Simplification of the generalized adaptive neural filter and comparative studies with other nonlinear filters

Henry Steven Hanek

New Jersey Institute of Technology

ABSTRACT

Simplification of the Generalized
Adaptive Neural Filter and Comparative
Studies with Other Nonlinear Filters

by
Henry Steven Hanek

Recently, a new class of adaptive filters called Generalized Adaptive Neural Filters (GANFs) has emerged. They share many characteristics in common with stack filters, and include all stack filters as a subset. The GANFs allow a very efficient hardware implementation once they are trained. However, there are some problems associated with GANFs. Three of these are slow training speeds and the difficulty in choosing a filter structure and neural operator.

This thesis begins with a tutorial on filtering and traces the GANF development up through its origin - the stack filter. After the GANF is covered in reasonable depth, its use as an image processing filter is examined. Its usefulness is determined based on simulation comparisons with other common filters. Also, some problems of GANFs are looked into. A brief study which investigates different types of neural networks and their applicability to GANFs is presented. Finally, some ideas on increasing the speed of the GANF are discussed. While these improvements do not completely solve the GANF's problems, they make a measurable difference and bring the filter closer to reality.

SIMPLIFICATION OF THE GENERALIZED ADAPTIVE NEURAL FILTER AND COMPARATIVE STUDIES WITH OTHER NONLINEAR FILTERS

by
Henry Steven Hanek

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

October 1993

APPROVAL PAGE

Simplification of the Generalized
Adaptive Neural Filter and Comparative
Studies with Other Nonlinear Filters

Henry Steven Hanek

Dr. Nirwan Ansari, Thesis Advisor (date)
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Zoran Siveski, Committee Member (date)
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member (date)
Assistant Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Henry Steven Hanek

Degree: Master of Science in Electrical Engineering

Date: October 1993

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Bachelor of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1992
- Associate in Applied Science in Electronics Engineering Technology,
  County College of Morris, Randolph, NJ, 1988

Major: Electrical Engineering

Presentations and Publications:

H. Hanek and N. Ansari, "Simplification of the Generalized Adaptive Neural Filter," *Proc. 1993 IEEE Regional Conference on Control Systems*, Newark, NJ, August 13-14, 1993, pp. 211-214.

H. Hanek, N. Ansari, and Z. Zhang, "Comparative Study on the Generalized Adaptive Neural Filter with other Nonlinear Filters," *Proceedings of ICASSP-93*, Vol.I, April 27-30, 1993, Minneapolis, Minnesota, pp. 649-652.

ACKNOWLEDGMENT

I would like to especially thank my thesis advisor, Dr. Nirwan Ansari for all he has done. His wisdom and caring attitude have done more for me than I can express on this page. Dr. Ansari believed before he saw anything. I also wish to thank him and Tan Hwang Beng for convincing me to get a Masters degree.

Next, I wish to thank my committee members, Dr. Zoran Siveski and Dr. Edwin Hou for offering meaningful suggestions and opening my mind to new directions. I thank Dr. Ali N. Akansu for the most brilliant explanation of DSP I have ever experienced. I thank Dr. Yeheskel Bar-Ness for improving my understanding of mathematics, and Dr. Timothy Chang for resurrecting my interest in control systems. I also thank the other teachers and staff who have been a part of my life at NJIT.

I thank my friends at school and Toastmasters who have been very supportive. I especially mention Z.Z. Zhang, A. Arulambalam, J. Hughes, A. Shah, T.H. Beng and T. Nalepka.

Finally, I wish to thank my father for providing me with an interest in all areas of technology and physics, and I thank the rest of my family for putting up with me over the past five years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
BACKGROUND INFORMATION

This thesis talks about a new class of nonlinear digital filters called *Generalized Adaptive Neural Filters* (GANFs) [1]. The latter part of this thesis will examine the GANF in depth and also discuss several new developments by the author. However, at the beginning it is important to focus on why and how the GANF came about. In order to accomplish this, we will begin with a look at the purpose of filtering in general.

1.1 Introduction to Linear Filtering

In electrical engineering, all signals take the form of a voltage, current, or resistance. In most cases, there is information embedded in these signals that has some type of meaning. The information may represent a physical quantity such as pressure (or sound), acceleration, or luminosity. On the other hand, the signal may convey control or timing information, as in a digital circuit. However, because the sensors and other devices we use are not perfect, other signal components are introduced which do not convey information. Also, through transmission or external physical effects, signals may be corrupted by noise. As a result, in the real world we always deal with a desired signal that is in some way corrupted by noise. In many cases we try to maximize the *Signal-to-Noise Ratio* (SNR) of the system. This allows the greatest system performance, as we then deal with more accurate information. For example, a high SNR would allow the output of a transducer to be related in a more definite way to the physical parameter being measured.

In order to improve the SNR, we can input the noisy signal to a device called a filter. The filter will provide an output which has a higher SNR than the input. In many cases, a linear filter can be used to accomplish this. A linear filter is a device whose output is a linear function of its inputs. In this thesis, all of the signals considered will be discrete in nature. They are assumed to be obtained by sampling and quantizing an analog signal in accordance with the sampling theorem. As a result, the output of a linear digital filter is simply a weighted summation of different samples of the input signal. Linear filters are very nice to work with. The mathematics necessary for their analysis is quite straightforward. However, there are situations where linear filtering does not adequately accomplish its objective.

When designing a filter, we try to minimize the mean squared error between the filter output $y(n)$ and a desired response $s(n)$ (which is the clean signal). This is another way of saying that we want to minimize the variance of the error between the filter output and what we would like to see:

$$\sigma_\epsilon^2 = E\{[s(n) - y(n)]^2\} \tag{1.1}$$

The filter output $y(n)$ is obtained by some operation on $r(n)$, the input signal. The variance of the error can be minimized by making the filter output, $y(n)$ equal the conditional mean of $s(n)$ given the sequence $r(n)$ [2]. That is, the optimal filter output can be described by the following equation:

$$y(n) = E[s(n)|r(n), \forall n] \tag{1.2}$$

If $s(n)$ and $r(n)$ are jointly Gaussian, then the solution of eq. (1.2) is a linear function. If the processes are not jointly Gaussian, then eq. (1.2) is not easy to solve. In these cases, the optimal filter cannot be described by a linear function [2].

1.2 Introduction to Nonlinear Filtering

Linear filters are most useful for additive Gaussian noise only and tend to mask out high frequency components in signals [3]. When applied to images, a linear filter will blur the edges and other high contrast areas which are needed for image clarity. Also, in many cases the noise encountered may be non-Gaussian, non-additive and may also be somehow related to the desired signal [3]. As a result, nonlinear filters must often be used to achieve satisfactory results. Although they do a better job in these cases, nonlinear filters introduce some new problems. Choosing a non-linear filtering function involves a complicated mathematical analysis which does not work well in practice [3]. Also, implementing a nonlinear filter may be difficult. It is sometimes difficult to design analog hardware to create a non-linear function.

CHAPTER 2

STACK FILTERS

2.1 Background Information

At this point, the need for easy to use nonlinear filters can be seen. This involves both ease of design (deciding on a nonlinear function to use) and ease of implementation (actually building something to accomplish the filtering). While all digital filters can be implemented in software, it is sometimes desirable to build fast, dedicated hardware to do the filtering. In 1986, a nonlinear filtering structure was developed to accomplish this. The filters were called "Stack Filters" and enabled a large group of nonlinear filters to be easily implemented with *Very Large Scale Integration* (VLSI) [3]. This stack filtering structure allows the construction of many nonlinear filters in a compact, modular form. The dedicated hardware will also permit much faster filtering as opposed to an algorithm in, say, a *Digital Signal Processor* (DSP) chip.

In order to understand the structure of a stack filter, it will first be necessary to introduce some definitions.

Definition 2.1 *Suppose we are given two vectors* $\mathbf{x}^T = [x_1 \ \ x_2 \ \ \cdots \ \ x_N]$ *and* $\mathbf{y}^T = [y_1 \ \ y_2 \ \ \cdots \ \ y_N]$. *Then, if the relationship* $\mathbf{x} \leq \mathbf{y}$ *implies* $x_i \leq y_i \ \ \forall i$, *the row vector* $\mathbf{x}^T$ *is said to stack on the vector* $\mathbf{y}^T$.

If the components x_i and $y_i \in \{0, 1\} \ \ \forall i$, then the vectors $\mathbf{x}$ and $\mathbf{y}$ are binary vectors. In this case, the condition $\mathbf{x} \leq \mathbf{y}$ means that $x_i = 1$ implies $y_i = 1$. When $x_i = 0$, then y_i can equal 0 or 1.

Next, let us consider a binary vector as an input to a Boolean function. Such a function would generate an output $\in \{0, 1\}$ for every possible input. If the length of the binary input vector is N, then there are 2^N possible inputs and 2^{2^N} possible assignments to the Boolean function.

Definition 2.2 *Suppose we input a binary vector* x *to a Boolean function* $\mathcal{B}$ *and generate an output* u. *We input a binary vector* y *to the same Boolean function and generate an output* v. *The function* $\mathcal{B}$ *is said to possess the stacking property if and only if* $u \leq v$ *whenever* $\mathrm{x} \leq \mathrm{y}$.

There are many Boolean functions which possess the stacking property. These functions are called positive Boolean functions, and can always be expressed in minimum sum-of-products (MSP) form with no complements of any of the variables [3]. For 3 inputs, there are exactly 20 positive Boolean functions. For 5 inputs there are 7581, and for 7 inputs, there are greater than 2^{35} positive Boolean functions [3]. In general, there are always greater than $2^{2^{B/2}}$ positive Boolean functions of B variables [4].

2.2 Structure of Stack Filters

With these definitions covered, we can now examine the structure of stack filters. Figure 2.1 shows an overview of the stack filtering process. First of all, our filter operates on a sequence of numbers fed in at its input. These numbers are all integers which are part of the set $\{0, 1, 2, ..., M-1\}$. The filter cannot process all of the information in such a signal, so only a finite window of elements is considered. Here, we choose an odd number for the length of the window and really do a smoothing operation on the data [5]. For each input sample that the window is centered on, we generate a corresponding output integer. This output also belongs to the set $\{0, 1, 2, ..., M-1\}$. Of course, for each successive filtering operation, the window is moved by one sample to the right (forward in time).

Examining the operation depicted in Figure 2.1 in more detail, we generate the diagram shown in Figure 2.2. This is a pictoral which shows the low level operation

. . . 3 5 1 . . .
STACK FILTER
. . . 3 . . .

Figure 2.1 The most general function of a stack filter.

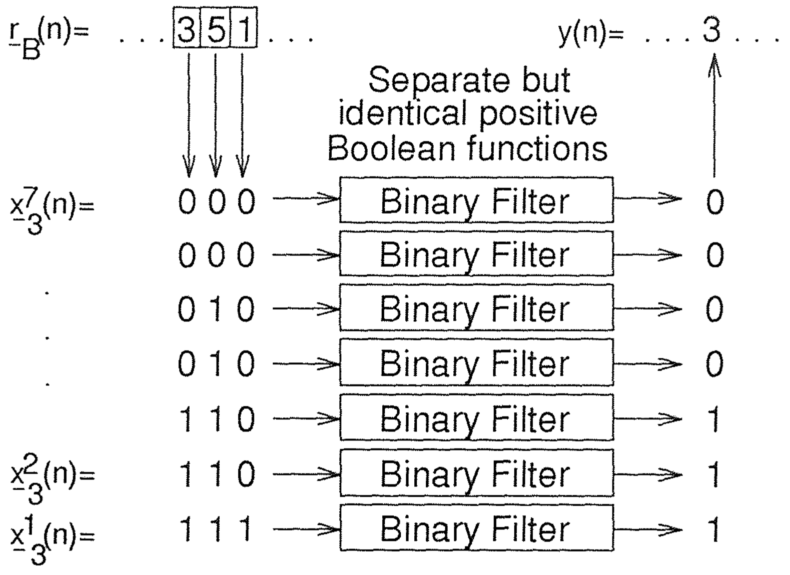

$\underline{r}_B(n)$= . . . 3 5 1 . . .
y(n)= . . . 3 . . .
Separate but identical positive Boolean functions
$\underline{x}_3^7(n)$= 0 0 0
0 0 0
0 1 0
0 1 0
1 1 0
$\underline{x}_3^2(n)$= 1 1 0
$\underline{x}_3^1(n)$= 1 1 1
Binary Filter
0
0
0
0
1
1
1

Figure 2.2 Stack filter example, with window width $B = 3$ and $M = 8$.

of a very simple stack filter (window width 3, and M=8). We have an input vector $\mathbf{r}_B(n)$ of length B, which is composed of elements in the range $\{0, 1, 2, \ldots, M-1\}$ as follows:

$$\mathbf{r}_B^T(n) = [r(n - \frac{B-1}{2}) \cdots r(n) \cdots r(n + \frac{B-1}{2})] \tag{2.1}$$

The vector $\mathbf{r}_B(n)$ is next broken down uniquely into $(M-1)$ binary vectors of length B. This is accomplished by a threshold decomposition operation. It is defined according to the following relation:

$$\mathbf{x}_B^i(n) = T^i[\mathbf{r}_B(n)], \tag{2.2}$$

where

$$T^i[\mathbf{r}_B(n)] = [T^i[r(n - \frac{B-1}{2})] \cdots T^i[r(n)] \cdots T^i[r(n + \frac{B-1}{2})], \tag{2.3}$$

and

$$T^i[x] \triangleq \begin{cases} 1, & \text{if } x \geq i \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

The stack filter always makes use of this threshold decomposition property. It should be noted that adding any column in the stack produces the integer from which that column was derived:

$$\mathbf{r}_B(n) = \sum_{i=1}^{M-1} T^i[\mathbf{r}_B(n)]. \tag{2.5}$$

As a result, the threshold decomposition is unique, and the results sum to produce the original integers. Also, if the levels are arranged as shown in the diagram (level 1 on the bottom up through level $(M-1)$ on top), the binary vectors $\mathbf{x}_B^i(n)$ stack on top of each other. In other words,

$$\mathbf{x}_B^i(n) \leq \mathbf{x}_B^j(n) \quad \forall \quad 1 \leq j \leq i \leq M-1. \tag{2.6}$$

This means that the binary input vectors possess the stacking property.

At this point, we have $(M-1)$ binary vectors of length B. Each vector is then used as an input to a separate Boolean function on each level. We have a total of $(M-1)$ such Boolean functions operating on B binary inputs and producing $(M-1)$ binary outputs. For a stack filter, all of the $(M-1)$ Boolean functions are the same. However, since the inputs (the vectors $\mathrm{x}_B^i(n)$) are not all the same, the Boolean function outputs may be different. In addition, the Boolean function is required to be a positive Boolean function. The reason for this will become clear shortly.

After processing things thus far, we are left with $(M-1)$ binary outputs from the Boolean functions on each level. To find the integer output $y(n)$, we must add all of the outputs of the Boolean functions. This can always be done, but there is a simpler way to implement this. Note that the inputs to the Boolean functions stack, and the Boolean functions are positive. Let $\mathrm{x}_B^i(n)$ and $\mathrm{x}_B^j(n)$ be the binary input vectors of two separate but identical positive Boolean functions $\mathcal{B}$. The outputs of these two functions are $y^i(n)$ and $y^j(n)$, respectively. Then, if $i > j$ the outputs must satisfy $y^i(n) \leq y^j(n)$. That is, by Definition 2.2, it can be seen that the level outputs will stack. This means that there will always be a column of 0's above a level output of 0 and a column of 1's below a level output of 1. There is only one point where a transition between 0 and 1 can occur. Let the 1-to-0 transition occur between levels K and $(K+1)$. Since

$$y(n) = \sum_{i=1}^{M-1} y^i(n) = \sum_{i=1}^{K} y^i(n) + \sum_{K+1}^{M-1} y^i(n), \tag{2.7}$$

$$y(n) = \sum_{i=1}^{K} 1 + \sum_{K+1}^{M-1} 0, \tag{2.8}$$

$$y(n) = K, \tag{2.9}$$

the output $y(n)$ will be equal to the greatest level number which has an output of 1. As a result, the filter output can be obtained through a binary search of the Boolean function outputs to determine where the 1 to 0 transition occurs. This enables a savings in VLSI chip area [3].

It is important to realize that the threshold decomposition also results in another important property when rank order operators are considered [3]. Note that, in general, a rank order filter consists of a hierarchy of *MIN* and *MAX* operations on subsets of elements in the window [6]. Because of the threshold decomposition, the operation of a stack filter defined in terms of *MIN* and *MAX* operations will translate directly into equivalent binary filter functions. In other words, when a rank order operation is applied to the window of integers, the same results will be achieved if this operation is applied to the binary vectors at each level in the filter. When the *MIN* and *MAX* operations are applied to binary numbers, they become the logical *AND* and *OR* operations, respectively [6]. This is known as the weak superposition property and is formally described as follows:

$$S_f[\mathrm{r}_B(n)] = S_f[\sum_{i=1}^{M-1} \mathrm{x}_B^i(n)] = \sum_{i=1}^{M-1} S_f[\mathrm{x}_B^i(n)], \tag{2.10}$$

where $S_f[\cdot]$ is the stack filter operator which, of course, always implements a rank order filter.

2.3 Configuring a Stack Filter

The Boolean function used on each level really defines the operation of the stack filter. By selecting an appropriate Boolean function, many types of nonlinear filters can be implemented. Included in this set are all rank order filters and all morphological filters [3]. For example, a median filter for window size 3 can be achieved using a Boolean function described by

$$y = x_1x_2 + x_2x_3 + x_1x_3. \tag{2.11}$$

As pointed out previously, there are a large number of positive Boolean functions, and therefore a large number of stack filters. The next question is how to pick a positive Boolean function suitable for a given filtering problem. There are many methods which can be used to accomplish this, and this thesis will discuss only a few of them.

2.3.1 Coyle/Lin Method

Filters can be "optimized" in a number of different ways depending on the criteria used. One common measure of performance is the mean absolute error (MAE) of the output. The MAE for a stack filter, $S_f(\cdot)$, can be represented by [6]

$$B(S_f) = E[|s(n) - S_f(\mathbf{r}_B(n))|]. \tag{2.12}$$

where $\mathbf{r}_B(n)$ is the windowed input and $s(n)$ is the desired output.

However, because of the threshold decomposition on the input side,

$$\mathbf{r}_B(n) = \sum_{i=1}^{M-1} \mathbf{x}_B^i(n). \tag{2.13}$$

Also, because of the weak superposition property described earlier,

$$S_f(\mathbf{r}_B(n)) = S_f(\sum_{i-1}^{M-1} \mathbf{x}_B^i(n)) = \sum_{i=1}^{M-1} S_f(\mathbf{x}_B^i(n)). \tag{2.14}$$

In addition,

$$s(n) = \sum_{i=1}^{M-1} s^i(n), \tag{2.15}$$

where $s^i(n) = \mathcal{T}^i[s(n)]$, and $\mathcal{T}^i[x]$ is as defined in Eq. (2.4). So

$$B(S_f) = E[|\sum_{i=1}^{M-1} s^i(n) - \sum_{i=1}^{M-1} S_f(\mathbf{x}_B^i(n))|], \tag{2.16}$$

$$B(S_f) = E[|\sum_{i=1}^{M-1} (s^i(n) - S_f(\mathbf{x}_B^i(n)))|]. \tag{2.17}$$

This can be represented by the following expression:

$$B(S_f) = E[|\sum_{i=1}^{a} 0 + \sum_{i=a+1}^{b} (\pm 1) + \sum_{i=b+1}^{M-1} 0|], \tag{2.18}$$

where $a = \min[s(n), S_f(\mathbf{r}_B(n))]$ and $b = \max[s(n), S_f(\mathbf{r}_B(n))]$.

The terms in the second summation will all be $+1$ or all be -1 since $s^i(n)$ and $S_f(\mathrm{x}_B^i(n))$ are constant for $a+1 \leq i \leq b$. This fact is due to the stacking property. As a result,

$$B(S_f) = E[|k \sum_{i=1}^{M-1} |s^i(n) - S_f(\mathrm{x}_B^i(n))||], \tag{2.19}$$

where $k = 1$ if $s(n) > S_f(\mathrm{r}_B(n))$ and $k = -1$ if $s(n) < S_f(\mathrm{r}_B(n))$.

Therefore,

$$B(S_f) = E[\sum_{i=1}^{M-1} |s^i(n) - S_f(\mathrm{x}_B^i(n))|], \tag{2.20}$$

$$B(S_f) = \sum_{i=1}^{M-1} E[|s^i(n) - S_f(\mathrm{x}_B^i(n))|]. \tag{2.21}$$

The MAE is equal to the sum of the MAEs on each level. Therefore, the MAE of the stack filter can be minimized by minimizing the MAE on each level. In order to accomplish this, the following cost function can be used:

$$\begin{aligned} COST = {} & C(desired = 0, actual = 0 \mid \mathrm{x}_B^i(n))P(0,0 \mid \mathrm{x}_B^i(n)) \\ & +C(desired = 1, actual = 0 \mid \mathrm{x}_B^i(n))P(1,0 \mid \mathrm{x}_B^i(n)) \\ & +C(desired = 0, actual = 1 \mid \mathrm{x}_B^i(n))P(0,1 \mid \mathrm{x}_B^i(n)) \\ & +C(desired = 1, actual = 1 \mid \mathrm{x}_B^i(n))P(1,1 \mid \mathrm{x}_B^i(n)), \end{aligned} \tag{2.22}$$

where $C(\cdot)$ is the cost of a certain action by the binary filter and $P(\cdot)$ is the probability of that action.

To simplify some further analysis, new notation will be introduced:

<u>Event A</u> means that the desired level output equals some value.

<u>Event B</u> means that the actual level output equals some value.

<u>Event C</u> means that the input pattern on level i is $\mathrm{x}_B^i(n)$.

We can now deal with probabilities of the form, $P(AB|C)$. Note that

$$P(AB|C) = \frac{P(ABC)}{P(C)} = \frac{P(ACB)}{P(C)} = \frac{P(A|CB)P(CB)}{P(C)}, \tag{2.23}$$

$$P(AB|C) = \frac{P(A|CB)P(B|C)P(C)}{P(C)}. \tag{2.24}$$

If we assume that event B is statistically independent of events AC and C,

$$P(A|CB) = P(A|C), \tag{2.25}$$

and

$$P(AB|C) = P(A|C)P(B|C). \tag{2.26}$$

If we apply this to eq. (2.22), we get the following result:

$$\begin{aligned} C(\mathcal{F}|\overline{W}, l) = C_l(\overline{W}, 0, 0)\pi_l(0|\overline{W}, l)P_{\mathcal{F}}(0|\overline{W}) + \\ C_l(\overline{W}, 1, 0)\pi_l(1|\overline{W}, l)P_{\mathcal{F}}(0|\overline{W}) + \\ C_l(\overline{W}, 0, 1)\pi_l(0|\overline{W}, l)P_{\mathcal{F}}(1|\overline{W}) + \\ C_l(\overline{W}, 1, 1)\pi_l(1|\overline{W}, l)P_{\mathcal{F}}(1|\overline{W}). \end{aligned} \tag{2.27}$$

Here, $C(\mathcal{F}|\overline{W}, l)$ is the total cost incurred by using filter $\mathcal{F}$ on level l to process input vector $\overline{W}$. $C_l(\overline{W}, i, j)$ is the cost of the binary filter on level l producing an output of j when the desired output for this filter is i. $\pi_l(y|\overline{W}, l)$ is the probability that the filter output on level l is y, given the input vector $\overline{W}$. Finally, $P_{\mathcal{F}}(k|\overline{W})$ is the probability that the desired output on level l is k given an input vector $\overline{W}$.

To represent the cost solely in terms of the Boolean function used, we must average eq. (2.27) over all possible input vectors for given signal and noise statistics. This produces

$$C(\mathcal{F}|l) = \sum_{\overline{W} \in Q_W} C(\mathcal{F}|\overline{W}, l)\pi_l(\overline{W}), \tag{2.28}$$

where Q_W is the set of possible binary patterns $\overline{W}$ and $\pi_l(\overline{W})$ is the probability of pattern $\overline{W}$ being observed on level l.

2.3.2 Ansari-Lin Method

The Coyle/Lin method for optimizing a stack filter through linear programming works great in theory. However, the number of constraints involved in the

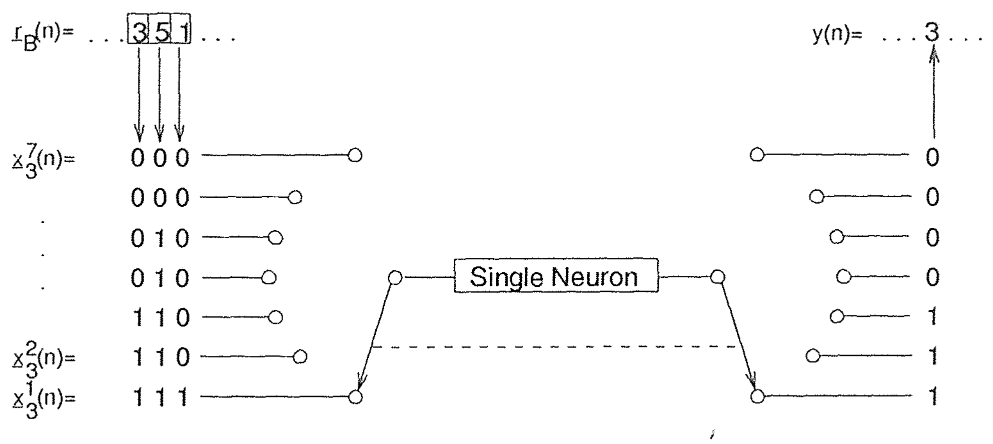


Figure 2.3 Single neuron for adaptive stack filtering.

optimization increases with order $B2^B$ where B is the window size [6]. Note that this is greater than an exponential increase. When the window sizes start to increase, the Coyle/Lin method for optimization gets out of control. As an example, for a window of length 16 the number of constraints is greater than 1 million [7].

To avoid this problem, a new method of stack filter optimization was created by Ansari, et al. [8]. This method involves using a single neuron to implement the positive Boolean functions required in the stack filter. The basic structure of the filter is shown in Figure 2.3. Here, everything works as in the stack filter with the exception of the positive Boolean function. Each separate positive Boolean function is replaced by a single neuron which rides up the levels to provide separate level outputs. In other words, the single neuron looks at the binary vector at a certain level and produces a binary output. Then it moves up a level and does the same thing. This is done for all $(M-1)$ levels. The stack filter output is taken to equal the level at which the output changes from 1 to 0. A detail of the neuron is shown in Figure 2.4. It consists of a summation node with $(B+1)$ weighted inputs and a non-linear threshold function. One of the inputs is permanently assigned the value of one; the others come from the binary input vector on a certain level. Because of the threshold function, the output is binary. To implement a classification, the

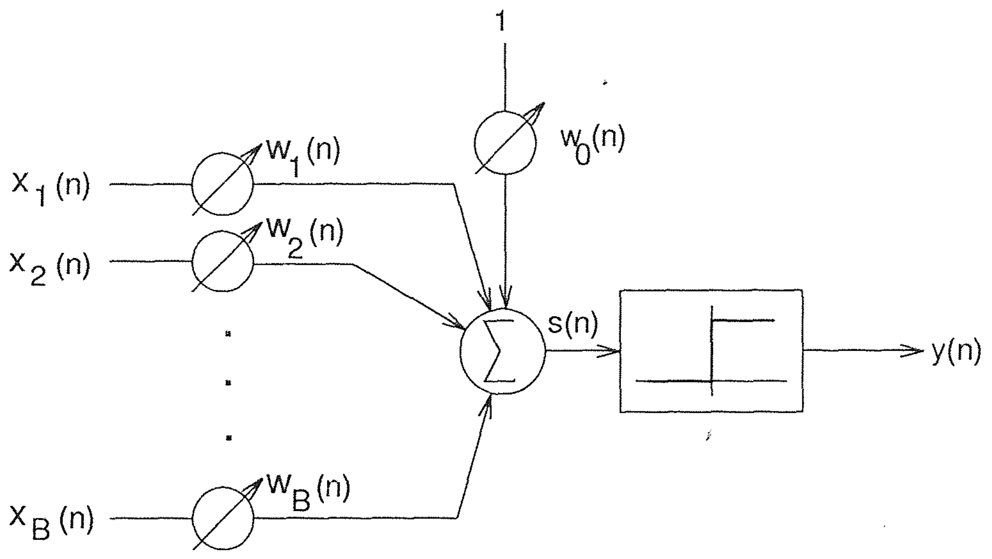


Figure 2.4 Single neuron.

neuron first generates an analog output:

$$s(n) = \mathrm{w}^T(n)\mathrm{x}(n), \tag{2.29}$$

where $\mathrm{w}^T(n) = [w_0(n) \;\; w_1(n) \;\; w_2(n) \;\; \cdots \;\; w_B(n)]$
and $\mathrm{x}^T(n) = [1 \;\; x_1(n) \;\; x_2(n) \;\; \cdots \;\; x_B(n)]$.

This analog output is then processed by a nonlinear function. In this case, the nonlinear activation function is the signum function:

$$y(n) = \frac{\mathrm{sgn}[s(n)] + 1}{2}, \tag{2.30}$$

$$y(n) = \frac{\mathrm{sgn}[\mathrm{w}^T(n)\mathrm{x}(n)] + 1}{2}. \tag{2.31}$$

Here, $\mathrm{x}(n)$ is a binary input vector with the first element set to 1. The other B elements are the binary values produced by the threshold decomposition on the window of B integers. The weight vector $\mathrm{w}(n)$ consists of $(B + 1)$ floating point numbers. By taking the inner product $\mathrm{w}^T(n)\mathrm{x}(n)$, a continuous (analog) output, $s(n)$ is produced. This output is hard-limited to generate the binary output $y(n) \in \{0, 1\}$. By adjusting the weights, different classifications of the binary input vectors can be achieved.

The weights can be adjusted in many ways. Ansari *et al* used both LMS and perceptron learning with good results [8]. Basically, a signal for which the desired response is known is processed by the filter. For each sample processed, an error is generated which is used to update the weights. For LMS, the error is analog,

$$\epsilon_{LMS}(n) = d(n) - s(n), \tag{2.32}$$

$$\epsilon_{LMS}(n) = d(n) - \mathrm{w}^T(n)\mathrm{x}(n), \tag{2.33}$$

and adaptation attempts to minimize this error for future samples. This is accomplished by updating the weights according to the relation

$$\mathrm{w}(n+1) = \mathrm{w}(n) - \frac{1}{2}\mu\nabla(n), \tag{2.34}$$

where $\nabla(n) = \frac{\partial J(n)}{\partial \mathrm{w}(n)}$, and

$$J(n) = |\epsilon_{LMS}|^2 = [d(n) - \mathrm{w}^T(n)\mathrm{x}(n)]^2. \tag{2.35}$$

This results in

$$\nabla(n) = \frac{\partial J(n)}{\partial \mathrm{w}(n)} = -2\mathrm{x}(n)d(n) + 2\mathrm{x}(n)\mathrm{x}^T(n)\mathrm{w}(n), \tag{2.36}$$

$$\nabla(n) = \frac{\partial J(n)}{\partial \mathrm{w}(n)} = -2\mathrm{x}(n)\epsilon_{LMS}(n). \tag{2.37}$$

So,

$$\mathrm{w}(n+1) = \mathrm{w}(n) + \mu\mathrm{x}(n)\epsilon_{LMS}(n). \tag{2.38}$$

The perceptron learning scheme is similar, but uses a discrete error,

$$\epsilon_{PTRON}(n) = d(n) - g(n), \tag{2.39}$$

where $g(n) = \mathrm{sgn}[\mathrm{w}^T\mathrm{x}(n)]$. Therefore,

$$\epsilon_{PTRON}(n) = d(n) - sgn[\mathrm{w}^T(n)\mathrm{x}(n)]. \tag{2.40}$$

With perceptron learning, the folowing criterion function is used:

$$J(n) = -\epsilon_{PTRON}(n)\mathrm{w}^T(n)\mathrm{x}(n). \tag{2.41}$$

Then

$$\nabla(n) = \frac{\partial J(n)}{\partial \mathrm{w}(n)} = -\epsilon_{PTRON}(n)\mathrm{x}(n). \tag{2.42}$$

Using eq. (2.34),

$$\mathrm{w}(n+1) = \mathrm{w}(n) + \mu\mathrm{x}(\frac{1}{2}\epsilon_{PTRON}). \tag{2.43}$$

Note that ϵ_{PTRON} is either $+2, 0$, or -2. Because of this, the update relation has the effect of moving the weight vector either toward a misclassified sample or away from it. If $\epsilon_{PTRON} = +2$, then the inner product $\mathrm{w}^T(n)\mathrm{x}$ is negative when it should be positive. In this case, the update formula gives us

$$\mathrm{w}(n+1) = \mathrm{w}(n) + \mu\mathrm{x}(n), \tag{2.44}$$

so that the inner product using the new weight vector is closer to the desired solution:

$$\mathrm{w}^T(n+1)\mathrm{x}(n) > \mathrm{w}^T(n)\mathrm{x}(n). \tag{2.45}$$

Similarly, if $\epsilon_{PTRON} = -2$, the weight vector is moved to provide an inner product which is less than the previous one. No change to the weight vector is made if the classification at time n was correct.

For both methods, however, there is no guarantee that the neuron will implement a positive Boolean function. In order to achieve this, negative weights can be set to zero. Also, it should be pointed out that a single neuron may not be able to implement all possible positive Boolean functions [3]. This means that the Ansari-Lin method may not find the optimal stack filter among all positive Boolean functions. It will, however, find the optimal stack filter among all threshold functions. Threshold functions are those which can be expressed in the form [9]

$$f = \begin{cases} 1, & \text{if} \quad w_1x_1 + w_2x_2 + \cdots + w_nx_n \geq t, \\ 0, & \text{if} \quad w_1x_1 + w_2x_2 + \cdots + w_nx_n < t, \end{cases} \tag{2.46}$$

where x_i are binary inputs, w_i are weights and t is a threshold. Since the Boolean functions which can be implemented by a threshold gate are a subset of positive

Boolean functions, this method produces the best filter among a subset of possible stack filters.

CHAPTER 3
THE GENERALIZED ADAPTIVE NEURAL FILTER

3.1 Description of the GANF

With the introduction of a single neuron to configure stack filters, the need for a more generalized filtering structure became apparent. At this point, the stack filtering structure was further extended by Ansari *et al* to form a new class of filters [1] [10]. These new filters are called Generalized Adaptive Neural Filters (GANFs) and they include all stack filters as a subset.

The GANF is depicted in Figure 3.1. Its structure resembles that of a stack filter, but things have been changed in two important ways. First of all, the identical Boolean functions on each level of the stack filter are replaced by independent neural operators. Each of these neural elements can be trained to implement a Boolean function. In the most general GANF, each of the neural functions may be different, and they may not necessarily be positive Boolean functions. The second change involves the inputs to the neural functions. The filter input is threshold decomposed exactly as in the stack filter, producing $(M - 1)$ levels of binary input vectors. However, more than one binary input vector may be used as input to a certain level. That is, the neural operator on a certain level may look at the binary input vectors on adjacent levels in addition to the binary input vector on its own level. We will now describe this in more detail.

As before, we have the binary vectors on each level produced by a threshold decomposition of the integer input sequence:

$$\mathrm{x}_B^i(n) = T^i[\mathrm{r}_B(n)], \tag{3.1}$$

where

$$T^i[\mathrm{r}_B(n)] = [T^i[r_1(n)] \;\; T^i[r_2(n)] \cdots T^i[r_B(n)]]. \tag{3.2}$$

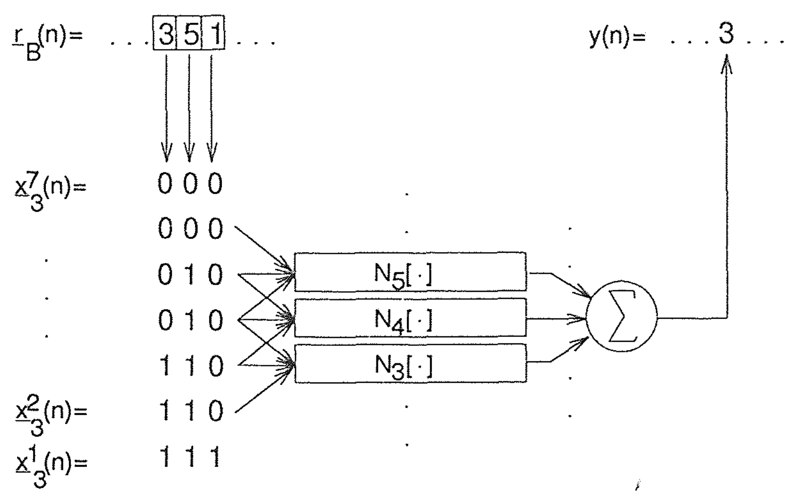


Figure 3.1 Generalized adaptive neural filter.

Here, $r_k(n)$ are elements in the window vector $\mathbf{r}_B(n)$ and

$$T^i[x] \triangleq \begin{cases} 1, & \text{if } x \geq i \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

Now, however, the filter on each level is a neural function $N_i[\cdot]$ and processes a binary input matrix,

$$\mathbf{X}_{I,B}^i \triangleq \begin{bmatrix} T^{i+I}[\mathbf{r}_B(n)] \\ \cdots \\ \cdots \\ \cdots \\ T^i[\mathbf{r}_B(n)] \\ \cdots \\ \cdots \\ \cdots \\ T^{i-I}[\mathbf{r}_B(n)] \end{bmatrix}. \tag{3.4}$$

The total filter output is the sum of the outputs of all of the neural operators. Each neural operator receives as input a $(2I+1) \times B$ binary matrix, where I equals the number of adjacent levels fed in. The GANF output can be described by

$$y(n) = F_{I,B}[\mathbf{r}_B(n)] = \sum_{i=1}^{M-1} N_i[\mathbf{X}_{I,B}^i(n)]. \tag{3.5}$$

Note that the GANF reduces to a stack filter if the following conditions hold:

1. No adjacent level inputs are used (I=0).

2. The neural functions are all identical positive Boolean functions.

We will show that if optimized properly, the GANF will always perform better, or in the worst case, as good as an optimal stack filter [10].

3.2 MAE Criterion

Ansari, *et.al.* proved a few important things about the GANF [1]. The first of these involves the mean absolute error (MAE).

Theorem 3.1 *The MAE of an optimal generalized adaptive neural filter using appropriate neural functions is less than or equal to that of an optimal stack filter.*

This can be shown by first expressing the MAE of the GANF as

$$C[F_{I,B}(\cdot)] \triangleq (\text{MAE of GANF}), \tag{3.6}$$

$$C[F_{I,B}(\cdot)] \triangleq E[|s(n) - y(n)|] = E[|\sum_{i=1}^{M-1}[s^i(n) - y^i(n)]|], \tag{3.7}$$

where $s^i(n) = T^i[s(n)]$ and $y^i(n) = N_i[\mathbf{X}^i_{I,B}(n)]$.

Now, if and only if the outputs, $y^i(n)$, possess the stacking property,

$$C[F_{I,B}(\cdot)] = \sum_{i=1}^{M-1} E[|s^i(n) - y^i(n)|]. \tag{3.8}$$

The reason for this is the same as discussed in section 2.3.1. Note that the MAE in eq. (3.8) is that of a stack filter. Therefore, if we represent this as

$$B[F_{I,B}(\cdot)] \triangleq \sum_{j=1}^{M-1} E[|s^j(n) - g^j(n)|]. \tag{3.9}$$

then, from the triangle inequality, $|A + B| \leq |A| + |B|$,

$$C[F_{I,B}(\cdot)] \leq B[F_{I,B}(\cdot)], \tag{3.10}$$

From this, we see that the MAE of the stack filter acts as an upper bound on the MAE of the GANF (when both filters are optimized). Also, it can be seen that the

GANF may not necessarily possess the stacking property. As a result, each neural operator can be trained independently. There is no need to enforce a level consistency or stacking constraint during training. $\square$

CHAPTER 4

NEURAL OPERATORS

It is important to choose a suitable neural operator for use in the GANF. There are four major things to consider in the selection of a neural operator. The first of these is classification ability. The neural operator is really implementing a Boolean function, which can be thought of as a classification operation. Some inputs will produce a 0 output (inputs assigned to class A) and other inputs will produce a 1 output (inputs assigned to class B). We do not know *apriori* what classification scheme will be required. For one type of input signal, perhaps a single neuron (linear discriminant function, or LDF) may be able to implement the classification. On the other hand, a linear discriminant function cannot implement all Boolean functions (as discussed previously). If the best filter requires a function which cannot be implemented by a LDF, the best that can be achieved is the minimization of classification error given that LDF. As a result, we would like to choose a neural operator which has a high probability of being able to implement an arbitrary classification. Ideally, this probability should be unity.

The second consideration in choosing a neural operator is that of complexity. As the classification abilities of a neural network increase, so does its complexity. If we choose a network with a high separation probability, it may be too cumbersome to implement. In some sense, the complexity of a network can be measured by the number of weights it has. As this number increases, the network becomes slower and requires more memory to be implemented. In addition, it requires more training, which leads to the next two considerations.

A third factor involved in network choice is that of generalization. This is a measure of the network correctly classifying things which it has not seen before (during training) [11]. We will discuss this in greater detail later. However, at this point let it be stated that the generalization can be roughly linked to the number of weights in the network.

The final topic involved in network selection is that of training. This is a very broad topic and has a great impact on the performance of a network. The first thing is selection of a training scheme. Most of the networks considered in this thesis use the backpropagation learning scheme. This is a very common method, but can lead to the weights being frozen at a local minimum of the error function. As a result, more advanced training schemes can be used with perhaps better results than those in this thesis. The second area of training involves the training data set. In some cases this is fixed, while in others the training data is unlimited. Throughout this thesis, it will be assumed that the size of the training set is fixed. Therefore, selection of a neural operator will depend on its capacity, complexity and generalization given a certain length of training data. We will now examine these areas in more depth.

4.1 Capacity

The capacity of a network is a measure of its ability to store information [11]. In our case, it is a measure of how many different classifications the network is capable of implementing. For a binary classifier with N binary inputs, there are 2^N binary patterns that can appear at the inputs. Since each pattern may be independently assigned to class A or class B, there are 2^{2^N} possible classifications. The *deterministic capacity*, C_D of a neuron equals the number of different patterns which it can classify with probability one [12]. Note that the deterministic capacity must be less than or equal to 2^N. Also, there is a parameter associated with networks called the statistical capacity, denoted by C_S [12]. This is the number of input patterns which can be arbitrarily classified by the network with probability $\frac{1}{2}$. For unknown binary data, the probability of separation can be expressed as

$$P_{SEP} = \frac{\text{no. of different classifications that the net can implement}}{2^{2^N}}. \tag{4.1}$$

4.2 Complexity

The complexity of a network depends on the number of elements in the network, the interconnections, the mathematical operations needed, and many other things. Large, intensely interconnected networks require many weights which must be stored, accessed and updated. Assuming standard floating point numbers (ANSI/IEEE-754-1985), each weight will require 4 bytes of memory [13]. In addition, each weight must be involved in mathematical operations for output generation and training (weight updating). As a result, we can use the number of weights as a measure of the network's complexity. To make implementation easier, we try to minimize the number of weights. This also improves the generalization of the net, as shown in the following section.

4.3 Generalization and Training

Note that we are training a neural network to classify input data into one of two classes. This is accomplished by showing the network input vectors and telling it which class they belong to. For a moment, let us assume that this training process is perfect. The question we ask is: How many examples must we show the network before it can "learn" the classification?

Let $\frac{n_A}{l}$ be defined as follows:

$$\frac{n_A}{l} \triangleq \frac{\text{network decides class A}}{\text{total number of training samples}}. \tag{4.2}$$

Since the network is being trained, eventually the ratio $\frac{n_A}{l}$ will equal the number of observed inputs in class A in a data set of length l. By Bernoulli's theorem, if we consider an infinite set of data, this ratio will equal P_A, the true probability of a sample being in class A:

$$\lim_{l \to \infty} \frac{n_A}{l} = P_A. \tag{4.3}$$

However, in real life, we do not have infinite samples of data available. Also, we may not have the time to train the network with enormous amounts of data. Therefore, the relative frequency of A cannot be assured of equaling P_A. As a result, all we can do is to let the ratio $\frac{n_A}{l}$ approach P_A. The closer these two numbers are, the more generalization has been achieved by the network.

In order to measure the generalization [14], we can find the difference $|\frac{n_A}{l} - P_A|$. Let us denote the maximum difference as

$$\pi^{(l)} = \max_{A \in S} |\frac{n_A}{l} - P_A|, \tag{4.4}$$

where A is the event: the input belongs to class A, and S is the sample space of inputs. Then $\pi^{(l)}$ represents the worst case generalization error of the ideal network. It is the maximum difference between the relative frequency of a class A decision and the true probability of the sample really belonging to class A. Therefore, the smaller this number is, the more we are able to generalize, or know about P_A from our observed ratio, $\frac{n_A}{l}$. It was found by Vapnik and Chervonenkis [15] that this worst case generalization error can be bounded. Some of the important results of their paper are presented below. First, however, let us define some basic concepts which are necessary for understanding the theorems. Let the set X_r be a subset of some space X, consisting of r elements:

$$X_r = \{x_1, x_2, \ldots, x_r\}, \tag{4.5}$$

Let an event $A \in S$ induce a subsample in X_r as defined below:

$$X_r^A = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \tag{4.6}$$

If we look at all the possible events $A_i \in S$, we can generate corresponding subsamples $X_r^{A_i}$. The number of *different* subsamples of size r induced by the events $A_i \in S$ will be denoted by $\Delta^S(x_1, \ldots, x_r)$. Now if we examine $\Delta^S(x_1, \ldots, x_r)$ for *all* samples of size r (that is, all $X_r \in X$), we can find its maximum value. Let us define

$$m^S(r) \triangleq \max_{X_r \in X} \Delta^S(x_1, \ldots, x_r). \tag{4.7}$$

We will call $m^S(r)$ the "growth function." Now let us look at the results of the paper [15].

Theorem 4.1 *The probability that the relative frequency of at least one event in Class S differs from its probability in an experiment of size l by more than ϵ, for $l \geq \frac{2}{\epsilon^2}$, satisfies the inequality*

$$P(\pi^{(l)} > \epsilon) \leq 4m^s(2l)e^{-\frac{\epsilon^2 l}{8}}. \tag{4.8}$$

From this, another theorem can be derived:

Theorem 4.2 *If $m^S(l) \leq l^n + 1$, then $P(\pi^{(l)} \to 0) = 1$.*

Also, the authors prove

Theorem 4.3 *The growth function $m^S(r)$ is either identically equal to 2^r or else is majorized*[1] *by the power function $r^n + 1$, where n is a positive constant equaling the value of r for which the equation*

$$m^S(r) = 2^r \tag{4.9}$$

is violated for the first time.

In Theorem 4.3, the positive constant n is called the VCdim of the system.

To apply this to our problem, we want to know when $P(\pi^{(l)} \to 0) = 1$, which is specified by Theorem 4.2. In our case, $\pi^{(l)} = \max_{A \in S} |\frac{n_A}{l} - P_A|$ is the worst case generalization error. $\frac{n_A}{l}$ is the observed relative frequency of class A determinations at the output and P_A is the true value of $\frac{n_A}{l}$ if we trained the network forever

[1]Majorized means that one function acts as an upper bound on another function.

($l \to \infty$). From Theorem 4.3, we see that $m^S(r) \leq r^n + 1$ for $r \geq n$. Applying this to the results of Theorems 4.1 and 4.2, we see that

$$m^S(l) \leq l^n + 1 \quad \text{if} \quad l \geq r \geq n. \tag{4.10}$$

As a result, we can establish a bound on the generalization error only when the number of training samples is greater than or equal to the VCdim of the system.

This discussion assumed a perfect training process. In reality, however, we encounter local minima, non-optimal step sizes, and deal with estimations of the gradient, etc. As a result, we must train the network with many more training samples than its VCdim. The accepted number in practice is 10 times the VCdim. In addition, it may be necessary to cycle through the training set a few times until convergence is achieved.

The next problem involves finding the VCdim. This is very difficult in some cases, but bounds have been established for many networks. First of all, it is important to understand that Theorem 4.3 is equivalent to Theorem 4.4 below [15] [16]:

Theorem 4.4 *The VCdim of a system is the size of the largest set X_r of data samples for which the system can implement all possible 2^r dichotomies on X_r, where $r = |X_r|$ = the number of elements in X_r.*

For a single perceptron, the VCdim has been shown to equal $(N+1)$ exactly, where N is the size of the input vector [17]. For a 2 layer, fully interconnected network, bounds on the VCdim can be found [16]:

$$2\lfloor \frac{N_1}{2} \rfloor n \leq VCdim \leq 2N_w \log_2(eN_N), \tag{4.11}$$

where $\lfloor \cdot \rfloor$ is the *floor* operator, N_1 is the number of nodes in the first layer, N_N is the total number of nodes in the network, N_w is the number of weights in the network, n is the dimension of the input pattern and e is the base of the natural logarithm. It is important to note that some assumptions were made in deriving eq. (4.11) which may not apply to networks using sigmoid nonlinearities. The VCdim of a radial basis function network can be shown [17] to be bounded by

$$VCdim \leq 2N_w \log_2 n(eN_N), \tag{4.12}$$

Here, notation is the same as in eq. (4.11). In general, the number of weights in a network can be used as an estimate of its true VCdim.

4.4 Examples of Neural Operators

The GANF makes use of neural operators to implement Boolean functions. As previously discussed, there are four important considerations in selecting the neural operators. While there are a vast number of neural operators to choose from, this section will discuss six possibilities.

4.4.1 Single Neuron

The structure of a single neuron is shown in Figure 4.1. The neuron receives N inputs, which we will describe as a vector, $\mathbf{u}$. Each element of the input vector is multiplied by an independent weight and added. Also added is the value of a bias weight, w_0. The operation can be described mathematically as follows:

$$s(n) = [w_0(n) \;\; \mathbf{a}^T(n)] \begin{bmatrix} 1 \\ \mathbf{u}(n) \end{bmatrix} \tag{4.13}$$

where $\mathbf{a}^T(n) = [w_1(n) \;\; w_2(n) \;\; \cdots \;\; w_N(n)]$ and $\mathbf{u}^T(n) = [x_1(n) \;\; x_2(n) \;\; \cdots \;\; x_N(n)]$. We can define

$$\mathbf{x}^T(n) \triangleq [1 \;\; \mathbf{u}^T(n)], \tag{4.14}$$

and

$$\mathbf{w}^T(n) \triangleq [w_0(n) \;\; \mathbf{a}^T(n)] \tag{4.15}$$

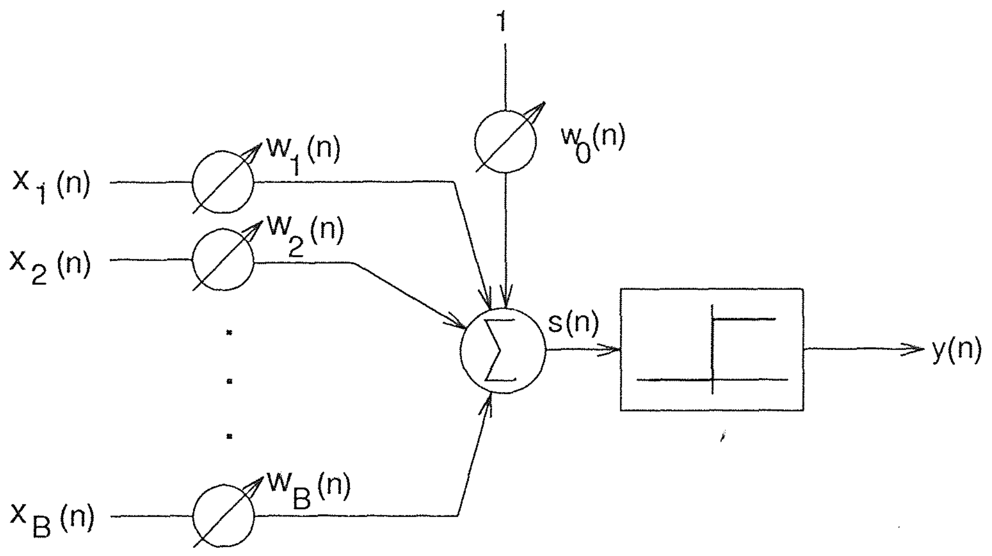


Figure 4.1 Single neuron.

to equivalently represent the operation by

$$s(n) = \mathrm{w}^T(n)\mathrm{x}(n). \tag{4.16}$$

The analog output, s(n), is then processed by a non-linearity. In our case, this nonlinear function is the signum function. Therefore, the complete operation of the neuron can be described by

$$y(n) = \frac{\mathrm{sgn}[s(n)] + 1}{2}, \tag{4.17}$$

or

$$y(n) = \frac{\mathrm{sgn}[\mathrm{w}^T(n)\mathrm{x}(n)] + 1}{2}. \tag{4.18}$$

For this and all cases discussed, we will assume $\mathrm{x}(n)$ is a binary vector. Because of this and the use of a hard-limiting sgn[·] function, the neuron's operation can be described by a Boolean function.

The neuron is trained by providing it with inputs and a desired response (classification of the input vector). The weight vector, $\mathrm{w}(n)$, is then updated by some type of learning rule, which tries to minimize the classification error in some sense. A very popular learning rule is called the LMS algorithm [19]. This algorithm works by using a gradient descent on the weight-error surface. However, expected values in the gradient formulas are replaced with their instantaneous estimates [5]. As shown in section 2.3.2, the resulting update formula for the weights is

$$\mathrm{w}(n+1) = \mathrm{w}(n) + \mu \mathrm{x}(n)[d(n) - \mathrm{w}^T(n)\mathrm{x}(n)]. \tag{4.19}$$

It should be pointed out that although the single neuron implements a Boolean function in N variables, it cannot implement all possible Boolean functions in N variables. As a result, the minimum MSE for the single neuron may not be the global minimum MSE that can be obtained with a more complicated net. However, the LMS algorithm will ideally find the best Boolean function within the set obtainable by the single neuron.

The number of implementable Boolean functions is a measure of the capacity of the single neuron. It was shown [18] that if M input patterns are in general position, a single neuron with $(N+1)$ weights can implement

$$2^{1-M} \sum_{i=0}^{N} \begin{pmatrix} M-1 \\ i \end{pmatrix} \tag{4.20}$$

distinct classifications. General position means that for a set of M data points in N-dimensional space, no subset of $N+1$ points lies on an $(N-1)$-dimensional hyperplane. However, we are dealing with binary input data which may not be in general position. Therefore, eq. (4.20) serves as an upper bound. The probability of separation is

$$2^{1-2^N} \sum_{i=0}^{N} \begin{pmatrix} 2^N-1 \\ i \end{pmatrix}. \tag{4.21}$$

The statistical capacity of a single neuron is

$$C_S = 2(N+1). \tag{4.22}$$

It can be shown that the VCdim of a single neuron is $N+1$. This follows from Theorem 4.4 in section 4.3. Note in this case that the VCdim equals the number of weights exactly.

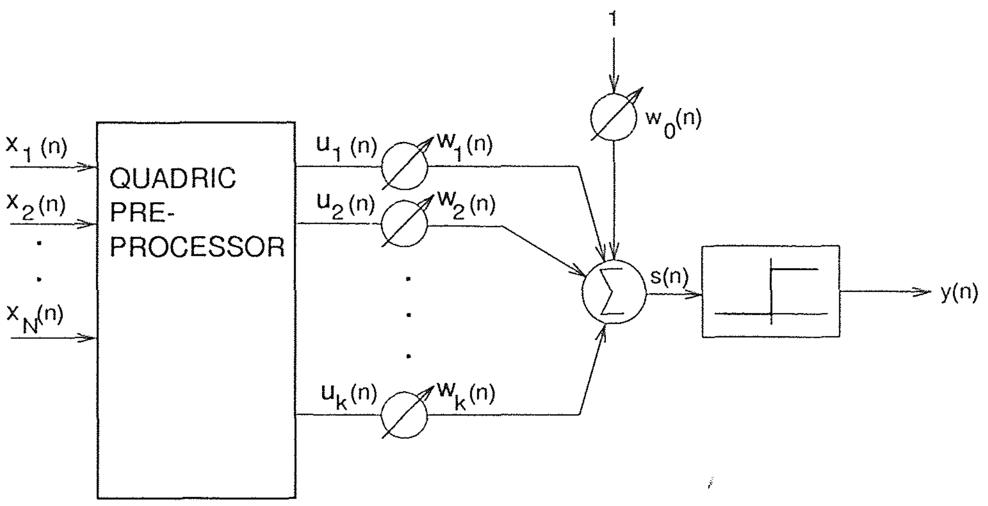


Figure 4.2 Quadric neuron.

4.4.2 Quadric Neuron

The structure of the quadric neuron is shown in Figure 4.2. It is very similar to the single neuron, except that the binary input vector has been pre-processed before reaching the summation junction. Because of the pre-processing, more weights have been added to accommodate the additional terms in the summation. For a quadric neuron, the discriminant function is represented by the following equation:

$$s(n) = w_0 + \sum_i w_i x_i + \sum_j \sum_k w_{jk} x_j x_k, \tag{4.23}$$

Note that terms in the third summation with $j = k$ will be redundant (since we are dealing with binary values). As a result, the quadric discriminant function will have a total of $(\frac{N(N+1)}{2} + 1)$ weights for an N bit binary input. We will assume that the VCdim equals the number of weights, or $(\frac{N(N+1)}{2} + 1)$.

4.4.3 Polynomial Discriminant Neuron

The structure of this neuron is shown in Figure 4.3. It is very similar to the quadric neuron, except more terms are added to the discriminant function:

$$s(n) = w_0 + \sum_i w_i x_i + \sum_j \sum_k w_{jk} x_j x_k + \sum_l \sum_m \sum_n w_{lmn} x_l x_m x_n + \cdots \tag{4.24}$$

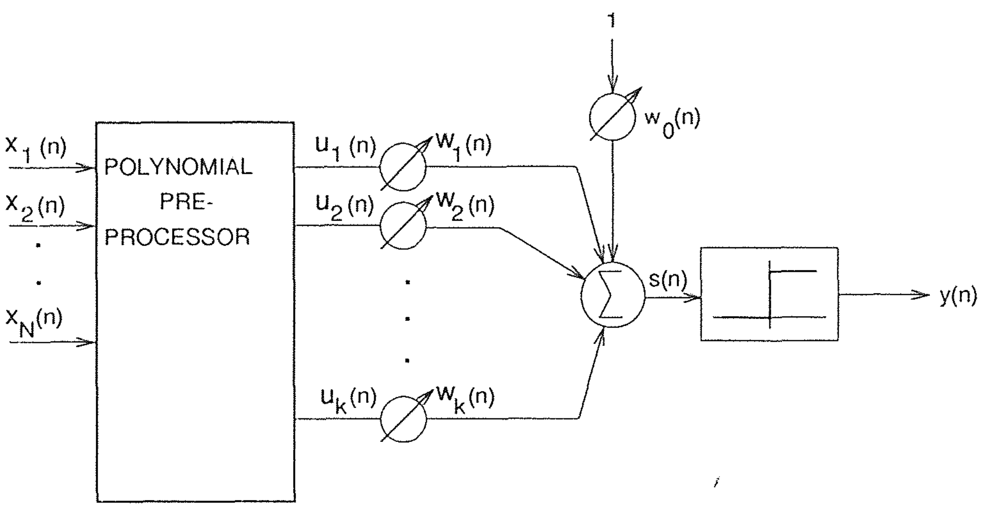


Figure 4.3 Neuron with polynomial discriminant function.

This neuron is trained in the same manner as are the quadric and single neurons. It can be shown that the discriminant function of eq. (4.24) can implement any Boolean function.

Proposition 4.1 *A single Neuron with polynomial pre-processing can implement any Boolean function.*

The proof of this is shown in Appendix A. As a result, the probability of separation is one. Also, with the polynomial pre-processor, there can be at most 2^N patterns presented to the neuron (which has 2^N weights). In fact, it is necessary for a neuron to have 2^N weights to be able implement all possible classification of N binary inputs [12]. Because of this, the VCdim will equal 2^N.

4.4.4 2-Layer

It was thought that a theorem of Kolmogorov could be applied to neural networks to justify the use of 2-layer networks [20] [21]. This theorem showed that any continuous function (mapping) could be exactly represented by a superposition of many continuous functions [22]. This meant that the first layer of a network could implement the continuous functions and the second layer could add the outputs of

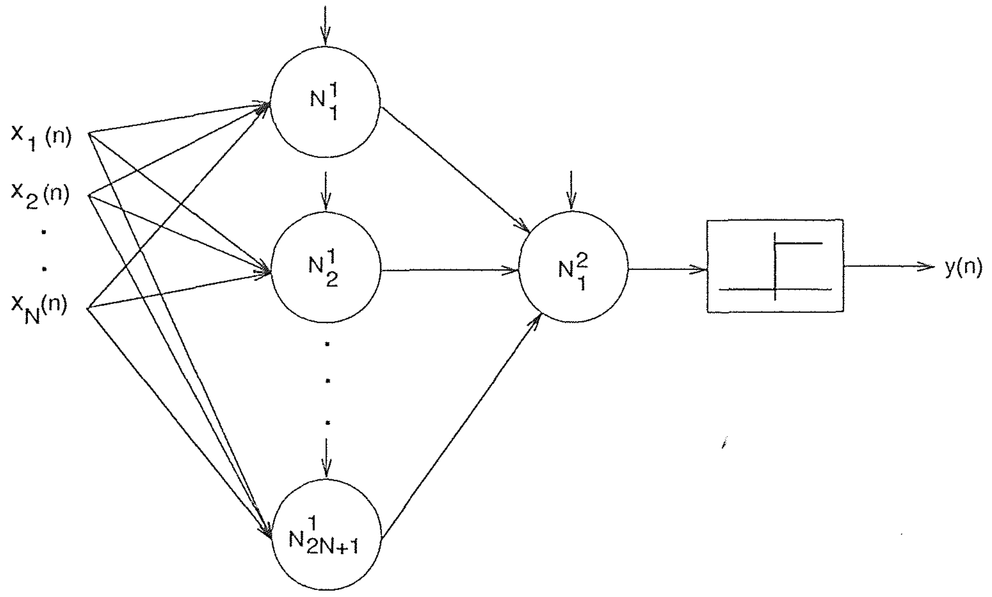


Figure 4.4 Minimal two-layer net.

the first layer [20]. Such a network could be constructed with $(2N + 1)$ neurons in the first layer and 1 neuron in the second [21]. As shown in Figure 4.4, this would involve a total of $(N + 1)(2N + 3)$ weights.

We are using this net to implement a Boolean function which, of course, is not a continuous function (except in trivial cases). Therefore, we use the 2-layer net to approximate the Boolean function. The VCdim is bounded by [16]:

$$2\lfloor \frac{N_1}{2} \rfloor n \leq VCdim \leq 2N_w \log_2(eN_N), \tag{4.25}$$

where $\lfloor \cdot \rfloor$ is the *floor* operator, N_1 is the number of nodes in the first layer, N_N is the total number of nodes in the network, N_w is the number of weights in the network, n is the dimension of the input pattern and e is the base of the natural logarithm.

4.4.5 Large 2-Layer

It was also shown [23] that the continuous functions in Kolmogorov's theorem must be highly non-smooth. These functions cannot be implemented by a standard 2-layer backpropagation net. This could mean degraded performance for the net discussed

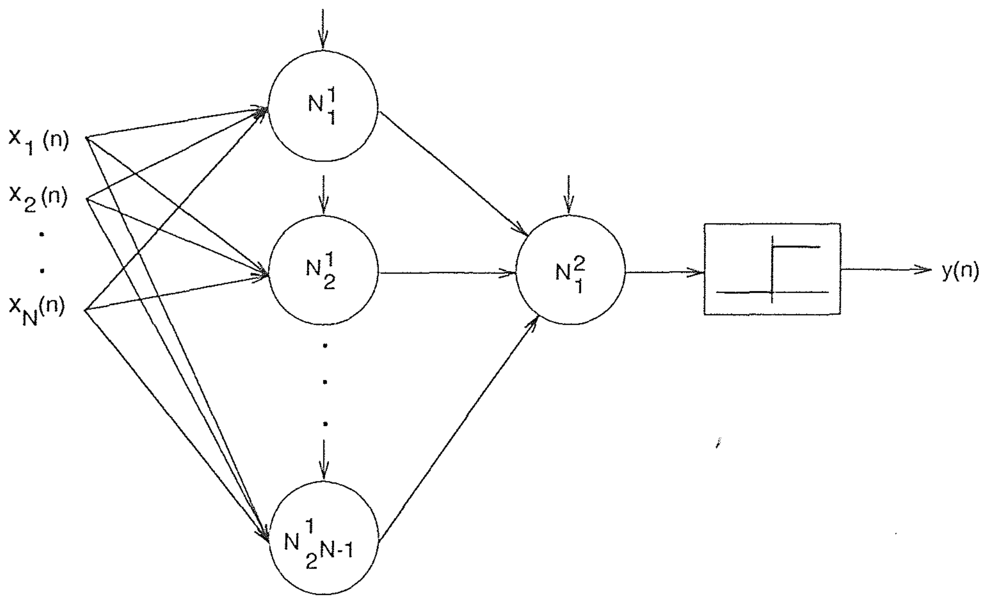


Figure 4.5 Large two-layer net.

in Section 4.4.4. As a result, we develop another 2-layer net here, which is shown in Figure 4.5. If we have 2^{N-1} neurons in the first layer and 1 neuron in the second layer, we can implement any Boolean function in N variables.

Proposition 4.2 *A 2 layer net with N inputs, 2^{N-1} neurons in the first layer and 1 neuron in the second layer can implement any Boolean function.*

The proof of this is shown in Appendix B. Since any function, or classification, could be implemented, its probability of separation is one. The VCdim of this network is bounded by [16]

$$2\lfloor \frac{N_1}{2} \rfloor n \leq VCdim \leq 2N_w \log_2(eN_N), \tag{4.26}$$

where the notation is the same as in eq. (4.25).

4.4.6 Radial Basis Function

The *Radial Basis Function* (RBF) network is another type of 2 layer network which can be used for pattern classification. Radial basis function networks are based on a theory of mathematics called regularization, which is applied toward solving ill-posed problems [24]. Note that when we train a network, we use a subset of possible inputs, and rely on generalization to provide valid outputs for inputs which were not in the training set. Because of this, there are many network functions which could be equal at a number of points (the training points), and which differ in between. As a result, the problem is ill-posed [24]. There are less constraints than required to define a solution.

This problem is solved in the following manner. We are really trying to have the network implement an unknown function that maps inputs to an output. In our case, the output is binary. Training informs the network of the exact value of the function at various points. Then, by assumptions (smoothness for example) or apriori knowledge, the net can interpolate the output for other inputs. As a result, the network will implement a mapping which approximates that of the true, but unknown, desired function [24] [25].

Radial basis functions are part of a very broad area in mathematics. Even among RBF networks, there are many variations. In this thesis, we will consider a basic RBF network of medium complexity [17]. This network appears as shown in Figure 4.6.

The RBF network receives inputs from a binary vector of length N. These N inputs are fed to K kernel elements in the first layer. The kernel elements serve as a basis with which the function can be generated. Ideally, K should equal the length of the training set. Then there would be a kernel function for every known point in the function. This, however, would be too cumbersome in practice. As a result, K is usually chosen to be much less than the length of the training set. These kernel functions are then positioned in the input space to minimize the error from the true solution [24].

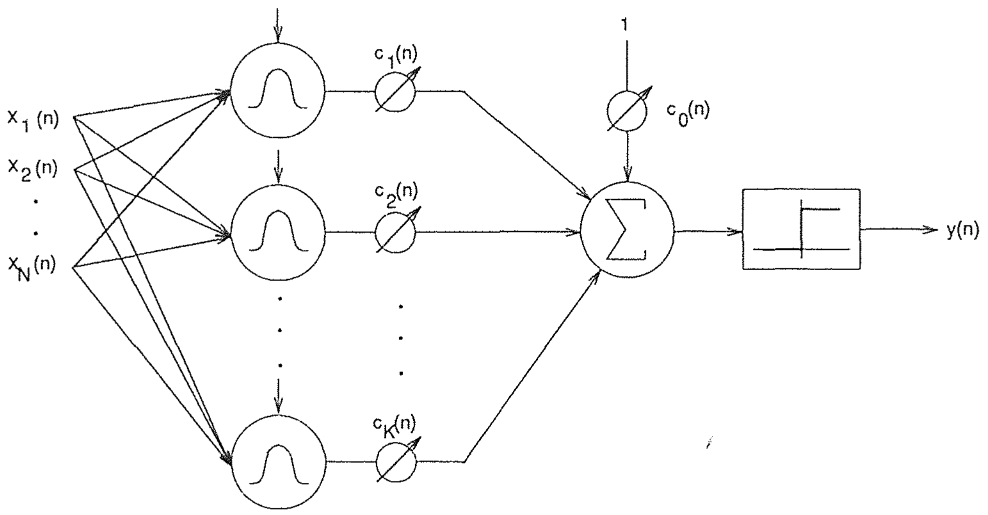


Figure 4.6 Radial basis function network.

While there are many possibilities for the kernel functions, a popular choice is shown below:

$$G(x) = \exp[-x^2] \tag{4.27}$$

When used in the RBF network, we have an independent function of this type implemented by each element in the first layer. Here, however, the inputs are vectors, and each kernel function can have its own "center", t_α, where $1 \leq \alpha \leq K$:

$$G\|\mathrm{x} - \mathrm{t}_\alpha\|^2_{\mathbf{W}_\alpha} = \exp[(\mathrm{x} - \mathrm{t}_\alpha)^T \mathbf{W}_\alpha^T \mathbf{W}_\alpha (\mathrm{x} - \mathrm{t}_\alpha)]. \tag{4.28}$$

If we define the matrix $\mathbf{W}_\alpha$ as

$$\mathbf{W}_\alpha = \frac{1}{\sqrt{2}\sigma_\alpha}\mathbf{I}, \tag{4.29}$$

where $\mathbf{I}$ is the identity matrix, then

$$G\|\mathrm{x} - \mathrm{t}_\alpha\|^2_{\mathbf{W}_\alpha} = exp[\frac{(\mathrm{x} - \mathrm{t}_\alpha)^T(\mathrm{x} - \mathrm{t}_\alpha)}{2\sigma_\alpha^2}]. \tag{4.30}$$

Note that for each element in the first layer, we can independently choose the function centers, t_α, and the variances, σ_α^2. Next, the complete network output is found from

$$\tilde{f}(\mathrm{x}) = c_0 + \sum_{\alpha=1}^{K} c_\alpha G\|\mathrm{x} - \mathrm{t}_\alpha\|^2_{\mathbf{W}_\alpha}, \tag{4.31}$$

where the parameters c_i are free to be chosen. They act as weights to a linear summation in the second layer. Of course, the final output will be hard limited by the $sgn[\cdot]$ function to implement a distinct classification.

The next topic of discussion is the training of the network. There are three sets of parameters which must be set: $\mathbf{t}_\alpha$, σ_α and c_α. There are many ways to set these parameters and we will consider only one method. First of all, since the number of elements in the first layer, K, is less than the length of the training set, we must position the centers of the K elements carefully. This is done using a k-means algorithm [17]. Basically, the n elements are dispersed in the data set so that all of the clusters of data are each represented by an element. That is, the centers of the kernel functions are moved so that they are in the vicinity of important clusters of the data.

After this step is completed, the variances for all of the kernel functions are set. These variances are actually a measure of the spread of data about the center of the kernel function. As a result, they are set equal to the average distance squared between a kernel function's center and the data points in its vicinity [17]:

$$\sigma_\alpha^2 = \frac{1}{N_\alpha} \sum_{\mathbf{x} \in \Theta_\alpha} (\mathbf{x} - \mathbf{t}_\alpha)^T (\mathbf{x} - \mathbf{t}_\alpha), \tag{4.32}$$

where Θ_α is the set of training data which is closer to kernel α than to any other kernel element and N_α is the size of this set.

Finally, the weights for the second layer summation node must be set. In this thesis, we set these using the LMS algorithm. This is accomplished the same way as for a single neuron. Here, inputs are applied to the first layer, which generates K outputs. These K values then act as inputs to the second layer summation. With this, adaptation takes place in the usual manner.

The VCdim of the radial basis function network can be shown [17] to be bounded by

$$VCdim \leq 2[K(N+1)+1] \log_2[e(K+1)]. \tag{4.33}$$

4.5 Summary

Tables 4.1 and 4.2 below summarize the information presented here in general terms for all of the nets. Tables 4.3 and 4.4 show the information for input vectors of lengths 9 and 15, respectively.

Table 4.1 Neural net summary, weights and VCdim.

Neural Network	Weights	VCdim
Single Neuron	$N+1$	$\sim N+1$
Quadric Neuron	$\frac{N(N+1)}{2}+1$	$\sim \frac{N(N+1)}{2}+1$
Polynomial Neuron	2^N	2^N
Small 2-Layer	$(2N+3)(N+1)$	$\leq 2(N+1)(2N+3)\log_2[2e(N+1)]$
Large 2-Layer	$(N+2)2^{N-1}+1$	$\leq [(N+2)2^N+2]\log_2[e(2^{N-1}+1)]$
RBF	$K(N+1)+1$	$\leq 2[K(N+1)+1]\log_2[e(K+1)]$

Table 4.2 Neural net summary, separation probability.

Neural Network	Separation Probability (2^N patterns)
Single Neuron	$\leq 2^{(1-2^N)}\sum_{i=0}^{N}\begin{pmatrix} 2^N-1 \\ i \end{pmatrix}$
Quadric Neuron	—
Polynomial Neuron	1
Small 2-Layer	—
Large 2-Layer	1
RBF	—

Table 4.3 Neural net summary with 9 inputs (3x3 window).

Neural Network	Weights	VCdim
Single Neuron	10	$\sim$10
Quadric Neuron	46	$\sim$46
Polynomial Neuron	512	512
Small 2-Layer	210	$\leq$2421
Large 2-Layer	2817	$\leq$53232
RBF	$10K + 1$	$\leq [20K + 2] \log_2[e(K + 1)]$

Table 4.4 Neural net summary with 25 inputs (5x5 window).

Neural Network	Weights	VCdim
Single Neuron	26	$\sim$26
Quadric Neuron	326	$\sim$326
Polynomial Neuron	3.35x10^7	3.35×10^7
Small 2-Layer	1378	$\leq$19686
Large 2-Layer	4.53x10^8	$\leq 2.30 \times 10^{10}$
RBF	$26K + 1$	$\leq [52K + 2] \log_2[e(K + 1)]$

CHAPTER 5
SIMPLIFYING THE GANF

In order to achieve good performance, the GANF must be trained on a large number of samples. As previously discussed, this number depends on the VCdim of the network used, in addition to the training scheme employed. In general, more training will improve the generalization. However, as the length of the training set is increased, the training time increases proportionately. Excessive training times can prevent the filter's use in practical, real world problems. Therefore, in order to train the GANF on enough samples and minimize the training time, the training time per sample must be minimized.

To get an idea of the practicality of the GANF, let us consider the use of the neural operators presented in section 4.4. With medium sized training sets, all of these GANFs required long training times. The times were shortest for the single neuron, but grew to excessive levels for the large 2-layer network. Considering the complexity of the networks, this is understandable. With an input vector of 8-bit precision, there are 255 levels of neural operators in the GANF. If these levels are independently trained on a data set of length 16384, there would be a total of 4.2 million training operations. Even at 1ms per level update, it would take 1.16 hours to train the filter. In addition, there are massive memory requirements necessary for implementing many of the networks. Once the network is trained, however, VLSI implementation would allow very fast operation. As a result, most of the need for speed increase is focused on the training. Of course, many improvements to the training could also be applied to filtering if microprocessor implementation (an algorithm) is chosen over VLSI.

Keep in mind that the hardware design will determine the relation of the training and filtering processes. The training could come first, after which a VLSI integrated circuit is made, or some type of gate array in a universal chip is burned. This approach would be alright if the filter is operated in a stationary environment. For non-stationary environments, the training and filtering could take place simultaneously. The filtering could take place in a VLSI chip, with some DRAM storing lookup tables for the Boolean functions. At the same time, a microprocessor could be running a training algorithm operating on every k^{th} sample, where k is determined by the speed adaptation routine and the hardware. When training is completed, the new Boolean functions would be dumped to the DRAM segment of the VLSI filter. The data loading could be made transparent to the filtering operation by using dual port RAM or by interleaving access times. Therefore, the training could take place at a slower rate. Nonetheless, it is still worthwhile to increase the speed of this training operation.

5.1 Simplified Stack Filters and the ATD Architecture

Before we discuss how to speed up the GANF, we will first mention another class of filters which achieves the same goal. This class, called *Adaptive Threshold Decomposition (ATD)* filters, was created by Lin, *et al* [26] to increase the speed of stack filter training. Since GANFs are based on stack filters, the framework of ATD filters can be applied to GANFs. Note that in a stack filter or GANF, there are a total of $(M - 1)$ binary vectors produced by the threshold decomposition. However, there are at most only $(B + 1)$ *different* binary vectors, where B is the window size. For a stack filter, since each Boolean function is the same, there are at most $(B + 1)$ unique outputs. This fact was recognized by Lin, *et al*, and led to their development of fast algorithms and fast stack filtering structures.

Prior to the advent of the fast structures, there existed mainly three methods for setting up stack filters [6] [8] [27]. Two of these were discussed in section 2.3.1, while the third approach involves an adaptive procedure which was not described.

The adaptive method involves keeping track of level-crossing statistics at each level in the stack filter [27]. A table is set up with locations for each of the possible binary inputs. Then, locations in the table are either incremented or decremented, depending on the desired level output for a particular binary input word. Finally, the table is converted to represent a Boolean truth table, and is adjusted to enforce the stacking constraint. The fast algorithm basically implements this procedure for the (at most) $(B+1)$ different entries present in the threshold decomposed input. The stacking constraint is enforced only for the $(B+1)$ table locations which were changed. This procedure results in a dramatically shorter training time [4].

Based on this FAST algorithm, Lin *et al* subsequently defined an entire class of filters called Adaptive Threshold Decomposition (ATD) filters [26]. These filters can be described as follows, using notation as in section 2.2. First, consider an integer input vector

$$\mathbf{r}_B^T(n) = [r(n - \frac{B-1}{2}) \cdots r(n) \cdots r(n + \frac{B-1}{2})], \tag{5.1}$$

or

$$\mathbf{r}_B^T(n) = [r_1(n) \;\; r_2(n) \;\; \cdots \;\; r_B(n)], \tag{5.2}$$

where $r_k(n) \in \{0, 1, \ldots, M-1\}$ are the B elements in the filter's window at time n. The threshold decomposition operation produces $(M-1)$ binary vectors of length B:

$$\mathbf{x}_B^i(n) = T^i[\mathbf{r}_B(n)], \tag{5.3}$$

where

$$T^i[\mathbf{r}_B(n)] = [T^i[r(n - \frac{B-1}{2})] \cdots T^i[r(n)] \cdots T^i[r(n + \frac{B-1}{2})], \tag{5.4}$$

and

$$T^i[x] \triangleq \begin{cases} 1, & \text{if } x \geq i \\ 0, & \text{otherwise.} \end{cases} \tag{5.5}$$

Now, let us define $R_{(k)}(n)$, $k \in \{1, 2, \ldots, B\}$ to be the kth smallest sample in the window at time n. Then let

$$\mathrm{u}_k(n) = \mathrm{x}_B^{R_{(k)}(n)}(n), \tag{5.6}$$

where $\mathrm{x}_B^{R_{(k)}(n)}$ is the binary vector resulting from the threshold decomposition on level $R_{(k)}(n)$.

Next, let us define Δ_k as the difference between samples of rank k and $k-1$:

$$\Delta_k(n) \triangleq R_{(k)}(n) - R_{(k-1)}(n), \tag{5.7}$$

Note here that the window is as defined in eq. (5.1) and $R_{(0)}(n)$ is always assigned the value of zero. We can then represent the ATD filter by

$$y(n) = S(\mathbf{r}_B(n)) = \sum_{k=1}^{B} f_k[\mathrm{u}_k(n)]\Delta_k(n). \tag{5.8}$$

where $f_k(\cdot)$ can be a Boolean function, but may be more general.

In their paper, Lin *et al* prove a number of different properties concerning the ATD filters. The results are beyond the scope of this thesis. However, one important result is that any nontrivial stack filter can be realized as an ATD filter. (Nontrivial means that $S_f(\cdot) \neq 1$ or 0 identically.) We will re-prove this here, but it is best to see [26] for a complete description. First of all, the output of a stack filter can be described by

$$y(n) = f(\mathbf{r}_B(n)) = \sum_{i=1}^{M-1} f(\mathrm{x}_B^i(n)), \tag{5.9}$$

where we define

$$f_k(\cdot) = S_f(\cdot) = f(\cdot). \tag{5.10}$$

$$y(n) = \sum_{i=R_{(0)}(n)+1}^{R_{(1)}(n)} f(\mathrm{x}_B^i(n)) + \sum_{i=R_{(1)}(n)+1}^{R_{(2)}(n)} f(\mathrm{x}_B^i(n)) + \cdots + \sum_{i=R_{(B)}(n)+1}^{M-1} f(\mathrm{x}_B^i(n)). \tag{5.11}$$

Note, however, that

$$\mathrm{x}_B^{R_{(j)}(n)} = \mathrm{x}_B^i(n) \;\; \forall \;\; R_{(j-1)}(n) < i \leq R_{(j)}(n). \tag{5.12}$$

As a result, we can re-write eq. (5.11) as

$$y(n) = \sum_{i=R_{(0)}(n)+1}^{R_{(1)}(n)} f(\mathrm{x}_B^{R_{(1)}(n)}(n)) + \sum_{i=R_{(1)}(n)+1}^{R_{(2)}(n)} f(\mathrm{x}_B^{R_{(2)}(n)}(n)) + \cdots + \sum_{i=R_{(B)}(n)+1}^{M-1} f([0 \;\; 0 \cdots 0]). \tag{5.13}$$

Since the argument of each summation term is no longer a function of i,

$$y(n) = f(\mathrm{x}_B^{R_{(1)}(n)}(n))[R_{(1)}(n) - R_{(0)}(n)] + f(\mathrm{x}_B^{R_{(1)}(n)}(n))[R_{(2)}(n) - R_{(1)}(n)] + \cdots + f([0 \;\; 0 \cdots 0])[M - 1 - R_{(B)}(n)], \tag{5.14}$$

$$y(n) = f(\mathrm{u}_1(n))\Delta_1(n) + f(\mathrm{u}_2(n))\Delta_2(n) + \cdots + f([0 \;\; 0 \cdots 0])[M-1-R_{(B)}(n)], \tag{5.15}$$

$$y(n) = \sum_{k=1}^{B} f(\mathrm{u}_k(n))\Delta_k(n) + f([0 \;\; 0 \cdots 0])[M - 1 - R_{(B)}(n)]. \tag{5.16}$$

But for non-trivial stack filters, $f([0 \;\; 0 \cdots 0]) = 0$. Therefore,

$$y(n) = \sum_{k=1}^{B} f(\mathrm{u}_k(n))\Delta_k(n), \tag{5.17}$$

which is the ATD filter, as described by eq. (5.8).

ATD filters can implement many filter types besides stack filters. In fact, by using neural operators to implement the functions $f_k[\cdot]$, we can implement a modified GANF in this form. Figure 5.1 shows a realization of an ATD filter using neural operators to implement the functions $f_k(\cdot)$. For this filter, the neural operator outputs are not required to be binary. In other words, they can take on any value $\in [0, 1]$ if it is so desired. It is important to note, however, that this ATD-GANF will not be exactly equivalent to a standard GANF. Here, we use only $(B + 1)$ neural functions in place of the $(M - 1)$ neural operators in the GANF. The neural operators in the ATD filter are not assigned to specific levels as they are in the GANF. To maintain equivalence with a standard GANF, the next section shows another possibility.

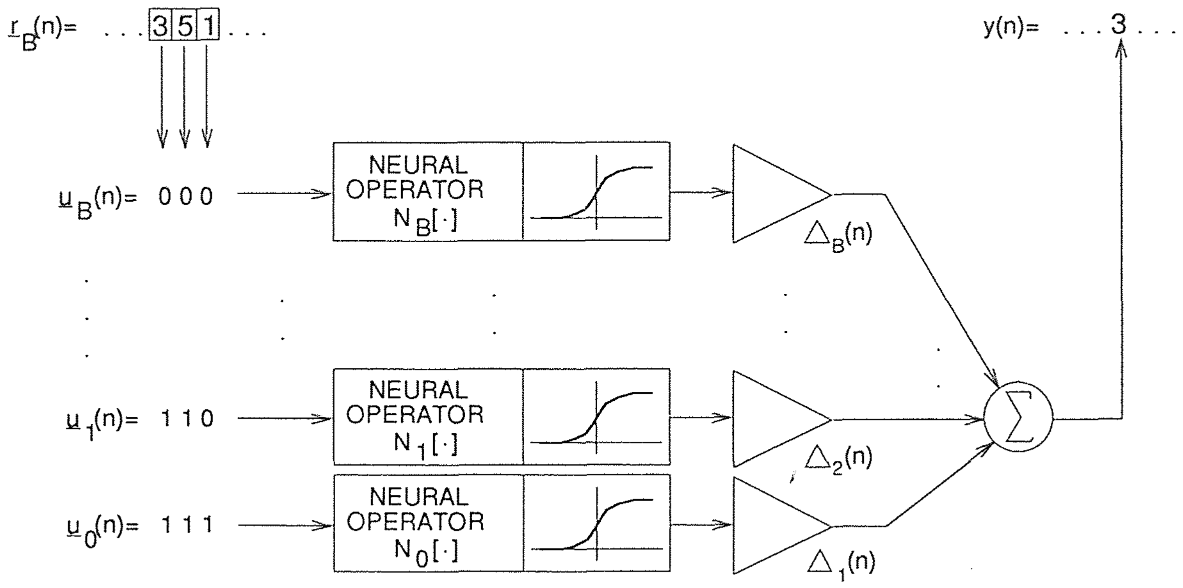

$\underline{r}_B(n)$= . . . 3 5 1 . . .
$\underline{u}_B(n)$= 0 0 0
NEURAL OPERATOR $N_B[\cdot]$
$\Delta_B(n)$
$\underline{u}_1(n)$= 1 1 0
NEURAL OPERATOR $N_1[\cdot]$
$\Delta_2(n)$
$\underline{u}_0(n)$= 1 1 1
NEURAL OPERATOR $N_0[\cdot]$
$\Delta_1(n)$
$\sum$
y(n)= . . . 3 . . .

Figure 5.1 FAST-GANF as an ATD filter.

5.2 The FAST-GANF

Besides the ATD structure, there is another way to speed up a GANF. A two step procedure can be used to first decrease the number of independent neural operators and then develop a simplified structure based on this. Unlike the ATD filters, though, this new simplified structure is identical in operation to a standard GANF.

5.2.1 Level Combinations

We will first look at combining neural functions in the GANF in terms of increasing the filtering speed. Later on, we will discuss how this can also lead to an increase in filtering ability. The idea here is to use the same neural operator to process information on a number of adjacent levels. Recall that the previously discussed GANF had the capability of implementing different Boolean operations on each of the levels. We will refer to this as a non-homogeneous GANF. With level combinations, only certain groups of levels are processed with independent neural operators. In other words, we will re-use the same Boolean function for a certain range of levels.

The limiting case for this is the use of the same neural operator for all of the levels. We will call this the homogeneous GANF.

We wish to combine levels, but at the same time, maintain the best possible filtering performance. As a result, levels cannot be combined randomly. Note that if two neural operators produce the same output for all given inputs, then they are identical. As a result, we can develop a measure of similarity of neural operators based on this. This would be a type of correlation between neural *responses* for a given input set. This idea is summarized as Proposal 5.1 below:

Proposal 5.1 *Suppose we are given two neural operators on different levels in the GANF. If, when the inputs to the two neural operators are the same, both functions generate the same output, the functions are consistent with each other. If this is the case for all inputs in the input set, then we can consider the two functions to be identical. As a result, a measure of function similarity is the probability of the functions producing the same outputs if they are operating on the same inputs.*

There are two methods which can be used to determine the similarity of the neural functions.

<u>Method 1</u>

We can define the measure of similarity as

$$g(l,j) = P\{\text{outputs on levels } i \text{ and } j \text{ are the same} \mid \text{inputs are the same}\}, \quad (5.18)$$

where l is a level number $\in \{0,1,\ldots,M-1\}$ and j is a different level number $\in \{0,1,\ldots,M-1\}$. Also, it is assumed that the binary input vectors on the two levels are the same. This is equivalent to

$$g(l,j) = P\{[(\text{level } i,j \text{ outputs are } 0) \cup (\text{level } i,j \text{ outputs are } 1)] | \text{same inputs}\}. \quad (5.19)$$

From the third axiom of probability theory [28],

$$g(l,j) = P\{(\text{outputs are both } 0)|\text{same input}\} + P\{(\text{outputs are both } 1)|\text{same input}\}$$
$$-P\{(\text{outputs are both } 0) \cap (\text{outputs are both } 1)|\text{same input}\}. \quad (5.20)$$

Therefore,

$$g(l,j) = P\{y^l(n) = 0, y^j(n) = 0|\text{same input}\} + P\{y^l(n) = 1, y^j = 1|\text{same input}\}, \quad (5.21)$$

where $y^l(n)$ is the desired output on level l at time n and $y^j(n)$ is the desired output on level j at time n. Next, we can assume without loss of generality that $j > l$. Then, if $s(n)$ is the desired output,

$$g(l,j) = P\{s(n) < l|\text{same input}\} + P\{s(n) \geq j|\text{same input}\}. \quad (5.22)$$

or

$$g(l,j) = P\{s(n) < l \mid \mathrm{x}_B^l(n) = \mathrm{x}_B^j(n)\} + P\{s(n) \geq j \mid \mathrm{x}_B^l(n) = \mathrm{x}_B^j(n)\}, \quad (5.23)$$

which equals

$$g(l,j) = \frac{P\{s(n) < l,\ \ \mathrm{x}_B^l(n) = \mathrm{x}_B^j(n)\} + P\{s(n) \geq j,\ \ \mathrm{x}_B^l(n) = \mathrm{x}_B^j(n)\}}{P\{\mathrm{x}_B^l(n) = \mathrm{x}_B^j(n)\}}. \quad (5.24)$$

To compute the best estimate of $g(l,j)$, we would have to keep track of all of the times that $\mathrm{x}_B^l(n)$ equals $\mathrm{x}_B^j(m)$ even if $n \neq m$. This would require a lot of effort. Also, we do not want the simplification method to make things more complicated than if it was not used at all. Therefore, we will estimate $g(l,j)$ by considering function similarities at the same time instants. In order to implement this simplification in estimating $g(l,j)$, we note that

Observation 5.1 *The two binary input vectors on levels l and j are equal if there exists no element, $r(k)$, in the window such that $l \leq r(k) < j$.*

Then, to compute an estimate of $g(l,j)$, we need three counter variables as follows.

$$g(l,j) \cong \frac{\sum_{n=0}^{N-1} I_1(l,j,n) + \sum_{n=0}^{N-1} I_2(l,j,n)}{\sum_{n=0}^{N-1} I_3(l,j,n)}, \tag{5.25}$$

where

$$I_1(l,j,n) = \left\{ \begin{array}{ll} 1, & \text{if } s(n) < l \text{ and } \exists\; l \leq r_k(n) < j \\ 0, & \text{otherwise} \end{array} \right\}, \tag{5.26}$$

$$I_2(l,j,n) = \left\{ \begin{array}{ll} 1, & \text{if } s(n) \geq j \text{ and} \exists\; l \leq r_k(n) < j \\ 0, & \text{otherwise} \end{array} \right\}, \tag{5.27}$$

$$I_3(l,j,n) = \left\{ \begin{array}{ll} 1, & \text{if } \exists\; l \leq r_k(n) < j \\ 0, & \text{otherwise} \end{array} \right\}, \tag{5.28}$$

where N is the length of the training set and $r_k(n)$ is an element of the window input vector. Note that to compute this measure, we really only need two counter variables, as $I_1(l,j,n)$ and $I_2(l,j,n)$ can be combined into one. Once $g(l,j)$ is computed, we can set a threshold, β, where $0 \leq \beta < 1$. If $g(l,j) \geq \beta$ and $I_3(l,j,n)$ is above a certain threshold, then we use the same neural operator for levels l and j. A large β should provide the best performance, although it could result in a complicated filter. A small β will provide a simpler filter, but may possibly decrease the filter's performance.

Method 2

To simplify the calculations, we will consider only the center element of the binary input vector instead of the entire input vector. In other words, we will use the responses to the same center pixels as a measure of the similarity of the two functions. This, of course, is not a true implementation of Proposal 5.1, but may be desirable in practice. In a way, the response to the center pixel is related to the response to the entire vector. If it was not, we could eliminate the center element in the window from the threshold decomposition.

Let us define the measure of level similarity as

$$g(l,j) = P\{y^l(n) = 0, y^j(n) = 0|\text{same center}\} + P\{y^l(n) = 1, y^j(n) = 1|\text{same center}\}, \tag{5.29}$$

where n is a time index, l is a level number $\in \{0, 1, \ldots, M-1\}$, and j is a different level number $\in \{0, 1, \ldots, M-1\}$. Also, it is assumed that the two levels have the same center bits in their binary input vectors. Next, we can assume without loss of generality that $j > l$. Then

$$g(l,j) = P\{s(n) < l|\text{same center input}\} + P\{s(n) \geq j|\text{same center input}\}. \tag{5.30}$$

We will now define two new functions as the terms in eq. (5.30):

$$g_1(l,j) \triangleq P\{s(n) < l|\text{same center input}\}, \tag{5.31}$$

$$g_2(l,j) \triangleq P\{s(n) \geq j|\text{same center input}\}, \tag{5.32}$$

with

$$g(l,j) = g_1(l,j) + g_2(l,j). \tag{5.33}$$

Note that $s(n)$ is the desired output for the given input center bit. Eq. (5.31) can now be simplified.

$$g_1(l,j) = \frac{P\{s(n) < l, x^l = x^j\}}{P\{x^l = x^j\}} \tag{5.34}$$

where x^l is the center element of $\mathrm{x}^l_B(n)$ and x^j is the center element of $\mathrm{x}^j_B(n)$.

$$g_1(l,j) = \frac{P\{s(n) < l, r(n) < l\} + P\{s(n) < l, r(n) \geq j\}}{P\{r(n) < l \cup r(n) \geq j\}}. \tag{5.35}$$

Next, we can approximate these probabilities and develop a usable measure:

$$g_1(l,j) \cong \frac{\sum_{n=0}^{N-1} I_1(l,j,n) + \sum_{n=0}^{N-1} I_2(l,j,n)}{\sum_{n=0}^{N-1} I_5(l,j,n) + \sum_{n=0}^{N-1} I_6(l,j,n)}, \tag{5.36}$$

where

$$I_1(l,j,n) = \begin{cases} 1, & \text{if } s(n) < l \text{ and } r(n) < l \\ 0, & \text{otherwise} \end{cases} \tag{5.37}$$

$$I_2(l,j,n) = \begin{cases} 1, & \text{if } s(n) < l \text{ and } r(n) \geq j \\ 0, & \text{otherwise} \end{cases} \tag{5.38}$$

$$I_5(l,j,n) = \begin{cases} 1, & \text{if } r(n) < l \\ 0, & \text{otherwise} \end{cases} \tag{5.39}$$

$$I_6(l,j) = \begin{cases} 1, & \text{if } r(n) \geq j \\ 0, & \text{otherwise} \end{cases} \tag{5.40}$$

Similarly, for $g_2(l,j)$,

$$g_2(l,j) \cong \frac{\sum_{n=0}^{N-1} I_3(l,j,n) + \sum_{n=0}^{N-1} I_4(l,j,n)}{\sum_{n=0}^{N-1} I_5(l,j,n) + \sum_{n=0}^{N-1} I_6(l,j,n)}, \tag{5.41}$$

where

$$I_3(l,j,n) = \begin{cases} 1, & \text{if } s(n) \geq j \text{ and } r(n) < l \\ 0, & \text{otherwise} \end{cases}, \tag{5.42}$$

$$I_4(l,j,n) = \begin{cases} 1, & \text{if } s(n) \geq j \text{ and } r(n) \geq j \\ 0, & \text{otherwise} \end{cases}, \tag{5.43}$$

and $I_5(l,j,n)$ and $I_6(l,j,n)$ are as defined before. Recall that $g(l,j) = g_1(l,j) + g_2(l,j)$, and can be found by adding eqs. (5.36) and (5.41). Also, note that given the training data set, we know all of the inputs to the neural operators and we know all of the desired outputs. As a result, finding $g(l,j)$ is no problem. We simply need to increment five counters based on the training data and level numbers. (Actually, we can combine the counter variables and use only two).

Once found, if $g(l,j)$ is greater than a certain threshold, $0 \leq \beta < 1$, then we can combine levels l and j. Again, the choice of β will depend on the compromise in performance which can be tolerated.

5.2.2 FAST Architecture

We can now make use of the decreased number of neural operators and create a FAST structure for the GANF. This FAST-GANF will be identical in operation to the non-FAST set-up, and may be desirable in situations where the input signal statistics vary with amplitude. (Recall that the level functions in an ATD filter process inputs based on relative amplitudes only). When constructing a FAST-GANF, the number of neural operators is determined by the parameter β as discussed previously. Considering for a moment the standard GANF, each of these neural operators, $N_i[\cdot]$, is assigned a range of operation (in terms of levels). In other words, $N_i[\cdot]$ will process binary input vectors on all levels between a_i and b_i inclusive. The FAST-GANF implements the same operation, but eliminates redundancy in the binary vectors. The integer input vector is threshold decomposed only on levels which have meaning. This both increases the efficiency of the threshold decomposition operation, and saves on neural operations which are not needed. In general, there will be *at most* $B + K$ decompositions and neural outputs, where B is the window size and K is the number of independent neural operators. The filter can be described in detail as follows:

We are given a GANF with $M-1$ levels, a window size B, no adjacent levels fed in, and K neural operators $N_i[\cdot]$ which process the input vectors $\mathbf{x}_B^i(n)$ for $a_i \le i \le b_i$. In other words, we use the same neural function, $N_i[\cdot]$, to provide outputs for the input vectors on levels a_i through b_i. The values of a_i and b_i, and thus the number of neural operators, K, are determined by either of the methods previously discussed (using eq. 5.25 or 5.33). To make a FAST structure out of this, we form a set,

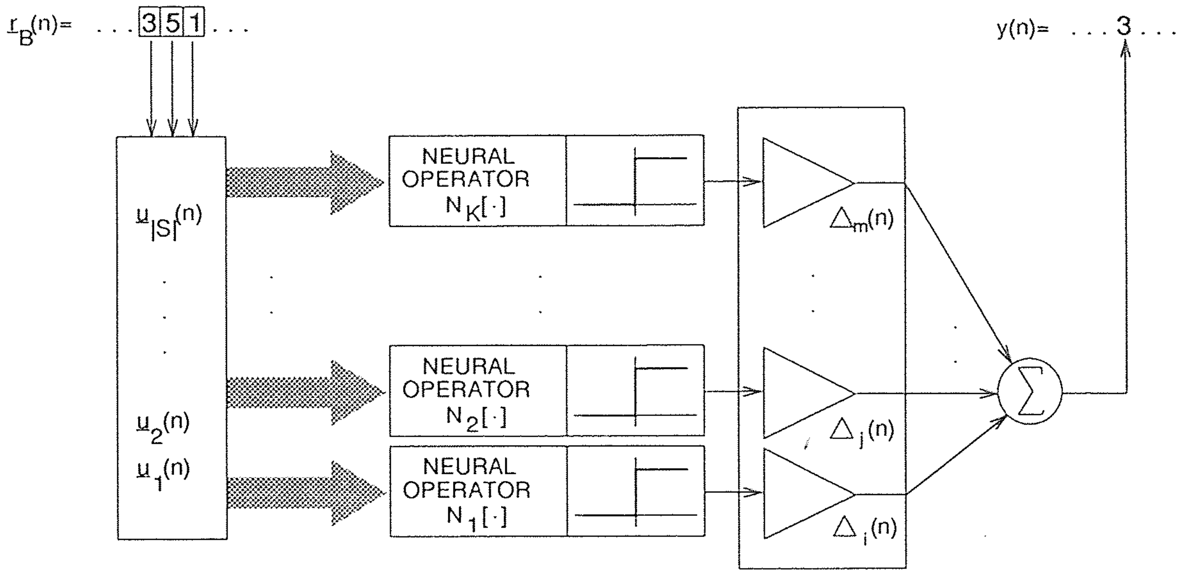
$$\underline{S} = \{r_1(n) \cup \cdots \cup r_B(n) \cup b_1 \cup \cdots \cup b_K\}, \tag{5.44}$$

where $r_j(n)$ are the integer window inputs and b_i are the greatest level numbers processed by the respective neural operator. Since $b_i \ne b_l$ always, there will be anywhere from K elements to $K + B$ elements in set $\underline{S}$. Next, we form set $\underline{Z}$ by ranking the elements in $\underline{S}$.

The output of this FAST-GANF can now be described by

$$y(n) = \sum_{j=1}^{K} \sum_{i=1}^{|\underline{S}|} N_j[\mathbf{u}_i(n)]\Delta_i(n) f(i,j), \tag{5.45}$$

where



$r_B(n)=$ . . . 3 5 1 . . .
y(n)= . . . 3 . . .
$u_{|S|}(n)$
$u_2(n)$
$u_1(n)$
NEURAL OPERATOR $N_K[\cdot]$
NEURAL OPERATOR $N_2[\cdot]$
NEURAL OPERATOR $N_1[\cdot]$
$\Delta_m(n)$
$\Delta_j(n)$
$\Delta_i(n)$
$\sum$

Figure 5.2 FAST-GANF maintaining equivalence to the standard filter.

$$f(i,j) = \begin{cases} 1, & \text{if } a_j \leq R_{(i)} \leq b_j \\ 0, & \text{otherwise.} \end{cases} \tag{5.46}$$

$$\mathbf{u}_i(n) = T^{R_{(i)}(n)}[\mathbf{r}_B(n)], \tag{5.47}$$

$$\Delta_i(n) = R_{(i)}(n) - R_{(i-1)}(n), \tag{5.48}$$

and $R_{(i)}(n)$, $i \in \{1, 2, \ldots, |\underline{S}|\}$ is the ith smallest sample in the set $\underline{S}$, with $N_j[\cdot]$ being neural operator j, $1 \leq j \leq K$.

The structure of this filter is shown in Figure 5.2 for a window size of 3. Here, the neural operators process only the vectors $\mathbf{u}_i(n)$ which are "in their range". The respective $\Delta_j(n)$ values will also vary accordingly. There is no longer one neural function and $\Delta_j(n)$ per $\mathbf{u}_j(n)$ as in the ATD structure. Note also that for this discussion, we consider the neural outputs $\in \{0, 1\}$, to maintain equivalence with the standard GANF.

FAST-GANF filters are similar to the ATD filters, except for $f(i,j)$ and a different set upon which the ranks are based. Note also that the FAST-GANF can be extended to the case where $I \neq 0$ by re-defining the set $\underline{S}$, the input vectors $\mathbf{u}_i(n)$, and the variable $f(i,j)$. However, this would most likely not end up simplifying things. To prove that the FAST-GANF is equivalent to the standard GANF for the case where $I = 0$, we start out by representing a GANF by

$$y(n) = \sum_{i=1}^{M-1} N_i[\mathrm{x}_B^i(n)], \tag{5.49}$$

or

$$y(n) = \sum_{j=1}^{K} \sum_{i=a_j}^{b_j} N_j[\mathrm{x}_B^i(n)]. \tag{5.50}$$

We know that the window is represented by

$$\mathrm{r}_B(n) = \{r_1(n) \;\; r_2(n) \cdots r_B(n)\}, \tag{5.51}$$

so that

$$\mathrm{x}_B^i(n) = \{\mathcal{T}^i[r_1(n)] \;\; \mathcal{T}^i[r_2(n)] \cdots \mathcal{T}^i[r_B(n)]\}. \tag{5.52}$$

Also, if

$$R_{(k-1)}(n) \leq i < R_{(k)}(n), \tag{5.53}$$

and

$$R_{(k-1)}(n) \leq j < r_{(k)}(n), \tag{5.54}$$

then

$$\mathrm{x}_B^i(n) = \mathrm{x}_B^j(n). \tag{5.55}$$

Now, let there exist $r_l(n) \ni \;\; a_j \leq r_l(n) < b_j$, where $l \in \{l_1, l_2, \ldots, l_p\}$ and $r_{l_1}(n) < r_{l_2}(n) < \cdots < r_{l_p}(n)$. Let us now create a set $\underline{S}_j$:

$$\underline{S}_j = \{r_{l_1}(n) \cup r_{l_2}(n) \cup \cdots r_{l_p}(n) \cup b_j\}, \tag{5.56}$$

or

$$\underline{S}_j = \{v_1, v_2, \ldots, v_q\}. \tag{5.57}$$

Then, because of eq. (5.55),

$$\sum_{i=a_j}^{b_j} N_j[\mathrm{x}_B^i(n)] = \sum_{k=1}^{q} N_j[\mathrm{x}^{v_k}(n)][v_k - v_{k-1} + 1], \tag{5.58}$$

where $v_0 = a_j$. Extending this to $N_j[\cdot] \;\forall\; 1 \leq j \leq k$ leads directly to equations (5.44) through (5.48).

5.3 Other Advantages

The obvious benefit of level combinations is that of speed. The level combinations allow the use of FAST structures. However, given a limited set of training data, the simplifications may also result in improved performance. Note that with a completely non-homogeneous GANF, $(M-1)$ neural operators must be trained using N samples of training data. However, at the top and bottom of the stack, there will be many binary inputs consisting of all 0's and all 1's, respectively. Therefore, many of these neural operators will be redundantly trained on these trivial inputs. To make this clearer, we will consider a GANF with $(M-1)$ levels and a window size B. In this case, there will exist a maximum of $(B+1)$ unique binary input vectors after the threshold decomposition operation. If the smallest integer in the window is A and the largest is C, then levels 1 through A will have inputs of $[1 \;\; 1 \cdots 1]$ and levels $(C+1)$ through $(M-1)$ will see inputs of all zeros, $[0 \;\; 0 \cdots 0]$. Here, of course, the vectors $[1 \;\; 1 \cdots 1]$ and $[0 \;\; 0 \cdots 0]$ consist of B elements. As a result, lower levels and upper levels may not experience a number of unique training samples equal to the size of the training data set.

A level, l, will not be trained with anything new at time n if

$$r_k(n) < l \;\; \forall \;\; 1 \leq k \leq B, \tag{5.59}$$

or if

$$r_k(n) \geq l \;\; \forall \;\; 1 \leq k \leq B, \tag{5.60}$$

where $r_k(n)$ are as defined in eq. (5.2).

To show this in a more quantitative manner, let us assume that $r_k(n)$ is uniformly distributed and independent of itself at other time instants. Then

$$P[\text{all elements in } \mathbf{r}(n) < l] = (\frac{l}{256})^B, \tag{5.61}$$

and

$$P[\text{all elements in } \mathbf{r}(n) \geq l] = (\frac{256-l}{256})^B. \tag{5.62}$$

Say, as a worst case, we look at level 1. Then

$$P[\text{all elements in } \mathbf{r}(n) \geq l] = (\frac{255}{256})^B. \tag{5.63}$$

The probability that this level sees an input vector other than $[1 \ \ 1 \cdots 1]$ is

$$1 - (\frac{255}{256})^B. \tag{5.64}$$

The expected number of non-trivial training samples given a training set of length N is

$$N[1 - (\frac{255}{256})^B]. \tag{5.65}$$

We would like this to be greater than or equal to 10 times the VCdim of the network. Therefore,

$$N[1 - (\frac{255}{256})^B] \geq 10\text{VCdim}, \tag{5.66}$$

or

$$N \geq \frac{10\text{VCdim}}{1 - (\frac{255}{256})^B}. \tag{5.67}$$

For a window size $B = 9$, this would be $N \geq 289$ times the VCdim, or in other words, a large number. As a result, the upper and lower levels may not receive enough new training samples to allow for proper generalization. Also, while the training may allow for adequate operation given the statistics of the signal at hand, it may not perform well on different signal distributions. Lack of unique training samples could result in improper generalization of the networks, thereby preventing robust operation. The FAST structures will, however, allow more neural operators to be trained with non-trivial samples at each time instant. As a result, the simplified structures not only increase the speed, but can also increase the filter's performance. This performance increase may be realized on the untrained segments of the same image or on signals with different statistical distributions.

CHAPTER 6
SIMULATIONS

The class of Generalized Adaptive Neural Filters is very broad. Because of this, it is difficult to examine their performance completely. In this section, we look at some GANFs of medium complexity. All of the GANFs considered use relatively small window sizes and none have adjacent levels fed in. Other than these things, however, the GANFs can be considered full blown. Simulations were conducted by filtering noisy images, the details of which will be discussed in section 6.2.

In addition to the GANFs, some other nonlinear filters were applied to the images. These filters are less complicated than the GANFs (in an algorithmic sense) and serve as a baseline with which to compare the GANF. We will start out with a brief summary of these nonlinear filters.

6.1 Comparison Nonlinear Filters and Wiener Filter

Most of the filters to be presented in this section are described in [29]. These filters were applied to images using either 3×3 or 5×5 square windows. At certain times in the filtering, the window extended beyond the edges of the image. To deal with this, the image was assumed to be periodic. In other words, window overhang was filled with image information from the opposite side of the image. As a result, all of the filtering can be considered to be an off-line operation.

The windowed input sequence is defined as

$$\mathbf{U}(n) = \begin{bmatrix} x_{i-W,j-W}(n) & \cdots & x_{i-W,j}(n) & \cdots & x_{i-W,j+W}(n) \\ \vdots & & \vdots & & \vdots \\ x_{i,j-W}(n) & \cdots & x_{i,j}(n) & \cdots & x_{i,j+W}(n) \\ \vdots & & \vdots & & \vdots \\ x_{i+W,j-W}(n) & \cdots & x_{i+W,j}(n) & \cdots & x_{i+W,j+W}(n) \end{bmatrix}. \qquad (6.1)$$

Here, the $(2W+1)\times(2W+1)$ window is centered on pixel (i,j). From this window, we define an input vector as

$$\mathbf{X}^T(n) = [x_1(n) \;\; x_2(n) \;\; \cdots \;\; x_B(n)], \tag{6.2}$$

where B is the number of elements in the window, or $(2W+1)\times(2W+1)$.

If we now arrange these B elements in ascending order, we generate the sequence

$$\mathbf{Z}(n) = [z_1(n) \;\; z_2(n) \;\; \cdots \;\; z_B(n)]. \tag{6.3}$$

The rank of each element in $\mathbf{X}(n)$ is denoted by $\mathbf{R}(n)$, where

$$\mathbf{R}(n) = [r_1(n) \;\; r_2(n) \;\; \cdots \;\; r_B(n)], \tag{6.4}$$

and

$$z_{r_k(n)}(n) = x_k(n). \tag{6.5}$$

Note that $z_1(n) \le z_2(n) \le \cdots \le z_B(n)$.

6.1.1 Alpha-Trimmed Mean Filter

The alpha-trimmed mean filter is based on the order statistics of the windowed input signal. Its operation is described by

$$y(n) = \frac{1}{B - 2\lfloor \alpha B \rfloor} \sum_{\lfloor \alpha B \rfloor + 1}^{B - \lfloor \alpha B \rfloor} z_i(n), \tag{6.6}$$

where α is a constant between 0 and 0.5 and $\lfloor \cdot \rfloor$ is the *greatest integer function.* This filter basically forms the average of a selected portion of elements in the window. A fraction of the smallest and largest elements in the window are thrown away and the arithmetic mean of the remaining elements is computed.

6.1.2 Modified Trimmed Mean Filter

This filter is described by

$$y(n) = \frac{\sum_{i=1}^{B} a_i z_i(n)}{\sum_{i=1}^{B} a_i}, \tag{6.7}$$

where a_i is defined as:

$$a_i \triangleq \begin{cases} 1, & \text{if } |z_i(n) - z_{N+1}(n)| < q, \\ 0, & \text{otherwise.} \end{cases} \tag{6.8}$$

and $B = (2W+1)(2W+1) \triangleq 2N+1$.

Here, samples are included in the average only if they fall within a certain range, q, of the median pixel value. Note that the number of samples included in the average is not fixed.

6.1.3 Double Window Modified Trimmed Mean Filter

This filter is similar to the modified trimmed mean filter except that the median pixel, $z_{N+1}(n)$, is found using a window of size $B = (2N+1)$ and the averaging is done on pixels in a window of size $(2L+1)$. For this filter, $L > N$ always. The operation is described by

$$y(n) = \frac{\sum_{i=1}^{2L+1} a_i z_i(n)}{\sum_{i=1}^{2L+1} a_i}, \tag{6.9}$$

where $z_i(n)$ represents order statistics of the window of size $(2N+1)$.

6.1.4 K-Nearest Neighbor Filter

This filter again computes the arithmetic mean of a subset of pixels in the window. Here, a pixel is included in the averaging if it is one of the K closest (in brightness) to the center pixel, $x_{N+1}(n)$. The operation is described by

$$y(n) = \frac{\sum_{i=1}^{B} a_i x_i(n)}{K}, \tag{6.10}$$

with

$$a_i \triangleq \begin{cases} 1, & \text{if } x_i(n) \text{ is one of the } K \text{ closest to } x_{N+1}(n), \\ 0, & \text{otherwise.} \end{cases} \tag{6.11}$$

There are two versions of this filter. Version 1 includes the center pixel $x_{N+1}(n)$ in the averaging while version 2 does not. Both versions average a total of K pixels.

6.1.5 Modified K-Nearest Neighbor Filter

This filter is identical to the K nearest neighbor, except for the definition of a_i. Here, the K closest values to the *median* pixel are averaged:

$$y(n) = \frac{\sum_{i=1}^{B} a_i x_i(n)}{K}, \tag{6.12}$$

and

$$a_i \triangleq \begin{cases} 1, & \text{if } x_i(n) \text{ is one of the } K \text{ closest to } z_{N+1}(n), \\ 0, & \text{otherwise.} \end{cases} \tag{6.13}$$

Again, two versions are defined: Version 1 includes the median pixel while version 2 does not.

6.1.6 Wilcoxon Filter

The Wilcoxon filter has an output described by the following equation:

$$y(n) = \text{med}\{\frac{x_i(n) + x_j(n)}{2} \mid i, j\}, \tag{6.14}$$

where med[$\cdot$] is the median operation, and i and j are taken over all possible values with i and j in the same row or column. Two versions of this filter are defined. Version 1 allows i and j to be equal, while version 2 excludes these cases from the med[$\cdot$] operation.

6.1.7 Adaptive Mean Filter

The output of this filter is described by

$$y(n) = \frac{\sum_{i=1}^{B} a_i x_i(n)}{\sum_{i=1}^{B} a_i}, \tag{6.15}$$

where

$$a_i \triangleq \begin{cases} 1, & \text{if } |x_i(n) - x_{N+1}(n)| \leq C, \\ 0, & \text{otherwise.} \end{cases} \tag{6.16}$$

Note that the samples included in the averaging must be within C of the center pixel.

6.1.8 Adaptive Median Filter

This filter is similar to the adaptive mean filter, except the arithmetic average operation is replaced with a median operation:

$$y(n) = \text{med}[x_i(n) \;\mid\; x_i(n) \in S], \tag{6.17}$$

where

$$x_i(n) \in S, \;\text{ if }\; |x_i(n) - x_{N+1}(n)| \leq C, \tag{6.18}$$

and

$$x_i(n) \notin S, \;\text{ if }\; |x_i(n) - x_{N+1}(n)| > C, \tag{6.19}$$

and

$$i = 1, 2, \ldots, B. \tag{6.20}$$

6.1.9 Conventional Median Filter

This is a very simple filter. Its output is described by

$$y(n) = \text{med}[x_i(n)] \;\; \forall i \in \{1, 2, \ldots, B\}, \tag{6.21}$$

or

$$y(n) = z_{N+1}(n). \tag{6.22}$$

It is simply the median of all the samples in the window.

6.1.10 Separate Median Filter

The output of this filter is the median of the medians along all of the rows. It operation is described by

$$y(n) = \text{med}[v_1, v_2, \ldots, v_{2W+1}], \tag{6.23}$$

where

$$v_i = \text{med}[u_{i,1}(n), u_{i,2}(n), \ldots, u_{i,2W+1}(n)]. \tag{6.24}$$

Here, $u_{i,j}$ are the elements in the window of eq. (6.1).

6.1.11 Max/Median Filter

The output of this filter is described by

$$y(n) = \max[v_1, v_2, v_3, v_4], \tag{6.25}$$

where

$$v_1 = \mathrm{med}[x_{i,j-W}(n), \ldots, x_{i,j}(n), \ldots, x_{i,j+W}(n)], \tag{6.26}$$

$$v_2 = \mathrm{med}[x_{i-W,j}(n), \ldots, x_{i,j}(n), \ldots, x_{i+W,j}(n)], \tag{6.27}$$

$$v_3 = \mathrm{med}[x_{i-W,j-W}(n), \ldots, x_{i,j}(n), \ldots, x_{i+W,j+W}(n)], \tag{6.28}$$

$$v_4 = \mathrm{med}[x_{i-W,j+W}(n), \ldots, x_{i,j}(n), \ldots, x_{i+W,j-W}(n)]. \tag{6.29}$$

6.1.12 Wiener Filter

The Wiener filter is a linear filter which minimizes the MSE between the output and a desired response [5]. As discussed in the beginning of this thesis, the Wiener filter is the optimal filter if the clean signal and the input signal are jointly Gaussian [2]. In other cases, it is the best *linear* filter, but a nonlinear filter may do a better job. The output of a Wiener filter is described by

$$y(n) = \sum_{k=1}^{M} w_k^* u(n-k+1), \tag{6.30}$$

where M is the filter order, w_i are constants and $u(j)$ is the process at the input of the filter. If we define

$$\mathbf{w}^T = [w_1 \ \ w_2 \ \ \cdots \ \ w_M], \tag{6.31}$$

and

$$\mathbf{u}^T(n) = [u(n) \ \ u(n-1) \ \ \cdots \ \ u(n-M+1)], \tag{6.32}$$

then

$$y(n) = \mathbf{w}^H \mathbf{u}(n). \tag{6.33}$$

We wish to find the weight vector, $\mathbf{w}$, which minimizes the mean square error between the filter output and a desired response:

$$J(\mathbf{w}) = E[e(n)e^*(n)], \tag{6.34}$$

$$J(\mathbf{w}) = E[(s(n) - \mathbf{w}^H\mathbf{u}(n))(s(n) - \mathbf{w}^H\mathbf{u}(n))]. \tag{6.35}$$

Here, $s(n)$ is the desired response. To find the optimal weight vector, the gradient of eq. (6.35) can be found.

$$\nabla = \frac{dJ(\mathbf{w})}{d\mathbf{w}} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w}, \tag{6.36}$$

where the vector $\mathbf{p}$ is the cross correlation between the windowed input process and the desired output, and R is the input autocorrelation matrix. These two parameters are specified below:

$$\mathbf{p} = E[\mathbf{u}^T(n)s^*(n)], \tag{6.37}$$

$$\mathbf{p} = E[[u(n) \;\; u(n-1) \;\; \cdots \;\; u(n-M+1)]^T s^*(n)]. \tag{6.38}$$

$$\mathbf{R} = E[\mathbf{u}(n)\mathbf{u}^H(n)], \tag{6.39}$$

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & \cdots & r(M-1) \\ r(-1) & r(0) & \cdots & r(M-2) \\ . & . & \cdots & . \\ . & . & \cdots & . \\ . & . & \cdots & . \\ r(-M+1) & r(-M+2) & \cdots & r(0) \end{bmatrix}. \tag{6.40}$$

where $r(k)$ is defined by

$$r(k) = E[u(n)u^*(n-k)]. \tag{6.41}$$

Here, we assume that the processes are stationary.

It can be shown that setting the gradient to zero provides the solution for the optimal weight vector [5]. This weight vector produces the minimum MSE that can be achieved with an Mth order linear filter.

$$-2\mathbf{p} + 2\mathbf{R}\mathbf{w} = 0, \tag{6.42}$$

$$\mathbf{R}\mathbf{w} = \mathbf{p}, \tag{6.43}$$

$$\mathbf{w} = \mathbf{R}^{-1}\mathbf{p}. \tag{6.44}$$

In order to implement the filter, expected values have to be estimated. First of all, our signals are real, so

$$r(k) = r(-k). \tag{6.45}$$

Then, the $r(k)$ values in eq. (6.40) can be estimated by

$$r(k) = \frac{1}{N-k}\sum_{n=1}^{N-k} u(n)u(n-k), \tag{6.46}$$

where N is the length of training data and $k = 0, 1, \ldots, M$.
The vector $\mathbf{p}$ could be estimated in a similar manner.

To implement this filter for image processing, the statistics were determined using the upper left hand corner of the images. A square window was used to define the input vector $\mathbf{u}(n)$, and from this, $\mathbf{p}$ and $\mathbf{R}$ were determined. Then the weight vector was found using eq. (6.44). After this, the entire image was filtered using these weights in eq. (6.30) or eq. (6.33).

6.2 Generalized Adaptive Neural Filters

In order to examine the filtering ability of the GANF, four noisy images were created. All of these were produced by adding noise to the clean image shown in Figure 6.1 and clipping where necessary. Since the image had been digitized with eight bits of precision, all pixel values must range between 0 and 255. If additive noise produced a pixel value less than zero or greater than 255, these pixels were assigned the values of zero or 255, respectively. Figures 6.2 and 6.3 resulted from adding epsilon mixtures of noise to the clean image. This noise was generated using the following probability density function:

$$P(x) = (1-\epsilon)\phi(\frac{x}{\sigma_1}) + \epsilon\phi(\frac{x}{\sigma_2}), \tag{6.47}$$

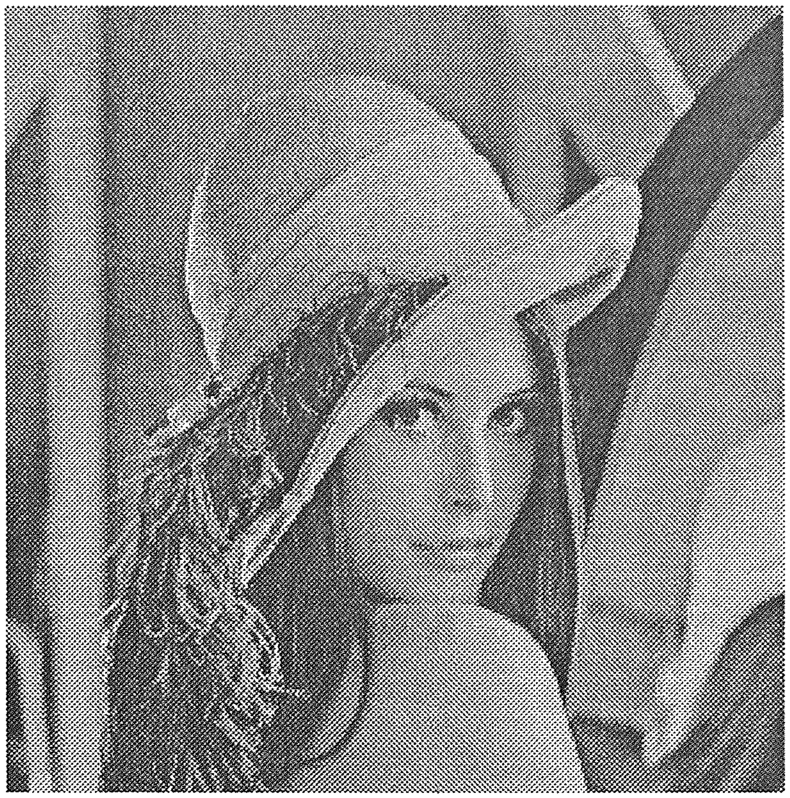


Figure 6.1 The clean image.

where $\phi(x)$ is the probability density function of a Gaussian random variable with zero mean and unit variance. In this equation, σ_1 is made small to represent a thermal background noise, and this occurs with probability $(1-\epsilon)$. σ_2 is made large to represent impulsive noise occurring with a probability ϵ. The two images with mixture noise differ in the parameters used in the noise generation equation. Figure 6.2 has a small amount of noise, created by using $\epsilon = 0.8$, $\sigma_1 = 2.5$ and $\sigma_2 = 50$ in eq (6.47). Figure 6.3 contains a large amount of noise, with $\epsilon = 0.8$, $\sigma_1 = 5$ and $\sigma_2 = 140$.

In addition to these images, two more were created with a different type of noise added. These images are shown in Figures 6.4 and 6.5, also representing small and large amounts of noise. These images were generated by adding zero mean Gaussian noise to the clean image and clipping where necessary. The image in Figure 6.4 used a σ of 50, while the other one had a σ of 140. To make things easier, we will refer

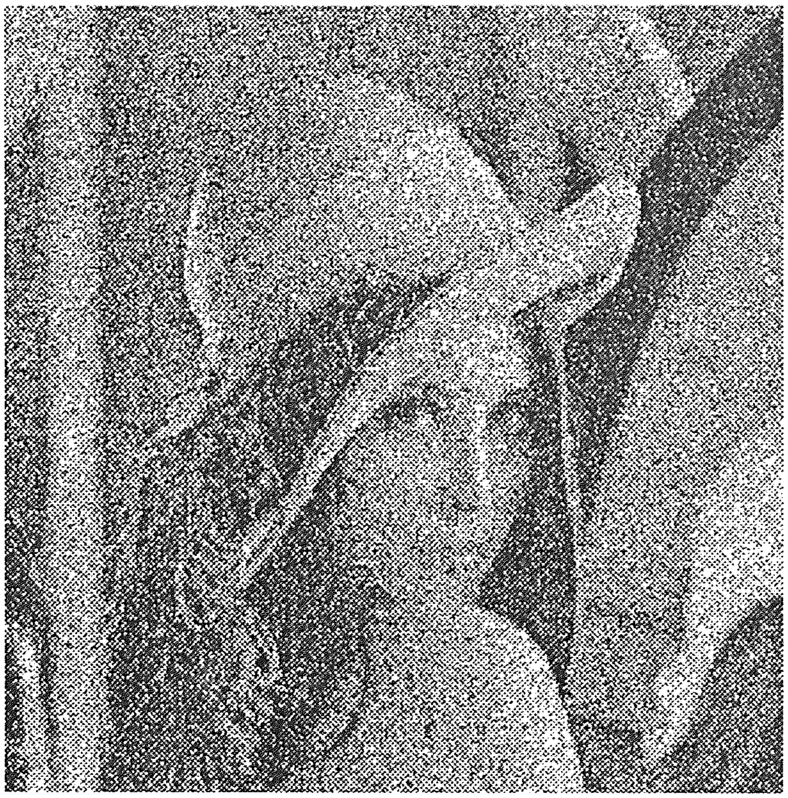


Figure 6.2 The image with a small amount of mixture noise.

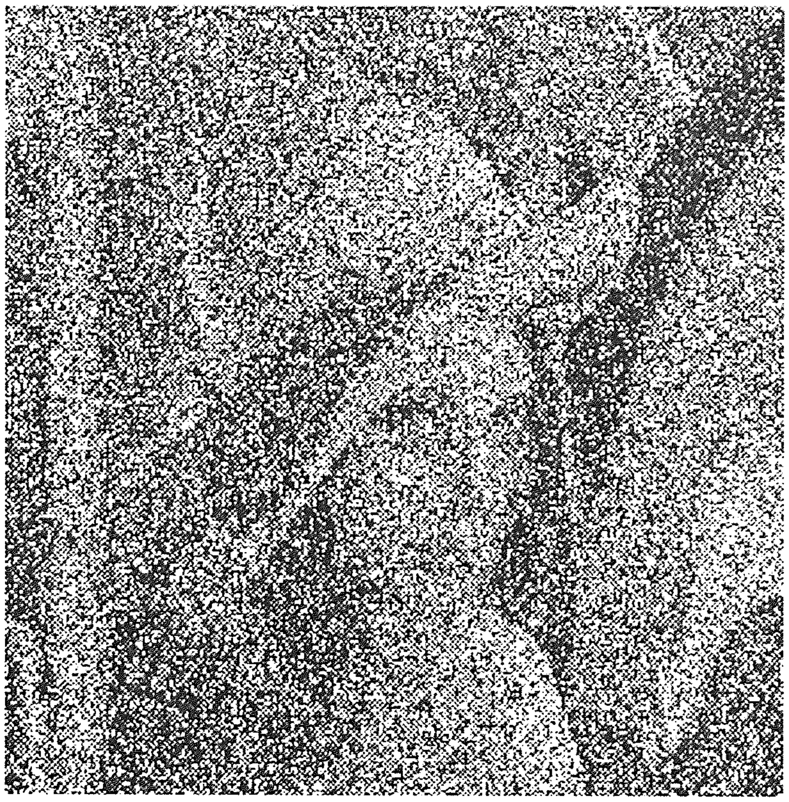


Figure 6.3 The image with a large amount of mixture noise.

to the images of Figures 6.2 through 6.5 as images 1 through 4, respectively. The mean absolute errors (MAE), mean squared errors (MSE) and signal to noise ratios (SNR) of these images are provided in Table 6.1. Here, all of the statistics were computed over the lower right hand three-quarters of the images. Since the Wiener filter and the GANFs were trained on the upper left hand quarters of the images, these sections were disregarded in all of the error and power calculations. The MAEs were computed using

$$MAE = \frac{\sum_i \text{clean}(i) - \text{noisy}(i)}{\sum_i 1}, \tag{6.48}$$

where clean(i) is the value of pixel i in the clean image, noisy(i) is the value of the corresponding pixel in the noisy image, and i is taken over quadrants I, III and IV of the images. The MSEs were computed with

$$MSE = \frac{\sum_i [\text{clean}(i) - \text{noisy}(i)]^2}{\sum_i 1}, \tag{6.49}$$

Finally, the SNRs were computed using the formula,

$$SNR = 10 \log \frac{\sum_i \text{clean}^2(i)}{\sum_i [\text{clean}(i) - \text{noisy}(i)]^2}. \tag{6.50}$$

Table 6.1 Statistics on test images.

Image Number	Description	MAE	MSE	SNR [dB]
—	Clean image	0	0	inf
1	Mixture noise, $\epsilon = 0.8, \sigma = 2.5$	30.87	1754.38	10.12
2	Mixture noise, $\epsilon = 0.8, \sigma = 50$	64.46	7067.66	4.07
3	Gaussian noise, $\sigma = 50$	38.29	2227.20	9.08
4	Gaussian noise, $\sigma = 140$	79.39	8806.64	3.11

All of these images were processed by the filters in section 6.1 to provide a performance baseline. Results were obtained using window sizes of 3×3 and 5×5 as shown in Tables C.1 through C.8 in Appendix C. The first four tables show the results on images 1 through 4 for a window size of 3×3. The next four show the results for a 5×5 window.

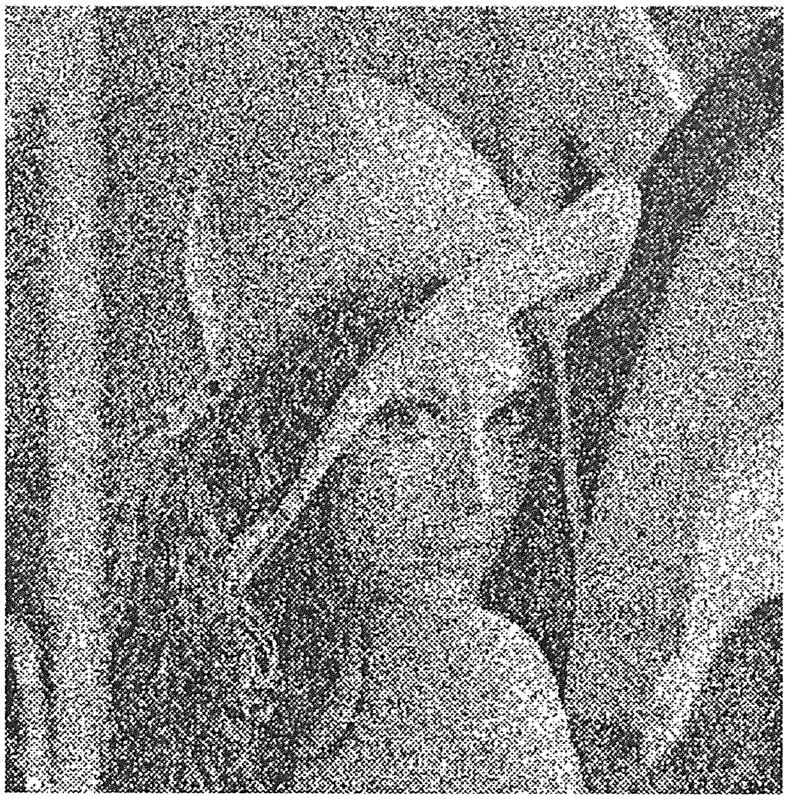


Figure 6.4 The image with a small amount of Gaussian noise.

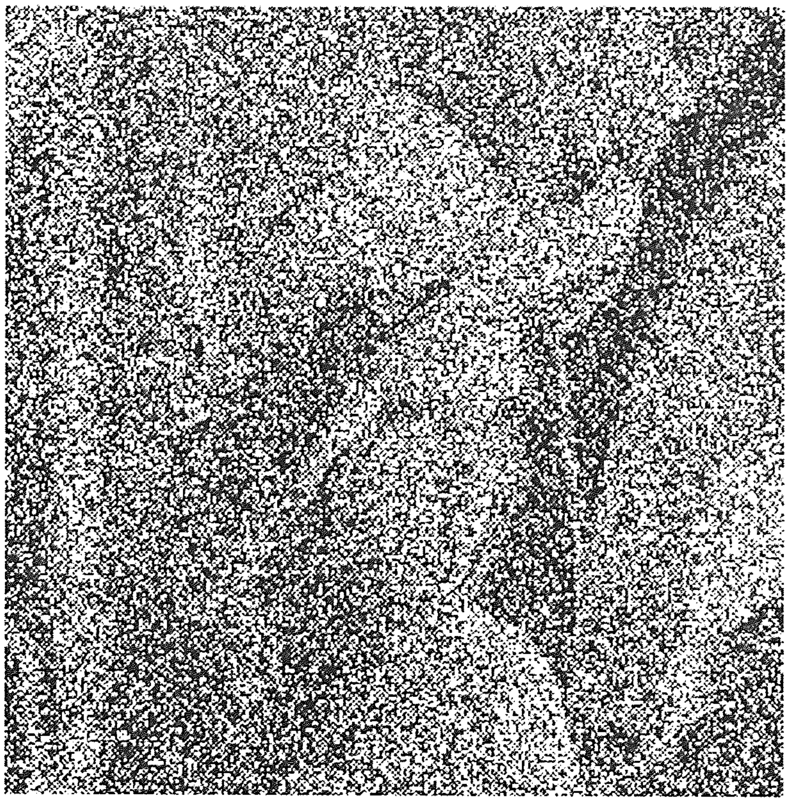


Figure 6.5 The image with a large amount of Gaussian noise.

Next, several GANFs were set up to process the same four images. The details of the GANFs used are discussed in the following subsections.

6.2.1 Completely Non-Homogeneous GANF

The first type of GANF used in the simulations appears as shown in Figure 3.1. This time, though, there were 255 levels (since $M = 256$) and window sizes of $B = 9$ and $B = 25$, corresponding to 3×3 and 5×5 windows. Since this GANF was completely non-homogeneous, a separate neural function was provided on each of the 255 levels. For this filter, no adjacent levels were fed in, so each neural operator receives either 9 or 25 inputs. A total of six filters were constructed with this layout. They differed only in the neural operators used. Each of these filters made use of one of the six neural operators discussed in section 4.4. For the 3×3 window size, simulations were conducted for all six of the GANFs. However, only the quadric neuron was used for processing with a 5×5 window. The results for 3×3 and 5×5 windows are provided in Appendix C in Tables C.9 through C.12.

6.2.2 Homogeneous GANF

The second GANF structure considered was identical to that used in section 6.2.1, but used only one neural operator to process all of the levels. In other words, this GANF was homogeneous. Again, for a 3×3 window, all four images were processed using four of the six neural operators. (Due to time constraints, two of the neural operators were not implemented.) For the 5×5 window size, only the GANF with a quadric neuron was used. The results using the homogeneous GANFs are shown in Tables C.13 through C.16 in Appendix C.

6.2.3 The FAST-GANF

Finally, FAST-GANF structures were used to filter the four images. The first FAST-GANF was set up using Method 1 to determine the number of independent neural operators. The second FAST-GANF used Method 2 to determine the structure. Both of these filters were used with quadric neurons to filter the four test images. These filters used a window size of 3×3, as shown in Table C.17 in Appendix C.

CHAPTER 7

ANALYSIS AND CONCLUSIONS

The four images described in Chapter 7 provide reasonably diverse filtering assignments for the GANFs and the comparison filters. Both small and large amounts of noise are simulated, for two entirely different noise distributions. The best results for the comparison filters using a window size of 3×3 are shown in Table 7.1, while Table 7.2 shows the results for a 5×5 window. For now, we will only discuss the results using the 3×3 window. As shown in the table, the comparison filters did quite well for all four images. It should be noted that these comparison filters were much faster and easier to implement (in algorithmic form) than the GANFs. However, in order to achieve the results shown here, various filter parameters needed to be carefully adjusted in some cases. Parameters which produced good results for some types of noise produced miserable results for others. In other words, a lot of user customization was required to produce good results (in most cases).

The results for the non-homogeneous GANFs using a 3×3 window are shown in Table 7.3. By comparing these results to those shown in Table 7.1, it can be seen that the best GANFs did a better job for the mixture noise (images 1 and 2), and did almost as good as the comparison filters for the Gaussian noise. For the large mixture noise (image 2), all GANFs except for the RBF did a measurable amount better than the comparison filters. However, for the rest of the images, many of the GANFs performed worse than the other filters. In other words, except for image 2, some GANFs were exceptional, while others were out-performed by the simpler comparison filters. Figure 7.1 shows the difference in SNR between the best GANFs and the best comparison filters.

In addition to the statistical results, we can also look at the filter outputs subjectively. Figure 7.2 duplicates the clean and noisy images presented in the

Table 7.1 Best comparison filters, (3 × 3) window.

Image	Filter Name	Window Size	Parameters	MAE	MSE	SNR [dB]
1	α-Trimmed Mean	3×3	$\alpha = 0.4$		297.13	17.83
	Adaptive Median	3×3		12.35		
2	α-Trimmed Mean	3×3	$\alpha = 0$		1168.98	11.88
			$\alpha = 0.5$	26.25		
	Mod. Trimmed Mean	3×3	$q = 1$	26.25		
	k-nearest Neighbor v.1	3×3	$k = 9$		1168.98	11.88
			$k = 9$		1168.98	11.88
	Mod. k-nearest Neighbor v.1	3×3	$k = 1$	26.25		
	Conventional Median	3×3		26.25		
3	Double Window MTM	$N = 0$ $L = 1$	$q = 212$	14.94	365.31	16.94
	Adaptive Mean	3×3	$C = 212$			16.94
4	Mod. k-nearest Neighbor v.2	3×3	$k = 8$	30.34	1419.14	11.04

Table 7.2 Best comparison filters, (5 × 5) window.

Image	Filter Name	Window Size	Parameters	MAE	MSE	SNR [dB]
1	Adaptive Median	5×5	$C = 137$		216.15	19.22
			$C = 150$	9.80		
2	α-Trimmed Mean	5×5	$\alpha = 0.45$		564.88	15.04
	Mod. Trimmed Mean	5×5 $q = 20$		15.39		
3	Double Window MTM	$N = 1$ $L = 2$	$q = 125$	12.44	282.80	18.05
			$q = 137$	12.44		
4	α-Trimmed Mean	5×5	$\alpha = 0.15$	23.72	896.34	13.04

Table 7.3 Best completely non-homogeneous GANF filters.

Image	Filter Name	Window Size	Parameters	MAE	MSE	SNR [dB]
1	Large 2-Layer	3×3	$\mu = 0.1$	12.17	277.68	18.13
2	Minimal 2-Layer	3×3	$\mu = 0.9$	22.34		
			$\mu = 0.5$		983.55	12.64
3	Polynomial DF	3×3	$\alpha = 0.0001$	15.55	398.96	16.55
4	Polynomial DF	3×3	$\alpha = 0.00005$	30.53	1428.47	11.01

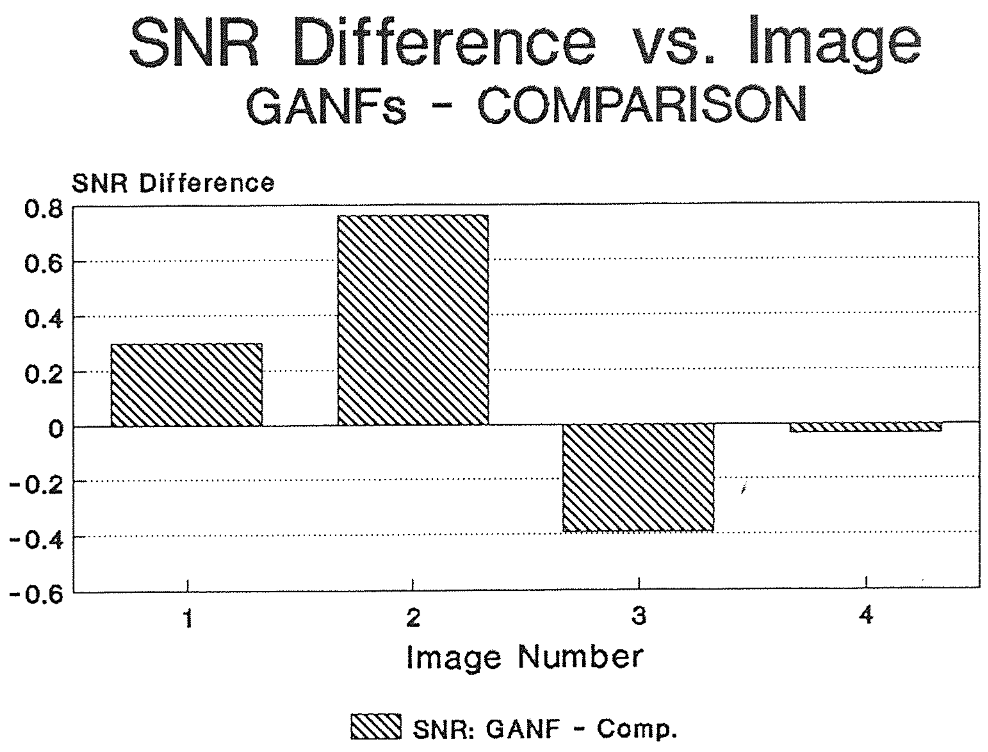
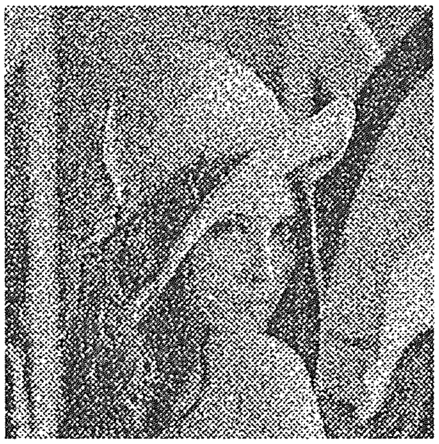


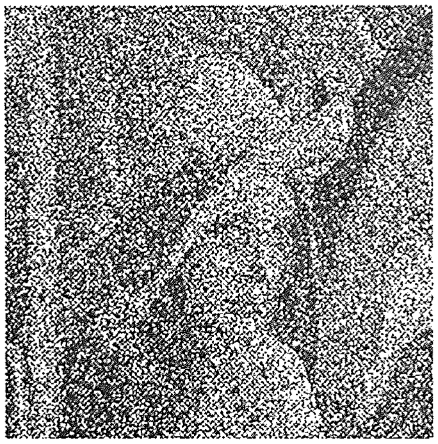
Figure 7.1 SNR difference vs. image for best GANFs and best comparison filters.

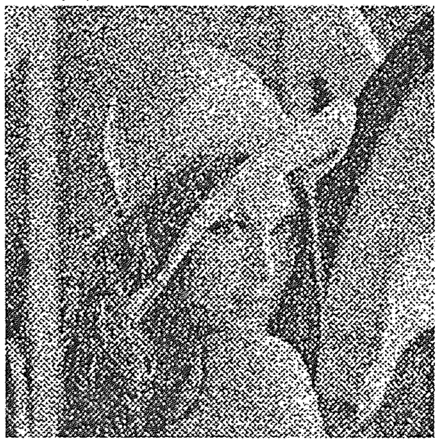
previous chapter. Figure 7.3 shows how the image with small mixture noise looked after filtering. Shown are the outputs of the best comparison filters and some GANFs. Unfortunately, the result from the GANF with the large 2 layer net was unable to be shown. Figure 7.4 shows the best comparison filter outputs along with some GANF outputs for the image with large mixture noise. Figures 7.5 and 7.6 show the important results for the images with Gaussian noise. While subjective impressions may vary, it can be stated that most of the outputs shown are very close in image quality.

The lack of a clearly superior output may appear to indicate that the GANF has limited usage, but a careful analysis of the data reveals something else. Even though in many cases the comparison filters did better than the GANFs, the difference was not great. Also, many comparison filters did great on some images, but performed poorly on others. For example, while the conventional median did a good job on images 1 and 2, it did much worse than the GANFs when used on images 3 and 4.

(a) The clean image


(b) Small mixture noise


(c) Large mixture noise.


(d) Small Gaussian noise

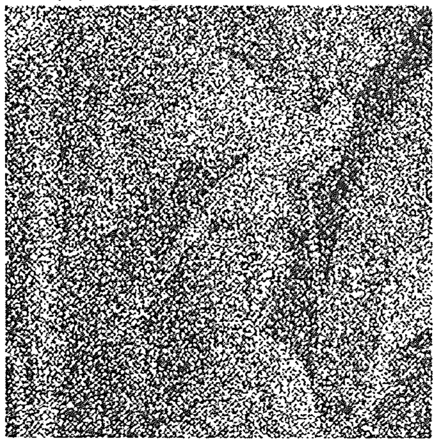
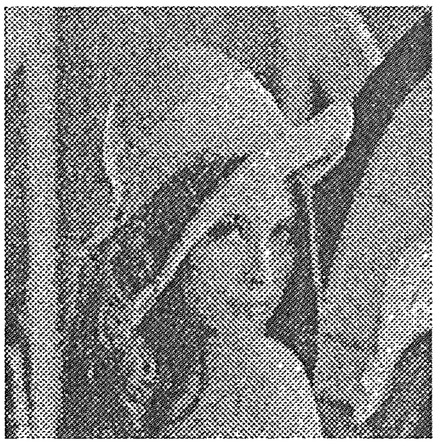
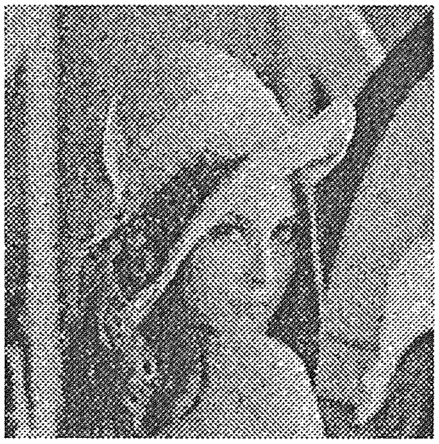
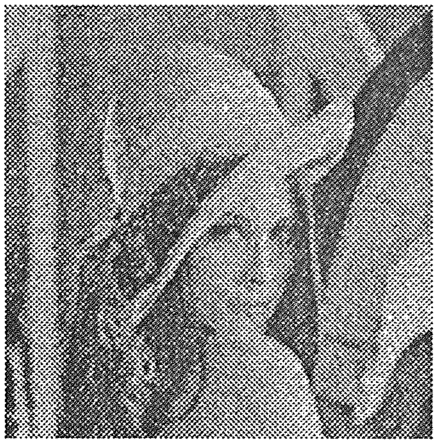
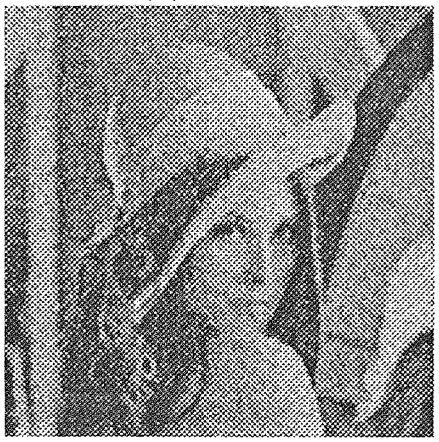

(e) Large Gaussian noise.

Figure 7.2 Input images.


(a) α-trimmed mean.


(b) Median


(c) GANF, single.


(d) GANF, quadric

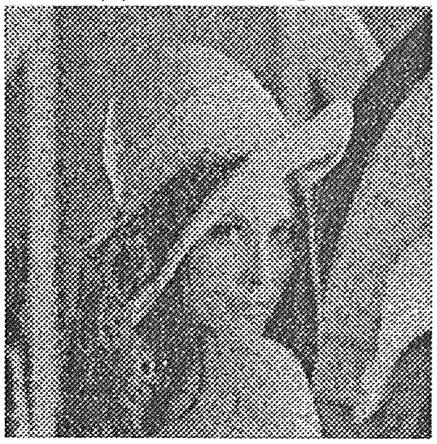
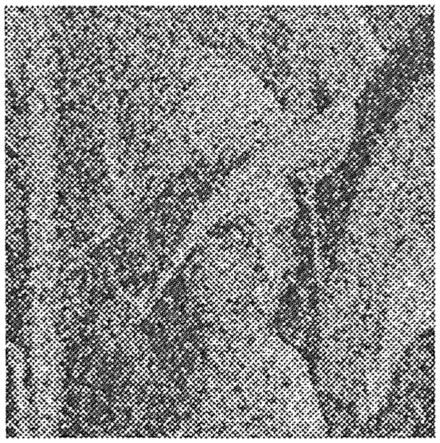
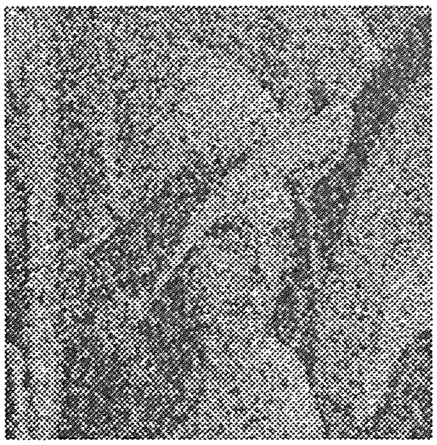
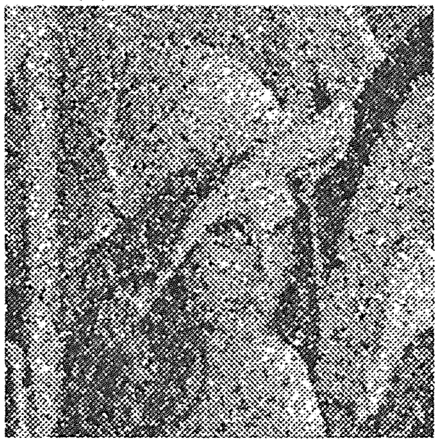
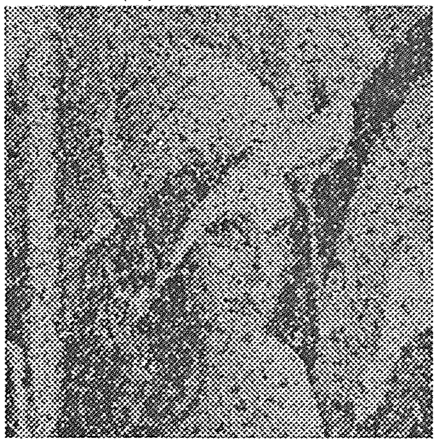

(e) FAST-GANF, quadric.

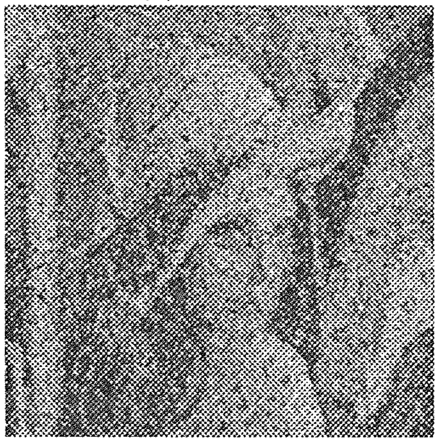
Figure 7.3 Output images for small mixture noise (IMAGE 1).


(a) α-trimmed mean.


(b) KNN ver.1


(c) Median.


(d) GANF, small 2-lyr.


(e) GANF, quadric.

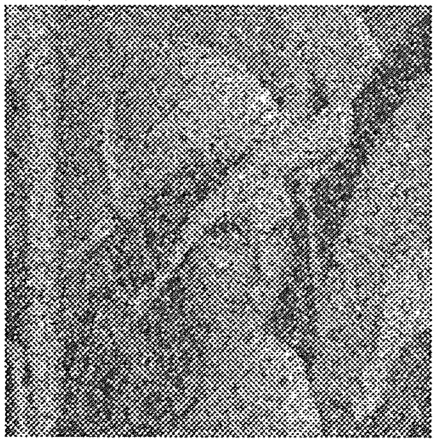
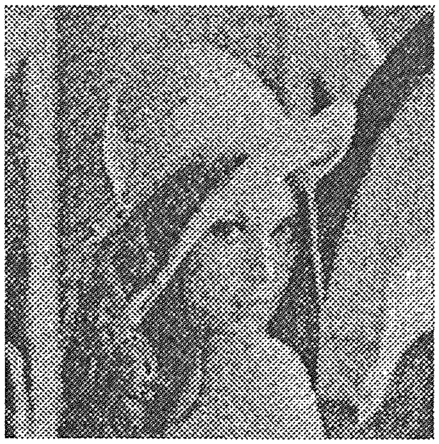
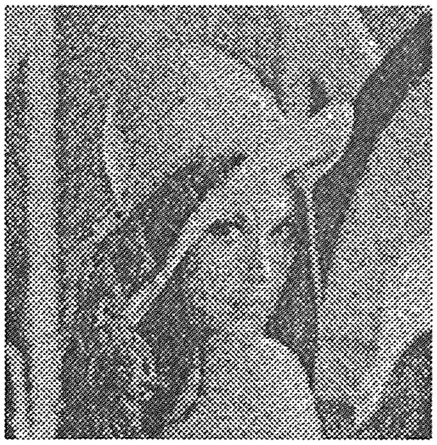
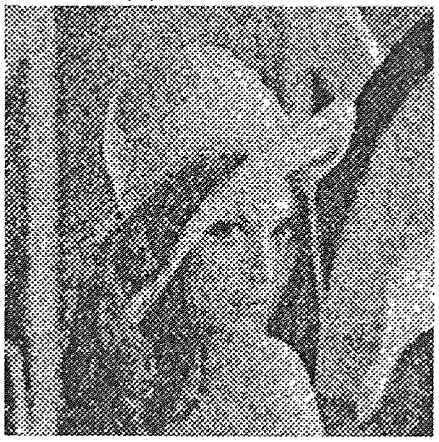
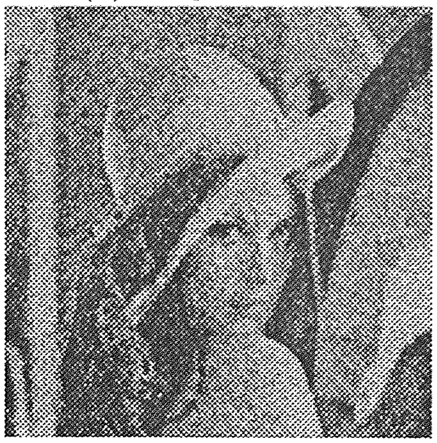

(f) FAST-GANF, quadric.

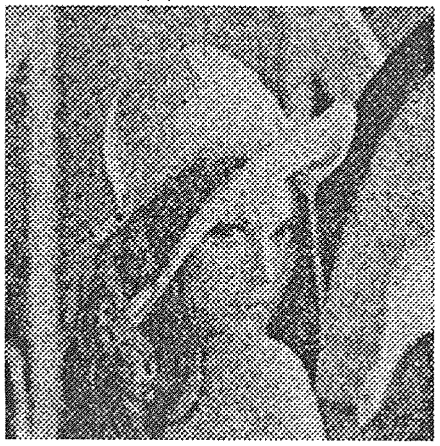
Figure 7.4 Output images for large mixture noise (IMAGE 2).


(a) DWMTM.


(b) Adaptive mean.


(c) Median.


(d) GANF, polynomial.


(e) GANF, quadric.

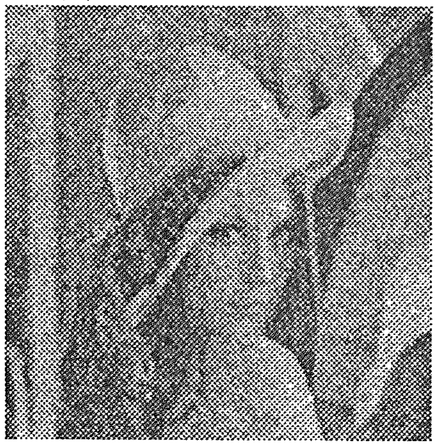
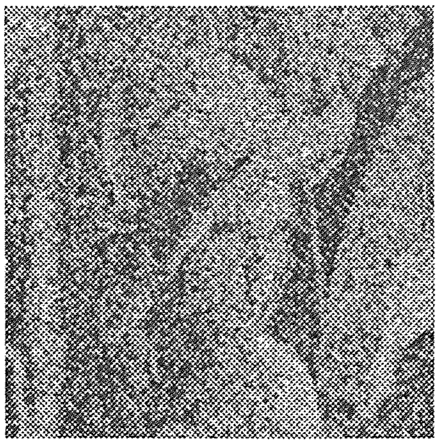
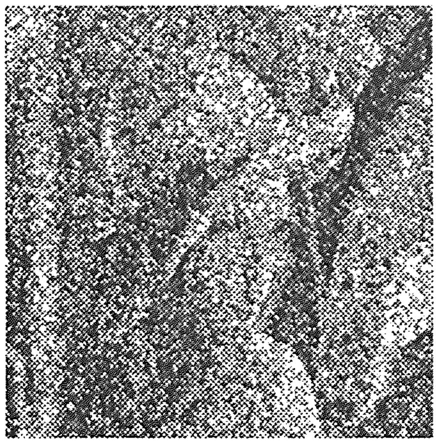
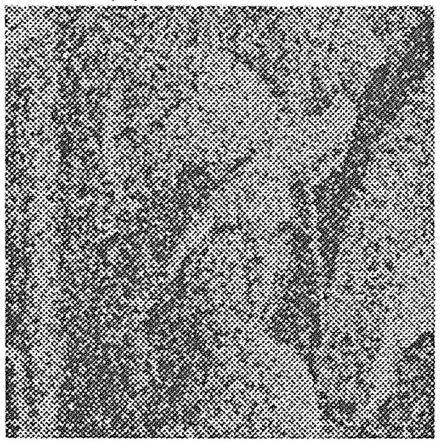
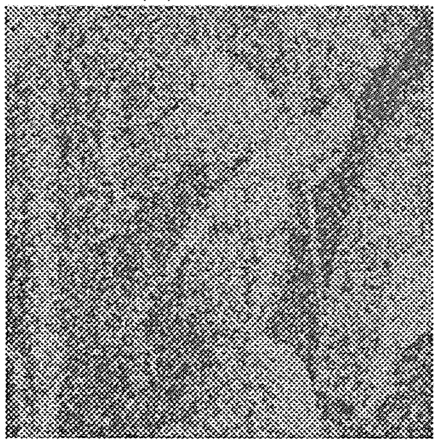

(f) FAST-GANF, quadric.

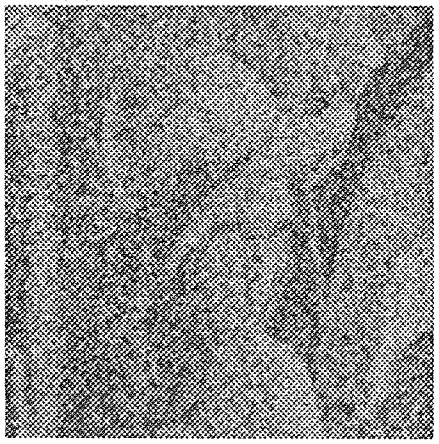
Figure 7.5 Output images for small Gaussian noise (IMAGE 3).


(a) MKNN ver.2


(b) Median.


(c) GANF, polynomial.


(d) GANF, quadric.


(e) GANF, single.

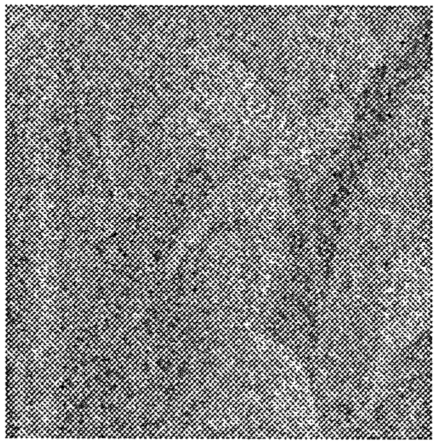

(f) FAST-GANF, quadric

Figure 7.6 Output images for large Gaussian noise (IMAGE 4).

It should be noted that the same GANF structure was used in filtering all of the images. This indicates that the GANF can set itself up to perform reasonably well when confronted with widely ranging noise types. It also seems likely that the GANF would be able to adapt to noise types which were not considered here. Therefore, the GANF's performance warrants its use as a filter in unknown or non-stationary environments. While other filters may perform better in selected cases, the GANF appears to have the best overall performance.

So far, we have looked only at the completely non-homogeneous GANF, and have considered its performance in general. The results of processing the images with the homogeneous GANFs are presented in Table 7.4. In some cases, we see that the homogeneous GANFs have increased performance over their non-homogeneous counterparts. In other cases, (especially for image 4), the homogeneous GANFs performed quite poorly. From this we can see that it is probably better to combine levels in the GANF based on some criteria, as in the FAST-GANF. Although not investigated here, it could be possible that the homogeneous filters perform better when applied to signals of different statistics.

Table 7.4 Best homogeneous GANF filters.

Image	Filter Name	Window Size	Parameters	MAE	MSE	SNR [dB]
1	Single neuron	3×3	$\alpha = 0.000001$	12.58	325.83	17.43
2	Single neuron	3×3	$\alpha = 0.00005$	26.25	1510.61	10.77
3	Single neuron	3×3	$\alpha = 0.000005$	18.20	530.68	15.32
4	Single neuron	3×3	$\alpha = 0.00005$	43.86	2923.69	7.90

Next, we can look at the complexity, capacity and generalization versus performance. By far the single neuron was the simplest of the structures. In fact, despite its limitations, it performed quite well except for image 3. In many other cases, it performed just as well or even better than nets with higher capacities. Figure 7.7 shows the SNR difference between the best GANFs and GANFs with quadric and linear discriminant functions. By comparing the single neuron, quadric neuron

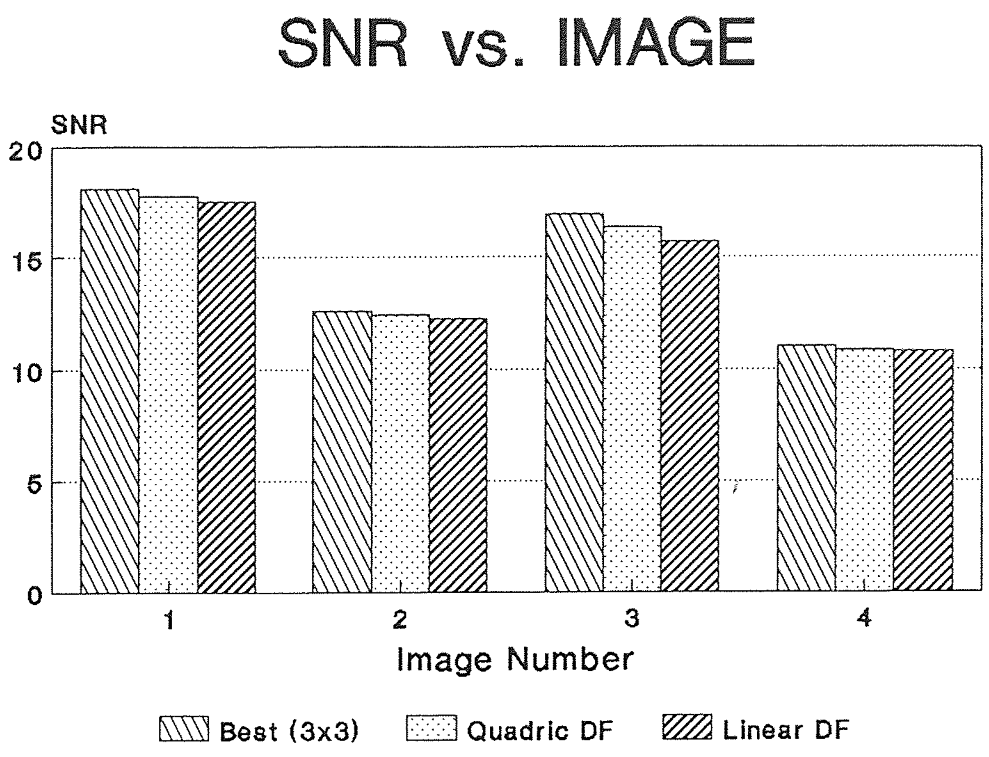


Figure 7.7 SNR vs. image for GANFs.

and polynomial neuron, it is clear that the capacity makes a difference only for images 3 and 4. Even for this Gaussian noise, the results can be tolerated. However, probably the best overall performer with tolerable complexity is the quadric neuron. Figures 7.8 and 7.9 show how the GANF with the quadric neuron performed versus the best comparison filters and the median filter. It performed comparably to the best filters and outperformed the median in all cases. The two layer nets did alright in some cases, but were much too slow to be practical. Finally, the radial basis function did not perform well at all. This is probably due to the use of only nine elements in the first layer. Some future efforts may be concerned with adaptively configuring this first layer of the RBF network.

To use the GANF in a practical situation, a good compromise would be the use of a quadric neuron. Smaller window sizes would be desirable at the start of training to achieve proper generalization. Then, the window could be expanded to achieve increased performance. Although they suffer from slightly degraded performance, in

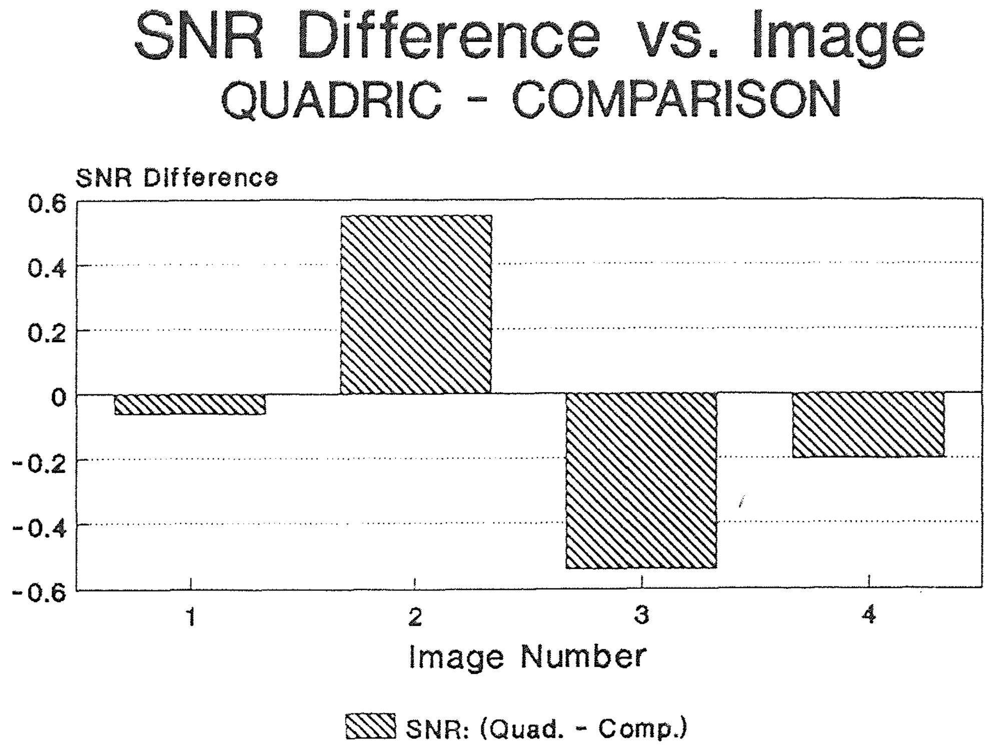


Figure 7.8 SNR difference vs. image for quadric GANF and comparison filters.

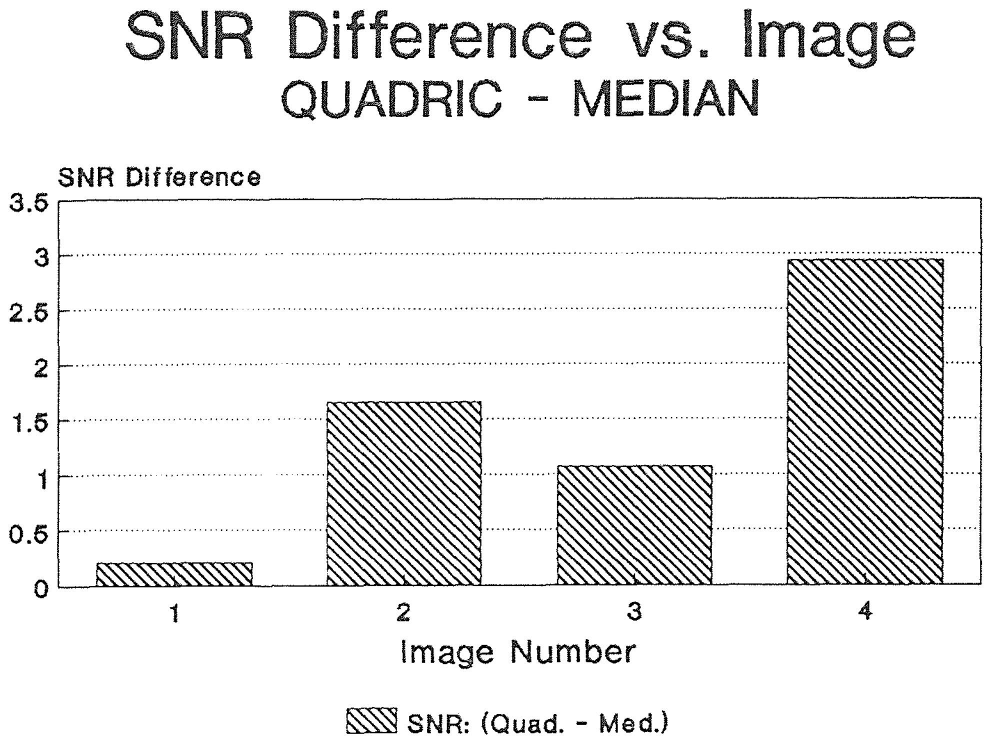


Figure 7.9 SNR difference vs. image for quadric GANF and median filter.

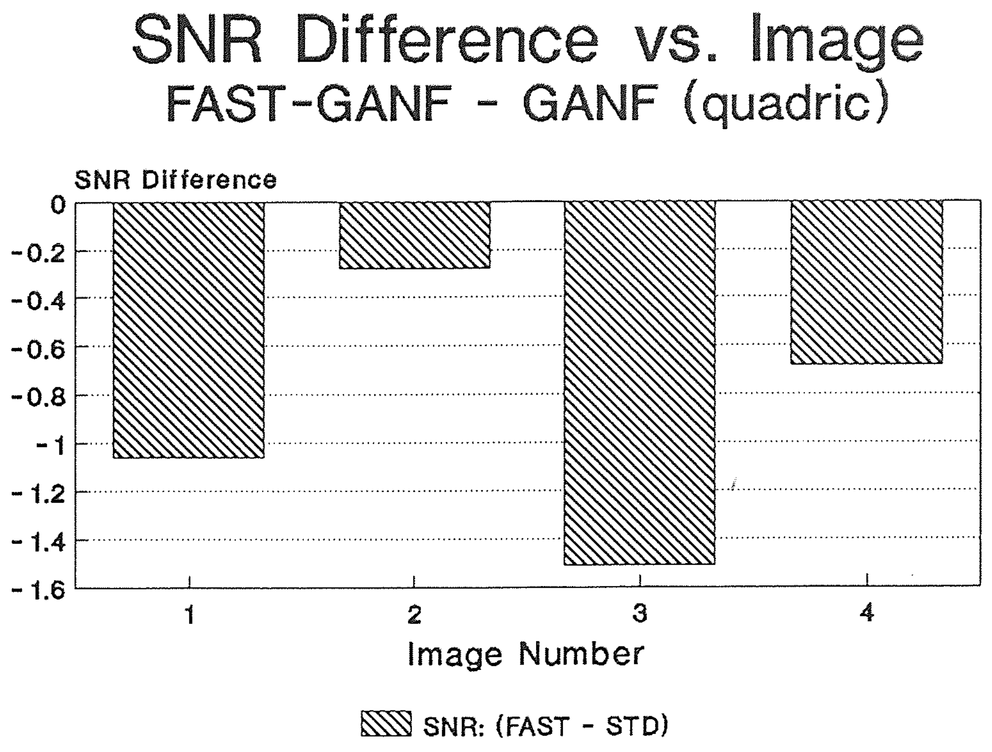


Figure 7.10 SNR difference vs. image for FAST-GANF and standard-GANF.

most cases the FAST-GANF would be the best way to train the GANF. Figures 7.10 and 7.11 compare the performance and speed differences of the standard and FAST-GANFs (quadric). Figure 7.11 shows the training time of the standard and FAST structures with 42 and 1 neural functions. It also provides the training time of a FAST structure set up by the user to implement a homogeneous filter. It can be seen that the FAST structures reduce the training time to about half of the standard training time. To further increase the usage of the GANF, any other improvements in speed would also be welcome. Also, it is important to point out that adjacent levels cannot realistically be involved in the neural inputs at this time. Adjacent inputs would slow things down by a factor of $(2I+1)$, and would greatly increase the VCdim of the networks used. As a result, massive amounts of training data would be required to achieve needed generalization.

Finally, some simulations were conducted using GANFs with 5×5 windows. However, due to time constraints, requests from other computer users and technical

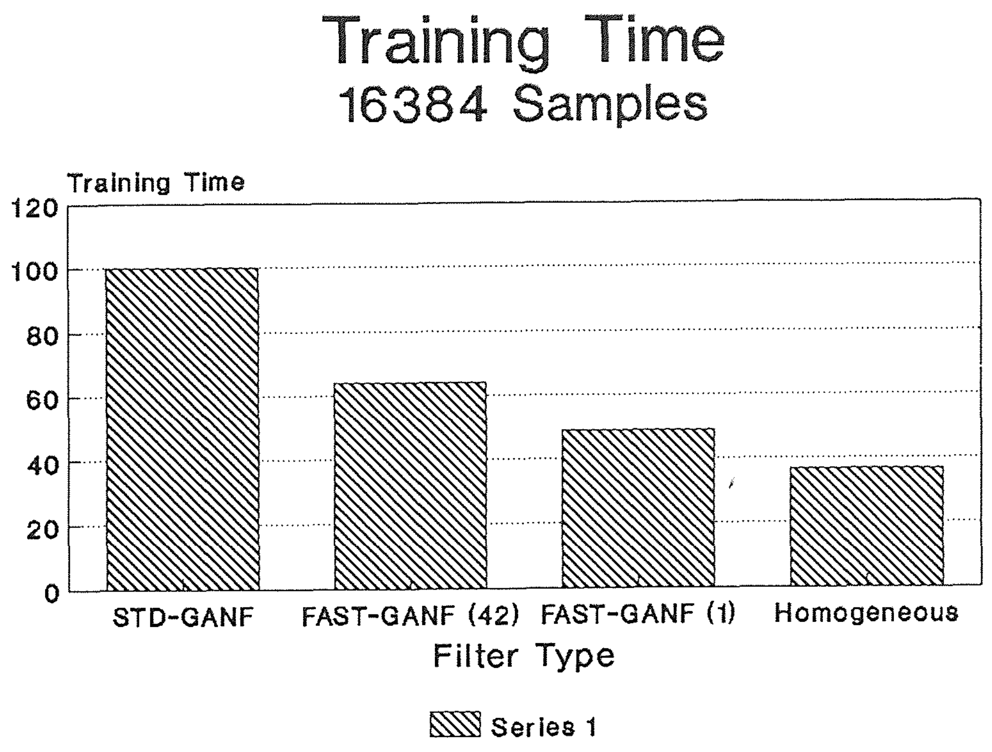


Figure 7.11 Training time vs. GANF type.

problems, very little data was obtained in these cases. The results that were obtained are most likely far from optimal. Because of these things, these results will not be discussed. The comparison results are included simply as reference points and to show the realities of non-optimized GANF performance. The simulations also brought to light some areas where future work is needed.

One of the major problems with the GANFs was the choice of the gradient search step parameter (α or μ). These values were set by trial and error by the user. This type of user intervention would prevent the filter's use in most practical circumstances. As a result, an adaptive learning rate would be desirable in practice [17]. In addition, this thesis considered only a simple learning rule – the LMS algorithm. Data was not cycled through either, as is recommended [17]. There are other learning rules which could have been investigated to increase network performance [17] [30]. Another area of needed improvement is that of speed. While the FAST method introduced in this thesis helps by a measurable amount, the filter is still slow. This method is perhaps one part of a combination of modifications which will be required to make the filter more practical.

APPENDIX A

In order to prove Proposition 4.1, we first need to prove some lemmas.

Lemma A.1 *We are given any two independent discriminant functions, $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ of identical form. These two functions can be combined with a new input $x_i \in \{0, 1\}$ to produce $g(\mathrm{x}, x_i)$:*

$$g(\mathrm{x}, x_i) = (1 - x_i)g_1(\mathrm{x}) + x_i g_2(\mathrm{x}), \tag{A.1}$$

where $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ have the same form, but (possibly) different weights. This function can achieve the following classification based on x_i:

$$g(\mathrm{x}, x_i) = \begin{cases} g_1(\mathrm{x}), & \text{if } x_i = 0, \\ g_2(\mathrm{x}), & \text{if } x_i = 1, \end{cases} \tag{A.2}$$

proof:

The proof can be done by inspection. Simply substitute $x_i = 0$ and $x_i = 1$ into the given equation. □

Note that in Lemma A.1, the functions $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ are independent. Since $g(\mathrm{x}, x_i)$ can equal $g_1(\mathrm{x})$ or $g_2(\mathrm{x})$ (depending on the state of x_i), we can implement independent discriminant functions for both input states. Next, we show that if $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ are polynomial discriminant functions, then eq. (A.1) in Lemma A.1 is equivalent to

$$g(\mathrm{x}, x_i) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_3 + w_5x_1x_3 + w_6x_2x_3 + w_7x_1x_2x_3 + \cdots \tag{A.3}$$

which also equals

$$g(\mathrm{x}, x_i) = w_0 + \sum_i w_i x_i + \sum_j \sum_k w_{jk} x_j x_k + \sum_l \sum_m \sum_n w_{lmn} x_l x_m x_n + \cdots. \tag{A.4}$$

Lemma A.2 *If $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ are polynomial discriminant functions, then the equation for $g(\mathrm{x}, x_i)$ in Lemma A.1 is also a polynomial discriminant function, which can be put in the form of eq.(A.3) or eq.(A.4).*

proof:

From eq. (A.1),

$$g(\mathrm{x}, x_i) = g_1(\mathrm{x}) - x_i g_1(\mathrm{x}) + x_i g_2(\mathrm{x}), \tag{A.5}$$

$$g(\mathrm{x}, x_i) = g_1(\mathrm{x}) + [g_2(\mathrm{x}) - g_1(\mathrm{x})]x_i. \tag{A.6}$$

Now, since we are dealing with polynomial discriminant functions,

$$g_1(\mathrm{x}) = v_0 + v_1 x_1 + v_2 x_2 + v_3 x_1 x_2 + \cdots \tag{A.7}$$

and

$$g_2(\mathrm{x}) = v_0^{'} + v_1^{'} x_1 + v_2^{'} x_2 + v_3^{'} x_1 x_2 + \cdots \tag{A.8}$$

From this,

$$g_2(\mathrm{x}) - g_1(\mathrm{x}) = (v_0^{'} - v_0) + (v_1^{'} - v_1)x_1 + (v_2^{'} - v_2)x_2 + (v_3^{'} - v_3)x_1 x_2 + \cdots \tag{A.9}$$

Substituting this into eq. (A.6) and simplifying, we get an equation of the form

$$g(\mathrm{x}, x_i) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_3 + w_5 x_1 x_3 + w_6 x_2 x_3 + w_7 x_1 x_2 x_3 + \cdots \tag{A.10}$$

This is equivalent to the polynomial discriminant function

$$g(\mathrm{x}, x_i) = w_0 + \sum_i w_i x_i + \sum_j \sum_k w_{jk} x_j x_k + \sum_l \sum_m \sum_n w_{lmn} x_l x_m x_n + \cdots, \tag{A.11}$$

since, for binary inputs,

$$x_{i_1} x_{i_2} \cdots x_{i_k} = x_i \quad for \quad i_1 = i_2 = \cdots = i_k = i. \tag{A.12}$$

In other words, eq. (A.12) makes it possible to reduce eq.(A.11) to eq. (A.10). □

It can be shown that the lemmas hold not only for $x_i \in \{0, 1\}$, but also for $x_i \in \{-1, 1\}$. We state this in the following two lemmas:

Lemma A.3 *We are given two independent discriminant functions, $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ of identical form. These two functions can be combined with a new input $x_i \in \{-1, 1\}$ to produce $g(\mathrm{x}, x_i)$:*

$$g(\mathrm{x}, x_i) = \frac{(1 - x_i)}{2} g_1(\mathrm{x}) + \frac{1 + x_i}{2} g_2(\mathrm{x}) \qquad \text{(A.13)}$$

where $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ have the same <u>*form*</u>*, but (possibly) different weights. We can adjust the weights to achieve*

$$g(\mathrm{x}, x_i) = \begin{cases} g_1(\mathrm{x}), & \text{if } x_i = -1, \\ g_2(\mathrm{x}), & \text{if } x_i = +1. \end{cases} \qquad \text{(A.14)}$$

proof:

This proof follows by inspection. Simply substitute $x_i = -1$ and $x_i = 1$ into the given equation. □

Lemma A.4 *If $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ are polynomial discriminant functions, then the equation for $g(\mathrm{x}, x_i)$ in Lemma A.3 is also a polynomial discriminant function, which can be put in the form of eq.(A.3) or eq.(A.4).*

proof:

From eq. (A.13),

$$g(\mathrm{x}, x_i) = \frac{1}{2} g_1(\mathrm{x}) + \frac{1}{2} g_2(\mathrm{x}) - \frac{x_i}{2} g_1(\mathrm{x}) + \frac{x_i}{2} g_2(\mathrm{x}), \qquad \text{(A.15)}$$

$$g(\mathrm{x}, x_i) = [\frac{1}{2} g_1(\mathrm{x}) + \frac{1}{2} g_2(\mathrm{x})] + [\frac{1}{2} g_2(\mathrm{x}) - \frac{1}{2} g_1(\mathrm{x})] x_i. \qquad \text{(A.16)}$$

Since any linear combination of $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ has the same form as $g_1(\mathrm{x})$ or $g_2(\mathrm{x})$, eq. (A.10) follows directly. This equation is equivalent to eq. (A.11) since

$$x_{i_1} x_{i_2} \cdots x_{i_k} = (-1)^{k+1} x_i \quad \textit{for} \quad i_1 = i_2 = \cdots = i_k = i. \qquad \text{(A.17)}$$

Therefore, the lemma is proved. □

Finally, we will prove a lemma which shows that a linear discriminant function in one variable can implement any Boolean function for this variable.

Lemma A.5 *The linear discriminant function,*

$$g_1(\mathbf{x}) = w_0 + w_1 x_1 \tag{A.18}$$

can implement any Boolean function in x_1:

$$y = \mathcal{B}(x_1), \tag{A.19}$$

where $\mathcal{B}(x_1) \in \{x_1, \overline{x}_1, 1, 0\}$.

proof:

If we want to implement the Boolean function, $y = x_1$, we generate the following two constraints on the weights w_0 and w_1:

$$x_1 = 1 \Longrightarrow g_1(\mathbf{x}) > 0 \Longrightarrow w_0 + w_1 > 0, \tag{A.20}$$

$$x_1 = 0 \Longrightarrow g_1(\mathbf{x}) < 0 \Longrightarrow w_0 < 0. \tag{A.21}$$

Therefore,

$$w_1 > -w_0, \tag{A.22}$$

and

$$w_0 < 0. \tag{A.23}$$

As a result, $w_1 > |w_0|$ and $w_0 < 0$ will implement the Boolean function $y = x_1$.

Now, if we want to implement $y = \overline{x}_1$, we generate different constraints on the weights:

$$x_1 = 1 \Longrightarrow g_1(\mathbf{x}) < 0 \Longrightarrow w_0 + w_1 < 0, \tag{A.24}$$

$$x_1 = 0 \Longrightarrow g_1(\mathbf{x}) > 0 \Longrightarrow w_0 > 0. \tag{A.25}$$

Therefore,

$$w_1 < -w_0, \tag{A.26}$$

and

$$w_0 > 0. \tag{A.27}$$

As a result, $w_1 < -|w_0|$ and $w_0 > 0$ will implement the Boolean function $y = \overline{x}_1$. If we want to implement $y = 1$,

$$x_1 = 1 \Longrightarrow g_1(\mathrm{x}) > 0 \Longrightarrow w_0 + w_1 > 0, \tag{A.28}$$

$$x_1 = 0 \Longrightarrow g_1(\mathrm{x}) > 0 \Longrightarrow w_0 > 0. \tag{A.29}$$

Therefore,

$$w_1 > -w_0, \tag{A.30}$$

and

$$w_0 > 0. \tag{A.31}$$

As a result, $w_1 > -|w_0|$ and $w_0 > 0$ will implement the Boolean function $y = 1$. Now, the only other possibility for eq. (A.19) with one variable is y=0. For this case,

$$x_1 = 1 \Longrightarrow g_1(\mathrm{x}) < 0 \Longrightarrow w_0 + w_1 < 0, \tag{A.32}$$

$$x_1 = 0 \Longrightarrow g_1(\mathrm{x}) < 0 \Longrightarrow w_0 < 0. \tag{A.33}$$

Therefore,

$$w_1 < -w_0, \tag{A.34}$$

and

$$w_0 < 0. \tag{A.35}$$

As a result, $w_1 < |w_0|$ and $w_0 < 0$ will implement the Boolean function $y = 0$. □

We are now ready to prove Proposition A.1:

Proposition A.1 *A single neuron with polynomial pre-processing can implement any Boolean function.*

proof:

We will prove this by induction. First of all, note that a polynomial discriminant function in one variable is identical to a linear discriminant function in one variable. Therefore, from Lemma A.5, it is clear that a polynomial discriminant function in one variable can implement all Boolean functions for this variable.

Next, let us suppose that we have two functions, $g_1(\mathrm{x})$ and $g_2(\mathrm{x})$ which can independently implement any Boolean function for a vector x of length $(i-1)$. If we add an input, it is clear from Lemma A.1 that the new function $g(\mathrm{x}, x_i)$ can implement any Boolean function in i variables. This results because we now are able to achieve independent Boolean functions in $(i-1)$ variables for each state of the added input. Also, because of Lemma A.2, the function $g(\mathrm{x}, x_i)$ can be represented in terms of a polynomial discriminant function. As a result, if a polynomial discriminant function in $(i-1)$ variables can implement any Boolean function, then a polynomial discriminant function in i variables will also implement any Boolean function. Therefore, the proposition is proved. $\square$.

This proposition can also be shown to hold for $x_i \in \{-1, 1\}$ instead of $x_1 \in \{0, 1\}$. This is done by using Lemmas A.3 and A.4 in place of Lemmas A.1 and A.2, respectively.

APPENDIX B

This Appendix contains the proof of Proposition 4.1. To start off with, we will prove some lemmas which will be needed.

Lemma B.1 *A single neuron with linear discriminant function,*

$$g(\mathrm{x}_N) = w_0 + w_1 x_1 + \cdots + w_N x_N \tag{B.1}$$

can implement the following Boolean function for N inputs:

$$y = \mathcal{B}_1 + \mathcal{B}_2 \cdots \mathcal{B}_N. \tag{B.2}$$

where the plus signs denote Boolean OR operations and $\mathcal{B}_i$ denote Boolean terms of the form

$$\mathcal{B}_i = \begin{cases} x_i \\ \overline{x}_i \\ 0 \end{cases} \tag{B.3}$$

In other words, a single neuron can implement a Boolean function, y, consisting of the sum (Boolean OR) of N terms chosen from the set $\{x_1, x_2, \ldots, x_N, \overline{x}_1, \overline{x}_2, \ldots, \overline{x}_N, 0\}$.

proof:

The lemma will first be proved with $x_i \in \{0, 1\}$. The single neuron with $N-1$ inputs can be described by a linear discriminant function of the form,

$$g(\mathrm{x}_{N-1}) = w_0 + \mathbf{w}_{N-1}^T \mathrm{x}_{N-1}, \tag{B.4}$$

where $\mathbf{w}_{N-1}^T = [w_1 \quad w_2 \quad \cdots \quad w_{N-1}]$ and $\mathrm{x}_{N-1}^T = [x_1 \;\; x_2 \;\; \cdots \;\; x_{N-1}]$, or

$$g(\mathrm{x}_{N-1}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_{N-1} x_{N-1}. \tag{B.5}$$

Suppose this function implements some Boolean function for the input vector $\mathrm{x}_{N-1}(n)$. We will denote this Boolean function by $\mathcal{F}(\mathrm{x}_{N-1})$. We can form a new function with N inputs:

$$g(\mathrm{x}_{N-1}, x_N) = g(\mathrm{x}_{N-1}) + w_N x_N + w_{NEW}. \tag{B.6}$$

Note that this also is a linear discriminant function since the weight w_{NEW} can be combined with w_0. We now wish to show that regardless of $g(\mathrm{x}_{N-1})$, we can have $g(\mathrm{x}_{N-1}, x_N)$ implement the Boolean function

$$y = \mathcal{F}(\mathrm{x}_{N-1}) + \mathcal{B}_N. \tag{B.7}$$

where the plus sign denotes the Boolean OR operation, $\mathcal{F}(\cdot)$ denotes a Boolean function in $N-1$ variables and $\mathcal{B}_N$ represents one choice out of the set $\{x_N, \overline{x}_N, 0\}$. Now let $g(\mathrm{x}_{N-1})$ take on any value between $-M$ and $+P$:

$$-M \leq g(\mathrm{x}_{N-1}) \leq +P. \tag{B.8}$$

To include the term x_N in the Boolean function of eq. (B.7), we let $w_{NEW} = 0$ and generate the following equations from eq. (B.6):

$$x_N = 1 \Longrightarrow g(\mathrm{x}_{N-1}, x_N) > 0, \tag{B.9}$$

$$x_N = 0 \Longrightarrow g(\mathrm{x}_{N-1}, x_N) = g(\mathrm{x}_{N-1}). \tag{B.10}$$

Since the added term $w_N x_N = 0$ for $x_N = 0$, eq. (B.10) is automatically satisfied. Eq. (B.9) is left to be satisfied, and can be re-stated as follows:

$$g(\mathrm{x}_{N-1}) + w_N > 0. \tag{B.11}$$

Because of eq. (B.8), where $M > 0$ and $P > 0$, the limiting condition on the weight w_N is

$$w_N > M. \tag{B.12}$$

With continuous weights, this can always be achieved. To include the term $\overline{x}_N$ in eq. (B.7) instead of x_N, we generate the following constraints on eq. (B.6):

$$x_2 = 0 \Longrightarrow g(\mathrm{x}_{N-1}, x_N) > 0, \tag{B.13}$$

$$x_2 = 1 \Longrightarrow g(\mathrm{x}_{N-1}, x_N) = g(\mathrm{x}_N). \tag{B.14}$$

with

$$g(\mathrm{x}_{N-1}, x_N) = g(\mathrm{x}_{N-1}) + w_N x_N + w_{NEW}. \tag{B.15}$$

Here, we will not set $w_{NEW} = 0$. For $x_N = 0$ we have

$$g(\mathrm{x}_{N-1}) + w_{NEW} > 0, \tag{B.16}$$

and, for $x_N = 1$,

$$g(\mathrm{x}_{N-1}) + w_N + w_{NEW} = g(\mathrm{x}_{N-1}). \tag{B.17}$$

These conditions can be satisfied by

$$w_{NEW} > M, \tag{B.18}$$

$$w_N = -w_{NEW}. \tag{B.19}$$

Finally, note that setting $w_i = 0$ allows us to ignore a particular input variable x_i. This is the the same as choosing 0 for the corresponding term in eq. (B.2). We can see from Lemma A.5 that eq. (B.1) can implement the Boolean function specified by eq. (B.2) for one variable. Also, we have just shown that we can extend it from $N-1$ inputs to N inputs. Therefore, by induction, Lemma B.1 is shown to be true. $\square$

Given the previous proof for $x_i \in \{0,1\}$, we can definitely find the w_N and w_{NEW} to solve our problem for $x_i \in \{0,1\}$. So, to prove the lemma for $x_i \in \{-1,1\}$, we represent each new term added as $w_A x_N + w_B$. (Before, we considered each added term to be $w_N x_N + w_{NEW}$.) From before,

$$(w_i x_i + w_{NEW})|_{x_i=0} = w_{NEW}, \tag{B.20}$$

and

$$(w_N x_N + w_{NEW})|_{x_N=1} = w_N + w_{NEW}. \tag{B.21}$$

We now show that we can find weights w_A and w_B to solve it with $x_N \in \{-1,1\}$.

$$(w_A x_N + w_B)|_{x_N=-1} = -w_A + w_B, \tag{B.22}$$

$$(w_A x_N + w_B)|_{x_N=1} = w_A + w_B. \tag{B.23}$$

To make the cases $x_i \in \{0,1\}$ and $x_i \in \{-1,1\}$ equivalent, we generate

$$-w_A + w_B = w_{NEW}, \tag{B.24}$$

and

$$w_A + w_B = w_N + w_{NEW}. \tag{B.25}$$

Or

$$w_B = \frac{w_N}{2} + w_{NEW}, \tag{B.26}$$

and

$$w_A = \frac{w_N}{2}. \tag{B.27}$$

In other words, given w_N and w_{NEW} for $x_i \in \{0,1\}$, we can find equivalent weights w_A and w_B to implement the same solution for $x_i \in \{-1,1\}$. Again, since the weights are continuous variables, this can always be achieved. $\square$

Now let us prove an additional lemma:

Lemma B.2 *We are given a single neuron represented by the linear discriminant function $g_1(\mathrm{x})$, with $(N-1)$ inputs. That is, x has $(N-1)$ components. New terms $w_A x_N + w_B$ can be added to produce the following new linear discriminant function: $g(\mathrm{x}, x_N) = g_1(\mathrm{x}) + w_A x_N + w_B$. This linear discriminant function can be made to implement the two cases,*

<u>case 1</u>:

$g(\mathrm{x}, x_N) = g_1(\mathrm{x})$ *if* $x_N = 0$

$g(\mathrm{x}, x_N) < 0$ *if* $x_N = 1$

or <u>case 2</u>:

$g(\mathrm{x}, x_N) < 0$ *if* $x_N = 0$

$g(\mathrm{x}, x_N) = g_1(\mathrm{x})$ *if* $x_N = 1$

where case 1 or case 2 is determined by selection of the weights w_A and w_B.

proof:

Let $g_1(\mathrm{x})$ take on any value between $-M$ and $+P$, where $M > 0$ and $P > 0$. That is,

$$-M \leq g_1(\mathrm{x}) \leq +P. \tag{B.28}$$

For <u>case 1</u> we first generate the condition,

$$(g_1(\mathrm{x}) + w_A x_N + w_B)|_{x_N=0} = g_1(\mathrm{x}), \tag{B.29}$$

$$g_1(\mathrm{x}) + w_B = g_1(\mathrm{x}), \tag{B.30}$$

$$w_B = 0. \tag{B.31}$$

The next condition is

$$(g_1(\mathrm{x}) + w_A x_N + w_B)|_{x_N=1} < 0, \tag{B.32}$$

From this we can see that

$$w_A < -P < 0 < M. \tag{B.33}$$

Therefore, the two weights must satisfy

$$w_A < -P, \tag{B.34}$$

and

$$w_B = 0. \tag{B.35}$$

For <u>case 2</u> we first generate the condition

$$(g_1(\mathrm{x}) + w_A x_N + w_B)|_{x_N=0} < 0. \tag{B.36}$$

This leads to

$$w_B < -P. \tag{B.37}$$

The next condition is

$$(g_1(\mathrm{x}) + w_A x_N + w_B)|_{x_N=1} = g_1(\mathrm{x}). \tag{B.38}$$

From this we can see that

$$w_A = -w_B. \tag{B.39}$$

Therefore, the two weights must satisfy

$$w_A = -w_B, \tag{B.40}$$

and

$$w_B < -P. \tag{B.41}$$

These conditions are always achievable since our weights are continuous. Also, this lemma and can be extended for $\mathbf{x} \in \{-1, 1\}$ in a manner similar to that used for Lemma B.1. We can now prove the supposition. □

Proposition B.1 *A 2 layer net with N inputs, 2^{N-1} neurons in the first layer and 1 neuron in the second layer can implement any Boolean function.*

proof:

We start by showing that two neurons in the first layer can each implement any product term in any Boolean function with two inputs. We represent the linear discriminant function of the first neuron in layer 1 as

$$g_1(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2. \tag{B.42}$$

The linear discriminant function of the second neuron in layer 1 is

$$g_2(\mathbf{x}) = w_0' + w_1' x_1 + w_2' x_2. \tag{B.43}$$

The possible Boolean functions for two inputs are shown in Table B.1. Note that a second layer neuron can perform an OR operation on the outputs of the two first layer neurons (from Lemma B.1). Then each first layer neuron must be able to implement a term of the form:

$$y = \mathcal{B}_1 \cdot \mathcal{B}_2, \tag{B.44}$$

Table B.1 Possible Boolean functions with 2 inputs.

Boolean Function $y = \cdots$
0
1
$\overline{x}_1\overline{x}_2$
$\overline{x}_1 x_2$
$x_1\overline{x}_2$
$x_1 x_2$
$\overline{x}_1\overline{x}_2 + \overline{x}_1 x_2$
$\overline{x}_1\overline{x}_2 + x_1\overline{x}_2$
$\overline{x}_1\overline{x}_2 + x_1 x_2$
$\overline{x}_1 x_2 + x_1\overline{x}_2$
$\overline{x}_1 x_2 + x_1 x_2$
$x_1\overline{x}_2 + x_1 x_2$
$x_1 + x_2$
$\overline{x}_1 + x_2$
$x_1 + \overline{x}_2$
$x_1 + x_2$

where the raised dot represents the Boolean AND operation and

$$\mathcal{B}_i \in \{x_i, \overline{x}_i, 1, 0\}. \tag{B.45}$$

This is equivalent to each neuron implementing

$$y = \mathcal{B}_1 + \mathcal{B}_2, \tag{B.46}$$

where the plus sign represents the Boolean OR operation and

$$\mathcal{B}_i \in \{x_i, \overline{x}_i, 1, 0\}. \tag{B.47}$$

It is easy to see (by inspection) how eq. (B.42) or eq. (B.43) can implement this. In general, x_i is included in the OR if $w_i > w_0$. Complements are achieved through multiplication by -1. A proof of this would follow the the form of that used in Lemma B.1.

From Lemma B.1, it is clear that a simple neuron in the second layer can implement the OR operation among its inputs, and also ignore selected inputs. We can now see that the proposition holds for 2 inputs. To extend it to N inputs, we note that for each input, x_N, added, the number of neurons in the first layer will double, and all of the neurons will have the terms $w_N x_N + w'_N$ added to their discriminant functions. Next, if we implement a Boolean function in $(N-1)$ variables for x_N in one state, and generate an independent Boolean function in $(N-1)$ variables for x_N in its other state, we can implement any Boolean function in N variables.

This doubling in size of the first layer is equivalent to doubling the size of a Karnaugh map when a new input is added. (2^{N-2}) neurons implement a Boolean function for $(N-1)$ inputs AND $x_N = 0$, while the other (2^{N-2}) neurons provide a Boolean function for $(N-1)$ inputs AND $x_N = 1$. Since Lemma B.2 showed that this is possible, the proposition is proved. Although not shown here, it can be extended to apply for $x_i \in \{-1, 1\}$. $\square$

APPENDIX C

Table C.1 Comparison filters (3 × 3) processing IMAGE 1.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0.4$	12.62	297.13	17.83
	$\alpha = 0.5$ (median)	12.38	316.22	17.56
Modified Trimmed Mean	$q = 1$	12.38	316.22	17.56
	$q = 20$	13.71	314.89	17.58
Double-Window MTM	$N = 0, L = 1, q = 200$	13.66	310.71	17.64
	$N = 0, L = 1, q = 187$	13.66	309.58	17.66
k-nearest Neighbor v.1	$k = 9$ (mean)	13.71	314.89	17.58
k-nearest Neighbor v.2	$k = 8$	14.74	367.54	16.91
Mod. k-nearest Neighbor v.1	$k = 1$	12.38	316.22	17.56
Mod. k-nearest Neighbor v.2	$k = 1$	13.86	410.44	16.43
	$k = 8$	14.12	330.64	17.37
Wilcoxon v.1		13.22	308.74	17.67
Wilcoxon v.2		13.76	322.29	17.48
Adaptive Mean	$C = 187$	13.66	309.56	17.66
Adaptive Median	$C = 187$	12.35	313.55	17.60
Conventional Median		12.38	316.22	17.56
Separate Median		13.55	372.08	16.86
Max/Median		25.26	1144.73	11.98
Wiener		14.05	330.54	17.37

Table C.2 Comparison filters (3×3) processing IMAGE 2.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0$ (mean)	27.10	1168.98	11.88
	$\alpha = 0.5$ (median)	26.25	1510.61	10.77
Modified Trimmed Mean	$q = 1$	26.25	1510.61	10.77
	$q = 250$	27.38	1193.24	11.80
Double Window MTM	$N = 0, L = 1, q = 250$	29.36	1412.82	11.06
k-nearest Neighbor v.1	$k = 9$ (mean)	27.10	1168.98	11.88
k-nearest Neighbor v.2	$k = 8$	28.35	1287.18	11.47
Mod. k-nearest Neighbor v.1	$k = 1$	26.25	1510.61	10.77
	$k = 9$ (mean)	27.10	1168.98	11.88
Mod. k-nearest Neighbor v.2	$k = 8$	27.92	1219.20	11.70
Wilcoxon v.1		27.23	1267.33	11.53
Wilcoxon v.2		28.42	1312.79	11.38
Adaptive Mean	$C = 250$	29.30	1406.80	11.08
Adaptive Median		28.62	1778.71	10.06
Conventional Median		26.25	1510.61	10.77
Separate Median		29.66	1880.67	9.82
Max/Median		56.01	5454.73	5.20
Wiener		27.23	1174.15	11.87

Table C.3 Comparison filters (3×3) processing IMAGE 3.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0$ (mean)	15.00	369.70	16.88
Modified Trimmed Mean	$q = 200$	15.00	369.70	16.89
Double Window MTM	$N = 0, L = 1, q = 212$	14.94	365.31	16.94
k-nearest Neighbor v.1	$k = 9$ (mean)	15.00	369.70	16.88
k-nearest Neighbor v.2	$k = 8$	15.02	371.62	16.86
Mod. k-nearest Neighbor v.1	$k = 9$ (mean)	15.00	369.70	16.88
Mod. k-nearest Neighbor v.2	$k = 8$	15.02	371.62	16.86
Wilcoxon v.1		15.86	408.64	16.45
Wilcoxon v.2		15.66	399.31	16.55
Adaptive Mean	$C = 212$	14.95	365.45	16.94
Adaptive Median		18.14	526.62	15.35
Conventional Median		18.18	530.05	15.32
Separate Median		19.63	616.94	14.66
Max/Median		31.88	1527.39	10.72
Wiener		15.31	384.91	16.71

Table C.4 Comparison filters (3×3) processing IMAGE 4.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0$ (mean)	30.39	1436.77	10.99
Modified Trimmed Mean	$q = 250$	31.02	1501.19	10.80
Double Window MTM	$N = 0, L = 1, q = 250$	34.43	1943.82	9.68
k-nearest Neighbor v.1	$k = 9$ (mean)	30.39	1436.77	10.99
k-nearest Neighbor v.2	$k = 8$	31.77	1575.44	10.59
Mod. k-nearest Neighbor v.1	$k = 9$ (mean)	30.39	1436.77	10.99
Mod. k-nearest Neighbor v.2	$k = 8$	30.34	1419.14	11.04
Wilcoxon v.1		33.45	1749.24	10.13
Wilcoxon v.2		32.58	1656.54	10.37
Adaptive Mean	$C = 250$	34.33	1931.46	9.70
Adaptive Median	$C = 250$	47.52	3491.91	7.13
Conventional Median		43.86	2923.69	7.90
Separate Median		47.24	3376.82	7.28
Max/Median		71.89	7353.65	3.90
Wiener		30.84	1463.06	10.91

Table C.5 Comparison filters (5×5) processing IMAGE 1.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0.45$	10.26	240.08	18.76
	$\alpha = 0.5$ (median)	10.16	243.73	18.69
Modified Trimmed Mean	$q = 1$	10.16	243.73	18.69
	$q = 10$	10.13	249.87	18.59
Double Window MTM	$N = 0, L = 2, q = 150$	11.70	252.11	18.55
	$N = 1, L = 2, q = 87$	11.16	240.87	18.74
	$N = 1, L = 2, q = 100$	11.16	239.15	18.78
k-nearest Neighbor v.1	$k = 23$	11.91	261.89	18.38
	$k = 24$	11.80	268.58	18.27
k-nearest Neighbor v.2	$k = 23$	11.91	279.08	18.11
Mod. k-nearest Neighbor v.1	$k = 1$	10.16	243.73	18.69
Mod. k-nearest Neighbor v.2	$k = 7$	10.21	257.06	18.46
	$k = 9$	10.25	255.53	18.49
Wilcoxon v.1		11.20	261.68	18.38
Wilcoxon v.2		11.56	271.21	18.23
Adaptive Mean	$C = 137$	11.76	252.02	18.55
	$C = 150$	11.70	252.36	18.54
Adaptive Median	$C = 137$	9.81	216.15	19.22
	$C = 150$	9.80	217.74	19.18
Conventional Median		10.16	243.73	18.64
Separate Median		10.98	273.65	18.19
Max/Median		22.15	870.84	13.16
Wiener		14.15	395.33	16.59

Table C.6 Comparison filters (5 × 5) processing IMAGE 2.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0.45$	16.56	564.88	15.04
	$\alpha = 0.5$ (median)	15.74	572.51	14.98
Modified Trimmed Mean	$q = 1$	15.74	572.51	14.98
	$q = 20$	15.39	596.31	14.81
Double Window MTM	$N = 0, L = 2, q = 250$	23.70	914.40	12.95
	$N = 1, L = 2, q = 187$	21.12	760.41	13.75
k-nearest Neighbor v.1	$k = 24$	21.38	741.27	13.86
k-nearest Neighbor v.2	$k = 23$	21.31	737.68	13.88
Mod. k-nearest Neighbor v.1	$k = 1$	15.74	572.51	14.98
	$k = 6$	15.53	612.89	14.69
Mod. k-nearest Neighbor v.2	$k = 1$	16.13	624.86	14.60
	$k = 5$	15.64	628.86	14.58
Wilcoxon v.1		19.82	697.52	14.13
Wilcoxon v.2		21.03	754.71	13.78
Adaptive Mean	$C = 250$	23.66	910.86	12.97
Adaptive Median	$C = 250$	17.52	707.03	14.07
Conventional Median		15.74	572.51	14.98
Separate Median		18.35	762.50	13.74
Max/Median		48.95	4171.81	6.36
Wiener		22.52	801.09	13.53

Table C.7 Comparison filters (5×5) processing IMAGE 3.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0.1$	12.85	310.84	17.64
Modified Trimmed Mean	$q = 125$	12.71	301.30	17.77
Double Window MTM	$N = 0, L = 2, q = 162$	12.62	285.95	18.00
	$N = 0, L = 2, q = 175$	12.60	287.38	17.98
	$N = 1, L = 2, q = 125$	12.44	282.80	18.05
	$N = 1, L = 2, q = 137$	12.44	284.28	18.02
k-nearest Neighbor v.1	$k = 24$	12.84	301.47	17.77
k-nearest Neighbor v.2	$k = 23$	12.91	310.37	17.64
Mod. k-nearest Neighbor v.1	$k = 24$	13.05	316.02	17.57
	$k = 25$ (mean)	13.03	320.50	17.50
Mod. k-nearest Neighbor v.2	$k = 23$	13.05	316.49	17.56
Wilcoxon v.1		12.88	307.51	17.68
Wilcoxon v.2		12.88	308.58	17.67
Adaptive Mean	$C = 162$	12.61	285.78	18.00
	$C = 175$	12.60	287.66	17.97
Adaptive Median	$C = 162$	13.84	331.30	17.36
Conventional Median		14.11	353.14	17.08
Separate Median		15.34	408.24	16.45
Max/Median		28.43	1205.71	11.75
Wiener		14.56	396.77	16.58

Table C.8 Comparison filters (5 × 5) processing IMAGE 4.

Filter Name	Filter Parameters	MAE	MSE	SNR [dB]
α-Trimmed Mean	$\alpha = 0.15$	23.72	896.34	13.04
Modified Trimmed Mean	$q = 200$	24.64	977.38	12.66
	$q = 250$	24.92	957.88	12.75
Double Window MTM	$N = 0, L = 2, q = 250$	27.80	1267.70	11.53
	$N = 1, L = 2, q = 250$	24.66	949.63	12.79
k-nearest Neighbor v.1	$k = 24$	24.45	931.28	12.87
k-nearest Neighbor v.2	$k = 23$	24.34	925.51	12.90
Mod. k-nearest Neighbor v.1	$k = 24$	24.01	906.01	12.99
Mod. k-nearest Neighbor v.2	$k = 23$	24.08	908.04	12.98
Wilcoxon v.1		24.80	970.68	12.69
Wilcoxon v.2		25.08	988.68	12.61
Adaptive Mean	$C = 250$	27.72	1258.81	11.56
Adaptive Median	$C = 250$	34.60	1911.34	9.75
Conventional Median		29.84	1400.24	11.10
Separate Median		33.35	1750.97	10.13
Max/Median		65.74	6121.00	4.70
Wiener		25.62	995.20	12.58

Table C.9 Completely non-homogeneous GANF processing IMAGE 1.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.0001$	13.08	315.81	17.57
Quadric DF	$3 \times 3, \alpha = 0.0001$	13.17	302.24	17.76
	$3 \times 3, \alpha = 0.00009$	13.18	301.42	17.77
Quadric DF	$5 \times 5, \alpha = 0.001$	11.59	267.15	18.30
Polynomial DF	$3 \times 3, \alpha = 0.0003$	13.33	311.86	17.62
Minimal 2-Layer	$3 \times 3, \mu = 0.5$	13.50	341.58	17.23
	$3 \times 3, \mu = 0.7$	13.46	341.46	17.23
	$3 \times 3, \mu = 0.8$	13.47	341.73	17.23
	$3 \times 3, \mu = 0.9$	13.46	341.56	17.23
Large 2-Layer	$3 \times 3, \mu = 0.1$	12.17	277.68	18.13
Radial Basis Function	$3 \times 3, \alpha = 0.01$	15.35	415.22	16.38

Table C.10 Completely non-homogeneous GANF processing IMAGE 2.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.0001$	24.84	1069.99	12.27
	$3 \times 3, \alpha = 0.01$	24.72	1133.60	12.02
Quadric DF	$3 \times 3, \alpha = 0.00009$	24.62	1033.56	12.42
	$3 \times 3, \alpha = 0.00008$	24.67	1031.31	12.43
Quadric DF	$5 \times 5, \alpha = 0.001$	18.87	685.05	14.21
Polynomial DF	$3 \times 3, \alpha = 0.0003$	24.81	1080.94	12.22
Minimal 2-Layer	$3 \times 3, \mu = 0.5$	22.58	983.55	12.64
	$3 \times 3, \mu = 0.9$	22.34	985.31	12.63
Large 2-Layer	$3 \times 3, \mu = 0.01$	24.17	1127.75	12.04
Radial Basis Function	$3 \times 3, \alpha = 0.01$	33.80	1886.14	9.81

Table C.11 Completely non-homogeneous GANF processing IMAGE 3.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.00001$	17.33	480.79	15.74
Quadric DF	$3 \times 3, \alpha = 0.00008$	15.95	413.48	16.40
Quadric DF	$5 \times 5, \alpha = 0.0001$	13.45	327.72	17.41
Polynomial DF	$3 \times 3, \alpha = 0.0001$	15.55	398.96	16.55
Minimal 2-Layer	$3 \times 3, \mu = 0.7$	17.28	475.15	15.80
Large 2-Layer	$3 \times 3, \mu = 0.01$	17.60	489.34	15.67
Radial Basis Function	$3 \times 3, \alpha = 0.01$	20.03	590.35	14.85

Table C.12 Completely non-homogeneous GANF processing IMAGE 4.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.00001$	31.82	1499.10	10.80
Quadric DF	$3 \times 3, \alpha = 0.00001$	31.77	1487.48	10.84
Quadric DF	$5 \times 5, \alpha = 0.0001$	24.84	966.05	12.71
Polynomial DF	$3 \times 3, \alpha = 0.00005$	30.53	1428.47	11.01
Minimal 2-Layer	$3 \times 3, \mu = 0.5$	33.35	1646.89	10.40
	$3 \times 3, \mu = 0.7$	33.19	1647.35	10.40
	$3 \times 3, \mu = 0.8$	33.18	1655.03	10.38
Large 2-Layer	$3 \times 3, \mu = 0.01$	34.77	1831.71	9.93
Radial Basis Function	$3 \times 3, \alpha = 0.000001$	43.90	2723,22	8,21
	$3 \times 3, \alpha = 0.00001$	44.65	2696.47	8.26

Table C.13 Homogeneous GANF processing IMAGE 1.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.000001$	12.58	325.83	17.43
Quadric DF	$3 \times 3, \alpha = 0.000008$	12.86	342.28	17.22
Quadric DF	$5 \times 5, \alpha = 0.00001$	10.60	249.53	18.59
Polynomial DF	$3 \times 3, \alpha = 0.0001$	15.44	476.62	15.78
Minimal 2-Layer	$3 \times 3, \mu = 0.2$	25.65	1154.02	11.94
Large 2-Layer	3×3	—	—	—
Radial Basis Function	3×3	—	—	—

Table C.14 Homogeneous GANF processing IMAGE 2.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.00001$	26.25	1510.61	10.77
	$3 \times 3, \alpha = 0.00005$	26.25	1510.61	10.77
	$3 \times 3, \alpha = 0.000001$	26.25	1510.61	10.77
Quadric DF	$3 \times 3, \alpha = 0.000001$	26.55	1545.55	10.67
Quadric DF	$5 \times 5, \alpha = 0.00001$	17.92	731.73	13.92
Polynomial DF	$3 \times 3, \alpha = 0.0001$	33.53	2263.95	9.01
Minimal 2-Layer	$3 \times 3, \mu = 0.2$	36.58	2713.12	8.23
Large 2-Layer	3×3	—	—	—
Radial Basis Function	3×3	—	—	—

Table C.15 Homogeneous GANF processing IMAGE 3.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.000005$	18.20	530.68	15.32
Quadric DF	$3 \times 3, \alpha = 0.000008$	18.46	544.68	15.20
Quadric DF	$5 \times 5, \alpha = 0.00001$	15.19	388.93	16.66
Polynomial DF	$3 \times 3, \alpha = 0.0001$	21.34	732.00	13.92
Minimal 2-Layer	$3 \times 3, \mu = 0.2$	20.57	674.25	14.28
Large 2-Layer	3×3	—	—	—
Radial Basis Function	3×3	—	—	—

Table C.16 Homogeneous GANF processing IMAGE 4.

Neuron Type	Parameters	MAE	MSE	SNR [dB]
Single neuron	$3 \times 3, \alpha = 0.00001$	43.86	2923.69	7.90
Single neuron	$3 \times 3, \alpha = 0.00005$	43.86	2923.69	7.90
Single neuron	$3 \times 3, \alpha = 0.000001$	43.86	2923.69	7.90
Quadric DF	$3 \times 3, \alpha = 0.000001$	44.14	2958.42	7.85
Quadric DF	$5 \times 5, \alpha = 0.00001$	33.76	1777.93	10.06
Polynomial DF	$3 \times 3, \alpha = 0.0001$	52.14	4133.04	6.40
Minimal 2-Layer	$3 \times 3, \mu = 0.2$	55.15	4550.50	5.98
Large 2-Layer	3×3	—	—	—
Radial Basis Function	3×3	—	—	—

Table C.17 FAST-GANF (3×3) image processing results.

Image	FAST-GANF Type	Parameters	MAE	MSE	SNR [dB]
1	Method 1	$\beta = 0.98, \alpha = 0.0005$	14.29	385.09	16.71
1	Method 1	$\beta = 0, \alpha = 0.0005$	14.49	396.97	16.58
1	Method 2	$\beta = 0.98, \alpha = 0.0001$	14.76	385.73	16.70
1	Method 2	$\beta = 0, \alpha = 0.0001$	14.38	401.88	16.52
2	Method 1	$\beta = 0.98, \alpha = 0.0001$	25.52	1116.51	12.08
2	Method 1	$\beta = 0, \alpha = 0.0001$	26.07	1168.99	11.88
2	Method 2	$\beta = 0.98, \alpha = 0.0001$	25.30	1099.67	12.15
2	Method 2	$\beta = 0, \alpha = 0.0001$	27.09	1427.82	11.02
3	Method 1	$\beta = 0.98, \alpha = 0.0001$	19.11	565.99	15.04
3	Method 1	$\beta = 0, \alpha = 0.0001$	19.54	590.84	14.85
3	Method 2	$\beta = 0.98, \alpha = 0.0001$	19.15	566.81	15.03
3	Method 2	$\beta = 0, \alpha = 0.0001$	19.89	614.26	14.68
4	Method 1	$\beta = 0.98, \alpha = 0.0001$	34.93	1740.18	10.16
4	Method 1	$\beta = 0, \alpha = 0.0001$	36.21	1867.67	9.85
4	Method 2	$\beta = 0.98, \alpha = 0.0001$	35.15	1753.05	10.12
4	Method 2	$\beta = 0, \alpha = 0.0001$	38.46	2330.53	8.89

REFERENCES

1. N. Ansari and Z. Z. Zhang, "Generalised Adaptive Neural Filters," *Electronic Letters*, vol. 29, no. 4, pp. 342–343, 18 Feb. 1993.

2. A. K. Jain, *Fundamentals of Digital Image Processing*, Englewood Cliffs: Prentice Hall, 1989.

3. P. D. Wendt, E. J. Coyle and N. C. Gallagher, "Stack Filters," *IEEE Trans. ASSP*, vol. ASSP-34, pp. 898–911, Aug. 1986.

4. Y. T. Kim and J. H. Lin, "Fast Training Algorithms for Stack Filters," *submitted to IEEE Trans. on Signal Processing.*

5. S. Haykin, *Adaptive Filter Theory 2nd edition*, Englewood Cliffs: Prentice Hall, 1991.

6. E. J. Coyle and J. H. Lin, "Stack Filters and the Mean Absolute Error Criterion," *IEEE Trans. ASSP*, vol. 36, no. 8, Aug. 1988.

7. B. Zheng, H. Zhou, and Y. Neuvo, "FIR Stack Hybrid Filters," *Optical Engineering*, vol. 30, pp. 965–975, July 1991.

8. N. Ansari, Y. Huang and J. H. Lin, "Adaptive Stack Filtering by LMS and Perceptron Learning," *Icon. ASSP*, 1992.

9. S. Muroga, *Logic Design and Switching Theory*, New York: John Wiley & Sons, 1979.

10. Z. Z. Zhang, N. Ansari and J.H. Lin, "On Generalized Adaptive Neural Filters," *Proc. IJCNN'92*, June 7–11, 1992, Baltimore, MD, pp IV.277–282.

11. T. M. Cover, "Geometrical and Statistical Properties of Systems of Linear Equalities with Applications in Pattern Recognition," *IEEE Trans. on Electronic Computers*, June 1965, pp. 326–334.

12. B. Widrow and M. A. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," *Proc. IEEE*, vol. 78, no. 9, Sept. 1990.

13. M. M. Mano, *Computer Engineering: Hardware Design*, Englewood Cliffs: Prentice Hall, 1988.

14. V. N. Vapnik and A. Ja. Chervonenkis, "Uniform Convergence of Frequencies of Occurrence of Events to their Probabilities," *Dokl. Akad. Nauk SSSR*, vol. 181, no. 4, 1968.

15. V. N. Vapnik and A. Ya. Chervonenkis, "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities," *Theory of Probability and its Applications*, vol. 16, no. 2, 1971.

16. E. B. Baum and D. Haussler, "What Size Net Gives Valid Generalization," *Neural Computation*, vol. 1, pp. 151–160, 1989.

17. D. R. Hush and B. G. Horne, "Progress in Supervised Neural Networks," *IEEE Signal Processing Magazine*, pp. 8–39, Jan. 1993.

18. N. J. Nilsson, *The Mathematical Foundations of Learning Machines*, San Mateo: Morgan Kaufmann, 1990.

19. B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Englewood Cliffs: Prentice Hall, 1985.

20. R. Hecht-Nielson, "Kolmogorov's Mapping Neural Network Existence Theorem", *Proc. IEEE Internat. Conf. of Neural Networks*, June 21–24, 1987, San Diego, CA, pp. III.11–13.

21. R. Hecht-Nielson, "Theory of the Backpropagation Neural Network", *Internat. Joint Conf. on Neural Networks*, June 18–22, 1989, Washington, DC, vol. 1, pp. 593–605.

22. A. N. Kolmogorov, "On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition," *Dokl. Akad. Nauk USSR*, 114, 953–956, 1957.

23. F. Girosi and T. Poggio, "Representation Properties of Networks: Kolmogorov's Theorem is Irrelevant," *Neural Computation*, vol. 1, pp. 465–469, 1989.

24. T. Poggio and F. Girosi, "Networks for Approximation and Learning," *Proc. IEEE*, vol. 78, no. 9, Sept. 1990.

25. F. Girosi and T. Poggio, "Networks and the Best Approximation Property," *Biological Cybernetics*, vol. 63, pp 169–176, 1990.

26. J. H. Lin, Y. T. Kim and G. Soemarwoto "Nonlinear Filtering Techniques Based on a Threshold Decomposition Architecture", *Proc. of the 26th Annual Conference of Information, Science and System*, Princeton. NJ, Mar. 18–20, 1992.

27. J. H. Lin, T. M. Selke and E. J. Coyle, "Adaptive Stack Filtering Under the Mean Absolute Error Criterion," *IEEE Trans. ASSP*, vol. 38, no. 6, pp. 938–954, June 1990.

28. B. V. Gnedenko, *The Theory of Probability 2nd edition*, New York: Chelsea, 1963.

29. Y. S. Fong, C. A. Pomalaza-Reaz, X. H. Wang, "Comparison study of nonlinear filters in image processing applications," *Optical Engineering*, vol. 29, no. 7, July 1989.

30. A. Van Ooyen and B. Nienhuis, "Improving the Convergence of the Back-Propagation Algorithm," *Neural Networks*, vol. 5, pp. 465–471, 1992.