epoch, but it takes much less number of epochs to complete the training process. Empirical results are provided to verify that SESPN outperforms other higher-order neural networks in terms of computation efficiency.

1. Introduction

Recently, higher-order networks have drawn great attention due to their superior performance in nonlinear input-output mapping, function approximation, and memory capacity. Just to name a few, Sigma-Pi network (SPN) [10], Product Unit neural network (PUNN) [2], Pi-Sigma network (PSN) [3], and Ridge Polynomial networks (RPN) [4]. Among them, SPN is the first one. With one trainable layer, SPN has been shown capable of performing nonlinear input-output mapping. In the layer between output and input layers is the HPU layer composed of Higher-order Processing Units (HPUs) [1] that generates correlations of inputs of various orders. The advantage of SPN is that it can be trained with the simple error correction algorithm [11]. Nevertheless, it suffers a drawback that the number of weights proliferates rapidly as the network order and/or input dimension increase [1]. Specifically, the product terms (higher-order correlations) can be any possible combinations of the network inputs, adding a network input or order will increase the total number of product terms which in turn rapidly increases the number of connection weights, namely the proliferation problem. Implementing SPNs can become unrealistic when the function to be approximated involves high dimensional input data. But in applications (e.g. image processing) where low input dimension is usually encountered, this problem is of no subject.

PSN, on the other hand, requires less number of weights than SPN in order to deliver the same performance, and the number of weights in PSN grows more mildly with the increasing input dimension. Despite these merits, PSN has some pitfalls. Particularly in PSN the input-sum terms are multiplied in succession, which prohibits derivation of the update equation directly based on the chain rule. As such, fully synchronous training in PSN is mathematically infeasible. Instead, in each epoch PSN must employ either random or asynchronous scheme [3] to update connection weights, resulting in worse computation efficiency. In contrast, SPN with fully synchronous update scheme can be trained as fast as a Single Layer Perceptron (SLP) network [9], provided that the product terms of all training patterns are pre-calculated. However, because the input multiplication involved in generating product terms, great scale differences among connection weights are likely to occur, which impose great difficulty in training SPN, and large number of training epochs are often needed in order to obtain satisfactory results [3].

In light of the foregoing problems, this paper presents the concept of Scale Equalization (SE). By conducting a matrix transformation, SE can significantly reduce the scale difference among connection weights. In so doing, training higher-order networks can be completed with much fewer epochs than otherwise. Following, some background necessary to understand the significance of this work is given in Section 2, in particular, we will briefly introduce SPN and PSN in order to reveal their differences in network structures. In Section 3 we elaborate the SE concept, demonstrate how to apply SE to build a higher-order network. Simulation results are given in Section 4 for verifying the effectiveness of SESPN (Scale Equalization Sigma-Pi Network). Finally, conclusions are given in Section 5.
2. Higher-order Networks

2.1. Sigma-Pi network (SPN)

Fig. 1 shows the architecture of SPN that contains sets of HPUs (higher-order processing units [10]). The input dimension is denoted as \(d\), and activation function (e.g. Sigmoid) as \(\sigma\). Assume the network order is \(K\), thus Fig. 1 contains \(K\) HPUs with order \(k\), \(k = 1 \ldots K\).

![Figure 1. Architecture of SPN](image)

Generally speaking, the greater the order \(K\) is, the more computationally capable the network can be. In order to apply the error correction learning algorithm to train SPN, we must take the product terms \(x_1, x_2, \ldots (x_1x_2), \ldots, (x_d \cdot x_d \ldots)\) as the input units to a SLP. Eq.(1) describes how to generate a HPU in SPN. Eq.(2) prescribes the final output of SPN.

\[
\text{term}_k (x_1 \ldots x_d) = \sum_{i=1}^{K} \sum_{j=1}^{d} \sum_{i_1=i} \ldots \sum_{i_k=i} \ldots \sum_{i_d=i} w_{i_2 \ldots i_k} x_{i_1} x_{i_2} \ldots x_{i_k} \quad (1)
\]

\[
y(x_1 \ldots x_d) = \sigma \left[ w_0 + \sum_{k=1}^{K} \text{term}_k \right] = \sigma \left[ w_0 + \sum_{k=1}^{K} \sum_{j=1}^{d} \sum_{i_1=i} \ldots \sum_{i_k=i} \ldots \sum_{i_d=i} w_{i_2 \ldots i_k} \left( \prod_{j=1}^{K} x_{i_j} \right) \right] \quad (2)
\]

Note that, unless the domain \textit{a priori} for function approximation is known, product terms other than the prescribed by Eq.(1) can be used in SPN.

An imminent effect of Eq.(1) is that the total number of weights (denoted by \(n\)) required in the HPU-based SPN proliferates rapidly with the growing of \(K\) and \(d\). Explicitly, 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 finish the training process.

2.2. Pi-Sigma network (PSN)

The forgoing discussions on SPN naturally lead to a simple conclusion that one can improve the computation performance of training process simply by reducing the number of weights. Aiming to build a network with fewer trainable weights than SPN, the Pi-Sigma network (PSN) was proposed [3]. The output \(y\) of PSN with order \(K\) and input dimension \(d\) is prescribed in Eq.(4).

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

Here, the number of weights equals to \((d+1)K\). Compared to SPN, the proliferation rate (w.r.t \(K\)) obviously is more mild. Although the error correction learning algorithm can be used to train each separate linear sum, training the whole PSN with a fully synchronous scheme is unstable [3]. On the other hand, the use of asynchronous or random update scheme inevitably deteriorates the computation performance.

Furthermore, despite the reduction in the number of trainable weights, computation time required for each training epoch in PSN increases more rapidly as \(K\) increases than in SPN. Table 1 and Fig. 2 show that a training epoch in PSN requires nearly twice the computation load required for training SPN. Note that it is not fair to compare the random scheme with the other two training schemes. This is because in each training epoch, the random scheme only updates a set of \((d+1)\) weights in one of \(K\) linear input sums in PSN, instead of all the \((d+1)K\) weights of PSN. Consequently, its computation load is only \(1/K\) of the asynchronous update scheme.
Table 1. Comparisons of computation load

<table>
<thead>
<tr>
<th>Training scheme</th>
<th>Number of Multiplications (in each training epoch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K(_{\text{th}}) order PSN (Asynchronous)</td>
<td>((d+1)K^2+Kd)</td>
</tr>
<tr>
<td>K(_{\text{th}}) order PSN (Random)</td>
<td>((d+1)K+d)</td>
</tr>
<tr>
<td>K(_{\text{th}}) order SPN (full synchronous)</td>
<td>(2\left(K^d+\frac{1}{K^d}\right)+3)</td>
</tr>
</tbody>
</table>

Figure 2. Number of multiplications vs. order

Despite HPU-based SPN requires less number of multiplications than PSN per epoch, the former needs much more epochs than the latter. According to [3], SPN requires more than 5,000 epochs for approximating a Gabor function, in contrast only 132 epochs for PSN (with the same order) to obtain similar result. The inefficient training in SPN is mainly due to the presence of great scale differences among the weights. Since the combinatorial terms in SPN are obtained from multiplying the input signals, it is likely to have great scale differences among the connection weights. To see this, a perceptron output prescribed in Eq.(5) is used,

\[
y(x_1...x_d) = \sigma\left(\sum_{i=1}^{d} x_i w_i\right) = \sigma(x_1w_1 + x_2w_2 + ... + x_dw_d) \tag{5}
\]

We multiply some of \(x_iw_i, x_iw_2, ..., x_dw_d\) in Eq.(5) for obtaining higher-order correlation terms in Eq.(1), and then rewrite them as

\[
term_k(x_1...x_d) = \sum_{i=1}^{d} \sum_{i' \neq i}^{d} \sum_{i'' \neq i, i'' \neq i'}^{d} \left[\left(w_i x_i\right)\left(w_{i'} x_{i'}\right)\left(w_{i''} x_{i''}\right)\right] \tag{6}
\]

Comparing Eq.(1) and Eq.(6) readily yields

\[
w_{i_1i_2...i_k} = w_{i_1}w_{i_2}...w_{i_k} \quad \text{if many of the trained weights } w_{i_1}, w_{i_2}, ..., w_{i_k} \text{ are less than } 1.0, \text{ the multiplied result } w_{i_1i_2...i_k} \text{ is very likely a very small value. In contrast, if all the trained weights are greater than one, } w_{i_1i_2...i_k} >> 1. \text{ That is, some connection weights being nearly zeros, while the rest being excessively large. Note that the larger the scale difference is, the more difficult it is to specify a suitable learning rate for the training algorithm employed.}

3. Scale Equalization neural networks

3.1. Scale Equalization with Matrix Transform

To solve the scaling problem, we transfer the original weight space into another in order to reduce the searching space during the training process. To proceed, we first rewrite Eq.(2) as Eq.(7),

\[
y(m_1,...,m_n) = \sigma\left(w_0 + \sum_{p=1}^{n} (w_p m_p)\right) = \sigma\left(\sum_{p=0}^{n} (w_p m_p)\right) \tag{7}
\]

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 total number of training patterns, from which \(n\) samples are picked. Fig. 3 illustrates the selection process.

If \(L < n\), we simply choose inputs near the \(L\) training patterns as the samples, and set the output of these samples as the desired output of the corresponding training patterns. The closer the input pattern to the sample pattern, the more similar their corresponding outputs are. If \(L \geq n\), the sample patterns are randomly picked from the pool of training patterns. Note that one can also use Vector Quantization [6] techniques, such as LBG algorithm [7] or Self-Organization Map (SOM) [8] for selecting samples.

After sampling process we will obtain \(n\) input features \(\tilde{x}_s = (x_1,...,x_s), s = 1...n\). Following that, the product vector is written as

\[
\tilde{z}_s = [m_0(\tilde{x}_s),...,m_n(\tilde{x}_s)]^T, s = 1...n
\]

where \(m_j(\tilde{x}_s)\) denotes the \(j\)th product term for the \(s\)th input feature.
We use a matrix \( Z = [\tilde{z}_1 \, \tilde{z}_2 \, \ldots \, \tilde{z}_n] \) to represent \( n \) product vectors. Because each vector \( \tilde{z} \) has \( n \) elements, \( Z \) is a \( n \)-by-\( n \) matrix. Likewise, \( \tilde{T} = [t_1 \, t_2 \, \ldots \, t_n] \), where \( t_i \) is \( i^{\text{th}} \) desired output. Consequently, Eq.(7) is rewritten as Eq.(8):

\[
\tilde{T} = \sigma([w_1 \, w_2 \, \ldots \, w_n] \cdot Z) = \sigma(\tilde{W} \cdot Z)
\]

Thus the initial weight vector for the SPN is set as in Eq.(9).

\[
\tilde{W} = \sigma^{-1}(\tilde{T}) \cdot Z^{-1}
\]

The weights determined by Eq.(9) enable the network to perform a precise input-output mapping for the selected \( n \) training patterns. The idea here is this. Compared to setting the initial weights by zeros or random numbers, using the initialization of Eq.(9) should result in more easily ending up a minimum when training is completed. Therefore, it is expected the training iterations will be greatly saved. Also, this initialization scheme can also be applied to other single layer networks, e.g. SLP.

Plugging Eq.(9) into Eq.(8) transfers the original weight space into another space. To compute in vector form, the product results of all \( L \) training patterns are calculated and arranged as \( Z_t = [\tilde{z}_1 \, \tilde{z}_2 \, \ldots \, \tilde{z}_L] \). Also, the corresponding \( L \) desired output patterns are arranged as \( \tilde{T}_t = [t_1 \, t_2 \, \ldots \, t_L] \). After that, we substitute \( T_t \) and \( Z_t \) to the Eq.(10) for obtaining Eq. (10):

\[
\tilde{T}_t = \sigma\left(\sigma^{-1}(\tilde{T}) \cdot Z^{-1}\right) \cdot Z_t = \sigma(\tilde{V} \cdot U)
\]

where \( \tilde{V} \) and \( U \) represent the vector of \( \sigma^{-1}(\tilde{T}) \) and the results of \( (Z^{-1} \cdot Z_t) \), respectively. If the inverse function of the employed activation function exists, we can set the initial weight vector \( \tilde{V} = \sigma^{-1}(\tilde{T}) \), otherwise \( \tilde{V} = \tilde{T} \). Decomposing the vectors \( \tilde{V} \) and the matrix \( U \) yields the network structure of SESPN shown in Fig. 4. The output formula is given in Eq.(11):

\[
y = \sigma\left(\sum_{i=1}^{l} u_i \cdot v_i \right) = \sigma(\tilde{v} \cdot Z^{-1} \cdot \tilde{z})
\]

where \( R_t \) denotes the \( i^{\text{th}} \) column of the inverse matrix \( Z^{-1} \). Thus, the input vector \( [u_1 \, u_2 \, \ldots \, u_n] \) can be viewed as a transformed input by matrix \( Z^{-1} \). We also obtain a new weight space \( \tilde{V} \). Due to the fact that initial weight vector \( \tilde{V} = \sigma^{-1}(\tilde{T}) \) or \( \tilde{V} = \tilde{T} \), the scale of the weights is the same with \( \sigma^{-1}(\tilde{T}) \) or \( \tilde{T} \).

### 3.2 Training SESPN

After the matrix \( U \) is calculated, training the resulting higher-order network will be the same as in SLP. Corresponding update equations can be derived from MSE (Mean Square Error) and gradient descent rule, as given in Eq.(12). Solving the partial differential equation yields the modification amount in weights.

\[
\Delta v_i = -\frac{\partial e}{\partial v_i} = \frac{1}{L} \sum_{t=1}^{L} (t_i - y_i)^2
\]

Replacing \( R_t \) by \( Z^{-1} \), \( \Delta v_i \) becomes \( \Delta \tilde{V} \). It is easy to obtain fully synchronous training rule from Eq.(13):

\[
\Delta \tilde{V} = \frac{1}{L} \sum_{t=1}^{L} (t_i - y_i) \sigma'(net_i)(Z^{-1} \cdot \tilde{z}_i)
\]

To further simplify, Eqs.(13 and 14) can be written in vector forms shown in Eq.(15, 16), respectively:

\[
\Delta \tilde{V} = \frac{1}{L} \left[ (\tilde{T}_t - \tilde{y}) \otimes \tilde{T} \right] H_i
\]

\[
\Delta \tilde{V} = \frac{1}{L} \left[ (\tilde{T}_t - \tilde{y}) \otimes \tilde{T} \right] U^T
\]

where \( H_i = (R_t Z_t)^T \), \( \tilde{T} = [\sigma'(net_1) \sigma'(net_2) \ldots \sigma'(net_L)] \), \( \tilde{y} = [y_1 \, y_2 \, \ldots \, y_L] \), and the symbol \( \otimes \) is defined as an vector multiply operator, i.e. \( [a \, b] \otimes [c \, d] = [ac \, bd] \). The elements of \( H_i \) and \( U^T \) are constant in training process, and they can be calculated before training.

As the error correction algorithm used in SPN, the derived update equations also consist of three parts: learning rate, errors, and inputs. To compare, we write the error correction algorithm in SPN in vector form:

![Figure 4. Network architecture of SESPN](image-url)
\[
\Delta w_j = \eta \sum_{i=1}^{L} \left[ (t_i - y_i) \cdot \sigma'(net_i) \cdot x_{ij} \right] \\
= \eta \left[ (\tilde{y}_j - \tilde{y}) \cdot (\tilde{x}_j - \tilde{x}) \right] \
\]

where \( x_{ij} \) means the \( i \)th element of the \( j \)th input pattern and \( \tilde{x}_j = [x_1, x_2, \ldots, x_L]^T \). By comparing Eq.(15) and Eq.(17), it is easy to see they are actually the same, hence the same computation load per epoch. More significantly, the proposed update equations have overcome the problem of great scale differences between weights in original SPN. Revisiting Table 1 should easily reveal the fact that training SESPN would require much less computation load than the asynchronous error correction algorithm for PSN.

4. Experimental Results

In this section experimental results are illustrate the performance of the proposed SESPN. The Gabor function is used as test input to show the function approximation capability. Fig. 5 shows the Gabor function, which traditionally plays an important role in physics. The formula of Gabor function is given by

\[
h(x, y) = \frac{1}{2\pi \lambda \sigma^2} e^{-\frac{(x/\lambda)^2 + y^2}{2\sigma^2}} e^{2\pi i(u_0x + v_0y)}
\]

where \( \lambda \) is the aspect ratio, \( \sigma \) is the scale factor and \( (u_0, v_0) \) the modulation parameters. If \( \lambda = 1 \), the function becomes circularly symmetric. In our simulation, the parameters are set as \( \lambda = 1, \sigma = 5, (u_0, v_0) = (1, 1) \).

To generate training patterns and test patterns, 256 points were picked from 16x16 grid on \(-0.5 \leq x \leq 0.5 \) and \(-0.5 \leq y \leq 0.5 \). We selected 128 points out of 256 points as training pattern, and the rest as test patterns. The output activation function used the hyperbolic tangent function. The network order \( K \) and learning rate \( \eta \) used 6 and 0.1, respectively. According to Eq.(3), the network with \( K=6 \) has 28 weights, i.e., \( n = 28 \). As mentioned earlier, we must sample \( n \) training patterns in the input feature space. Here the sampling patterns were picked using uniformly random method, as shown in Fig. 6. In simulation, the proposed network SESPN achieved MSE = 0.005 after 203 epochs, and the total computation time used in Pentium IV is 0.0833 sec. (0.077 sec. for training and 0.0063 sec. for the preparation work). Fig. 7 shows the learning curve of the simulation.

To compare SESPN and its original counterpart PSN, we also tested the PSN using the same Gabor function. In this experiment, PSN with \( K=6 \) achieved MSE =0.005 after 142 epochs (total time used: 0.230 sec.). Table 2 compares SESPN, PSN and HPU-based SPN in terms of MSE.

![Figure 5. Data made by Gabor Function.](image)

![Figure 6. Samples selected uniformly.](image)

![Figure 7. The learning curve using the SESPN](image)

<table>
<thead>
<tr>
<th>Network</th>
<th>MSE (training)</th>
<th>MSE (testing)</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESPN</td>
<td>0.005</td>
<td>0.0067</td>
<td>203</td>
</tr>
<tr>
<td>HPU-based SPN</td>
<td>0.028</td>
<td>0.037</td>
<td>5000</td>
</tr>
<tr>
<td>PSN</td>
<td>0.005</td>
<td>0.0075</td>
<td>142</td>
</tr>
</tbody>
</table>
To show the effect of scale equalization, the trained weights \( \{v_i : i = 1...28\} \) and the trained weights of HPU-based SPN \( \{w_i : i = 1...28\} \), are compared in Fig. 8, where \( S(w_i) = \log_{10}(|w_i|) \) and \( S(v_i) = \log_{10}(|v_i|) \). The scale differences between the weights in the transferred space was \( 10^1 \), whereas \( 10^4 \) in the original space. Moreover, result in Fig. 8 has verified that the scale equalization technique is indeed effective in equalizing the scales of connection weights by transforming input space.

### 5. Conclusions

We have presented the concept of Scale Equalization which is useful for implementing higher-order neural networks. An exemplar network is built based on Sigma-Pi Network (SESPN). The problem of scale divergence has been successfully alleviated. Simulation results have justified the computation efficiency of SESPN. In particular, it takes fewer epochs to finish training SESPN than its original SPN counterpart. Also, SESPN used less computation time than PSN for performing function approximation. In short, the proposed SESPN is indeed more computationally efficient than HPU-based SPN and PSN.

In SESPN, the input-output mapping capability relies on the network order, the greater the order is, the more powerful the capability. However, the network order is closely related to the number of HPUs. Clearly, it is desired to design a non-fixed higher-order network for unknown function approximations. Hence the future work will focus on the use of incremental techniques [12] to further improve SESPN in minimizing the network order. The feasibility of applying SE in alleviating the proliferation problem will be examined. For example, if the corresponding weight of a product term is too small, it can be eliminated from the network.

### Acknowledgment:

This research was supported by the National Science Council of Taiwan under grant: NSC 92-2213-E-019-011

### References


