Chapter 3

Higher-Order Neural Networks

Higher-order neural networks are networks that utilize higher combinations of its inputs. A goal of this thesis is to train PUNNs, which are examples of higher-order neural networks. In this context, this section provides an overview of higher-order neural networks. This chapter discusses four types of higher-order neural networks and the problems associated with the training of product unit neural networks specifically.

3.1 Sigma-Pi Networks

Hidden units of a sigma-pi neural network calculate a product (or conjunct) of the inputs [Lee Giles 1987, Maxwell *et al* 1986]. In sigma-pi neural networks a weight is applied, not only to each input, but also to the second and possibly higher-order products or conjuncts of the inputs. The connections in sigma-pi neural networks allow one unit to gate another: Thus, if one unit of a multiplicative pair of units is zero, then the other member can have no effect on the output. On the other hand, if one unit of a pair has a value 1, the output of the other unit is passed unchanged to the receiving unit. In this way a polynomial function of the inputs is presented as input to the transfer function of the output layer, i.e. the value of the output unit O_k is

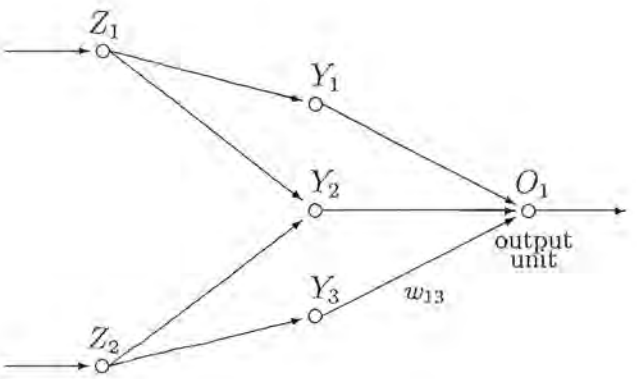


Figure 3.1: Sigma-pi network

$$o_k = f\left(\sum_{q \in conjunct} w_{qk} \prod_{k=1}^{N} z_{qk}\right)$$

where f is the activation function, w_{qk} a synaptic weight, $z_{q1}, z_{q2}, ..., z_{qN}$ are the N input signals combined to form the product or conjunct, and q indexes the conjuncts or products that are used in unit k; conjunct is the set of all conjuncts of subscripts for the inputs. The architecture derived from the above function presents a method of constructing higher-order networks.

Figure 3.1 illustrates a sigma-pi network with two inputs, where multiplication instead of summation is performed in the hidden layer, followed by a summation unit in the output layer. That is, for example, $y_2 = z_1 z_2$ and $y_1 = z_1$, where y_j is the output of hidden unit Y_j. The weight between hidden unit Y_j and output unit O_k is denoted by w_{kj}. In sigma-pi networks a polynomial function of the inputs is presented to the activation function of the output layer. For the example in figure 3.1,

$$\begin{aligned} o_1 &= f(w_{11}y_1 + w_{12}y_2 + w_{13}y_3) \\ &= f(w_{11}z_1 + w_{12}z_1z_2 + w_{13}z_3) \end{aligned} \tag{3.1}$$

Although the terms contain products of inputs there are no powers of each input greater than one, this gives rise to the name *multi-linear* for the terms in this kind of expression. Nodes with multi-linear terms are also called higher-order nodes, since their activation depends on terms whose multiplicative order is greater than one. The problem with sigma-pi units is that the number of terms, and therefore the weights, increase very rapidly with the number of inputs, thus becoming unacceptably large for use in many situations [Durbin *et al* 1989, Lee Giles 1987]. Thus a disadvantage of this type of architecture (refer to figure 3.1) is that a combinatorial explosion in the number of weights may result if conjuncts are not hand coded [Lee Giles 1987]. Researchers combat this problem by restricting the number of units, i.e. the number of terms, to a configuration sufficient to achieve the desired degree of accuracy using a priori knowledge about the given task. Normally, only one or a few of these terms are relevant in neural networks [Lee Giles 1987]. The most common approach to determine the best architecture is to let the network grow incrementally. In this approach an initial network consisting of a few terms is chosen and new terms are added to the network as soon as the error cannot be reduced using the existing architecture. This incremental growth process is repeated until the desired error level or accuracy is reached.

3.2 Pi-Sigma Networks

Ghosh and Shin introduced another higher-order network, the 'pi-sigma' network, which avoided the exponential increase in the number of weights and processing units normally associated with higher-order networks [Ghosh *et al* 1992]. A pi-sigma network (PSN) consists of an input layer, a single hidden layer of linear summation units and product units in the output layer. The term pi-sigma comes from the fact that these networks use *products of sums* of input components. PSNs have only one layer of adjustable weights, the weights of the output layer is normally fixed at 1,

resulting in PSNs to exhibit fast learning [Ghosh *et al* 1992].

The output of a pi-sigma network is computed as follows,

$$o_k = f\left(\prod_{j=1}^{J} y_{kj}\right) \tag{3.2}$$

where

$$y_{kj} = \sum_{i=1}^{I+1} w_{kji} z_i \tag{3.3}$$

where f is the activation function, $z_i, ..., z_I$ are the input signals, z_{I+1} an input to the bias unit, w_{kji} is the weight between input unit Z_i and hidden unit Y_{kj} for the k^{th} output unit O_k, $w_{kj,I+1}$ is the threshold (or bias), y_{kj} is the output of hidden unit Y_{kj} and o_k is the output of output unit O_k. Each hidden unit is connected to only one output unit, as indicated to by the subscript k in y_{kj}. Thus equation (3.2) also shows that for multiple output PSNs an independent summing unit is required for each output unit. PSNs show a combinatorial explosion of higher-order terms as the number of inputs to the network increases.

Figure 3.2 illustrates a typical pi-sigma network, where w_{kji} is the weight between input unit Z_i and hidden unit Y_{kj}, $f(\cdot)$ is the standard logistic function applied to the output units and all the weights leading to the output unit are fixed to 1. The hidden layer consists of summation units and the output layer of product units. Let y_{kj} be the output of the j^{th} summation unit of the k^{th} output unit, O_k. A linear activation is assumed for the hidden units. A PSN provides only constrained approximation of a power series, resulting in the PSN not to uniformly approximate all continuous multivariate functions that can be defined on a compact set. However, universal approximation can be attained by summing the outputs of several PSNs of different order. The resulting network of PSNs is called a Ridge Polynomial Network

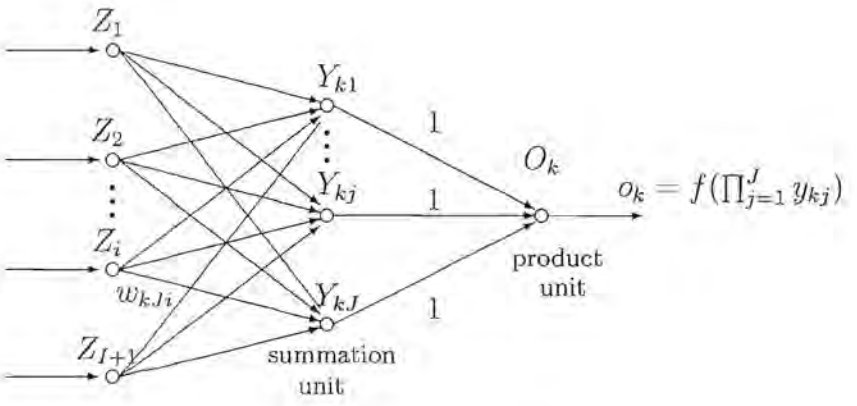


Figure 3.2: Pi-Sigma Network

(RPN) [Shin *et al* 1995]. A PSN can accept both analog and binary input/output by using suitable non-linear activation functions. The logistic function can be used as a non-linear activation function and the signum or thresholding function can be used for binary outputs.

The learning rule used by Ghosh *et al* for PSN is a randomized version of the gradient descent procedure [Ghosh *et al* 1992]. During each training cycle of a PSN, a summing unit is randomly selected and all the weights associated with this summing unit are updated using gradient descent. This modification of updating only a subset of weights, instead of all the weights, in each training cycle resulted in reduced training time of a PSN. Ghosh and Shin reported that pi-sigma networks using only three or four summing units could tackle fairly complex approximation and classification problems [Ghosh *et al* 1992].

3.3 Functional Link Networks

Functional link networks (FLNs) also generate higher-order functions of the input components [Pao 1989, Pao *et al* 1992]. FLNs are usually single-layer networks that are able to handle linearly non-separable classes by increasing the dimension of the input space by using non-linear combinations of the inputs. In FLNs, the input vector is augmented with a suitably enhanced representation of the input data, thereby artificially increasing the dimension of the input space [Ghosh *et al* 1992, Hussain *et al* 1997, Pao 1989, Pao *et al* 1992]. The extended input data are then used for training, as for standard feed-forward neural networks. Basically, the inputs are transformed in a well understood mathematical way so that the network does not have to learn basic math functions. Figure 3.3 depicts a typical functional network. In figure 3.3 the I inputs are denoted by $z_1, z_2, ..., z_I$, the bias to the hidden layer is denoted by z_{I+1} and the M extended inputs for functional links by $h_1, h_2, ..., h_M$.

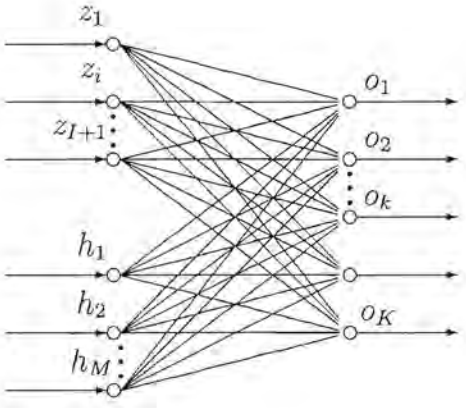


Figure 3.3: Functional Link Network

The dimensionality of the input space for FLNs can be increased in two ways [Pao 1989]:

- The *tensor* or *output product model*, where the cross-products of the input terms

are added to the model. For example, for a network with three inputs z_1, z_2, and z_3, the cross products are: z_1z_2, z_1z_3, z_2z_3, therefor adding second order terms to the network. Third order terms such as $z_1z_2z_3$ can also be added.

- *Functional expansion of base inputs*, where mathematical functions, such as sin, cos, log, etc. are used to transform the input data.

The number of terms generated using these methods grow rapidly with the increase of the dimension of the input vector. In FLNs no new information is added, but the representation of the input is merely enhanced. An advantage of FLNs is reduced training time due to the higher-order representation of the inputs, since the network does not have to learn these higher-order terms. Klassen *et al* found that functional links not only increases learning rates, but also has an effect of simplifying the learning algorithms [Klassen *et al* 1988]. Another advantage of FLNs is that it can outperform multilayer networks in certain cases due to its intrinsic mapping properties [Ghosh *et al* 1992].

3.4 Second-Order Neural Networks

Another type of higher-order neural network, the second order neural network, was developed by Milenković *et al* [Milenković *et al* 1996]. The research of Milenković *et al* was inspired by a greedy constructive neural network algorithm called the Hyperplane Determination from Examples (HDE) that suggested a discrete approach to neural network optimization suitable for parallel and distributed implementation [Fletcher *et al* 1995]. The objective of the neural network architecture developed by Milenković *et al* was to overcome the HDE local minima problem by allowing hidden units with higher representational power. The higher representational power was achieved by allowing neurons with input interactions of the following forms:

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i \tag{3.4}$$

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i + \sum_{i=1}^{I} w_i^{(2)} z_i z_i \tag{3.5}$$

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i + \sum_{i=1}^{I} w_i^{(2)} z_i z_i + \sum_{i=1}^{I-1} \sum_{j=i+1}^{I} w_{ij}^{(3)} z_i z_j \tag{3.6}$$

where f is the activation function, $\vec{z}$ is the input vector to the network, $w_i^{(1)}$, $w_i^{(2)}$ are weight parameters associated with the i^{th} input value z_i, while $w_{ij}^{(3)}$ is a weight associated with the product of the i^{th} and j^{th} input values z_i and z_j. First-order neural networks contain neurons only constructed with interaction functions described by equation (3.4). Feed-forward neural networks that are constructed using neurons as described by all three interaction functions above, i.e. as described by equations (3.4), (3.5) and (3.6), are referred to as second-order neural networks.

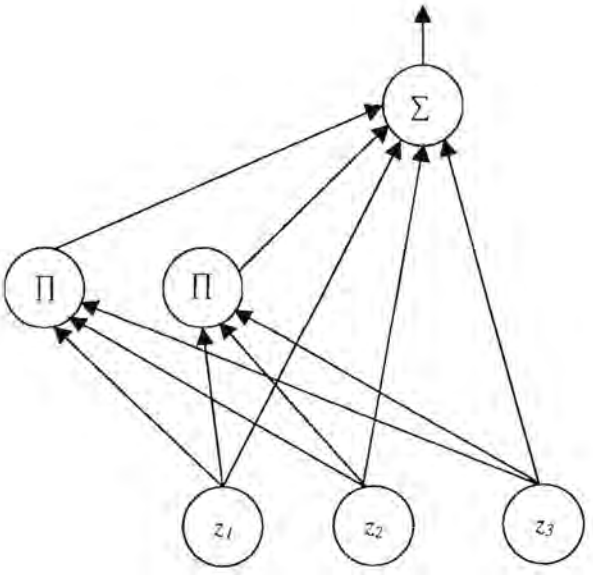
3.5 Product Unit Neural Networks

Product unit neural networks were introduced by Durbin and Rumelhart [Durbin *et al* 1989], and further explored by Janson and Frenzel [Janson *et al* 1993] and Leerink *et al* [Leerink *et al* 1995]. Durbin and Rumelhart suggested two types of networks incorporating PUs [Durbin *et al* 1989]. In the one network type (refer to figure 3.4(a)) each SU is directly connected to the input units, and also connected to a group of dedicated PUs. The other network (refer to figure 3.4(b)) consists of alternating layers of product and summation units, terminating the network with a SU.
Product units compute the net input signal as:

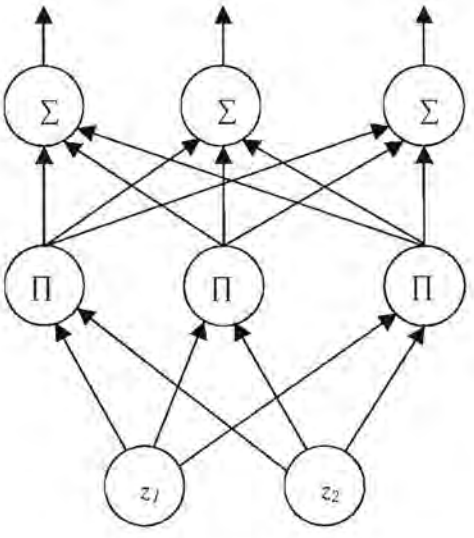
$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} + z_{I+1} \cdot v_{j,I+1}$$

instead of

$$net_{y_j} = \sum_{i=1}^{I+1} z_i v_{ji}$$



(a) Summing units fed by inputs and dedicated product units



(b) Alternating layers of product and summing units

Figure 3.4: Two types of PUNNs

A product unit can automatically learn the higher-order term that is required by the network, unlike pi-sigma and sigma-pi units where the higher-order terms are hard-coded in the network. Product units can learn polynomials such as,

$$f(z) = a_0 + a_1 \cdot z^1 + a_2 \cdot z^2 + ... + a_n \cdot z^n \tag{3.7}$$

and any other function that can be represented by a polynomial. It can be shown that any function can be represented by a polynomial of degree n, which is a Fourier series expansion of z. The problem however, is to determine what the value of n should be when approximating a specific function using a Fourier series. Product units are much more general than sigma-pi units: While a sigma-pi unit is constrained to using just polynomial products, product units can use fractional and even negative products [Durbin *et al* 1989]. The net input to a product unit is computed as follows,

$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} - \theta_j$$

where net_{y_j} is the net input to hidden unit Y_j, Z_i is an input unit, v_{ji} is the weight between input unit Z_i and hidden unit Y_j, the threshold is denoted by θ_j and I is the total number of input units. Durbin and Rumelhart suggested two types of networks incorporating PUs [Durbin *et al* 1989]. This thesis assumes a network architecture which consists of an input layer, a hidden layer consisting of product units and an output layer consisting of summation units. Linear activations are assumed for all units. It is assumed that bias units occur in both the input and hidden layers, that respectively serve as bias to hidden units and bias to output units. However, it is shown in section 3.7.2 that the bias unit in the input layer is redundant and thus omitted from the input layer of PUNNs and replaced by a 'distortion unit' in this thesis, while retaining the bias in the hidden layer.

Neural networks are trained using learning or training rules. The next section derives

the product unit training rule for the PUNN architecture used in this thesis, assuming gradient descent as optimization algorithm.

3.6 Product Unit Training Rule

Using the architecture outlined above, the activation of a product unit for a specific pattern p is expressed in terms of logarithms and exponentials:

$$\begin{aligned} y_{j,p} &= net_{y_j,p} \\ &= \prod_{i=1}^{I} z_{i,p}^{v_{ji}} + z_{I+1,p} \cdot v_{j,I+1} \qquad (3.8) \\ &= e^{\ln(\prod_{i=1}^{I+1} z_{i,p}^{v_{ji}})} + z_{I+1,p} \cdot v_{j,I+1} \\ &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_{i,p}|} + z_{I+1,p} \cdot v_{j,I+1} \\ y_{j,p} &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_{i,p}|}(\cos(\pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_i) + \imath \cdot \sin(\pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_i)) + z_{I+1,p} \cdot v_{j,I+1} \qquad (3.9) \end{aligned}$$

where

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_{i,p} \geq 0 \\ 1 & \text{if } z_{i,p} < 0 \end{cases} \qquad (3.10)$$

and $z_{i,p} \neq 0$. The complex part of equation (3.9) is omitted for training the PUNN, since Durbin and Rumelhart have discovered in their experiments that apart from the added complexity of working in the complex domain, i.e. doubling of equations and weight variables, no substantial improvements in results were gained [Durbin *et al* 1989].

Equation (3.9) then simplifies to (refer to appendix A),

$$y_j = e^{\rho} \cdot \cos(\pi\phi) + z_{I+1} \cdot v_{j,I+1} \qquad (3.11)$$

where

$$\rho = \sum_{i=1}^{I} v_{ji} \ln|z_i| \qquad (3.12)$$

and

$$\phi = \sum_{i=1}^{I} v_{ji}\mathcal{I}_i \qquad (3.13)$$

The objective of supervised training of NNs is to minimize the error between the approximation by the NN and the target function. The error in approximation is usually expressed as the mean squared error (MSE)

$$\mathcal{E}_{MSE} = \frac{1}{2PK} \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{3.14}$$

where P is the total number of training patterns (or observations), K is the number of outputs, $t_{k,p}$ is the desired (target) output for the k^{th} output unit, O_k, for a specific pattern p, and $o_{k,p}$ is the actual output of the NN. If gradient descent is used, the change in hidden-to-output weights are

$$\Delta w_{kj} = -\eta \cdot \frac{\partial E}{\partial w_{kj}} \tag{3.15}$$

and for input-to-hidden weights,

$$\Delta v_{ji} = -\eta \cdot \frac{\partial E}{\partial v_{ji}} \tag{3.16}$$

where η is the learning rate, w_{kj} is the weight between hidden unit Y_j and output unit O_k, and v_{ji} is the weight between input Z_i and hidden unit Y_j. For more detail on the derivations of these equations refer to appendix A. In the case of the hidden-to-output weights, the equations are as for standard feed-forward networks, i.e.

$$\frac{\partial E}{\partial w_{kj}} = -\delta_{o_{k,p}} \cdot f'(net_{o_k,p}) \cdot y_{j,p} \tag{3.17}$$

$$= -(t_{k,p} - o_{k,p}) \cdot y_{j,p} \tag{3.18}$$

where $f'(net_{o_k,p})$ is the derivative of the activation function used for output unit O_k (which is equal to one in the case of linear activation functions), $\delta_{o_k,p}$ is the output error, $\delta_{y_j,p}$ is the hidden layer error, $net_{o_k,p}$ and $net_{y_j,p}$ are the net input to the k^{th} output unit and j^{th} hidden unit respectively and $y_{j,p}$ is the activation of the j^{th} hidden unit. For the input-to-hidden weights,

$$\frac{\partial E}{\partial v_{ji}} = -\delta_{y_j,p} \cdot D_{ji,p} \tag{3.19}$$

where

$$\delta_{y_j,p} = \sum_{k=1}^{K} \delta_{o_k,p} \cdot w_{kj} \cdot f'(net_{y_j,p})$$

with $f'(net_{y_j,p})$ the derivative of the activation function used for hidden unit Y_j (which equals one in this case). In equation (3.19), $D_{ji,p}$ is computed as (refer to appendix A for the derivations)

$$D_{ji,p} = e^{\rho} \cdot [\ln|z_{i,p}| \cdot \cos(\pi\phi) - \pi\mathcal{I}_i \cdot \sin(\pi\phi)]$$

with

$$\rho = \sum_{i=1}^{I} v_{ji} \ln_e |z_i|$$

and

$$\phi = \sum_{i=1}^{I} v_{ji}\mathcal{I}_i$$

3.7 The Bias Unit

For the equations derived up to now, it was assumed that both the hidden units and output units receive a bias. This section shows that it is sufficient to use a bias only for the output units. This section refers to networks with biases for both hidden and output units as case 1, and networks with biases for only the output layer as case 2. The aim is to show that the learning rules for these two cases for PUNNs are equivalent, which justifies the removal of the hidden unit biases for the remainder of this thesis. For both cases 1 and 2, consider a PUNN consisting of 1 input unit, 2 hidden units and a single output unit. Figure 3.5 illustrates a network of case 1, while a network of case 2 is illustrated in figure 3.6. In figure 3.6 a bias occurs only in the output layer. An extension of PUNNs of case 2 is considered in section 3.7.3, where a 'distortion unit' is included in the product term of the product units as illustrated in figure 3.7.

3.7.1 Case 1 PUNNs

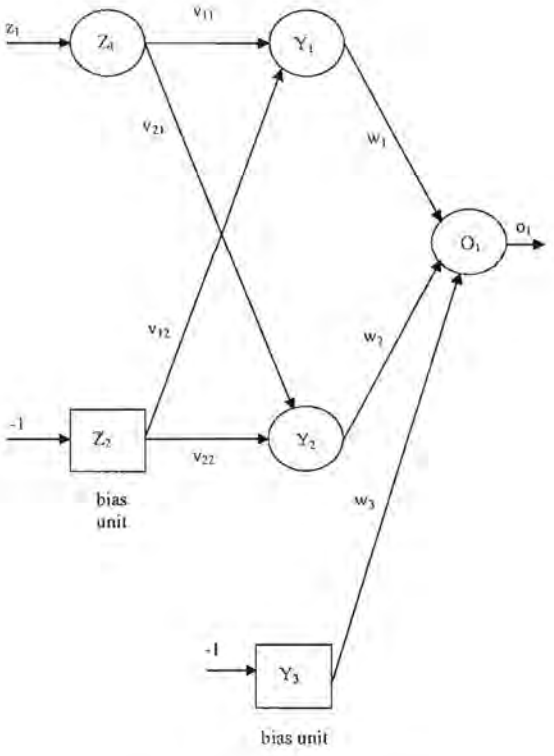


Figure 3.5: Case 1 PUNN

The output of the PUNN in figure 3.5 is,

$$o_1 = \sum_{i=1}^{I+1} y_i \cdot w_i \tag{3.20}$$

$$= y_1 \times w_1 + y_2 \times w_2 + y_3 \times w_3 \tag{3.21}$$

where w_3 is the bias and y_3 is the net input of the bias unit (always -1). Equation (3.21) simplifies to,

$$o_1 = y_1 \times w_1 + y_2 \times w_2 + c_0 \tag{3.22}$$

where $c_0 = -w_3$. We now proceed to compute the activation values y_1 and y_2 for the hidden units Y_1 and Y_2 respectively:

$$y_1 = z_1^{v_{11}} + z_2 \cdot v_{12}$$

$$= z_1^{v_{11}} - v_{12} \tag{3.23}$$

$$y_2 = z_1^{v_{21}} + z_2 \cdot v_{22}$$

$$= z_1^{v_{21}} - v_{22} \tag{3.24}$$

Substitutition of (3.23) and (3.24) in (3.22) yields,

$$\begin{aligned} o_1 &= (z_1^{v_{11}} - v_{12}) \times w_1 + (z_1^{v_{21}} - v_{22}) \times w_2 + c_0 \\ &= (z_1^{v_{11}} + c_1) \times w_1 + (z_1^{v_{21}} + c_2) \times w_2 + c_0 \\ &= z_1^{v_{11}} \cdot w_1 + c_1 \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_2 \cdot w_2 + c_0 \\ &= z_1^{v_{11}} \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_3 \end{aligned} \tag{3.25}$$

where $c_3 = c_0 + c_1 \cdot w_1 + c_2 \cdot w_2$. All the c_i's (i.e. c_0, c_1 and c_2) are basically weights obtained through training and can thus be replaced and trained as a single weight, c_3.

3.7.2 Case 2 PUNNs

Now consider the second case where the bias is removed from the hidden layer. The PUNN in figure 3.6 represents a 1:2:1 network configuration.

The output for the PUNN in figure 3.6 is,

$$o_1 = \sum_{i=1}^{I+1} y_i \cdot w_i \tag{3.26}$$

$$= y_1 \times w_1 + y_2 \times w_2 + y_3 \times w_3 \tag{3.27}$$

Once again, the net input of the bias unit, Y_3, is assumed as -1. Equation (3.27) then simplifies to,

$$o_1 = y_1 \times w_1 + y_2 \times w_2 + c_4 \tag{3.28}$$

where $c_4 = -w_3$. The values for y_1 and y_2 are computed as,

$$y_1 = z_1^{v_{11}} \tag{3.29}$$

$$y_2 = z_1^{v_{21}} \tag{3.30}$$

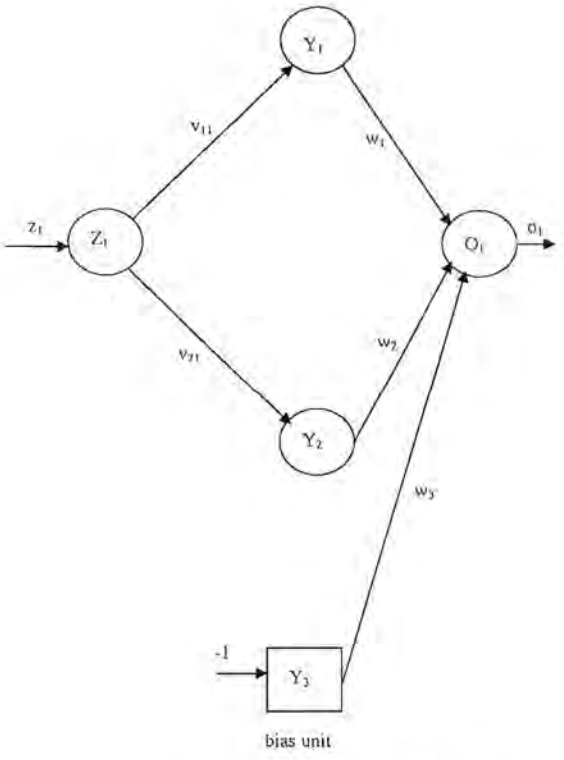


Figure 3.6: Case 2 PUNN

Substitution of (3.29) and (3.30) in (3.28) yields,

$$o_1 = z_1^{v_{11}} \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_4 \tag{3.31}$$

A comparison of equations (3.25) and (3.31) indicates that these two equations are equivalent if and only if c_3 is equal to c_4. In equation (3.25), c_3 is a function of biases and weights that are not dependent on inputs, clearly indicating that c_3 is basically a constant. Thus, instead of learning three different constants (i.e. c_0, c_1 and c_2), only one constant, namely c_4, can be learnt, which will result in equations (3.25) and (3.31) to be equivalent. Note, however, that 'case 1' has more weights and thus more degrees of freedom than 'case 2'. The higher the degrees of freedom, the higher the probability of getting poor results, since an increase in the number of weights causes a corresponding increase in dimensionality of the search space, that will inevitably contain more local minima and plateaus. Generally, case 2 should therefor produce

better results than 'case 1'.

3.7.3 The Distortion Unit

In addition to having biases only for the output units, the PUNNs studied in this thesis were further extended by including a 'distortion' factor in the product term of the product units. The PUNN in figure 3.7 represents a 1:2:1 network configuration with a distortion unit replacing the bias unit in the input layer.

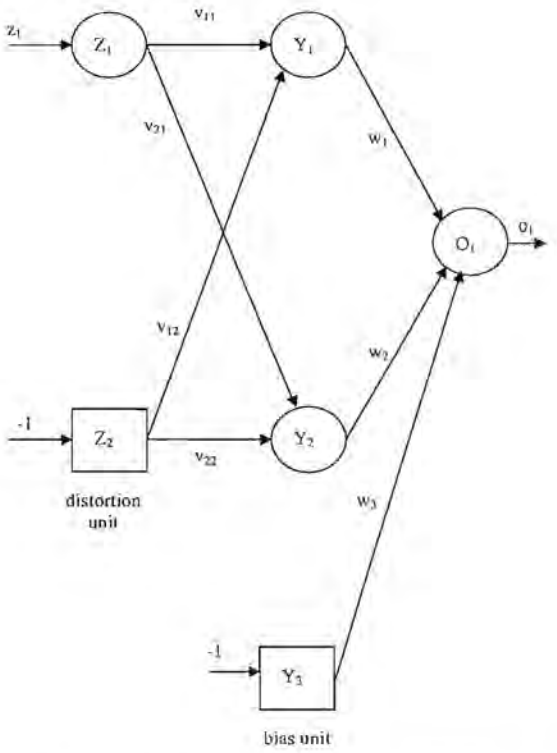


Figure 3.7: PUNNs with a distortion unit

For the product units, the net input signal is therefor calculated as

$$net_{y_j} = \prod_{i=1}^{I+1} z_i^{v_{ji}} \tag{3.32}$$

where $z_{I+1} = -1$ and $v_{j,I+1}$ is the distortion factor. The distortion unit has a constant input of -1. Thus, $z_{I+1}^{v_{j,I+1}}$ simplifies to $(-1)^{v_{j,I+1}}$. This expression, $(-1)^{v_{j,I+1}}$, is not defined for all values of $v_{j,I+1}$ in the real domain. However, in the derivation of the learning equations for PUNNs the calculation of $(-1)^{v_{j,I+1}}$ is performed in the complex domain (refer to equations A.20 to A.22 on page 196). Since, $z_{I+1} = -1$, $ln|z_{I+1}|$ reduces to $ln|-1|$ which equals 0. Thus, the distortion unit does not make any contribution to term ρ, however the value of the weight $v_{j,I+1}$ is added in calculating term ϕ on page 196 when $z_{I+1} = -1$. This shows that $(-1)^{v_{j,I+1}}$ is defined for all negative values of $v_{j,I+1}$. The distortion unit acts to assist in shaping the activation function to more accurately fit the true function as represented by the training data.

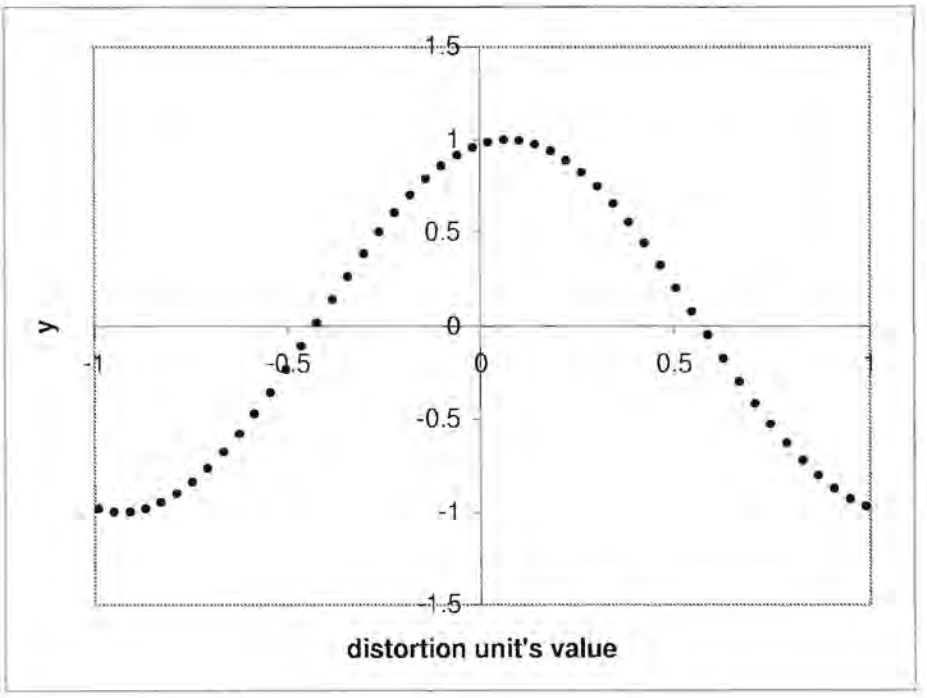


Figure 3.8: Effect of the distortion unit in approximating $f(z) = z^2$

This unit cannot be seen as a bias, since it is not added to the learning rule and plays no role in offsetting the origin of the function, but is rather included in the product. Upon inspection of $net_{y_j} = e^{\rho} \cdot cos(\pi \cdot \phi)$ (from equation (A.45) on page 200), it is observed that the distortion factor's contribution to term ρ, is 0, since in $e^{weight\_of\_distortion\_unit \times ln(|-1|)}$, $ln|-1| = 0$ thus reducing $e^{weight\_of\_distortion\_unit \times ln(|-1|)}$ to 1. The distortion unit thus only contributes to term ϕ. The net effect of the distortion factor on net_{y_j} is thus limited to the contribution of $cos(\pi \cdot \phi)$ to the net input signal. Thus,

$$net_{y_j} = e^{\rho} \cdot \cos(\pi\phi) \tag{3.33}$$

where

$$\rho = \sum_{i=1}^{I} v_{ji} \ln |z_i| \tag{3.34}$$

and

$$\phi = \sum_{i=1}^{I+1} v_{ji} \mathcal{I}_i \tag{3.35}$$

To explain the purpose of the distortion unit, consider approximation of function $f(z) = z^2$ as illustrated in figure 3.9. Further inspection of the distortion term, in the case of function $f(z) = z^2$, $for\ -1 < z < 1$, revealed that the unit mapped a function of the form $cos(3z)$ over the data, re-affirming the fact that this distortion unit effectively assists in shaping the function to better fit the set of training data. The remainder of this thesis assumes PUNNs with a distortion unit.

3.8 Problems with Training of PUNN using Gradient Descent

Gradient descent (GD) is one of the most popular optimization algorithms used to train multilayer neural networks that employ summation units, resulting in the so-called

back-propagation neural network [Werbos 1974]. Gradient descent works best when the search space is relatively smooth, with few local minima or plateaus. In such cases the minima are not too deep and any randomness added to the training process will prevent the network from getting stuck in local minima [Zurada 1992]. This section shows that GD has difficulties in training networks that use product units. These difficulties arise from the increased number of local minima and more convoluted search space due to PUs [Durbin *et al* 1989, Leerink *et al* 1995].

Durbin and Rumelhart have constructed a neural network consisting of 1 hidden product unit and a standard summing output unit to solve the 6-parity problem where the weights were calculated from first principles [Durbin *et al* 1989]. The parity function, when implemented using summation unit neural networks, require as many hidden units as inputs. Leerink *et al* [Leerink *et al* 1995], however, have found that the *back-propagation algorithm could not train* a product unit neural network on the 6-parity problem, due to the following reasons:

- **Incorrect weight initialization:**

  The initial weights of a network is usually computed as small random values in order to use the dynamic range of the sigmoid function and its derivative. Leerink *et al* argued that this is the worst possible choice of initial weights for PUNNs, and suggested that larger initial weights be used instead [Leerink *et al* 1995]. From own experience, back-propagation only manages to train product unit neural networks when the weights are initialized in close proximity of the optimal weight values. The optimal weight values are, however, usually not available. Gradient descent procedures are usually not able to compensate for bad initial values of weights and biases, getting stuck in local minima. To combat the problem of bad initial weights, global optimization algorithms can be used to find

initial weights and GD subsequently applied to train the network, as suggested in [Ismail *et al* 2000].

- **Increased number of local minima:**

  A major drawback of product units is an increased number of local minima, deep ravines and valleys on its error surface. The search space for product units is usually extremely convoluted [Janson *et al* 1993]. This is because the exponent component, v_{ji}, in equation (3.8) can cause large changes in the computation of the total error. Back-propagation by gradient descent therefore frequently gets *trapped in local minima* that it cannot escape from, or becomes *paralyzed* if a local minimum is reached, thus resulting in no adjustment of the weights due to the fact that the error with respect to the current weight is close to zero; the weight vector thus remains the same for the remainder of the training session.

As an example to illustrate the complexity of the search space for product units, and the problems mentioned above, consider the approximation of the function $f(z) = z^3$, with $z \in [-1, 1]$. To approximate this function using PUNNs, one PU is sufficient, resulting in a minimal 1-1-1 architecture. In this case the optimal weight values are $v = 3$ (the input-to-hidden weight) and $w = 1$ (the hidden-to-output weight), where the bias and the distortion are both equal to zero. Figure 3.9 visualizes the search space for $v \in [-1, 4]$ and $w \in [-1, 1.5]$. The error is computed as the mean squared error over 500 randomly generated patterns. Figure 3.9 clearly illustrates 2 local minima, one located at $v = -0.55$ and the other at $v = 1.25$. The global minimum is at $v = 3$. Initial small random weights will cause the network to be trapped in one of the local minima (having very large MSE). Large initial weights may also be a bad choice. Assume an initial weight $v \geq 4$ (or $v \leq -1$). The derivative of the error with respect to v is extremely large due to the steep gradient of the error surface. Consequently, a large

weight update will be made which may cause jumping over the global minimum. The neural network either becomes trapped in a local minimum, or oscillates between the extreme points of the error surface.

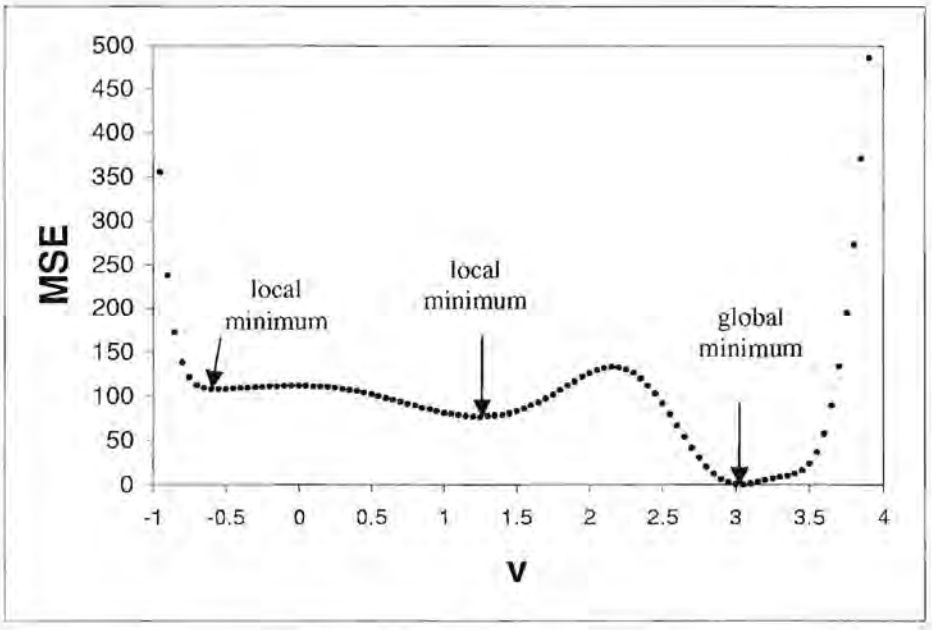


Figure 3.9: MSE values with weight w fixed at 1 for $f(z) = z^2$

Another example to illustrate the numerous local and global minima that occur in the search space of PUNNs is illustrated in figure 3.10, for approximation of the function $f(z_1, z_2) = z_1^2 + z_2^2$.

The function $f(z_1, z_2) = z_1^2 + z_2^2$, with $z_1, z_2 \in [-1, 1]$, can be approximated with a PUNN that contains a minimum of 2 hidden PUs which amounts to 6 weights. A PUNN to approximate $f(z_1, z_2) = z_1^2 + z_2^2$, comprising a 2:2:1 configuration, is represented in figure 3.11. The search space for this PUNN is thus 6-dimensional, making visualization of the error surface very difficult. However, slices of the error surface can be viewed by

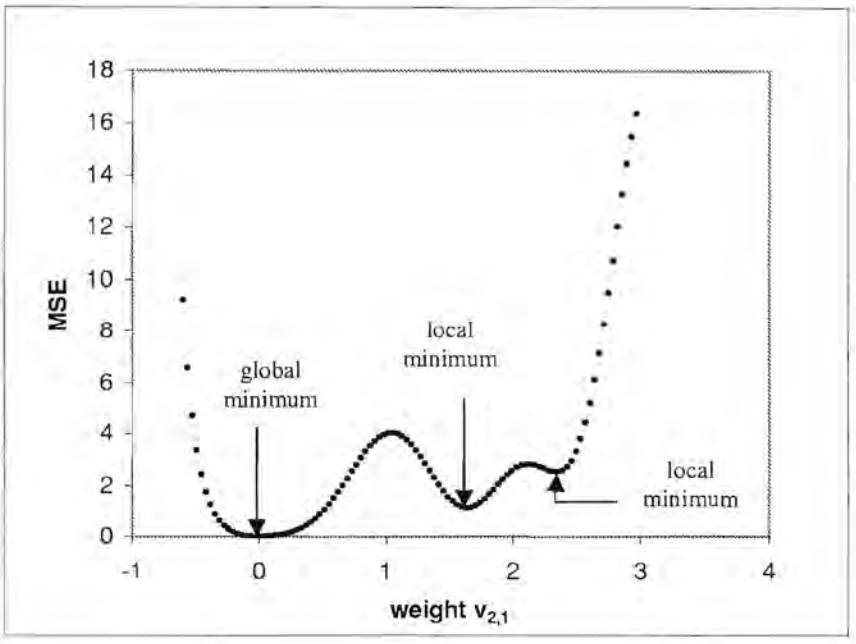


Figure 3.10: Error surface for the straight line between 3 minima, $f(z_1, z_2) = z_1^2 + z_2^2$

fixing most of the weights and varying only one or two of the weights. Figure 3.10 visualizes the search space for all weights fixed, except $v_{21} \in [-1, 4]$. Three minima are illustrated with the global minimum at $v_{21} = 0$. Initial small random weights will cause the network to converge to the global minimum. Large initial weights, however, will cause the network to be trapped in one of the local minima resulting in a large MSE. Initial weights $v_{21} > 3$ or $v_{21} < -1$ will also be a bad choice, since the derivative of the error with respect to v_{21} is extremely large due to the steep gradient of the error surface. A large weight update will be made which may result in overshooting the global minimum. Thus, the neural network becomes trapped in a local minimum, or oscillates between extreme points of the error surface.

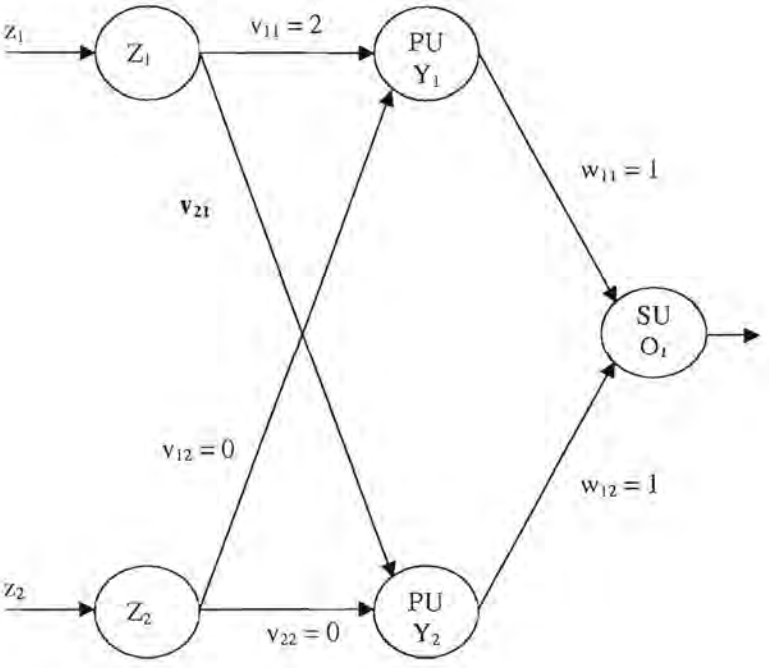


Figure 3.11: PUNN to approximate $f(z_1, z_2) = z_1^2 + z_2^2$

3.9 Conclusion

This chapter discussed the problems encountered when GD is used to train PUNNs. Gradient descent is frequently trapped by local minima that occur in the search space for PUNNs. Local minima are particularly prevalent in networks containing PUs, due to the effect of the exponential terms in the learning equations. These exponential terms cause large weight adjustments that result in the network to be trapped or oscillate between the extreme points. To alleviate these problems, global optimization algorithms should be used to train PUNNs.

The next chapter discusses various global optimization algorithms to train PUNNs.