ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

□□□□□ □□□□□□□□□□□□ □□□□□□□□□ & □□□□□□□□□ □π□□□□□□□□□

□□π□□□□□□□□ □□□□□□□

"Μέθοδοι εύρεσης πολύ μικρής περιστροφής ολοκληρωμένων κυκλωμάτων μέσω ανάλυσης της ψηφιακής εικόνας του τυπωμένου κυκλώματος."

□□□□□□□ □□□□□□□

□□□□□□□□□□ □π□□□□π□

1. □□□. □□□□□□□ □□□□□□□□ (□π□□□□π□□)
2. Eπ. □□□. □□□□□□□ □□□□□□□□□
3. □π. □□□. □□□□□□□□□□□□□

To my parents Nikolaos and Athanasia Iliadis.

Imagination is more important than knowledge.

Albert Einstein.

Acknowledgnments

I would like to thank all those, who have helped me in the preparation of this case study. First I would like to thank professor Mr. Michalis Zervakis whose experience and ideas helped me refine my thinking. I would also like to thank professors Mr. Giorgos Rovithakis and Mr. Dionisios Pnevmatikatos for participating in my jury and for their comments and corrections. Finally, I would like to thank my roommate and a very close friend Nikos Ntarmos for his advice, particularly in the technical part of this application, that made the implementation of this case study possible.

Contents

1 Preliminaries.

2 Extracting characteristics from the PCB image.

2.0 Introduction.

2.1 The PCB image.

2.2 Extracting characteristics

2.2.1 Modeling of Data

2.2.2 General linear least squares

2.2.3 Solution by the use of normal equations

2.2.4 Method to detect the direction of the rotation

3 Artificial Neural Networks

4.0 Introduction

4.1 The Neuron

4.2 Basic Neural Network Architectures

4.3 Benefits of Neural Network

4 Higher order Neural Networks (HONN)

5.0 Introduction

5.1 Approximation capabilities

5.2 Learning traits

5.3 Least squares learning method

5 Genetic Algorithms

2.3 Introduction

2.4 Description

2.5 Basic Genetic Algorithm operations

2.6 Genetic algorithm traits

2.7 Genetic algorithm versus traditional optimization methods

6 Selection of HONN structure for function approximation using Genetic Algorithms.

6.0 Introduction

6.1 Methods for improving the structure of a neural network

6.2 Description of the algorithm

6.3 Structure of a Genetic Algorithm.

7 Alternative way to measure the rotational angle of a PCB component

7.0 Introduction

7.1 Measure the rotational angle using the Radon transform

7.2 Results.

Appendix A

About the PCB images

Appendix B

Manual for the Genetic Algorithm Software

Appendix C

Manual for the main application software

Chapter 1. Prelimineries

1.1 General Description of the HIPER project for post-Placement Inspection.

The manufacture of electronics requires sophisticated machinery to place components (SMDs) on printed circuit boards. Central to the assembly process is the components' placement function. Speed, flexibility and accuracy requirements and the end-user application largely dictate the type of machines that will be used. Because of the continuing drive for higher accuracy and speed, the placement machines feature machine-vision equipment for board and component alignment. However, to be able to verify the position of the placed component, a post-placement inspection is needed. In order to be able to immediately get the information about the placement accuracy, this post placement inspection should be executed right after placement of each component.

With the existing vision systems, the post placement inspection is carried out on the board, after that the board is fully SMD mounted. A drawback of this method is the fact that the inspection data is not available before the PCB is fully mounted. Another problem is the poor dynamic range of the used CCD cameras together with the fixed video frame rate.

1.2 HIPER Application Perspectives.

Placement process control plays an important role within the zero-defects policy of today's electronics manufacturing industry. Parameters that are currently checked / controlled with vision sensors are:

- Component's presence check (before and/or after placement)
- Placement force
- Component pick correction (=x, y pipette shift before pick)
- Fiducial alignment measurement (=x, y φ positioning of PCB)
- Component alignment (CA) measurement (=x, y, φ positioning of component)
- Component lead inspection together with CA

- Component placement position: as a result of nominal required position plus FA / CA corrections.

In fact the only parameter that is not checked is the positional accuracy of the component (SMD) after placement. Factors that could influence this positional accuracy are:

- Shift between SMD and nozzle due to acceleration / nozzle tip vibrations
- Forces acting on SMD due to PCB warpage
- Other eventual effects: thermal expansion of the PCB was not clamped sufficiently.

This after-placement positional accuracy of SMDs in not measured up to now. Reasons are:

- It is not possible to make reliable measurements with “common” CCD cameras. The difficulty here is that a distinction should be made between a shiny copper footprint and a shiny SMD lead. The dynamic reach of CCD cameras is insufficient for this.
- Random pixel access is not possible with conventional CCD cameras. Therefore it takes a lot of time to collect all pixel data and process this data.
- It is complex to integrate vision sensors and illumination around the placement head. This integration is essential to minimize time loss for post placement inspection.

The HIPER project’s aim is the implementation of an integrated CMOS sensors / image processing device for an optics / illumination system that will be situated around the placement head of the SMD placement machine. In the HIPER project, an image acquisition module will be installed around the placement head of the ACM (Advanced Component Mounter) machine. This image acquisition module must be capable of taking top-view images of the respective mounted SMD in its belonging solder land. Analysis of the images should verify post placement criteria.

1.4 Case study and Chapter organization

This case study is a part of the above HIPER application. It analyzes the images taken from the HIPER camera in order to detect and to measure extremely small rotational angles of the SMD components on a printed circuit board. This analysis will check the positional accuracy of the components (SMD) on a mounted PCB in a rotational sense. The key features of this method must be high-speed, flexibility and accuracy of the rotational measurement in order to be reliable and to be used in real-time systems.

Although we develop two completely different methods to detect the rotational angle of an SMD component, we focus our case study in the first one which utilizes neural networks. The Chapter organization is as follows:

- **Chapter 2.**

This chapter analyzes the preprocessing part of our application which utilizes neural networks. It shows how we extract, from a given image, useful characteristics which we later provide as data to our neural network. We also make here all necessary modifications to our data in order to make our application independent of external factors such as illumination and SMD geometry.

- **Chapter 3**

Here we make a brief introduction to artificial neural networks (ANN). We examine how an artificial neural network works and how ANNs can store information from the training data. We also examine the basic architectures of neural networks, as well as their advantages and disadvantages.

- **Chapter 4**

In this chapter we focus on Genetic Algorithms (GAs). These algorithms comprise a special kind of search algorithms, which we use to find the optimal structure of our neural network. We examine the basic operations of genetic algorithms and how they can be used as an optimization technique for ANNs. Finally we mention all the tradeoffs between the GAs and other optimization methods.

- **Chapter 5.**

In chapter 5 we analyze in detail the kind of neural networks we use in our application, the higher order neural networks (HONN). Here we place all the mathematical proofs that show how HONNs are capable of approximating functions as well as all the learning traits of higher order neural networks.

- **Chapter 6**

In earlier chapters we represented the basic concepts of neural networks and HONNs as well as those of the genetic algorithms. Here we describe the way GAs can be used as an optimization method to our higher order neural network. We give a description of the algorithm that we use for that purpose and we examine in detail every single step of this algorithm. Finally, we mention all the results, such as neural network error neuron status etc., taken from the implementation of the above algorithm.

- **Chapter 7**

In this chapter we examine a completely different approach to the problem. The technique used here is based on the Radon transform and it is a traditional method to detect object rotations. In chapter 7 we discuss all the pros and the cons of the Radon transform and mention the results taken from the implementation of this method.

Besides these chapters we also have three appendices which concern the implementation techniques of this application. They might help the reader to understand how each program works.

- **Appendix A**

The appendix A discusses the method by which the training samples for our neural network are taken. We also refer to the difficulties we encounter during this process and the reasons for these problems.

- **Appendix B**

The appendix B is the manual of the implementation of the Genetic algorithm program. It is a general description of the implementation of the program and it shows what files are needed in order for it to work

- **Appendix C**

Here we examine the implementation of the main program. It shows the block diagram of the program and the main steps the program follows. We also provide a time profiler which shows how the total execution time is distributed in each function. We finally have a list of error codes that the program returns in case of an internal or external error.

The last page shows the results of the program for some unknown (to the neural network) images for each side of the component.

Chapter 2. Extracting characteristics from the image.

2.0 Introduction

As we already know image processing is often time-consuming and requires in most cases large amounts of memory to be accomplished. The above reasons make image processing applications very difficult to be used in real time and embedded systems. In this chapter we will discuss in detail the way we produce, from a given image, useful characteristics for our application that will later be used as input to our system, thus making the whole application suitable for real time systems. The characteristics mentioned above must be enough, in order to cover every aspect of the problem without any loss of information for the specific application. Also the whole method for the extraction of these characteristics must be abstract, so that the same method can be applied to similar type problems without any significant changes.

2.1 PCB image.

In order to gather any useful information for our purpose - to measure the rotational angle of the component if such rotation exists - it is necessary to observe more closely the image itself. Such an image is given below (image 2.1). It shows a PCB circuit board. It constitutes of a component (Topline TQFP120T15.7) and the pad.

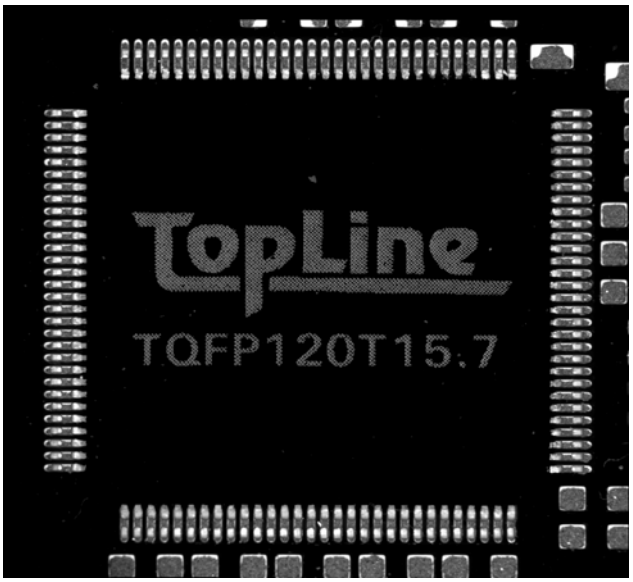


Image 2-1

This specific application requires an identification error goal of ± 0.01 degrees in an interval from -0.6 to 0.6 degrees. Due to this very small identification error goal as well as to the poor image quality, all information regarding the rotation of the component can only be deduced by the edges of the component, namely the pins. Taking as a fact that every side of the component is rotated with the same angle respectively and that the camera taking the image is in a totally horizontal position, we can assume that analyzing only one side of the component can drive us to the same results as if we analyzed all sides of the component. This assumption is necessary in order to speed up the whole application and to reduce the total amount of memory needed.

As a result of the above assumptions we can create a new image (only for representational reasons; in real conditions we just allocate only the amount of memory needed, without having to create a new image), having only one side of the component. Without any loss of generality we will continue our application with the upper-side of the component (image 2.2).

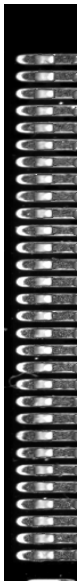


Image 2-2
(Upper side of the component in horizontal position)

Note that in the images 2-1 and 2-2, we not only show the pins of the component but also the solder paste between the component and the pad.

2.2 Extracting characteristics from the PCB image.

One important feature we could use to detect the rotational angle of the component is the first derivative of the horizontally projected values of each pixel of the up side.

More analytically, if we project every single pixel of the component in a vertical line by summing its values and then take the first derivative of that projection, we will have the following graphs:

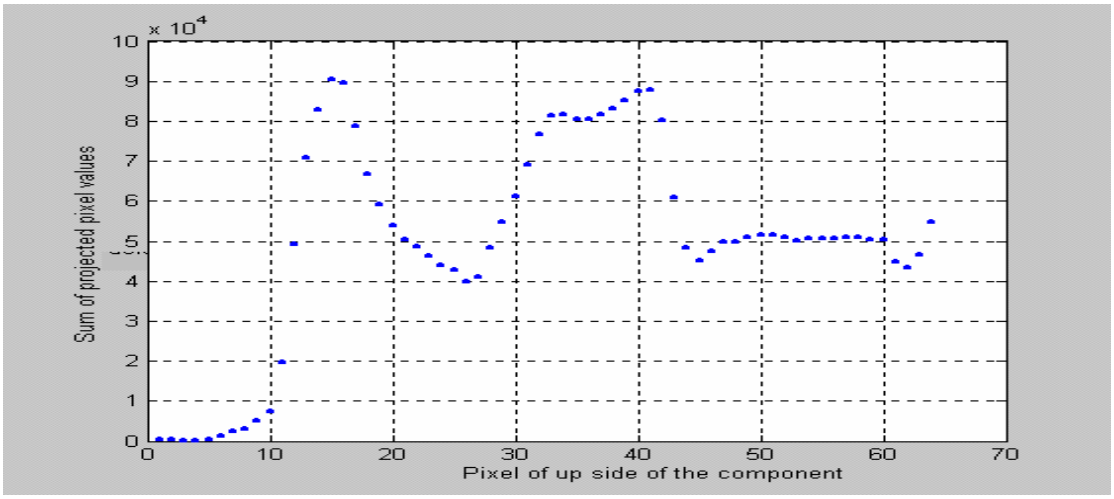


Figure 2-1

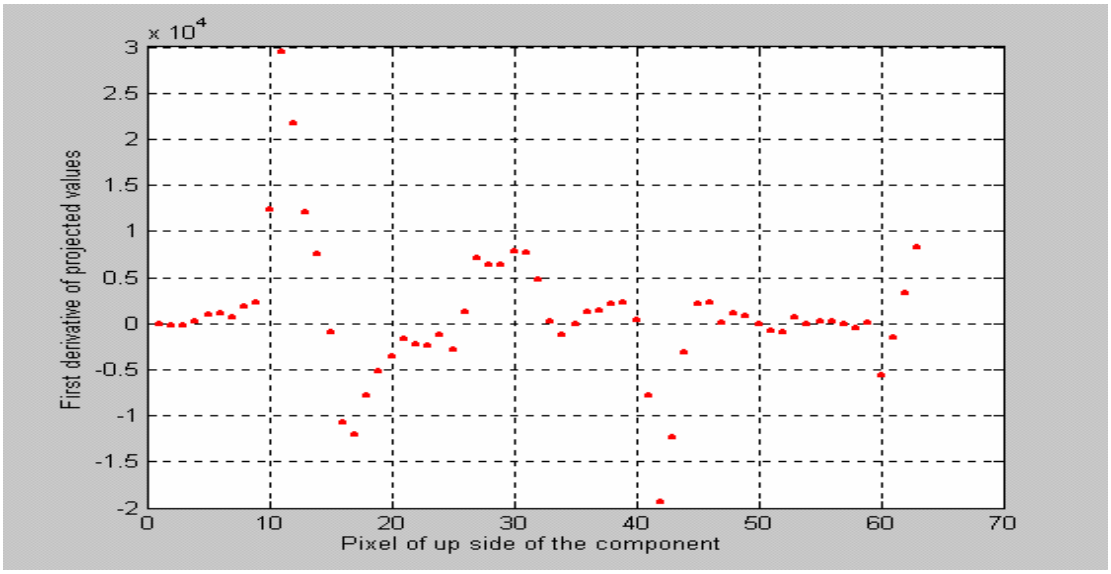


Figure 2-2

Intuitively we could expect that if the component had 0 degrees of rotation it would have the largest first derivative in the projected values presented above. Respectively if the rotational angle is large, the first derivative will be relatively small, due to the smooth rising of the edge of the component.

Note that the first large derivative in figure 2-2 belongs to the pad and not to the component so the relative position and value won't change with any rotational angle of the component. Our concern is for the second large derivative which corresponds to the edge of the component.

Truly, if we compare the first derivative of the component for various rotation angles we can see that our hypothesis is indeed true. The graph presented below represents the first derivative according to rotational angle, from -0.6 to 0.6.

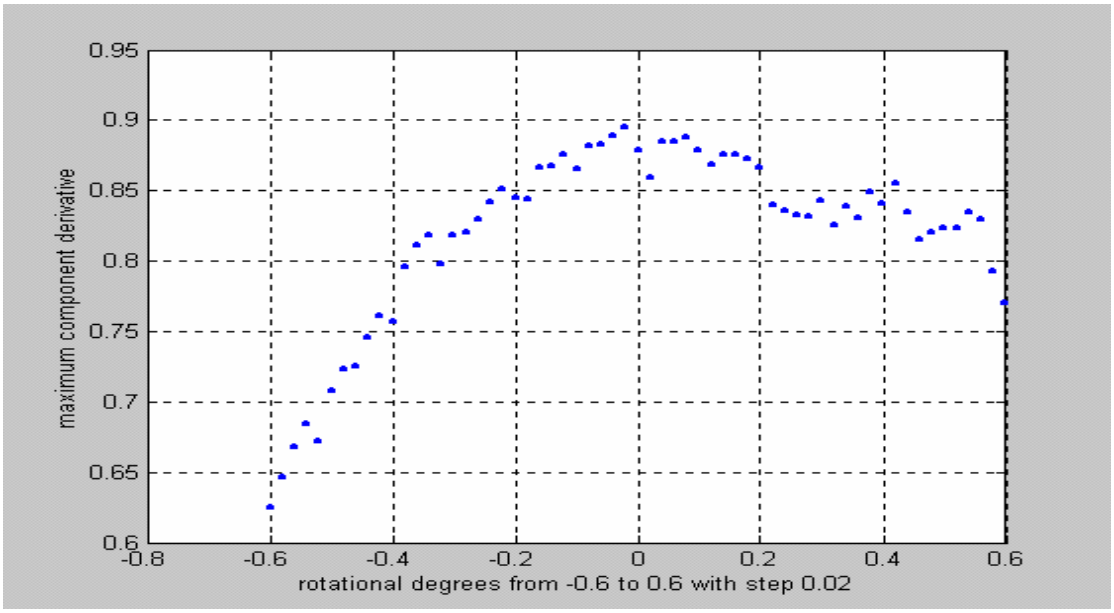


Figure 2-3

As we can see from the figure 2-3 the first derivative that corresponds to the component (that is the second maximum derivative of the figure 2-2) tends to reduce as the rotational angle gets larger and it gets larger as the rotational angle gets smaller. However, due to noise in the original image this behavior is corrupted significally. To reduce the noise effect without changing the nature of the data gathered above we will try to model our data (the projected values and not the derivatives) with a certain polynomial in order to get the desired response.

2.2.1 Modeling of Data

Our initial approach to this case is to try to model our data with a rather large order polynomial so that to minimize the noise and to achieve as much good fit we could without loosing any useful information since we require as much detail as possible. As a maximum likelihood estimator we considered the least Square function.

The method discussed below is used to achieve the fitting procedure:

2.2.2 General linear least squares

Let considered a polynomial of degree M-1:

$$\text{y(x)}= a_1 + a_2 x + a_3 x^2 + \ldots + a_M x^{M-1} \qquad (2.1)$$

The general form of this kind of model is

$$\text{y(x)}= \sum_{k=1}^{M} a_K X_K(x) \qquad (2.2)$$

Where $X_1(x),\ldots,X_M(x)$ are fixed arbitrary functions of x, called basis functions. Note that the functions Xm(x) can be wildly non linear functions of x. In this discussion "linear" refers only to the model's dependence on its parameters a_k.

For these linear models we define a merit function

$$X^2 = \sum_{i=1}^{N} \left[\frac{Y_{\,i} - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_t} \right]^2 \qquad (2.3)$$

Where σ_t is the measurement error (standard deviation) of the ith point presumed to be known. If the measurement error is not known they may all be set to the constant value σ = 1.

Out task is to find the best parameters those that minimize X^2. They are several techniques available for finding this minimum. To introduce our method we need some notation.

Let A be a matrix whose N x M components are constructed from the M basis functions evaluated at the N abscissas xi, and from the N measurement errors σi, by the prescription

$$A_{ij} = \frac{Xj(xi)}{\sigma i} \qquad (2.4)$$

The matrix A is called the design matrix of the fitting problem. Notice that in general A has more rows than columns, N>=M, since there must be more data points (pixel points in our application) that model parameters to be solved for. The design matrix is shown schematically in Figure 2.4.We also define a vector b of length N by

$$b_i = \frac{y_i}{\sigma_i} \qquad (2.5)$$

and denote the M vector whose components are the parameters to be fitted $a_1,...,a_M$ by a.

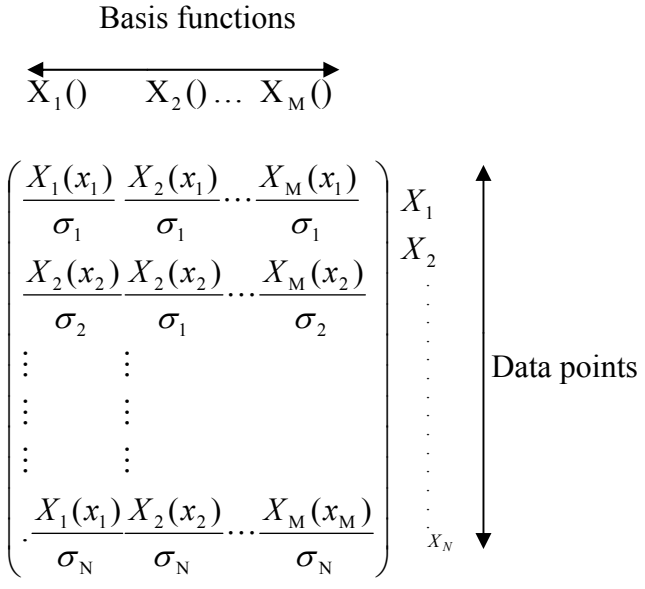


Figure 2.4 Design matrix for the least squares fit of a linear combination of M basis function to N data points. The matrix elements involve the basis functions evaluated at the values of the independent variable at which measurements are made, and the standard deviations of the measured dependent variable. The measured values of the dependent variable do not enter the design matrix.

2.2.3 Solution by Use of the Normal Equations

The minimum of equation (2.3) occurs where the derivative of X^2 with respect to all M parameters a_k vanishes. This condition yields the M equations

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) \quad \text{k=1,…,M} \tag{2.6}$$

Interchanging the order of summations, we can write the above equation as the matrix equation

$$\sum_{j=1}^{M} a_{kj} a_j = \beta_k \tag{2.7}$$

where

$$a_{kj} = \sum_{i=1}^{N} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \quad \text{or equivalently} \qquad [\alpha] = A^T . A \tag{2.8}$$

an MxM, matrix and

$$\beta_\kappa = \sum_{\iota=1}^{N} \frac{y_j X_k(x_i)}{\sigma_\iota^2} \qquad \text{or equivalently} \qquad [\beta] = A^T \cdot b \tag{2.9}$$

The equations (2.6) and (2.7) are called the normal equations of the least-squares problem. They can be solved for the vector of the vector of parameters a by the standard methods notably LU decomposition and backsubstitution, Choleksy decomposition, or Gauss-Jordan elimination.

Acting accordingly to the above theoretical method we manage to achieve a very good fit on the data in figure 2-1. Although we could try to fit the entire function, we choose to fit only a specific part of the function in which we are truly interested.

This region of interest is placed between the first maximum derivative (i.e. that of the pad) and to the second maximum derivative (i.e. that of the component). The order of the polynomial used for the fitting was 10 and the results are plotted in the graph below where the red cross (+) represents the fitting:

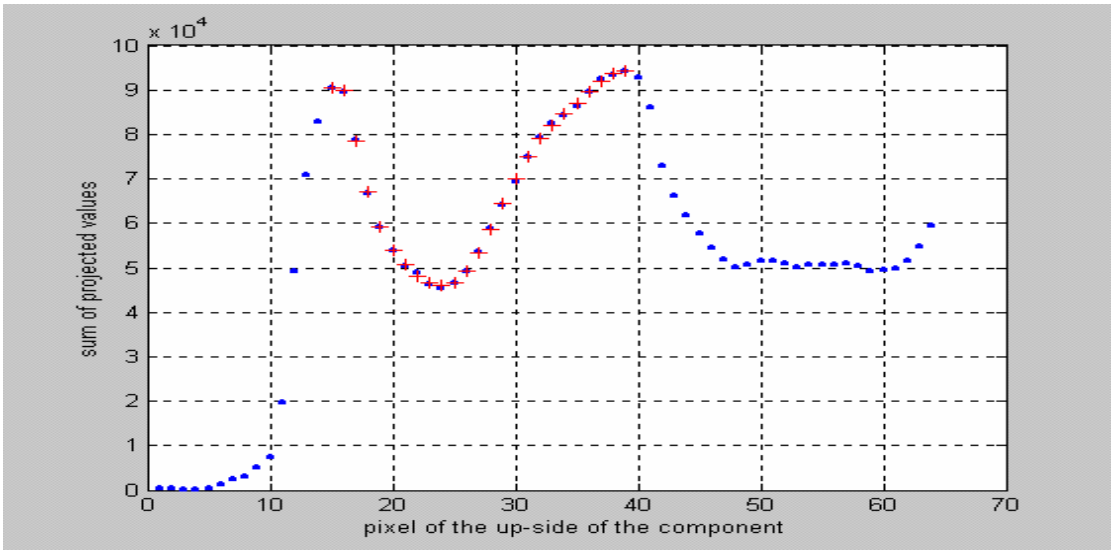


figure 2-4

If we now follow the exactly same procedure to obtain the maximum derivatives for various rotational angles, we get the following graph.

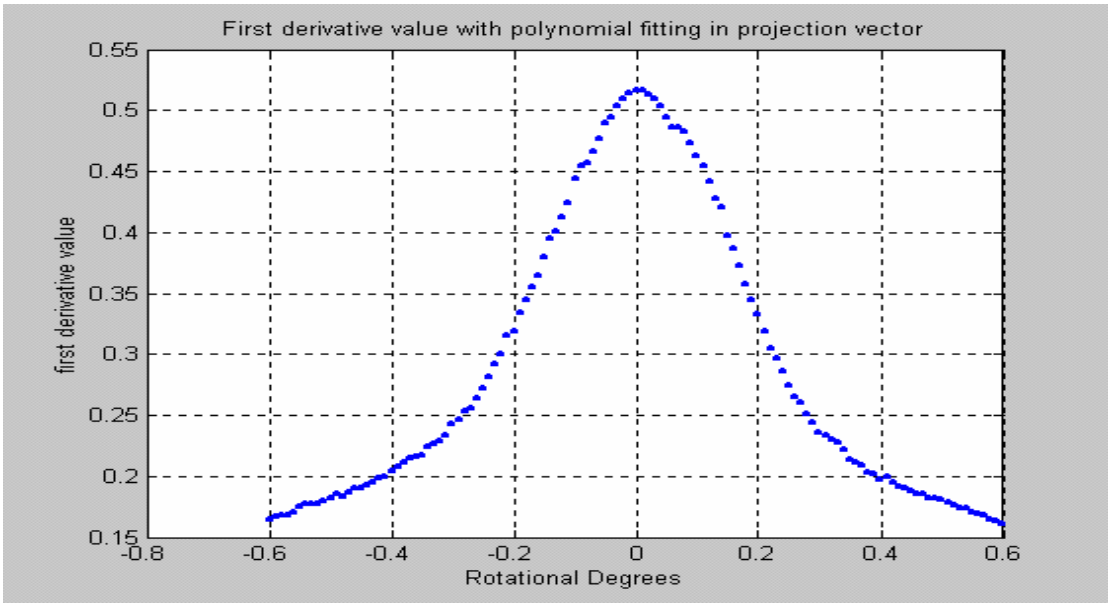


Figure 2-5

As we can see from the diagram, the first derivative has the information that we need, since its values are distinct enough for various angles, even for as small ones as those required by the application.

2.2.4 Method to detect the direction of the rotation

Due to the nature of the curve in figure 2.5, each possible value of the first derivative corresponds to two different angles, one positive and one negative. For this reason we first need to determine the direction of the rotational angle. As we probably noticed, the projection vector doesn't give us any information about the direction (positive or negative) of the rotation of the component. This information must be acquired from the original image and, more specifically, from the point where the maximum derivative of the component appears. If we consider again the image 2.2 we will notice that when the rotation is clockwise, more bright pixels due to the component are placed in the lower side of the vertical of the specific point. When the rotation is counterclockwise bright pixels are placed in the upper side of the same vertical. To represent this idea more clearly, we give the following simple example.

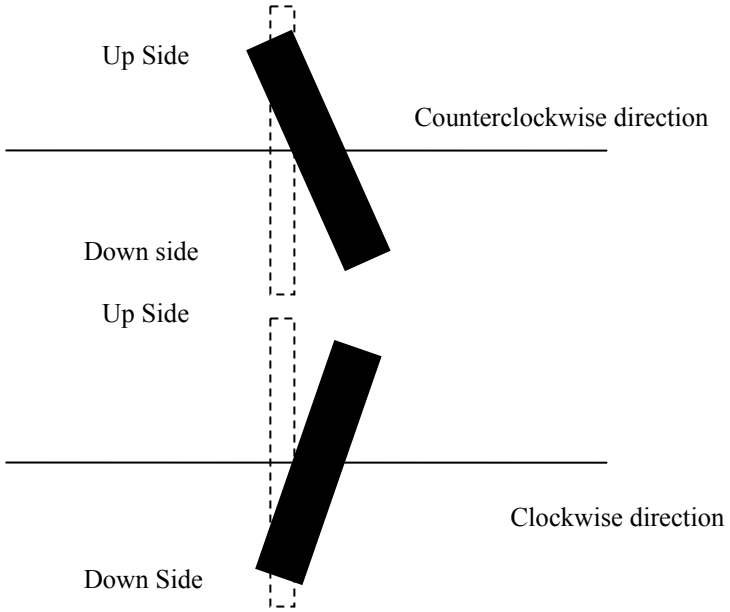


Figure 2.6.

The above figure represents the same idea with a simpler object (rectangular) instead of the pins of a PCB component. We assume that the dashed rectangular is at the point where we have the maximum derivative of the component. As we can see, when the rotation of the object is clockwise we have more black pixels in the lower side of the dashed rectangular. Instead, when the rotation is counterclockwise more, black pixels are placed in the upper side. According to this idea, if we compare the two sides of the dashed rectangular we can determine the direction of the rotation. Namely if N is the number of the black pixels (in our example), we have:

N(Upper side) > N(Lower side) counterclockwise rotation

N(Upper side)< N(Lower side) clockwise rotation

N(Upper side)=N(Lower side) zero degrees rotation

The performance of this idea in our application was very good giving errors only to extremely small angles (less than 0.01 degrees). The direction of the rotation is given to our system as an input and has only two possible values: 1 when the direction is clockwise and -1 when the direction is counterclockwise.

Note that the rotation of the PCB component is done around the center of the component.

2.2.5 Making first derivative illumination independent

The value of the first derivative, as figure 2.5 indicates, has the disadvantage that it depends on illumination factors. A brighter image with the same component rotation would produce a different maximum derivative. To make the value of the first derivative, and thus the whole application, illumination independent we need to make some sort of normalization in the projected values. Let's consider again the projected values of the upper window given below.

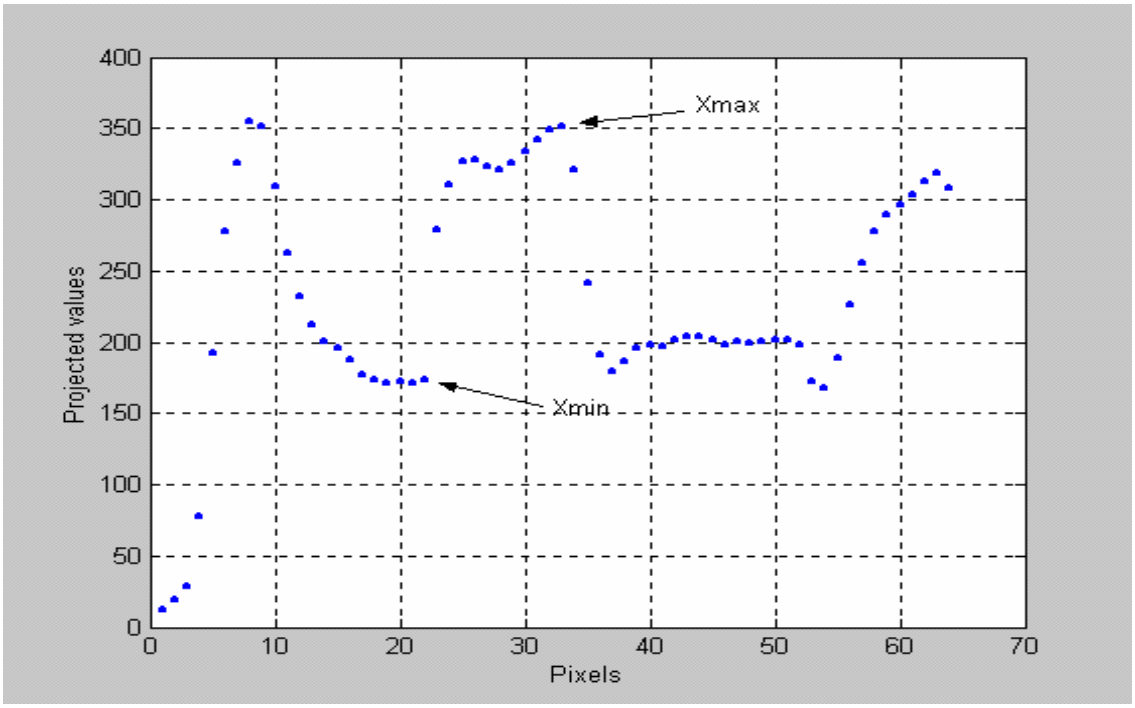


Figure 2.7

As we already said we are concerned in the region where the component is placed, namely in the region [Xmin Xmax]. A brighter image with the same component rotation would produce different but proportional Xmin Xmax values. To make the first derivative independent from the Xmin and Xmax values we use a linear normalization in both regions so that their values will be in the region from 0 to 1. For an arbitrary point X in the region [Xmin Xmax] the normalization equation is:

$$X_{norm} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

The above relation guaranties that any point between [Xmin Xmax] will have a value between [0 1]. This method removes the illumination factor from the first derivative, as all the values of the points in the projected vector (the specific region) are now scaled from 0 to 1, as long as the illumination changes are distributed uniformly on every side of the component (left, right, down, up).

Note. This method assumes that the values of Xmin and Xmax remain the same under the same illumination and for different rotational angles. Although this assumption isn't true, for very small rotational angles, the values Xmin and Xmax change insignificantly, producing only a very small measurement error, of order of

3% or less. However, in greater angles this normalization method produces much greater errors.

Using the above method we manage to make our application, illumination independent having only a rather small measurement error. This process is extremely important because now we can detect the rotational angle of components in a PCB image regardless of the illumination factor (an external factor), making the application abstract enough to be used in practical purposes.

2.2.5 Making first derivative independent of the geometry of the component.

Intuitively we could say that a bigger PCB component would affect our maximum derivative results as taken before. Truly, this assumption is indeed true and can be proved mathematically as follows:

Let's assume that the values in our projected vector in the region of interest (that is where the component is placed), after the normalization, have the form of the figure 2.8.

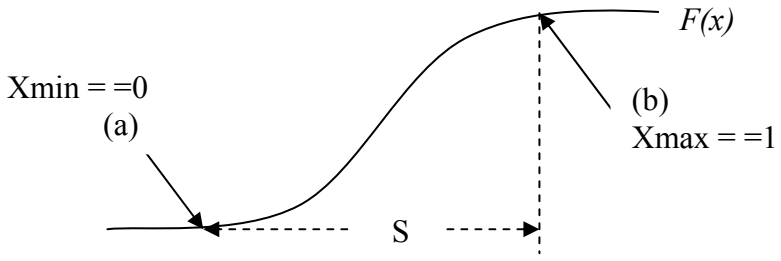


Figure 2.8

As we can see from the above figure the projected values (*f(x)*) are normalized and they vary from Xmin==0 to Xmax==1. Lets say that S is the length from point a to point b.

The first derivative of the *f(x)* would be:

$$g(x) = \frac{df(x)}{dx} \qquad (1)$$

If we take the integral of the above relation we would have:

$$\int_a^b g(x)dx = 1 \Rightarrow A(b-a) = 1 \qquad (2)$$

The equation 2 says that when the derivative increases, due to the component rotation, the distance between Xmax and Xmin decreases. Therefore, for various rotational angles we will have the following graph:

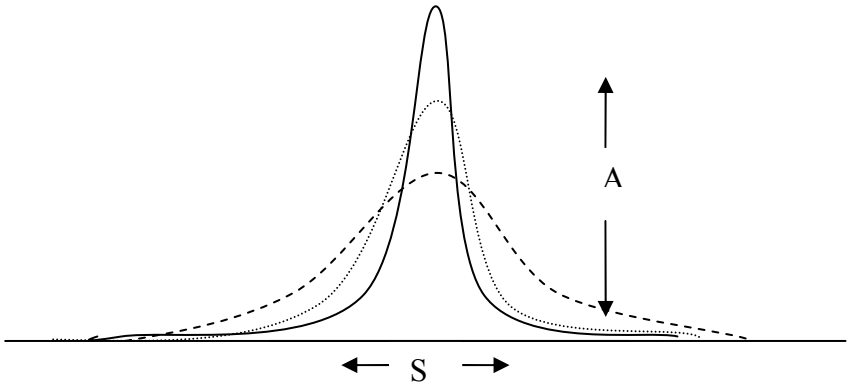


Figure 2.9

That represents the equation

$$S(\varphi)A(\varphi) \cong 1^{(1)} \Rightarrow A(\varphi) \cong \frac{1}{S(\phi)} \qquad (3)$$

However the distance $S(\varphi)$ is equal to:

$$S(\varphi,R) \cong 2R\phi \qquad (4)$$

Where R is the distance from the side of the component to the center of the rotation.

The equations (3), (4) show clearly that the first derivative depends not only on the rotational angle φ but also on the geometry of the component. However knowing the parameter R in our samples we can eliminate this factor of an unknown component simply by multiplying the first derivative, taken from the unknown

component, with a factor ($\frac{R}{R'}$), where R' is the distance from the side of the new component to the center of the rotation which is the center of the component.

This process gives the application the important capability to be used with the same data (such as the value of the first derivative) in various components in PCB images without any changes.

Giving these two elements as inputs to our system; (normalized) max derivative and direction of the rotation, it must be capable of recognizing small rotational angles of a component (as max derivative shows) in a PCB as the application requires. We manage to extract useful information from the image in order to avoid handling the image itself in our system, thus making the whole application much faster. The same principles are applied in any PCB component, so those characteristics can be used in any similar situation e.g. for other PCB chips. In later Chapters we will examine in detail the main system and we will show how these two elements can give us the rotational angle.

(1). As we can see from the figure 2.9 the value S(φ) can be used as another characteristic that can define the rotational angle of the component. However we choose not to use this parameter later, because as you can see from the equation (3) the value of S(φ) can be completely defined from the value of the first derivative. Having as input to our neural network the value of S(φ) will provide no additional information to our system.

Chapter 3. Artificial Neural Networks (ANN)

3.0 Introduction

Here, we make an introduction to artificial neural networks. In our application we make use of a specific type of neural network (higher order neural network), so it is imperative to know what neural networks are, how they work and what are their basic principles.

Work on artificial neural networks has been motivated right from its inception by the recognition that the human brain computes in an entirely different way than the conventional digital computer .The brain is a highly complex, nonlinear and parallel computer (information processing system). It has the capability to organize its structural constituents, known as neurons, so as to perform certain computations (e.g. pattern recognition, perception, and motor control) many times faster than the fastest digital computer.

In its most general form, a neural network is a machine that is designed to model the way in which the brain performs a particular task. We may use the following definition of a neural network viewed as an adaptive machine:

Definition: A neural network is a massively parallel distributed processor made up of simple units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. Knowledge is acquired by the network from its environment through a learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

The procedure used to perform the learning process is called a learning algorithm, the function of which is to modify the synaptic weights of the network in an orderly fashion to attain a desired objective.

3.1 Neuron

As we have already said a neuron is an information-processing unit that is fundamental to the operation of a neural network. The Block diagram 4.1 shows the model of a neuron.

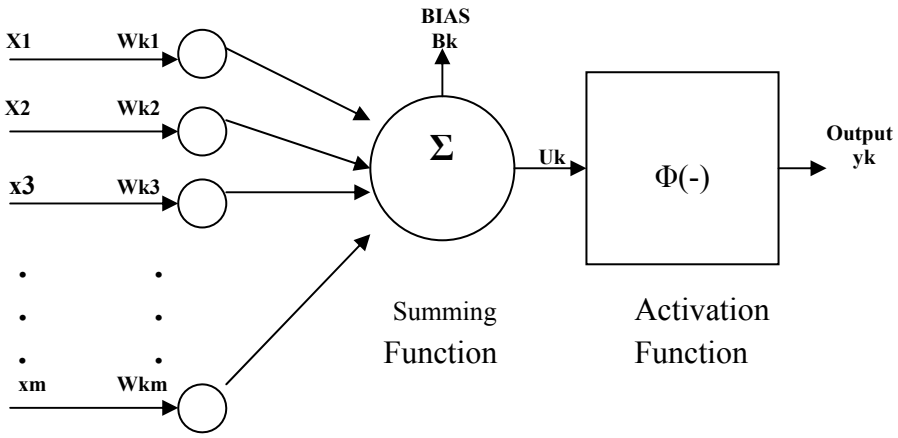


Figure 4.1

We can identify from the above diagram three basic elements of a neuron model:

1. A set of synapses or connecting links, each of which is characterized by a weight or strength of its own.
2. An adder for summing the input signals, weighted by the respective synapses of the neuron. The operations described here constitute a linear combiner.
3. An activation function for limiting the amplitude of the output of a neuron. The activation function is also referred to as a squashing function in that it squashes (limits) the permissible range of the output signal to some finite value.

The neuronal model as described is deterministic in that its input-output behavior is precisely defined for all inputs.

3.2 Network architectures

The manner in which the neurons of a neural network are structured is intimately linked with the learning algorithm used to train the network. We may therefore speak of learning algorithms (rules) used in the design of neural networks as being structured.

In general, we may identify three fundamental different classes of network architectures:

1. **Single-layer Feedforward network**

In a layered neural network the neurons are organized in the form of layers. In the simplest form of a layer network, we have an input layer of source nodes that projects onto an output layer of neurons, but not vice versa. As an example considered the figure 4.2.Such a network is called single-layer network.

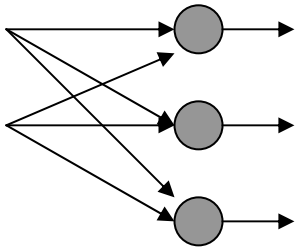


Figure 4.2 Feedforward network with a single layer of neurons

2. **Multilayer Feedforward networks**

The second class of a feedforward neural network distinguishes itself by the presence of one or more hidden layers, whose computation nodes are correspondingly called hidden neurons or hidden units. The function of hidden neuron is to intervene between the external input and the network output in some useful manner. By adding one or more hidden layers, the network is enabled to extract higher-order statistics. As an example of a fully connected neural network (every node in each layer is connected to every other node in the adjacent forward layer) see the figure 4.3.

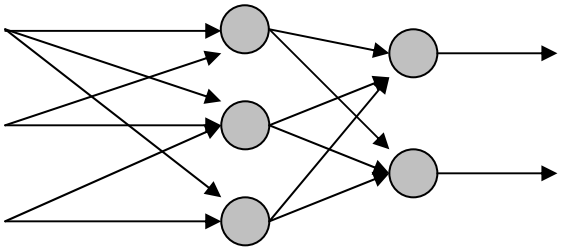


Figure 4.3

Input layer | layer of hidden layer | layer of output neurons

3. Recurrent Networks

A recurrent neural network distinguishes itself from a feedforward neural network in that it has at least one feedback loop. For example, a recurrent network may consist of a single layer of neuron with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in the architectural graph in figure 4.4.In the structure depicted in this figure there are no self-feedback loops in the network. Self-feedback refers to a situation where the output of a neuron is fed back into its own input.

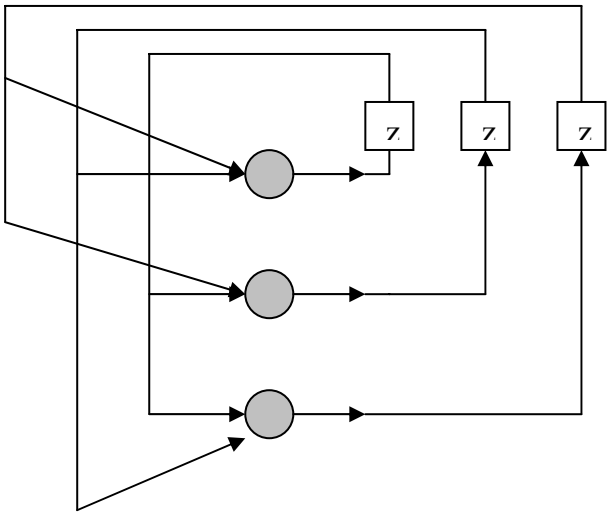


Figure 4.4 Recurrent network with no self feedback loop and no-hidden neurons

3.3 Benefits of neural networks.

It is apparent that a neural network derives its computing power through, first; from its massively parallel distributed structure and second from its ability to learn and therefore generalize. Generalization refers to the neural network producing reasonable outputs for inputs not encountered during training (learning).These two information-processing capabilities make it possible for neural network to solve complex (large-scale) problems that are currently intractable.

The use of (ANN) offers the following useful properties and capabilities:

1. Nonlinearity. An artificial neural neuron can be linear or nonlinear. A (ANN) made up of an interconnection of nonlinear neurons is itself nonlinear. Moreover, the nonlinearity is of special kind in the sense that it is distributed through the network.

2. Adaptivity. Neural networks have a built-in capability to adapt their synaptic weights to change in the surrounding environment. In particular, a (ANN) trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environment (i.e. one where statistics change with time); a neural network can be designed to change its synaptic weights in real time.

3. Evidential Response. In the context of pattern classification, a neural network can be designed to provide information not only about which particular pattern to select, but also about the confidence in the decision make. The later information can be used to reject ambiguous patterns, should they arise, and thereby improve the classification performance.

4. Fault tolerance. A neural network, implemented in hardware form, has the potential to be inherently fault tolerant, or capable of robust computation, in the sense that its performance degrades gracefully under adverse operating

conditions. For example, if a neuron or its connecting links are damaged, recall of a stored pattern is impaired in quality. However, due to the distributed nature of information stored in the network, the damage has to be extensive before the overall response of network is degraded seriously.

In contrast to the above useful capabilities the artificial neural network offers, it is often difficult to design good ANNs, since many of the basic principles governing information processing in ANNs are hard to implement, and the complex interactions among network units usually make engineering techniques, like divide and conquer, inapplicable. Fortunately newer algorithms such as Genetic algorithms, as we will see, tend to eliminate the above problem.

Chapter 4. Higher order Artificial Neural Networks (HONN)

4.0 Introduction

A higher order neural network is a network with only one hidden layer that involves high order activation functions. In mathematical terms a high order neural network can be expressed as:

$$\hat{y} = W^T S(x) \qquad (4, 1)$$

Where $x, \hat{y}$ are the inputs and the correspondents outputs, W is an L-dimension vector that expresses the connection weights of the network and S(x) a respective L-dimension vector with elements

$$S_i(x) = s^i(x), \; i = 1,2..., L \qquad (4, 2)$$

In equation (4, 2) s(x) is a continuous activation function. For our purposes we have used a Gaussian function of the form:

$$s(x) = \frac{m}{\sqrt{2\pi\sigma}} e^{-l\frac{(x-c)^2}{2\sigma^2}} + \lambda \qquad (4, 3)$$

where m, l, σ^2 are the magnitude, the gradient and the variance respectively and c, λ the horizontal and vertical transposition. We could also use other activation functions such as sigmoid functions of the form:

$$s(x) = \frac{m}{1 + e^{-l(x-c)}} + \lambda \qquad (4, 4)$$

In this case m, l represents the magnitude and the maximum gradient of the function and c, λ the horizontal and vertical transposition respectively.

In general, when the input is an N-length two dimensional vector the elements of s(x) are of the form:

$$\underbrace{s^i(x_1)\cdot s^j(x_2)\cdots s^k(x_N)}_{N\ldots times} \qquad (4,5)$$

For example a sixth order (HONN) with only one input and one output will have the following form:

$$y = 1 + s(x) + s(x)^2 + s(x)^3 + s(x)^4 + s(x)^5 + s(x)^6$$

while a third order (HONN) with 2 inputs and 1 output will have the form:

$$\begin{aligned} y &= 1 + s(x_1) + s(x_1)^2 + s(x_1)^3 + s(x_2) + s(x_2)^2 + s(x_2)^3 \\ &+ s(x_1)s(x_2) + s(x_1)s(x_2)^2 + s(x_1)s(x_2)^3 + \ldots \\ &\ldots + s(x_1)^3 s(x_2) + s(x_1)^3 s(x_2)^2 + s(x_1)^3 s(x_2)^3 \end{aligned}$$

The general equation that gives the maximum number of the higher order terms in each neuron is:

$$\beta = (q+1)^d \qquad (4.6)$$

where q, d the order of the neural network and the number of inputs respective. From (4, 6) we can deduce that as the order of the neural network and the number of inputs increase, we have a respective exponential increase in the number of neurons in the neural network

.

4.1 Approximation capabilities

Higher order neural networks have the ability to approximate regular functions at least in a region of their range of definition.

The following theorem describes the approximation capabilities of a (HONN).

Theorem 4.1 .We assume that we have a regular but unknown function y=f(x), and the mathematical model $\hat{y} = W^T S(x)$ .Then $\forall \varepsilon > 0$ there is an integer number L and a vector W^*, such that the output of (HONN) with L higher order connections and weight value $W = W^*$ to satisfy

$$\sup|\hat{y} - y| \leq \varepsilon$$

for all $x \in \Omega$ where $\Omega \subset A \subset \Re^n$ a region in the input region of A of the function.

The above theorem guarantees that if we have a satisfactory number of higher order connections in our (HONN), then is possible to approximate any continuous function with any accuracy in a given space.

4.2 Learning traits.

Let assume that $x \in \Re$ is the input of the (HONN) and $y \in \Re$ is the correspondent output and the f is the unknown function. It is obvious that y=f(x). Let also assume that $\hat{y} = \hat{f}(x)$ is an approximation function of the real f. Then the error that comes up would be:

$$e = y - \hat{y} = f(x) - \hat{f}(x) \qquad (4.7).$$

We can also observe that the error can be calculated without any further information even if f(x) is a completely unknown function. Now, let's assume a first order filter

$$\dot{z} = -az + e = -az + f(x) - \hat{f}(x) \qquad (4, 8)$$

where $z \in \Re$ the output of the filter and $a > 0$ a design constant of the filter. Due to the theorem 4.1 we can assume without loss of generality that the unknown variable in (4, 8) can be replaced from higher order neural network plus an approximation error $\omega(x)$ .In other words (4, 8) can be written as follows:

$$\dot{z} = -az - W^T S(x) + W^{*T} S(x) + \omega(x) \qquad (4.9)$$

where W^* the unknown weight values. For the approximation error we make the useful following assumption, direct result of the theorem 4.1:

Assumption 4.1: In a region $\Omega \subset \Re$ the approximation error satisfies the following condition

$$|\omega(x)| \leq \bar{\varepsilon}$$

where $\bar{\varepsilon} \geq 0$ is an unknown upper bound.

If we examine (4.9) more closely we can perceive the elements W as estimators of the unknown weight values W^*.If we define $\tilde{W} = W - W$ the equation (4, 9) will then take the form:

$$\dot{z} = -az - \tilde{W}^T S(x) + \omega(x) \qquad (4.10)$$

This last equation is of the form bounded input – bounded output (BIBO) meaning that if the term $\tilde{W}^T S(x) + \omega(x)$ is bounded and approaches to zero then the output of the stable filter (4, 9) also approach to zero. For the above if the (HONN) learns a real but unknown function is like forcing z to converge into a real small region of zero through weight learning W.

4.3 Least squares learning method.

Although there are many learning methods (lyapynov etc), for our proposes we will expound the least squares learning method.

The above method is described from the following theorem:

Theorem 4.2. Given the filter (4, 9), the learning rule

$$\dot{W} = -\gamma W + zS(x) \tag{4.11}$$

with γ a designed constant, guarantees that the output z of the filter will converge in an arbitrary small region

$$Z = \left\{ z \in \Re : | z | \leq \frac{\varepsilon}{\alpha} + \frac{1}{2}\sqrt{\left(\frac{\varepsilon}{\alpha}\right)^2 + \frac{2\gamma \mid W^* \mid^2}{\alpha}} \right\}$$

(arbitrary small region because the size of the region depends on the design constants α, γ and the upper bound of the approximation error $\omega(x)$).

The above theorem guarantees that if we have a starting condition z(0) within region Z then z remains in Z $\forall t \geq 0$. However if $z(0) \notin Z$ then there exist limited time T that $z(t) \in Z, \forall t \geq \mathrm{T}$.So out primarily goal to converge the output z in a arbitrary very small region of zero has been accomplished.

It is also obvious from the theorem 4.2 that the size of the set Z can be changed choosing suitable values for the design constants α, γ. In general, the γ value must be kept small in order to minimize the error that comes from the $|W^*|$, while α must be kept relatively large since it appears in the denominator of the equation that defines z.

Chapter 5. Genetic Algorithms

5.0 Introduction

In this chapter we make a brief introduction to genetic algorithms. It is necessary to understand the way genetic algorithms work since, as we shall see in later chapters, we use such processes to find an optimal structure for our neural network. Here we shall see all theoretical concepts of genetic algorithms and their practical implementations in various problems. Also we will compare genetic algorithms to other methods and we will examine all the advantages and disadvantages of each method.

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming. Evolutionary algorithms are ubiquitous nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, economics, operations research, ecology, and population genetics, studies of evolution and learning, and social systems.

A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as the chromosome, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces which are too large to be exhaustively searched. The symbol alphabet used is often binary, although other representations have also been used, including character-based encodings, real-valued encodings, and -- most notably -- tree representations.

Definition:A genetic algorithm is defined as GA=(B_0, M, Ω, Γ, Φ, Θ, t) where:

- B_0=(A_1 ,….,Aμ) ε P(B) the population from the set of all possible populations P(B) where Ai either as a binary or as real number coding
- M size of population
- Ω: P(B)→R fitness function
- Γ: P(B)→P(B) crossover function
- Φ:P(B)→P(B) mutation function
- Θ:P(B)→P(B) selection strategy
- t: R→(0,1) termination function

5.1 Description

The standard genetic algorithm proceeds as follows: an initial population of chromosomes is generated at random or heuristically. Every evolutionary step, known as a *generation*, the chromosomes in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), chromosomes are *selected* according to their fitness. Many selection procedures are currently in use, one of the simplest being Holland's original *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Thus, high-fitness (``good'') individuals stand a better chance of ``reproducing'', while low-fitness ones are more likely to disappear

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space. These are generated by genetically-inspired operators, of which the most well known are *crossover* and *mutation*. Crossover is performed with probability p_{cross} (the ``crossover probability'' or ``crossover rate'') between two selected individuals, called *parents*, by exchanging parts of their genomes (i.e., encodings) to form two new individuals, called *offspring*; in its simplest form, substrings are exchanged after a randomly selected crossover

point. This operator tends to enable the evolutionary process to move toward ``promising'' regions of the search space. The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some probability p_{mut}.

Genetic algorithms are stochastic iterative processes that are not guaranteed to converge; the termination condition may be specified as some fixed maximal number of generations or as the attainment of an acceptable fitness level. Figure 4.1 presents the standard genetic algorithm in pseudo-code format.

```
begin GA
  g:=0  { generation counter }
  Initialize population P(g)
  Evaluate population P(g)  { i.e., compute fitness values }
  while not done do
    g:=g+1
    Select P(g) from P(g-1)
    Crossover P(g)
    Mutate P(g)
    Evaluate P(g)
  end while
end GA
```

Figure 5.1: Pseudo-code of the standard genetic algorithm.

5.2 Basic Genetic Algorithms operations

There are three basic operations found in almost very algorithm:

1. Reproduction
2. Crossover
3. Mutation

Reproduction

This operation allows individual strings to be copied for possible inclusion in the next generation. The chance that a string will be copied is based on the string's fitness value, calculated from a fitness function. For each generation the reproduction

operator chooses strings that are placed into a mating pool, which is used as the basis for creating the next generation. For example, look at the table below.

String	**Fitness value**	**Percentage**
01001	5	19%
10000	12	46%
01110	9	35%

From this table the string *1000* is the fittest, and it should be selected for reproduction about approximately *46%* of the time, contrary to the string *01001, that* it is the weakest and should only be selected *19*% of the time.

Crossover

The crossover operation allows new solutions to be produced by randomly mixing the traits of older solutions. In particular the genetic algorithm (GA) select two strings at random from the mating pool .The strings may be different or identical it doesn't matter. The GA then calculates whether crossover should take place using a parameter called *Crossover probability.* If the GA decides not to perform crossover, the two select strings are simply copied to the new population (they are not deleted from the mating pool, they may be used multiple times during crossover).If the crossover does take place, then random splicing points is chosen in a string, the two strings are spliced and the spliced regions are mixed to create two (potentially) new strings. These child strings are then placed in the new population. For example say that the strings *10000* and *01110* are selected for the crossover operation. The GA selects a splicing point let say 3, so the following occurs:

$$\begin{matrix} 100|00 \\ \\ 011|10 \end{matrix} \quad \rightarrow \quad \begin{matrix} 01100 \\ \\ 10010 \end{matrix}$$

The newly created strings are *10010* and *01100.*

Mutation

The specific operation is performed on a single element value and as a result mutation changes the element value into a new one. In binary coding 1s are changed to 0s and 0s to 1s.In real number coding a random noise is added to coded values. The usefulness of this operation is to preserve the variety of solutions in each population. Mutation can be applied either during selection or crossover (though crossover is more usual).The GA has a mutation probability ,m, which dictates the frequency at which mutation occurs although this probability must be kept small as a high mutation rate can destroy fit strings. As an example, say that the GA decides to mutate bit position 4 of the string *10010*

$$10010 \rightarrow 10000$$

Example of mutation

5.3 Genetic Algorithm traits

Genetic algorithms provide robustness, efficiency and flexibility when searching a space for the optimum solution. GAs judiciously use the idea of randomness when performing a task. However GAs are not simply random search algorithms which can be inherently inefficient due to directionless nature of their search. GAs are not directionless. They utilize knowledge from previous generations of strings in order to determine a new generation that will approach the optimal solutions.

Genetic algorithms versus traditional optimization methods

The fundamental differences between GAs and other traditional methods are

- GAs use a set or population of points to conduct a search and not just a single point on the problem space. This gives GAs the power to search noisy spaces with local optimum points (in contrast with other techniques such as Hill climbing techniques that find only the local optimum in the neighbourhood of the current point).

- GAs are probabilistic in nature and not deterministic. This is a direct result of the randomization techniques used by the GAs.
- GAs use only limited information to guide them through the problem area. Many other techniques need a variety of information in order to converge in a solution. The only information GAs needs is a measure of fitness about a point in space. (Known as fitness function).This characteristic make genetic algorithms extremely useful real problems where additional information is hard to find.
- However one of the main disadvantages GAs presents is that they demand a great amount of time in order to converge, which makes their application nearly impossible in real-time problems.

Chapter 6. Selection of HONN structure for function approximation using genetic algorithms.

6.0 Introduction.

In earlier chapters we presented the basic concepts of neural networks especially for higher order neural networks as also for the genetic algorithms. We described the advantages and disadvantages of both neural networks and genetic algorithms. In this chapter we will describe the way genetic algorithms can be used in order to find the best (HONN) structure for our purpose, which is to approximate the functions given as inputs to the HONN. This process is very important since we will use the same method in our system in order to find the optimum structure to approximate the characteristics that we extracted from our image.

6.1 Methods for improving the structure of a neural network.

One fundamental problem of neural networks is to determine their internal structure. This procedure is often a very difficult and time-consuming task because it requires many tests and verifications of every topology. Moreover, the structure of a neural network is affecting both the ability of generalization and the training time required. Until now there is no analytical background to determine which topology is the optimum one, without using some heuristic methods which usually are problem depended. In general, these methods include:

- Method of testing and verification: According to this method neural networks with different structures are trained and the smallest structure with the best performance in the training sample is chosen.

- Method of destruction: In this method we start with a neural network with a large structure and, during the training, we reduce the number of neurons in the network.

- Method of construction: This method is the exact opposite. Namely we start with a small structure network and during the training we insert more neurons into the structure.

- Method using genetic algorithms: According to this method we use genetic algorithms in order to find the optimum topology using some measures such as the fitness value.

We choose the last one for being a more automated procedure as well as a more accurate method than the others.

6.2 Description of the algorithm

The algorithm we use to determine the structure of a given neural network using genetic algorithms is given below:

Training Cycle

- Increment Iteration Count
- Neural Net Learning
- Check Criteria To Stop? If Yes --> Stop
- Check Time To Evolve? If Yes --> Evolution Cycle
    - Evaluate Network Population
    - Rank Network Population
    - Store Image of Fittest Network
    - Select Most Fit Parents
    - Elitism
    - Crossover & Mutation
    - Increment Generation Count
    - Update Network Population
    - Continue training cycle

Fig. 6.1. The training cycle with the nested evolution cycle.

In order to show the above algorithm more analytically, let's consider a structure of a higher order neural network:

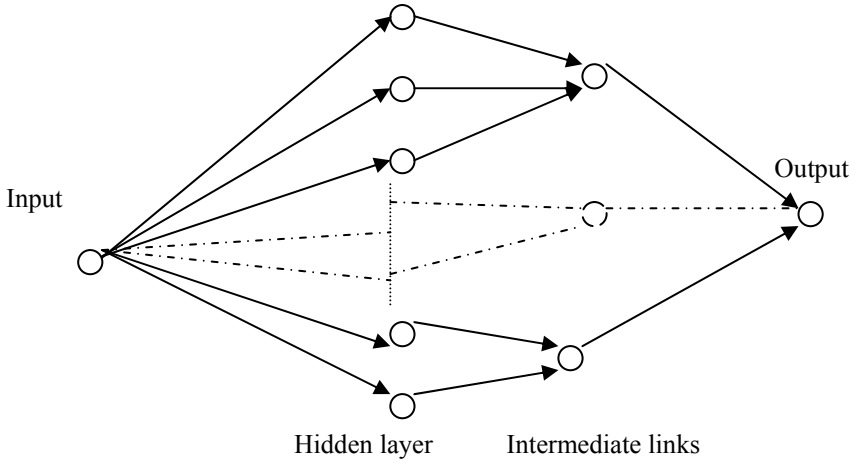


Figure 6.2 A simple structure of a HONN

As we have already mentioned a higher order neural network contains one hidden layer. The intermediate nodes have the following meaning: We assume that our network consists of a set of higher order networks; these networks are combined in order to give the final result .We will explain below that each subnetwork can consist of neurons with different activation functions.

6.3 Structure of genetic algorithm.

From the above structure of a higher order neural network, we can understand that a genetic algorithm must define all the activation connections in the network as well as the parameters of the Gaussian function for each subnetwork. The genetic algorithm must find an optimal combination of all the above from a relative large number of possible combinations.

More analytically, a genetic algorithm must have the following structures.

Binary Form.

In this structure we have the status of both the activations of an entire sub networks and the activations of single neurons in our HONN. For example, if we consider a 6^{th} order HONN with 4 subnetworks and if we define as structure A the structure that contains the activation of the subnetworks and B the structure that represents the activation of the neurons, a possible form of these two structures would be the following:

1	1	0	0	1	1	1
0	-	-	-	-	-	-
1	1	0	1	0	1	0
1	1	1	1	1	1	1

Structure A Structure B

Figure 6.3 Structures of the digital part of the chromosome

The dashes (-) in the activation status of the neurons in the second subnetwork (structure B) have the meaning that once this subnetwork is off (structure A) those neurons don't take part in the result of the genetic algorithms whatever activation value they might have. These two structures constitute the digital part of the genetic algorithm.

Real form.

This structure refers to the parameters of the activation function (eg. Gaussian) As we already mentioned, each subnetwork can have different values for every parameter in its activation function. If we consider again the example of the 4 subnetworks, a possible form of this structure can be as following (as activation function we choose the Gaussian).

	l	m	λ	c	α	γ	σ^2
1	1.82	0.983	-2.3	0.4	3.45	0.03	0.7
0	-	-	-	-	-	-	-
1	9.67	0.672	4.78	0.8	9.87	0.02	0.96
1	1.567	0.541	3.67	0.2	1.23	0.001	0.451

Structure A Structure C

Figure 6.4 .Structure of the analog part of a chromosome.

The dashes (-) have the exact same meaning as above. This structure is the analog part of the genetic algorithm.

Initialize population.

In the initialization stage we create a population of different topologies. The number of these different topologies remains the same during the search process. For the structures A and B we give possible values 1 or 0 with probability 50% and we also initialize structure C with random values within certain bounds that we give as inputs for all the parameters in the activation function.

Fitness value.

Every chromosome represents a structure of the network. During the training process we solve a differential equation system, which is:

$$\dot{z} = -\alpha \cdot z + (y - \hat{y}) \tag{7.1}$$

$$\dot{W} = -\gamma \cdot W + z \cdot s(x_i) \tag{7.2}$$

where α and γ the design constants, $s(x_i)$ the correspondent output of the i_{th} hidden neuron, W the active weights, namely the weights of activated connections which change during training, y the desired output and finally $\hat{y}$ the output of the HONN. When this training process ends we compute the fitness function for every structure. The fitness value can be compute from the following equation:

$$fitness\_value = \frac{ct}{1 + f} \tag{7.3}$$

Where ct is a constant and *f* the mean absolute error given from:

$$f = \frac{1}{N} \sum_{i=1}^{N} |y - \hat{y}| \tag{7.4}$$

where N is the size of the training set.

Selection.

In this stage, each structure is evaluated using a fitness function and assigned a fitness value. On the basis of their relative fitness values, structures in the current population are selected for reproduction. A stochastic procedure ensures that the expected number of offspring associated with a given structure s is u(s)/u(P), where u(s) is the observed performance of s and u(P) is the average performance of all structures in the current population. Thus structures with high performance are more likely to be chosen for replication while poor performing structures are eventually removed from the population. In the absence of other mechanisms, such a selective process would cause the best performing structures in the initial population to occupy an increasingly larger proportion of the population over time. So when the fitness value is computed for all the chromosomes, the chromosomes are sorted and a certain percentage of the best chromosomes are chosen to be the base for the next generation.

Crossover process

In this stage the selected structures are recombined using crossover, with two complementary search functions. First, it provides new points for further testing of structures already present in the population; Secondly, it introduces instances of new structures into the population. There are several ways to cross the parent nets. One way is to produce new offspring by randomly mating parents. As we know, every chromosome consists of three fields. The crossover operation is applied in each field separately with probability $p=1- p_c$ for each structure where p_c is the crossover probability.

For the digital part of the chromosome, we generate two random real numbers r_1, r_2 in the interval [0, 1]. If $r_1>p$ then we perform a single point crossover for the structure B. The exact same procedure is used for the structure B. We can see the crossover process in the following scheme.

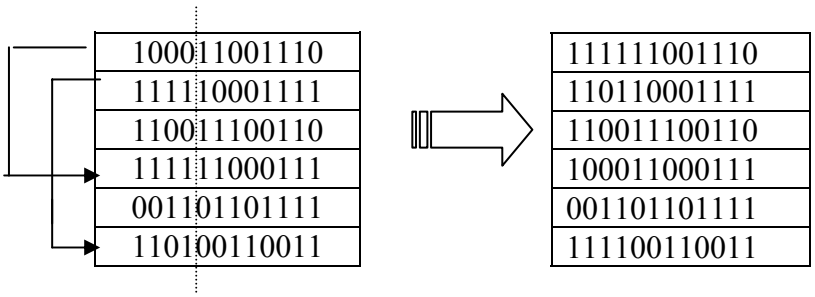


Figure 6.5 Crossover process for the structure B.

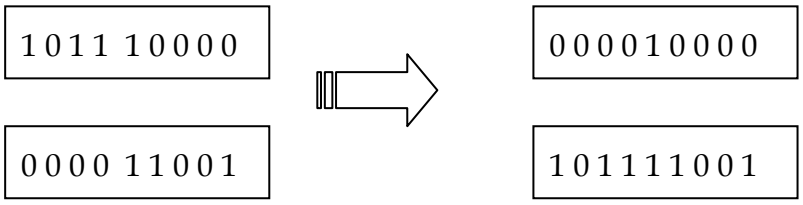


Figure 6.6: Crossover process for the structure A.

For the analog part of the chromosome we use the same procedure. Namely, for every parameter of the activation function we randomly choose a real number. If this number is larger than p then we perform crossover between the parents in this exact parameter. This operation is schematically depicted in figure 7.7

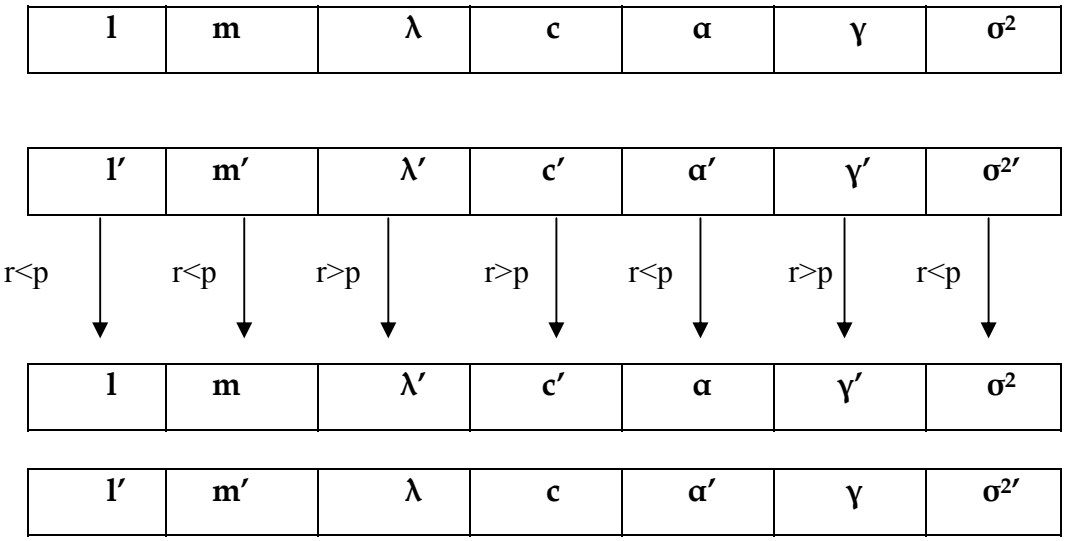


Figure 6.7. Crossover process for the analog part of the chromosome.

Mutation.

As in the crossover operation, we have here two types of coding: digital and analog. We define the mutation probability as pm .For each chromosome we generate a random real number, $r \in [0, 1]$**.** If r<pm then a mutation operation is performed either in structure A or in structure C. The reason why the mutation operation isn't performed on structure B is that the mutation operation hasn't the sense of fine tuning so there is no need to mutate the structure that is responsible for the activation or not of the subnetworks. In case structure A is chosen, the mutation operation randomly changes one bit of the structure. In case structure B is chosen, a random real number is added in one of the parameters of the activation function. Schematically the mutation operation is depicted in figure 7.8

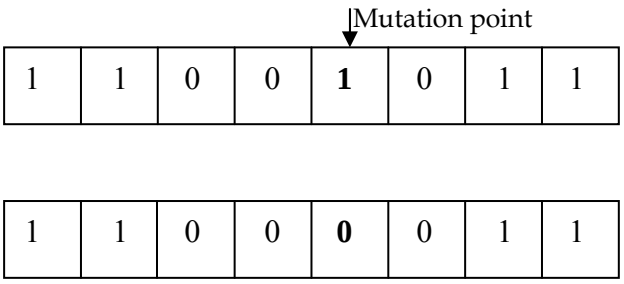


Figure 6.8 mutation operation.

Elitism

According to this procedure, the best chromosome (that is the chromosome with the best fitness value) always propagates to the next generation. The reason for this procedure is that without the Elitism the best chromosome could be lost in any generation and there is no guaranty that it will appear again in the following generations. In plain words, Elitism is a process that guarantees the asymptotic converge to the global optimum.

Stoping Criteria.

The above genetic algorithm stops in two ways. Either when the number of generations reaches a specific number or the mean absolute error e is smaller or equal to an also specific value.

6.4 Implementation of the algorithm and results

Here, we will represent the results taken from the implementation of the above algorithm. Our goal is to make a function approximation of the maximum derivatives taken from images with component rotation at various angles, as chapter 2 indicates. The preprocessing that we make at the training samples aims to normalize all maximum derivatives in a region from 0 to 1. As activation function for our higher order network we choose the Gaussian functions, namely functions of the form:

$$s(x) = \frac{mu}{\sqrt{2\pi\sigma}} e^{-l\frac{(x-c)^2}{2\sigma^2}} + \lambda$$

We choose the specific function because it gave better results (we also tried sigmoid functions but the HONN had very poor performance). Although we tried several approaches to implement this algorithm we finally conclude, for better accuracy, in using two different higher order neural networks, for when the component's rotation direction is clockwise or counter clockwise respectively. All the results below are produced by summing the results of both networks. Our initial training samples were taken from images with component rotation from -0.6 degrees to 0.6 with step 0.02 degrees (61 samples in total). However, we interpolate our data vector so as to reduce the step to 0.01 degrees, so our train samples were 121.

Our starting hypotheses for both neural networks were:

- Maximum number of epochs 20000
- Mean absolute error to stop the training process 1%
- Maximum number of HONN subnetworks 10
- Maximum order 10
- Crossover probability 0.3

- Mutation probability 0.2
- Pick 30% of the best chromosomes in each population.

Given all the above, our system gives the following results after the genetic process training:

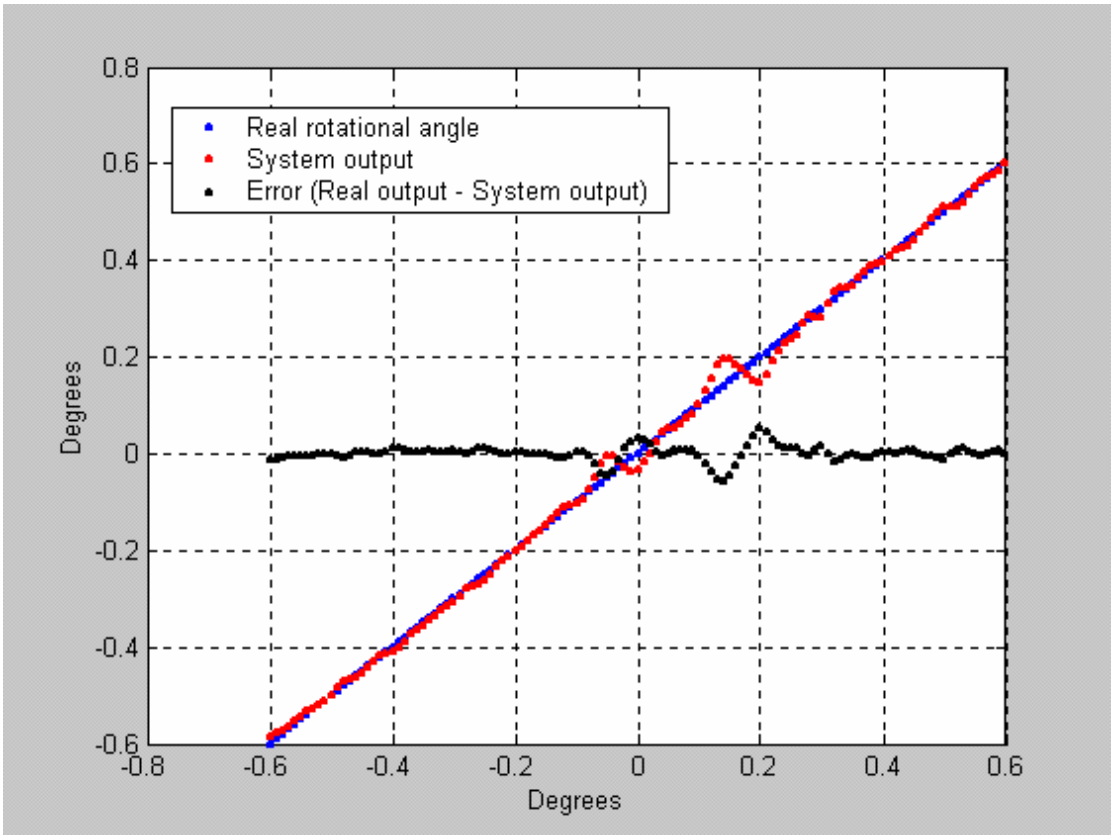


Figure 6.9

As the above figure indicates, the blue line represents the real rotational angle in our training samples, the red line is the system output degrees for each training sample and the black line the error in each sample. As we can see, the red line almost fits in with the blue one except in a very small region near zero where we have a wild oscillation of the system output. The mean absolute error of the above figure was about 0.01 degrees. This error represents the 1.7% and it is within application goals.

In the next figures we show the error in respect to the number of epochs in the genetic process in both neural networks

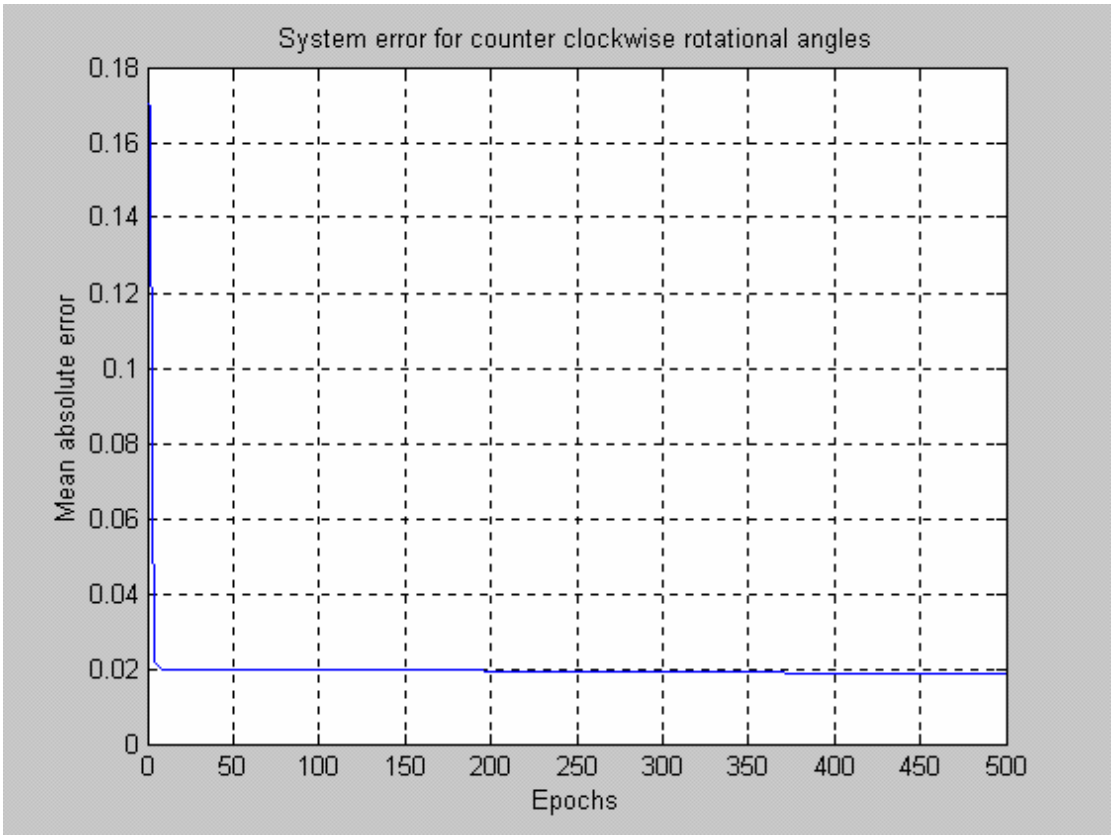


Figure 6.10

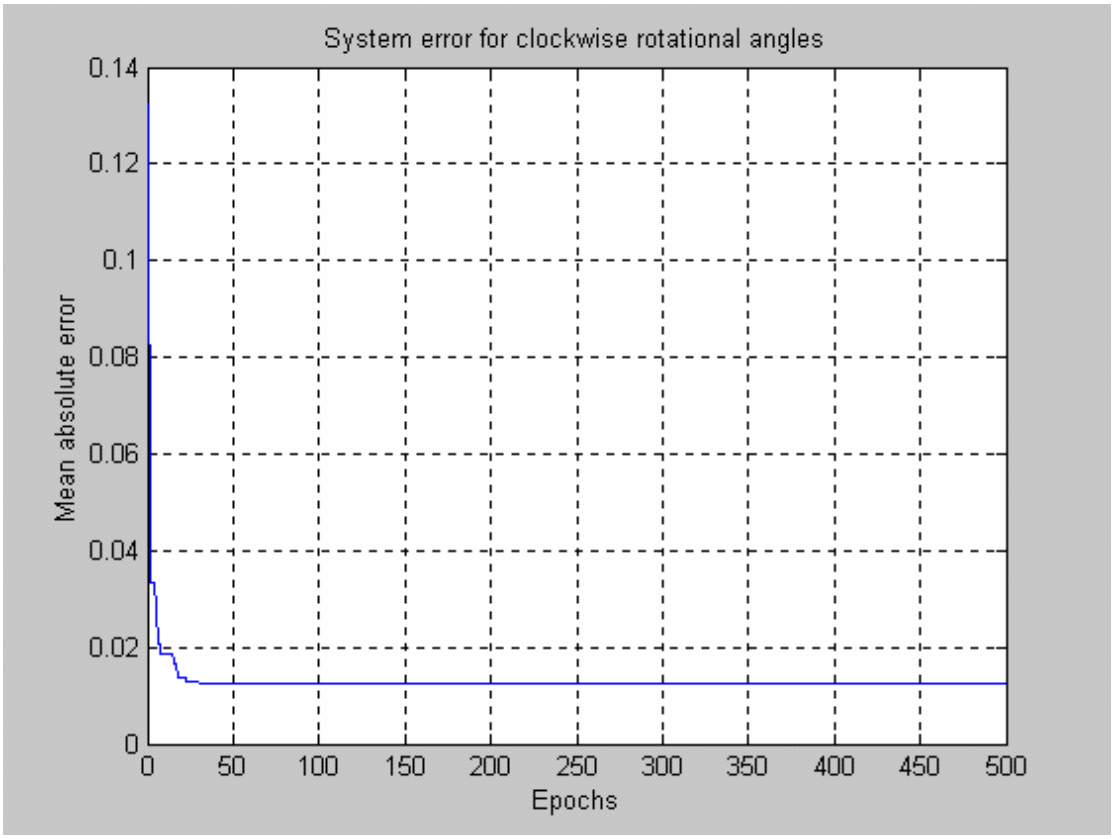


Figure 6.11

As we can see from figure 6.10 and 6.11 in both neural networks the average error is falling significantly in the first 100 epochs, after that the falling rate is very small. Although our stopping criteria were at 200000 epochs only minor changes were observed after the first 1000 epochs.

After the completion of the genetic process, the optimum structures of the higher order neural networks are given below.

Neuron status (structure B)	Subnetwork status (Structure B)
0.000000 0.000000 0.000000 0.000000 0.0000000.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000	0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

1.000000 0.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 0.000000	

Figure 6.12 (HONN structure for clockwise rotational angles)

In figure 6.12 we can see in detail the digital part of the genetic algorithm process that corresponds to the status of the neurons of the HONN (structure A) and of the subnetworks (structure B). Although we define our genetic algorithm to start with 10 subnetworks of second order, we see that the optimum structure was found having only one subnetwork (those having 1 in structure B).

We can make the exact same observations with the second HONN structure shown in the figure 6.13

Neuron status (structure B)	Subnetwork status (Structure B)
0.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 1.000000 0.000000 1.000000 1.000000 0.000000 1.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000 1.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 1.000000 1.000000 1.000000	0.000000 0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Figure 6.13 (HONN structure for counterclockwise rotational angles).

Conclusively, we can say that in this chapter we analyze in detail how genetic algorithms can be applied to determine the structure of a neural network and more specific of a higher order neural network. This process was necessary since we use a HONN in our main application to find the component's rotational angle in our PCB

images. As we saw from the above results, this method has a quite good performance, not only in accuracy, but also, as we shall see in appendix C, the main application has a very small execution time, so it can be used in real time systems. An important drawback of this method is that the execution time of the genetic algorithm process is very large (it took about 2 days to be completed), while it needs large amounts of memory, especially when we train HONNs of larger order. However, it is used only to determine the structure of the HONN and it is not taking part in the main application.

Chapter 7. Alternative way to measure the rotational angle of a PCB component.

7.0 Introduction

In previous chapters we analyze how to measure the rotational angle of a PCB component using higher order neural networks involved with genetic algorithms. In this chapter we introduce another technique and we will compare both methods, outlining all the pros and the cons of each of them. We must note that in our application we are concerned not only in the percentage of the error produced in each method but also in the speed and the size of each application as well. So, in order to compare their corresponding performance, we must take the above factors into consideration.

7.1 Measure the rotational angle using the RADON transform.

In recent years the radon transform has received much attention. The radon transform is able to transform two-dimensional images with lines into a domain of possible line parameters, where each line in the image will give a peak positioned at the corresponding line parameters. This has led to many line detection applications within image processing, computer vision, and seismics. The Radon transform was first established in 1917 by Johann Radon.

Several definitions of Radon transform exist, but they are related to each other. A very popular form expresses lines in the form

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

Where p is the smallest distance to the origin of the coordinate system, and θ the angle as it appears in the figure 7.1

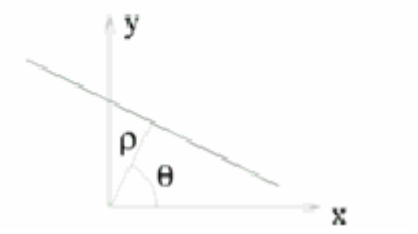


Figure 7.1

The Radon transform for a set of parameter (ρ,θ) is the line integral through the image g(x,y), where the line is positioned corresponding to the value (ρ,θ). The delta() is Dirac's delta function, which is infinite at 0 and zero for all other arguments (and whose integral over all values is equal to one), while in the digital version, the Kronecker delta is used.

$$\tilde{g}(\rho,\theta) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)\delta(\rho - \chi\cos\theta - y\sin\theta)dxdy$$

or the identical expression

$$\tilde{g}(\rho,\theta) = \int_{-\infty}^{\infty} g(\rho\cos\theta - s\sin\theta, \rho\sin\theta + s\cos\theta)ds$$

Using this definition, an image containing two lines is transformed into the Radon transform as shown below.

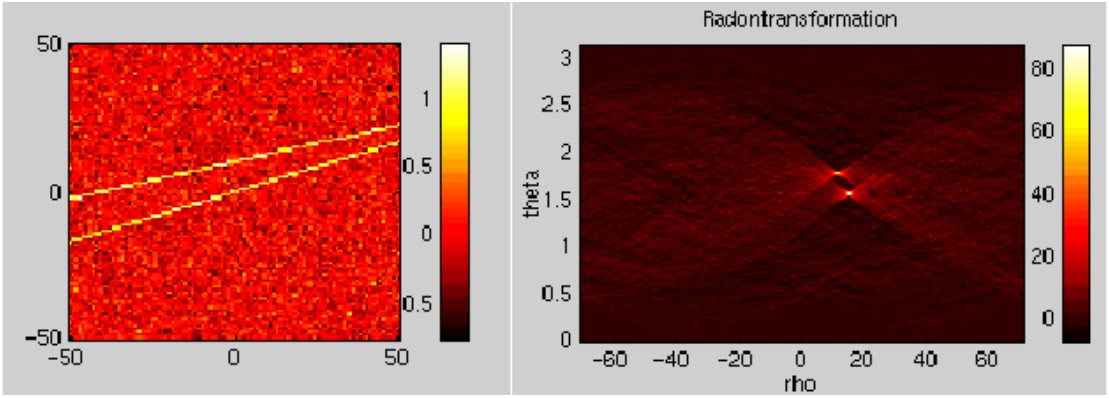


Figure 7.2 (Radon transform).

It can be seen that two very bright spots are found in the Radon transform, and the positions show the parameters of the lines in the original image. A very strong property of the Radon transform is the ability to extract lines from very noisy images, while an important drawback of this transform is that it's global in nature, so it can not tell the difference between long and short lines.

We can measure the rotational angle of a PCB component using a combination of edge detection and the above Radon transform.

Step 1. Perform edge detection. Let's consider the PCB image in chapter 2.1. To make Radon transform work we need to perform edge detection in our image so as to measure the rotation of the pins in the PCB component. We have tried several edge

detection algorithms (Sobel, Prewitt, Roberts, Laplacian of Gaussian, zero-cross, Canny) but we choose the Laplacian of Gaussian method because it gave the best results in a mean absolute error sense. The Laplacian of Gaussian method finds edges by looking for zero crossings after filtering the image with a Laplacian of Gaussian filter. Moreover, we tried to detect only the vertical edges, so as to leave out of the Radon transform the edges corresponding to the pad.

*Step 2.*Compute the Radon transform of the edged image. The Radon transform computes projections of the image at various angles. Peaks in the Radon transform indicate the possible presence of lines.

The results of the above process gave us the following diagram which indicates the output of the Radon transform.

7.2 Results

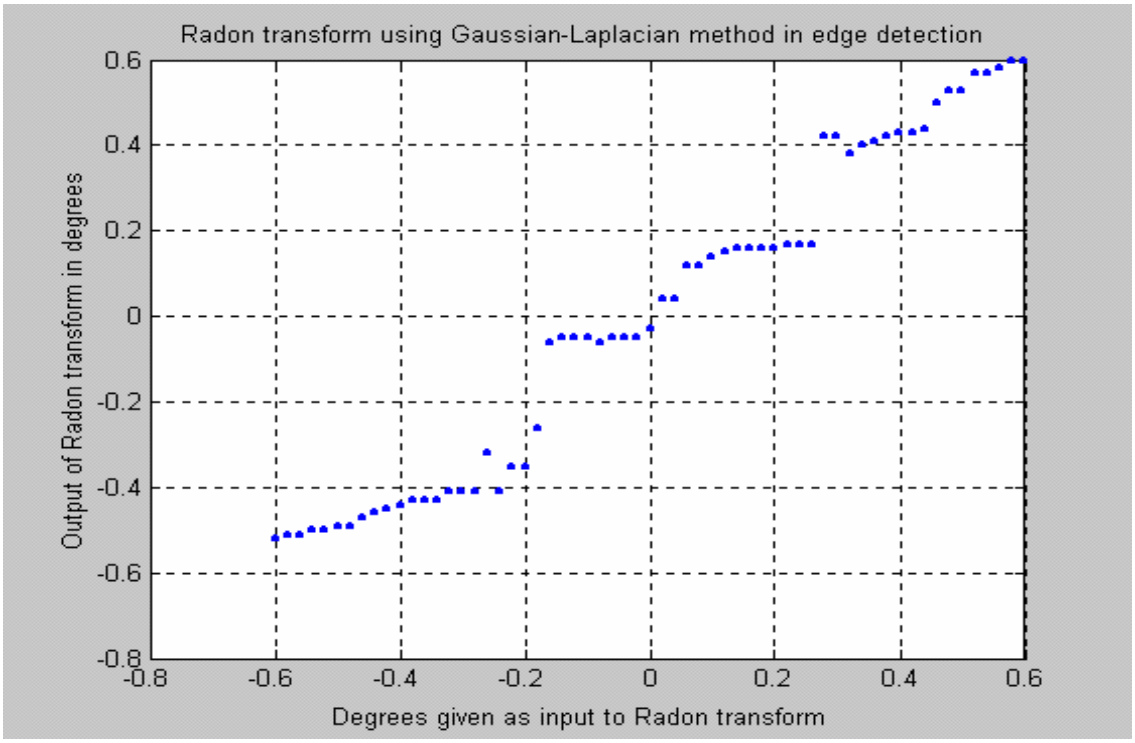


Figure 7.3

As we can see from the figure 7.3 Radon transform can measure the rotational angle of the PCB component. The whole concept of this method is very simple and easy to use. Due to the fact that it can compute projections through the image at various angles, this method doesn't require training samples as our previous

application did (neural network application). However, it lacks both in speed and accuracy as we can see from the following figure.

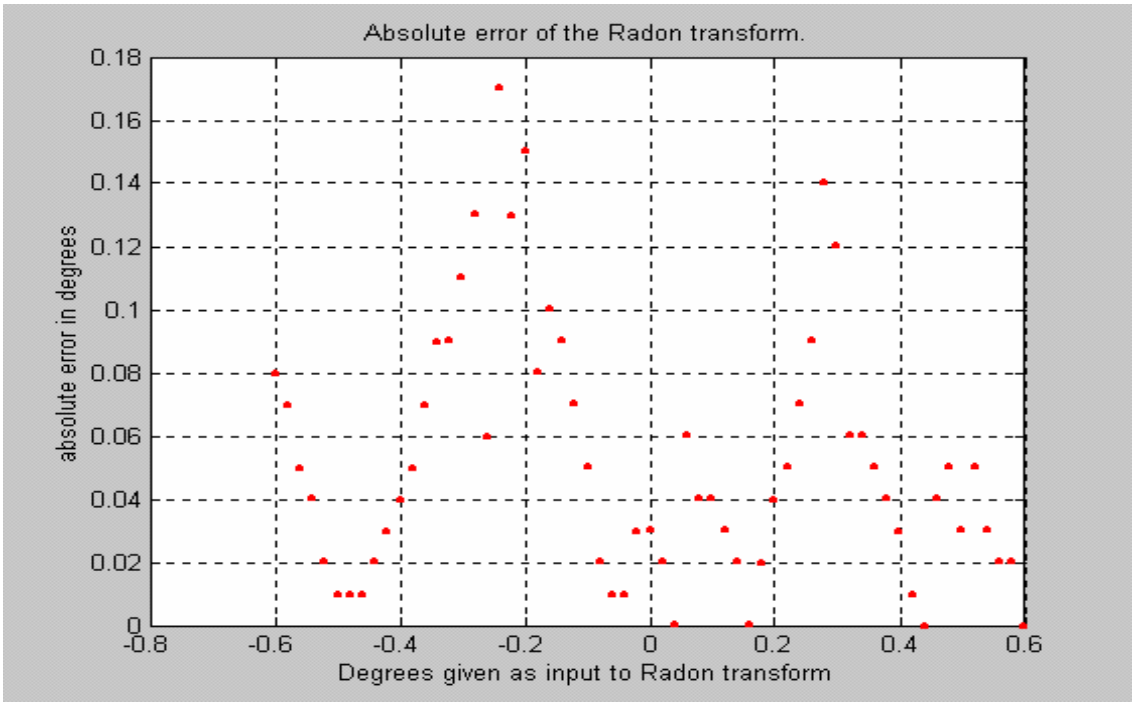


Figure 7.4

As we can see from the above figure, the absolute error of the Radon transform is large, reaching in some cases a 26% (as for -0.16 degrees), therefore being hardly within our application goals. Finally, the figure below indicates that this method is too slow to be used in real time systems.

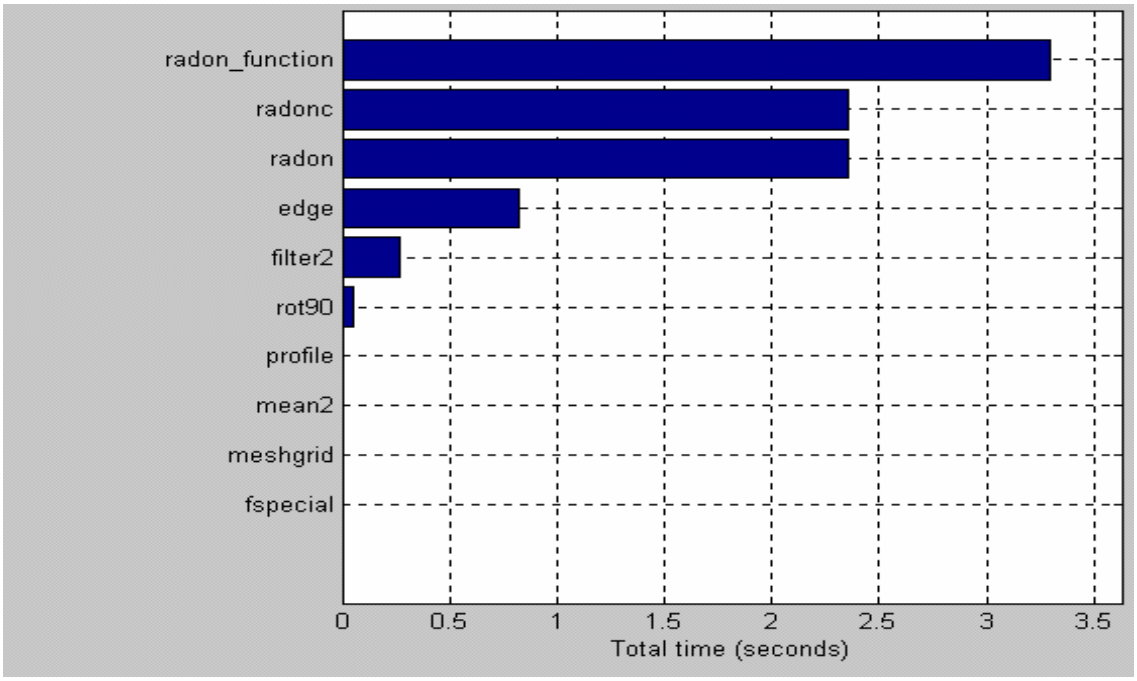


Figure 7.5

We must note here that the time required for this method to be accomplished is relevant and it depends on the accuracy parameter given to the Radon transform in order to compute the projections at various angles. The above time was measured using the Matlab environment (this method was constructed only in Matlab) and it will probably be faster under C/C++ environment. However, we should note here that the execution time would be by default (due to the Radon transform) too large for this algorithm to be used in real time systems.

We can conclude this chapter by saying that the Radon transform, as a traditional method for measuring the rotational angle of an object, as desirable might be, due to its simplicity, it does not cover the application standards, both in error and speed. In our previous model we make a quite productive trade-off between simplicity and performance that make it by far the best choice.

Appendix A PCB images

Here, we examine in detail the way we produced all samples for our neural network, namely all the PCB images. Due to the reason that we had no real samples, apart from two separate images (one with only the pad and one only with the component of the PCB), it was necessary to make artificial samples. This process, although not a part of the application, is very important for all the assumptions we made so far.

In order to make those samples we used two separate images. The first named h_qfp_14, is an 8bit 1024x1024 BMP image, containing the pad, and the second named h_qfp_11, is also an 8bit 1024x1024 BMP image, containing the PCB component. Without any former knowledge of both images, we assumed that the PCB component had an original rotation. To define this rotation, we rotate the component for various angles and we make use of the method specified in chapter 2. Namely, we take the first derivative of the projections of each side of the component and we compare the maximum first derivative in each component sample. The rotation was made around the center of the component which was calculated manually (calculate the width and height of the component and divide by two). We also rotate the component using a bilinear method in order to be more accurate. This process showed us that the PCB component had an original rotation of 0.2 degrees in the up and left side of the component and -0.02 degrees in the down and right side. This abnormal situation can be explained in one or more of the following ways.

Reasons for having different component angles in each side of the same component:

- The center of the rotation doesn't match with the center of the component
- The component has different side lengths (horizontal and vertical).
- The length of the lead varies in its side
- The axis of the camera that took the PCB image isn't in vertical position with the horizontal PCB axis or we have a gradient between the component and the PCB.

Therefore, we continued by rotating the component by -0.2 degrees and taking only the upper side of it. The exact same procedure was also followed for the second bmp image with the pad.

Our application goal was to detect and to measure high-handed very small rotational angles of the component in a bound region from -0.6 to +0.6 degrees. Due to the discrete nature of the image, we took samples of rotated components with step 0.01 degrees because no smaller step was possible (step smaller than 0.01 produced the exact same image even when rotating the component with bilinear method). Although in our application we train our neural network with characteristics from different images with step 0.01 degrees, this was done with interpolation of the characteristics with this step. However we considered our goal to be detecting and measuring angles with maximum estimated error of 0.01 degrees.

To produce the final image containing both the pad and the component we used a threshold (after aligning the centers of both the pad and the component), in order to overlay the pad with the component. More specifically, we chose as component threshold the value t=58. When the pixel value of the component is equal to or greater than t, we replace the pad's pixel value with that of the component; otherwise we leave the pad's pixel value intact. All the image samples used in this application were taken using this process. If we need to alter in some way this process (e.g. change the threshold value) all possible assumptions we made for this application (e.g. errors, neural network weights etc) must be recalculated.

Appendix B Manual for the Genetic algorithm software

Here we analyze the way genetic algorithm software works. This program (Genetic.exe) is used in order to train and find an optimum structure for the higher order neural network, as described in previous chapters. As activation function for the higher order neural network we used the Gaussian function, which has the form:

$$s(x) = \frac{m}{\sqrt{2\pi\sigma}} e^{-l\frac{(x-c)^2}{2\sigma^2}} + \lambda$$

The figure 1 shows schematically how this program works.

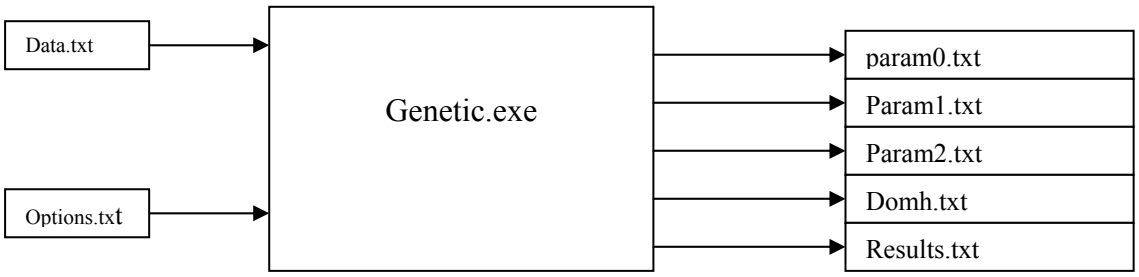


Figure 1

Input files.

As the figure 1 shows, this program takes as inputs two txt files. The first called ***data.txt*** contains the data of the function we want to approximate. The format of this file has the form:

$Input1_1\ Input2_1 \ldots Output1_1\ Output2_1 \ldots$

$Input1_N\ Input2_N \ldots Output1_N\ Output2_N \ldots$

Where N the number of the training data.

The second input file is called ***Options.txt*** and has the following structure:

IVal1 IVal2 IVal3 IVal4 IVal5 fVal1 fVal2 fVal3 fVal4 fVal5 fVal6 fVal7 fVal8 fVal9 fVal10 fVal11 fVal12 fVal13.

Where:

IVal1: The order of HONN
IVal2: The number of the subnetworks
IVal3: Number of inputs of HONN
IVal4: Number of outputs of HONN
IVal5: Number of epochs for the GA (stopping criteria)
fVal1: Mean absolute error for the GA(stopping criteria)
fVal2: Crossover probability
fVal3: Mutation probability
fVal4: Lower bound for the parameter c in Gaussian function
fVal5: Upper bound for the parameter c
fVal6: Lower bound for the parameter l in Gaussian function
fVal7: Upper bound fir the parameter l
fVal8: Lower bound for the parameter m
fVal9: Upper bound for the parameter m
fVal10: Lower bound for the parameter lambda (λ)
fVal11: Upper bound for the parameter lambda (λ)
fVal12: Lower bound for the parameter sigma (σ^2)
fVal13: Upper bound for the parameter sigma (σ^2).

Note that all the *Ival* values must be printed as integers and all the *fVal* values as floats

Output files.

After the completion of the training the genetic.exe produces five txt files.

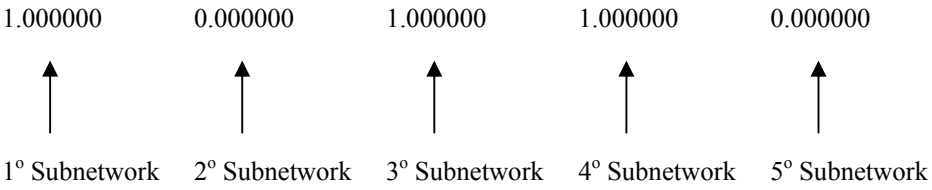
Param0.txt.

This file contains the activation status of each neuron of the HONN ('1'-On, '0'-Off).The format of this file would be like the following.

1.000000	0.000000	1.000000	1.000000	0.000000	→	1^o Subnetwork
0.000000	1.000000	1.000000	0.000000	0.000000	→	2^o Subnetwork
0.000000	0.000000	0.000000	0.000000	0.000000	→	3^o >>
1.000000	1.000000	1.000000	1.000000	0.000000	→	4^o >>
1.000000	0.000000	1.000000	0.000000	1.000000	→	5^o >>

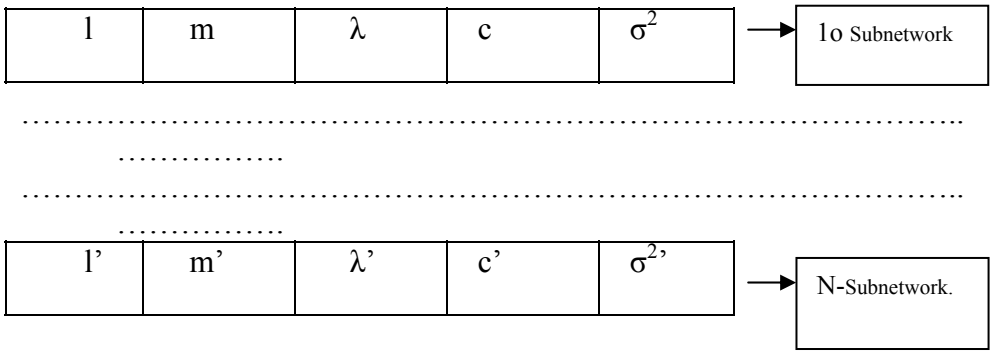
Param1.txt

This file contains the activation status of each Subnetwork ('1'-On, '2'- Off). The format of this file would be like this:



Param2.txt

This file contains the parameters of the Gaussian activation function for each subnetwork and it has the following form.



All of three files are used for the main application program. Additionally the genetic.exe program produces two more file that corresponds to the evolution of the genetic process. These files are:

Domh.txt

This file contains the evolution of the Genetic process and has the following structure:

Epoch number *fitness value* *Mean absolute Error*

Results.txt

This file contains the error per training sample and has the following structure:

Input *Output* *Output of HONN* *Error*

The time of completion of this program depends on numerous factors (hardware, number of inputs, order of HONN, subnetworks of HONN, number of epochs, mean absolute error) so it won't make any sense to profile the time needed by the program. Suggestively we say that for our application, using the parameters mentioned in chapter 6, the time of completion of this program was about 2,1/2 days

in a PENTIUM II 350MHz running Windows 98. Due to this rather large time of execution we present a new program which is a variation of genetic.exe. This program has the ability to load the txt files of a previous training (param0, param1 param2) and to further continue the training process, reducing the estimated error of the previous training. In plain words, with this program (Load_val_genetic.exe) we can stop and continue the training process. However, we also need the first program to produce the original txt files.

Finally, we provide one additional function that is totally irrelevant to this application program. This function is named ***splint.cpp*** and it is used to perform an interpolation of a given vector of data. It is used in order to calculate possible maximum derivatives of rotational components with step smaller than 0.02 degrees. By doing so, we multiply our training samples for the HONN, thus giving the network a better training process. We choose as interpolation method the spline, because splines tend to be more stable than polynomials with less possibility of wild oscillation.

Appendix C Manual for the main application software

In this appendix we analyze how the main application program works. Before this program is started it is imperative to have the neural network trained, since here we make use of the txt files produced by the genetic.exe program. In this program we are extracting characteristics from an image with unknown rotational angle and as system output we have the component's angle with an estimated error. Schematically the main application function is showed in figure 1.

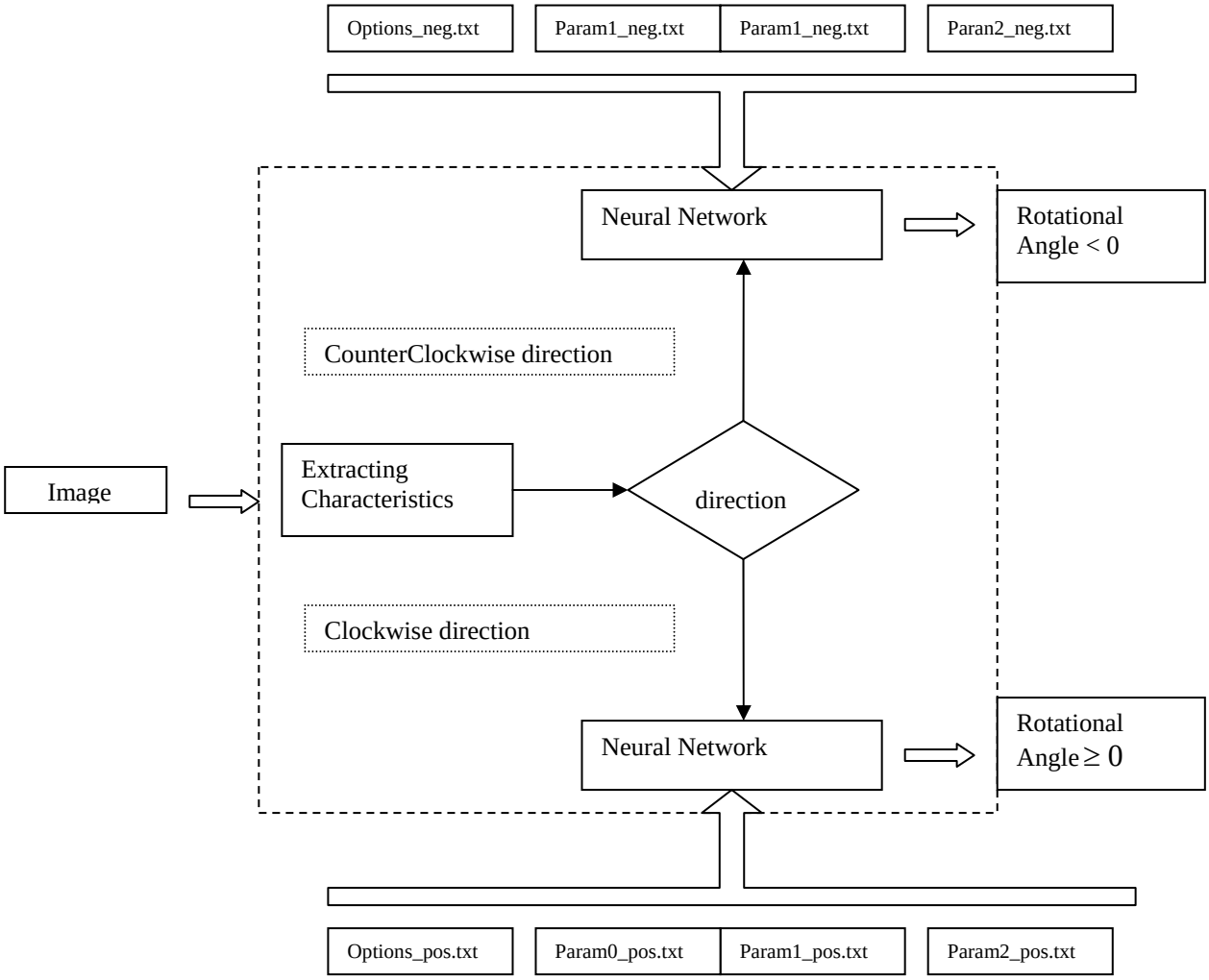


Figure 1

In the above figure the dashed rectangular indicates the main application program and the block arrows represent the system inputs. In more detail, the "extracting characteristics" part, implements the following three steps:

Step1. It loads the image in memory.

Step2. It holds in memory only the up side of the component, which it rotates in a way that the pins of the component face left.

Step3. It makes a vector by vertically projecting every pixel value.

Step4. It models the projected vector with a polynomial

Step5. It takes the first derivative and it gives as first output the maximum first derivative

Step 6. It finds the maximum and the minimum values around the maximum derivative and it normalizes the value of the maximum derivative

Step6. In the point where the first derivative was found, it computes the direction of the rotation and gives that information as the second output.

The two outputs of this part (normalized maximum derivative and direction) become inputs to the neural network system which implements the following steps: According to the direction of the rotation it can load the higher neural network that corresponds either to the clockwise direction or to the counter clockwise direction as figure 1 shows. In any case, the steps of the neural network subsystem are:

Step1. Load from the options.txt file the structure of the HONN.

Step2. Load from the param0.txt, param1.txt, param2.txt the status of the HONN network (activation status of neurons, activation status of the subnetwork, parameters of the activation function for each subnetwork).

Step3. Given the input characteristic, compute the output of the HONN, namely the rotational angle of the component.

We must say that steps 1 and 2 are implemented only once and that if we have more that one images with unknown rotation then the neural network system implements only the last step.

We have tested the above application in a PENTIUM II 350MHz running Windows 98. Below we list the total amount of execution time needed. Note, that we

measured the execution time for the first inserted image so additional time was required loading all the txt files as illustrated in figure 1. The execution time would be significantly smaller for other images after the first one. Moreover, we didn't make use of any optimization options of the compiler; because we left the user to decide about time/memory trade offs.

The profiler records how many times each function was called (hit count) as well as how much time is spent in each function call. The Func Time column reports how much time it took for the function to run in milliseconds. The next column shows what percent of the total profile time is represented by the function time. Func+ child Time refers to the total time profiled for the function as well as any function called by that function. The next column shows what percent of the total profile time is represented by the Func+child time. The Hit count column reports the number of times the function was called. The function column displays the decorated name of the function.

Profile: Function timing, sorted by time

Program Statistics

Command line at 2001 Aug 10 04:09: "C:\Project\HYPER\Debug\HYPER"
Total time: 216,658 milliseconds
Time outside of functions: 4,468 milliseconds
Call depth: 5
Total functions: 45
Total hits: 49
Function coverage: 51,1%
Overhead Calculated 7
Overhead Average 7

Module Statistics for hiper.exe

Time in module: 212,190 milliseconds
Percent of time in module: 100,0%
Functions in module: 45
Hits in module: 49
Module function coverage: 51,1%

Func Time	%	Func+Child Time	%	Hit Count	Function
111,415	52,5	171,958	81,0	1	Load_Image
58,803	27,7	58,803	27,7	1	RapFileLoad
19,406	9,1	212,190	100,0	1	main

6,902	3,3	6,902	3,3	1	Acquire_ROI
6,698	3,2	6,698	3,2	1	Make_projection
3,020	1,4	3,020	1,4	1	load_val
2,511	1,2	2,586	1,2	1	polyfit
1,146	0,5	1,146	0,5	1	RapFileSize
0,537	0,3	0,537	0,3	1	RapImgFree
0,515	0,2	0,681	0,3	1	procces_net
0,354	0,2	0,354	0,2	2	find_local_max
0,262	0,1	0,262	0,1	1	Rot_detect
0,191	0,1	0,191	0,1	20	funcs
0,166	0,1	0,166	0,1	7	summing
0,075	0,0	0,075	0,0	1	solve
0,073	0,0	0,073	0,0	1	fitting_parameters
0,024	0,0	0,215	0,1	1	derivative
0,021	0,0	0,021	0,0	1	RapImgInit (rhapsody.dll)
0,020	0,0	0,020	0,0	1	RapImgGetPar (rhapsody.dll)
0,018	0,0	0,700	0,3	1	learning_step
0,016	0,0	0,016	0,0	1	RapInitialise (rhapsody.dll)
0,015	0,0	0,715	0,3	1	learning_procces
0,001	0,0	0,001	0,0	1	feed_net

Profiler for the main application program

As we can see from the time profiler, about 80% of the total execution time is spent to load the image file from the disk to memory. However, the total execution time is quite small, so the specific application can be used in real time systems.

Below we will examine in detail what each function of the application does.

Structures.h

In this header file we are defining all the structures that we use in the application. These structures are:

- struct image_rap_info{
  unsigned char **data;
  int h,w;
  }

This structure contains the image matrix as well as the height h and the width w of the image matrix. We use this structure to allocate memory not only for the whole image matrix but also for the upper side of the component.

- struct _chromosome{
  float **nurs;
  float *dikt;
  float *analog[5];
  float fitness;

```
    int valid;
}
```

With this structure we define a chromosome for the genetic algorithm process. It contains all the information a chromosome needs. This structure uses a matrix named nurs of length (subnetwork x neurons) which contains the status of the neurons in the HONN (‘1’-On, ‘0’-Off). It also has a vector of length (subnetworks) which contains the status of the subnetworks of the HONN. Finally it contains a matrix of length (subnetwork x 5) with the parameters of the activation function. Note that in this application we are not dealing with the genetic algorithm process so the fitness and the valid parameters of the chromosome are not used. This structure is simple code reuse from the genetic.exe program and it helps us to load the optimum structure of the HONN from the files that genetic.exe produced.

- ```
  struct _NN{
      int order;
      int neuron_num;
      int networks;
      int in_num;
      float *inputs;
      int hidden_num;
      float *S;
      float ***W;
      int out_num;
      float **outputsin;
      float *outputs;
      float **H;
      float *I;
  }
  ```

In this structure we define the higher order neural network structure. Where order, neuron_num, networks, in_num, out_num, hidden_num, are the order of the HONN, the number of neurons in one subnetwork of the HONN, the number of subnetworks, the number of the inputs, the number of the outputs and the number of hidden neurons respectively. We also provide a vector named “inputs” of length in_num that contains the inputs of the HONN as well a vector named “outputs” of length “out_num” that it contains the output of the network. The vector S has length “hidden_num” and contains the activation function as it has described at chapter 5. The 3D-matrix W has a size of (networks x neuron_num x out_num) and contains the weights for each neuron in the neural network. The vector outputsin has a length (out_num x networks) and its element contains the output value of its subnetwork
  ```

(note that each subnetwork has the same number of inputs and outputs as the whole network). The matrix H has length (networks x neuron_num) and it contains the status of each neuron in the HONN. Finally the vector I has length (networks) and its element of this vector contains the status of each subnetwork of the HONN.

- struct _io_pair{
    float *in;
    float *out;
  }

We use this structure to pass our inputs to the neural network. Although we could use simple vectors instead of pointers, since we know the number of inputs we use beforehand, we did that for code reuse purposes only (the same program with minor changes can be used with more or less inputs and outputs).

Constants.h

In order to extract the characteristics from the image we must give the geometry of the component. This file defines all the necessary geometry information of the image (centre of the component, length of pad, distance from first to last pad) to extract the up side of the component. It also defines the search region used in *find_local_max* function as well as a variable named *second_max* used in *fitting_parametets* function. These two variables are used to determine the maximum derivative as we will see later. Note that all numbers in *Constants.h* file represents pixels.

Load_image.c

This is the first function that we use in our application. It allocates the memory and loads the image matrix. This function uses other rhapsody functions that are contained in rhapsody.lib file. Images are allocated in 32-bit boundary line. When the number of pixels in not on a 32-bit boundary, a line will be present on right-hand side of the image page, because that part of the page is not overwritten by loading an image from file. In normal situations *Load_image.cpp* it returns 1 unless one or more rhapsody functions returns an error message. To see the return values of rhapsody functions please refer to the RhapsodyUserManual.

Acquire_ROI.c

This function holds in memory only the upper side of the component. Doing so, all the memory allocated in the previous function is now freed. The upper side of the component is allocated in such way that the pins of the component are facing left. We do that by rotating this side by 90 degrees and then inversing the object.

Make_projection.c

Here we make the projection vector by summing vertical every pixel value of the upper side of the component. To normalize the results so that can be from 0-1 we divide each projection with the highest possible value they might have namely with ROI_columns*256 since the images are 8-bit grey scale. This function returns 1 I normal execution.

Find_local_max.c

This function is used to find local maximums in a given vector. In order to find maximums within an area, we must provide a specific search_region (given at *constants.h* file) .This function looks for the largest element in a region 2xsearch_region (search_region pixels to the left and search_region pixels to the right). Due to the very noisy images we also we provide this function with a certain threshold variable so that every maximum derivative must have value greater than this threshold. In normal situation this function returns 1 unless no maximum derivatives above the threshold value found in the data vector.

Fitting_parameters.c

Here we initialize all parameters that will be used in *polyfit* function.

Polyfit.c

In this function we make a polynomial fitting in a given vector of data. The order of the polynomial is given as input to the function however the function doesn't check their parameter inputs e.g does not check if the order of the polynomial is positive or negative, so extra cautious must be used for the inputs of this parameters. The memory for the parameter used by polyfit was reserved by fitting_parameter function. Finally this function make use the solve function in order to solve the simultaneous equations from the polynomial fitting. The solve.cpp is internal function of polyfit.cpp and is used only in that point.

derivative.c

Here we compute the first derivative of the modelling data and the results are placed in a vector given as input to the function.

Rot_detect.c

In this function we compute the direction of the rotation of the component. The method followed here has been analyzed in detail at chapter 2. Due to the reason that the upper side of the component wasn't in a central position we need to adjust the up and down side vectors of chapter 2. Also for greater accuracy we took tree columns instead of one (± 1 columns form the point where max derivative found). The result of this function is placed to a variable and can have only two possible values, 1 that indicates that the rotation has counter clockwise direction or 0 that indicates the opposite. In success this function returns 1.

Besides these functions the application also makes use of two more functions that assist all the above.

Save_ROI.c

Here we save a previously allocated image page in a file. This function also makes use of Rhapsody functions so in case of abnormal execution see the Rhapsody User manual. This function is helpful in debugging mode only (to examine if the upper side of the component has been correctly allocated) and it is not used in other way,

Funcs

This function computes the result of the polynomial fitting. It takes as inputs the parameters of the basis functions a_M and the position x and it calculates the equation:

$$y(x)= a_1 + a_2 x + a_3 x^2 + ... + a_M x^{M-1} \qquad (2.1)$$

Where M the order of the polynomial.

myAssert

This function is used in every step of the whole application. It takes as input an expression argument and it controls the normal execution of the program as well it evaluates expressions. When the result is FALSE it exits the program with an integer that has a specific meaning. Also this function is used to identify logic errors and to

handle unexpected outputs of the functions. The application can run without this function however it is extremely useful, especially in debug mode.

Below we give possible return values of myAssert function.

Error values	**Meaning**
1-16……	Error in main function. Cannot allocate memory for the neural network structure.
490……	Error in main function. The file that contains the weights of the neurons (weights_pos.txt or weight_neg.txt) doesn't exist or is not in same directory with the executable program.
491……	Error in main function. The file that contains the structure of the HONN (option_pos.txt or options_neg.txt) doesn't exist or is not in same directory with the executable program.
250……	Load original image failed. The image file may not exist or there is no available memory. If none of the above please check of Rhapsody User manual
101-102……..	Internal errors on load_Image function. Image file may be invalid or there is no available memory. If none of the above there is an error in Rhapsody functions. Please check the Rhapsody User Manual.
260……	Error in Acquire_ROI function.
209……	Internal error in Acquire_ROI function. There is no available memory.
210……	Internal error in Acquire_ROI function. There is no available memory to allocate the matrix.
280……	Error in Make_projection function.

401…… Internal error in Make_projection function. There is no available memory to allocate variables.

Error values	Meaning
290……	Error in find_local_Max function. No local maximum found
501…….	Internal error in find_local_Max function. There is no available memory to the system
502…….	Internal error in find_local_Max function. The search_region defined in constants.h file is greater than the whole data vector.
503……	Internal error in find_local_Max function. This shouldn't have happened. Check the normalization procedure.
300……	Error in fitting_parameters function. Possible due to memory problems.
6011……..	Internal error in fittitng_parameters function. The search region given in constants.h file used for polynomial fitting is greater that the data vector.
601-609……	Internal errors in fitting_parameters function. No memory available to allocate specific parameters
400……	Error in polyfit function. This shouldn't have happened. Check the inputs of the functions.
500……	Error in derivative function. Possible due to memory problems
801-802……..	Internal errors in derivative function. There is no available memory to allocate specific parameters.
501…….	Error in find_local_max function in different part of the program.than error 290.No maximum derivative found.

450……. Error in Rot_detection function. This shouldn’t have happened.

Error values	Meaning
800……	Error in load_val function. Check the inputs of the function.
980……	Internal error in load_val function. The file that contains the activation status of the subnetworks of the HONN (param1_pos.txt or param1_neg.txt) does not exist or it isn’t in the same directory with the main application program.
981……	Internal error in load_val function. The file that contains the activation status of the neurons of the HONN (param0_pos.txt or param0_neg.txt) does not exist or it isn’t in the same directory with the main application program.
982……	Internal error in load_val function. The file that contains the parameters of the Gaussian activation function for each subnetwork of the HONN (param2_pos.txt or param2_neg.txt) does not exist or it isn’t in the same directory with the main application program.
900…….	Error in the learning_process function. Check the inputs of this function.
901-902……..	Internal error in learning_process function. Cannot allocate memory for specific parameters.
903……...	Internal error in learning_process function and error in feed_net function. Check the rightness of the inputs.

904……...Internal error in learning_process function and error in process_net function. Check the rightness of the inputs.

Error values	Meaning
905……...Internal error in learning_process function and error in summing_net function. Check the rightness of the inputs.	

In the next page we will see some new images with unknown rotational angles. As we have already said in Appendix A, it is possible that every side of the component has different rotational angles due to some predefined factors. In the next table we will show the rotational angles of every side of the component of the 20 new images (from h01_1Qf-h10_2Qf).

The first four columns show the result of our application in degrees for each side. For example, for the image h_01_1Qf, the left side has a rotational angle of about 0.25 degrees where the bottom side of the same component has a rotational angle of about 0.061 degrees.

To explain this extraordinarily big difference between the rotational angles of the sides of the same component we measure the distance S for each side. The distance S is the same parameter from the equation (3) at chapter 2.2.5 which is the horizontal distance between the first and the last pin of the component. As we can imagine a big S corresponds to a small first derivative and thus to a big angle. For example we can see that the left side of the first image has a very big rotational angle because the horizontal distance between the first pin and the second pin of the component is about 4.375 pixels whereas the correspondent distance for the bottom side is only 2.1872. Apparently the centre of the rotation wasn't the centre of the component. The columns H-K show the distance S of each side of the component.

To be able to produce one "general" rotational angle we compute firstly the average derivative of the four sides in each component and then give this average derivative as input to out neural network. This Average rotation is shown at column L.