DEMOCRITUS UNIVERSITY OF THRACE

SCHOOL OF HEALTH SCIENCIES

DEPARTMENT OF MOLECULAR BIOLOGY & GENETICS

Master's Programme of Studies

«Translational Research in Molecular Biology and Genetics»

Title

"Approaching the phase problem of X-Ray crystallography with feed forward neural networks"

By Chatzidimitriou Dimitrios

Supervisor: Glykos M. Nicholas

MASTER THESIS

April 2015

Abstract

Crystallography has a huge impact in our understanding of how biological molecules interact and function. It is an experimental scientific field where the main purpose of scientists is to determine the arrangement of atoms in crystalline solids. X- ray crystallography is a technique in which the atoms in a crystal under study cause a beam of incident x-rays to diffract and give a diffraction pattern. From this diffraction pattern someone can measure the angles and intensities of the diffracted beams. However, in order to produce a three-dimensional picture of the density of electrons within a crystal, a crystallographer also needs the phases of the diffracted waves. This constitutes the phase problem in crystallography. Several methods have been developed to overcome this obstacle, among them chemical modification methods and more mathematical approaches, which consider the relationships between the measured intensities and treats them with analytical and statistical techniques. The later are called direct methods and the purpose of this study, is to investigate whether artificial neural networks, are able to approximate the relationships used in these methods and therefore estimate the phases of the diffracted waves, using only the observed intensities. Neural networks are algorithmic models, which can be trained to estimate or approximate functions dependent on a large amount of inputs and are generally unknown. For the aforementioned purpose several feed-forward networks have been tested, using backpropagation learning techniques. The results obtained showed that these networks were unable to assign proper phases to the aforementioned intensities. That leads to the conclusion that this class of neural networks is unable to learn such relationships, and more sophisticated custom networks or techniques may be needed to obtain better results.

Key words: X –ray crystallography, phase problem, direct methods, artificial neural networks

Contents

Chapter 1: Introduction to Crystallography and Artificial Neural Networks

As the title suggests, this thesis is mainly concerned with crystallography (more specifically with the phase problem) and artificial neural networks. Before the analysis of the main subject of this project begins, it would be useful for the reader, to become familiar and understand some basic concepts, regarding these two scientific fields. This is the purpose of this chapter – to introduce those concepts to the reader and set the theoretical framework. A thorough presentation falls outside the scope of this introduction but more information can be found, by consulting the provided references.

After this introduction, in chapter 2 the aim of this study is explained. That section could also be considered as a 'preface' of this study. A description of some important characteristics of neural networks follows and the implementation of these networks is discussed. In chapter 3 the experiments that have been made are described and their results are presented and analyzed. The study closes with the conclusions in which the outcome of the experiments is been discussed and suggestion for future research are been made.

In the text wherever mathematical relationships are discussed the following notation has been used:

The scalars are represented with italic letters.

The vectors are represented with **bold** small letters.

The matrices are represented with **BOLD** capital letters.

1.1 Introduction to Crystallography

X- ray crystallography, can de defined as a tool with which scientists can identify the atomic and molecular structure of a crystal. The knowledge of this structure can be very important in many scientific fields. However it is not possible to 'see' this structure by the usual methods because these components of matter (atoms and molecules) are too small for us to view. X- ray crystallography as the name implies, uses x-rays which are a form of electromagnetic radiation. Microscopes which are used to see objects that are too small for the naked eye use light. Light itself is a form of electromagnetic radiation of wavelength between about 400nm (violet) and 800nm (red). ([1] p.18) Although microscopes also use electromagnetic radiation to picture small objects, their resolution is

not high enough to enable us to define the molecular structure of an object under study. We cannot see the details of an object with a microscope unless these details are separated by at least half the wavelength of the radiation used to view them. [21] Because of this and the spectrum of the visible light, there is no point in trying to view atoms that are separated in molecules by distances of the order of 10^{-8} cm (0.1 Angstrom) with a microscope. In order to see atoms we need to use appropriate radiation with wavelengths in the nanometer range. X-rays are this kind of radiation, with wavelengths ranging from 0.01 to 10 nanometers, corresponding to frequencies in the range of 30 petahertz to 30 exahertz (3×10^{16} Hz to 3×10^{19} Hz) and energies in the range of 100eV to 100 keV. That been said, if we could built a super microscope that would enable us to view atoms, then this microscope would have to employ X rays rather than visible light. ([4] p.1-5)

1.1.1 A microscope analogy

When we use an ordinary optical or electron microscope, a beam of radiation falls into the object under study and this radiation is then scattered by the object. This scattered radiation is then recombined by an appropriate lens system and results in an image of the scattering matter appropriately magnified. A light microscope operates by means of a series of lenses which diverge and focus the light passing through them, while an electron microscope uses magnetic lenses in an analogous manner. In microscopes like these where lenses are used to recombine the scattered waves, the relationship between the phases of those waves is being maintained. The above mentioned relationship between the phases refers to the phase difference, which is the difference, expressed in degrees, between two waves having the same frequency and referenced to the same point in time. For example Fig.1-1 shows the appearance of two waves with the same frequency and wavelength, being propagated side by side, as for instance, in two adjacent strings. Although the waves are identical in general form they are seen to be ‘out of step’, as if the wave in each string had set out at different times. ([1]-p. 114-121)

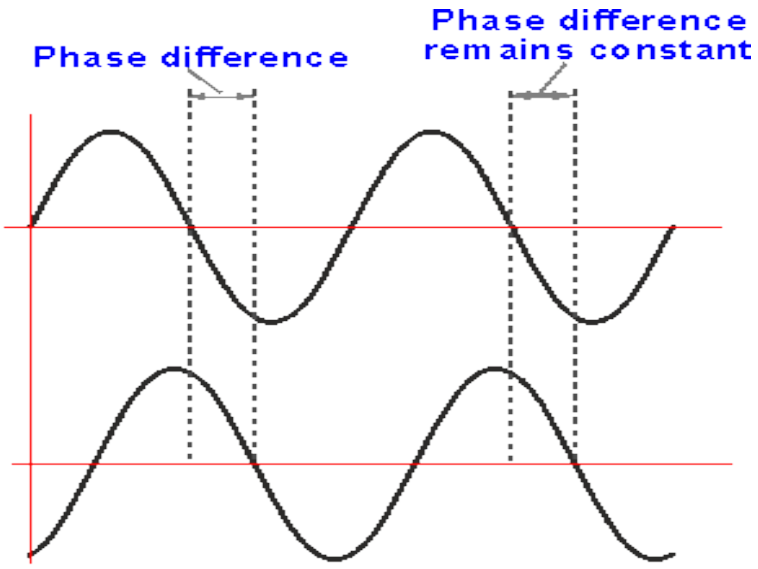

Figure 1-1. Phase difference between two waves

The reason we are interested in the phase relationship between the waves is that it plays an important role on the result of the summation of waves. The net amplitude caused by two or more waves traversing the same space is the sum of the amplitudes which would have been produced by the individual waves separately (principle of superposition). Waves exactly in phase (phase difference=0) reinforce each other because their wave crests coincide. This is known as constructive interference. Waves exactly out of phase (phase difference=π) cancel each other, because troughs coincide with crests, giving no resultant wave. This is known as destructive interference. If the waves are partially out of phase then the degree to which they affect each other depends on their phase difference and the relative magnitudes of the amplitude maxima of each wave. ([1]-p. 114-121)

On a smaller scale than that of optical microscopes which employ visible light, X-rays whose wavelengths are much shorter, are scattered by the electrons in atoms. If an appropriate lens system existed, we could recombine those scattered X-rays, as in a hypothetical 'X-ray microscope' and get the requested image. Scattered X-rays, however, cannot be focused normally by any currently known experimental technique, hence direct visualization of an image as would be formed in a microscope is beyond present technology. Longer wavelength X-rays [22] or scanning tunneling microscope techniques [23] enable us to get a picture on the shape of molecules, but only those on the surface of a structure. ([4] p.1-5)

This is the reason that leads us to resort to the less obvious phenomenon of diffraction. It is an alternative but indirect way to view the molecules and it involves studies of solids in a crystalline arrangement. This crystalline arrangement (from which the term

crystallography comes from) is essential because the diffraction pattern of a single molecule is too weak to be observable. When the molecules are arranged in a crystal, however, this pattern is reinforced because of the repeating units (molecules) and can be readily observed at specific points that have a direct relationship with the shape and form of the crystal. A diffraction pattern has potentially the same information content as a direct image, but this information is not obviously expressed in structural terms. The result of the analysis of this pattern is a complete three-dimensional elucidation of the arrangement of atoms in the crystal under study. The information is obtained as two elements. The first one is the atomic positional coordinates which indicates the position of each atom in the repeated units of the crystals. From these a scientist can calculate, with high precision inter-atomic distances and angles of the atomic components. Thus, he can learn about the conformation of molecules in a crystal. The second element is the atomic displacement parameters which indicate the extent of atomic motion or disorder in the molecule. ([4] p.1-5, [1] p. 21-22)

As mentioned above x-rays is a form of electromagnetic radiation with wavelengths shorter than visible light. Thus, the effects of these rays when they impinge on an object with similar dimensions to their wavelength can de described by using the theory of Christian Huygens [24] who suggested that light was a wave like motion analogous to water waves. Diffraction effects are observed only if the obstacle used is not too much greater than the wavelength of the waves that impinge on it. Each wave has an undulating displacement, which is called the amplitude of the wave and a distance between crest, which constitutes its wavelength. The displacement is periodic in time and/or space. When an object scatters a beam of light (or x-rays), if someone wants to assess the disturbance of the beam, he needs to know the relative phases (phase differences) of all the scattered waves. That happens because the relative phases have a profound effect on the intensities of the scattered beams, as a result of the principle of superposition. In an x-ray experiment the beams scattered by an object (crystal) are captured on photographic films, or some other detectors (see [2] Chapter 6). The relative phases and intensities of the scattered beams are determined by the atomic arrangement in the crystal. X-ray detection devices however, can only sense the intensity of the waves. ([4] p.1-5, [1] p. 21-22)

This is not the case in optical microscopes, where the lenses used, are composed of glass and refract visible light waves, collect the waves scattered and combine them again with due appreciation for their relative phases. That been said a comparison can be made between light microscopy and X-ray diffraction as shown Fig.1-2.

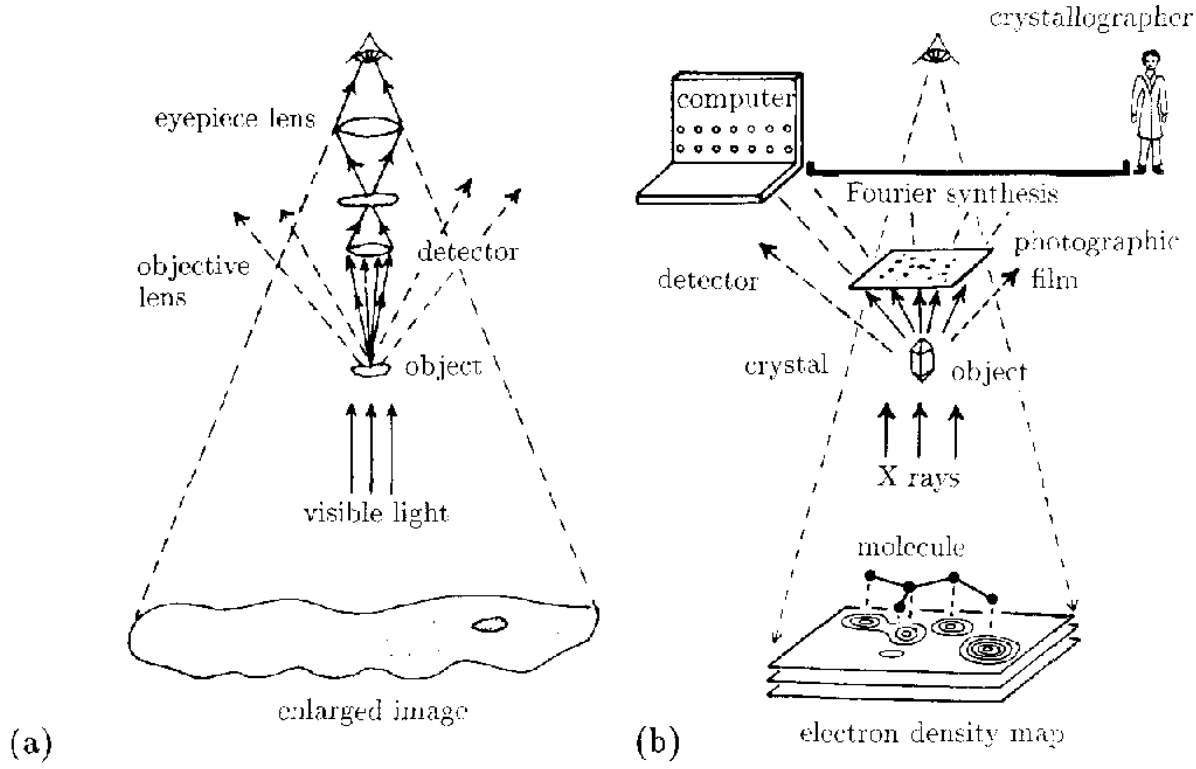

Figure 1- 2. Comparison between a) light microscopy and b) X-ray diffraction

In x- ray diffraction, beams which are scattered by electrons in the crystal are essential in order to see small objects, such as atoms. As mentioned earlier we cannot use lenses in this situation. The only things we can directly obtain are the intensities of the diffracted beams (the diffraction pattern) on the photographic film (or detection device). However, the relative phases are lost, and they must be derived in a different way. This lack of phase information is what we refer to as the **phase problem**. Should the relative phases be obtained, a crystallographer can then recombine all the information (intensities-phases) with proper mathematical computation techniques (Fourier synthesis) and obtain an electron density map which in turn gives information about the structure of the molecules in the crystal under study.

1.1.2 Crystals, Lattices and Symmetries

In the previous section we introduced the term ‘crystal’. In this section this term will be described. The established basic scientific definition of a crystal is that is a solid composed of a regularly repeated arrangement of atoms. ([4] p.33) Solids which possess a long-range, well- defined, three dimensional ordering of molecules which are in close proximity to one another and have relatively strong interactions between them, belong to a state known as crystalline state. ([1] p.21) The most obvious property of a crystal is it’s macroscopic geometrical shape which is a direct result of the aforementioned ordering of molecules. Crystals are bound by plane faces. These faces are a consequence of the regular stacking of molecules in layers. Every face in a crystal represents a plane parallel to a molecular layer. However, someone must be careful before he characterizes a solid as crystal because with the unaided eye, some materials may seem as crystals while they are not. For example fragments of glass and quartz (which is a crystalline) look similar to each other, but glass in not crystalline. Glass is amorphous because it has an atomic arrangement that shows only short-range order over a few atomic dimensions. ([1] p.11-21, [3] p.7)

A crystallographer who wants to study the molecular structure of a material with x-ray diffraction must have a crystal of this material. For example biologists and biochemists who want to study molecules such as proteins, can form crystals of the macromolecules of interest, provided that the appropriate conditions for their crystallization can be found. (More on this subject can be found in [2] - chapter 4, [4] - chapter 2, and [3] p. 489-499)

As mentioned above crystals present an internal regularity. We call that structure which is regularly repeated in space the motif. The motif is the structural unit and can be quite complex. It could be a single molecule or a group of several molecules, a group of ions, or whatever is appropriate to describe the overall geometric arrangement. Now that the motif is defined we can create a conceptual array of points which defines the geometrical relation between the motifs. This array is called the lattice. ([1] p.60-63) The lattice can be created if each group of atoms, that is repeated in the crystal at regular intervals (the motif) is replaced by a representative point. This collection of points is called the crystal lattice. A crystal structure therefore can be expressed with the conceptual relationship:

$$\text{Crystal structure} = \text{lattice} * \text{motif}$$

Where the operator ‘*’ stands for convolution, a mathematical operation, which in this case associates the motif in each point in the lattice. This can be pictured in Fig. 1-3.

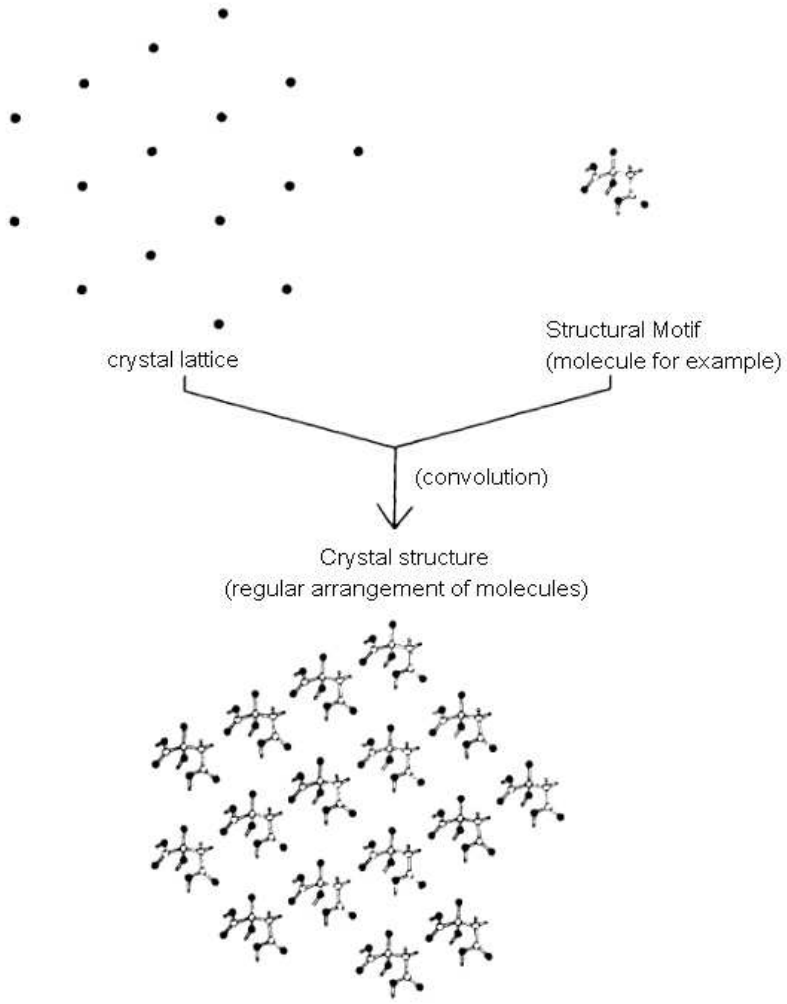

Figure 1- 3. Relationship between Crystal lattice, Motif and Crystal structure

The two- dimensional lattices, as the one depicted in Fig.1-3 are called plane lattices while, three- dimensional lattices are called space lattices. Both of these lattices are defined in terms of crystallographic unit vectors and crystallographic unit cells. Unit cells help us represent the regularity and symmetry of the lattice. The unit cell is the basic building unit of a crystal and is repeated continuously in three (or two) dimensions to form it. It can be considered as the analogous of bricks which are repeated in order to construct a wall. Using the same analogy the edges of a crystal (plane faces) are represented by the edges of that wall.([4] p.7)

A conventional crystallographic unit cell (in three dimensions) is a parallelepiped defined by the space lattice and serves to display the symmetry of the lattice in a convenient manner. This cell in turn is defined by three (or two- in two dimensions) non-coplanar lattice vectors. These vectors specify three directions which define the

crystallographic axes. In Fig. 1-4 a unit cell in three dimensions is depicted which is described by the unit vectors **a**,**b**,**c** which in turn define the crystallographic axes respectively. Various two dimensional lattices are also shown with their unit cells and the vectors that define them.

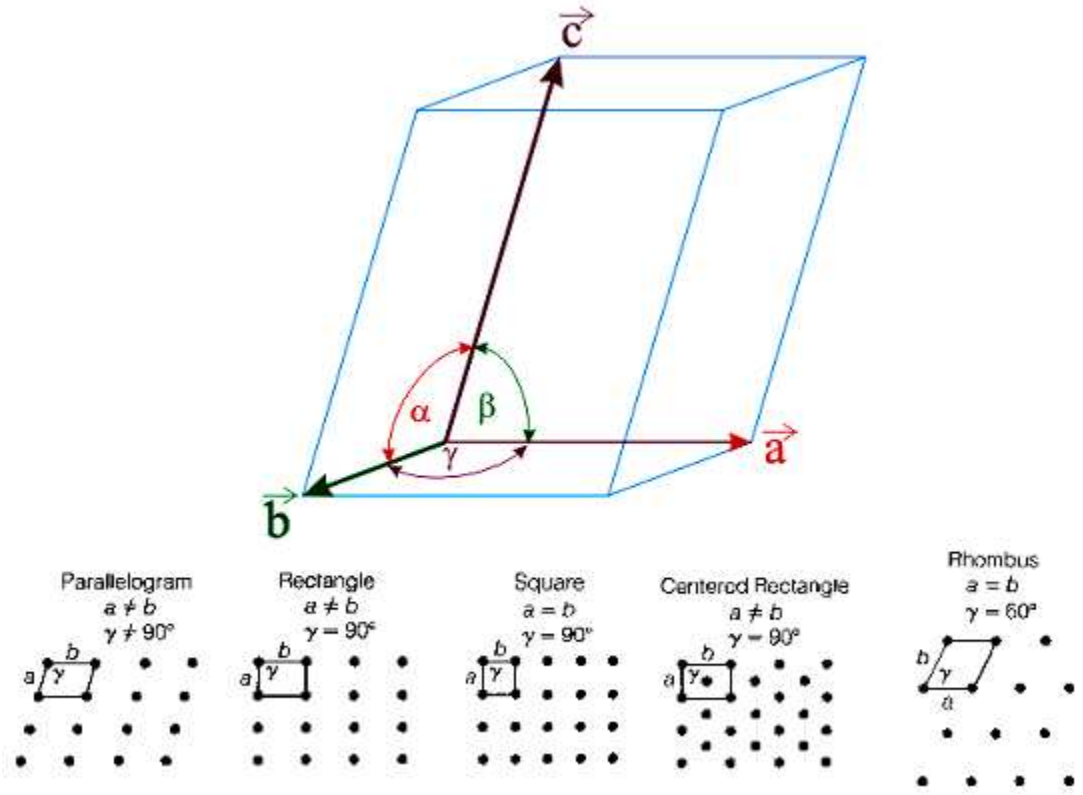

Figure 1-4. Up: 3-D crystallographic unit cell with axes a,b,c. Down: 2-D lattice described by crystallographic unit vectors a,b

Any point in a three dimensional lattice can be described using the crystallographic unit vectors as:

$$r = p\vec{a} + q\vec{b} + r\vec{c} \tag{1.1}$$

When someone performs an operation (such as a rotation) on a geometrical object and this operation results in an object that cannot be distinguished from the original disposition, then this operation is called a symmetry operation. The geometrical locus which remains unchanged by such an operation comprises a symmetry element. Crystal structures may be characterized by a set of symmetry elements which is an important property of well- ordered, geometrical objects. The symmetry of the lattices is such that in two dimensions only five lattices exist, while in three dimensions fourteen lattices are defined, known as Bravais lattices. These lattices are called plane and space lattices respectively. Symmetry operations which do not involve a translation may be regarded as

acting on a single point. In two dimensions when we consider all the possible combinations of symmetry operations which do not involve a translation, then this produces 10 point groups. When symmetry operations that include a translation are also regarded we get 17 space groups. In three dimensions combinations that do not involve a translation result in 32 point groups or crystal classes. When translation is involved Symmetry elements can be combined in groups and it can be shown that 230 distinctive arrangements are possible. ([1]p.89-91) Each of these arrangements is called a space group and they are all listed and described in volume A of the International Tables for Crystallography.[7] More on symmetry elements, groups and classes can be found in [1] p.59-91, [4] p. 33-68 and p.105-138 and in [8] p.1-31

1.1.3 X-rays are electromagnetic waves

The fact that in the present day we consider the nature of light as a wave motion is the result of the work of Christiaan Huygens who considered light to be a wave disturbance, in contrast to Isaac Newton who claimed that light is a stream of particles. ([4] p.74) Visible light and x-rays are actually electromagnetic waves, where an electric field **E** and a magnetic field **H** oscillate in a wave-like form in two mutually perpendicular planes (**E, H** are not matrices according to the notation, in this situation, but vectors). The difference between light and x-rays is the range of their wavelengths (x-rays have much shorter wavelengths). The nature of an electromagnetic wave is shown in Fig. 1-5.

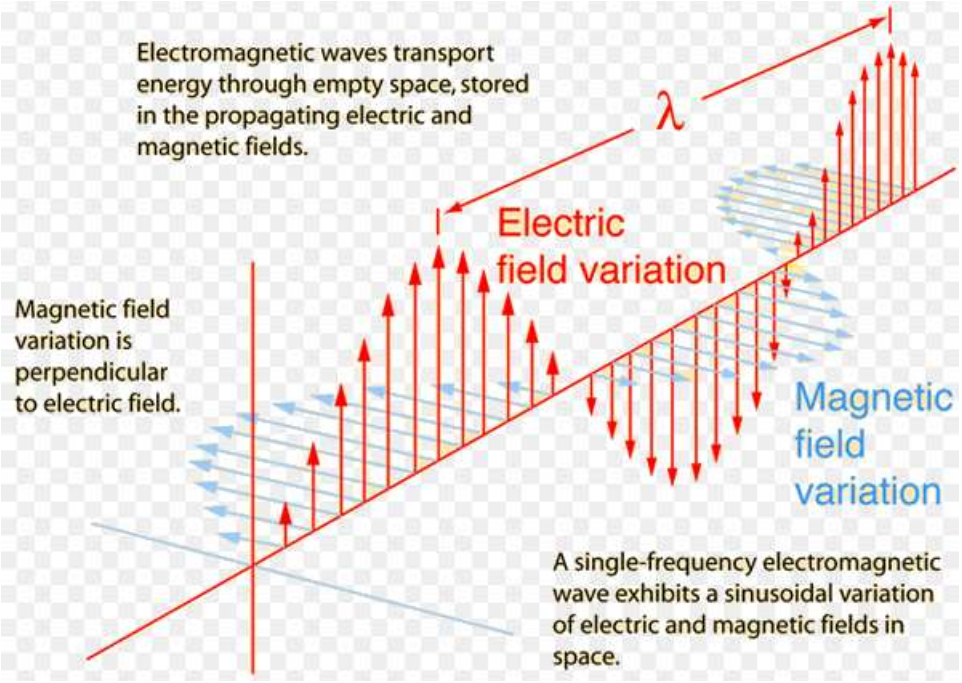

Figure 1-5. Nature of electromagnetic waves

Each of the fields which constitute the electromagnetic wave can be described with wave equations. A simple (plane) wave such as a sine wave (or sinusoid) can be expressed with the equation:

$$\psi(\vec{r},t) = \psi_0 e^{i(\vec{k}\,\vec{r}-\omega t)} \tag{1.2}$$

Where ψ(**r**,t) is an amplitude function (for example of the electric field) with respect to a point in space **r**(x,y,z), in a given time t and ψ_0 is the amplitude maximum. Vector **k,** is a vector with the same direction as that of the propagation of the wave and has a magnitude equal to 2π/λ where λ is the wavelength. The term ω is equal to 2πf, where f is the frequency of the wave.

Waves follow the principle of superposition which states that a total wave disturbance due to an array of sources is the sum of the individual disturbances due to each source.([1]p.133) This principle can be elucidated, if we consider the general equation for (plane) waves:

$$\psi(\vec{r},t) = \sum_n \psi_n e^{i(\vec{k}\,\vec{r}-\omega_n t+\varphi_n)} \tag{1.3}$$

This equation shows that a complex wave can be analyzed in a sum of simpler waves. It also shows that, in order to find the total disturbance, the relative phases of the waves are necessary, as mentioned in paragraph 1.1. The term in parenthesis at the exponential represents the phase of a wave, and the term φ_n in particular is the phase difference between the current wave and a wave considered as origin [such as the wave in equation (1.2)].

The intensity of a wave is the square magnitude of the wave amplitude,

$$I = |\psi(\vec{r},t)|^2 \tag{1.4}$$

A more detailed analysis of waves is given in [1] p.93-133 and [25].

1.1.4 Diffraction, Fourier Transformations and Reciprocal lattices.

When an electromagnetic wave passes through matter, the electric field causes charged particles such as protons and electrons to oscillate. When a particle oscillates it creates wave-like disturbances in the electric field and this result in the particles becoming secondary sources of electromagnetic radiation. This effect is known as scattering, and it is the main phenomenon that occurs when x-rays pass through materials. On a larger scale diffraction is a phenomenon which results from the scattering of x-rays in crystals. Basically these two phenomena are different sides of the same coin. We talk about scattering when the wave-obstacle interaction is such that the dimensions of the obstacle and the wavelength of the wave motion are comparable. On the other hand, we talk about diffraction when the wave-obstacle interaction is such that the dimensions of the obstacle are much larger than that of the wavelength of the wave motion. In other words, a combination of scattering events gives rise to the macroscopic event of diffraction. ([1] p185-186)

X-ray scattering is mainly a result of electrons. That happens because x-rays have high frequencies and protons are much heavier than electrons. Therefore electrons scatter more efficiently x-rays to the point where we consider that the total diffraction pattern due to x-rays is essentially a product of electron scattering only. ([1] p133)

When someone performs an X-ray experiment his purpose is to determine the intensities of the scattered waves. From these intensities with various mathematical calculations which simulate the action of a lens, a crystallographer tries to define an electron density map of the crystal. The electron density map of a crystal is a three-dimensional description of the electron density in a crystal structure. This map describes the contents of the unit cells averaged over the whole crystal and not the contents of a single unit cell (this is important where structural disorder is present). Three-dimensional maps are often evaluated as parallel two-dimensional contoured sections at different heights in the unit cell. The electron density, is the concentration of electrons per unit volume, expressed as electrons per $Å^3$ and as a function of position x,y,z, is the Fourier transform of the structure factors (which are discussed later). [E-1] After the construction of this map, atomic nuclei positions may be associated with peaks of the electron density and the more dense the peak is, the higher the atomic number of the nucleus is at that site.([1]p.133)

As mentioned earlier whenever a wave motion, such as x-rays, interacts with an obstacle, such as a crystal, diffraction occurs. The diffracted waves form a pattern which is determined by the nature and structure of the obstacle. This pattern of radiation scattered by the object is called its diffraction pattern. At this point an example with light can help the reader understand the nature of diffraction and how can someone obtain information

from a diffraction pattern. We know from experience that, when a beam of light falls on an object which is large compared to the wavelength of light, relatively sharply defined shadows are cast, because light is considered to travel in straight lines. However, when the wavelength of the light and the size of the object are of the same order of magnitude, an amount of light spreads into the area expected to be in shadow. This effect can be observed when light passes through a narrow slit or from the fringes of parallel dark and bright bands that are produced when light passes through two narrow slits. These fringes constitute the diffraction pattern of these two narrow slits. In diffraction the straight-line path of a wave front travelling through a uniform medium is caused to change direction. This is a different phenomenon from refraction in which the bending of light occurs when the nature of the medium changes. The diffraction pattern contains information on the structure of the diffracting obstacle. ([4] p.75-77, [1] p.220-224) It can be shown ([1] p.185-220) that if an obstacle can be described by an amplitude function f(**r**), where **r** is a vector in one (in one dimension r is a scalar), two or three-dimensional space and f(**r**) is zero everywhere outside the boundaries of the obstacle, then the diffraction pattern amplitude F(**k**) is given by the Fourier transform of f(**r**):

$$F(\vec{k}) = \int_{all\ \vec{r}} f(\vec{r}) e^{i\vec{k}\,\vec{r}}\ d\vec{r} \tag{1.5}$$

Where **k** is defined as in equation (1.3) for the scattered waves and the integral is double if it refers to two dimensions and triple if it refers to three dimensions. This equation gives us the diffraction pattern if the obstacle (f(**r**)), is known.

Conversely, if the diffraction pattern is known, then using the Fourier inversion theorem, we can infer the amplitude function f(**r**) (hence, the description of the object) in terms of F(**k**) as follows:

$$f(\vec{r}) = \int_{all\ \vec{k}} F(\vec{k}) e^{-i\,\vec{k}\,\vec{r}}\ d\vec{k} \tag{1.6}$$

Fourier transforms are mathematical expressions which serve as tools for calculations in many different branches of science (such as engineering and signal processing). There are several common conventions for defining the Fourier transform. One of these is:

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x) e^{ikx} dx \tag{1.7}$$

Basically Fourier transforms enable us to express a function originally expressed in terms of a variable x, in terms of another variable k (in signal processing for example it can switch from time expressions of a signal to frequency expressions) and has some very interesting and appealing properties such as the easy calculation of convolutions. If someone wants to switch back to the original variable, he can use the inverse Fourier transform:

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} F(k) e^{-ikx} dk \quad (1.8)$$

The difference between expressions (1.7), (1.8) and (1.5), (1.6) is the fraction outside the integral, which was omitted for simplification. Fourier transforms are generally complex numbers by nature. Some interesting properties of these transforms are:

- The more extended a function f(x) is, the narrowest its Fourier transform is as a function of k.
- The narrowest a function f(x) is, the wider its Fourier transform is as a function of k.
- The Fourier transform of a non-symmetrical function is complex (it contains a real and imaginary part)
- The Fourier transform of an anti-symmetrical function (f(x) = -f(x)) is imaginary.
- The Fourier transform of a symmetrical function (f(x) = f(-x)) is real. ([1] p.135-183, [8] p.85-88)

Just as visible light can be diffracted by small objects, x-rays are diffracted by electrons. The effect is about the same and only the scale differs. The diffracted waves have the same wavelength and frequency, in general, as the incident waves but are propagated in all directions in space. That happens because the oscillation of the electrons that produces those secondary waves, have the same frequency as the electromagnetic wave that causes them to oscillate. This phenomenon is referred to as coherent scattering. ([2] p.81) The diffraction patterns observed are the result of the interference between diffracted waves. Interference between waves occurs when they are travelling in the same direction. If these waves have the same wavelength, as in the case of x-ray scattering, they express the effects of constructive or destructive interference as discussed in section 1.1. The amplitude of the wave that results from the interference of two (or more) waves with the same wavelength is determined by their path difference which is the fraction of a wavelength one wave is out of phase with another. The extent of this path difference (which results in phase difference, see Fig.1-1) depends on the angular deviation of the direction of the diffracted beam from that of the direct beam and on the wavelength of radiation. At large angles the diffracted beams are out of phase and that results in destructive interference. Thus the diffracted beam in these angles is weak. One wave is more out of phase with its 'neighbor', the larger this angle is and the shorter the wavelength of radiation. At angles where the diffracted waves are in phase the diffracted beam is strong while at other angles the diffracted waves may cancel each other (π phase difference) and no diffracted beam is observed. In Fig. 1-6 the diffraction of light from a single slit is shown. Since a single slit has width, waves travelling in the same direction from the two edges will interfere with each other in the manner mentioned above. The phase difference between those waves is greater as the width of the slit increases. Thus the diffraction beam will be narrower. As someone can notice in Fig.1-6 the intensity

variation is bell-shaped. This shape is called envelope profile and in inversely proportional to the width of the slit. This is in agreement with equation (1.5) because we mentioned that the more extended a function of the object is (wider slit), the narrowest its Fourier transform (and diffraction pattern) is. ([2] p.74-80)

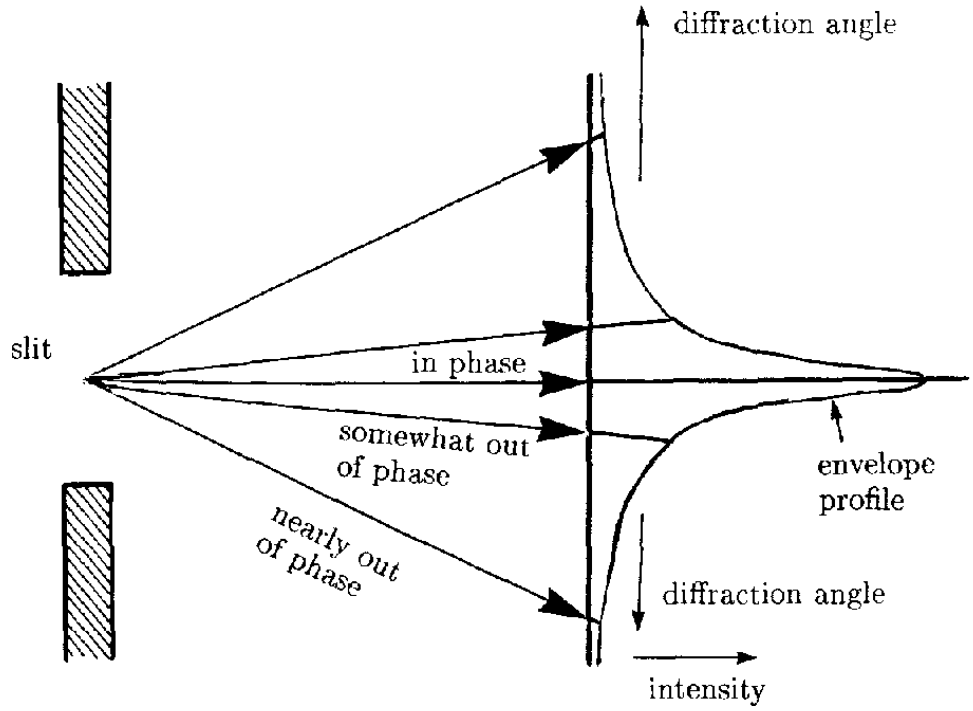

Figure 1-6. Explanation of the diffraction pattern of a single slit. At higher angles the intensity of the diffracted beam is weak.

Up to now, the diffraction pattern of a single slit has been considered. The diffraction pattern of a crystal is basically the diffraction pattern of a well-defined array. In this array there are molecules at well defined distances between them, that act as scattering centers. The diffracted waves from these scattering centers are combined and give a precise and clear diffraction pattern. This is not the case for example, for a liquid, whose diffraction pattern is rather diffuse and imprecise because of the random position of the molecules. The diffraction pattern of a regular array retains intensity only at isolated regularly spaced positions. These positions can be considered as sampling regions. As an example let's consider a one-dimensional array, an array of slits, at equal distances, with the same width. Their diffraction pattern can be considered as a composition of two elements. These elements are depicted in Fig. 1-7. The first element consists of the sampling regions mentioned above which result from the interference of waves scattered from equivalent points in different slits. The locations of these regions are connected through a reciprocal relationship with the spacing between the slits, which could be roughly considered as a crystal lattice of a one-dimensional crystal. The second element is the

envelope profile which has the shape of a bell and results from the interference of waves scattered by an individual slit. ([2] p.80, [1] chapter 7)

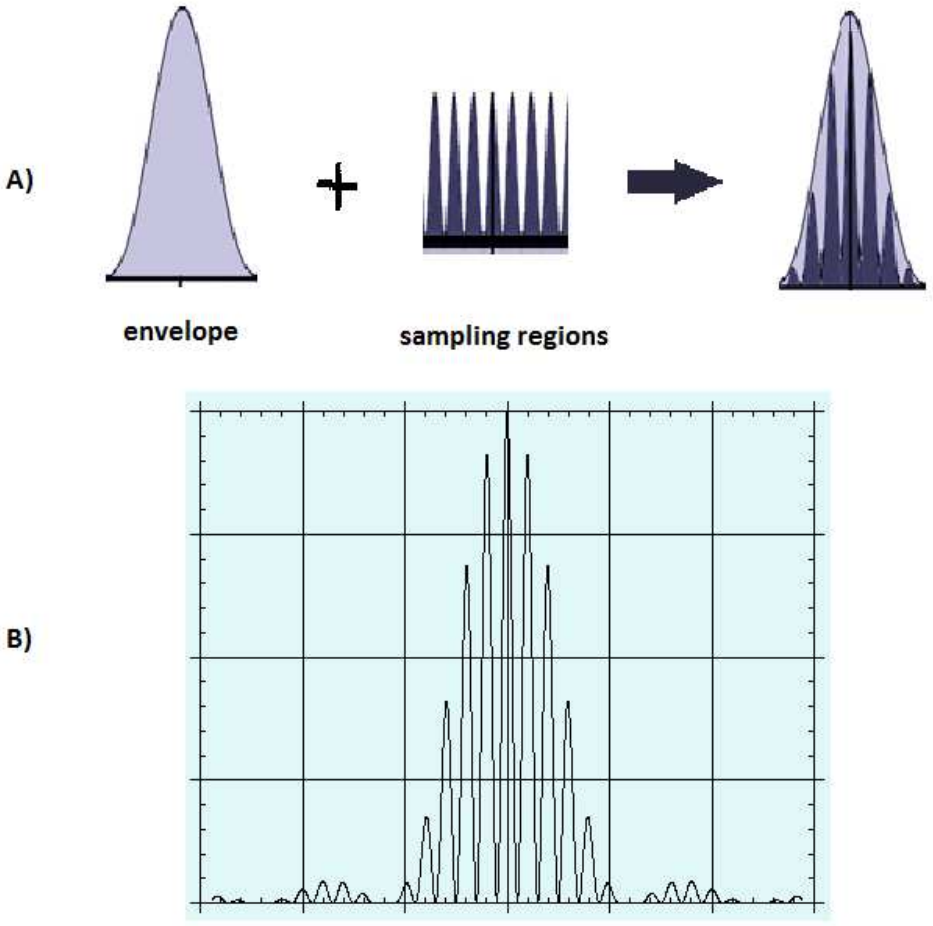

Figure 1-7. Elements of diffraction pattern. A) The envelope is wider for narrow slits, and vice versa. The sampling regions are denser for larger spacing between slits and sparser for small spacing between slits. B) The diffraction pattern (intensities of beams) of N narrow slits.

From these patterns some rules can be inferred about the way in which they give information of the structure of a crystal like obstacle. ([1] p.250)

- The position of the main peaks can give information on the lattice of a crystal.
- The shape of each peak gives information on the overall object shape (e.g. the size of the crystal)
- The set of intensities of all the main peaks gives information on the structure of the motif (unit cell)

The positions of the sampling regions, where the intensities of the diffracted beams reach their peaks (the intensity of each peak is determined by the envelope), constitute the 'reciprocal lattice' of the array of slits, if this array is considered as a one-dimension crystal lattice. This result can be extended in two and three dimension crystal lattices

from which someone can derive their reciprocal lattices which represent the sampling points. The term 'reciprocal', can be justified from the fact that the distances between points in the original crystal lattice and the distances between points in the reciprocal lattice, which is connected to the diffraction pattern, are inversely related. ([2] p.89-97) The reciprocal lattice, can be described in terms of three base vectors **a***,**b***,**c*** in a similar way the crystal lattice is described with crystallographic unit vectors in equation (1.1). The reciprocal lattice base vectors **a***,**b***,**c*** are defined by specific equations in terms of the crystallographic unit vectors **a**,**b**,**c**. (see [1] p.274) When vectors **a***,**b***,**c*** represent a primitive reciprocal lattice we can define a vector **G** which represents a vector from the origin to any point in this lattice, such as:

$$\vec{G} = h\,\overrightarrow{a^*} + k\,\overrightarrow{b^*} + l\,\overrightarrow{c^*} \tag{1.9}$$

Vector **G** defines a reciprocal lattice vector. If the reciprocal lattice is primitive then h,k,l in (1.9) are integers only. A lattice is primitive when in contains primitive cells. A primitive cell contains only one lattice point (each vertex of the cell sits on a lattice point which is shared with the surrounding cells, each lattice point is said to contribute 1/n to the total number of lattice points in the cell where n is the number of cells sharing the lattice point). [E-3] Conventionally, reciprocal lattice points are represented with these three integers h,k,l written as a triplet hkl with neither parentheses or commas. If any of the h,k,l is negative this is indicated by a superscript bar. We refer to this lattice points such as 100, 231 etc and the corresponding reciprocal lattice vectors are written as $\mathbf{G}_{100,}$ $\mathbf{G}_{231}$. ([1] p.275)

In Fig. 1-8, the scattering vector of a diffraction event is presented. The vector $\mathbf{k_i}$ is the wave vector of the incident beam of x-rays to the crystal (direction of $\mathbf{k_i}$, is parallel to the propagation of the incident beam). Vector $\mathbf{k_d}$ represents the wave vector of the diffracted beam. Both $\mathbf{k_i}$ and $\mathbf{k_d}$ have the same magnitude. $\Delta\mathbf{k} = \mathbf{k_d} - \mathbf{k_i}$ represents a vector describing the change in direction which has occurred as a result of the diffraction and is called the scattering vector. [E-2] Angle 2θ is called the scattering angle, while angle θ is called Bragg angle.

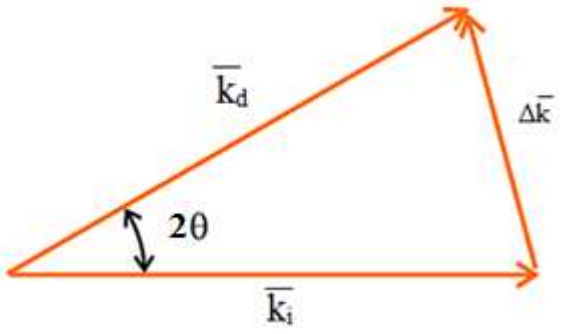

Figure 1-8. Scattering vector Δk and scattering angle θ.

Three dimensional diffraction patterns are usually described in terms of the scattering vector $\Delta\mathbf{k}$. When the reciprocal lattice vector **G** is multiplied by a scalar 2π, is an allowed solution of Laue equations, and so represents a scattering vector corresponding to a diffraction maximum. Laue equations derive from the realization that the path length differences for waves diffracted by atoms separated by one crystal lattice translation must be an integral number of wavelengths for diffraction to occur.([2] p.81) This condition must be true for all three dimensions at the same time for the three crystal lattice translations and is expressed by Laue equations:

$$\Delta\vec{k}\,\vec{\alpha} = 2h\pi\ , \quad \Delta\vec{k}\,\vec{b} = 2k\pi\ , \quad \Delta\vec{k}\,\vec{c} = 2l\pi \tag{1.10}$$

which have to be satisfied simultaneously (h,k,l are the same as in equation (1.9)). A more mathematical approach on Laue equations can be found in ([1] p.260-272).

Exactly how the reciprocal lattice corresponds to the actual directions in which waves are diffracted from a crystal, and the manner in which is related to the array of spots on an x-ray diffraction photograph can be explained with the help of the Ewald sphere. Ewald sphere or sphere of reflection is a geometric construct which shows the relationship between the wave vectors of the incident and diffracted beams, the diffraction angle for a given reflection, and the reciprocal lattice of the crystal. [26] Ewald sphere can be considered one of the geometrical interpretations of Bragg's law:

$$n\lambda = 2d_{hkl}\,sin\theta_{hkl} \tag{1.11}$$

where λ is the wavelength, θ is one half of the scattering angle for this particular point in the diffraction pattern (which refers to the hkl point in reciprocal lattice) and d_{hkl} is the perpendicular spacing between the (hkl) set of crystal lattice planes. Crystal lattice planes are planes in the crystal lattice that are rich in lattice points. ([2] p. 83-84) Parallel sets of planes in a crystal lattice may be identified by a set of three integers (hkl), called the Miller indices, which have no common factor. The reciprocal lattice vector $\mathbf{G}_{hkl}$ in equation (1.9) is perpendicular to the (hkl) set of planes in the real lattice. ([1] p.283-286). Bragg tells us that in a diffraction experiment, we measure the intensities of waves scattered from planes (denoted by hkl) in the crystal. In equation (1.11) n is an integer, the order of reflection. This is an integer associated with a given interference fringe of a diffraction pattern. The first order arises as a result of a path difference, between diffracted waves, of one wavelength. Nth order represents a path difference of n wavelengths. Bragg's law and Laue equations are equivalent and describe the same phenomenon in different ways. ([2] p. 81- 87)

1.1.5 Structure factors and electron density maps.

Until now the effects of the periodic nature of a crystal to its diffraction pattern has been discussed. It has been mentioned earlier in the previous paragraph, that the information about the contents of the unit cell in a crystal, is contained in the set of intensities of all the main peaks (envelope effect). The diffraction of x-rays by the contents of the unit cell is determined by Thomson (or coherent) scattering by the electrons. Since the unit cell is associated with the lattice and the diffraction pattern is sampled at specific regions (the reciprocal lattice points hkl) the amplitude of the diffraction pattern is associated with particular lattice points. These specific regions of the diffraction pattern at the reciprocal lattice points are associated with terms called Structure factors, which are given by:

$$F_{hkl} = V \iiint_0^1 \rho(x,y,z)\, e^{2\pi i\,(hx+ky+lz)}\; dx\, dy\, dz \qquad (1.12)$$

where V represents the volume of the unit cell and ρ(x,y,z) is the electron density map as a function of position x,y,z (from now on referred as electron density map).

Someone can notice that the set of structure factors in equation (1.12) is in fact the Fourier transform of the electron density map ρ(x,y,z) and that structure factors are generally complex quantities. Therefore if a crystallographer can determine these structure factors from an x-ray diffraction experiment, he is able to calculate the electron density map through an inverse Fourier transform. Since this inverse transform does not refer to a continuous function but to a discrete function the integrals in the inverse Fourier transform are converted to sums. The electron density map therefore is given by:

$$\rho(x,y,z) = \frac{1}{V} \sum_h \sum_k \sum_l F_{hkl}\; e^{-2\pi\imath\,(hx+ky+lz)} \qquad (1.13)$$

The calculation of the electron density map using structure factors is called Fourier analysis and it is the ultimate purpose of a crystallographer who performs a crystal structure analysis. ([1] p.320-362)

Electron density maps ρ(x,y,z) in many occasions are plotted as two-dimensional contours of x,y, at regular intervals of z and they are combined to form a final three dimensional contour map. From this contour map nuclear positions and molecular structure information can be derived. (Figures 1-9, 1-10)

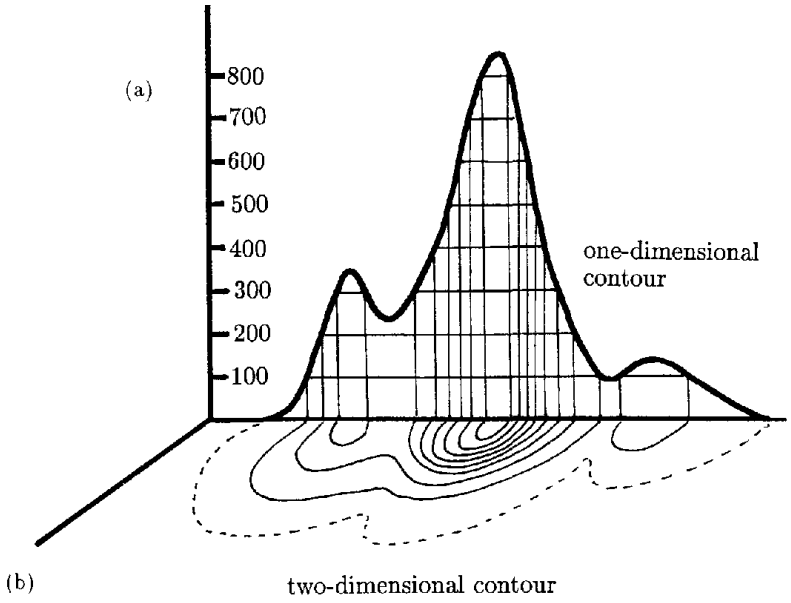

Figure 1-9. Numerical values of the calculated electron density (a) at grid points and (b) at a two-dimensional contour plot, showing how contours are drawn in two dimensions

.

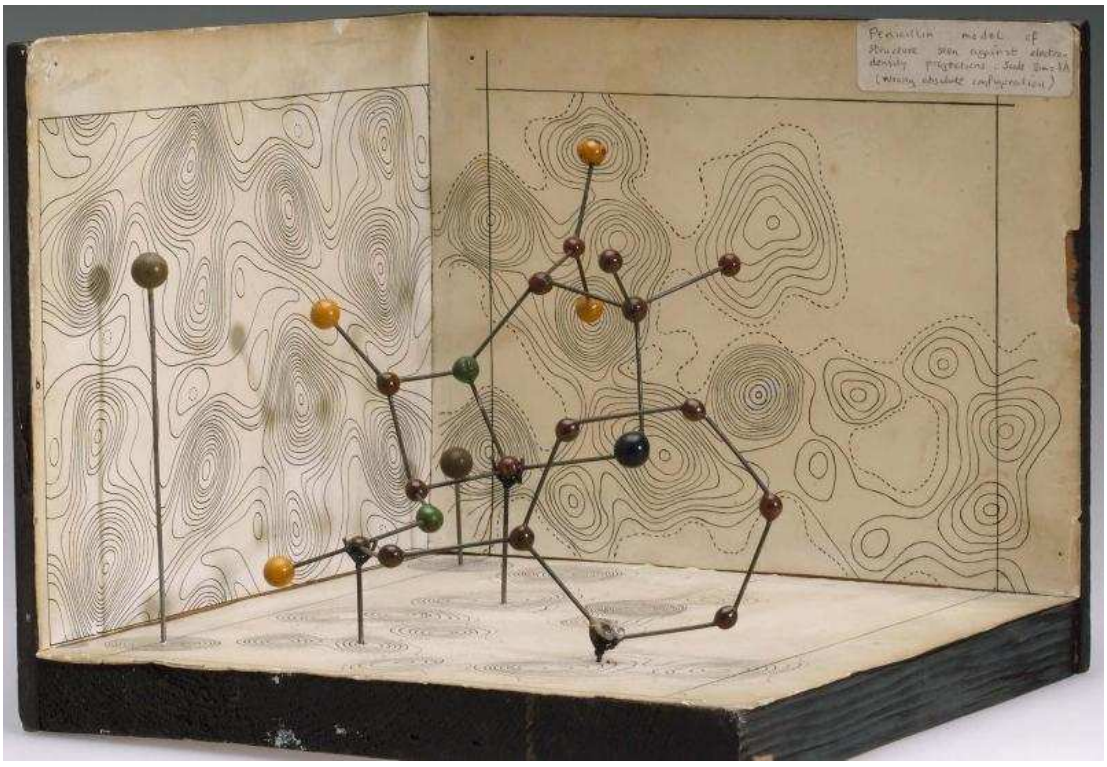

Figure 1-10. Electron density map and model of Penicillin created by Dorothy Crowfoot Hodgkin in 1945 based on her work on X-ray crystallography

The form of equation (1.12) in not the most appropriate for the calculation of the structure factors from the contents of the unit cell. A more appropriate equation for this calculation can be derived. If the jth atom in a molecular structure has coordinates (x_j, y_j, z_j) the equation for the structure factors can be defined as:

$$F_{hkl} = \sum_j f_j e^{2\pi i\,(hx_j+ky_j+lz_j)} \tag{1.14}$$

where f_j is the atomic scattering factor. ([1] p.320-362)

Since x-rays are scattered by electrons, the amplitude of the scattered beam by an atom is proportional to the number of electrons there are around that atom. That number is its atomic number Z. Atomic scattering factors f_j are expressed as the ratio of the scattering of an atom to the scattering by a single electron under the same conditions. ([2] p.87-89) Values of atomic scattering factors are influenced by the scattering angle and as the scattering angle increases the value of the scattering factors falls. This is a result of the size of the atoms. Atoms in reality are not just ideal points, and this causes some destructive interference between waves scattered by various regions of the atoms. If the electron cloud around the atom is wide the falloff in the intensity of the diffracted beams is greater at higher scattering angles. A zero value of f_j for a particular scattering angle means that the incident beam is not deflected at all in that direction. Values of scattering factors can be derived theoretically and are listed for individual atoms as a function of the scattering angle in International Tables for X-ray crystallography. Generally f_j can be considered as a real quantity, if anomalous scattering is not present. If an atom shows anomalous scattering then f_j is a complex quantity. Anomalous scattering happens when the energy of the incident beam is close to that which will change the quantum state of an electron within an atom. In this occasion scattering does not obey the rules (the equation) of Thomson scattering.

In this discussion, for the derivation of the diffraction pattern of the structure of the unit cell, we silently assumed that the atoms are located at fixed points in space. The assumption that the nuclei are fixed at single points in the lattice, and stay motionless there, is true only at temperatures very close to zero. At non-zero temperatures, and as the temperature increases, the thermal motion of the atoms causes a variation in the exact real lattice structure. With proper calculations we can take this effect into account and correct the atomic scattering factors for thermal vibrations. ([1] p.377-383, [2] p.272-273) Corrected scattering factors are given by:

$$(f_j)_T = f_j\, e^{-B_j(\frac{sin^2\theta}{\lambda^2})} \tag{1.15}$$

B_j is the temperature factor, and its value depends on the atom under study and the temperature at which the diffraction takes place. B_j is a positive quantity and its value for

atoms at various temperatures can be found in Vol.III of the International Tables for X-ray Crystallography. The factor ($\sin^2\theta/\lambda^2$) is also positive.

Figure 1-11 shows the relationships between crystal lattices, reciprocal lattices, structure factors, and contents of the unit cell discussed so far.

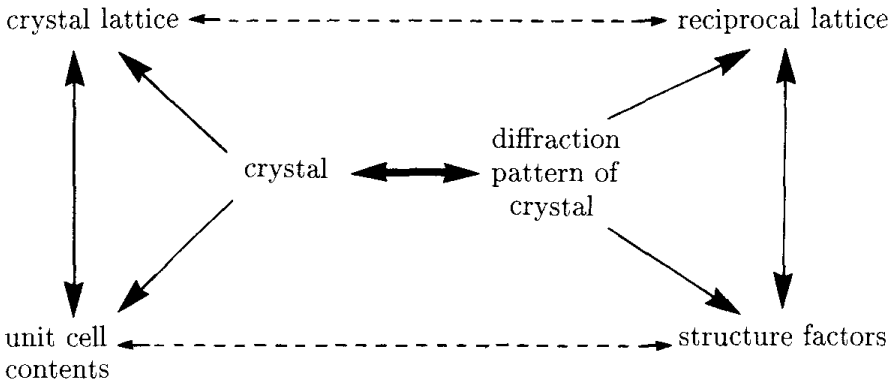

Figure 1-11. relationships between crystal lattices, reciprocal lattices, structure factors, and contents of the unit cell

1.1.6 The phase problem

The aim of a crystallographer, who performs an x-ray diffraction experiment, is to determine the molecular structure of a crystal by calculating the electron density function according to the Fourier synthesis described by equation (1.13). In order to calculate this equation we need the values of all the structure factors. Structure factors as stated in paragraph 1.5 are generally complex quantities. That means that they are composed of two elements, just as any complex number: their magnitude and phase. This is shown in the following equation:

$$F_{hkl} = |F_{hkl}|\, e^{i\,\alpha_{hkl}} \tag{1.16}$$

where α_{hkl} represents the phase of the structure factor. Therefore the calculation of the electron density map requires the knowledge of all structure factors in both magnitude and phase. In x-ray diffraction experiments, however, only the intensities of the reflections are measured, and information on the relative phases is lost because the intensity of a structure factor is equal to $|F_{hkl}|^2$. The phases cannot be obtained directly from physical measurements. Thus half the information is lost during this procedure. This problem can be further illustrated in an Argand diagram (Fig.1-12) in which complex numbers are represented as points. In order for a simple point to be represented, both the magnitude and the angle have to be defined. The magnitude alone defines only the radius

of a circle. If only the magnitude $|F_{hkl}|$ is known then the structure factor can have any value for the angle α_{hkl} from 0 to 2π, and can be any point in the circle defined by the dashed red line. ([1] p.411-416)

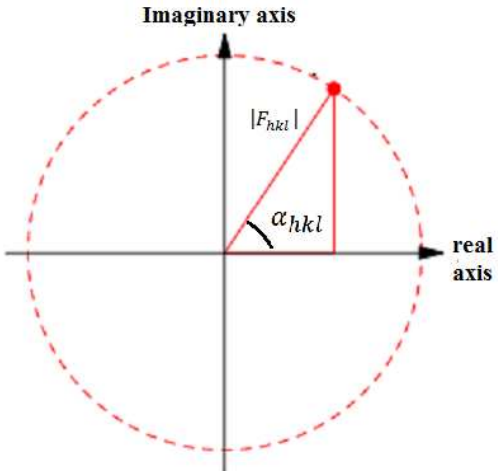

Figure 1-12. Argand diagram for a structure factor, with determined magnitude and unknown phase.

The derivation of the correct values of the structure factors in both magnitude and phase from the relative intensities $|F_{hkl}|^2$ constitutes the Phase problem. A crystallographer must be able to reconstruct an electron density map in a systematic way, by approximating as far as possible, a correct set of phases for the structure factors. The importance of phases in obtaining the correct structural information can be illustrated in Fig.1-13(from Kevin Cowtan's Book of Fourier [E-4]) In this figure we can see that the calculation of an `electron-density map' using amplitudes from the diffraction pattern of a duck and phases from the diffraction pattern of a cat results in a cat.[27]

The phase problem is fundamental in crystallography. One important feature of this problem derives from the fact that, mathematically speaking the relationship between structure factor magnitudes and electron density in not necessarily a one-to-one relationship. The operation of going from an arbitrary electron density to structure factors amplitudes is unique, but this is not the case when we consider the opposite path. A set of structure factors does not necessarily correspond to one electron density function. In this sense as Taylor G. says in [27]: “There is no formal relationship between the amplitudes and phases; the only relationship is via the molecular structure or electron density. Therefore, if we can assume some prior knowledge of the electron density or structure, this can lead to values for the phases. This is the basis for all phasing methods”. The practical success of crystallographic methods lies in the fact, that in reality, electron density functions are not arbitrary distributions, but are connected with real objects, and are subject to certain restrictions. For example, electron density function is expected to be

real, positive, continuous and concentrated into spherical regions which represent atoms. These restrictions are severe and form the theoretical basis for direct methods. Also atoms should be located at places where they form reasonable molecules, in terms of angles and interatomic distances. It has been shown theoretically that the first restrictions are not sufficient to guarantee uniqueness of solution while the effects of the second are harder to judge. ([2] p265-267)

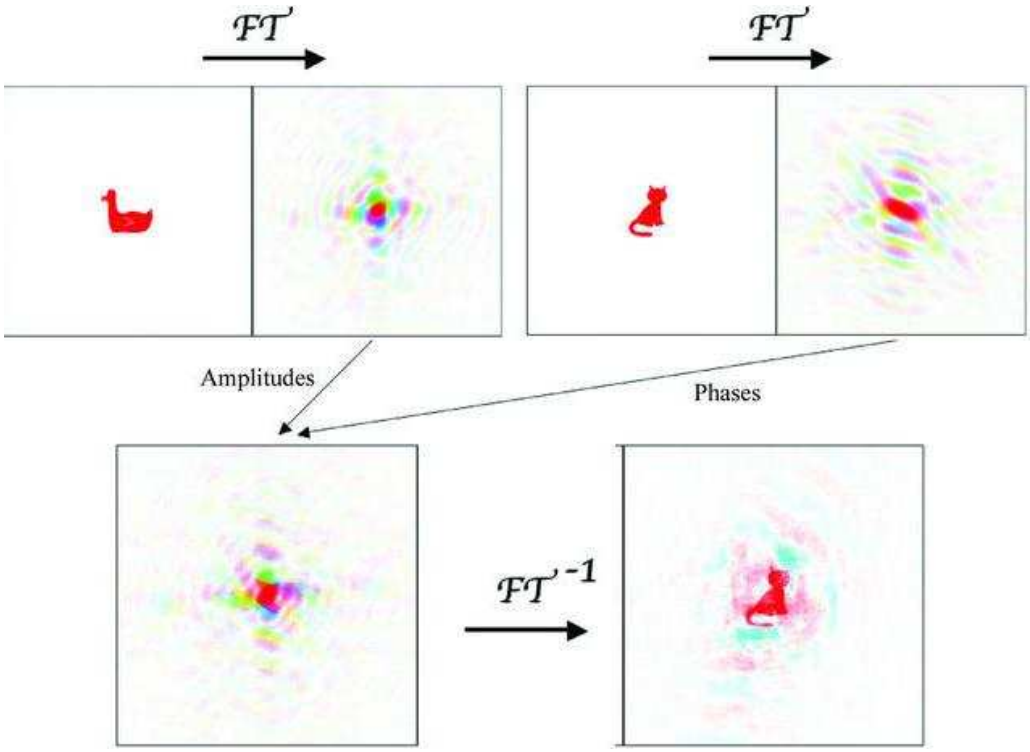

Figure 1-13. The importance of phases in carrying information. Top, the diffraction pattern, or Fourier transform (FT), of a duck and of a cat. Bottom left, a diffraction pattern derived by combining the amplitudes from the duck diffraction pattern with the phases from the cat diffraction pattern. Bottom right, the image that would give rise to this hybrid diffraction pattern. In the diffraction pattern, different colors show different phases and the brightness of the color indicates the amplitude.

Occasionally, various methods have been proposed to deduce the relative phase angles of the structure factors. "For many crystallographers, the very existence of the phase problem makes the structure solution of a crystalline substance an adventure to be tackled with chemical and physical intuition, imagination and mathematical expertise." ([5] p.1) Some commonly used ways to deal with the challenge of phase determination are described below.

Constraints of direct methods: These methods are based on the positivity and atomicity of electron density and leads through statistical methods to phase relationships between normalized structure factors. Direct methods will be discussed in more detail in chapter 2.

Interpretation of interatomic vectors: In these methods an initial trial structure is obtained through Patterson and heavy atom methods. From these structures calculated phase angles can be derived. (See [1] p.398-411 and p. 425-427, [2] p.270-299, [4] p. 301-317) These methods are very useful when the molecules contain heavy atoms, such as metal complexes, or when they have considerable symmetry.

Isostructural crystal comparison: The definition of isostructural or isomorphous crystals as given by [E-5] is: "Two crystals are said to be isomorphous if (a) both have the same space group and unit-cell dimensions and (b) the types and the positions of atoms in both are the same except for a replacement of one or more atoms in one structure with different types of atoms in the other (diadochy), such as heavy atoms, or the presence of one or more additional atoms in one of them (isomorphous addition)." Phases can be estimated by comparing such crystals and this method is the method of choice for macromolecular (such as protein) phase determination. ([4] p.284-285)

1.2 Introduction to Artificial Neural Networks.

Kröse B. states that: “Artificial neural networks can be most adequately characterized as computational models with particular properties such as the ability to adapt or learn, to generalize, or to cluster or organize data and which operation is based on parallel processing.” ([14] p.13) Artificial neural networks can be described as statistical learning models that were inspired by biological neural networks and were originally introduced as very simplified models of brain function. Biological neural networks constitute the central nervous systems of animals, and particularly the brain. Humans throughout the ages have always been concerned about the function of their brains. Questions about the production of our mental activities, and how the physical activities of our brain, gives rise to our thoughts emotions and behaviors, always intrigued scientists. The “father of neuroscience,” Santiago Ramon y Cajal, argued at the turn of the 20th century that the brain was made up of neurons woven together in a highly specific way.[29] Researchers since then explore new methodologies that shed light on this age-old mystery.

These questions, however, are not of special interest to neuroscientists and biologists only. Computer scientists are also attracted by features of human thought that would like to embed in their computational models. A computer usually works following the directions of an algorithm, which is described in a fixed program. That is one way to ‘teach’ a computer to do a certain task. Nonetheless there are certain problem categories that cannot be formulated as algorithms (e.g. the purchase price of a real estate). These problems usually depend on many subtle factors which our brains can calculate approximately while computers cannot, in a traditional manner.

One of the important differences between the function of human brains and computers, that help us tackle the kind of problems mentioned above, is that humans can learn while computers cannot. In a conventional computer, usually there exist a single processor implementing a sequence of arithmetic and logical operations, nowadays at speeds approaching billion operations per second, but they luck the adaptability of a brain and the ability to learn through examples. A computer is static, while a biological neural network can reorganize itself during its lifespan. A neural network does not need to be programmed, it can learn from training samples or by means of encouragement.

This feature, the capability to learn, results in the capability of networks to generalize and associate data. After successful training in a class of problems, a neural network can find solutions in problems of the same class. For example if someone in math class learns through examples how to solve certain problems, then he can solve problems, similar but not the same to the problems in the examples used. Another example connected to

computer vision and pattern recognition problems is our ability to separate trees from bushes, although we have not seen these exact bushes and trees before.

This capability of generalization in turn results in a degree of fault tolerance. Fault tolerance refers to the ability of a system to produce reasonable responses when it is provided with noisy input data, or when the system has internal errors (e.g. when damaged). Someone can understand that when he considers the fact that human brains have a vast amount of interconnected neurons, which reorganized themselves but are also influenced by external factors. External factors such as alcohol, certain health conditions, drugs, environmental influences etc can destroy neurons, yet our cognitive abilities are not significantly affected (when the damage is not too extended). ([16] p. 3-8) A demonstration of our ability to interpret noisy data is the fact that most humans can read jumbled words, for example the following sentence: "I cnduo't bvleiee taht I culod aulaclty uesdtannrd waht I was rdnaieg." (Translation: I couldn't believe that I could actually understand what I was reading.)

Another important feature of neural networks is their parallelism. The largest part of the brain is working continuously with its highly interconnected simple units (neurons) working in parallel. That enables the brain to achieve high performances (as a matter of time). Today's computer processors have a speed 106 times faster than the basic processing elements of the brain. When the abilities are of processors and neurons are compared, the neurons are much simpler. The difference between a computer and a neural network lies in the structural and operational trends. While in a conventional computer the instructions are executed sequentially in a complicated and fast processor, the brain is a massively parallel interconnection of relatively simple and slow processing elements. ([10] chapter 1) Artificial neural networks are also inherently parallel algorithms and can take advantage of multicore CPUs (central processing units) and clusters of computers arranged to make parallel calculations, in order to reduce computation times.

For these reasons artificial neural networks can help scientists approach problems in an alternative way, and actually 'teach' computers to solve problems with the use of learning paradigms or reinforcement learning. Today artificial neural networks are being used in many areas because of their appealing features and capabilities. Google uses neural networks for image tagging (automatically identifying an image and assigning keywords), while Microsoft has developed neural networks that can help convert spoken English speech into spoken Chinese speech.

Despite the advantages and effectiveness of neural networks in many areas, they tend to consume considerable amounts of time and money. From a more practical side, the implementation of large and effective software neural networks (for difficult problems), demands considerable processing and storage resources. Biological neural networks are

especially made to work through processing of signals in neurons highly interconnected, and by propagating signals from one neuron to another in a most efficient way. However, the simulation of this procedure in Von Neumann architectures is not always effective in terms of computer memory, hard disk space and processing power, because of the amount of interconnections between neurons.

Nevertheless advances in computer technology have limited these restrictions. Faster computers and faster algorithms made possible the usage of neural networks to solve complex industrial problems that required too much computation. The number of neural network applications, the money that has been invested in neural network software and hardware, and the depth and breadth of interest in these devices is enormous. The applications of artificial neural networks are expanding in many scientific fields because neural networks are capable of solving problems in fields such as engineering, science and mathematics, medicine, business, finance and literature. Specifically medical applications of neural networks include Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement and emergency room test advisement. ([12] chapter 1, p.1-5)

1.2.1 Simplified Biological Neural Networks

Artificial neural networks appeared as simplified models of biological neural models and particularly as models of the brain function. In this section a simplified model of a biological neural network is presented in order to consider the analogies between artificial and biological networks. The network discussed in this section is only remotely related to its biological counterpart.

An adult male human brain contains on average 86.1 +/- 8.1 billion neurons which are highly interconnected. [30] Neurons are cells with specialized membranes which allow the transmission of signals to neighboring neurons. In Fig.1-14 the structure of a neuron is illustrated. These neurons, for the purposes of this analysis have three basic components: the dendrites, the cell body and the axon. The dendrites are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body. The axon extends from the basic cell body and constitutes the output of a neuron. Axon typically divides into sub-branches and terminates at a synapse. Synapse is called the point of contact between an axon of one cell and a dendrite of another cell. The axon serves as an electrical pulse transmitter from the cell body to the synapses and it does so by transferring Na+ ions. The arrival of a voltage pulse through the axon to a synapse stimulates the release of neurotransmitting chemicals across the synaptic cleft towards the postsynaptic cell, which is the receiving part of the next neuron. This postsynaptic cell passes the signal via the dendrite to the main part of the (next) neuron body. The signals

which arrive to the cell body of a neuron through dendrites constitute its input. The cell body then sums the inputs (the incoming signals), and if this sum has sufficient strength to overcome a certain threshold, an output signal is produced and delivered through the axon to the synapses and the neurons that follow. The output signal from the neuron is some nonlinear function of the input stimuli. It is the arrangement of neurons and the strengths of the individual synapses, determined by a complex chemical process that establishes the function of the neural network. ([13] p.1-2)

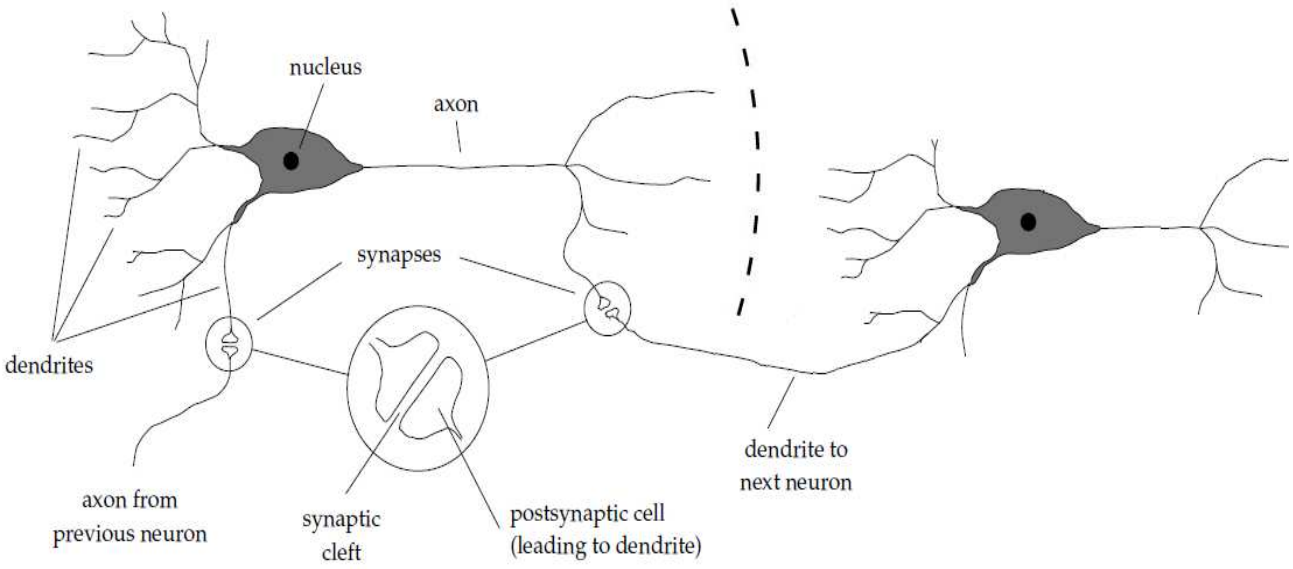

Figure 1-14. Simplified structure of a neuron and connection between two neurons

Each neuron has between a few and a few thousand synapses on its dendrites, giving a total of about 10^{14} synapses (connections) in the brain. Part of the structure of the neural network is defined at birth while other parts are developed later through learning processes, as new connections are made, others are strengthened and others are weakened. The 'strength' of the synaptic connection between neurons can be chemically altered by the brain in response to favorable and unfavorable stimuli, in such a way as to adapt the organism to function optimally within its environment. The process of learning a new face is actually a process of altering various synapses in the neural network structure. ([12] chapter 1, p.1-9) This is the reason why synapses are believed to be the key to learning in biological systems.

As stated earlier this is an oversimplified model, used to demonstrate the similarities between biological and artificial neural networks. The later do not approach by any means the complexity of the former but two key similarities may be observed. The first

similarity is that both networks consist of simple building blocks, the neurons, which serve as computational devices. Artificial neurons are of course simpler than biological. The second important similarity is that learning influences the strength of the connections between the neurons and that these connections basically determine the function of the network. When someone trains a network he basically tries to determine the appropriate connections between neurons. ([12] chapter 1, p.1-9)

1.2.2 Neuron model and architecture of neural network

As stated earlier an artificial neural network is an information-processing system that has certain performance characteristics in common with biological neural networks. In the previous section we described a simplified biological neural network. In this section a basic mathematical model of an artificial neuron we will be presented and the interconnection between those neurons will be described. The purpose of this section is to help the reader understand the basic operation of an artificial neural network (from now on referred simply as neural networks).

A neural network consists of a large number of simple processing elements called neurons, units, or nodes. Each neuron is connected with other neurons by means of directed communication links which are associated with a numerical value called 'weight'. The weights, just as the strength of the synapses in biological neural networks represent information being used by the network to solve a problem. Neurons accept signals and have an internal state, called activity level, or activation of the neuron, which is a function of the inputs it has received. This function is called transfer function (or activation function). The activation of the neuron may also be referred as the output of the neuron. A neuron typically sends its activation to several other neurons. It can send only one signal at a time, although that signal can be sent to several other neurons. ([9] p.3-6) In some cases each neuron also has an external input, that means an input that does not represent a variable of the problem, nor a signal from other neurons, which is also connected with the neuron through a weighted link. This external input is usually set to 1 and the weight of the link that connects it to the neuron is called bias, or offset.

Fig.1-15 shows a single input neuron. The input p which is a scalar is multiplied by the weight w, which is also a scalar and is sent to a summer (Σ). The other (external) input is weighted by a bias b and is also set to the summer. The summer output n is often referred as the net input, and it goes into a transfer function f which produces the output α of the neuron.

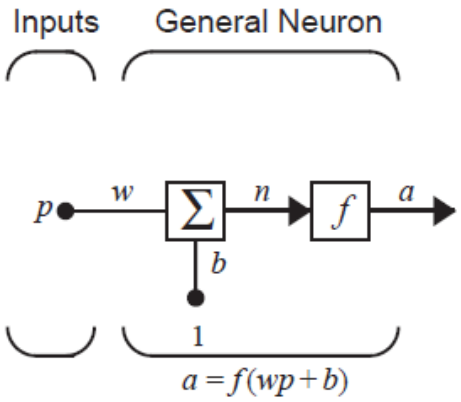

Figure 1-15. A single- input neuron with bias

Scalars w and b are both adjustable parameters of the neuron. Transfer functions are usually chosen by the designer of the neural network and the parameters w and b are adjusted by some learning rule so that the input/output relationship of the neuron meets a specific goal. The contribution for positive weights w_n is considered as an excitation and for negative w_k as inhibition.

Neurons rarely have a single input and bias as in Fig.1-15. They usually have more than one input. In this case the output α of a neuron which has R inputs $p_1,p_2 \ldots p_R$ with weights corresponding to these inputs w: $w_1,w_{2 \ldots} w_R$ and a bias b will be:

$$a = f\ (b + \sum_{j=1}^{R} w_j p_j) \tag{1.16}$$

The previous equation can also be written in matrix form, where **w** and **p** are vectors containing the weights $w_1,w_{2 \ldots} w_R$ and inputs $p_1,p_2 \ldots p_R$ respectively:

$$a = f(\boldsymbol{wp} + b) \tag{1.17}$$

This multiple input neuron is shown in Fig. 1-16. This type of neuron that calculates its output based on equation (1.16) is called sigma unit (because of the Greek letter representing the summer). The term in the parenthesis in equation (1.16), which estimates the net input of the unit, is called propagation rule of the unit. It is called that way because it is used to estimate the signal (output) the neuron will propagate to neurons connected with it. The propagation rule in equation (1.16) is not the only available rule for the estimation of the net input of a neuron, but it is the most popular. The propagation rule defines the name of the unit. Thus, apart from sigma units, there are also product units [31], which use multiplication and sigma-pi units ([14] p.16) which have a more complicated rule, using sums and products. The later rules are not commonly used but can help in the solution of some problems.

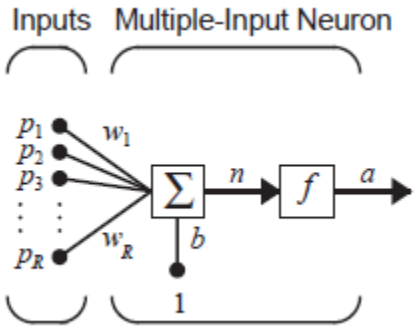

Figure 1-16. Multiple-input neuron

In most problems a single neuron does not provide the necessary flexibility in a neural network (with one neuron after all we can barely call it a network) to adjust its weights and approach acceptable solutions. A larger amount of neurons is usually needed to operate in parallel in what is called a layer of neurons. One such layer is shown in Fig.1-17, with S neurons in the layer. In the same manner as before, **p** is the input vector. The output now includes S values as the number of neurons, and can be represented with a vector **α**. The same applies for biases, which are now represented by vector **b**. Of particular interest is the weight matrix which is no longer a vector but a matrix with dimensions SxR (Neurons in the layer X Number of inputs). Each row represents a neuron and each column an input. Hence the weight matrix has the form:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} \quad (1.19)$$

The row indices of the elements of matrix indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. Thus, the indices in $w_{4,5}$ say that this weight represents the connection to the forth neuron from the fifth source.

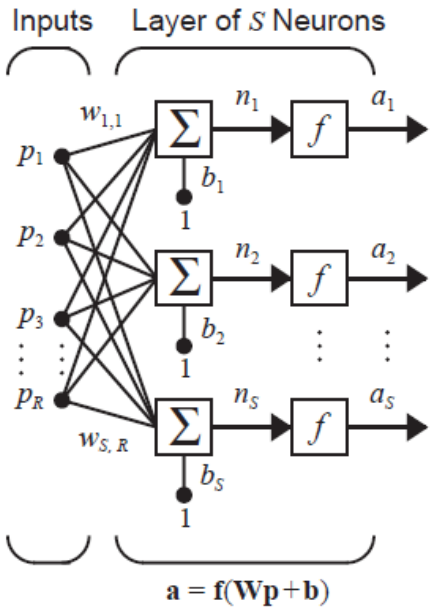

Figure 1-17. Layer of S neurons and its output

Most of the time the number of neurons in a layer, does not equal to the numbers of inputs. Also transfer functions in a layer do not have to be the same. Different transfer functions can exist in a layer. It is the same as combining some of these networks (with same transfer functions in their layers) in parallel. Both would have the same inputs, and each of them would produce some of the outputs.

Layers of functions can be combined to create various architectures of neural networks with different capabilities. In Fig. 1-18, a multilayer neural network is shown. Multilayer neural networks are layers of neurons combined in series. In this layer the output of the first layer serves as input to the second layer, and the output of the second layer, serves as input to the third layer. The third layer is usually called the output layer. The first and second layers are called hidden layers, because we have no access to them through inputs or outputs (it is like considering the neural network as a black box). Multilayer neural networks are complicated by nature and we will need complicated symbolism to describe them. Specifically superscripts will be used to identify the layers. Thus, $\mathbf{W}^1$ refers to the weight matrix of the first layer, $\mathbf{b}^2$ refers to the bias vector of the second layer and so on. In Fig. 1-18 all the layers have the same number of neurons (S) and this result in square weight matrices for the second and the third layer (SxS). It should be noted that this is not always the case as layers with different number of neurons can be combined. A final note on Fig. 1-18, in order to avoid confusion, is that the symbol $w^3{}_S{}^3{}_S{}^2$ means: the weight of the third layer that connects the s neuron of the second layer (source) to the s neuron of the third layer (destination). ([12] p.1-12)

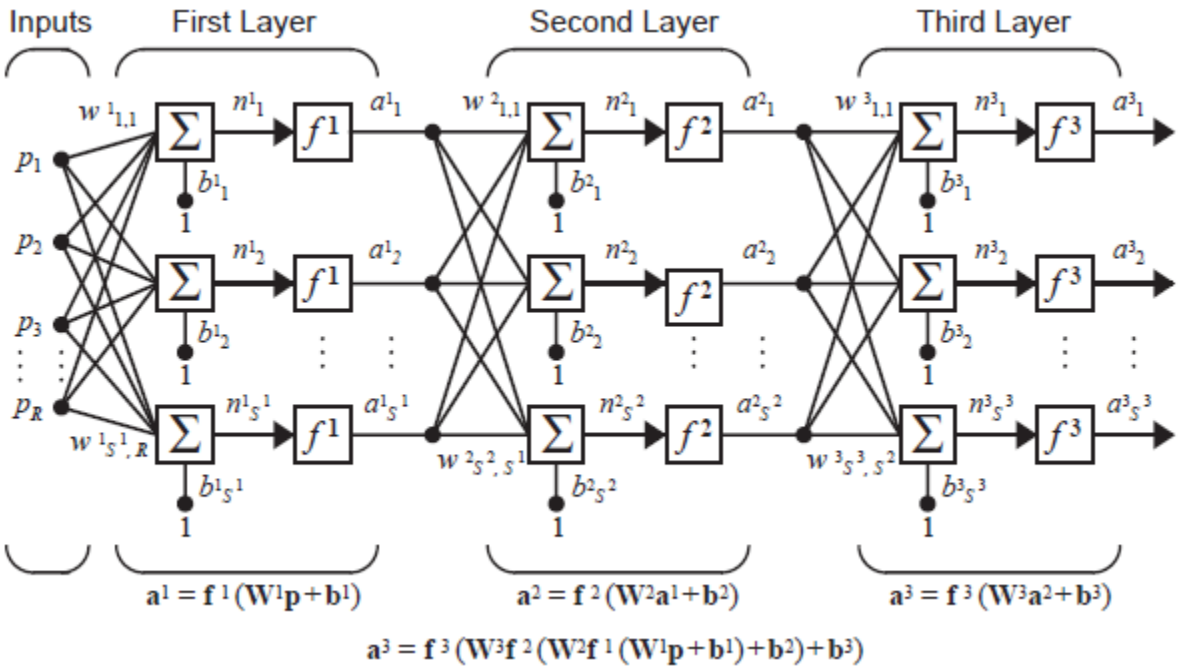

Figure 1-18. Three- layered (multilayer) neural network and its output

The connections in this neural network are characterized as directed links, in the sense that signals are always propagated from the first layer to the second and from the second to the third, and not in the opposite direction. These networks are called feedforward networks.

The feedforward network architecture is not the only available architecture in neural networks. Many other network architectures are available, with different features and capabilities. The network described so far is a static network. This means that the output can be calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs or states of the network. Such networks are recurrent networks which can have some feedback (recurrent) connections and contain memory elements. ([12] Chapter 14, p.2) Other recurrent neural networks, called bi-directional neural networks allow the flow of signals in connections in both directions. [32] There are also networks used for pattern classification, which are based on competition and contain connections to themselves, while other networks of the same type (competitive) such as counterpropagation networks may have more complex architectures. ([9] p.156-202) These networks architectures will not be considered here, as they are not used in this project.

1.2.3 Transfer functions and training of artificial neural networks

Transfer function as described in the previous section is basically the rule which gives the effect of the total input on the activation of the unit. Usually, the activation function is a non decreasing monotonic function of the total input of the unit, although activation functions are not restricted to non decreasing functions. Some sort of hard limiting threshold function is used many times (such as a sign function), while others a linear or semi-linear function is preferred. For these smoothly limiting functions often an S-shaped sigmoid function is used (for example a hyperbolic tangent). In some cases, the output of a unit can be a stochastic function of the total net input of the unit (as for example in radial basis function networks). In these cases the activation (output) of the neurons is not deterministically determined by the neuron's net input, but the input determines the probability p that a neuron gets a high activation value. ([14] p.16-17) Fig. 1-19, shows various activation functions for a neuron.

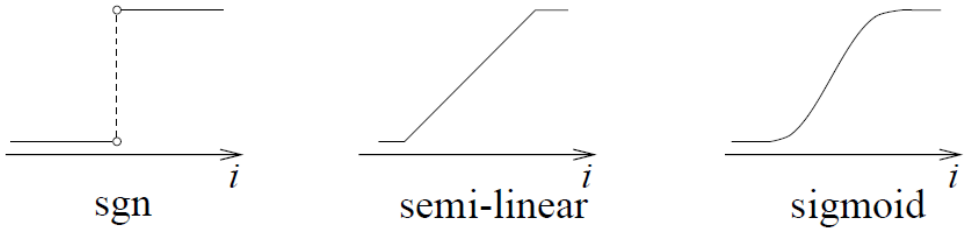

Figure 1-19. Various types of activation functions for a neuron

As mentioned earlier, neural networks adapt their weights, to meet the needs of a particular problem by means of a procedure called training. Thus, in addition to the architecture, a very important characteristic of neural networks is the method by which they set the values of their weights. Training procedures can be classified into three general types: unsupervised training, reinforcement learning and supervised training. These types are presented below:

Unsupervised learning is the biologically most plausible method, but is not suitable for all problems. Only the input patterns are given; the network tries to identify similar patterns and to classify them into similar categories. Self-organizing nets group similar input vectors together without the use of training data to specify what typical member of each group looks like or to which group each vector belongs. Unsupervised learning can also be used in other tasks, other than clustering.

Reinforcement learning is a procedure in which the network receives a logical or a real value after network completion of a sequence, which defines whether the result is right or wrong. Intuitively someone can say that this procedure should be more effective than unsupervised learning since the network receives specific criteria for problem-solving.

Supervised learning is perhaps the most typical neural network setting, where training is accomplished by presenting a sequence of training vectors, or patterns, each associated with a target output vector. The training set consists of input patterns as well as their correct results in the form of the precise activation of all output neurons. Thus, for each training set that is fed into the network, the output can directly be compared with the correct solution and the network weights are changed according to their difference. The purpose is to change the weights in a way that the network will not only remember the correct values for the inputs it has been trained with, but also to provide plausible answers to unknown, similar input vectors (generalization).

Despite the importance of the training method and the adjustability of the weights, some nets occasionally have fixed weights and do not need the iterative procedure of training but these are just special cases. Neural networks are being used to accomplish different tasks that fall into many areas such as mapping, clustering, constrained optimization and function approximation. There is a correspondence between the type of training that is appropriate and the type of problem someone wishes to solve. ([9] p. 15-17)

Chapter 2: Approaching the phase problem with feed forward neural networks

In the first chapter an attempt to introduce some basics concepts on crystallography and neural networks has been made. These basic terms will be needed for the chapters that follow. With what has been presented so far in mind, in chapter 2, the purpose of this project will be defined and the details of the implementation of the neural networks that have been used will be discussed.

2.1 Aim of the study

Crystallography enables us to "see" our molecular biology in images and one of its major focuses is the study and determination of the 3D structures of proteins to gain functional insights on the structure of proteins. Our understanding of how biological molecules interact is often presented as a number of images derived from bio-molecular crystallography. In molecular biology, crystallographic approaches are often used to study the structure and conformation of proteins which are involved in various regulatory processes under normal and diseased conditions. Being a sub-nano imaging technique, crystallography provides unbiased insight into the complex features of proteins, DNA, and RNA, and their relationships to substrates, inhibitors, and binding partners.

X-ray crystallography is sometimes regarded as a science in its own right and, indeed, there are many professional crystallographers who devote all their efforts to the development and practice of the subject. The fundamental problem these scientists face is the phase problem. Various methods have been proposed each with its own advantages and disadvantages, yet the phase problem has not gone away.

A substantial proportion of the techniques used for phase determination are the direct methods. Those are techniques of phase determination which involve solely the comparison of structure factor magnitudes derived from the study of a crystal without any reference to prior knowledge of likely structures, or isomorphous crystals. At present, direct methods are the preferred method for phasing crystals of small molecules having up to 1000 atoms in the asymmetric unit and they are routinely used for the solution of small structures. However, they are not generally capable to determine structures of larger molecules such as proteins by themselves [33] and have only recently been applied to the solution of small proteins containing about 50 amino acid residues. [34]

The starting point of this project, was the thought that neural networks could be used to help in the determination of phases of protein structures, by means of only the directly

observed experimental data. If neural networks through training could learn the necessary relationships, as those used in direct methods, or at least approximate them in a reasonable fashion, it would greatly enhance the work of crystallographers working towards this direction. Furthermore, successful neural networks could be analyzed later and maybe provide more information in this area. An additional appealing feature of neural networks is their parallel structure: all of the neurons are operating at the same time. Even though most artificial neural networks are currently implemented on conventional digital computers, their parallel structure makes them ideally suited to implementation using VLSI, optical devices and parallel processors. This ability of neural networks gives an additional motivation, because really efficient systems could be created (in terms of computational time) that would aid in protein structure determination.

The aim of this study is to examine whether training of neural networks that reach this target is feasible. This is done by considering the case of simple two dimensional, centrosymmetric arbitrary structures. This is a much simpler case than that of a real structure in many ways, but if neural networks produce successful results for cases like this, additional research can be done for more complicated structures.

Neural networks are being used to accomplish different tasks that fall into many areas such as mapping, clustering and function approximation. The strategy followed for neural networks in this project is that of function approximation. For this reason feed- forward networks were selected to be trained, using backpropagation algorithms, because these networks are considered to possess universal function approximation capabilities.

2.2 Direct Methods

Direct methods are those mathematical techniques that attempt to solve the phase problem from the observed amplitudes through purely mathematical techniques, with no recourse to structural chemical information. This is the feature that these techniques have in common with the neural network we wish to determine, because this neural network after training will not have access to structural chemical information when new structures will be presented to it, in order to determine its phases. The only information it will be given are the observed amplitudes, just as the case of direct methods. So, successful neural networks, most probably would be function approximations of relationships used in direct methods. This is the reason why direct methods are briefly described in this section.

Direct methods are based on two basic criteria. The first is that given a set of observed amplitudes |F(hkl)| , the corresponding phases must be such as to produce (or be consistent with) non negative electron density values everywhere. The second criterion is that, if the atoms in a structure are identical and spherically symmetrical, and do not overlap in space, diffracted amplitudes and phases are connected by exact equations. The second criterion may be extended in a probabilistic sense to the case of unequal atoms. ([5] p. 2-6)

Direct methods involve the comparison of the structure factors magnitude, and in order to generalize the mathematics a unitary structure factor U_{hkl} is defined:

$$U_{hkl} = \frac{F_{hkl}}{F_{000}} = \frac{F_{hkl}}{\sum_j f_j} \tag{2.1}$$

The unitary structure factor has absolute values that range from 0 to 1 and they are actually structure factors with the effects of atomic size removed. U_{hkl} is a structure factor which has the same phase as F_{hkl} but whose values range from -1 to +1. These extreme values correspond to the cases where all atoms scatter in phase. Unitary structure factors with unity absolute value, are rarely if ever found but the larger the value observed the greater are the constraints which are placed on the atomic position.([2] p.323-325) U_{hkl} represents the fractional value of a given structure factor as compared with its maximum possible value.

Another useful quantity in direct methods is the normalized structure factor E_{hkl} :

$$E_{hkl} = \frac{U_{hkl}}{<|U_{hkl}|^2>^{1/2}} \tag{2.2}$$

where

$$< |U_{hkl}|^2 > = \frac{1}{N} \sum_{hkl} |U_{hkl}|^2$$

in which the summation is over the N values of $|U_{hkl}|^2$ corresponding to all reciprocal lattice sites, hkl. $< |U_{hkl}|^2 >^2$ therefore, represents the average value of $|U_{hkl}|^2$. The advantage of E_{hkl} is that it has been shown that its use is tantamount to regarding the atoms within a structure as points which do not suffer from thermal motion. ([1] p.459-460) Another advantage is that, since all classes of Bragg reflections are normalized to the same value, it is possible to compare set of reflections. The distribution of the $| E_{hkl} |$ values is in principle, and often in practice, independent of the size and content of the unit cell. It does depend however, on the presence or absence of a center of symmetry in the space group. This way the structure factors are modified so that the maximum information on atomic position can be extracted from them.

The first direct methods to be introduced were the Harker-Kaspar inequalities. In this method which assumes that we work on centrosymmetric crystals, all structure factors F_{hkl} and hence all unitary structure factors U_{hkl} are real. This is a result of the Fourier transform properties, because as mentioned in the first chapter the Fourier transform of a symmetrical function (f(x) = f(-x)) is real. This means that the structure factors have 0 or π phases and therefore are assigned with positive or negative signs respectively.

This method is based in the following relationship:

$$U_{hkl}{}^2 \leq \frac{1}{2}(1 + U_{2h\ 2k\ 2l}) \tag{2.3}$$

In case $U_{hkl}{}^2 >$ ½ then $U_{2h\ 2k\ 2l} \geq 0$ or in other words the sign of reflection 2h2k2l is positive whatsoever its absolute value is. Note that the sign of hkl may have both values. In practice the case $U_{hkl}{}^2 >$ ½ does not occur often. However, when $|U_{2h\ 2k\ 2l}|$ is large, the expression (2.3) requires the sign of 2h2k2l to be positive even if U_{hkl} is somewhat smaller than ½. Moreover, when $|U_{hkl}|$ and $|U_{2h\ 2k\ 2l}|$ are reasonably large, but at the same time (2.3) is fulfilled for both signs of $U_{2h\ 2k\ 2l}$ it is still more likely that the sign of 2h2k2l is positive than negative. For example, for $|U_{hkl}| = 0{,}4$ and $|U_{2h\ 2k\ 2l}| = 0{,}3$ when the sign of 2h2k2l is assumed positive, from equation (2.3) we have 0,16≤0.5+0.3 which is certainly true, and when the sign of 2h2k2l is assumed negative we have 0,16≤0.5-0.3 which is also true. Then probability arguments indicate that the positive sign is more likely. This probability is a function of $|U_{hkl}|$ abd $|U_{2h\ 2k\ 2l}|$ and in this example the positive sign probability for 2h2k2l is >90%. ([6] p.8)

Different inequalities may be generated for all the 230 space groups. This represents a formidable range of possible relationships to try, but certain difficulties arise. One of them is the fact that the use of inequalities alone cannot identify all the phases unambiguously. The drawback of inequalities is that, to obtain definite results the reflections used must have amplitudes which are very sizable fractions of F_{000}. Another limitation is that inequalities are restricted in centrosymmetric crystals. The resolution of a positive/negative sign is a far simpler problem than the identification of a phase angle anywhere between 0 and 2π. No useful inequalities have been found for the non centrosymmetric crystal. ([1] p.460-463)

Another, more powerful approach to the problem of phase determination is provided by the probability relationships. When the magnitudes of the unitary structure factors are not large enough for inequalities but still relatively large, it is possible to set up equations of signs which are probably true, and from these to extract phase information. These sign relationships are based on the Sayre equation:

$$F_{hkl} = \varphi_{hkl} \sum_{h'} \sum_{k'} \sum_{l'} F_{h'k'l'}\ \ F_{h-h',k-k',l-l'} \tag{2.4}$$

where φ_{hkl} is a calculable scaling term. The implication of Sayre's equation is that any structure factor F_{hkl} can be calculated as the sum of the products of pairs of structure factors whose indices add to give (hkl). At first sight equation (2.4) may appear useless since to determine one factor we must know the other two. However, Sayre pointed out, that for the case where F_{hkl} is large, the series must strongly tend towards one direction (+/-, talking about centrosymmetric cases) and that this direction is generally determined by the agreement in sign among products between large structure factors. Thus for the case of three large reflections:

$Sign(F_{hkl}) \approx Sign(F_{h'k'l'})\, Sign(F_{h-h',k-k',l-l'})$ or

$$Sign(F_{hkl})\, Sign(F_{h'k'l'})\, Sign(F_{h-h',k-k',l-l'}) \approx 1 \qquad (2.5)$$

where "≈" means "probably equal". The sign function may return ±1. Equation (2.5) is the probability equation derived from Sayre's equation and is the basis for most of the current methods of phasing. Since the phases of unitary structure factors and normalized structure factors are the same as those of the corresponding structure factors F_{hkl} , it also applies to these modified factors. In addition, equation (2.5) holds in those cases in which inequalities also apply, for this reason inequalities are often ignored in practice. Sayre's equation can be extended in the case of non centrosymmetric crystals. ([2] p.321-324) For structure factors with large values:

$$\varphi_{-h-k-l} + \varphi_{h'k'l'} + \varphi_{h-h',k-k',l-l'} \approx 0 \qquad (2.6)$$

where φ_{hkl} is the phase of the hkl reflection. This is known as the 'sum of angles' formula and reflections (-h-k-l), (h'k'l'), (h-h' k-k' l-l') define a "triplet" of reflections whose angles are related. ([5] p.4-6) The Bragg reflections $(\bar{2}13)$, $(42\bar{5})$ and $(\bar{6}12)$ for example provide such a triplet.Various analyses have been carried out to determine the statistical interpretation of relation (2.6) so that the probability of its being correct may be quantified.

These relative phase combinations are of great importance in direct methods and they are called structure invariants, because they are uniquely determined by the crystal structure and are independent of the choice of origin. There also exist many more linear combinations of relative phases whose values remain unchanged when the location of the origin is changed provided these changes are made subject to specific space group symmetry constraints. These are called structure seminvariants. ([4]p. 290-293)

Back to the centrosymmetric case, one of the problems associated with the use of probabilities is that they can give the signs of a certain structure factor only if it is associated with two others already known. Often a few signs may be known unambiguously or may be arbitrarily allotted. For most of the space groups, the phases of three Bragg reflections may be chosen arbitrarily according to certain rules, and this

choice will define the origin of the unit cell. These reflections are called origin-fixing reflections. ([3] p.352-354) The definition of the origin of the unit cell is important because although it will not affect the magnitudes of the structure factors it will affect their relative phases. Starting with these phases several others may be determined. But one of the features of the use of the probabilities is that it is not possible to assign phases to all structure factors simply by moving from one reciprocal point to the next. The technique works for sets of triplets, and sooner or later it becomes stuck. We can use sign symbols to determine other phases. Suppose a certain structure factor which has a large magnitude but unknown phase. We can allot a sign symbol a to the phase of this structure and then define other structure factors phases in terms of a, which means same sign as the first structure factor whose phase we allotted this sign, or $-\alpha$, which means the opposite sign. This process may be repeated as often as needed if the procedure stuck again, and eventually most of the structure factors may be given either as an unambiguous sign or one of the symbols α,b,c… we have assigned.

These symbols represent one of two possibilities, a positive or a negative sign. If n symbols were used there are 2^n possible ways in which the set of structure factors may be associated with signs. For a structure expressed with only a small number of signs, it is relatively easy to calculate a Fourier synthesis and an electron density map for each of these possibilities Inspection of these maps may allow the selection of one of them as being consistent with other information known about the structure.

The problem is far more complicated for non centrosymmetric structures, because the phase can take values from 0 to 2π. There is one particular probability relation, known as the tangent formula which is applicable to these cases, and has been widely known:

$$\tan a_h = \frac{<|E_{h'}|\,|E_{h-h'}|\sin(a_{h'}+a_{h-h'})>_{h'}}{<|E_{h'}|\,|E_{h-h'}|\cos(a_{h'}+a_{h-h'})>_{h'}} \tag{2.7}$$

where $<\ldots>_{h'}$ notation stands for an average taken over all values of h`k`l`. ([1] p.463-465)

2.3 Neural Network Implementation

Neural networks in this project were implemented in MATLAB. MATLAB (matrix laboratory) is implemented for many operating systems. In this study it was used in Microsoft Windows 7 environment and in Linux based Ubuntu operating system. The simulations of neural networks run on a computer based on an Intel i7 core central processor with 4 gigabyte RAM.

MATLAB was used because it is already in use in many institutions and is practical for data analysis, data extraction, data processing, plotting and many more. It is a multi-paradigm numerical computing environment and fourth-generation programming language and in many cases prototype solutions are obtained faster in MATLAB than solving a problem by using other programming languages. This software is widely available and, because of its matrix/vector notation and graphics, is a convenient environment in which to experiment with neural networks. It offers a simple, flexible and structured script language with many similarities with Pascal and supports the easy creation and linking of libraries. One of the libraries it offers is the Neural Network Toolbox. With this toolbox, neural network algorithms can be quickly implemented, and large scale problems can be tested conveniently. Specifically, version 8.0 of the Neural Network Toolbox was used.

One of the main disadvantages of MATLAB is that applications written in this language usually perform worse, in terms of calculation time, than applications written in more classic programming languages such as C, C++, Fortran etc. However, this is not necessarily the case with the implementation of neural networks examined in this project. In a recent study the implementation in MATLAB of a neural network training algorithm, the back propagation algorithm which is used in this project, was compared with several, other back propagation programs which were written in the C++ language. The MATLAB implementation was about 3 times faster and that happens because neural networks algorithms make heavy use of matrices multiplications (for example as in equation (1.17)) and MATLAB is optimized for this kind of calculations.

Scripts and programs used in this project can be found in the appendix at the end of thesis.

2.3.1 Structures under Study – Input/output data

In this section the data used to train, test and evaluate the neural networks are presented. For this study simple two dimensional centrosymmetric structures are employed to examine the ability of neural networks to approximate phase relationships. More specifically arbitrary structures that belong to the oblique p2 two-dimensional space group are used. The oblique p2 space group is illustrated in Fig.2-1, in the form given in the International Tables. The twofold axis is at the origin of the cell and it will reproduce one of the structural units, represented by an open circle, in the way shown. The right hand diagram shows the symmetry elements; the twofold axis manifests itself in two dimensions as a centre of symmetry. It will be seen that three other centers of symmetry are generated at the points (x, y) = (½,0), (0,½) and (½,½). The four centers of symmetry are all different in that the structural arrangement is different as seen from each of them. ([8] p.25)

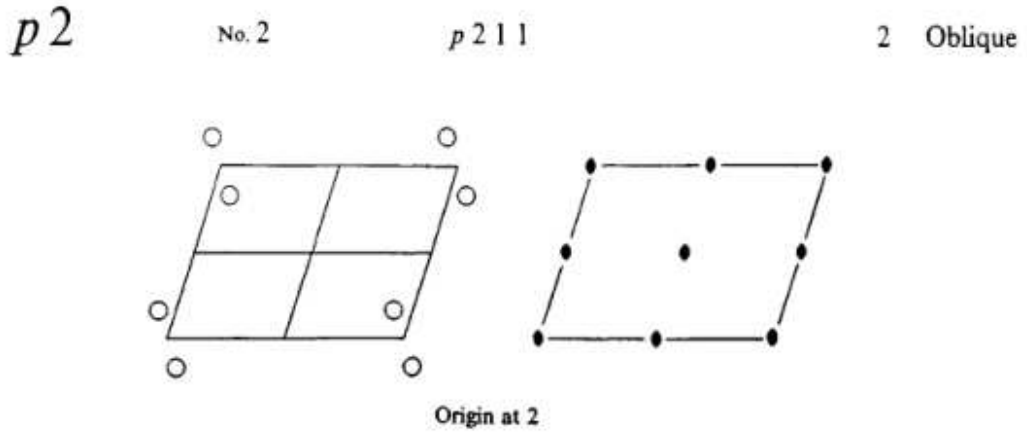

Figure 2-1. The two dimensional space group p2 as it appears in International Tables for X-ray crystallography

The dimensions of the sides of the unit cell which is used in our experiments are 10 *Å* and *15 Å*. The angle is set at 110 degrees. In each of these unit cells we suppose that there are 30 identical atoms, with 2 electrons each, at random positions.

A program in C language has been created, that produces these structures that belong to the oblique p2 space group. More specifically the program sets random x,y coordinates for the position of the atoms, and given a maximum resolution value which defines the number of reflections, it calculates the structure factors for this arrangement of atoms in

the unit cell, based on equation (1.14). In this group of crystals, the origin is usually taken on the center of symmetry in the unit cell at 0, 0. The equation that gives the structure factors of the unit cell with an origin of coordinates at the point 0, 0 is: ([3] p.353)

$$F_{hk_{0,0}} = \sum_{j=1}^{N} f_j \cos 2\pi(hx_j + ky_j) \quad (2.8)$$

A thermal motion is also taken into account and that is why a thermally corrected scattering factor is needed in the equation (as the one in equation (1.15)). That means that a temperature factor is also necessary to be defined in the program. Resolution represents the average uncertainty for all atoms. In contrast, the temperature factor quantifies the uncertainty for each atom. For each arbitrary structure created, the output of the program are the values of the structure factors of the hk reflections (we use two indices because we are in two dimensions) defined both in magnitude and phase. The phase has the form of a positive or negative sign since we refer to a centrosymmetric structure, because its Fourier transformation is a real number.

In this project's experiments a value of 1 Å is used as maximum resolution, and a value of 10 is assigned to the temperature factor.

When we perform a Fourier synthesis of the structure factors calculated from this program we can get the electron density map of the structure it represents. Two examples of structures produced by the C program are shown in Fig. 2-2. The Fourier synthesis of the structure factors has been made with the "Pepinsky's machine" program. [E-6]

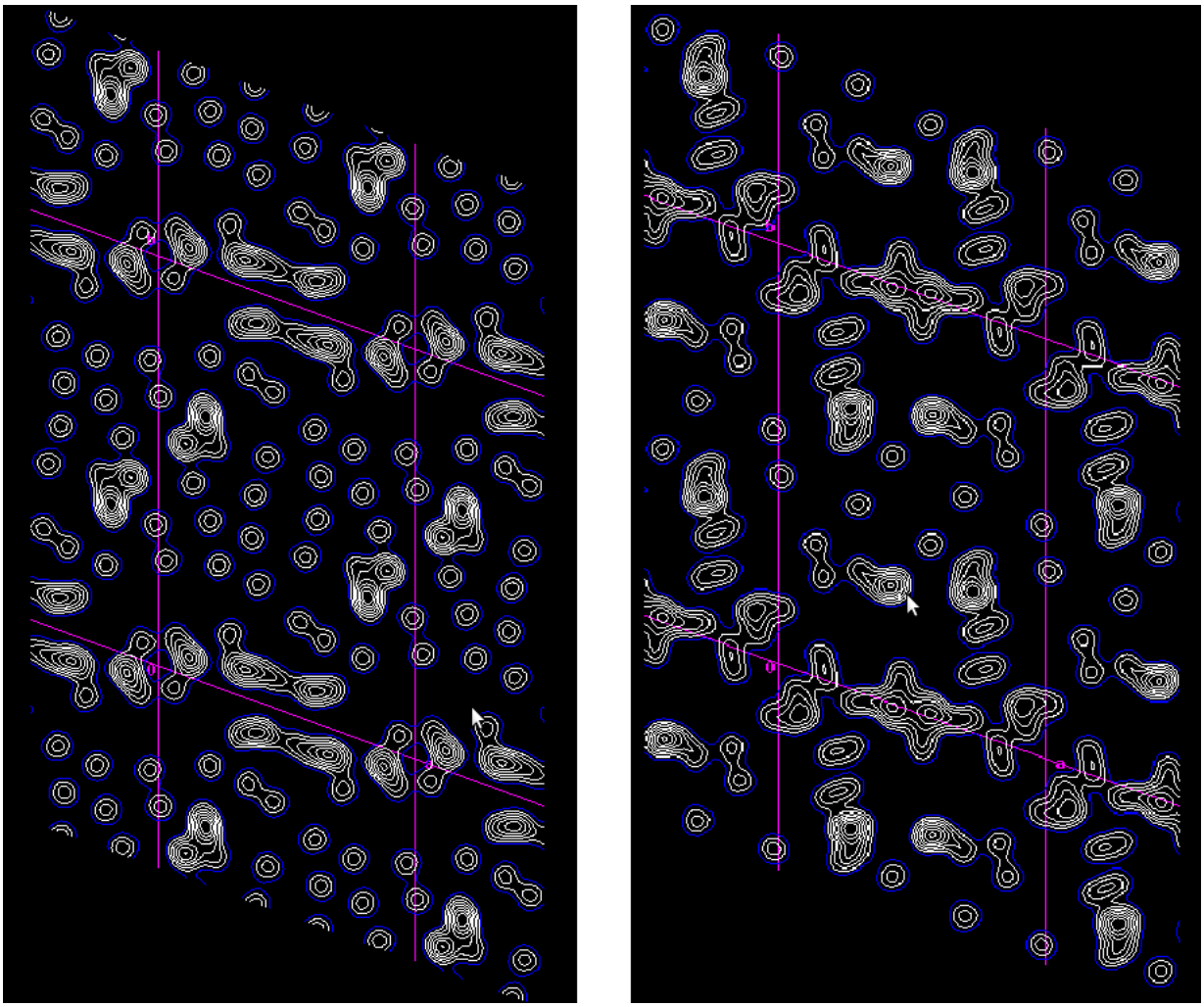

Figure 2-2. The electron density maps of two random structures created to be used with the neural networks

From the outputs of this program we determine the inputs of the neural networks. The input in the desired neural network has to be something that is directly connected to the observed data of a diffraction experiment. That is, the amplitude of the structures factors without their phases (sign). Thus the input of neural networks is the absolute value of the structure factors produced be the program in C. Other forms of input data that could be useful, since they are related with the use of direct methods, are the absolute values of the unitary structure factors (given from equation (2.1)) and the absolute values of normalized structure factors (given from equation (2.2)). These factors can be used as inputs, because they can be readily estimated from the observed experimental data (absolute values of structure factors).

Unitary structure factors are obtained using equation (2.1) in the outputs the program in C produces. Normalized structure factors have a more complex formula and they are estimated more easily directly from the C program, when we omit the thermal correction for the scattering factor. This is equivalent to a temperature factor B_j equal to 0 (equation (1.15)). This result comes from the fact that the use of normalized structure factors is tantamount to regarding the atoms within a structure as points which do not suffer from thermal motion. The absolute values of structure factors, unitary and normalized factors are all used as inputs in different networks.

The output data of the neural networks present a wider variety and depends on the structure of the neural network and the input used. The only thing all networks have in common, regarding their outputs is the number of these outputs. Each network needs to return something that will give us information on the phase of each structure factor. That means we need as many outputs as inputs. For each input which consists of the absolute value of a structure factor we must get a result, an output, which will inform us about its phase. One thing must be noted here: the outputs mentioned in this paragraph, which are produced basically from the C program, are the correct, the desired outputs of the neural networks we wish to create. They are used in the training procedure, as training patterns and their values are often referred as targets.

Up to now the number and the type of inputs in the neural network has been defined, as well as the number of outputs. Another important decision is the type of the network.

As mentioned earlier neural networks are being used to accomplish different tasks that fall into many areas such as pattern classification, clustering and function approximation. The strategies of pattern classification and clustering were not used because they don't seem to match the needs of the problem. The purpose is the creation of a neural network that is able to estimate the phases of the structure factors of a crystal under study by means of the observed amplitudes of the diffraction pattern. In the simple case of a simplified two dimensional structure such as the one described above, 225 structure factors are produced from the C program. That means we need 225 inputs and 225 outputs, one for each phase. Now if someone wants to train a pattern classification neural network, during training he has to present at least one sample of each pattern in the network. In this case the different patterns the neural network wishes to classify is the different combinations of +/- signs in the outputs. That means the network has to be trained with at least 2^{225} samples with different outputs, even for a simple structure like this, let alone for real life proteins. The same holds for a clustering strategy, the clusters formed from the problem as it is defined reaches impractical sizes. For these reasons the strategy followed for neural networks in this project is that of function approximation, which is implemented with feed forward neural networks. These neural networks through training could approximate relationships used in the direct methods such as the sign relationships of triplets.

2.3.2 Feed Forward Networks and Function approximation.

Feed forward neural networks can be used as general function approximators. In this section the flexibility of these networks for implementing functions will be illustrated with an example. The notation of section 1.1.2 will be used.

In Fig. 2-4 a multilayer feed forward neural network is shown.

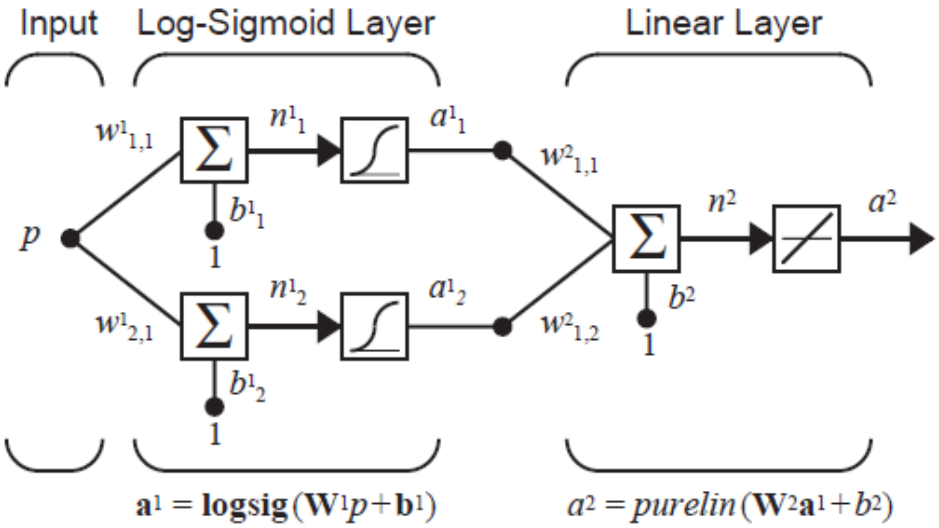

Figure 2-3. Function approximation network with the outputs of each layer

This network consists of two layers. The first layer, described as Log-sigmoid layer is the hidden layer of the network. This layer is called log-sigmoid because it uses the following S-shaped logistic sigmoid transfer function:

$$f^1(n) = \frac{1}{1+e^{-n}} \tag{2.9}$$

where the $f^1(n)$ notation represents the transfer function of the first layer, n being the net input of the neurons. Fig. 2-4 shows the graph of this function and its representative symbol.

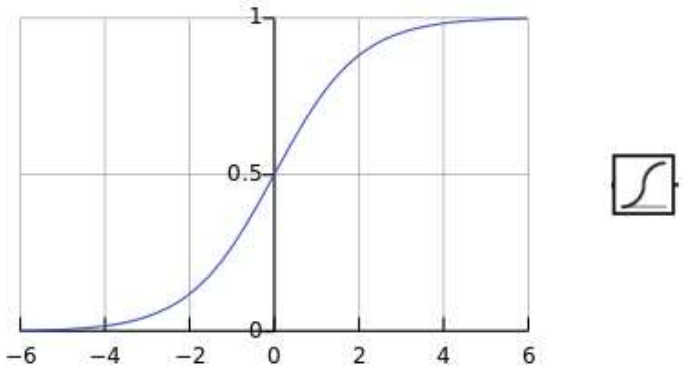

Figure 2-4. Graph and symbol of the logistic sigmoid function (logsig)

The second layer, which is also the output of the neural network, has a linear function (actually it is the linear parent function) as transfer function:

$$f^2(n) = n^2 \tag{2.10}$$

where n^2 does not represent the squared value of n, but stands for the net input of the second layer. This linear function will be called purelin (as noted in MATLAB) from now on and its graph and symbol is given in Fig. 2-5.

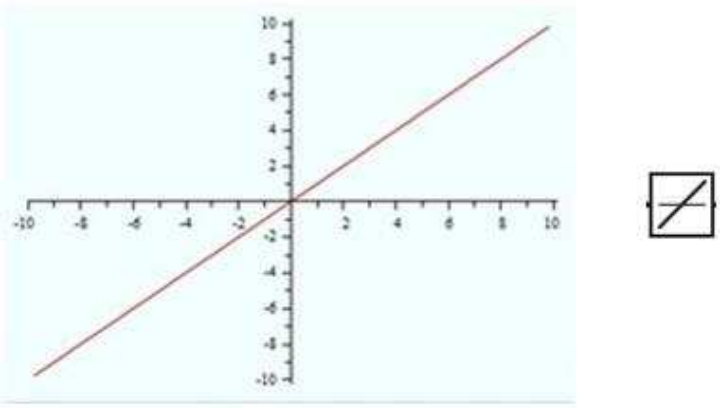

Figure 2-5. Graph and symbol of the linear parent function (purelin)

The two layer network in Fig. 2-3 is called a 1-2-1 network because it has one input, 2 neurons in the hidden layer and one output. For this example the following values are set for the biases and weights of the network.

$$w_{1,1}{}^1 = 10 \qquad w_{2,1}{}^1 = 10 \qquad b_1{}^1 = -10$$
$$b_2{}^1 = 10, \qquad w_{1,1}{}^2 = 1 \qquad w_{1,2}{}^2 = 1 \qquad b^2 = 0$$

The network response for these parameters is shown in Figure 2-6, which plots the network output α^2 as the input p is varied over the range [-2,2].

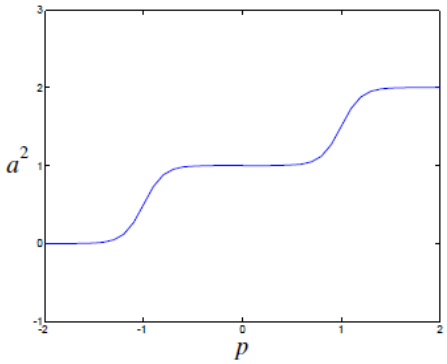

Figure 2-6. Network response for specified values of parameters

The response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters the shape and location of each step changes, as described in Fig. 2-7. This figure shows how the network biases in the first (hidden) layer can be used to locate the position of the steps in (a).In (b) the effect of the weights to the slope of the steps is illustrated, while (d) shows how the bias in the second (output) layer shifts the entire network response up or down.

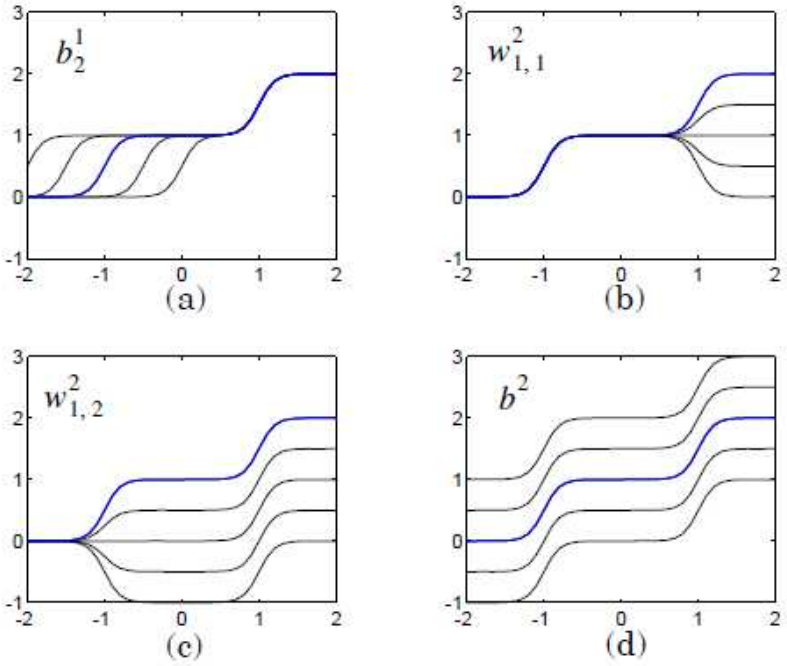

Figure 2-7. Effects of various parameters changes in the network response

This example shows the flexibility of this type of networks. (example used from [12] p.4-7) These feed forward networks can be used to approximate almost any function, given a sufficient number of neurons in the hidden layer. A non linear activation function, such as the logistic function is used to squash the sum of the net input. It has been shown that

networks with two layers, non-linear transfer functions in the hidden layer and a linear transfer function in the output layer can approximate a wide variety of functions of interest to any degree of accuracy, given the appropriate parameters [33].

During this project mostly neural networks similar or based to the one described here were used. These networks mainly differ in the transfer function of the hidden layer and the number of neurons in the hidden layer. Specifically the networks use two types of transfer functions in the hidden layer. The first type is the logistic sigmoid function in equation (2.9), and the second is the hyperbolic tangent sigmoid which is simply a scaled and shifted version of the logistic sigmoid. The equation for this function is given by:

$$f(n) = \frac{2}{1+e^{-2n}} - 1 \tag{2.11}$$

The plot and the symbol of this function is given in Fig.2-8.

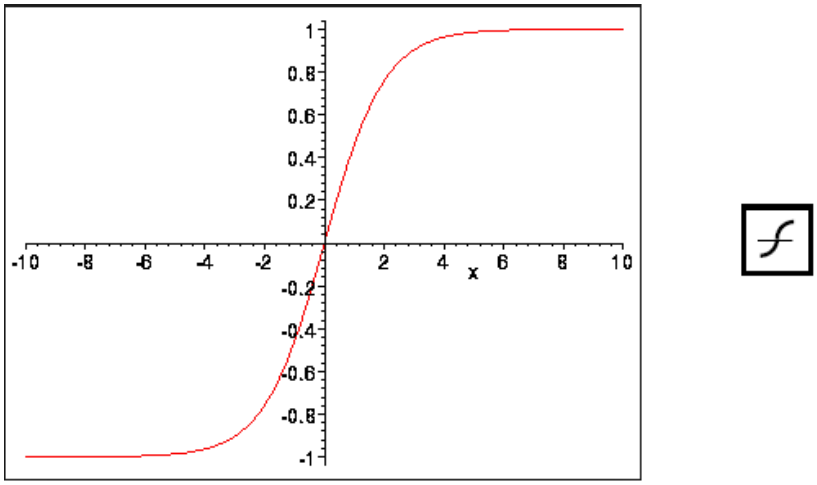

Figure 2-8 Plot and symbol of the Hyperbolic tangent sigmoid transfer function

In function approximation neural networks usually hyperbolic tangent sigmoid functions are used, however these two functions are considered equivalent in backpropagation networks (discussed later) especially with the use of a linear transfer function in the output layer of the network. [35] Although they are equivalent in some cases the use of one of these transfer functions, instead of the other may present advantages, depending on the problem.

The question which arises now is how the parameters which will allow us to approximate a function are defined. This is done during the process of training as described in the next section.

2.3.3 Training algorithm

As written above, one of the most appealing characteristic of neural networks is their capability to familiarize with problems by means of training and, after sufficient training, to be able to solve unknown problems of the same class. This approach is referred to as generalization.

A neural network during training, adapts to the problem and 'learns'. Theoretically a neural network could learn by developing new connections, deleting existing connections, changing connection weights, changing the threshold values of the activation function of the neurons, varying some of the neuron transfer functions, developing new neurons, or deleting existing neurons. Most of the abovementioned functions can be performed by simply changing the weights, which is the most common procedure. A connection may be deleted by assigning a zero value to its weight, while an inactive connection can by activated by changing its zero value to something else. The threshold values of activation functions can be modified through the biases. Thus, we perform any of the first four of the learning paradigms by just changing the weights.([16] p. 51-53)

The change of neuron functions is difficult to implement, not very intuitive and not exactly biologically motivated. This is why it is not used during training in this project. The development and deletion of new neurons, is similar to trying different networks with different numbers of neurons in their hidden layer, and this subject will be addressed later.

Thus, a training algorithm is needed that is going to adjust the weights to suitable values. In this case supervised learning will be used, because the neural networks are going to be trained with examples with known, inputs and outputs. In supervised learning the training set consists of input patterns as well as their correct results in the form of the precise activation of all output neurons. These patterns are given from the C program with the procedure discussed in 2.3.1 and are called training patterns. The correct results are referred as targets. Thus, for each training set that is fed into the network the output, for instance, can directly be compared with the correct solution (target) and the network weights can be changed.

Another issue that has to be regarded is when the weights are adjusted during the training procedure. There are two choices: incremental and batch training. In incremental or online training the weights and biases are updated after each input-target pattern is presented, and these patterns are presented in the neural networks as sequences. In batch or offline training the weights and biases are only updated after all of the inputs and targets are presented. The total error of all the patterns presented is calculated by means

of an error function operation and the weights are adjusted properly. These two methods have been found to perform similarly in a wide variety of problems. [34] In this study we use the batch training method for all networks.

The basic training algorithm used in multilayer feed forward networks, is called backpropagation and is an abbreviation for "backpropagation of errors". This algorithm involves the mathematical basis of an optimization technique called gradient descent.

Gradient descent is a first-order optimization algorithm and its concept will be discussed in the following paragraph. Gradient descent procedures are generally used where we want to minimize n-dimensional functions. As an example we can use a network just as the one in Fig.2-3. Let's say a training pattern is presented to this network. The training pattern consists of an input and a desired output (target). When the input is presented to the network it produces an output according to the values of its weights. This output is then compared with the target of the training pattern (the desired output) and an error function is produced. Error functions can take many forms according to the problem, but they always depend on the difference between the output of the network and the target. As someone can see this error function depends on the output of the neural network which depends on its weights. Thus, the error function is a function of the weights of the neural network. The purpose of the training of a neural network is to minimize this error. In our example suppose that all the weights are fixed except for two weights, say $w^2_{1,2}$ and $w^2_{1,2}$. We can then plot the error function in a 3d graph. The independent variables are the values of the two weights. The error function surface can probably take many forms and depends on the problem. Fig. 2-9 presents an example. In order to find the weights for which the error function minimizes (the minima of the error function) we use the gradient of the error function. Gradient is a vector whose components are the partial derivatives of the function and is always perpendicular to the contour lines, showing the direction in which the function ascends. Thus, the negative of the gradient shows the direction of descent. From a random starting point using the gradient descent algorithm, we can move from that point, with small steps, whose directions are dictated by the negative gradient, to the minima of the error function. ([16] p.61-66)

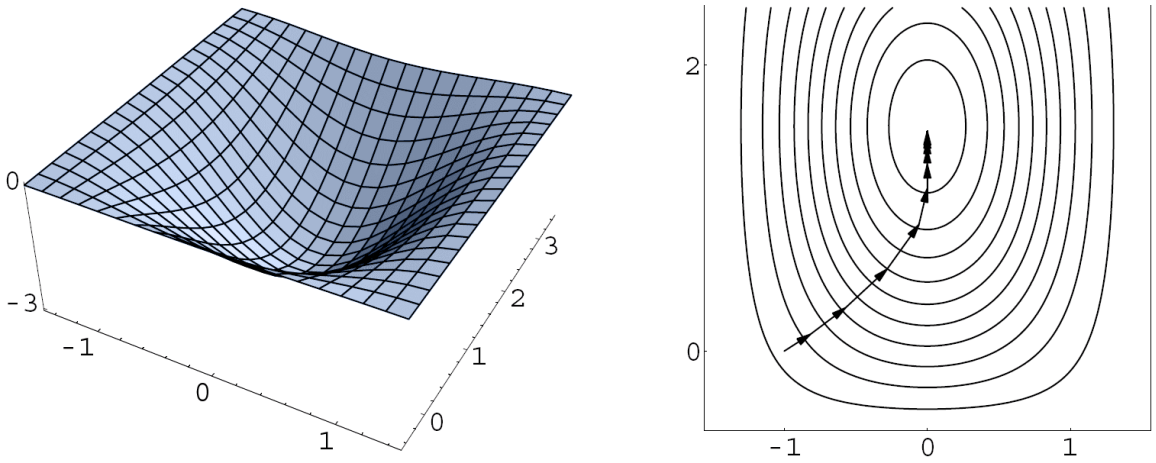

Figure 2-9 Visualization of the gradient descent on a two-dimensional error function. x,y axes represent the weights, while z axis is the value of the error function. (left) Contour plot of the error function (right)

This concept can extend to n-dimensional functions, and is it used by the backpropagation algorithm, through a rule, which is called the generalized delta rule and defines how the weights are adjusted, after the presentation of a training pattern. The detailed description and the equations of the generalized delta rule are omitted, because the point is to present the philosophy of the algorithm and not the detailed mathematical description. After all, as discussed later, different equations may be used, but the steps of the algorithm remain the same, as the principle of descent gradient.

The basic backpropagation training algorithm, involves two phases. During the first phase the input is presented and propagated forward through the network to compute the output value for each unit in the output layer. This output, the network produces based on its current weights, is then compared with the targets of the training pattern (each training pattern contains the input and the target). This comparison results in an error signal δ for each output unit, which is estimated from the generalized delta rule. The second phase involves a backward pass through the network (analogous to the initial forward pass) during which the error signal is passed to each unit in the network and the appropriate weight changes are made (dictated by the generalized delta rule again). This second backward pass allows the recursive computation of δ for each layer. The first step is to compute δ for each of the output units. This is simply the difference between the actual and desired output values times the derivative of the squashing function. We can then compute weight changes for all connections that feed into the final layer. After this is done, then compute δ's for all units in the penultimate layer. This propagates the errors back one layer, and the same process can be repeated for every layer. The backward pass has the same computational complexity as the forward pass. [11]

The whole procedure just described, constitutes one iteration of the algorithm, or as it is called one epoch. After this, the weights of the network are changed and hopefully in a direction that minimizes the error between the outputs of the network and the desired values. However, rarely if not ever the network produces the derided outputs with just one pass of this algorithm. This algorithm works with small steps towards the minima of the error function. After the two phases are completed and the weights change, the training pattern must be presented again in the network. If the difference between the outputs of the network and the targets, produce an acceptable error then the training of the network stops. Otherwise, the training procedure continues until the outputs reach the desired accuracy.

Gradient descent procedures provide good results in many cases and are promising, but not foolproof techniques. Gradient descents often converge against suboptimal minima. Every gradient descent procedure can, for example, get stuck within a local minimum. This problem is increasing proportionally to the size of the error surface, and there is no universal solution. Fig 2-9 presented a simple error surface with well defined and "easily accessible" global minima, but this is not always the case. Take for example an error surface like the one in Fig. 2-10. This error surface has many local minima and gradient descent procedures can easily get stuck in one of them, preventing it from accessing the global minima. In reality, one cannot know if the optimal minimum is reached and considers training successful, if an acceptable minimum is found. Another problem presents when the algorithm passes through a very flat surface of the error surface. In this occasion the gradient of the error function is really small and the algorithm converges really slowly, requiring a large number of training epochs.

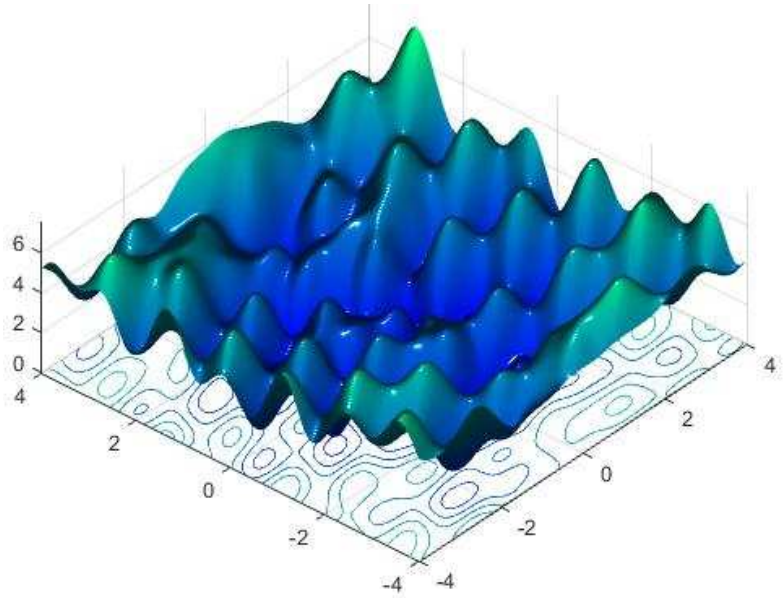

Figure 2-10. A complicated error surface with many local minima.

Neural networks created in this project do not use the basic backpropagation algorithm but a variation. There are various alternatives to the backpropagation algorithm, all of them with their own advantages and disadvantages. Most of them were created with faster convergence of the algorithm in mind. In the backpropagation algorithm there is a parameter called learning rate, which is defined by the designer to small values. A really small value of the learning rate makes the algorithm converge too slowly, requires a large number of iterations and may stuck easier in local minima.. A large value for the learning rate makes the algorithm faster, and it prevents it from becoming stuck in swallow local minima, but it renders the algorithm unstable and may overshoot the point of global minima. The choice of the learning rate is a challenge, when designing neural networks, and it was the starting point from which the variations on backpropagation begun.

The variation used in this project belongs to the numerical optimization methods and is a member of the conjugate gradient algorithms. As mentioned above, the basic backpropagation algorithm adjusts the weights in the steepest descent direction (negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions. In the training algorithm discussed up to this point, the learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms, the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size, which minimizes the error function along that line.([20] chapter 5, p.17)

Specifically the algorithm used is the scaled conjugate gradient which is too complex to explain in a few lines, but the basic idea is to combine a model-trust region approach, with a conjugate gradient approach. This algorithm was chosen based on various tests performed by Mathworks (creators of MATLAB) in which they compared the speed of their training algorithms in different problems. ([20] chapter 5, p.32-50) From these tests it seems that the scaled conjugate gradient, seem to perform well over a wide variety of problems, particularly for networks with a large number of weights, such as the networks designed in this project. Another important advantage of this algorithm is that it has relatively modest memory requirements.

The progress of these algorithms depends on the starting point of the network in the error surface. That is the initial values of all the parameters (weights) of the network before any training. Usually the weights are initialized with random small values (most commonly between -0.5, +0.5). ([9] p.296-297) If the network starts its training from different random points in the error surface (different initial parameters) then it might converge at different points which represent local or global minima. The choice of initial

weights will influence whether the net reaches a global or local minima of the error and how quickly it converges. There is no method to guarantee that this point is a global or local minima. To overcome this difficulty the network should be re-initialized and trained more times. The network that produces the smallest error, is the one that performs better. Usually five times of re–initialization are enough.

One modification of the common random weight initialization is the Nguyen-Widrow Initialization. The initialization of the weights from the inputs to the hidden units is designed to improve the ability of the hidden units to learn. This is accomplished by distributing the initial weights and biases so that, for each input pattern, it is likely that the net input to one of the hidden units will be in the range in which that hidden unit will learn more readily (in other words in the range where the derivative of its transfer function is greater). This modification has a factor of randomness and each time the net is initialized it produces different values. That network using this modification also needs to be initialized some times, but its convergence will be faster. ([9] p.296-298)

The Nguyen-Widrow Initialization is the chosen method for initialization of all neural networks in this project and is implemented in the neural network toolbox of MATLAB.

2.3.4 Network size and Generalization

One of the key issues in designing a multilayer network is determining the number of neurons to use. The number of neurons used in the output and input layer of the neural networks has been determined by the nature of the problem in 2.3.1. Another important feature of the network architecture that needs to be determined is the number of neurons in the hidden layer and the number of hidden layers.

Deciding upon the appropriate number of hidden nodes and layers is largely a matter of experience. There are no hard and fast rules for this issue. With many problems, sufficient accuracy can be obtained with one or two hidden layers and 5–10 hidden nodes in those layers. In practice, such a large number of nodes may be required that it is more efficient to go to a second hidden layer. If the number of neurons are less as compared to the complexity of the problem data then underfitting may occur. Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set. The complexity of a neural network is determined by the number of free parameters that it has (weights and biases). If a network is too complex for a given data set, that means if it has unnecessary more neurons, then it is likely to overfit and to have poor generalization (it will memorize the data). [13]

Many researcher put their best effort in analyzing the solution to the problem that how many neurons are kept in hidden layer in order to get the best result, but unfortunately no body succeed in finding the optimal formula for calculating the number of neurons that should be kept in the hidden layer so that the neural network training time can be reduced and also accuracy in determining target output can be increased.[18] Practically, it is very difficult to determine a good network topology just from the number of inputs and outputs. It depends critically on the number of training examples and the complexity of the problem.

In terms of neural networks, the simplest model is the one that contains the smallest number of free parameters (weights and biases), or, equivalently, the smallest number of neurons. To find a network that generalizes well, we need to find the simplest network that fits the data.

There are at least five different approaches which have been used to produce simple networks: growing, pruning, global searches, regularization, and early stopping. Growing methods start with no neurons in the network and then add neurons until the performance is adequate. Pruning methods start with large networks, which likely overfit, and then remove neurons (or weights) one at a time until the performance degrades significantly. Various forms of this technique have been called optimal brain damage and optimal brain surgeon. Global searches, such as genetic algorithms, search the space of all possible network architectures to locate the simplest model that explains the data. The other two techniques are called regularization and early stopping and are discussed later in this section.

In this project we are not interested in the optimal neural network which approaches the problem. The purpose is to collect evidence on whether neural networks are able to represent relationships that will enable to deduce the phases of observed structure factors. So pruning or growing techniques will not prove particularly useful. A number of neural networks with different sizes are tested ranging from a few neurons to some hundred neurons in the hidden layer. In those structures the technique of early stopping is used, instead of pruning/growing techniques, to avoid overfitting. Also architectures with two hidden layers are tested. Multiple hidden layers are usually used in applications where accuracy is important and the training time is not an issue. The drawback of using multiple hidden layers in the neural network is that they are more prone to fall in bad local minima. [13]

As mentioned earlier one of the problems that occurs during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations. The problem is illustrated in Fig.2-11.

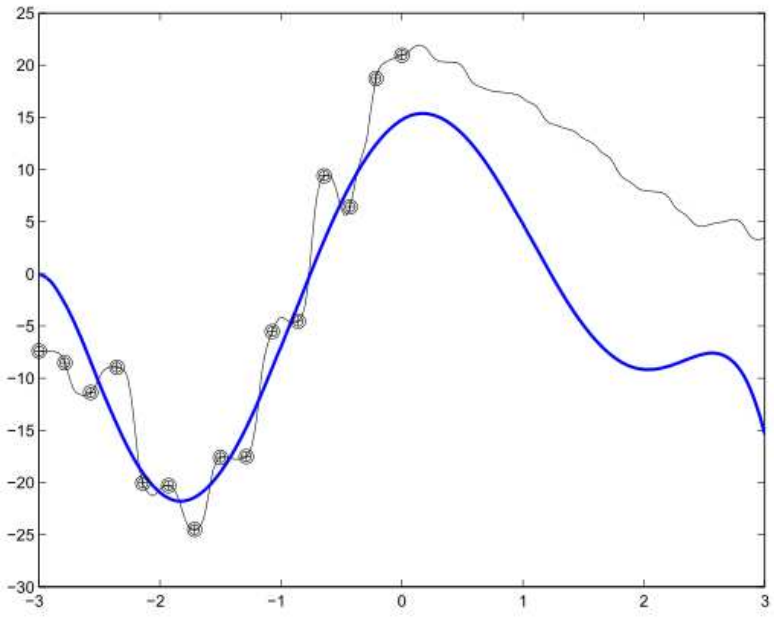

Figure 2-11. Overfitting and poor extrapolation problems

In Fig. 2-11, the function under approximation is represented with the blue line. The black dots represent the training data with which a neural network has been trained, and it seems that they are noisy, because they are not on the blue line. The black line represents the response of the neural network. As someone can notice, the neural network perfectly represents its training pattern but it cannot generalize because it does not follow the blue line as it should. That happens in the range of values [-3, 0] of the independent variable (horizontal axis). This is probably happening because the network has too many neurons and can represent a more complex function than the one in blue line. A smaller network would not have enough power to overfit the data. Instead of removing neurons from the network we could train it with less iterations.

In Fig.2-11 the problem of extrapolation is also depicted. This can be shown in the range of values [0, 3], where we can see that the neural network response is not similar to the function. That happens because there are no training data for this area. In this case the network is extrapolating beyond the range of the input data.

The neural network in Fig.2-11 is trained again with the proper number of iterations and its response is shown again in Fig.2-11.

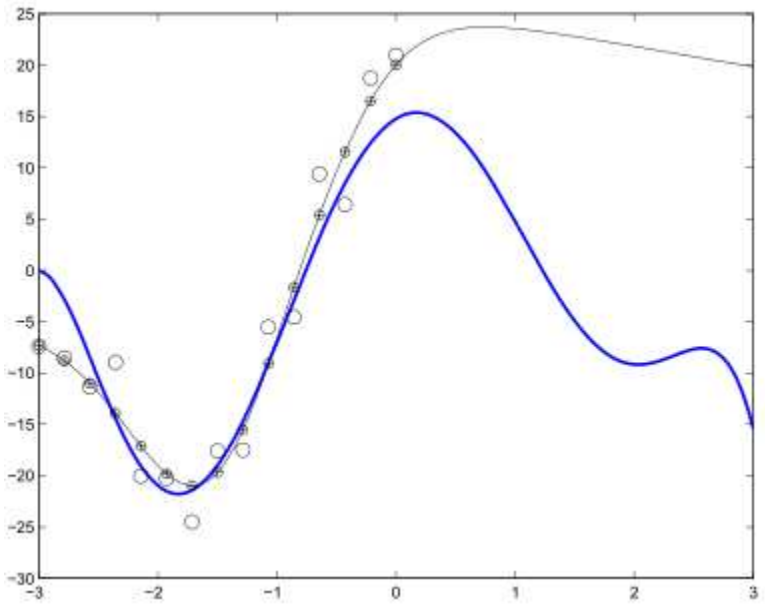

Figure 2-12. Good generalization and poor extrapolation

In this case it is noticeable that the neural network generalizes well in the range [-3, 0], where adequate training data is present, but the extrapolation problem still remains, in the range [0, 3]. This is understandable, since the network has been provided with no information about the characteristics of the function in this range. The network response outside this range will be unpredictable. This is why it is important to have training data for all regions of the input space where the network will be used. In our case the only way to ensure that is by using a large number of training patterns. Also if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. The networks used in this study have a really large numbers of free parameters and although it is easy to collect more data and increase the size of the training set with the program in C, training a large network like this with a vast amount of training patterns may be impractical. That is why the early stopping method has been chosen.

As has been said, the strategy of early stopping is used to prevent overfiting of the data. The early stopping procedure is simpler than regularization mentioned before and it provides almost the same results. ([12] chapter 13) That is why it has been selected as the method to prevent overfiting.

In the early stopping method we divide the data into two sets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases using the algorithm mentioned at 2.3.3. The second subset is the validation set. The data in the validation set are not used as a training pattern and represents data

unknown to the network. The error on the validation set is monitored during the training process. The validation error will normally decrease during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set will typically begin to rise. When the validation error increases for a specified number of iterations (called validation checks), the training is stopped, and the weights and biases at the minimum of the validation error are returned.

Except of overfiting the early stopping method also prevent us from training a neural network aimlessly. In a condition when the error on the validation data rises, even if the network has not memorized the training set, that usually means that the network is unable to represent a solution to the problem and something different should be tried.

Except of these two sets in this project we also use a third set, called the test set. The test set error is not used during the training, but it is used to compare different models. It is useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this may indicate a poor division of the data set. ([20] p.55-56)

2.3.5 Output format of Neural Network experiments and representation of results

Neural networks are implemented in MATLAB using its neural network toolbox. Every network is given a set of training patterns which are created with the C program described in 2.3.1. From the outputs of this program input data for the neural network are created using the absolute values of the structure factors or the absolute values of the unitary and normalized factors.

The outputs which represent the phases of the structure factors can take various forms depending on the experiment. Specifically three types are used:

Structure factor form: The target outputs are defined as the signed structure factors.

Bipolar form: In bipolar form the target output for each class of structure factor can take the values +1 when the corresponding structure factor is positive (phase=0) and -1 when it is negative (phase=π). Remember that structure factors from centrosymmetric structures are real numbers.

Binary form: In binary form the target outputs are represented with 0 when the corresponding structure factors are positive and 1 when they are negative.

The structure factor form obviously is the most natural representation, but it may be easier for the neural networks to learn relationships with bipolar or binary representation, because structure factor form demands from the network to learn the production of continuous values, while the other forms have only two possible options.

It is not necessary for the networks to produce outputs with high accuracy, as long as these outputs can be classified as positive or negative phases. For example it is not necessary for the binary form outputs to be exactly zero or one. If a threshold that separates the two phases is created, that is enough. A network that assigns to its outputs values larger than 0,5 for positive phases and values smaller than 0,5 to negative values can be considered successful (of course 0,5 can be some other value). The same holds for the structure form where every positive output can be considered as a positive phase and every negative value as a negative phase, regardless of the absolute value of the output.

For most of the experiments three graphs are created. The first one is the performance graph which shows how the error function minimizes during training. This function, also called performance function, is the one the network wishes to minimize with its training algorithm. The performance function used in all networks in this study, is the mean squared error function (mse):

$$mse = \frac{1}{n}\sum_{i=1}^{n}(t_i - o_i)^2 \tag{2.12}$$

where t_i is the target value for the ith output of the neural network and o_i is its actual output. The number of outputs is n.

An example of a performance function graph (from now on, simply called performance) is shown in Fig.2-13.

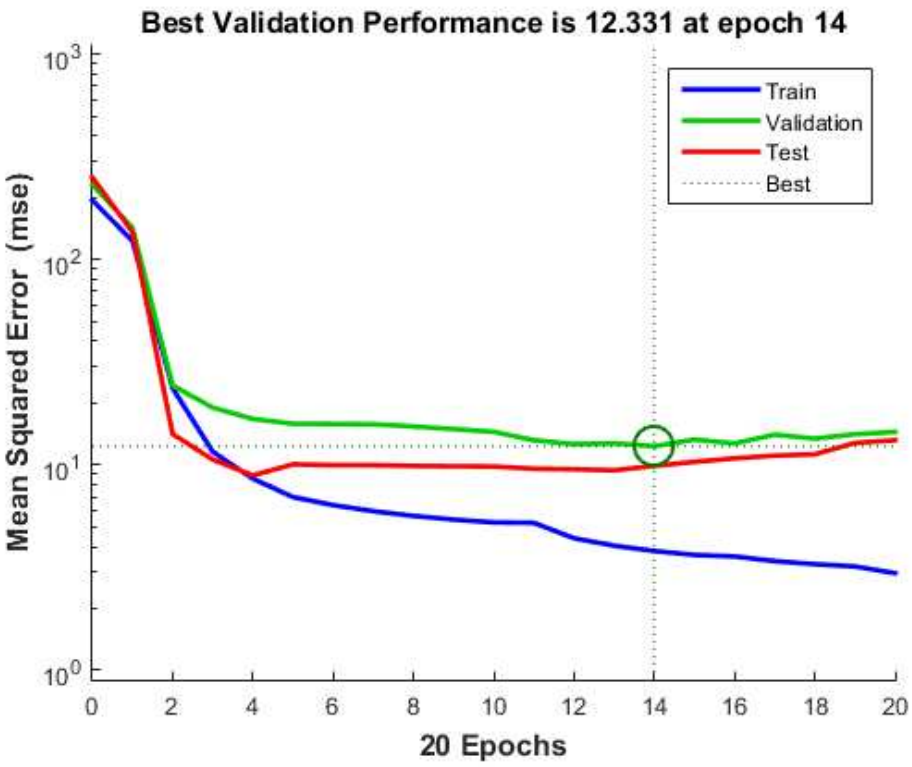

Figure 2-13. An example performance plot

In the plot in Fig.2-13, someone can see how the performance is altered after each iteration of the training algorithm. This is a typical result where the performance of the training set decreases continuously, while the validation set decreases up to the point where overfiting occurs. In this graph we can also see that maybe the validation set is not so well defined (poor division of data sets) because the curves of the test and he validation sets have important differences according to where their minimums are located. This network was trained for 20 epochs, but after the 14th epoch (iteration) the validation performance started to increase (remember: a good performance is a performance with small value). After six validation checks the network stops its training and returns to the state where the minimum validation performance was observed. This is indicated by the green circle.

The second plot created from the experiments is a linear regression plot of targets relative to outputs of the neural. Fig. 2-14 shows an example regression plot.

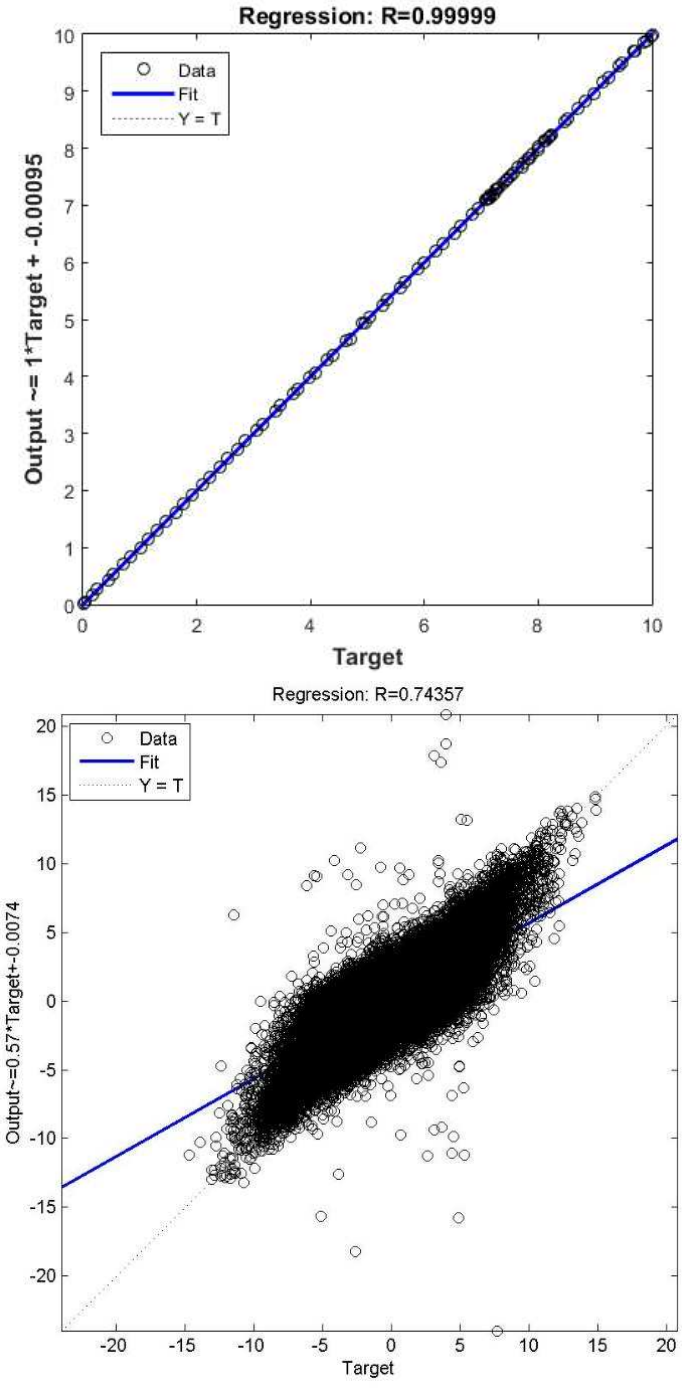

Figure 2-14. Examples of regression plots

In these plots someone can see how the targets are related to the actual outputs. In the upper plot, when the target value of an output unit is 8, the plot shows that the network produces a value really close to 8 for this output unit. The R value at the top of the plot is the correlation coefficient and has a value of one when the targets match the outputs

perfectly. When R is one a perfect linear fit is accomplished. The perfect linear fit is represented by a diagonal dashed line in the plot. The blue line represents the best linear fit for the network's data. The closer this line is to the dashed line the better the network performs, and the targets match to the outputs. An R value of 0 represents uncorrelated data and a negative value of R represent data correlated inversely.

The third graph which is created is the train state graph. An example is shown in Fig.2-15

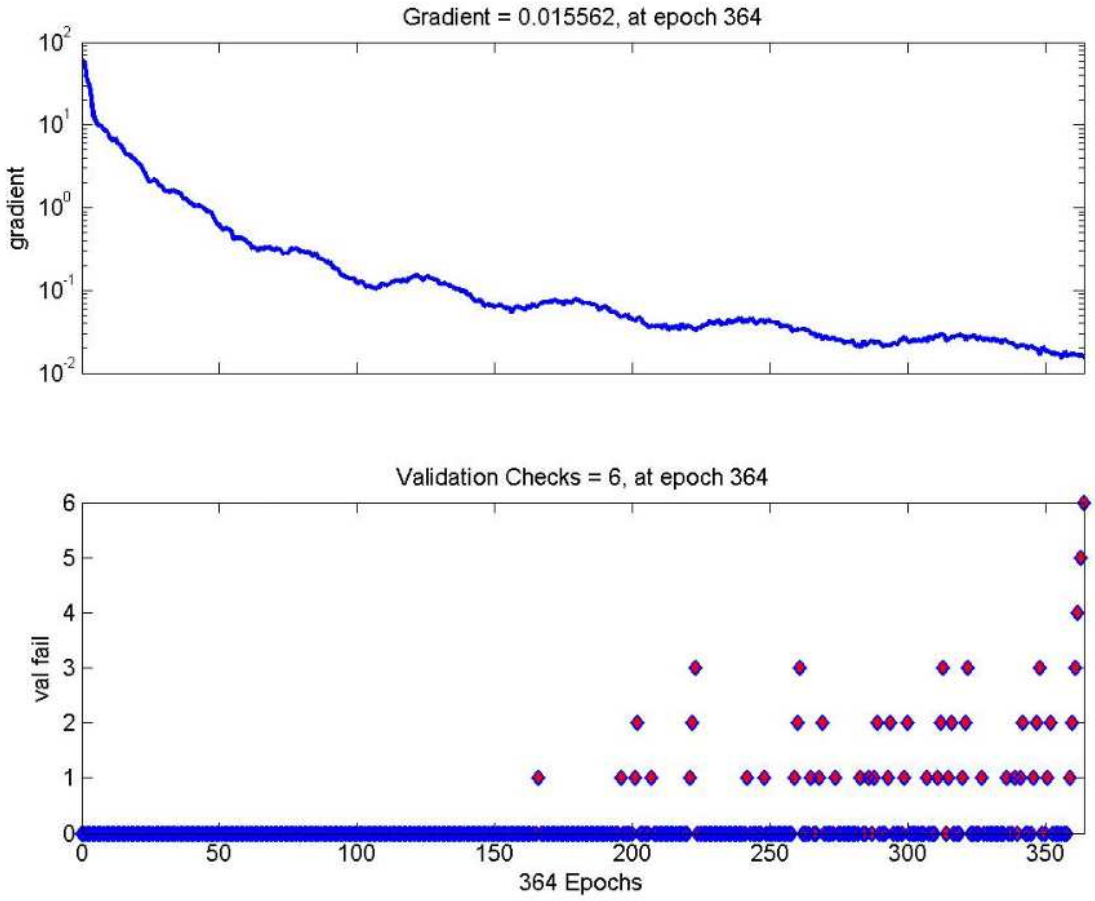

Figure 2-15. Example train state graph

This graph gives information on the training process of the neural network. The upper plot shows how the value of the gradient changes during training, and the plot below shows events of validation checks. A validation check occurs, when the performance (error) on the validation data set increases after a training epoch, as mentioned in paragraph 2.3.4. Validation checks are represented by red diamonds with blue outline in the graph. In the example in Fig. 2-15, a network is trained and the performance of the validation data set (from now on, called validation performance) is minimized after each training cycle, until epoch 165 (approximately- as someone can see from the graph).

When a diamonds lies in the zero line of the vertical axis ("val fail" in the vertical axis stands for validation fail) it means that after this training epoch the validation performance decreases. At epoch 165 a diamond can be seen not at the zero line, but at the line of the vertical axis representing the value '1'. A diamond in this line shows that after the completion of the 165 training cycle, the validation performance increased. At this point the minimum validation performance achieved so far is that after the completion of the 164 training epoch. The training continues with one more training epoch. After the completion of the 166 training epoch, the validation performance is lower than that of epoch 164. In this case the training continues normally. However, if the validation performance did not decrease we would have a second validation check event, represented by a diamond at the line of the vertical axis representing the value '2'. As it should be clear so far the values in the vertical axis represent the number of successive iterations (or training cycles – training epochs) that the validation performance fails to decrease. When a maximum number of such successive iterations is reached the training procedure stops and we return to that state of the neural network where the minimum validation performance was reached. This number of maximum successive iterations without validation performance decrease is a parameter whose value is determined by the designer of the neural network. In the example of Fig. 2-15 the number of maximum successive validation checks is set to 6. As someone can see from the graph when the validation checks reach a maximum of 6 the training process stops at epoch 364. The best validation performance was at epoch 364-6=358.

These three plots give enough information about the training process of a neural network and its performance and allow the user to evaluate the network and come to useful conclusions.

There are four conditions for a training process to stop. The first one is to reach the maximum number of validation checks (early stopping). The second is to reach a defined performance goal. A zero performance goal is used to all networks. A neural network which approximates a function almost never reaches a zero performance goal of course. The zero value was set because a certain goal is not defined, the networks are trained until they reach their maximum potential. The third occasion for the training to stop, is when the gradient reaches a really small value ($e^{-6} \sim 0.002478$). In this case the algorithm probably is in a very flat surface of the error function and further training will not lead anywhere after a reasonable number of iterations. In the abovementioned graphs someone can see why a training process may have stopped. The last condition is when the training has reached a maximum number of training iterations (epochs).

2.3.6 The example of a satisfactory network

Now that the tools for the evaluation of a network have been given a question arises: how the results of an efficient network would look like? As mentioned earlier, it is not necessary for the networks to produce outputs with high accuracy, as long as these outputs can be classified as positive or negative phases with the use of some sort of threshold. In this section an example of a satisfactory result will be presented. Suppose that a network, with binary representation of the outputs, after training produced the regression plot in Fig.2-16 for inputs with which it has not been trained.

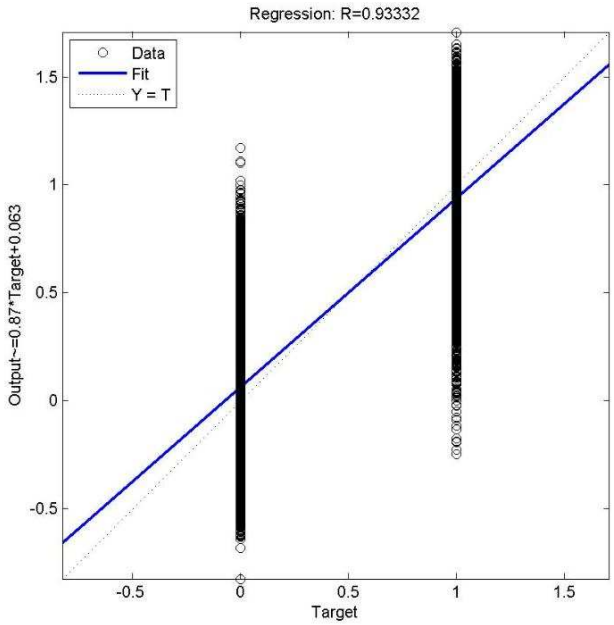

Figure 2-16. Regression plot of a successful neural network.

From this plot it can be observed that the blue line (best linear fit) is close enough to the dashed line (perfect linear fit) and the R value is close to one. This graph was created after the presentation of 4000 unknown patterns. All the networks have 225 outputs as discussed in 2.3.1. That means that this plot has 4000 x 225=900000 points. It can be observed that many of the points that should be one take zero or even negative values, and many of the values that should be 0 take values close to 1. A threshold therefore may be difficult to be defined and because the points on the plot are really dense we can make only assumptions about their distribution. For this reason a histogram should be created. Two histograms are presented in Fig.2-17. These histograms have been created from the same data presented in Fig.2-16. The red histogram represents the outputs of the network that should be one, while the blue histogram represents the values of the outputs that should be zero.

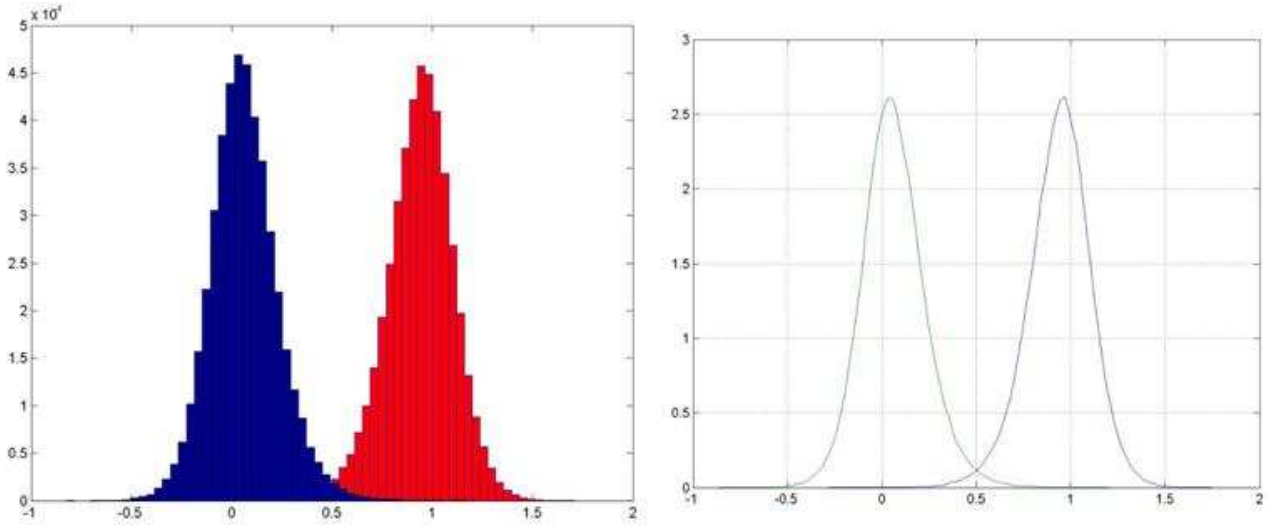

Figure 2-17 (left). Histograms of the outputs of a successful neural network

Figure 2-18 (right). Probability density function of a successful neural network

In these histograms we see that these points follow a Gaussian bell shaped distribution, and so we can plot a probability density function using Gaussian kernel density estimation as the one in Fig. 2-18. The blue line represents the probability density function of the outputs that should be one. In this figure it is obvious that a threshold of 0.5 would assign the correct phase to almost all the structure factors, with a small number of false signs. This is the reason this network would be considered as a really good network for the purposes of this project. The performance of this network (mse) was 0.02.

The data presented in this section do not come from a properly trained and tested neural network, but from a network trained with almost identical training and validation sets for reasons of demonstration only. With these graphs in mind a comparison between the results obtained from the experiments and what is the desired outcome is easier.

Chapter 3: Experiments and results

In chapter 2, the philosophy of the neural networks and the training algorithm has been explained. In this chapter, details about the architecture and the procedure of each experiment will be presented, along with the results these experiments produced.

3.1 Abbreviated notation of neural networks.

In this chapter an abbreviated notation of neural networks is being used, instead of the one used so far. This notation is used by MATLAB and is more practical for complex networks with a large amount of neurons and connections. It would be useful to sacrifice a few lines to describe it. In Fig. 3-1 the notation used so far (a) and the abbreviated notation (b) are illustrated for the neural network of Fig.1-16 in the first chapter.

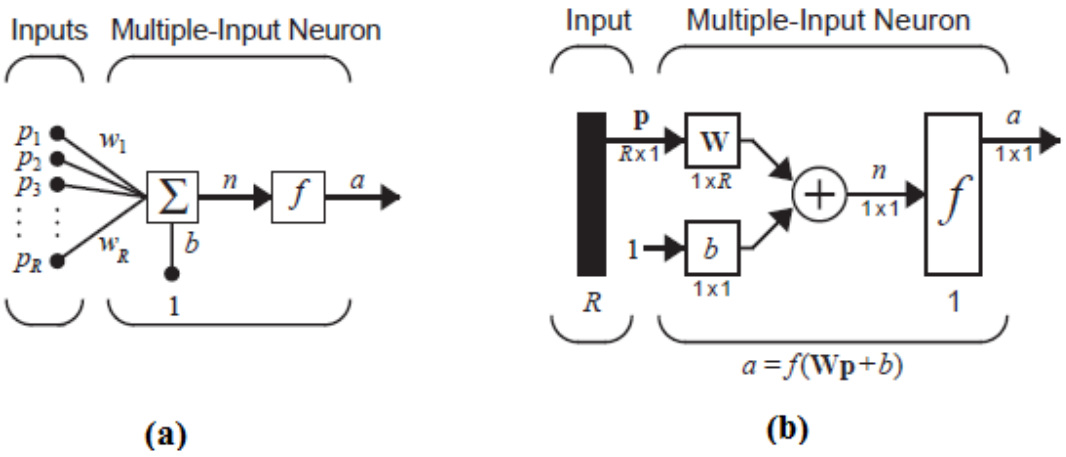

Figure 3-1. (a) Notation with all the connections drawn. (b) Abbreviated notation.

As shown in Figure 3-1, the input vector **p** is represented by the solid vertical bar at the left. The dimensions of **p** are displayed below the variable as Rx1 , indicating that the input is a single vector of elements. These inputs go to the weight matrix **W**, which has columns but only one row in this single neuron case. If there were S neurons in the layer the dimension of **W** would be SxR. A constant 1 enters the neuron as an input and is multiplied by a scalar bias b. In the case of the S neurons the bias would be a vector **b** with S elements. The net input to the transfer function f is n, which is the sum of the bias b and the product **Wp**. The neuron's output is a scalar in this case. If we had more than one neuron, the network output would be a vector.

After this example there should be no confusion with the representation of the neural networks that follow.

3.2 Experiments with structure factors as input data

The first neural networks that have been tested were networks whose input data were the absolute values of structure factors, produced by random structures of the two dimensional p2 space group as explained in section 2.3.1.

3.2.1 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data.

Networks of this type are shown in Fig. 3-1, as they are presented in MATLAB.

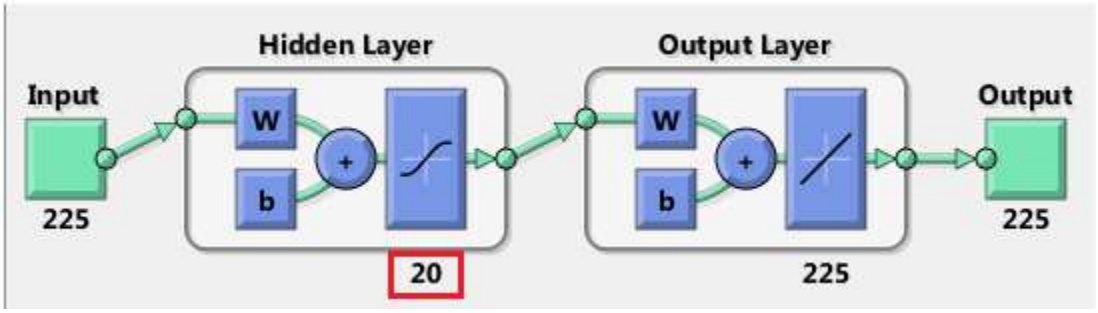

Figure 3-2. Hyperbolic tangent sigmoid transfer function networks

As someone can notice this network has 225 inputs, one for each structure factor, 225 outputs which correspond to the phases of these structures and 225 output neurons in the output layer (output layers produce the outputs of the network, thus outputs=neurons in the output layer). In this figure the hidden layer has 20 neurons, and is drawn in a red box to show that this number is variable. Various networks with different number of neurons in the hidden layer are tested and evaluated.

The target outputs which represent the phases of the structure factors have the bipolar form ±1, with 1 representing a positive value (phase=0), and -1 a negative value (phase=π) as mentioned in section 2.3.1. From now on when hyperbolic tangent sigmoid transfer functions are used, bipolar outputs will be considered the default target output, unless something different is mentioned.

The networks have two layers, one hidden layer with a hyperbolic tangent sigmoid transfer function and one output layer with a linear transfer function.

These networks were trained with different numbers of training pairs. Specifically each one of them was trained in cycles, where 2000, 4000, 6000, and 10000 structure patterns were presented to them. In every occasion 60% of these patterns were allocated to the training set, 20% of them to the validation set and 20% of them to the test set. The patterns of each set are chosen randomly. These sets are used for early stopping procedures and evaluation of the networks. By training the networks with different training pattern sizes, an estimation of how this size affects the performance of the network can be made.

The number of neurons in the hidden layer ranges from 5 to 600 neurons, with large intervals. Specifically networks with 5, 10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 400, 500 and 600 neurons in the hidden layer were tested. As mentioned in the previous chapter the purpose in not to define the optimum number of neurons for a network, but to see if this kind of network can represent a solution to the phase problem. By assigning a range of different number of neurons in the hidden layer an estimation of how this parameter affects the performance of these networks can be made. Larger networks can represent more complex relationships, while smaller networks are trained easier, need fewer training pairs, and generalize better (if they have enough power to represent the relationships). The consideration of smaller intervals would create many more networks to train which is a time consuming procedure especially in the range of 200-600 neurons.

One of the main factors that make these networks and the networks that follow really complex is the large number of inputs and outputs. For example a network with 20 neurons in the hidden layer has 9245 adjustable weight parameters. This number derives from the fact that each input is connected to each neuron in the hidden layer, so there are 225X20=4500 connections there. Each of these neurons has a bias, so in the first layer there are 4500+20=4520 connections. These neurons are connected with the 225 neurons of the output layer, so there are 4500 connections there also. The output neurons have also biases, which make the total number of connections to the network equal to 4520+4500+225=9245. The large number of these connections makes the error function surface really complex, and this creates problems to the training procedure which is more likely to get stuck at local minima.

All of these networks were re-initialized and trained three times. For the initialization of weights the Nguyen-Widrow method was used. This strategy helps in avoiding bad local minima in the error surface. Also, a maximum number of 4000 epochs and 6 validation checks was defined for the training. After each re-initialization the training patterns are divided again in three subsets in a random fashion (60% training set, 20% validation set, 20% test set).

Thus, there were trained:

(number of different sizes of structure patterns) x

(number of different sizes of neurons in the hidden layer) x

(number of re-initializations) = 4 x 15 x 3= 180 networks.

These networks produced similar results to those depicted in Fig. 3-3. Fig 3-3, shows the performance plot of the second re-initialization of a network with 40 neurons in the hidden layer, trained with 6000 patterns (patterns during the remainder of the text may also be referred as samples). At this point it should be noted that the number of re-initialization is not an important factor and from now on, it will be omitted.

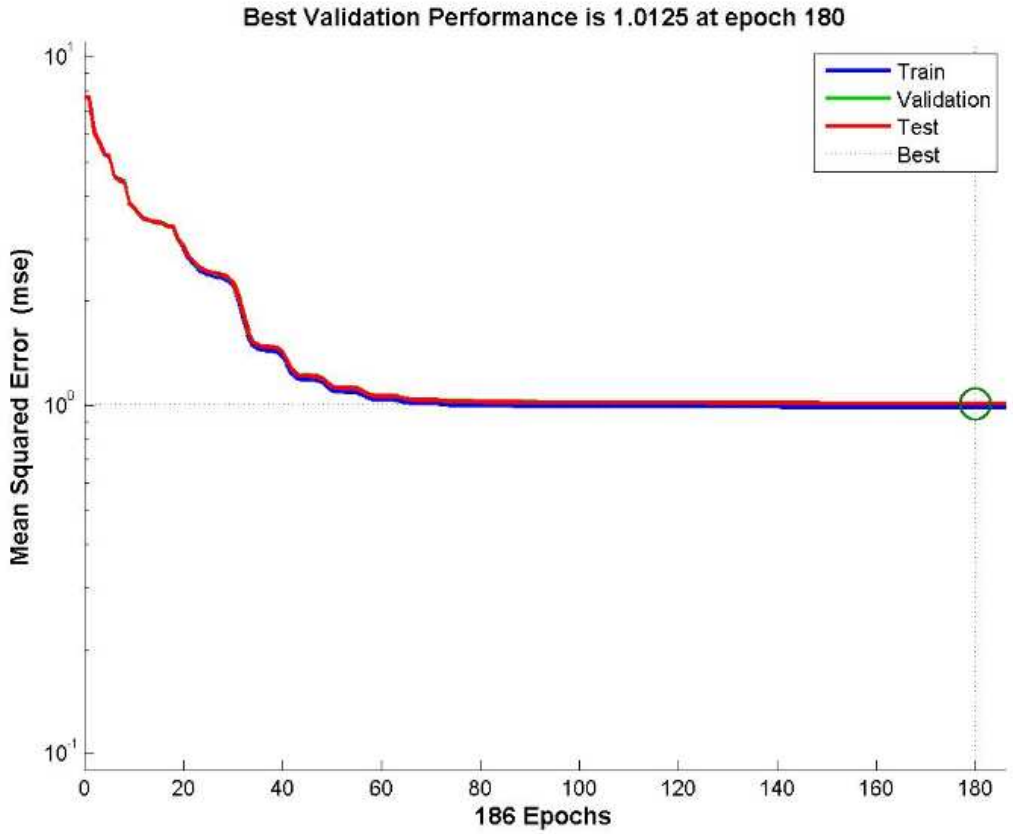

Figure 3-3. Performance plot of the second re-initialization of a network with 40 neurons in the hidden layer, trained with 6000 patterns

From Fig. 3-3, it should be observed that the training, validation and test sets have identical performance plots. This is an indication of excellent data division. The validation set is representative of the problem (ensured through the test set) and early stopping happens at the time the network no longer converges to the solution of the

problem. This is happening because a large number of patterns was presented to the network, and an adequate percentage of them was allocated to the test and validation sets. Similar plots are produced from experiments with 2000, 4000 and 10000 samples.

The regression plot for the same network is presented in Fig. 3-4 and is representative of the results of all the networks been trained. In this plot the results for each subset (training, validation, test) is drawn separately and the result of the whole set is plotted at the lower right part under the title 'Overall'.

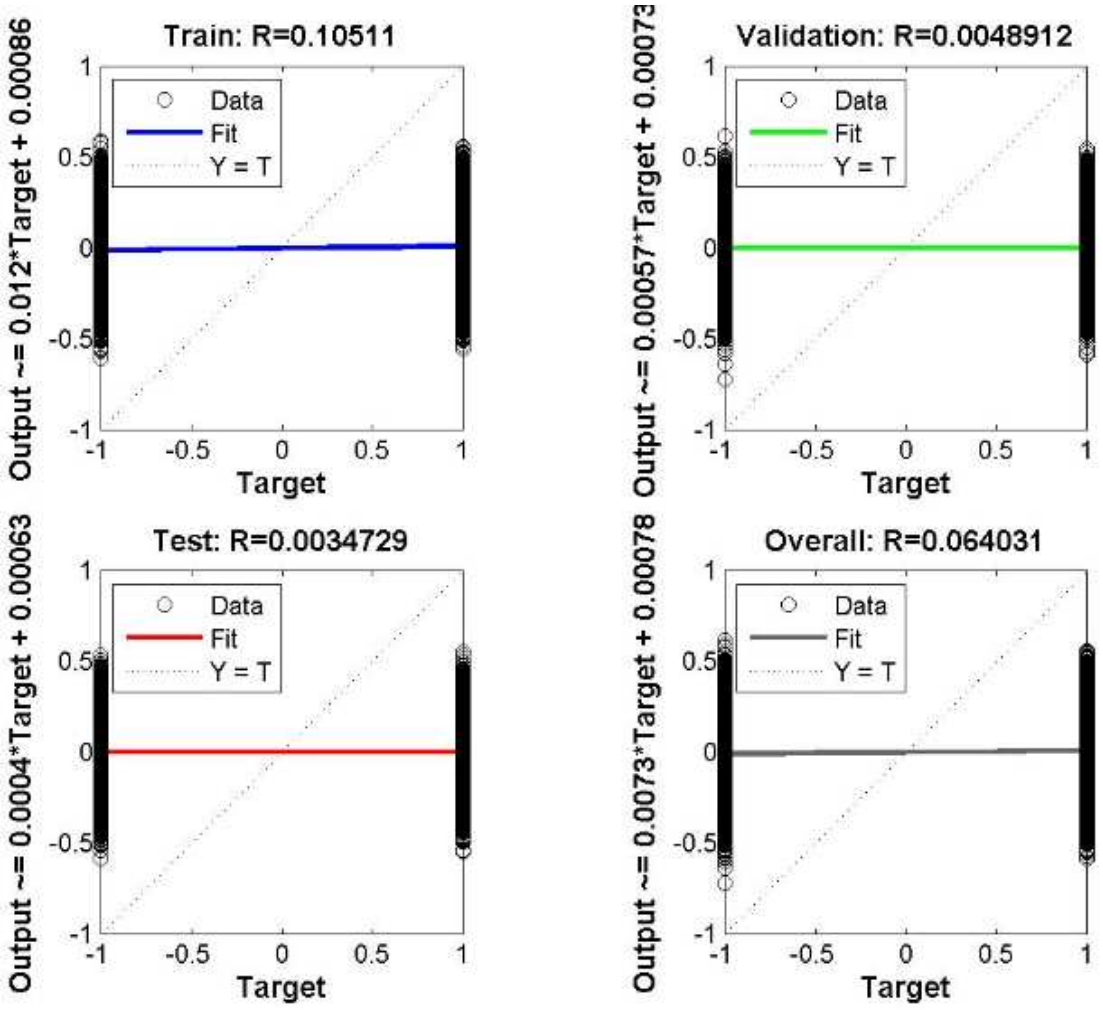

Figure 3-4. Regression plots of the second re-initialization of a network with 40 neurons in the hidden layer, trained with 6000 patterns

It is clear that network training did not stop because the network overfitted the training data, as the best linear fit for the training data set is far from the dashed diagonal line, but because the validation set's error performance was not decreased after the 180th epoch (Fig.3-3). That means the network is not able to approach a solution to the problem, and

cannot map the data in a reasonable manner that provides generalization. The network's performance is really poor and this is indicated by the fact that the error function has a value of ~1, in a target space of -1 and 1. That means, in average values that should be 1, are 0 and values that should be -1 are also 0. This is more clearly shown in Fig. 3-4. This figure shows in a more direct way that regardless of the value of the target for each output neuron, the actual output of the network is somewhere between 0.5 and -0.5 with values similarly distributed for both targets around 0. That means that a threshold which separates negative from positive phases cannot be set; practically meaning the network cannot give information on the phases of the structure factors. This is further illustrated by the histogram in Fig.3-5. The data of this histogram are the same as those used for the regression plot. The blue shape is for those outputs that should be -1, and the red for those outputs that should be 1. These histograms overlap each other with only a slight shift which is not enough to separate phases.

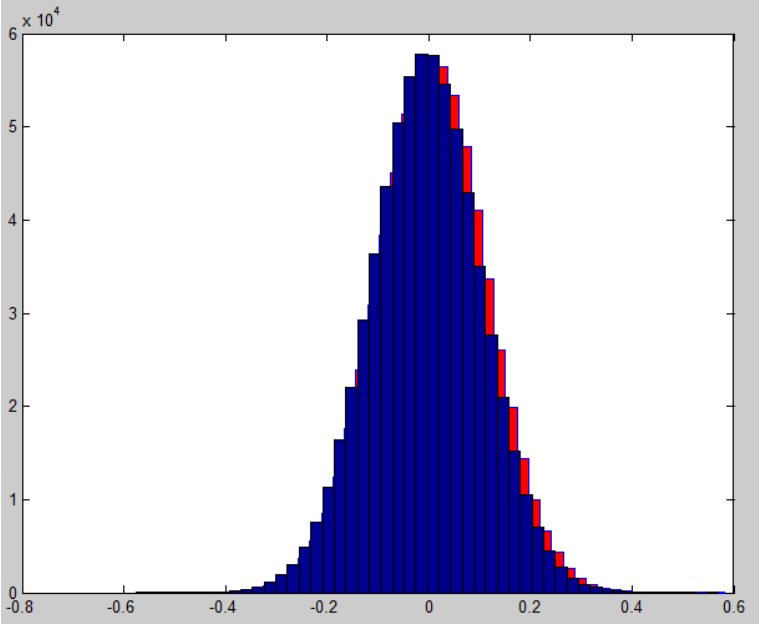

Figure 3-5. Histogram of the outputs of the second re-initialization of a network with 40 neurons in the hidden layer, trained with 6000 patterns

When someone compares Fig. 3-4 with Fig. 2-16 in section 2.3.6 which shows an example of a satisfactory network (unlike the one in Fig.3-4) a question may arise: Why the network in paragraph 2.3.6 seems to work while the network in this paragraph does not? As mentioned in paragraph 2.3.6 the data presented in Fig. 2-16 do not come from a properly trained and tested neural network. This network was trained with 4000 patterns but only 1000 of them were different. This means that 1000 different patterns were presented 4 times (at each training epoch) at the network. For this reason, when the data was separated in training, validation and test sets, the validation and test sets contained

information almost identical to that of the training set. As a result, after a training epoch the performance (error) in the training set decreases (even slightly, the training performance always decreases after a training epoch) and so does the performance in the validation set (since the data in both sets is similar). This leads to many training epochs, as the early stopping technique is not triggered because the algorithm (falsely) supposes that the network generalizes well (although it actually memorizes the patterns). Many training epochs in turn lead to the memorization of these 1000 different patterns and the results in Fig. 2-16 are produced. All this procedure mentioned above was just a 'trick' to produce an example of how a satisfactory network would look like in our results. Of course when this network is tested with a pattern different than the 1000 patterns it was trained with, it does not produce good results. This fact shows that memorization of some patterns in not helpful but generalization is needed. On the other hand the network of Fig. 3-4 was trained with 6000 different patterns. Even if the training, validation and test sets were identical as before the memorization of 6000 instead of 1000 patterns is more difficult and would require more training epochs. In addition this network is properly trained. This means its training, validation and test sets do not contain the same patterns. At the first few epochs the training and validation performance decrease and training continues without problems. After a few epochs however, the validation performance begins to increase and the early stopping technique stops the training procedure before the network begins to reproduce the training pattern (the R-value in Fig. 3-4 is really small even for the training set). At this point the early stopping technique prevents us from wasting time training a network which cannot generalize. It is neither necessary nor useful for a network to fit the training data as long as the validation performance in not decreasing. If the network continued its training process, after a relatively large number of training iterations it could reproduce its training data set (the training R value would be significantly higher) but that would be useless because the network would just memorize its training set (the situation we described before for the network of section 2.3.6). The reason why the network of Fig 3-4 has a small training data set R-value is that it is not given enough training iterations (as they are not necessary).

Networks with a higher numbers of neurons in the hidden layer performed similarly. The only difference was that their outputs were more concentrated near 0 values, the ranges of outputs been smaller, and their histograms been narrower. In Fig. 3-6 the regression plot of a network with 600 neurons is shown (trained with 10000 samples).

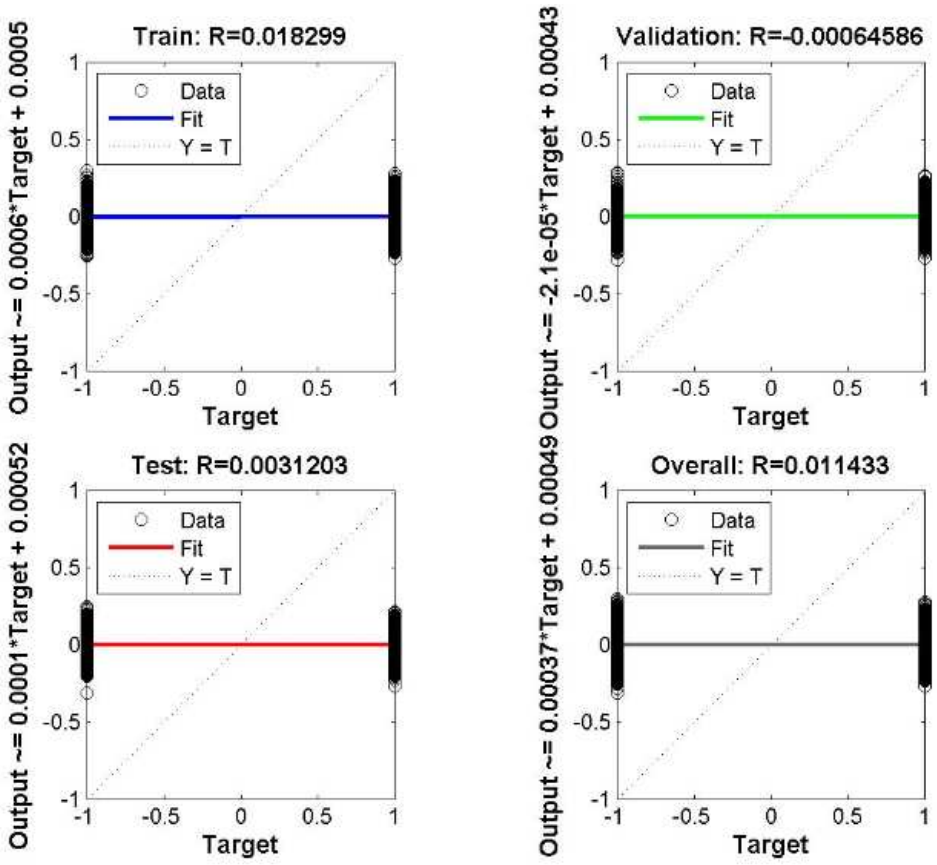

Figure 3-6. Regression plots of a network with 600 neurons in the hidden layer, trained with 10000 patterns

The results for all the 180 neural networks can be summarized in a plot of the best error function performance they achieved for their validation sets. This is shown in Fig. 3-7. This plot represents discrete data points. A continuous line was used, however to connect those points, in order to render the plot more visible and interpretable. The horizontal axis is the number of neurons in the hidden layer and is presented 3 times for each neuron (one for each re-initialization). Smaller values in this plot mean better performances.

In Fig.3-7, someone can notice that using more training patterns has a positive result in the network performance regardless the number of neurons, which was something expected. In addition, neural networks which have over 100 neurons behave better especially when a smaller number of training patterns is available (blue and red lines). One important characteristic is that the performance when more neurons are used tends to a value of one, but never below it. This is an indication that more neurons will not help in this situation, and probably this is the best performance this neural network architecture can achieve given this form of data.

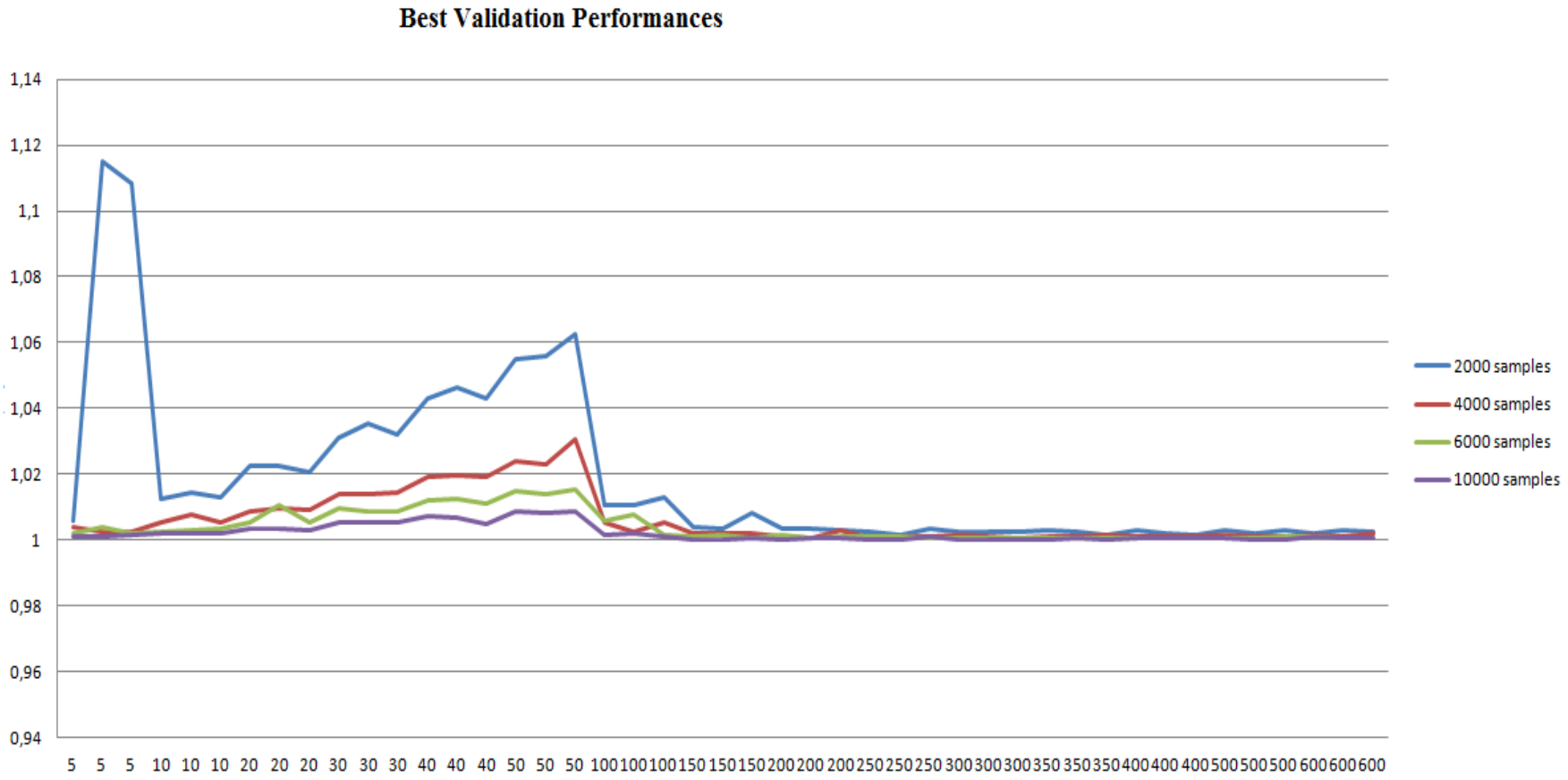

Figure 3-7. Best validation performances of networks with hyperbolic tangent sigmoid transfer function and bipolar output data, using structure factors as inputs.

The R values (correlation coefficient) for each of the training, test and validation sets can also be plotted in a similar manner. For the figures to be interpretable in Fig. 3-8, the R values have been plotted in different graphs, according to the number of training samples used. The horizontal axis is, again the ascending number of neurons in the hidden layer. Blue lines represent the training set's R value, green lines the corresponding values for the test set, and red lines the validation set. Only the general shape is of interest and not the exact values in this plot. The greater the R value, the better the targets match to the outputs.

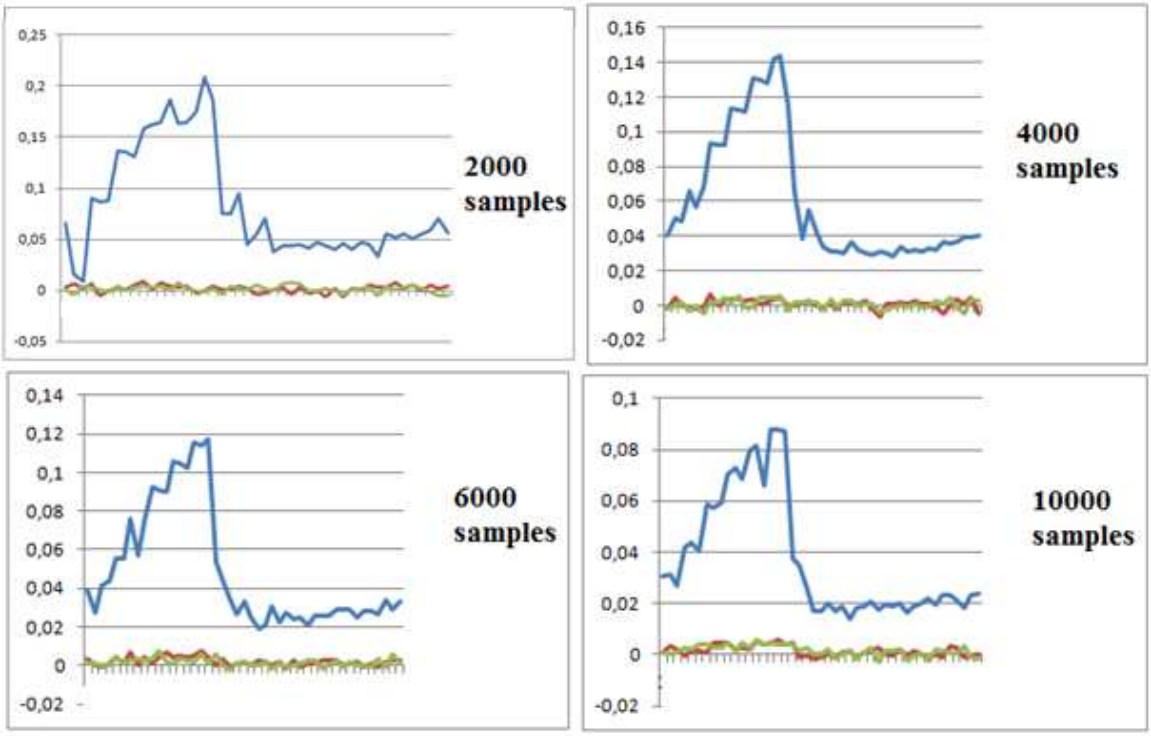

Figure 3-8. R values plots of hyperbolic tangent transfer function neural networks. Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

The information someone can get from Fig. 3-8 is that the training set's R values are higher for all cases. All R values are really small to produce useful networks but here just a comparison is been made. In all networks the R values for the validation and the test sets are of the same magnitude, which is something someone wants, it indicates good data division. Actually if we expand the plots so that details can be distinguished, the R values in the validation sets are slightly higher which is something expected, as the network decided to stop its training based on this set, and in a way it is 'tailored' to this set. This is not a problem in this case because as we mentioned many times, the data division is good, and the R value differences are really small.

From now on mainly the validation error function (mse) performance will be used for the comparison of neural networks, because it seems to produce more concise results.

As far as the number of training epochs is concerned the only thing that was observed was that the epoch's mean value of all networks been trained with a particular size of samples, is increased with the sample size which is a logical and expected result (Fig.3-9).

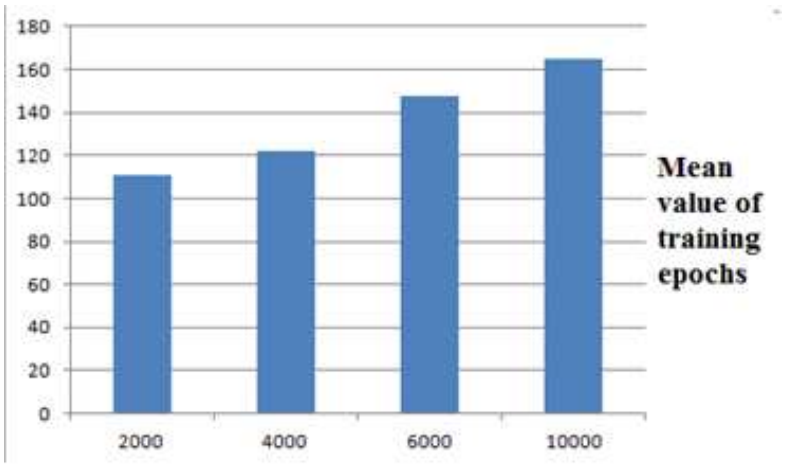

Figure 3-9. Mean value of training epochs with respect to the number of training patterns used

Such a detailed analysis, as the one made in this section, is not necessary for the purposes of this project, especially when the performances of the networks examined are poor (as the ones obtained so far). After all, the values compared for these networks were slightly different. The above analysis was made just to illuminate some characteristics of the behavior of the networks.

In the cases of the following experiments a detailed analysis will be omitted if it is not necessary.

3.2.2 Networks with logistic sigmoid transfer function and binary output data.

Networks of this type are shown in Fig. 3-10, as they are presented in MATLAB.

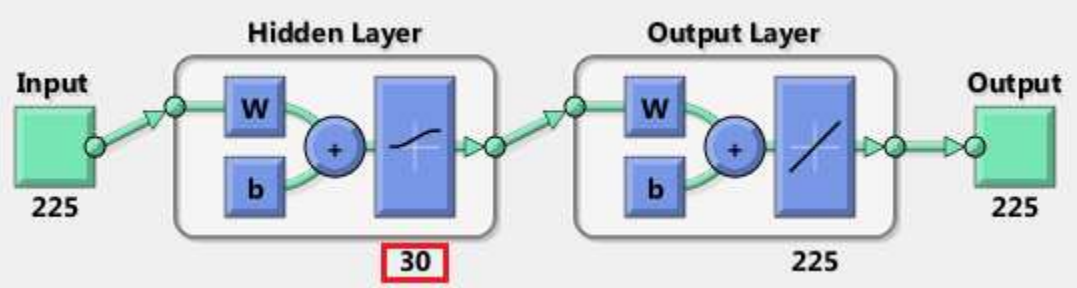

Figure 3-10 Logistic sigmoid transfer function networks

The only difference between these networks and the hyperbolic tangent sigmoid transfer function networks presented in the previous section is the use of a logistic sigmoid transfer function in the neurons of the hidden layer. Everything said about the hyperbolic tangent sigmoid transfer function networks is valid for the networks in this section too, with the exception of the output format.

In these networks, a binary representation of outputs for the phases is more appropriate, because the range of their transfer function in the hidden layer is between 0 and 1. Of course they can represent any value, because of the output linear layer, but their training is considered easier if the binary form is used.

The target outputs which represent the phases of the structure factors have the binary form (0 or 1), with 0 representing a positive value (phase=0), and 1 a negative value (phase=π) as mentioned in section 2.3.5. From now on when logistic sigmoid transfer functions are used, binary outputs will be considered the default target output, unless something different is mentioned.

The same rules were followed for the experiments of these networks and 180 networks have been tested just as in the case of the hyperbolic tangent networks. The results obtained were similar. The networks showed poor performance and stopped training before they had the opportunity to overfit the training set's data. The difference between the results is that in this case neural networks with smaller number of neurons in their hidden layer had better performances than networks with a large number of neurons in the hidden layer. Also, neural networks with small number of neurons had narrower distribution of outputs, than those of networks with many neurons. These results are exactly the opposite of the results obtained with hyperbolic tangent networks and can be

summarized in Figures 3-11 and 3-12. The details on these figures are not so important, that is why they are not depicted in large pictures.

These networks also performed poorly and they are not appropriate for determining the phases of structure factors.

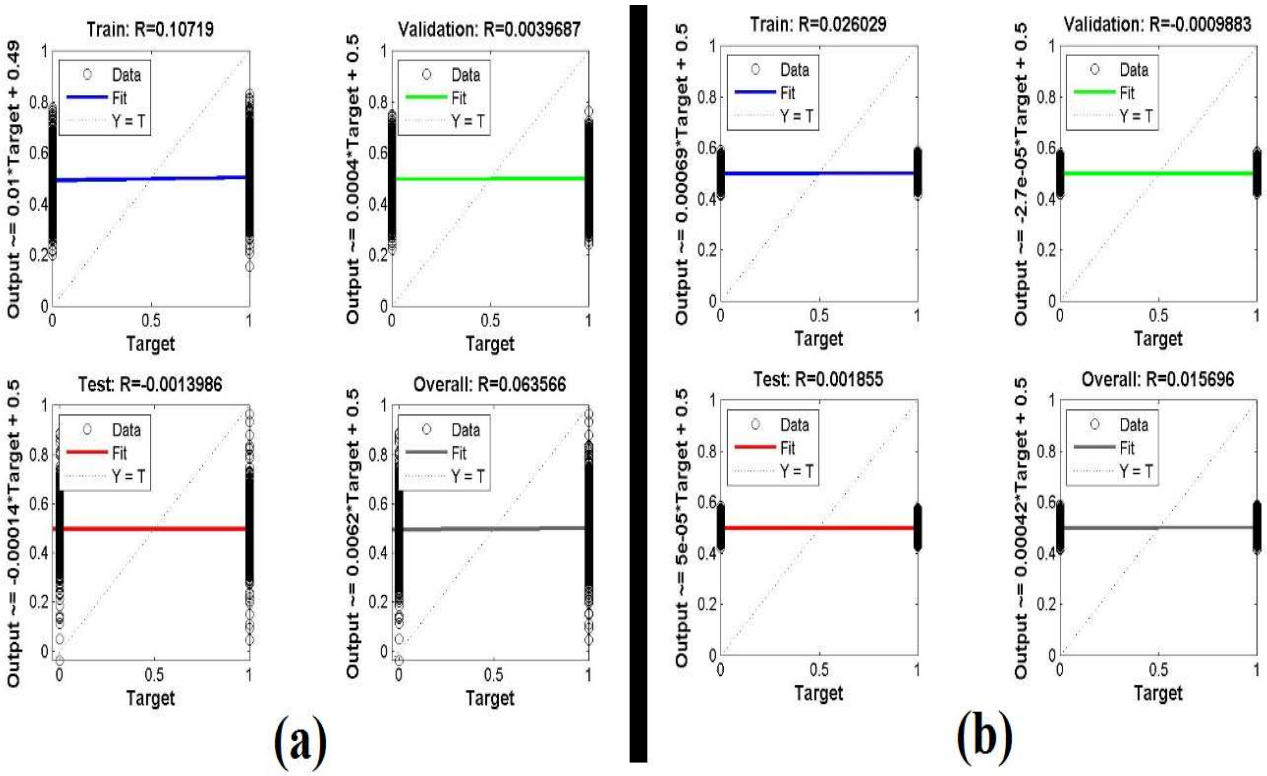

Figure 3-11.Typical regression plots of Logistic sigmoid transfer function : (a) A network with a large number of neurons (500) and (b) A network with fewer number of neurons (20)

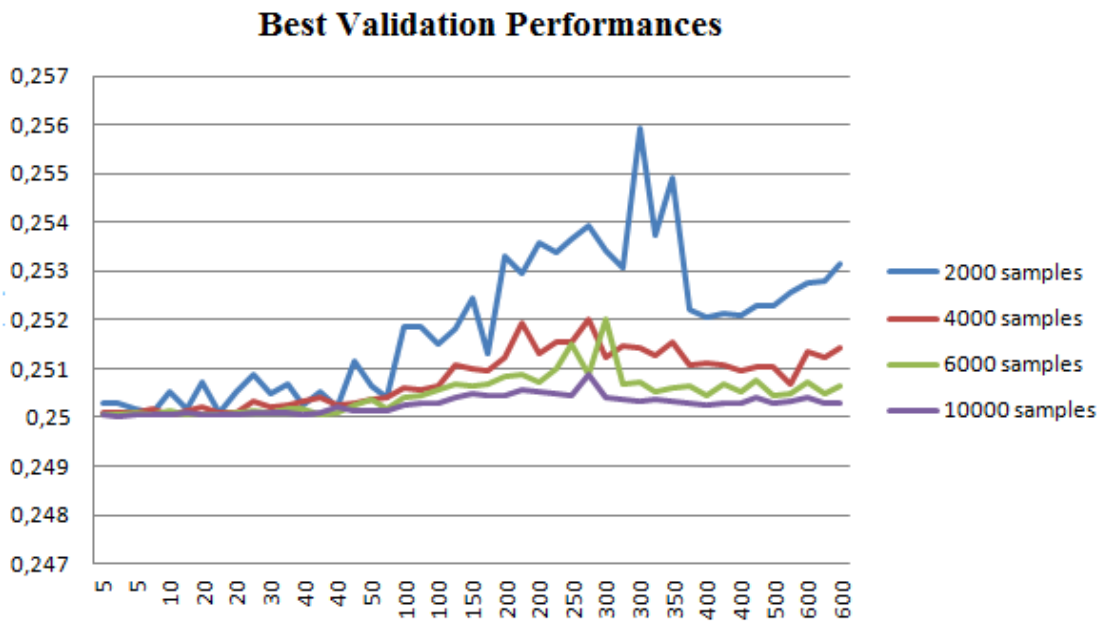

Figure 3-12. Best validation performances of neural networks with logistic sigmoid transfer function. Horizontal axis: Number of neurons in the hidden layer.

Another thing that was observed during the training of these networks is that they demanded fewer iterations to reach their max validation performance compared to hyperbolic tangent networks. This is shown in Fig. 3-13, where the networks from left to right are presented as follows: first networks trained with 2000 samples are presented with ascending order neurons in the hidden layer, then networks trained with 4000 samples are presented with ascending order of neurons in the hidden layer and so on.

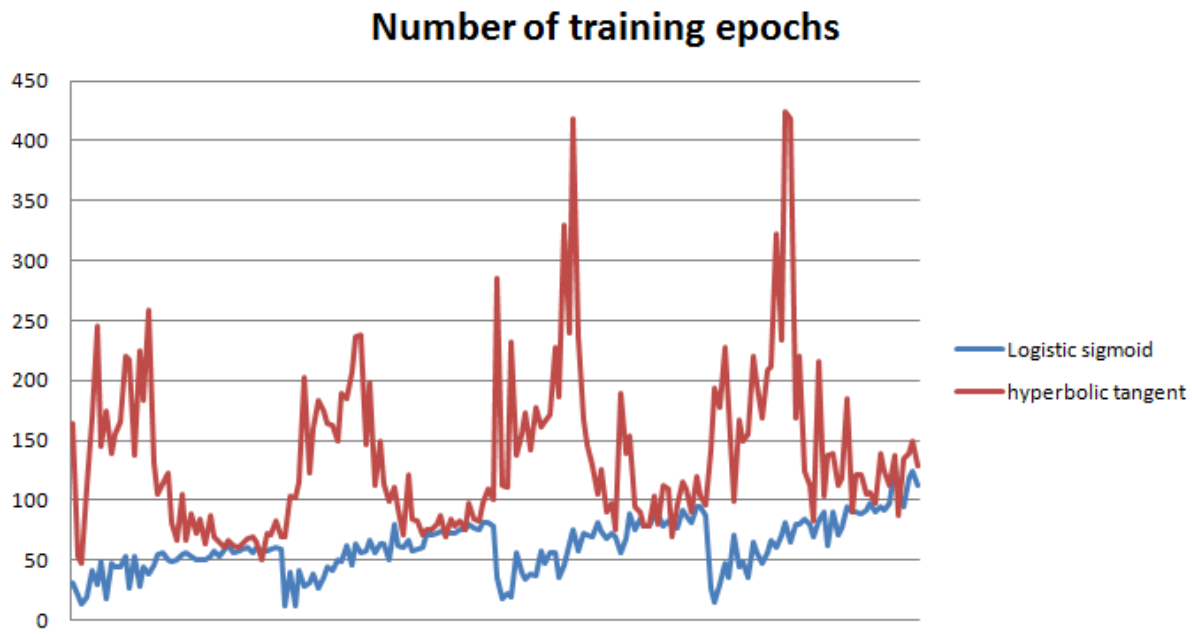

Figure 3-13. Comparative graph of number of training epochs for the logistic sigmoid neural networks and the hyperbolic tangent sigmoid networks

3.2.3 Networks with logistic sigmoid transfer function and structure output data type.

These networks are like the networks in the previous section and are shown in Fig. 3-10.

The difference of these networks is that the target outputs of the network are the signed structure factors (its actual values). The networks tested so far didn't provide good results and an alternative approach had to be tried. It was thought that if the targets of the outputs were the structure factors which have a range of values (in the data used in the experiments) between -27.4510 and 31.1890, that better results could be obtained. The reason behind this is that neural networks will try to approximate at most times a value larger than 1 for positive signs and smaller than -1 (or 0) for negative values. The average absolute value of the structure factors in the data set is 2.3399. Even if these outputs do not approach their target values sufficiently at least they have more chances, to be positive when the target is positive and negative when the target is negative. Another reason is that Sayre's equation (2.4) from which the relationship of triplets is derived, uses the values of the structure factors and not just the signs, so it is possible for a neural network to be able to approximate this relationship easier.

Just like the cases before 180 neural networks were tested. A typical regression plot of the response of the network is shown in Fig. 3-14

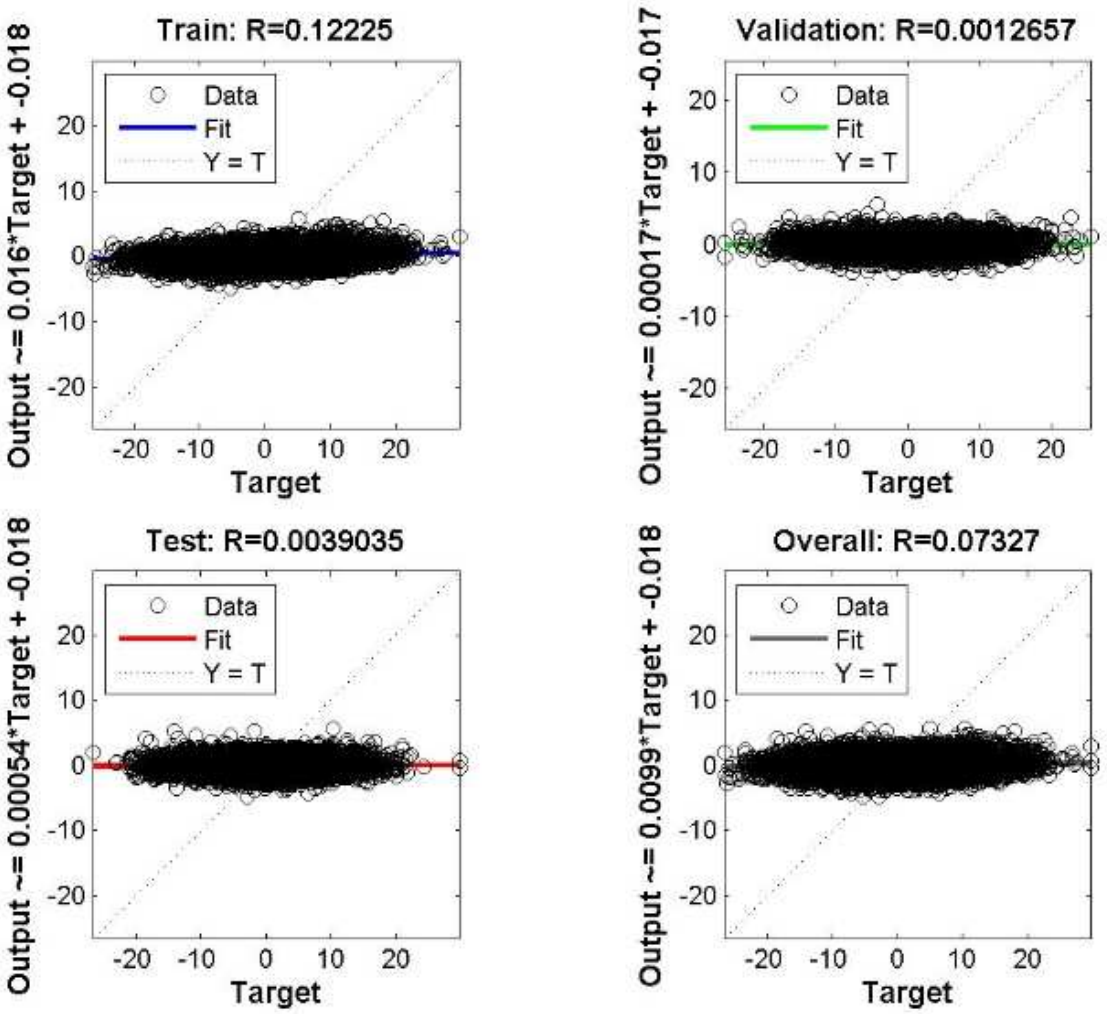

Figure 3-14. Typical regression plot of neural networks with structure factor target outputs (this particular network has 150 neurons in the hidden layer)

These neural networks provide similar results as those shown in Fig. 3-14. The graphs of the performances of these networks were not distinguishable (they overlapped each other) so an average of the three re-initializations of each network was taken and a bar graph was plotted instead. This is shown in Fig. 3-15.

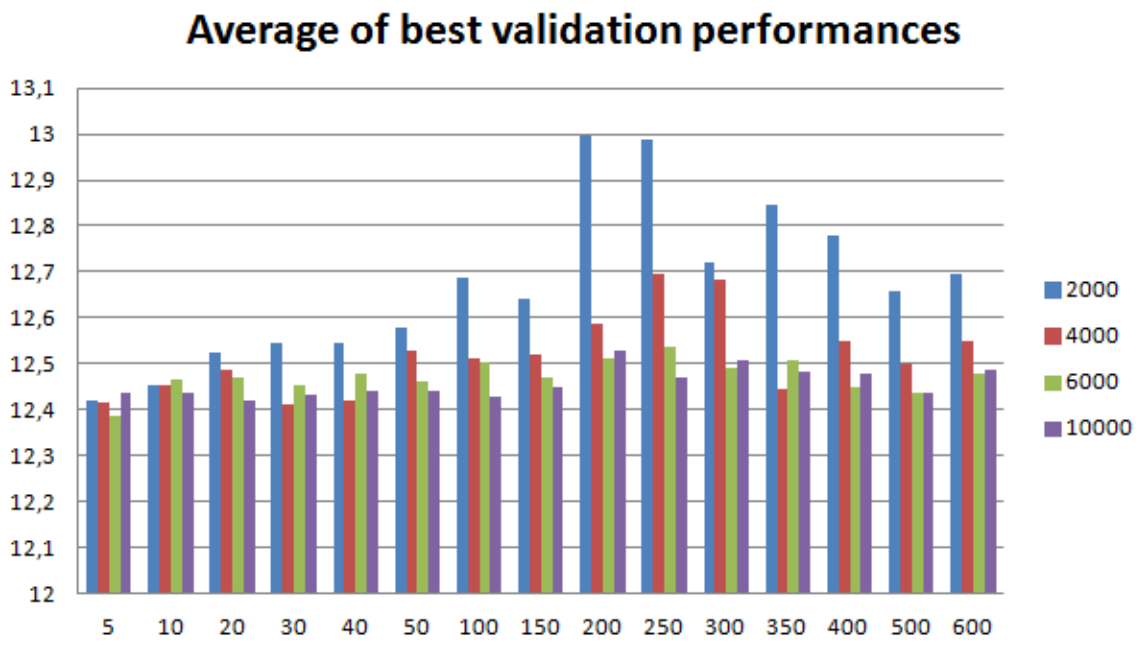

Figure 3-15. Average of best validation performances of neural networks with structure factor target outputs

It is obvious that performances have great values, and the networks cannot approach the structure factor's values. However, what about their signs? The outputs of 10 of the neural networks with the smaller performances were assigned values equal to 1 if they had positive values and -1 if they had negative values. The same happened for the original signed structure factors. These signs of the structure factors were the targets and the signs of the responses of the network the output. Then the regression plot of these was created, and is presented in Fig. 3-16. All the points of this plot are assigned to the four edges (1, 1), (-1, -1), (1, -1), (-1, 1) and are represented by four circles. The best linear fit indicated by the blue line, shows that these signs are not correlated. It was estimated that 49,4785% of the signs were allocated correctly. This is a number very close to 50%, and that is equivalent to a random distribution of signs. Therefore these networks cannot be uses for assignment of phases in structure factors.

The number of training epochs of these structures is between the logistic sigmoid networks and the hyperbolic tangent networks.

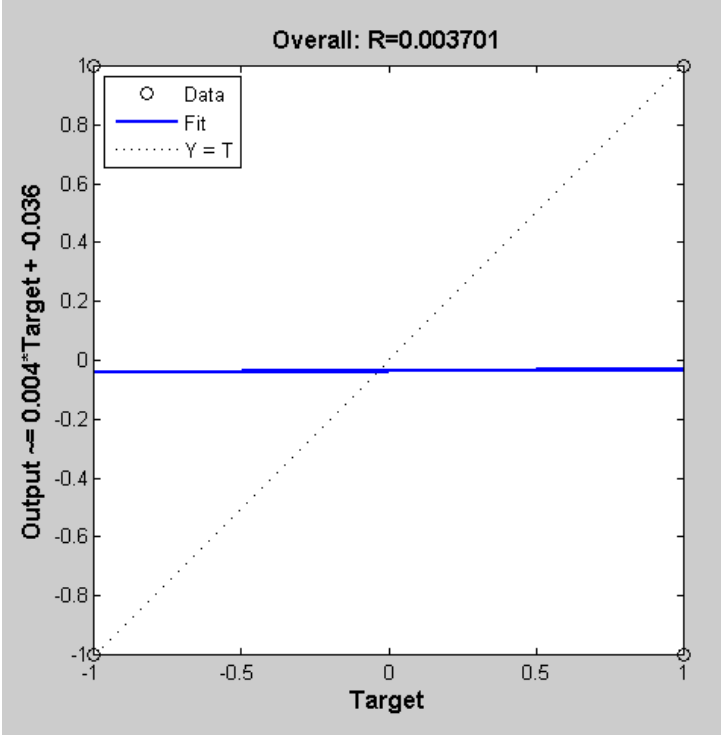

Figure 3-16. Typical regression plot of the sign relationship of targets and outputs

3.2.4 Product net input Neural Networks with hyperbolic tangent sigmoid transfer function and bipolar outputs.

The architecture of these networks is shown in Fig. 3-17.

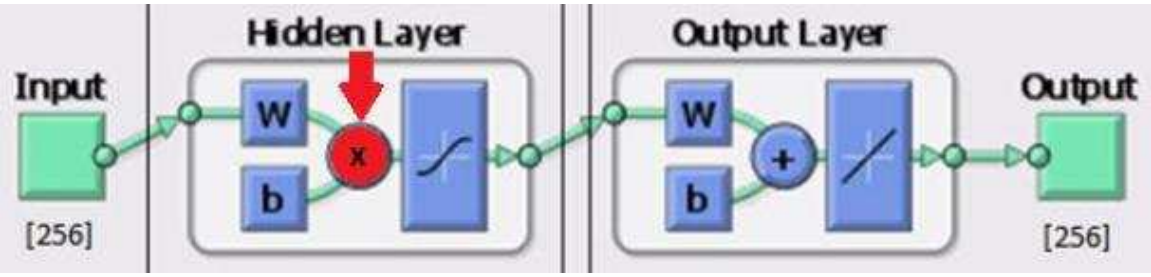

Figure 3-17. Product net input Neural Networks with hyperbolic tangent sigmoid transfer function

These inputs have an importance difference in comparison with the others, pointed by the red arrow in Fig. 3-17. The net input of the neuron is not the sum of the weighted inputs and biases, but their product. The other features of the network and the experiment procedure remain the same.

In Fig. 3-18 the best validation performances are shown. As someone can see these networks also perform poorly and are not capable to determine the signs of structure factors. A typical regression plot of the responses of these networks is very much like the regression plots of hyperbolic tangent networks shown in Fig. 3-6.

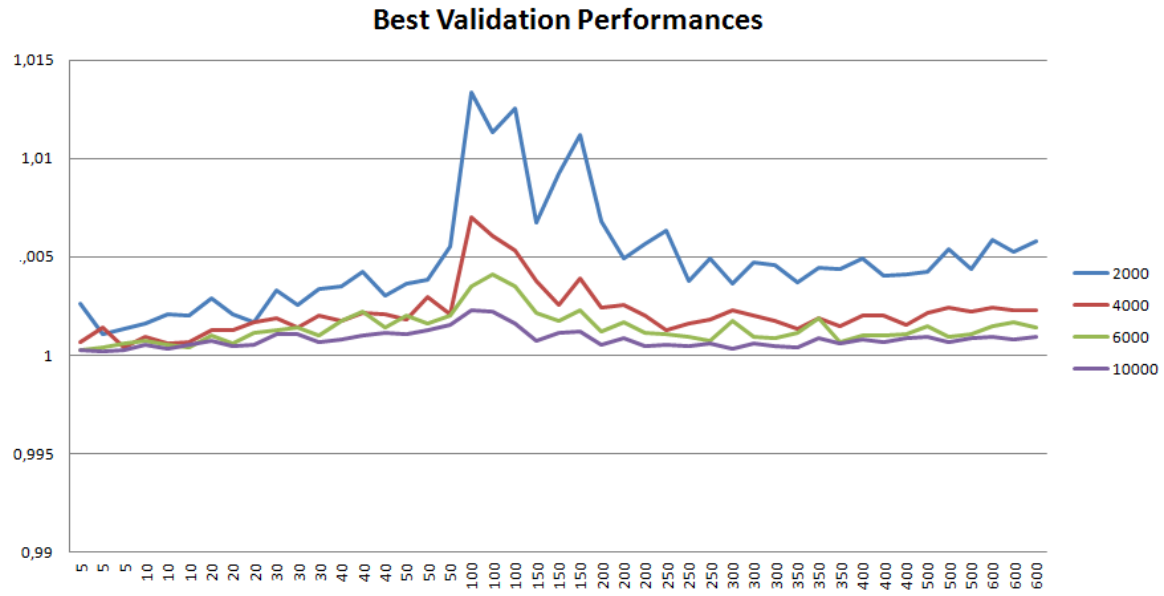

Figure 3-18. Best validation performances of product net input neural networks with hyperbolic tangent sigmoid transfer function

3.2.5 Neural Networks with 2 hidden layers hyberbolic tangent transfer function and bipolar output data.

Neural networks with 2 hidden layers are usually used when accuracy is needed. They are more complex networks and their architecture is shown in Fig. 3-19.

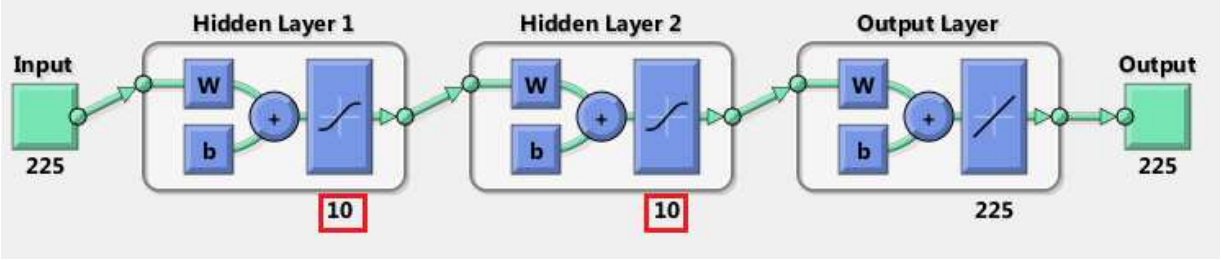

Figure 3-19. Neural Networks with 2 hidden layers and hyberbolic tangent transfer functions in both of them

The red boxes show that the number of neurons in these layers is adjustable. In this experiment the number of neurons in these two layers, is the same because that way the network trained easier. [18]

In this section when it is said that a network has 5 neurons, it means that it has 5 neurons in the first hidden layer and 5 neurons in the second hidden layer. Sample sizes of 4000, 6000 and 10000 training patterns are presented to the networks and the networks tested consists of 5, 10, 20, 40, 50, 100, 150, 200, 250, 300, and 350 neurons. The networks are reinitialized 3 times.

A typical regression plot of the responses of these networks is shown in Fig. 3-20. It can be noticed that these networks have very narrow distributions.

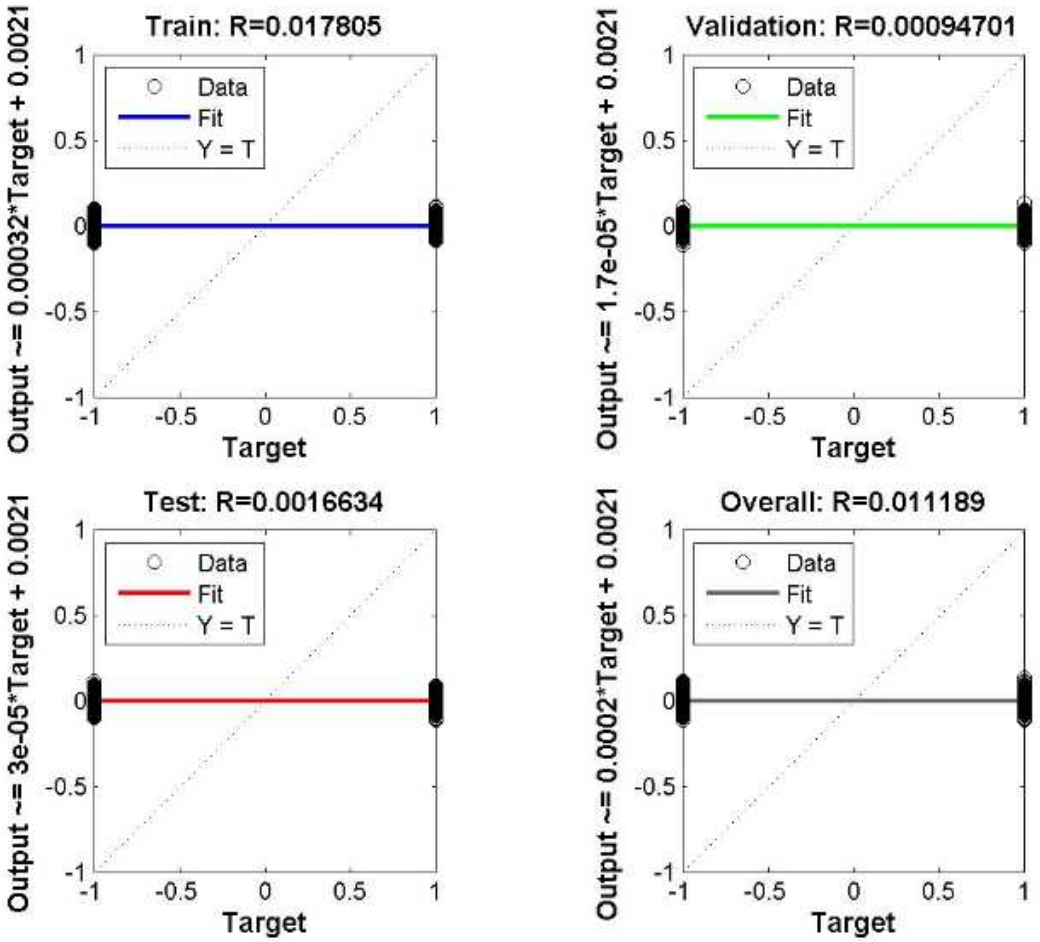

Figure 3-20. A typical regression plot of the responses of 2 hidden layers hyberbolic tangent transfer function networks (in this example 50-50 neurons in the hidden layers)

The best validation performance graph is shown in Fig. 3-21.

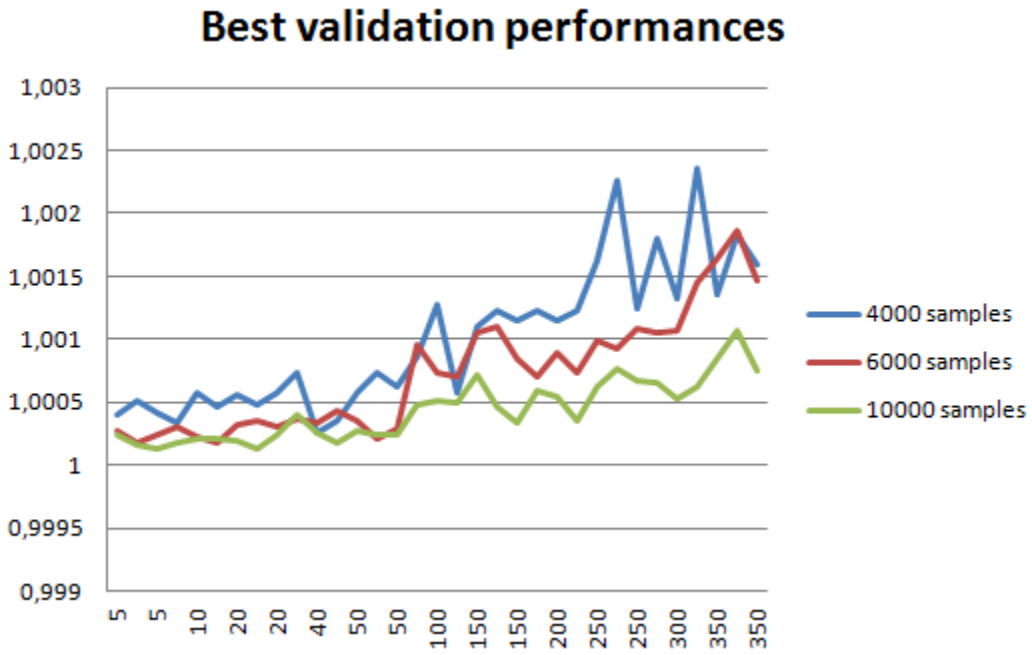

Figure 3-21. Best validation performances of neural networks with 2 hidden layers and hyperbolic tangent transfer functions

3.3 Experiments with unitary structure factors as input data

These experiments follow the philosophy of the experiments presented so far, but instead of structure factors, in the inputs of the neural network are presented the unitary structure factors. The target outputs of the networks are either bipolar or binary formats of the signs of the factors, or the values of the unitary structure factors with signs, in the case of structure output format. The reason why unitary structure factors are used is because they are present in important relationships in direct methods and because their absolute values are in the range of 0 to 1. Values in that range presented at the input of a neural network may be beneficial for the training of a neural network because of the form of the transfer functions. [9]

Because the experiments are identical, the details will be skipped (as they were analyzed in the previous sections) and only the results of the experiments will be presented.

3.3.1 Neural Networks with hyperbolic tangent sigmoid transfer function

In these experiments the training pattern sizes (or sample sizes) used were 4000, 6000 and 10000. The size of 2000 was omitted after the first experiments because it was considered small. The best validation performances are presented in Fig. 3-22. The results are similar in form with that of structure factors for the same network architecture, but the performances values are significantly smaller. However the typical regression plot, in Fig. 3-23 shows, that the signs at the outputs are still uncorrelated with the correct signs. Also the distribution of the outputs is narrower for all the networks than those of neural networks with structure factors as inputs.

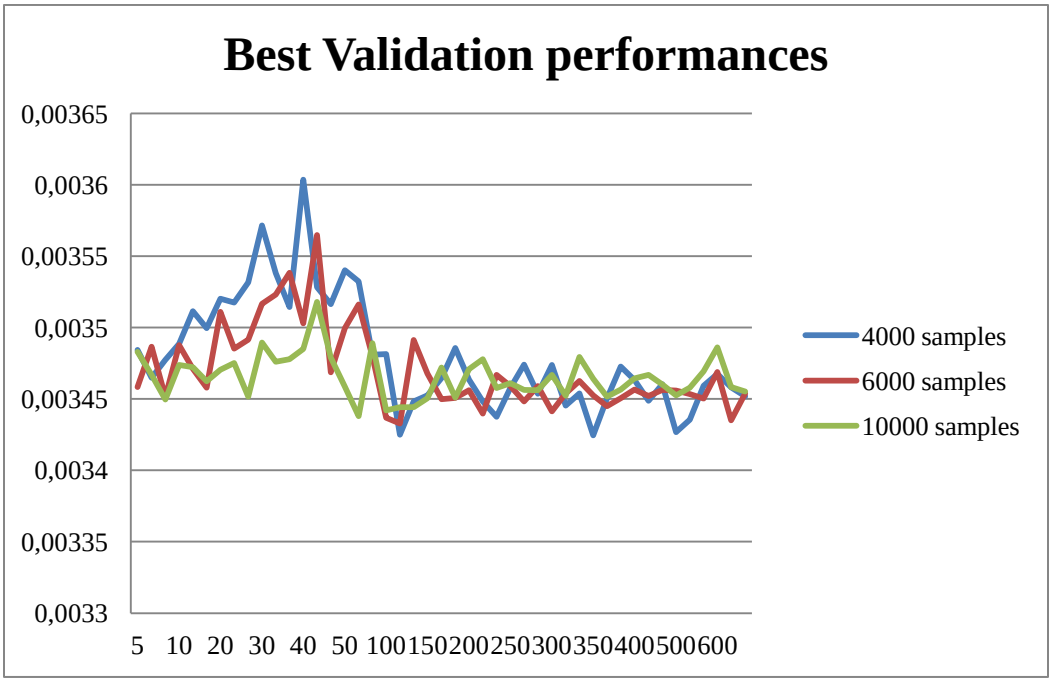

Figure 3-22. Best Validation performances of neural networks with hyperbolic tangent sigmoid transfer function (unitary)

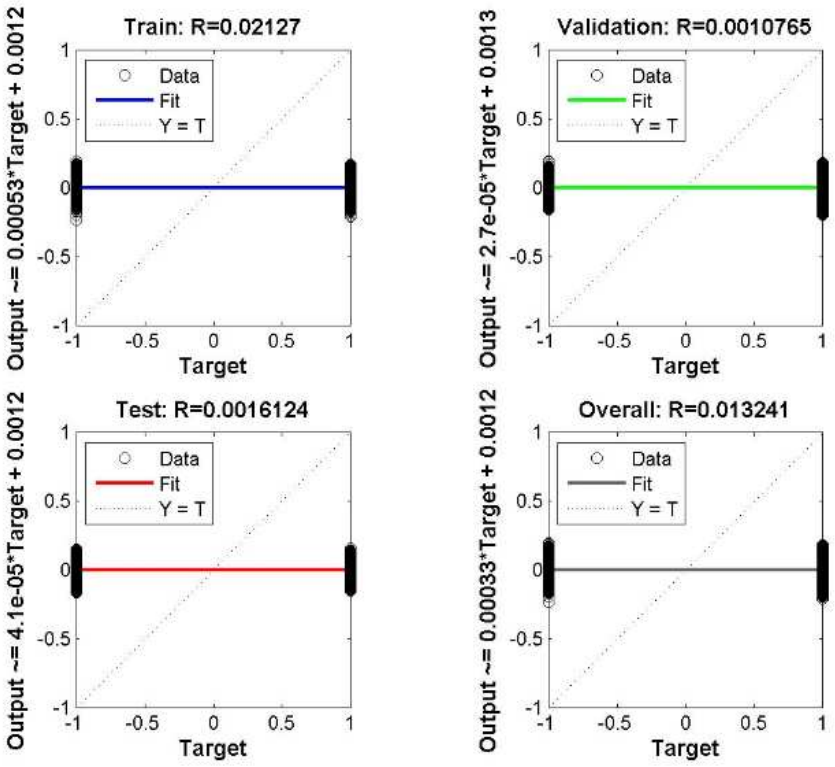

Figure 3-23. Typical regression plot of the responses of neural networks with hyperbolic tangent sigmoid transfer function (unitary) – (in this example 200 neurons -10000 sample size)

3.3.2 Neural Networks with logistic sigmoid transfer function

These networks were tested for sizes of 6000 and 10000 training patterns. The results are shown in Fig. 3.24. The lines are similar with those of the corresponding neural network architecture tested with structure factors, with smaller performances. The typical regression plots of the responses are shown in Fig. 3-25. A figure for all the R-values is shown in Fig. 3-26, which has the same philosophy as Fig. 3-8. It is drawn here just to illustrate that all the R-values are very small and the signs are still uncorrelated. Similar plots can be produced for all the networks examined so far.

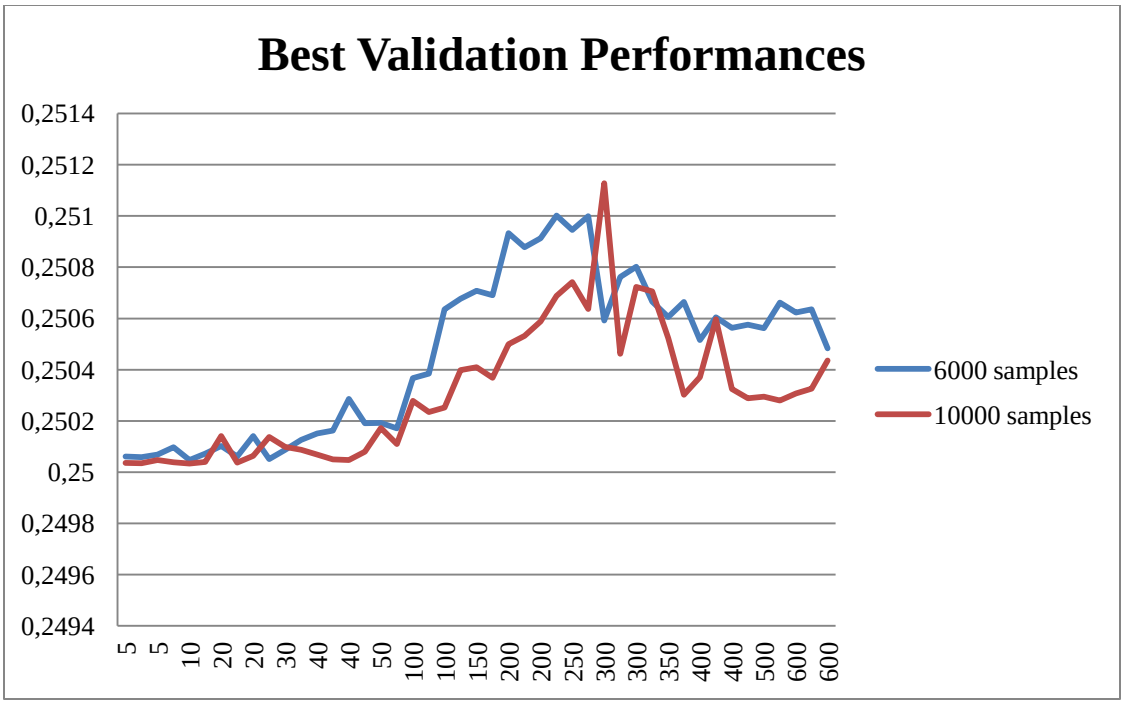

Figure 3-24. Best Validation performances of neural networks with logistic sigmoid transfer function (unitary)

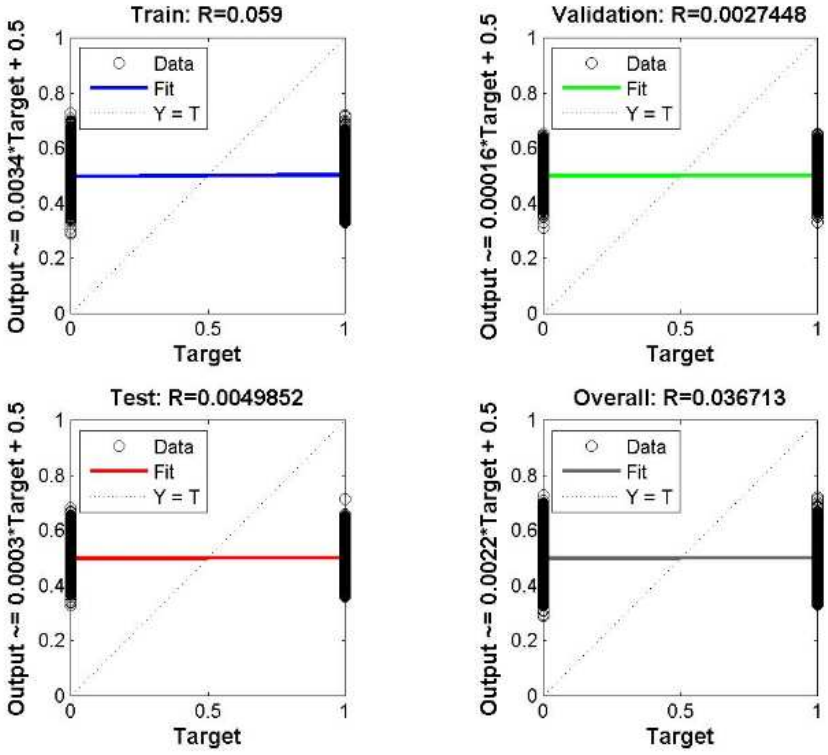

Figure 3-25. Typical regression plot of neural networks with logistic sigmoid transfer function (unitary) (in this example 300 neurons with 6000 samples)

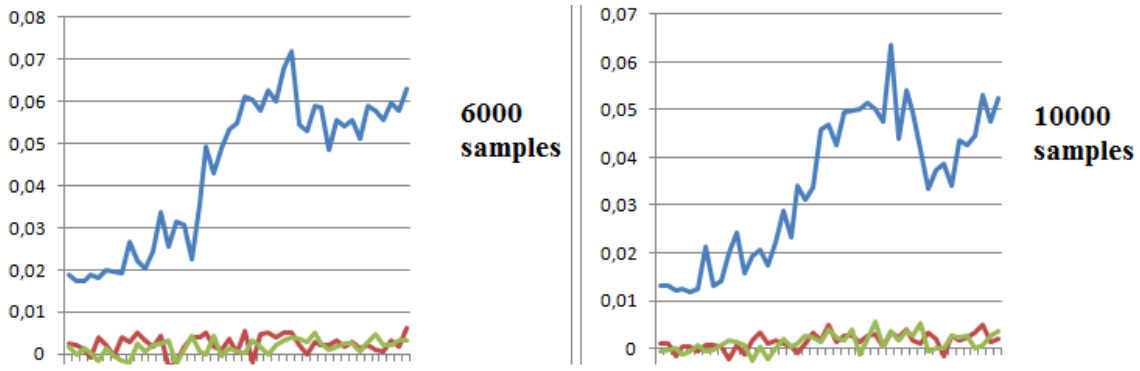

Figure 3-26. R values plots of neural networks with logistic sigmoid transfer function (unitary). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.3.3 Networks with logistic sigmoid transfer function and structure output format.

The results of these experiments are presented in Figures 3-27 and 3-28.

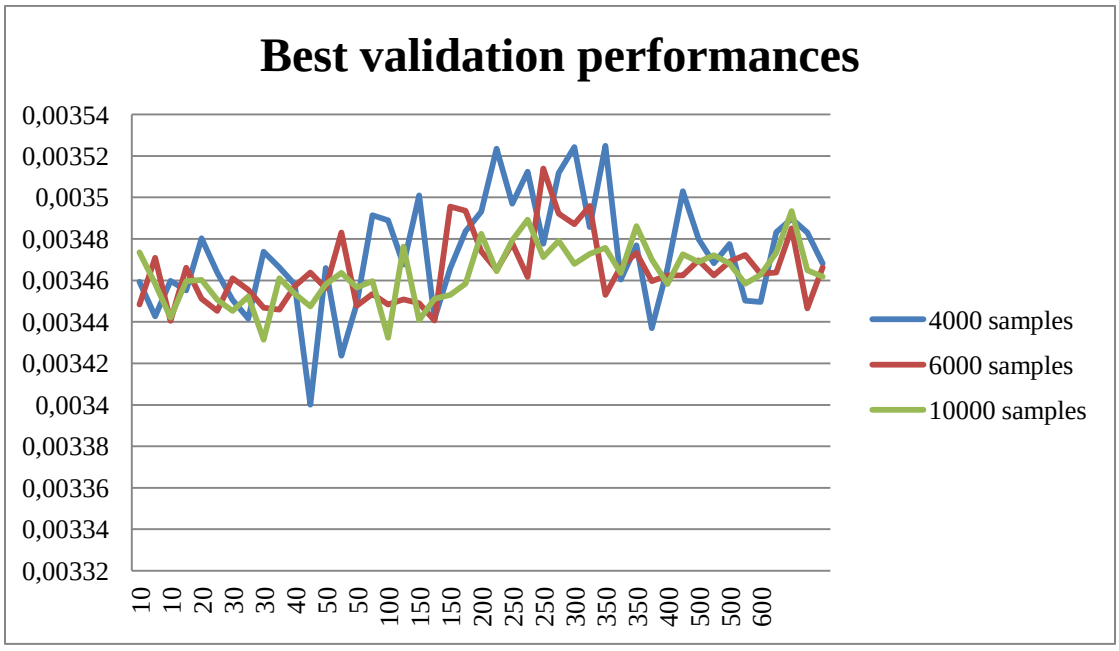

Figure 3-27. Best Validation performances of neural networks with logistic sigmoid transfer function (unitary- structure output format)

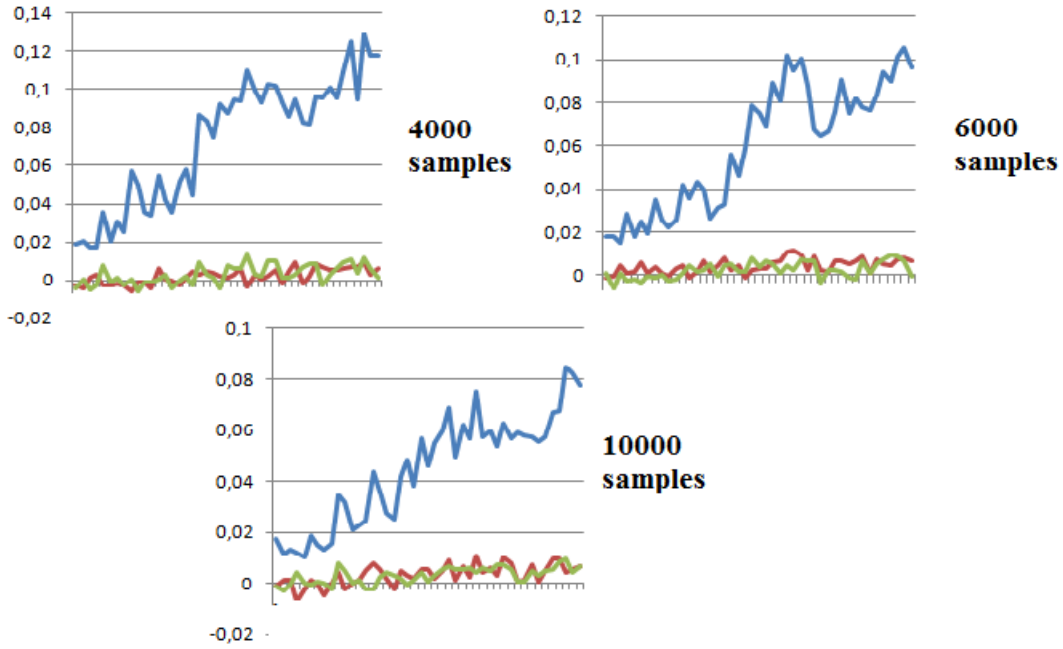

Figure 3-28. R values plots of neural networks with logistic sigmoid transfer function (unitary-structure output format). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.3.4 Product net input Neural Networks with hyperbolic tangent sigmoid transfer function and bipolar outputs.

The results of these experiments are presented in Figures 3-29 and 3-30.

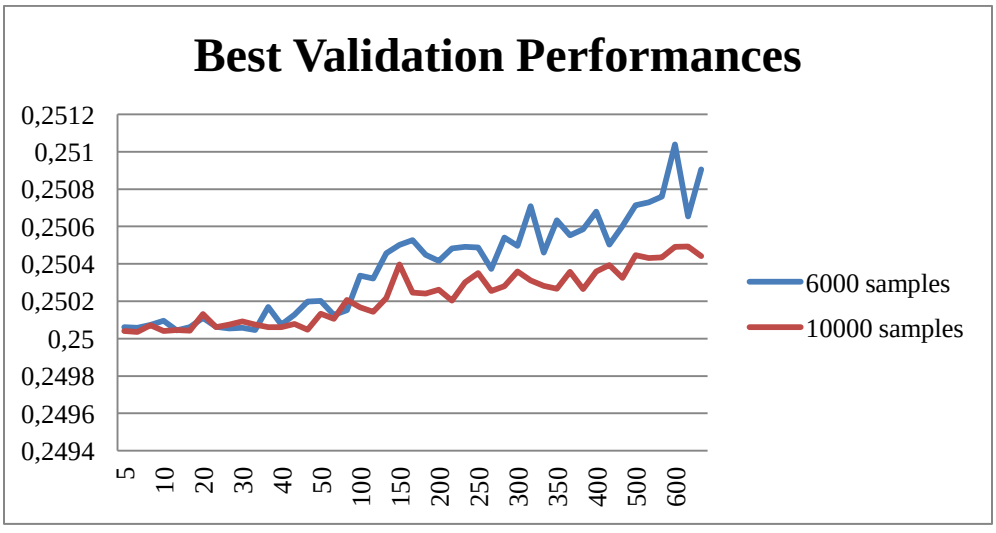

Figure 3-29. Best Validation performances of neural networks with product net input and hyperbolic tangent sigmoid transfer function (unitary)

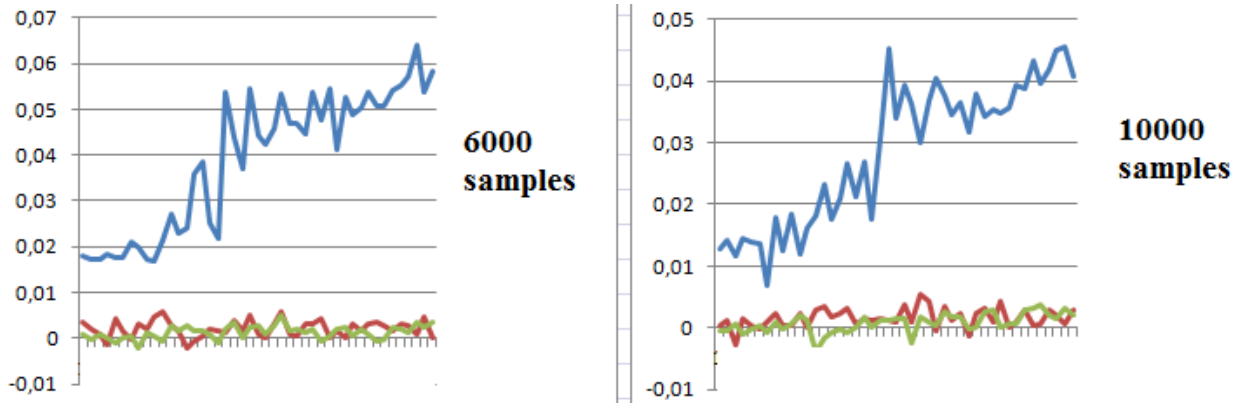

Figure 3-30. . R values plots of neural networks with product net input and hyperbolic tangent sigmoid transfer function (unitary). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.4 Experiments with normalized structure factors as input data

These experiments follow the philosophy of the experiments presented so far, but normalized structure factors are used. The target outputs of the networks are either bipolar or binary formats of the signs of the factors, or the values of the normalized structure factors with signs, in the case of structure output format. The reason why normalized structure factors are used is because they are present in important relationships in direct methods and because these factors are modified so that the maximum information on atomic position can be extracted from them. There is a chance that neural networks can interpret the values of these factors in a more efficient way.

Because the experiments are identical, the details will be skipped (as they were analyzed in the previous sections) and only the results of the experiments will be presented.

3.4.1 Neural Networks with hyperbolic tangent sigmoid transfer function

The networks in these experiments were re-initialized only two times.

The results of these experiments are presented in Figures 3-31, 3-32 and 3-33.

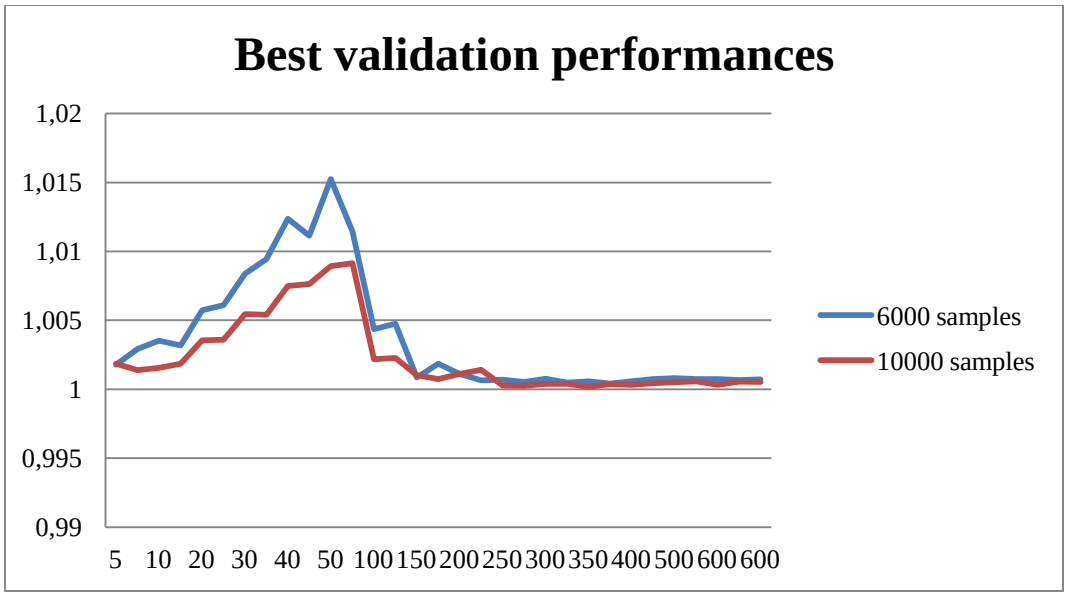

Figure 3-31 Best Validation performances of neural networks with hyperbolic tangent sigmoid transfer function (normalized)

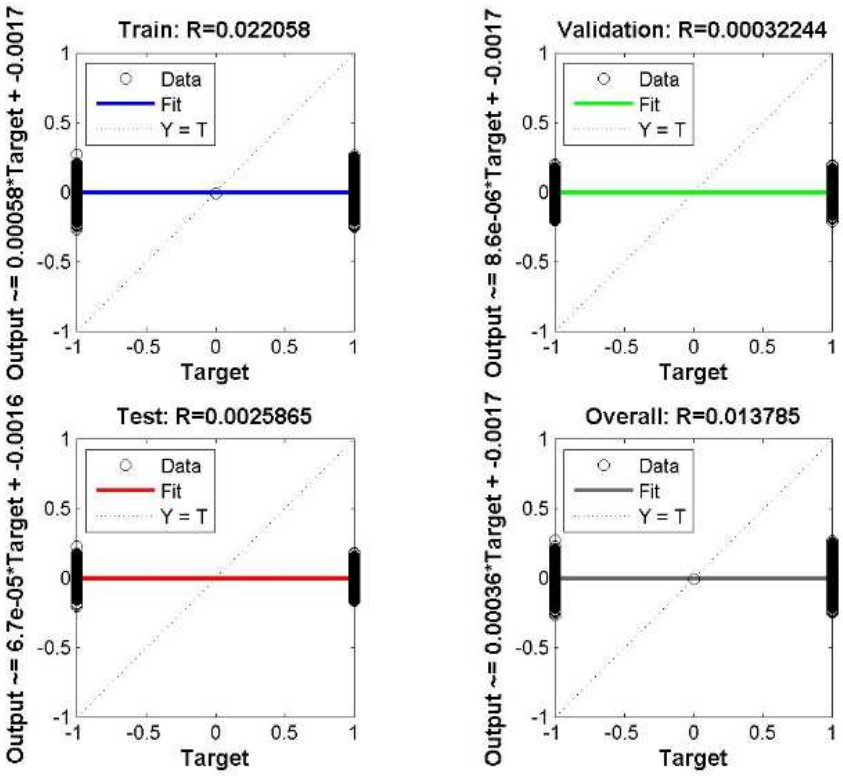

Figure 3-32. Typical regression plot of neural network with hyperbolic tangent sigmoid transfer function (normalized) - (in this example 250 neurons with 6000 samples)

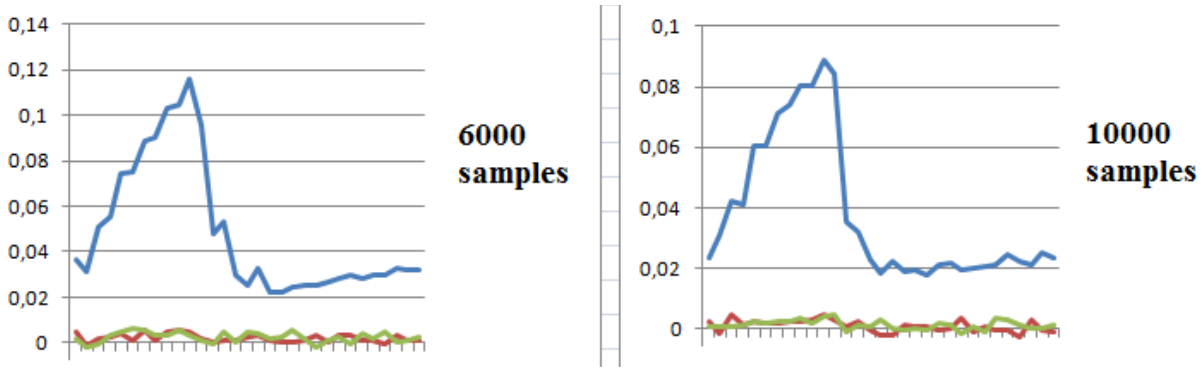

Figure 3-33. R values plots of neural network with hyperbolic tangent sigmoid transfer function (normalized). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.4.2 Neural Networks with logistic sigmoid transfer function

The networks in these experiments were re-initialized only two times and only the set of 10000 training patterns was used.

The results of these experiments are presented in Figures 3-34, 3-35 and 3-36.

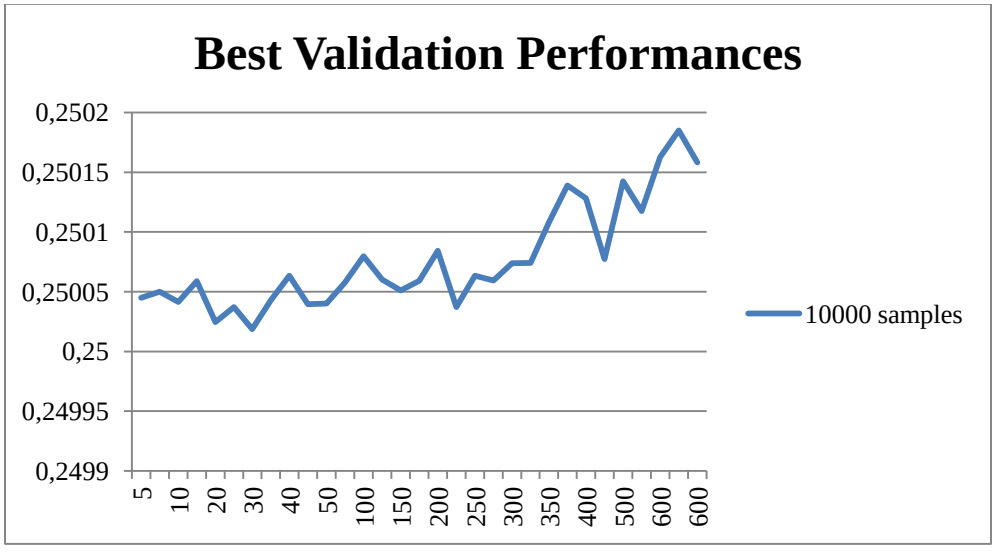

Figure 3-34 Best Validation performances of neural networks with logistic sigmoid transfer function (normalized)

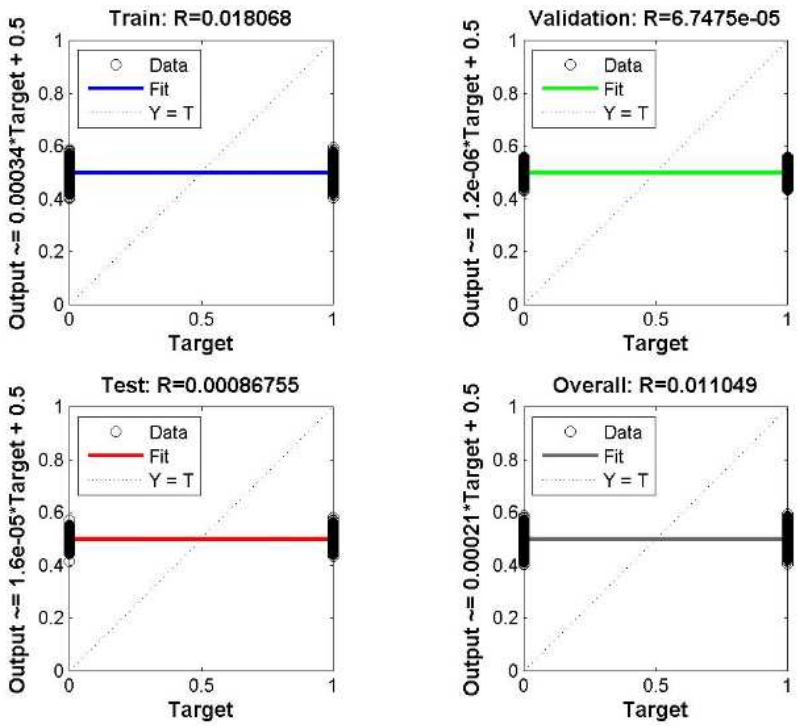

Figure 3-35. Typical regression plot of neural networks with logistic sigmoid transfer function (normalized) - (in this example 200 neurons with 10000 samples)

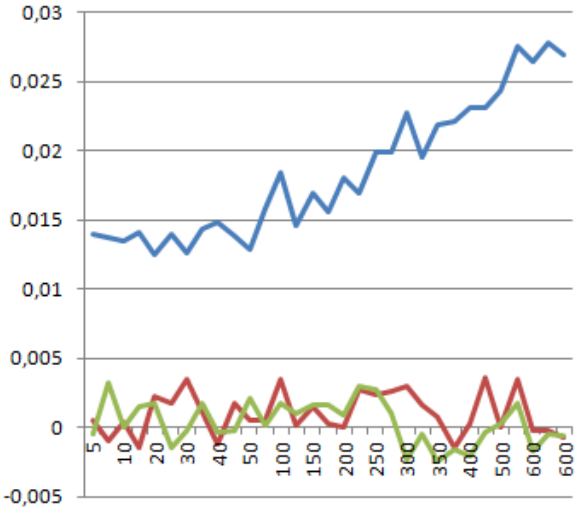

Figure 3-36. . R values plots of networks with logistic sigmoid transfer function (normalized). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.4.3 Networks with logistic sigmoid transfer function and structure output format.

The results of these experiments are presented in Figures 3-37, 3-38 and 3-39.

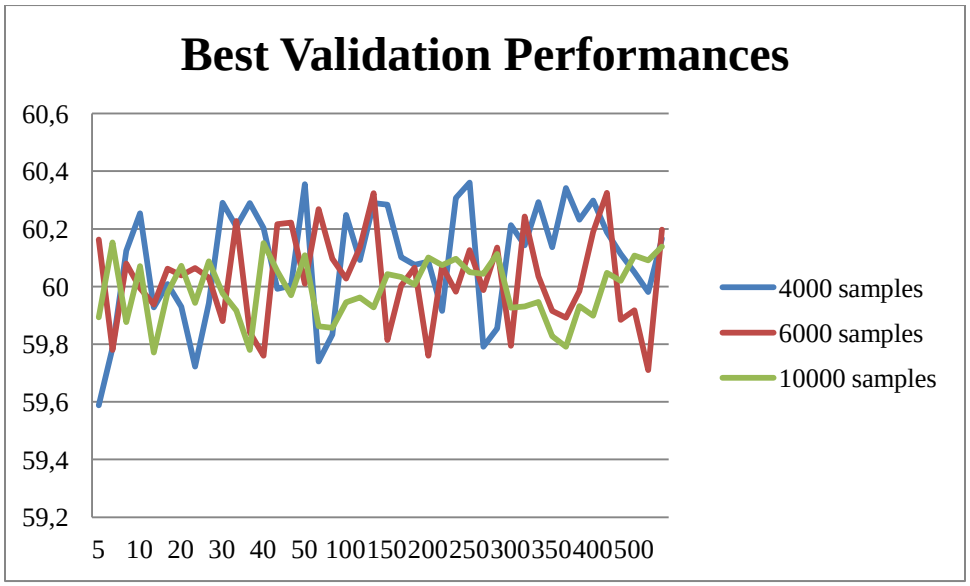

Figure 3-37. Best Validation performances of neural networks with hyperbolic tangent sigmoid transfer function (normalized-structure format output)

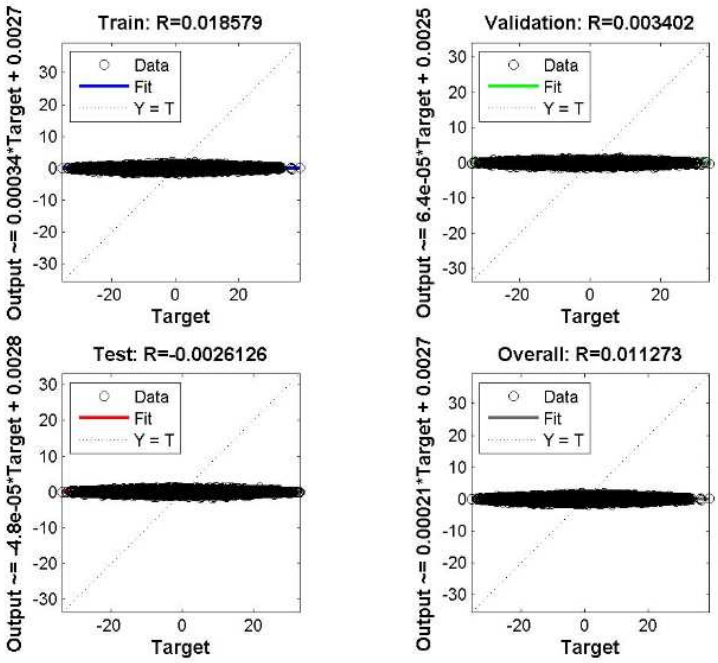

Figure 3-38. Typical regression plot of neural networks with logistic sigmoid transfer function (normalized- structure output format) - (in this example 200 neurons with 10000 samples)

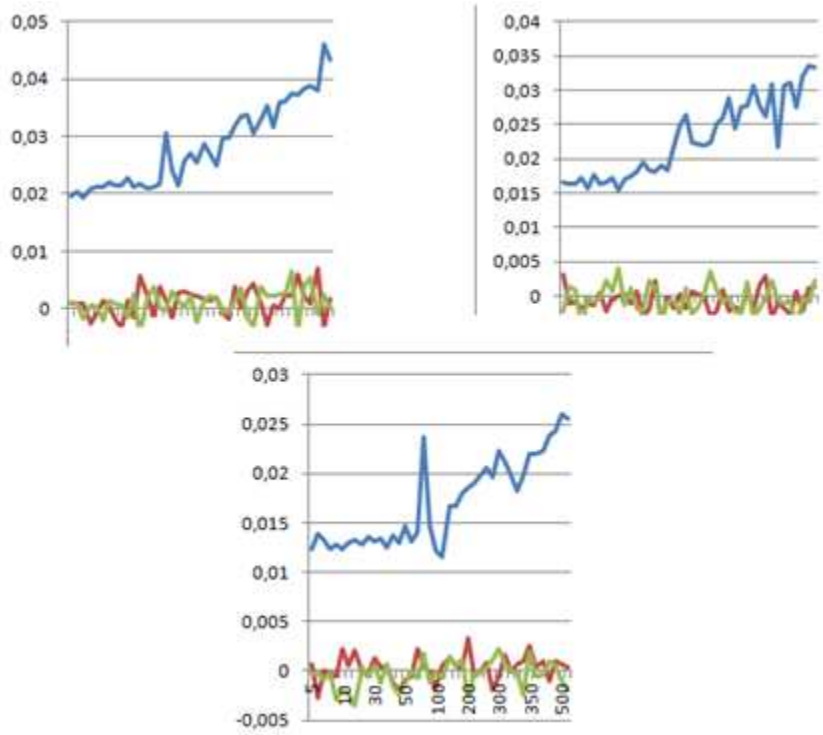

Figure 3-39. . R values plots networks of neural networks with logistic sigmoid transfer function (normalized- structure output format). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.4.4 Product net input Neural Networks with hyperbolic tangent sigmoid transfer function and bipolar outputs.

The results of these experiments are presented in Figures 3-40, 3-41 and 3-42.

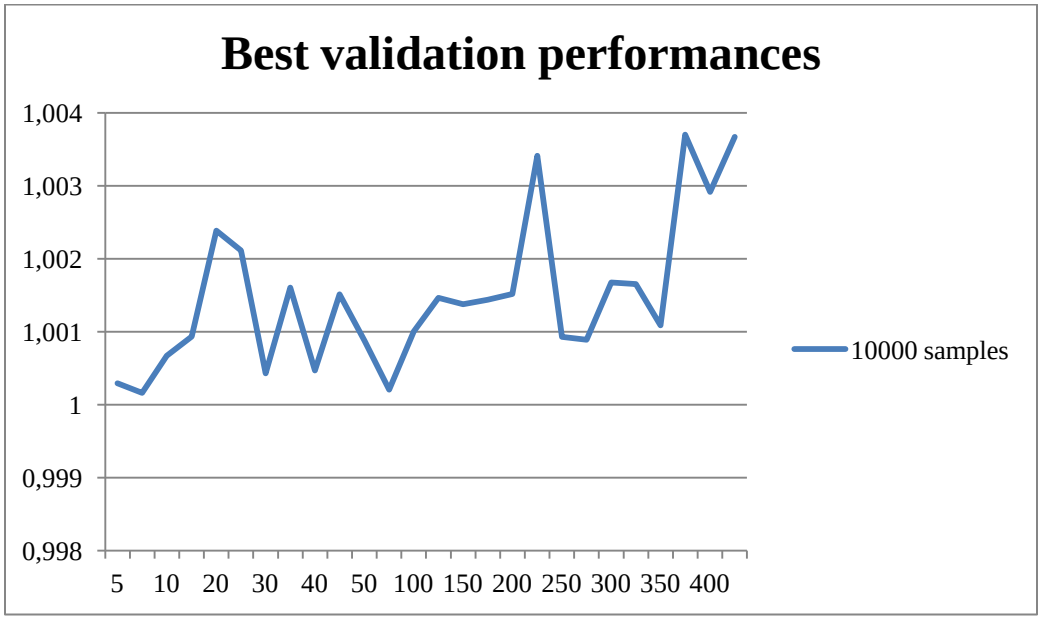

Figure 3-40. Best Validation performances of neural networks with product net input hyperbolic tangent sigmoid transfer function (normalized)

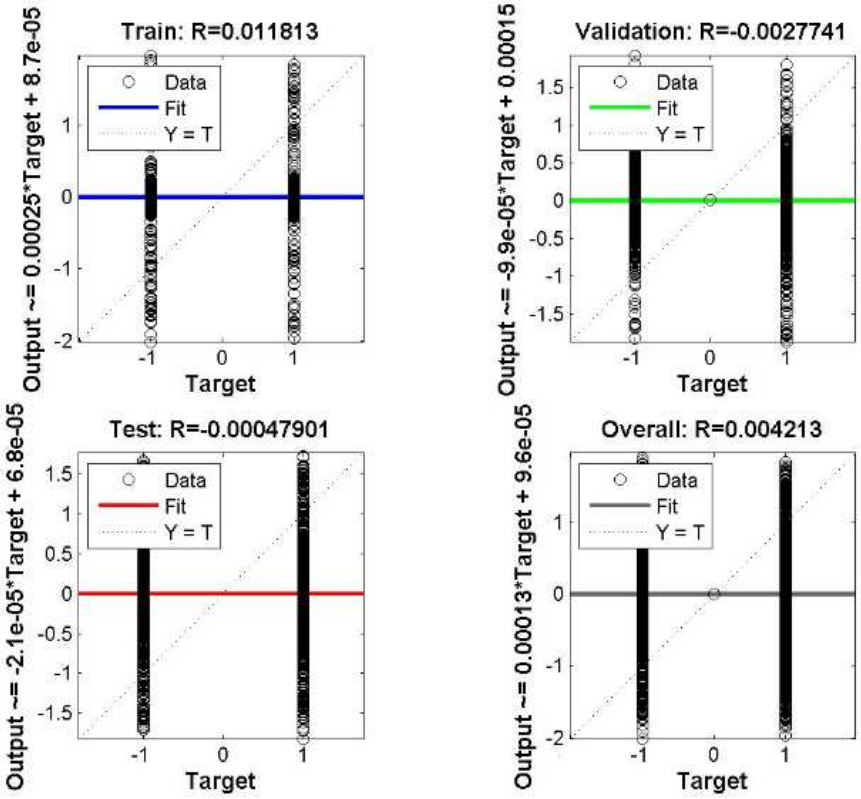

Figure 3-41. Typical regression plot of neural networks with product net input and hyperbolic tangent sigmoid transfer function (normalized)- (in this example 100 neurons with 10000 samples)

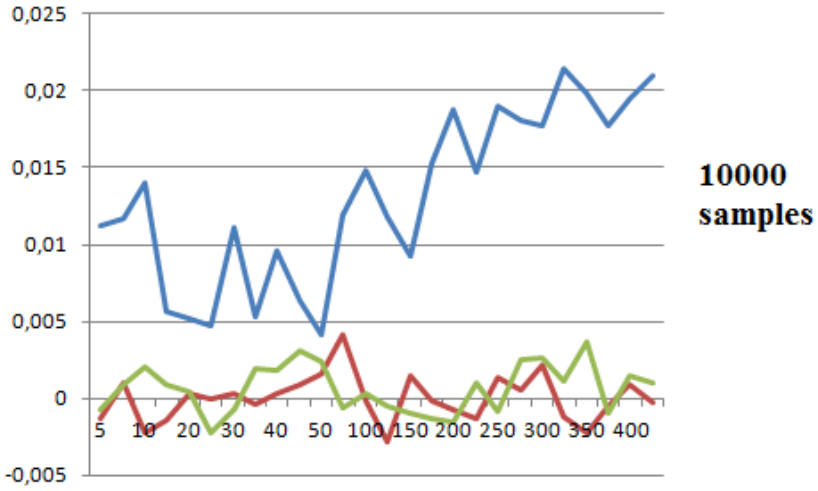

Figure 3-42. . R values plots of neural networks with product net input hyperbolic tangent sigmoid transfer function (normalized)). Horizontal axis: ascending number of neurons in the hidden layer. Blue lines: Training set. Green lines: test set Red lines: Validation set

3.5 Experiments with cascade forward networks.

The architecture of these networks is shown in Fig. 3-43.

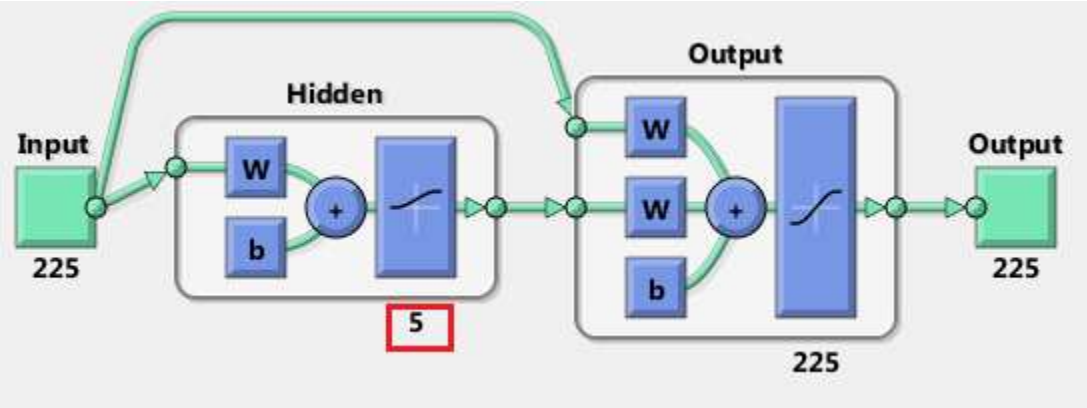

Figure 3-43. Architecture of cascade forward neural networks.

These networks are similar to feed-forward networks, but include a connection from the input and every previous layer to following layers. As with feed-forward networks, a two-or more layer cascade-network can learn any finite input-output relationship arbitrarily well given enough hidden neurons.

Again the red box symbolizes that neurons in this layer can be adjusted.

The target outputs of these experiments are the signed normalized structure factors. Only a small number of experiments were done with cascade forward networks and these were conducted using normalized structure factors. That is why there are no graphs for these experiments but the results are given in Table 3-1.

neurons	samples	best performance	training R value	Validation R value	Test R value
5	10000	60,16960723	0,099175492	0,010195462	0,006966992
30	10000	60,0049261	0,018951802	0,001298844	-6,30E-05
100	10000	60,42573374	0,0199853	0,001772751	-0,00099319
300	10000	60,27218647	0,019831413	0,001165748	0,000627646
600	10000	59,9736326	0,046939668	0,005939168	0,000606556

Table 3-1. Results of cascade forward neural networks (structure output format)

A typical regression plot of the responses of these networks is given in Fig. 3-45.

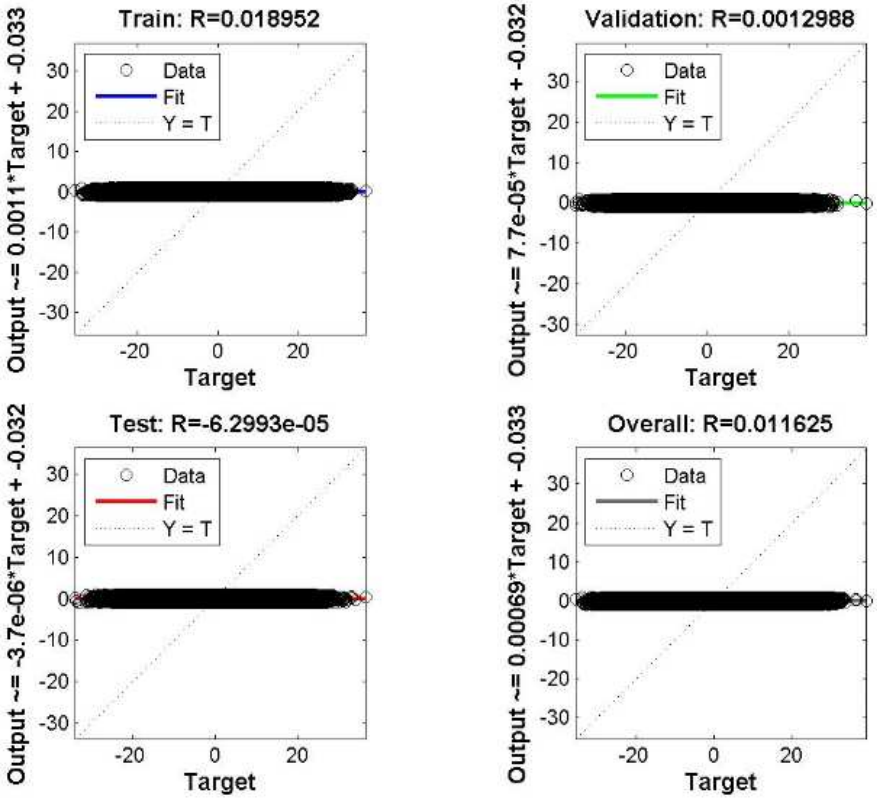

Figure 3-45. Typical regression plot of cascade forward neural networks (structure output format)- (in this example 30 neurons with 10000 samples)

These network although they were not tested as extensively as the other networks don' t seem to perform better.

In these networks a different approach was also tested when they were trained with bipolar target outputs. Up to now, the maximum validation checks were set to 6. As mentioned the surface of the error function in this problem may be very complex including many local minima. One thought is to increase the maximum validation checks to see how the training process behaves, and if someone can obtain better results by altering this parameter. The results in Table 3-2 were obtained with the maximum validation checks set to 100.

neurons	samples	best performance	Training R value	Validation R value	Test R value
5	10000	1,051627195	0,165167915	0,008464952	0,009956194
30	10000	1,042042179	0,183878431	0,009708404	0,012371126
100	10000	1,048879361	0,164089265	0,008513067	0,006864098
100	10000	1,039575157	0,188088904	0,006883337	0,010243637
300	10000	1,040865027	0,170099876	0,007244676	0,007243171
300	10000	1,036592754	0,13387581	0,007221175	0,005511023
600	10000	1,028847286	0,144741432	0,005645011	0,005967183
600	10000	1,03819209	0,119049656	0,005131547	0,003821427

Table 3-2. Results of cascade forward neural networks (bipolar outputs) – Validation checks set to 100

A typical regression plot of the responses of these networks may by shown in Fig. 3-46.

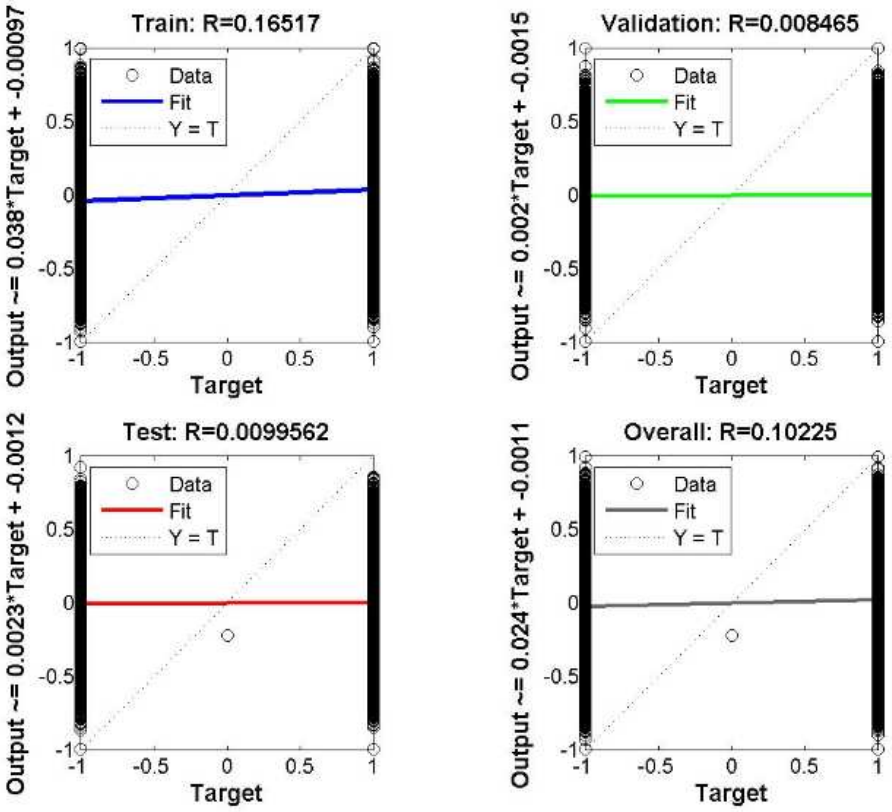

Figure 3-46. Typical regression plot of cascade forward neural networks – bipolar targets – validation checks set to 100

In all cases except of one, the network reached its best validation performance after 6 validation checks. Only in one case (with 30 neurons) the network performed about 50 validation checks and then produced a better performance by adjusting its weights. This case is shown in Fig. 3-47.

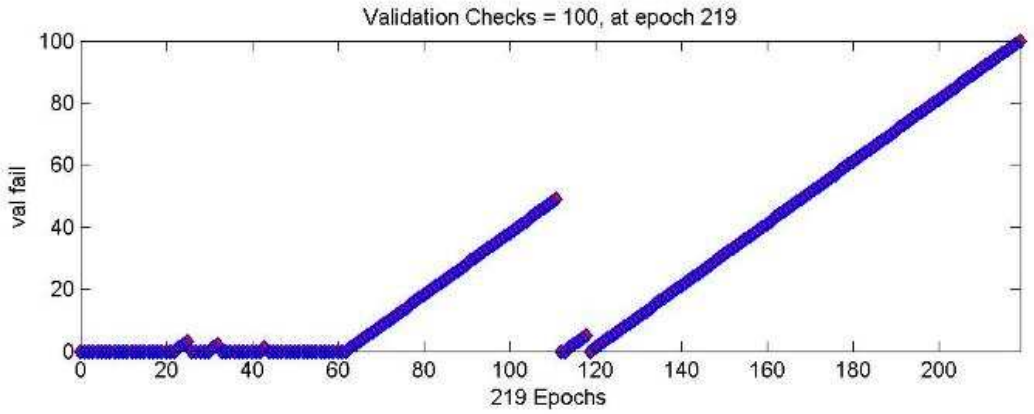

Figure 3-47. Validation checks of cascade forward neural networks – bipolar targets – 30 neurons – max validation checks set to 100

Another method tested was to bypass the early stopping procedure and train the network for 4000 epochs, without considering a validation set. Only a training set was used at the end of training to evaluate the outputs of the network to unknown inputs. This test was performed only to three networks

The results of these tests are shown in Table 3-3.

neurons	samples	training R value	Test R value
5	10000	0,154997568	0,011726415
30	10000	0,145105423	0,006360455
100	10000	0,281009047	0,004999469

Table 3-3. Results of cascade forward neural networks (bipolar outputs) – Maximum epochs=4000, no early stopping.

As it has been expected we obtain better results for training R values, because the networks slowly learns the training patterns, but the test R-value remains low, because the network does not generalize. Also the error function performance after a number of epochs becomes almost flat, which means that the network learns very slowly. This is shown in Fig. 3-48.

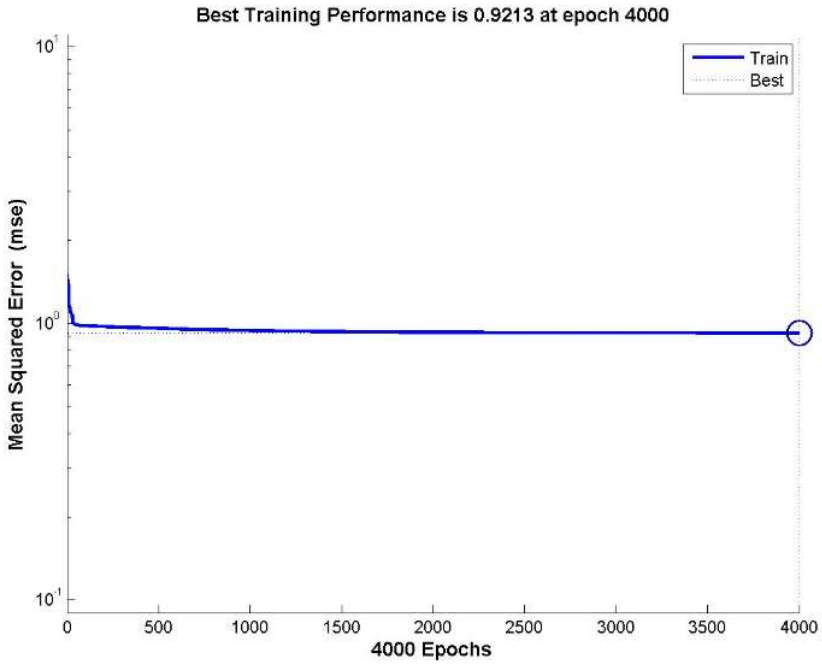

Figure 3-48. Performance plot of cascade forward neural network (300 neurons-10000 samples). Maximum epochs=4000, no early stopping.

In all cases the efficiency of the networks are not good enough.

3.6 Neural Networks and origin.

During the experiments of this study, none of the tested architectures showed promising results. That fact led to a thought that neural networks may be able to represent the phases of the structure factors in a form that is not visible to us using the regression plot.

Specifically neural networks may be able to represent relationships of triplets but produce results (phases) with an origin of coordinates fixed at a different point than the one used to produce the structure factos in the training data (equation (2.8)). Remember that when the origin changes, the absolute values of the structure factors (and the unitary and normalized factors) remain the same, but their phases change. Input data does not give information on the origin, only on the relative magnitudes of the structure factors. Target outputs on the other hand refer to an origin at 0,0 but the network may be capable to learn just the relationships of the signs (e.g. the triplets) and not information on the origin.

Having that in mind, a modified algorithm was created for networks trained with training patterns whose target outputs are the signed structure factors. This algorithm trains the

network as usual but has a difference in the early stopping process. The difference is that the validation set has as output targets the absolute values of the structure factors. When the network after each iteration, is evaluated with the validation set, the inputs of the validation set are presented to the network, its outputs are produced, and then the absolute values of the outputs of the network are compared with the absolute values of the structure factors of the validation set. From this comparison the error function performance is estimated and the algorithm continuous as usual. With this algorithm the network supposedly will be encouraged to learn the structure factors independently of the sign, and the number of training epochs is expected to increase, before early stopping halts the training process. This procedure is not supported by the neural network toolbox, because we are comparing different outputs during the presentation of training sets and different "outputs" during the presentation of the validation set. The algorithm which was created overcomes this difficulty in the expense of larger computational times.

The results are shown in Fig. 3-49 and Fig.3-50. The normalized structure factors are used. Each network (with hyperbolic tangent transfer function) was re-initialized 3 times. In Fig. 3-48 the error function performance is estimated by comparing the absolute values of the outputs and the absolute values of the normalized structure factors in the validation set. In Fig. 3-50 the regression plot for 4 situations is drawn. In 'train' the target outputs (signed normalized structure factors) and the outputs of the network are plotted for the training set. In 'Train absolutes' the absolute values of the target outputs and the absolute values of the outputs of the network are plotted for the training set. In 'validation-test signed' the signed structure factors (which are not the target outputs) and the outputs of the network are plotted for the validation set. In 'Validation' the target outputs (absolute values of the structure factors) and the absolute values of the outputs of the network are plotted for the training set (this is where the error function is based).

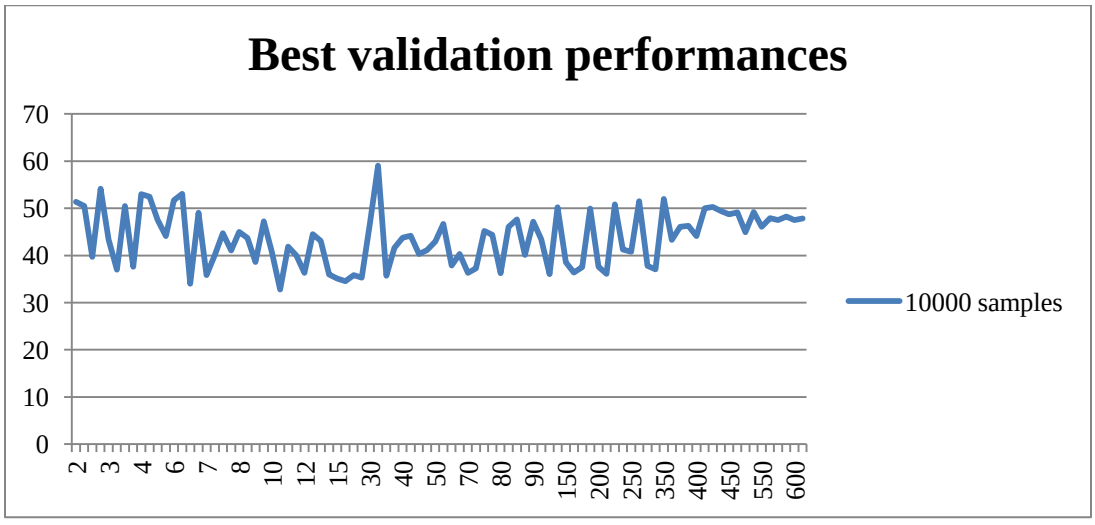

Figure 3-49. Best validation performances of a neural network with hyperbolic tangent transfer function, trained with the modified early stopping algorithm.

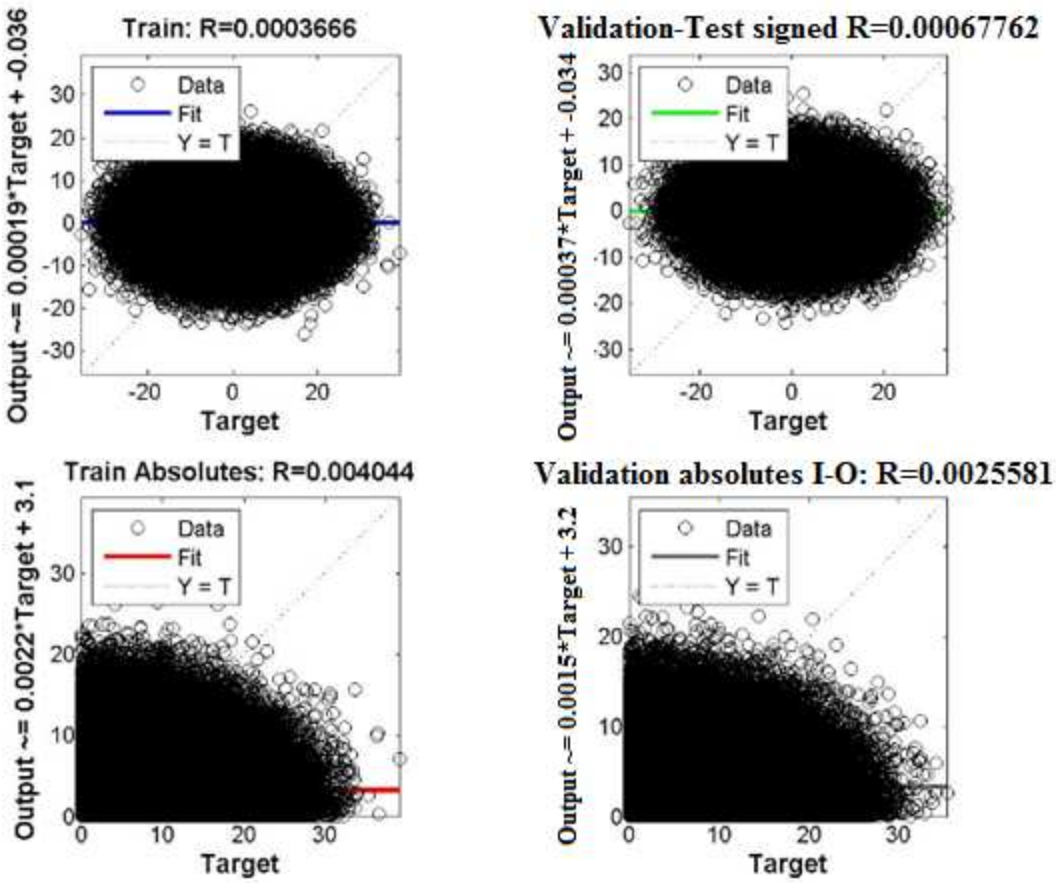

Figure 3-50. Typical regression plot of a neural network with hyperbolic tangent transfer function, trained with the modified early stopping algorithm – (in this example 300 neurons – 10000 samples)

During this experiment an important increase in the training epochs was not observed. The R- values remained low, indicating uncorrelated data once again even for the absolute values of the normalized structure factors. In this case in order to increase the training epochs it would be easier to skip the early stopping process altogether. After all with a large size of training patterns the network is not very likely to overfit within a reasonable number of epochs.

The method to evaluate if the neural networks produce results that have a different origin-fixing is to calculate the Fourier synthesis of their outputs and compare it with the Fourier synthesis of the original structure factors. Specifically we get the signs of the outputs the neural networks produce and the absolute values of the original structure factors (which are known after a real diffraction experiments). Then we calculate the Fourier synthesis of these 'hybrid' structure factors and we compare it with the correct structure factors. If the electron density maps produced are similar, but the one produced by the 'hybrid' structure factors is shifted compared to the original that means that the network produces

results with different origin-fixing. In this case the signs are different from those in the training patterns but the relationships among the signs are those dictated by direct methods and produce the correct electron density map and subsequently the correct structure.

Some of the networks with the best performances were compared (not just from this experiment but also from the experiments described in the previous sections) with four structures. The results were too many to describe them all so some representative examples will be shown here.

The first interesting result of this procedure concerns networks with a few numbers of neurons in their hidden layer (~up to 10-15). These networks most of the time produce very similar results in their outputs, regardless their inputs. If the Fourier synthesis of the outputs of these networks is made they give almost the same results for almost any input (the Fourier synthesis from the outputs, not the 'hybrid' factors described above). An example is given in Fig.3-51. Five different molecular structures were tested with this neural network and it produced the same results. This is not the case with networks which have a larger number of neurons in their hidden layers. Different inputs produce different outputs in these networks. When the 'hybrid' structure factors are used in a Fourier synthesis then the results are different for different inputs, even for networks with small number of neurons in the hidden layer.

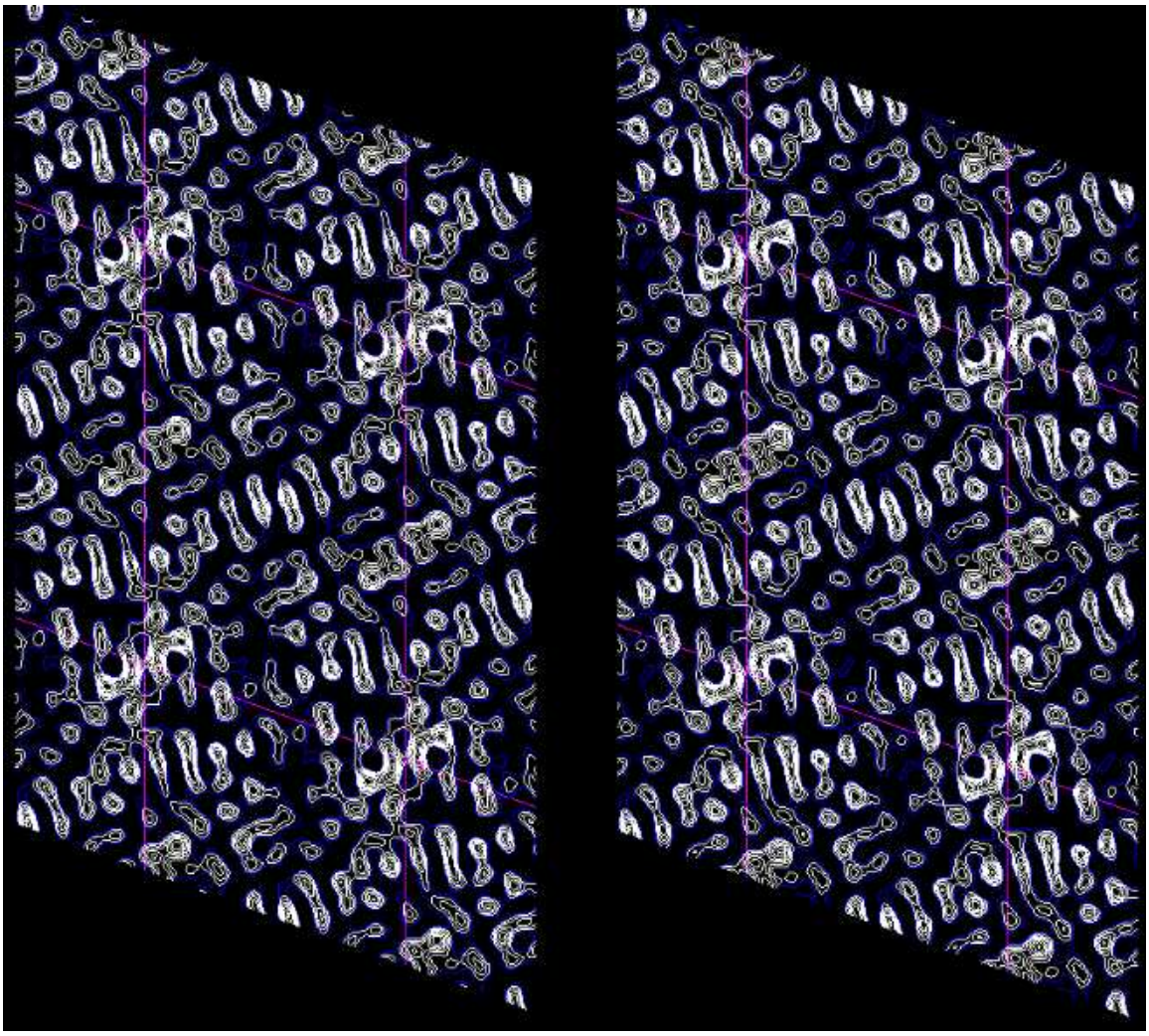

Figure 3-51. Fourier synthesis of the outputs of a neural network with 10 neurons trained with 10000 samples using the modified algorithm of early stopping (hyperbolic tangent sigmoid transfer function) The outputs were produced by two different inputs. However, the Fourier synthesis of the outputs is almost identical.

The second interesting but unfortunate result is that all the networks tested with the abovementioned method (15 networks) using the 'hybrid' structure factors had different Fourier syntheses than that of the electron density of the two dimensional molecular structures under determination. Thus, the hypothesis that neural networks produced phases with the origin fixed at different points than 0, 0 was wrong. These 15 networks were selected so as to cover a range of neurons in the hidden layer, and had good performances when compared with other networks from the same experiment. Two examples will are shown in Fig. 3-52 and 3-53.

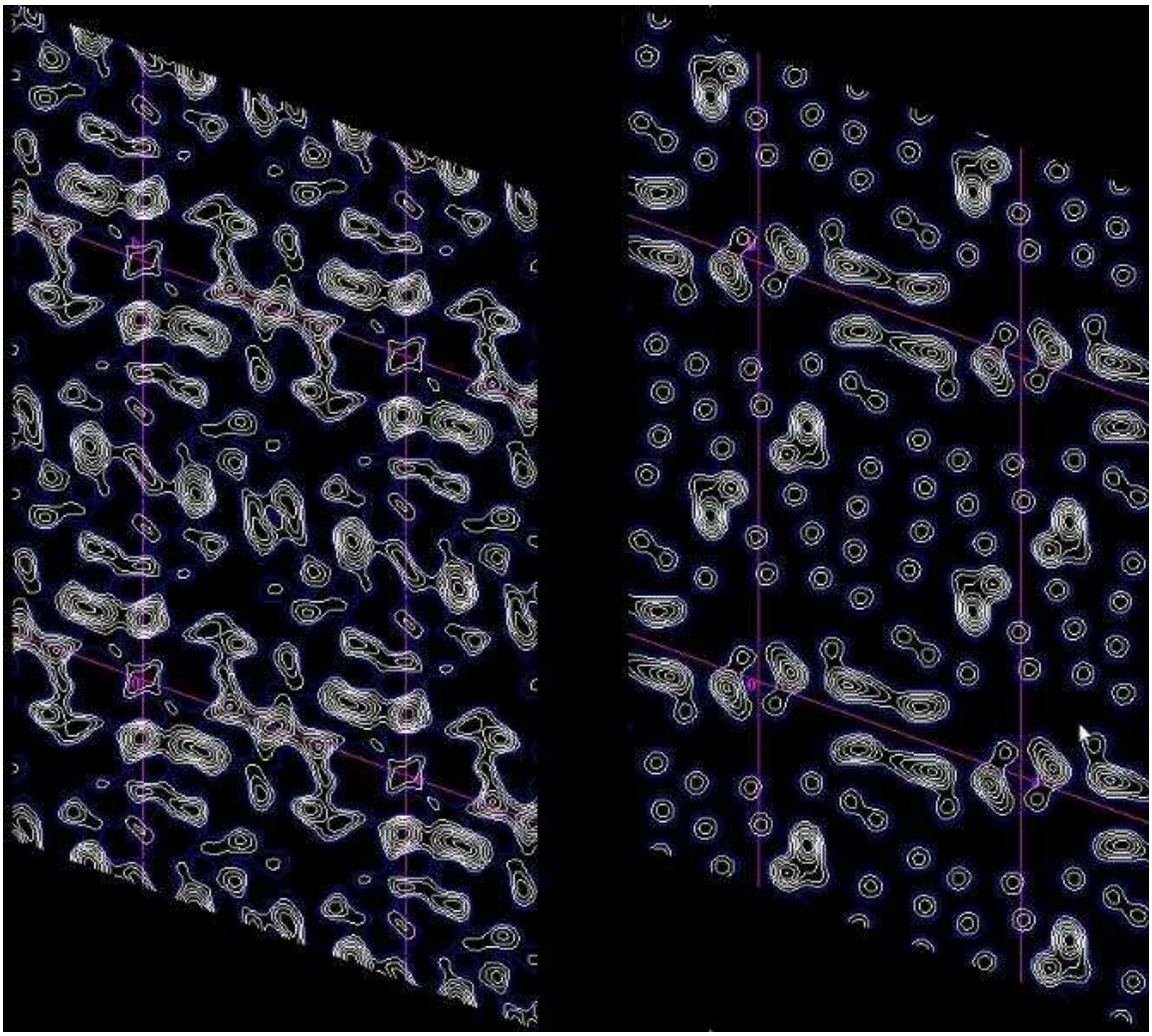

Figure 3-52. A comparison between the Fourier syntheses of the real electron density map of a structure (right), and the 'hybrid' structure factors provided by a neural network with 150 neurons in the hidden layer (left). (Hyperbolic tangent sigmoid transfer function- trained with 10000 samples)

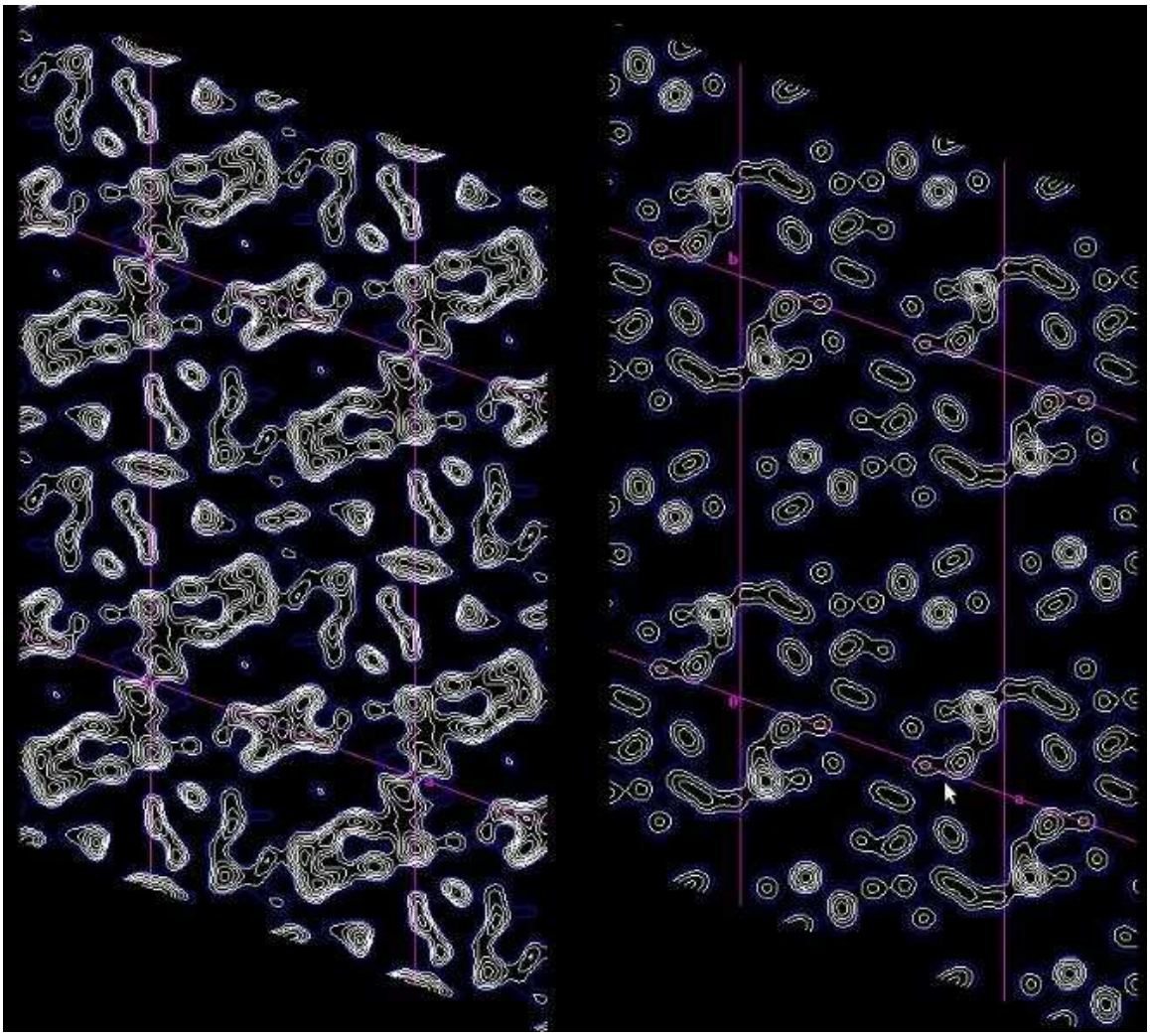

Figure 3-53. A comparison between the Fourier syntheses of the real electron density map of a structure (right), and the 'hybrid' structure factors provided by a neural network with 550 neurons in the hidden layer (left). (hyperbolic tangent sigmoid transfer function- trained with 10000 samples)

Conclusion

The phase problem was, and remains a major problem in X- ray crystallography. There is no formal relationship between the amplitudes and phases; the only relationship is via the molecular structure or electron density. Therefore, if someone can assume some prior knowledge of the electron density or structure, this can lead to values for the phases. Throughout this study that prior knowledge was given in the form of training patterns to the neural networks and it was hoped that neural networks would be able to generalize and represent relationships based on constraints of the electron density of molecules.

A large number of experiments have been conducted with a variety of feed forward neural networks architectures which can approximate functions arbitrarily well. In these experiments the performance of neural networks reached a plateau regardless the number of neurons used in the hidden layer. This plateau was different when modified structure factors (unitary, normalized) were presented to the network, and when the transfer function of the hidden layers changed. However, one thing in common was that all networks had small correlation coefficient values when the target outputs and the actual outputs of the networks were considered. This result showed uncorrelated values of target outputs and actual outputs. Even when only the signs of the outputs were taken into consideration and were compared with the correct signs, it seemed that neural networks assigned signs to structure factors in a random fashion. Finally, the possibility of neural networks producing phases, with an origin of coordinates fixed at a different point, other than the one used to calculate the structure factors of the training patterns, was considered. This possibility was examined by comparing the Fourier syntheses of the outputs of the neural network, and this comparison showed that this is not the case.

Generally as a conclusion, it can be said that the results of this study, showed that feed forward neural network are not capable of representing relationships between phases of structure factors or modified structure factors, at least not in a simple manner. This could be due to the fact that the restrictions of the electron density distributions are not sufficient to guarantee a unique solution ([2] p.268) and that maybe inhibits the learning procedure of neural networks. The possibility of a modified neural network or learning algorithm that could lead to a successful sign assignment has not been excluded. Further research could be conducted to determine such a network or algorithm and different approaches can be taken.

Suggestions for further research could be the use of radial basis transfer functions (although these also work as general function approximators) and the processing of inputs before their presentation to the network. Input patterns could be presented in a self-organizing feature map (such as a Kohonen network) trained using unsupervised learning. This network then could classify the input vectors according to some characteristics and

then these classes of inputs could be used to train different networks for sign assignment. This method could possibly help the training of networks.

Closing this study it should be mentioned that the phase problem remains a challenging field of research. Existing techniques have drawbacks, severely limiting the rate at which important new structures are solved. The structure determination of a number of biologically important molecules has been hampered by technical challenges regarding this issue. The development of an efficient phase determination technique could boost the progress of molecular biology (and related) sciences even further.

Figure Sources

Figure 1- 1. Phase difference between two waves
Source: http://www.radio-electronics.com/info/rf-technology-design/pll-synthesizers/phase-locked-loop-tutorial.php

Figure 1- 2. Comparison between a) light microscopy and b) X-ray diffraction
Source: Glusker, J.P. and Trueblood, K.N. Crystal structure analysis. A Primer. 2nd edition. Oxford University Press: New York, Oxford (1985)

Figure 1- 3. Relationship between Crystal lattice, Motif and Crystal structure
Source: [4] page 63

Figure 1- 4. Up: 3-D crystallographic unit cell with axes a,b,c. Down: 2-D lattice described by crystallographic unit vectors a,b
Source: Up: http://ictwiki.iitk.ernet.in/wiki/index.php/File:Jk1_7.png Down: http://www.sci.sdsu.edu/TFrey/Bio750/Bio750X-Ray.html

Figure 1- 5. Nature of electromagnetic waves
Source: http://hyperphysics.phy-astr.gsu.edu/hbase/waves/emwavecon.html

Figure 1- 6. Explanation of the diffraction pattern of a single slit. At higher angles the intensity of the diffracted beam is weak.
Source: [4] page 79

Figure 1- 9. Numerical values of the calculated electron density (a) at grid points and (b) at a two-dimensional contour plot, showing how contours are drawn in two dimensions
Source: [4] page 351

Figure 1- 10. Electron density map and model of Penicillin created by Dorothy Crowfoot Hodgkin in 1945 based on her work on X-ray crystallography
Source: http://dataphys.org/list/electron-density-map-and-molecular-model-of-penicillin/

Figure 1- 11. relationships between crystal lattices, reciprocal lattices, structure factors, and contents of the unit cell
Source: [4] page 89

Figure 1- 14. The importance of phases in carrying information. Top, the diffraction pattern, or Fourier transform (FT), of a duck and of a cat. Bottom left, a diffraction pattern derived by combining the amplitudes from the duck diffraction pattern with the phases from the cat diffraction pattern. Bottom right, the image that would give rise to this hybrid diffraction pattern. In the diffraction pattern, different colours show different phases and the brightness of the colour indicates the amplitude.
Source: [2]

Figure 2-11. Overfitting and poor extrapolation problems
Source:[12] chapter 13, page 14

Figure 2-12. Good generalization and poor extrapolation
Source: [12] chapter 13, page 14

Figure 3-1. (a) Notation with all the connections drawn. (b) Abbreviated notation.
Source: [12] chapter 2, page 8

Bibliography

1. Sherwood , Dennis. Crystals, X-rays and Proteins. John Wiley & Sons, 1976

2. Stout, George H., and Lyle H. Jensen. X-Ray Structure Determination: A practical guide. London: The Macmillan Company Collier-Macmillan Limited

3. Ladd, Mark, and Rex Palmer. Structure Determination by X-ray Crystallography. 5th edition, Springer, 2013

4. Glusker, Jenny P., Mitchell Lewis, and Miriam Rossi. Crystal Structure Analysis for Chemists and Biologists. John Wiley & Sons, 1994

5. Ladd, M.F.C., and R.A. Palmer. Theory and Practice of Direct Methods in Crystallography. New York: Plenum Press, 1980

6. Schenk, H. An Introduction to Direct Methods. The most Important Phase Relationships and their Application in Solving the Phase Problem. Wales: University College Cardiff Press, 1984

7. Hahn, Theo. International Tables for Crystallography, Volume A, Space-Group Symmetry. Wiley & Sons, 2006

8. Woolfson, M.M. An introduction to X-ray crystallography. Second edition, Cambridge University Press, 1997

9. Fausett, Laurene V. Fundamentals of Neural Networks: Architectures, Algorithms And Applications. 1st edition, Pearson, 1993

10. Halici, Ugur. Artificial Neural Networks. EE543 Lecture Notes, METU EEE, Ankara

11. Rumelhart, D.E., G.E. Hinton, and R.J. Williams. Parallel distributed processing: explorations in the microstructure of cognition, vol. 1. Pages 318-362, MIT Press, 1986

12. Hagan, Martin T.,Howard B Demuth, Mark H Beale, and Orlando De Jesús. Neural Network Design. 2nd edition, Martin Hagan, 2014

13. Bailer,Jones, Coryn A.L. , Ranjan Gupta, and Harinder P. Singh, eds. An introduction to artificial neural networks. Automated Data Analysis in Astronomy, Narosa Publishing House, New Delhi, India, 2001

14. Kröse ,Ben, and Patrick van der Smagt. An Introduction to Neural Networks. University of Amsterdam, 1996

15. Rojas, Raúl. Neural Networks, A Systematic Introduction. Springer-Verlag, Berlin, 1996

16. Kriesel, David. A brief introduction to Neural Networks. Available at: http://www.dkriesel.com , 2007

17. Suzuki, Kenji, Isao Horiba, and Noboru Sugie. A Simple Neural Network Pruning Algorithm with Application to Filter Synthesis. Netherlands: Kluwer Academic Publishers, 2001

18. Karsoliya, Saurabh. Approximating Number of Hidden layer neurons in Multiple Hidden Layer BPNN Architecture. International Journal of Engineering Trends and Technology, Volume3 Issue6, 2012

19. Moustafa , Akram A. Performance Evaluation of Artificial Neural Networks for Spatial Data Analysis. Contemporary Engineering Sciences, Vol. 4, no. 4, pages 149 – 163, 2011

20. Demuth ,Howard, and Mark Beale. Neural Network Toolbox, For Use with MATLAB: User's Guide. Version 4, Mathworks Inc, 2002

21. Porter, A.B. On the diffraction theory of microscopic vision. Phil.Mag. 11, pages 154-166 (1906)

22. Sayre,D. J.Kirz, R.Feder, B.Kim, and Spiller E. Potential operating region for ultrasoft X-ray microscopy of biological materials. Science 196, pages 1339-1340 (1977)

23. Binnig, G., H.Rohrer, Gerber C., and Weibel E. Tunneling through a controllable vacuum gap. J. Appl. Phys.40, pages 178-180 (1982)

24. Huygens, C. Treatise on light in which the causes of the events that result in reflection and refraction are explained. Of particular interest is the unusual refraction of Iceland spar. Pierre van der Aa: Leiden (1960). English translation: Thompson, S.P. Macmillan: London (1912)

25. Grifiths, David J. Introduction to Electrodynamics. Second edition, Prentice Hall, International. Sections 8.1.1 to 8.2.2

26. Ewald, P. P.. "Introduction to the dynamical theory of X-ray diffraction". Acta Crystallographica Section A 25: 103 (1969)

27. Taylor, G. The phase problem. Acta Crystallographica Volume 59, Part 11, Pages 1881-1890 (November 2003)

28. Institute of Medicine (US) Forum on Neuroscience and Nervous System Disorders.

29. From Molecules to Minds: Challenges for the 21st Century. Workshop Summary, US, Washington(DC). National Academies Press (2008) Available at: http://www.ncbi.nlm.nih.gov/books/NBK50989/
30. Azevedo, Frederico A.C. et al. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. Journal of Comparative Neurology,Volume 513, Issue 5, pages 532–541, 10 April 2009

31. Ismail A., and A. P. Engelbrecht Training product units in feedforward neural networks using particle swarm optimization. Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E4DB29BE6E86E86606E0496523262DF7?doi=10.1.1.33.8647&rep=rep1&type=pdf

32. Schuster, and Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45:2673–81, November 1997.

33. G Cybenko, Continuous-Valued Neural Networks with Two Hidden Layers Are Sufficient, Tehnical Report, Department of Computer Sciene, Tufts University, Medford, Massahusetts,1989.

34. Read ,Jesse, Albert Bifet , Bernhard Pfahringer , and Geoff Holmes. Batch Incremental versus Instance-Incremental Learning in Dynamic and Evolving Data. Advances in Intelligent Data Analysis XI, Lecture Notes in Computer Science Volume 7619, 2012, pages 313-323

35. Harrington ,Peter B. Sigmoid transfer functions in backpropagation neural networks. Anal. Chem., 65 (15), pages 2167–2168 (1993)

Electronic sources

E-1. http://reference.iucr.org/dictionary/Electron_density_map
E-2. http://www.rodenburg.org/theory/scatteringvector18.html
E-3. http://www.doitpoms.ac.uk/tlplib/crystallography3/unit_cell.php
E-4. http://www.ysbl.york.ac.uk/~cowtan/fourier/fourier.html
E-5. http://reference.iucr.org/dictionary/Isomorphous_crystals
E-6. http://utopia.duth.gr/~glykos/pepinsky.html

Appendix: Scripts and Programs

A.1 Structure data creation (C language program)

This is the program mentioned in section 2.3.1.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define getrand ((float)(random())/RAND_MAX)

/*Unit cell */
#define CELL_A 10.0
#define CELL_B 15.0
#define CELL_BETA 1.91986217719376 /*This is 110 degrees in rad */

/* Number of atoms per assymetric unit and their temperature factor */
#define NOF_ATOMS 30
#define TEMP_FACT 10

/* Maximum resolution for data in Amstrgong */
#define RESO 1.0

main()
{
    int data_set, data_sets;
    FILE *out;
    char filename[300];
    float reso;
    int h,k,i;
    float x[NOF_ATOMS];
    float y[NOF_ATOMS];
    float F;

    srandom( time(NULL));

    printf("Number of data sets to produce: ");
    scanf("%d", &data_sets);

    for(data_set=0; data_set<data_sets; data_set++)
    {
        sprintf(filename,"p2_%05d.dat", data_set);
        out=fopen(filename,"w");

        /*produce atomic positions */
        for(i=0; i<NOF_ATOMS; i++)
        {
```

```
            x[i]=getrand;
            y[i]=getrand/2.0;
        }

        /* for all reflection indeces ... */
        for(h=-(int)(CELL_A/RESO+1); h<=(int)(CELL_A/RESO+1); h++)
            for(k=0;k<=(int)(CELL_B/RESO+1);k++)
            {
                /* is resolution within limits? */

reso=sqrt(1.0/((1.0/(sin(CELL_BETA)*sin(CELL_BETA)))*(h*h/(CELL_A*CELL_
A)+k*k/(CELL_B*CELL_B)-(2*h*k*cos(CELL_BETA)/(CELL_A*CELL_B)) )));

                /*if yes, calculate structure factor and write out.
Apply temerature factor of TEMP_FACT A^2 */
                if(reso>=RESO && !(h==0&&k==0))
                {
                    F=0.0;
                    for(i=0;i<NOF_ATOMS;i++)
                        F+=2*cos(2*M_PI*(h*x[i]+k*y[i]))*exp(-
TEMP_FACT/(4.0*reso*reso));
                    fprintf(out, "%5d %5d %15.5f 1.0\n", h,k,F);
                }
            }


            fprintf(out, "\n");
            fclose(out);
    }
}
```

A.2 Matlab code that checks if all files created with the program in C contain the same number of reflections

```
function [status] = Reflection_check( num_of_files )
%
% If status is '1', files are OK -- same number of reflections in all
files

status=logical(0);
Num_of_anaklaseis=int8(0);

    for i=1:num_of_files
        string=strcat('reflection files/p2 (',num2str(i),').dat');
%file path
        string  %show the file path in workspace
        data=importdata(string);
        data=data(:,3);
            if(Num_of_anaklaseis==0)
                Num_of_anaklaseis=length(data);
```

```
            else
                status=Num_of_anaklaseis==length(data);
                    if(~status)
                        return
                    end
            end
    end

end
```

A.3 Matlab programs for experiments with structure factors as input data

A.3.1 Creation of training patterns suitable for neural networks

```
function [ neural_inputs, neural_outputs ] = Input_loader(
num_of_files, type, test )
%SYNTAX [inputs,outputs]=Input_loader (num_of_files, "type", "test")
%
%  num_of_files is the number of files (samples) that will be used
%  Type = output type  //  "Bipolar" (outputs=-1,1) "Binary"(outputs
0,1)
%"Structure"(output= structure factor with sign)
%  Test= "Train" if we need data from the training folder or "Test" for
the
%  test folder
%
%--It takes a files that contain reflections of structures (created by
the program in C)
% and returns as inputs of a neural network the absolute value of
structure
% factors and outputs depending on "Type" variable

structure_cell=cell(1);

if(strcmp(test,'Train'))   %check for train or test folder
    folder='reflection_files\';
elseif(strcmp(test,'Test'))
    folder='reflection_files\testing_reflection_files\';
else
    msg='There is a fault in the test argument'
    return
end

    for i=0:num_of_files-1
        %how many zeros for the filename (00001 klp) make filename
        if(i<=9)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_0000',num2str(i),'.dat');
```

```
        elseif(i<=99)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_000',num2str(i),'.dat');
        elseif(i<=999)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_00',num2str(i),'.dat');
        elseif(i<=9999);

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_0',num2str(i),'.dat');
        else

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_',num2str(i),'.dat');

        end

        %load
        data=importdata(string);
        structure_cell(i+1)={data(:,3)};
    end

%inputs regardless of type
neural_inputs=abs(cell2mat(structure_cell));

%outputs depending on type
if(strcmp(type,'Bipolar'))
    neural_outputs=sign(cell2mat(structure_cell));
elseif(strcmp(type,'Binary'))
    neural_outputs=double(cell2mat(structure_cell)>0);
elseif(strcmp(type,'Structure'))
    neural_outputs=double(cell2mat(structure_cell));
else
    msg='There is a fault in the type argument'
    return
end

end
```

A.3.2 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data

A.3.2.1 Neural network training and application – basic function

```
function [line_results]= Neural_test_function(sample_size,Num_neurons,
iteration_no)
% function with 1 hidden layer -- tansig transfer function
% line_results saves 9 columns
% 1: Neurons used in the neural
```

```
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced (iteration_no)

%input files
[inputs,targets]=Input_loader(sample_size,'Bipolar','Train');
[rows,columns]=size(inputs);

clear rows;

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
net = newff(inputs,targets,numHiddenNeurons);
net.trainFcn='trainscg';

net.divideParam.trainRatio = 60/100;  % Adjust as desired
net.divideParam.valRatio = 20/100;  % Adjust as desired
net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set max epochs = 4000
net.trainParam.epochs=4000;

%create filename string
filename_string=strcat('
1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'samples_
_iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

    %separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));

    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
```

```
    testOutputs=outputs(:,tr.testInd(:));

                        % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons;
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs

%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;

eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.3.2.2 Multiple applications of basic function

```
function [results] =
Testing_neural(sample_sizes,neuron_numbers_array,iterations)
%iterations--> how many times to re-initialize the network and try the
experiments again

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[row,numberOf_DifferentSingleHiddenExperiments]=size(neuron_numbers_arr
ay);
clear rows

%first check small samples first in all neural networks for faster
%production of results

%creation of results array
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii);

line_results=Neural_test_function(sample_size,Num_neurons,iteration_no)
;
                    %save in the index
                    results(index,:)=line_results;
                    index=index+1;
                end
            end

    end
end
```

A.3.2.3 Execute multiple applications of basic function for the experiments

```
function [void] = Executer

diary('log file___date_TANSIG')

matlabpool open

%WATCH RATIOS BEFORE EXECUTION!!!

%test with bipolar- tansig
 sample_array=[2000 4000 6000 10000];
```

```
 neuron_array=[5 10 20 30 40 50 100 150 200 250 300 350 400 500 600];
 results1=Testing_neural(sample_array,neuron_array,3);
 save ('_results', 'results');

matlabpool close

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.3.3 Networks with logistic tangent sigmoid transfer function and binary output data

A.3.3.1 Neural network training and application – basic function

```
function [line_results]=
Neural_test_function_binary(sample_size,Num_neurons,iteration_no)
%function with 1 hidden layer - logsig transfer function
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files
[inputs,targets]=Input_loader(sample_size,'Binary','Train');
[rows,columns]=size(inputs);

clear rows;

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
net = newfit(inputs,targets,numHiddenNeurons);
net.trainFcn='trainscg';

net.divideParam.trainRatio = 60/100;  % Adjust as desired
net.divideParam.valRatio = 20/100;  % Adjust as desired
```

```
net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set transfer function
net.layers{1}.transferFcn='logsig';

%set max epochs to 4000
net.trainParam.epochs=4000;

%create filename string
filename_string=strcat('
LOGSIG_1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'s
amples_4000epochs__iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

%separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));

    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
    testOutputs=outputs(:,tr.testInd(:));

                  % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons;
```

```
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs

%save all
command_string=strcat('save',filename_string,'Variables')
clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;

eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.3.3.2 Multiple applications of basic function

```
function [results] =
Testing_neural_binary(sample_sizes,neuron_numbers_array,iterations)

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[rows,numberOf_DifferentSingleHiddenExperiments]=size(neuron_numbers_ar
ray);
clear rows

%Results creation
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii);
```

```
line_results=Neural_test_function_binary(sample_size,Num_neurons,iterat
ion_no);
                    results(index,:)=line_results;
                    index=index+1;
                end
            end
    end
end
```

A.3.3.3 Execute multiple applications of basic function for the experiments

```
function [void] = Executer

diary('log file___date_logsig)

matlabpool open

%WATCH RATIOS BEFORE EXECUTION!!!

%test with binary- logsig
 sample_array=[2000 4000 6000 10000];
 neuron_array=[5 10 20 30 40 50 100 150 200 250 300 350 400 500 600];
 results1= Testing_neural_binary(sample_array,neuron_array,3);
 save ('_results', 'results');


matlabpool close

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.3.4 Networks with logistic tangent sigmoid transfer function and structure output data

A.3.4.1 Neural network training and application – basic function

```
function [line_results]=
Neural_test_function_structures(sample_size,Num_neurons, iteration_no)
% function me 1 hidden layer - tansig and structure output data
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
```

```
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files
[inputs,targets]=Input_loader(sample_size,'Structure','Train');
[rows,columns]=size(inputs);

clear rows;

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
net = newfit(inputs,targets,numHiddenNeurons);
net.trainFcn='trainscg';

net.divideParam.trainRatio = 60/100;  % Adjust as desired
net.divideParam.valRatio = 20/100;  % Adjust as desired
net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set transfer function
net.layers{1}.transferFcn='logsig';

%set max epochs to 4000
net.trainParam.epochs=4000;

%create filename string
filename_string=strcat('
LOGSIG_1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'s
amples_4000epochs_STRUCTURE__iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

   %separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));
```

```
    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
    testOutputs=outputs(:,tr.testInd(:));

                        % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons;
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs

%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;

eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;
```

```
end
```

A.3.4.2 Multiple applications of basic function

```
function [results] =
Testing_neural_structures(sample_sizes,neuron_numbers_array,iterations)
%iterations--> how many times to re-initialize the network and try the
experiments again

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[row,numberOf_DifferentSingleHiddenExperiments]=size(neuron_numbers_arr
ay);
clear rows

%results creation
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii);

line_results=Neural_test_function_structures(sample_size,Num_neurons,it
eration_no);

                    results(index,:)=line_results;
                    index=index+1;
                end
            end

    end
end
```

A.3.4.3 Execute multiple applications of basic function for the experiments

```
function [void] = Executer

diary('log file___date_structure)

matlabpool open

%WATCH RATIOS BEFORE EXECUTION!!!
```

```
%test with bipolar- tansig-structures
 sample_array=[2000 4000 6000 10000];
 neuron_array=[5 10 20 30 40 50 100 150 200 250 300 350 400 500 600];
 results1= Testing_neural_structures(sample_array,neuron_array,3);
 save ('_results', 'results');


matlabpool close

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.3.5 Product net input Neural Networks with hyperbolic tangent sigmoid transfer function and bipolar output data

A.3.5.1 Neural network training and application – basic function

```
function [line_results]=
Neural_test_function_netprod(sample_size,Num_neurons, iteration_no)
% function with 1 hidden layer and product instead of sum for net input
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files
[inputs,targets]=Input_loader(sample_size,'Bipolar','Train');
[rows,columns]=size(inputs);

clear rows;

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
```

```
net = newfit(inputs,targets,numHiddenNeurons);
net.trainFcn='trainscg';

net.divideParam.trainRatio = 60/100;  % Adjust as desired
net.divideParam.valRatio = 20/100;  % Adjust as desired
net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set net input function
net.layers{1}.netInputFcn='netprod';

%set max epochs
net.trainParam.epochs=4000;

%create filename string
filename_string=strcat('
PRODUCT_1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'
samples__iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

%separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));

    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
    testOutputs=outputs(:,tr.testInd(:));

                % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close


plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
```

```
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons;
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs

%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;

eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.3.5.2 Multiple applications of basic function

```
function [results] =
Testing_neural_product(sample_sizes,neuron_numbers_array,iterations)
%iterations--> how many times to re-initialize the network and try the
experiments again

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[row,numberOf_DifferentSingleHiddenExperiments]=size(neuron_numbers_arr
ay);
clear rows

%Results creation
```

```
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii);

line_results=Neural_test_function_netprod(sample_size,Num_neurons,itera
tion_no);

                    results(index,:)=line_results;
                    index=index+1;
                end
            end
    end
 end
```

A.3.5.3 Execute multiple applications of basic function for the experiments

```
function [void] = Executer

diary('log file___date_product)

matlabpool open

%WATCH RATIOS BEFORE EXECUTION!!!

%test with bipolar- tansig -- product
 sample_array=[2000 4000 6000 10000];
 neuron_array=[5 10 20 30 40 50 100 150 200 250 300 350 400 500 600];
 results1= Testing_neural_product(sample_array,neuron_array,3);
 save ('_results', 'results');


matlabpool close

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.3.6 Neural Networks with 2 hidden layers and hyperbolic tangent sigmoid transfer functions and bipolar output data

A.3.6.1 Neural network training and application – basic function

```
function [line_results]=
Neural_test_function_layers_epochs(sample_size,Num_neurons,max_epochs,
iteration_no)
% function with 2 hidden layers
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced


%input files
[inputs,targets]=Input_loader(sample_size,'Bipolar','Train');
[rows,columns]=size(inputs);
[rows,no_hidden_layers]=size(Num_neurons);
clear rows;


% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_Neurons;
net = newfit(inputs,targets,numHiddenNeurons);
net.trainFcn='trainscg';

net.divideParam.trainRatio = 60/100;  % Adjust as desired
net.divideParam.valRatio = 20/100;  % Adjust as desired
net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set max epochs
net.trainParam.epochs=max_epochs;


%create filename string
filename_string=strcat(' _',num2str(no_hidden_layers),'hidden_');
```

```
   for i=1:no_hidden_layers

filename_string=strcat(filename_string,num2str(numHiddenNeurons(i)),'-
');
    end
filename_string=strcat(filename_string,'neurons',num2str(columns),'samp
les_',num2str(max_epochs),'maxEpochs__iteration_',num2str(iteration_no)
,'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');


 %separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));

    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
    testOutputs=outputs(:,tr.testInd(:));


                  % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons(1);
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
```

```
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs


%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;

eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.3.6.2 Multiple applications of basic function

```
function [results] =
Testing_neural_multiple_layers(sample_sizes,neuron_numbers_array,
epochs, iterations)
%iterations--> how many times to re-initialize the network and try the
experiments again

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[numberOf_DifferentSingleHiddenExperiments,columns]=size(neuron_numbers
_array);
clear columns

%results creation
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii,:);
```

```
line_results=Neural_test_function_layers_epochs(sample_size,Num_neurons
, epochs,iteration_no);

                    results(index,:)=line_results;
                    index=index+1;
                end
            end
    end
end
```

A.3.6.3 Execute multiple applications of basic function for the experiments

```
function [void] = Executer

diary('log file___date__2_hidden_tansig)

matlabpool open

%WATCH RATIOS BEFORE EXECUTION!!!

sample_array=[4000 6000 10000];

neurons=[5 5];
results1=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results1', 'results1');

neurons=[10 10];
results2=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results2', 'results2');

neurons=[20 20];
results3=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results3', 'results3');

neurons=[40 40];
results4=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results4', 'results4');

neurons=[50 50];
results5=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results5', 'results5');

neurons=[100 100];
results6=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results6', 'results6');


neurons=[150 150];
results7=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results7', 'results7');
```

```
neurons=[200 200];
results8=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results8', 'results8');


neurons=[250 250];
results9=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results9', 'results9');


neurons=[300 300];
results10=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results10', 'results10');


neurons=[350 350];
results11=Testing_neural_multiple_layers(sample_array,neurons,4000,3);
save ('results11', 'results11');


matlabpool close

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.4 Matlab programs for experiments with unitary structure factors as input data

A.4.1 Creation of training patterns suitable for neural networks

```
function [ neural_inputs, neural_outputs ] = Input_loader_unitary(
num_of_files, type, test )

%works with unitary structure files from ref_files
%
%IMPORTANT!!!!
% REMEMBER TO CHANGE NUMBER OF ATOMS IF NECESSARY (how many atoms are
% supposed to be in the ref_files (p2_xxxx.dat) )

%SYNTAX [inputs,outputs]=Input_loader (num_of_files, "type", "test")
%
%  num_of_files is the number of files (samples) that will be used
```

```
%  Type = output type  //  "Bipolar" (outputs=-1,1) "Binary"(outputs
0,1)
%"Structure"(output= structure factor with sign)
%  Test= "Train" if we need data from the training folder or "Test" for
the test folder
%
%--It takes a file that contains reflections of structures (created by
the program in C)
% and returns as inputs of a neural network the absolute value of
unitary structure factors
% factors and outputs depending on "Type" variable

number_of_atoms=30;

structure_cell=cell(1);

if(strcmp(test,'Train'))   %check for training or test folder
    folder='reflection_files\';
elseif(strcmp(test,'Test'))
    folder='reflection_files\testing_reflection_files\';
else
    msg='There is a fault in the test argument'
    return
end

    for i=0:num_of_files-1
        %how many zeros for the filename (00001 klp) make filename
        if(i<=9)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_0000',num2str(i),'.dat');
        elseif(i<=99)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_000',num2str(i),'.dat');
        elseif(i<=999)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_00',num2str(i),'.dat');
        elseif(i<=9999);

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_0',num2str(i),'.dat');
        else

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
p2_',num2str(i),'.dat');

        end

        %load
        data=importdata(string);
        structure_cell(i+1)={data(:,3)/(2*number_of_atoms)};
    end

%inputs regardless of type
```

```
neural_inputs=abs(cell2mat(structure_cell));

%outputs depending on type
if(strcmp(type,'Bipolar'))
    neural_outputs=sign(cell2mat(structure_cell));
elseif(strcmp(type,'Binary'))
    neural_outputs=double(cell2mat(structure_cell)>0);
elseif(strcmp(type,'Structure'))
    neural_outputs=double(cell2mat(structure_cell));
else
    msg='There is a fault in the type argument'
    return
end

end
```

A.4.2 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data

A.4.2.1 Neural network training and application – basic function

The code is the same as the code in A.3.2.1 but instead of line:

```
[inputs,targets]=Input_loader(sample_size,'Bipolar','Train');
```

We use the line:

```
[inputs,targets]=Input_loader_unitary(sample_size,'Bipolar','Train');
```

A.4.2.2 Multiple applications of basic function

Same as the code in section A.3.2.2

A.4.2.3 Execute multiple applications of basic function for the experiments

Same as the code in section A.3.2.3

A.4.3 Remaining experiments with unitary structure factors.

The codes are the same as described for section A.3. The difference is that instead of function “Input_loader”, the function “Input_loader_unitary” is used, when necessary (as described in section A.4.2.1).

A.5 Programs for experiments with unitary structure factors as input data

A.5.1 Structure data creation for unitary structure factor experiments (C language program)

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define getrand ((float)(random())/RAND_MAX)

/*Unit cell */
#define CELL_A 10.0
#define CELL_B 15.0
#define CELL_BETA 1.91986217719376 /*This is 110 degrees in rad */

/* Number of atoms per assymetric unit and their temperature factor */
#define NOF_ATOMS 30
#define TEMP_FACT 10

/* Maximum resolution for data in Amstrgong */
#define RESO 1.0

main()
{
    int data_set, data_sets;
    FILE *out, *uni_out;
    char filename[300];
    char uni_filename[300];
    float reso;
    int h,k,i;
    float x[NOF_ATOMS];
    float y[NOF_ATOMS];
    float F;
    float E;

    srandom( time(NULL));

    printf("Number of data sets to produce: ");
    scanf("%d", &data_sets);

    for(data_set=0; data_set<data_sets; data_set++)
    {
        sprintf(filename,"p2_%05d.dat", data_set);
      sprintf(uni_filename,"evalue_p2_%05d.dat", data_set);
        out=fopen(filename,"w");
```

```
      uni_out=fopen(uni_filename,"w");

        /*produce atomic positions */
        for(i=0; i<NOF_ATOMS; i++)
        {
            x[i]=getrand;
            y[i]=getrand/2.0;
        }

        /* for all reflection indeces ... */
        for(h=-(int)(CELL_A/RESO+1); h<=(int)(CELL_A/RESO+1); h++)
            for(k=0;k<=(int)(CELL_B/RESO+1);k++)
            {
                /* is resolution within limits? */

reso=sqrt(1.0/((1.0/(sin(CELL_BETA)*sin(CELL_BETA)))*(h*h/(CELL_A*CELL_
A)+k*k/(CELL_B*CELL_B)-(2*h*k*cos(CELL_BETA)/(CELL_A*CELL_B)) )));

                /*if yes, calculate structure factor and write out.
Apply temerature factor of TEMP_FACT A^2 */
                if(reso>=RESO && !(h==0&&k==0))
                {
                    F=0.0;
                E=0.0;
                    for(i=0;i<NOF_ATOMS;i++){
                        F+=2*cos(2*M_PI*(h*x[i]+k*y[i]))*exp(-
TEMP_FACT/(4.0*reso*reso));
                  E+=2*cos(2*M_PI*(h*x[i]+k*y[i]));
                  }
                    fprintf(out, "%5d %5d %15.5f 1.0\n", h,k,F);
                fprintf(uni_out, "%5d %5d %15.5f 1.0\n", h,k,E);
                }
            }


          fprintf(out, "\n");
        fprintf(uni_out, "\n");
          fclose(out);
        fclose(uni_out);
    }
}
```

A.5.2 Creation of training patterns suitable for neural networks

```
function [ neural_inputs, neural_outputs ] = Input_loader_evalue(
num_of_files, type, test )

%SYNTAX [inputs,outputs]=Input_loader (num_of_files, type, test?)

structure_cell=cell(1);

%file paths below lead to the evalue_p2_xxxx.dat files created by the c
program in section A.4.1
```

```
if(strcmp(test,'Train'))   %check for training or test folders
    folder='ref_files_me_evalues\eval\';
elseif(strcmp(test,'Test'))
    folder='ref_files_me_evalues\test_set\eval\';
else
    msg='There is a fault in the test argument'
    return
end


    for i=0:num_of_files-1
        %how many zeros for the filename (00001 klp) make filename
        if(i<=9)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_0000',num2str(i),'.dat');
        elseif(i<=99)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_000',num2str(i),'.dat');
        elseif(i<=999)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_00',num2str(i),'.dat');
        elseif(i<=9999);

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_0',num2str(i),'.dat');
        else

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_',num2str(i),'.dat');

        end


        %load
        data=importdata(string);
        structure_cell(i+1)={data(:,3)};
    end

%inputs regardless of type
neural_inputs=abs(cell2mat(structure_cell));

%outputs depending on type
if(strcmp(type,'Bipolar'))
    neural_outputs=sign(cell2mat(structure_cell));
elseif(strcmp(type,'Binary'))
    neural_outputs=double(cell2mat(structure_cell)>0);
elseif(strcmp(type,'Structure'))
    neural_outputs=double(cell2mat(structure_cell));
else
    msg='There is a fault in the type argument'
    return
end
```

```
end
```

A.5.3 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data

A.5.3.1 Neural network training and application – basic function

The code is the same as the code in A.3.2.1 but instead of line:

```
[inputs,targets]=Input_loader(sample_size,'Bipolar','Train');
```

We use the line:

```
[inputs,targets]=Input_loader_evalue(sample_size,'Bipolar','Train');
```

A.5.3.2 Multiple applications of basic function

The code is the same as the code in A.3.2.2

A.5.3.3 Execute multiple applications of basic function for the experiments

The code is the same as the code in A.3.2.3

A.5.4 Remaining experiments with unitary structure factors.

The codes are the same as described for section A.3. The difference is that instead of function “Input_loader”, the function “Input_loader_evalue” is used, when necessary (as described in section A.5.3.1).

A.6 Programs for experiments with cascade forward nets

A.6.1 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data—Maximum validation checks set to 100 - basic function

```
function [line_results]=
Neural_test_function_cascade(sample_size,Num_neurons, iteration_no)
% function with 1 hidden layer -cascade network
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files
[inputs,targets]=Input_loader_evalue(sample_size,'Bipolar','Train');
[rows,columns]=size(inputs);

clear rows;

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
net = cascadeforwardnet(numHiddenNeurons);
net.trainFcn='trainscg';

 net.divideParam.trainRatio = 60/100;  % Adjust as desired
 net.divideParam.valRatio = 20/100;  % Adjust as desired
 net.divideParam.testRatio = 20/100;  % Adjust as desired

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set max epochs = 4000
net.trainParam.epochs=4000;

net.trainParam.max_fail=100;

%create filename string
```

```
filename_string=strcat('
Cascade_1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'
samples_EVALUES_iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

net=init(net);
% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

    %separate your training,validation and test data sets and results
    trainTargets=targets(:,tr.trainInd(:));
    validTargets=targets(:,tr.valInd(:));
    testTargets=targets(:,tr.testInd(:));

    trainOutputs=outputs(:,tr.trainInd(:));
    validOutputs=outputs(:,tr.valInd(:));
    testOutputs=outputs(:,tr.testInd(:));

                % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',validTargets,validOutp
uts,'Validation',testTargets,testOutputs,'Test',targets,outputs,'Overal
l');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close


% Line result implementation
line_results(1)=numHiddenNeurons;
line_results(2)=sample_size;
line_results(3)=tr. best_vperf;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
x=corrcoef(validTargets,validOutputs);
line_results(5)=x(2);
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
x=corrcoef(targets,outputs);
line_results(7)=x(2); clear x;
line_results(8)=tr.gradient(end);
```

```
line_results(9)=tr.num_epochs;   %NOT tr.best_epochs

%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;


eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end


%***** NOTE: for the same experiment with maximum validation checks =6
just set the variable net.trainParam.max_fail=6.  ***************
```

A.6.2 Networks with hyperbolic tangent sigmoid transfer function and bipolar output data—No validation set - basic function

```
function [line_results]=
Neural_test_function_no_val(sample_size,Num_neurons, iteration_no)
% function me 1 hidden layer -cascede network -no validation set
% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression)
% 6: Test R-value (regression)
% 7: Overall R-value (regression)
% 8: Gradient at the LAST epoch
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files
[inputs,targets]=Input_loader_evalue(sample_size,'Bipolar','Train');
[rows,columns]=size(inputs);

clear rows;
```

```
% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_neurons;
net = cascadeforwardnet(numHiddenNeurons);
net.trainFcn='trainscg';

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=true;

%set max epochs = 4000
net.trainParam.epochs=4000;

%net.trainParam.max_fail=100;

net.divideFcn='';

%create filename string
filename_string=strcat('
Cascade_1hidden',num2str(numHiddenNeurons),'neurons',num2str(columns),'
samples_EVALUES_iteration_',num2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

net=init(net);
% Train and Apply Network
[net,tr] =
train(net,inputs,targets,'useParallel','yes','showResources','yes');
outputs = sim(net,inputs,'useParallel','yes','showResources','yes');

%validation final outputs --> normal, not for mse
[val_inputs, val_targets2 ] = Input_loader(3000, 'Bipolar','Test' );
val_outputs2=sim(net,val_inputs,'useParallel','yes','showResources','no
');

    %separate your training,validation and test data sets and results
     trainTargets=targets(:,tr.trainInd(:));
%     validTargets=targets(:,tr.valInd(:));
     testTargets=val_targets2;
%
     trainOutputs=outputs(:,tr.trainInd(:));
%     validOutputs=outputs(:,tr.valInd(:));
     testOutputs=val_outputs2;


                    % Plot  and save -- then clear memory
plotperform(tr)
%print -dtiffn 1hidden2neurons100samples_PERFORMANCE
command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
eval(command_string)
close

plottrainstate(tr)
```

```
command_string=strcat('print -djpeg',filename_string,'_TRAIN_STATE');
eval(command_string)
close

plotregression(trainTargets,trainOutputs,'Train',testTargets,testOutput
s,'Test');
%print -dtiffn 1hidden2neurons100samples_REGRESSION
command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
eval(command_string)
close

% Line result implementation
line_results(1)=numHiddenNeurons;
line_results(2)=sample_size;
line_results(3)=NaN;
x=corrcoef(trainTargets,trainOutputs);
line_results(4)=x(2);
%x=corrcoef(validTargets,validOutputs);
line_results(5)=NaN;
x=corrcoef(testTargets,testOutputs);
line_results(6)=x(2);
%x=corrcoef(targets,outputs);
line_results(7)=NaN; clear x;
line_results(8)=NaN;
line_results(9)=NaN;    %NOT tr.best_epochs

%save all variables
command_string=strcat('save',filename_string,'Variables')

clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;


eval(command_string)

%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.7 Programs for experiments with neural network and origins – modified early stopping algorithm

A.7.1 Creation of training patterns suitable for neural networks with modified early stopping algorithm

The "input_loader_evalue" function is used (described in A.4.2) as well as the function that follows:

```
function [ neural_inputs, neural_outputs ] = Input_loader_org2_evalue(
num_of_files, test )
%SYNTAX [inputs,outputs]=Input_loader_org2_evalue (num_of_files, test?)
%
%for the origins experiment --> gives the validation set where inputs=
target outputs

structure_cell=cell(1);

if(strcmp(test,'Train'))   %check for train or test folder
    folder='ref_files_me_evalues\eval\';
elseif(strcmp(test,'Test'))
    folder='ref_files_me_evalues\test_set\eval\';
else
    msg='There is a fault in the test argument'
    return
end

    for i=0:num_of_files-1
        %how many zeros for the filename (00001 klp) make filename
        if(i<=9)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_0000',num2str(i),'.dat');
        elseif(i<=99)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_000',num2str(i),'.dat');
        elseif(i<=999)

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_00',num2str(i),'.dat');
        elseif(i<=9999);

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_0',num2str(i),'.dat');
        else

string=strcat('C:\Users\Dimitris_bio\Documents\MATLAB\matlab\',folder,'
evalue_p2_',num2str(i),'.dat');

        end


        %load
        data=importdata(string);
```

```
        structure_cell(i+1)={data(:,3)};
    end

%inputs regardless of type
neural_inputs=abs(cell2mat(structure_cell));
%outputs for structure type
neural_outputs=neural_inputs;

end
```

A.7.2 Neural networks with modified early stopping algorithm —basic function

```
function [line_results]=
Neural_test_function_structures_org3_no_paral(sample_size,Num_neurons,i
teration_no,valid)
% function me 1 hidden layer  valid==max_validation checks
%modified early stopping!!

% line_results saves 9 columns
% 1: Neurons used in the neural
% 2: Samples used
% 3: Best Validation Performance
% 4: Train R-value (regression)
% 5: Validation R-value (regression) I-O (apo ena simeio kai meta)
% 6: Validation R-value (regression) signed
% 7: Train R-value absolutes (regression)
% 8: Gradient at the LAST epoch -> not applied here
% 9: Number of (all) epochs
%this line is used in the excell presentantion as part of an array
(line)
%which iteration (re-initialization) of the neural is this? used in the
%file name produced

%input files

[inputs,targets]=Input_loader_evalue(sample_size,'Structure','Train');
[rows,columns]=size(inputs);

clear rows;

%validation set input
max_validation_input=4000; %how many files in testing folder

val_size=uint16((sample_size/100)*30);          %30 train set
if(val_size>max_validation_input)
val_size=max_validation_input;
end
[val_inputs, val_targets ] = Input_loader_org2_evalue( val_size, 'Test'
);
```

```
number_of_epochs=0; %counts how many epochs are needed for the best
performance

% Create Network
numHiddenNeurons = Num_neurons;  % Adjust as desired
clear Num_Neurons;
net = cascadeforwardnet(numHiddenNeurons);
net.trainFcn='trainscg';

net.divideFcn='';

%set trainParam
net.trainParam.showWindow=false;
net.trainParam.showCommandLine=false;

%set transfer function
net.layers{1}.transferFcn='tansig';

net=init(net);

%create filename string
filename_string=strcat('
TANSIG_ORG3_1hidden',num2str(numHiddenNeurons),'neurons',num2str(column
s),'samples_4000epochs_EVALUE_Valid_',num2str(valid),'__iteration_',num
2str(iteration_no),'_');
desktop_message=strcat('training',filename_string)
clear desktop_message;

%set initial performance value
val_outputs=sim(net,val_inputs,'useParallel','yes','showResources','yes
');
val_outputs=abs(val_outputs);

old_perf = mse(net,val_targets,val_outputs)

index_i=1; %index that helps to fill the performance array
perf_array(1)=old_perf;
number_of_epochs=0;
epoch_array(1)=0;

outputs=sim(net,inputs,'useParallel','yes','showResources','no');
train_perf=mse(net,targets,outputs);
train_perf_array(index_i)=train_perf;

%set initial max epochs to 2 (not 1- does not work)
net.trainParam.epochs=2;

max_validation=0;

new_net=net;
%----- the iterations for 4000 epochs max start HERE **********

for ii=1:4000
    % Train Network external iteration
```

```
    [new_net,new_tr] =
train(new_net,inputs,targets,'useParallel','no','showResources','yes');

    %train outputs and performance

outputs=sim(new_net,inputs,'useParallel','no','showResources','no');
    train_perf= mse(new_net,targets,outputs);
    index_i=index_i+1;
    train_perf_array(index_i)=train_perf;
    number_of_epochs=number_of_epochs+1;

    epoch_array(index_i)=number_of_epochs;

    %validation outputs and performance

val_outputs=sim(new_net,val_inputs,'useParallel','no','showResources','
no');
    val_outputs=abs(val_outputs);

    new_perf = mse(new_net,val_targets,val_outputs);
    perf_array(index_i)=new_perf;

    %show progress every ten epochs in workspace
    if(mod(number_of_epochs,10)==0)number_of_epochs
        new_perf
    end

    %all good continue training
    if(new_perf<old_perf)
        old_perf=new_perf;
        net=new_net;
        tr=new_tr;
        %reset validation check
        max_validation=0;
    end

    %worse performance--> go to previous state and decrease epochs
    if(new_perf>old_perf)

         max_validation=max_validation+1;
    end

     %max validation check reached --
    if(max_validation==valid)
        msg='max_validation_reached'
         break
    end

end

%for testing

val_outputs=sim(net,val_inputs,'useParallel','yes','showResources','no'
);
     val_outputs=abs(val_outputs);
```

```
     test_perf = mse(net,val_targets,val_outputs);


%train outputs
outputs = sim(net,inputs,'useParallel','yes','showResources','no');

%validation final outputs --> normal, not for mse
[val_inputs, val_targets2 ] = Input_loader_evalue( val_size,
'Structure','Test' );
val_outputs2=sim(net,val_inputs,'useParallel','no','showResources','no'
);

perf_array
tr.epoch=epoch_array;
tr.perf=perf_array;
tr.vperf=train_perf_array;
[col,rows]=size(epoch_array);
tr.num_epochs=rows-1;
tr.tperf=train_perf_array; %just for debugging
tr.best_vperf=test_perf;
tr.best_epoch=rows-1-valid;

                    % Plot  and save -- then clear memory
 plotperform(tr)
 %plot(epoch_array,perf_array,epoch_array,train_perf_array,'--')
% plot(perf_array,epoch_array,train_perf_array,epoch_array,'--')
%close
%semilogy(epoch_array,perf_array,epoch_array,train_perf_array,'--')
  command_string=strcat('print -djpeg',filename_string,'_PERFORMANCE');
  eval(command_string)
  close

  plot(epoch_array,perf_array,epoch_array,train_perf_array,'--')
  command_string=strcat('print -
djpeg',filename_string,'_PERFORMANCE2');
  eval(command_string)
  close

plotregression(targets,outputs,'Train',val_targets2,val_outputs2,'Valid
ation-Test signed',abs(targets),abs(outputs),'Train
Absolutes',val_targets,val_outputs,'Validation absolutes I-O');  %the
last is used in mse
 %print -dtiffn 1hidden2neurons100samples_REGRESSION
 command_string=strcat('print -djpeg',filename_string,'_REGRESSION');
 eval(command_string)
 close

% Line result implementation
line_results(1)=numHiddenNeurons;
 line_results(2)=sample_size;
 line_results(3)=test_perf;
 x=corrcoef(targets,outputs);
 line_results(4)=x(2);
 x=corrcoef(val_targets,val_outputs);
```

```
 line_results(5)=x(2);
 x=corrcoef(val_targets2,val_outputs2);
 line_results(6)=x(2);
 x=corrcoef(abs(targets),abs(outputs));
 line_results(7)=x(2);
 line_results(8)=NaN;
 line_results(9)=number_of_epochs;    %NOT tr.best_epochs

%save all
filename_string=strcat(filename_string,'Variables');
%save(filename_string, 'net','tr');
save(filename_string, 'net','tr');
clear columns;      %clear what you don't need to be saved
clear filename_string;
clear trainTargets trainOutputs validTargets validOutputs testTargets
testOutputs;



%clear all except line_results
clear Num_neurons command_string inputs net numHiddenNeurons outputs
sample_size targets testTargets tr;

close all;

end
```

A.7.3 Multiple applications of basic function

```
function [results] =
Testing_neural_structures_org3_no_paral(sample_sizes,neuron_numbers_arr
ay,iterations,valid)
%iterations--> how many times to re-initialize the network and try the
experiments again

[rows,numberOf_SampleExperiments]=size(sample_sizes);
[row,numberOf_DifferentSingleHiddenExperiments]=size(neuron_numbers_arr
ay);
clear rows

%results creation
results=zeros(
(numberOf_SampleExperiments*numberOf_DifferentSingleHiddenExperiments*i
terations ),9);
index=1;

    for i=1:numberOf_SampleExperiments
        sample_size=sample_sizes(i);
            for ii=1:numberOf_DifferentSingleHiddenExperiments
                for iteration_no=1:iterations
                    Num_neurons=neuron_numbers_array(ii);
```

```
line_results=Neural_test_function_structures_org3_no_paral(sample_size,
Num_neurons,iteration_no,valid);

                    results(index,:)=line_results;
                    index=index+1;
                end
            end
    end
 end
```

A.7.4 Execute multiple applications of basic function for the experiments (example)

```
function [void] = Executer

diary('Origin_evalues_tansig_experiment')


%matlabpool open----→ optional command, in this program sometimes
causes problems because training procedures are called multiple times
for short periods
matlabpool open

samples=10000;
neuron_array=[20 300 500];

%maximum validation checks are set to 60
results=Testing_neural_structures_org3_no_paral(samples,neuron_array,1,
60);
save ('_results', 'results');

matlabpool close    % only if matlabpool open is used at the start of
the program…

%play alert sound
Data = load('handel.mat');
sound(Data.y, Data.Fs)

diary off

end
```

A.8 Creation of histogram plots

A.8.1 Separation of values that should represent positive or negative phases at the output of a neural network

```
function [zeros,ones]= zeros_ones(filename)
% for binary output "0-1"
%and for bipolar (instead of zeros in zeros variable -1 are saved)
%For structures data type this function makes no sense

load(filename);
%tar_sum=sum(targets(:))
[rows,columns]=size(targets);

%fill ones and zeros- creates an array that saves which values should
be 1
%in ones variable and which values should be zero 0 at zeros variable
%(or -1 for bipolar outpu data)

ones=1;
zeros=1;
x=1;
y=1;
for row=1:rows;
    for column=1:columns;
        if(targets(row,column)==1)
            ones(x)=outputs(row,column);
            x=x+1;
        else
            zeros(y)=outputs(row,column);
            y=y+1;
        end
    end
end

end
```

A.8.2 Histogram creation

```
function [void]= histogram_plot(filename,bins)
%SYNTAX [void]= histogram_plot(filename,bins)

% zeros,ones function
[zeros,ones]=zeros_ones(filename);
filename=strcat(' _',filename);
filename=regexprep(filename,'Variables','Histogram');
```

```
hold off

%plot them together
hist(ones,bins);

%set red colour for "ones"
h = findobj(gca,'Type','patch');
set(h,'FaceColor','r','EdgeColor','b')


hold on;
hist(zeros,bins);

    %save plot
command_string=strcat('print -djpeg',filename,'.jpeg');
eval(command_string)
close

end
```