Wavelet Neural Networks

and their application in the study of dynamical systems

David Veitch

Dissertation submitted for the MSc in Data Analysis, Networks and Nonlinear Dynamics.

Department of Mathematics
University of York
UK

August 2005

Abstract

The main aim of this dissertation is to study the topic of wavelet neural networks and see how they are useful for dynamical systems applications such as predicting chaotic time series and nonlinear noise reduction. To do this, the theory of wavelets has been studied in the first chapter, with the emphasis being on discrete wavelets. The theory of neural networks and its current applications in the modelling of dynamical systems has been shown in the second chapter. This provides sufficient background theory to be able to model and study wavelet neural networks. In the final chapter a wavelet neural network is implemented and shown to accurately estimate the dynamics of a chaotic system, enabling prediction and enhancing methods already available in nonlinear noise reduction.

Contents

Notations

- Orthogonal : Two elements v_1 and v_2 of an inner product space E are called orthogonal if their inner product $\langle v_1, v_2 \rangle$ is 0.
- Orthonormal : Two vectors v_1 and v_2 are orthonormal if they are orthogonal and of unit length.
- Span : The span of subspace generated by vectors v_1 and $v_2 \in \mathbb{V}$ is

$$\text{span}(v_1, v_2) \equiv \{rv_1 + sv_2 | r, s \in \mathbb{R}\}.$$

- $\oplus$: Direct Sum.
  The direct sum $U \oplus V$ of two subspaces U and V is the sum of subspaces in which U and V have only the zero vector in common. Each $u \in U$ is orthogonal to every $v \in V$.
- $L^2(\mathbb{R})$: Set of square integrable real valued functions.

$$L^2(\mathbb{R}) = \int_{-\infty}^{\infty} |f(x)|^2 dx < \infty$$

- $C^k(\mathbb{R})$: A function is $C^k(\mathbb{R})$ if it is differentiable for k degrees of differentiation.
- Spline : A piecewise polynomial function.
- $\Psi(f)$: Fourier Transform of $\psi(u)$.

$$\Psi(f) \equiv \int_{-\infty}^{\infty} \psi(u) e^{-i2\pi fu} du$$

- $\{F_k\}$: Orthonormal Discrete Fourier Transformation of $\{X_t\}$.

$$F_k \equiv \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} X_t e^{-i2\pi tk/N} \quad \text{for } k = 0, \ldots, N-1$$

- MAD : Median Absolute Deviation, standard deviation estimate.

$$\sigma^2_{(mad)} \equiv \text{median}(|x_i - \text{median}(x_i)|) \quad \text{for } x_i \sim Gaussian$$

- $\text{sign}\{x\} \equiv \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$
- $(x)_+ \equiv \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{else} \end{cases}$
- Sigmoid Function : The function $f : \mathbb{R} \to [0, 1]$ is a sigmoid function if
  (1) $f \in C^\infty(\mathbb{R})$
  (2) $\lim_{x \to \infty} f(x) = 1$ and $\lim_{x \to -\infty} f(x) = 0$
  (3) f is strictly increasing on $\mathbb{R}$
  (4) f has a single point of inflexion c with
    (a) f' is strictly increasing on the interval $(-\infty, c)$
    (b) $2f(c) - f(x) = f(2c - x)$

- Lorenz Map :

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = rx - y - xz$$
$$\dot{z} = xy - bz$$

  where $\sigma, r, b > 0$ are adjustable parameters. The parameterisation of $\sigma = 10$, $r = 28$ and $b = \frac{8}{3}$ will be studied in this document unless otherwise stated.
- Logistic Map: $f(x) = xr(1-x)$ for $0 \leq r \leq 4$. The Logistic map exhibits periods of chaotic behavior for $r > r_\infty (= 3.5699\ldots)$. The value of $r = 0.9$ will be used throughout this document unless otherwise stated.

CHAPTER 1

Wavelets

1.1. Introduction

Wavelets are a class of functions used to localise a given function in both position and scaling. They are used in applications such as signal processing and time series analysis. Wavelets form the basis of the *wavelet transform* which "cuts up data of functions or operators into different frequency components, and then studies each component with a resolution matched to its scale" (Dr I. Daubechies [**3**]). In the context of signal processing, the wavelet transform depends upon two variables: scale (or frequency) and time.

There are two main types of wavelet transforms: continuous (CWT) and discrete (DWT). The first is designed to work with functions defined over the whole real axis. The second deals with functions that are defined over a range of integers (usually $t = 0, 1, \ldots, N-1$, where N denotes the number of values in the time series).

1.2. What is a Wavelet?

A *wavelet* is a 'small wave' function, usually denoted $\psi(\cdot)$. A small wave grows and decays in a finite time period, as opposed to a 'large wave', such as the sine wave, which grows and decays repeatedly over an infinite time period.

For a function $\psi(\cdot)$, defined over the real axis $(-\infty, \infty)$, to be classed as a wavelet it must satisfy the following three properties:

(1) The integral of $\psi(\cdot)$ is zero:

$$\int_{-\infty}^{\infty} \psi(u)du = 0 \tag{1.2.1}$$

(2) The integral of the square of $\psi(\cdot)$ is unity:

$$\int_{-\infty}^{\infty} \psi^2(u)du = 1 \tag{1.2.2}$$

(3) Admissibility Condition:

$$C_\psi \equiv \int_0^{\infty} \frac{|\Psi(f)|^2}{f} df \qquad \text{satisfies} \qquad 0 < C_\psi < \infty \tag{1.2.3}$$

Equation (1.2.1) tells us that any excursions the wavelet function ψ makes above zero, must be cancelled out by excursions below zero. Clearly the line $\psi(u) = 0$ will satisfy this, but equation (1.2.2) tells us that ψ must make some finite excursions away from zero.

If the admissibility condition is also satisfied then the signal to be analysed can be reconstructed from its continuous wavelet transform.

One of the oldest wavelet functions is the Haar wavelet (see Figure 1.2.1), named after A. Haar who developed it in 1910:

$$\psi^{(H)}(u) \equiv \begin{cases} +1 & \text{if } 0 \le u < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \le u < 1 \\ 0 & \text{else} \end{cases} \tag{1.2.4}$$

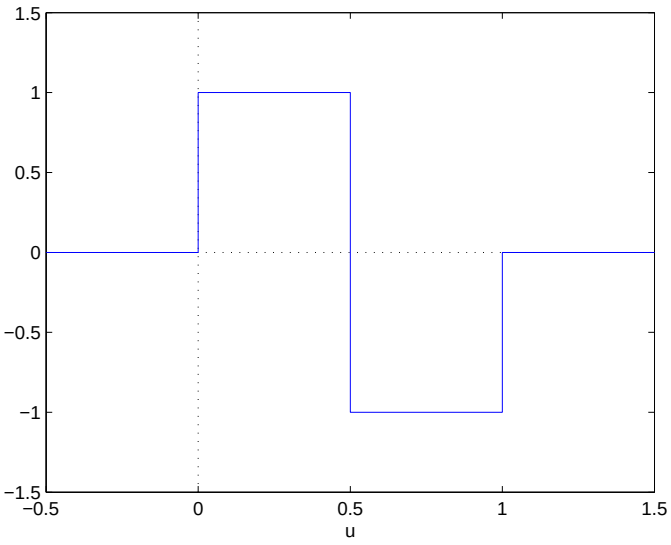

FIGURE 1.2.1. The Haar Wavelet Function $\psi^{(H)}$

1.3. Wavelet Analysis

We have defined what a wavelet function is, but we need to know how it can be used. First of all we should look at Fourier analysis. Fourier analysis can tell us the composition of a given function in terms of sinusoidal waves of different frequencies and amplitudes. This is perfectly ample when the function we are looking at is stationary. However, when the frequency changes over time or there are singularities, as is often the case, Fourier analysis will break down. It will give us the average of the changing frequencies over the whole function, which is not of much use. Wavelet analysis can tell us how a given function changes from one time period to the next. It does this by matching a wavelet function, of varying scales and positions, to that function. Wavelet analysis is also more flexible, in that we can chose a specific wavelet to match the type of function we are analysing. Whereas in classical Fourier analysis the basis is fixed to be sine or cosine waves.

The function $\psi(\cdot)$ is generally referred to as the *mother wavelet*. A doubly-indexed family of wavelets can be created by translating and dilating this mother wavelet:

$$\psi_{\lambda,t}(u) = \frac{1}{\sqrt{\lambda}}\psi\left(\frac{u-t}{\lambda}\right) \tag{1.3.1}$$

where $\lambda > 0$ and t is finite[1]. The normalisation on the right-hand side of equation (1.3.1) was chosen so that $||\psi_{\lambda,t}|| = ||\psi||$ for all λ, t.

We can represent certain functions as a linear combination in the discrete case (see equation (1.4.9)), or as an integral in the continuous case (see equation (1.3.3)), of the chosen wavelet family without any loss of information about those functions.

[1]As we shall see, λ and t can be either discretely or continuously sampled.

1.3.1. Continuous Wavelet Transform. The continuous wavelet transform (CWT) is used to transform a function or signal $x(\cdot)$ that is defined over continuous time. Hence, the parameters λ and t used for creating the wavelet family both vary continuously. The idea of the transform is, for a given dilation λ and a translation t of the mother wavelet ψ, to calculate the amplitude coefficient which makes $\psi_{\lambda,t}$ best fit the signal $x(\cdot)$. That is, to integrate the product of the signal with the wavelet function:

$$\langle x, \psi_{\lambda,t} \rangle = \int_{-\infty}^{\infty} \psi_{\lambda,t}(u) x(u) du. \tag{1.3.2}$$

By varying λ, we can build up a picture of how the wavelet function fits the signal from one dilation to the next. By varying t, we can see how the nature of the signal changes over time. The collection of coefficients $\{< x, \psi_{\lambda,t} > \mid \lambda > 0, -\infty < t < \infty\}$ is called the CWT of $x(\cdot)$.

A fundamental fact about the CWT is that it preserves all the information from $x(\cdot)$, the original signal. If the wavelet function $\psi(\cdot)$ satisfies the admissibility condition (see Equation (1.2.3)) and the signal $x(\cdot)$ satisfies:

$$\int_{-\infty}^{\infty} x^2(t) dt < \infty$$

then we can recover $x(\cdot)$ from its CWT using the following inverse transform:

$$x(t) = \frac{1}{C_\psi} \int_0^{\infty} \left[\int_{-\infty}^{\infty} \langle x, \psi_{\lambda,u} \rangle \psi_{\lambda,u}(t) du \right] \frac{d\lambda}{\lambda^2} \tag{1.3.3}$$

where C_ψ is defined as in Equation (1.2.3).

So, the signal $x(\cdot)$ and its CWT are two representations of the same entity. The CWT presents $x(\cdot)$ in a new manner, which allows us to gain further, otherwise hidden, insight into the signal. The CWT theory is developed further in Chapter 2 of [**3**].

1.3.2. Discrete Wavelet Transform. The analysis of a signal using the CWT yields a wealth of information. The signal is analysed over infinitely many dilations and translations of the mother wavelet. Clearly there will be a lot of redundancy in the CWT. We can in fact retain the key features of the transform by only considering subsamples of the CWT. This leads us to the discrete wavelet transform (DWT).

The DWT operates on a discretely sampled function or time series $x(\cdot)$, usually defining time $t = 0, 1, \ldots, N-1$ to be finite. It analyses the time series for discrete dilations and translations of the mother wavelet $\psi(\cdot)$. Generally, 'dyadic' scales are used for the dilation values λ (i.e. λ is of the form $2^{j-1}, j = 1, 2, 3, \ldots$). The translation values t are then sampled at 2^j intervals, when analysing within a dilation of 2^{j-1}.

Figure 1.3.1 shows the DWT of a signal using the Haar wavelet (the matlab code used to perform the DWT is given in Appendix A.1). The diagram shows the transform for dilations j of up to 3. The signal used was the sine wave with 10% added noise, sampled over 1024 points. The first level of the transform, removes the high frequency noise. Subsequent transforms remove lower and lower frequency features from the signal

and we are left with an approximation of the original signal which is a lot smoother. This approximation shows any underlying trends and the overall shape of the signal.

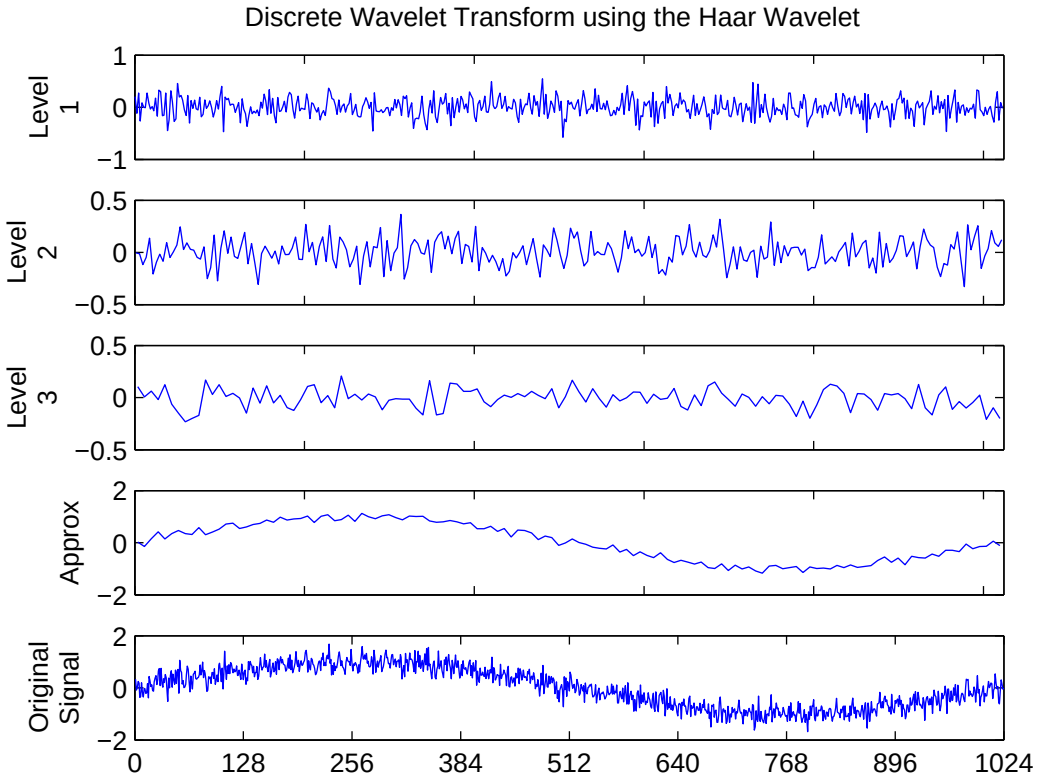

FIGURE 1.3.1. DWT using the Haar Wavelet

The DWT of the signal contains the same number, 1024, of values called 'DWT coefficients'. In Figure 1.3.1 they were organised into four plots. The first three contained the 'wavelet coefficients' at levels 1, 2 and 3. The scale of the wavelet used to analyse the signal at each stage was 2^{j-1}. There are $N_j = \dfrac{N}{2^j}$ wavelet coefficients at each level, with associated times $t = (2n+1)2^{j-1} - \frac{1}{2}, n = 0, 1, \ldots, N_j - 1$. The wavelet coefficients account for 896 of the 1024 DWT coefficients. The remaining 128 coefficients are called 'scaling coefficients'. This is a smoothed version of the original signal after undergoing the preceding wavelet transforms.

As with the CWT, the original signal can be recovered fully from its DWT. So, while sub-sampling at just the dyadic scales seems to be a great reduction in analysis, there is in fact not loss of data.

The DWT theory is developed further in Section 4 of [**19**] and Chapter 3 of [**3**].

1.3.3. Multiresolution Analysis. Multiresolution Analysis (MRA) is at the heart of wavelet theory. It shows how orthonormal wavelet bases can be used as a tool to describe mathematically the "increment of information" needed to go from a coarse approximation to a higher resolution of approximation.

DEFINITION 1. ***Multiresolution Analysis*** [**10**]
A multiresolution analysis is a sequence V_j of subspaces of $L^2(\mathbb{R})$ such that:

(1) $\{0\} \subset \cdots \subset V_0 \subset V_1 \subset \cdots \subset L^2(\mathbb{R})$.
(2) $\bigcap_{j\in\mathbb{Z}} V_j = \{0\}$.
(3) $\text{span} \bigcup_{j\in\mathbb{Z}} V_j = L^2(\mathbb{R})$.
(4) $x(t) \in V_j \iff x(2^{-j}t) \in V_0$.
(5) $x(t) \in V_0 \iff x(t-k) \in V_0$ *for all* $k \in Z$.
(6) $\{\phi(t-k)\}_{k\in\mathbb{Z}}$ *is an orthonormal basis for* V_0*, where* $\phi \in V_0$ *is called a scaling function.*

It follows, from Axiom (4), that $\{\phi(2^j t - k)\}_{k\in\mathbb{Z}}$ is an orthogonal basis for the space V_j. Hence, $\{\phi_{j,k}\}_{j\in Z}$ forms an orthonormal basis for V_j, where:

$$\phi_{j,k} = 2^{j/2}\phi(2^j t - k) \tag{1.3.4}$$

For a given MRA $\{V_j\}$ in $L^2(\mathbb{R})$ with scaling function $\phi(\cdot)$, an associated wavelet function $\psi(\cdot)$ is obtained as follows:

- For every $j \in \mathbb{Z}$, define W_j to be the orthogonal complement of V_j in V_{j+1}:
$$V_{j+1} = V_j \oplus W_j \tag{1.3.5}$$
with W_j satisfying $W_j \perp W_{j'}$ if $j \neq j'$.
- If follows that, for some $j_0 < j$ where $V_{j_0} \subset V_j$, we have:
$$\begin{aligned} V_j &= V_{j-1} \oplus W_{j-1} \\ &= V_{j-2} \oplus W_{j-2} \oplus W_{j-1} \\ &\vdots \\ &= V_{j_0} \oplus \bigoplus_{k=j_0}^{j-1} W_k \end{aligned} \tag{1.3.6}$$
where each W_j satisfies $W_j \subset V_{j'}$ for $j < j'$ and $W_j \perp W_{j'}$ for $j \neq j'$.
- It follows from Axioms (2) and (3) that the MRA $\{W_j\}$ forms an orthogonal basis for $L^2(\mathbb{R})$:
$$L^2(\mathbb{R}) = \bigoplus_{j\in\mathbb{Z}} W_j \tag{1.3.7}$$
- A function $\psi \in W_0$ such that $\{\psi(t-k)\}_{k\in\mathbb{Z}}$ is an orthonormal basis in W_0 is called a *wavelet function.*

It follows that $\{\psi_{j,k}\}_{k\in\mathbb{Z}}$ is an orthonormal basis in W_j, where:

$$\psi_{j,k} = 2^{j/2}\psi(2^j t - k) \tag{1.3.8}$$

For a more in depth look at multiresolution analysis see Chapter 5 of [**3**].

1.4. Discrete Wavelet Transform Algorithm

Since $\phi \in V_0 \subset V_1$, and $\{\phi_{1,k}\}_{k\in\mathbb{Z}}$ is an orthonormal basis for V_1, we have:

$$\phi = \sum_k h_k \phi_{1,k} \tag{1.4.1}$$

with $h_k = \langle \phi, \phi_{1,k}\rangle$ and $\sum_{k\in\mathbb{Z}} |h_k|^2 = 1$.

We can rewrite equation (1.4.1) as follows:

$$\phi(t) = \sqrt{2}\sum_k h_k \phi(2t - k) \tag{1.4.2}$$

We would like to construct a wavelet function $\psi(\cdot) \in W_0$, such that $\{\psi_{0,k}\}_{k\in\mathbb{Z}}$ is an orthonormal basis for W_0. We know that $\psi \in W_0 \iff \psi \in V_1$ and $\psi \perp V_0$, so we can write:

$$\psi = \sum_k g_k \phi_{1,k} \tag{1.4.3}$$

where $g_k = \langle \psi, \phi_{1,k}\rangle = (-1)^k h_{m-k-1}$ and m indicates the support of the wavelet.

Consequently,

$$\begin{aligned}
\psi_{-j,k}(t) &= 2^{-j/2}\psi(2^{-j}t - k) \\
&= 2^{-j/2}\sum_n g_n 2^{1/2}\phi(2^{-j+1}t - (2k+n)) \\
&= \sum_n g_n \phi_{-j+1,2k+n}(t) \\
&= \sum_n g_{n-2k}\phi_{-j+1,n}(t)
\end{aligned} \tag{1.4.4}$$

It follows that,

$$\langle x, \psi_{-j,k}\rangle = \sum_n g_{n-2k}\langle x, \phi_{-j+1,n}\rangle \tag{1.4.5}$$

Similarly,

$$\begin{aligned}
\phi_{-j,k}(t) &= 2^{-j/2}\phi(2^{-j}t - k) \\
&= \sum_n h_{n-2k}\phi_{-j+1,n}(t)
\end{aligned} \tag{1.4.6}$$

and hence,

$$\langle x, \phi_{-j,k}\rangle = \sum_n h_{n-2k}\langle x, \phi_{-j+1,n}\rangle \tag{1.4.7}$$

The MRA leads naturally to a hierarchical and fast method for computing the wavelet coefficients of a given function. If we let V_0 be our starting space. We have $x(t) \in V_0$ and $\{\phi(t-k)\}_{k\in\mathbb{Z}}$ is an orthonormal basis for V_0. So we can write:

$$x(t) = \sum_k a_k \phi(t-k) \quad \text{for some coefficients } a_k \tag{1.4.8}$$

The a_k are in fact the coefficients $\langle x, \phi_{0,k}\rangle$. If we know or have calculated these then we can compute the $\langle x, \psi_{-1,k}\rangle$ coefficients by (1.4.5) and the $\langle x, \phi_{-1,k}\rangle$ coefficients by (1.4.7). We can then apply (1.4.5) and (1.4.7) to $\langle x, \phi_{-1,k}\rangle$ to get $\langle x, \psi_{-2,k}\rangle$ and $\langle x, \phi_{-2,k}\rangle$ respectively. This process can be repeated until the desired resolution level is reached. i.e up to the Kth level coefficients we have:

$$\begin{aligned}
x(t) &= \sum_{k=0}^{\frac{N}{2}-1} \langle x, \phi_{-1,k}\rangle \phi_{-1,k} + \sum_{k=0}^{\frac{N}{2}-1} \langle x, \psi_{-1,k}\rangle \psi_{-1,k} \\
&= \sum_{k=0}^{\frac{N}{4}-1} \langle x, \phi_{-2,k}\rangle \phi_{-2,k} \\
&+ \sum_{k=0}^{\frac{N}{4}-1} \langle x, \psi_{-2,k}\rangle \psi_{-2,k} + \sum_{k=0}^{\frac{N}{2}-1} \langle x, \psi_{-1,k}\rangle \psi_{-1,k} \\
&\vdots \\
&= \sum_{k=0}^{\frac{N}{2^K}-1} \langle x, \phi_{-K,k}\rangle \phi_{-K,k} + \sum_{j=1}^{K} \sum_{k=0}^{\frac{N}{2^j}-1} \langle x, \psi_{-j,k}\rangle \psi_{-j,k}
\end{aligned} \tag{1.4.9}$$

1.4.1. Example Using the Haar Wavelet. The Haar scaling function $\phi(\cdot)$, see Figure 1.4.1, is defined as:

$$\phi^{(H)}(u) = \begin{cases} 1 & \text{if } 0 \le u < 1 \\ 0 & \text{else} \end{cases} \tag{1.4.10}$$

We calculate the h_k from the inner product:

$$\begin{aligned}
h_k = \langle \phi, \phi_{1,k}\rangle &= \sqrt{2} \int \phi(t)\phi(2t-k)dt \\
&= \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0, 1 \\ 0 & else \end{cases}
\end{aligned}$$

Therefore,

$$\phi_{0,0} = \frac{1}{\sqrt{2}}\phi_{1,0} + \frac{1}{\sqrt{2}}\phi_{1,1} \tag{1.4.11}$$

which is in agreement with the definition of Equation (1.4.1).

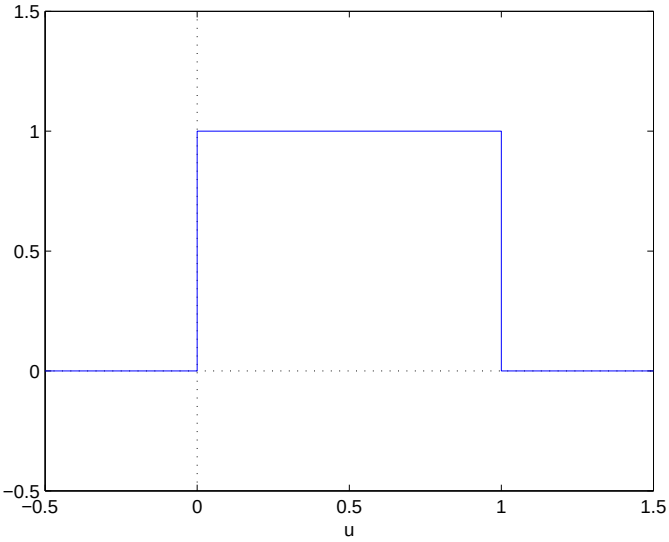

FIGURE 1.4.1. The Haar Scaling Function $\phi^{(H)}$

The g_k can be calculated using the relation $g_k = (-1)^k h_{m-k-1}$. In this case $m = 2$, so:

$$g_0 = (-1)^0 h_{2-0-1} = h_1$$
$$g_1 = (-1)^1 h_{2-1-1} = -h_0$$

Therefore,

$$\psi_{0,0} = \frac{1}{\sqrt{2}}\phi_{1,0} - \frac{1}{\sqrt{2}}\phi_{1,1} \tag{1.4.12}$$

If we have a discretely sampled signal $x(\cdot)$, as shown in Figure 1.4.2, we have shown it can be expressed as the following sum:

$$\begin{aligned} x(t) =& \sum_{k=0}^{N-1} \langle x, \phi_{0,k} \rangle \phi(t-k) \\ =& 5\phi(t) + 8\phi(t-1) + 3\phi(t-2) + 5\phi(t-3) \\ &+ 4\phi(t-4) + 3\phi(t-5) + 7\phi(t-6) + 6\phi(t-7) \end{aligned}$$

Now, the same signal can also be written as a linear combination of translations of the scaling and wavelet functions, each dilated by a factor of 2 (see Figure 1.4.3 for a pictorial representation).

$$x(t) = \sum_{k=0}^{\frac{N}{2}-1} \langle x, \psi_{-1,k} \rangle \psi_{-1,k} + \sum_{k=0}^{\frac{N}{2}-1} \langle x, \phi_{-1,k} \rangle \phi_{-1,k} \tag{1.4.13}$$

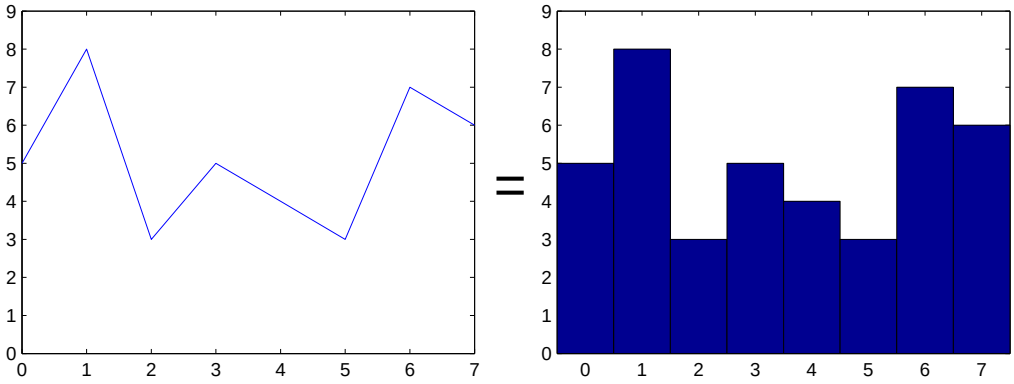

FIGURE 1.4.2. Discretely Sampled Signal for $N = 8$

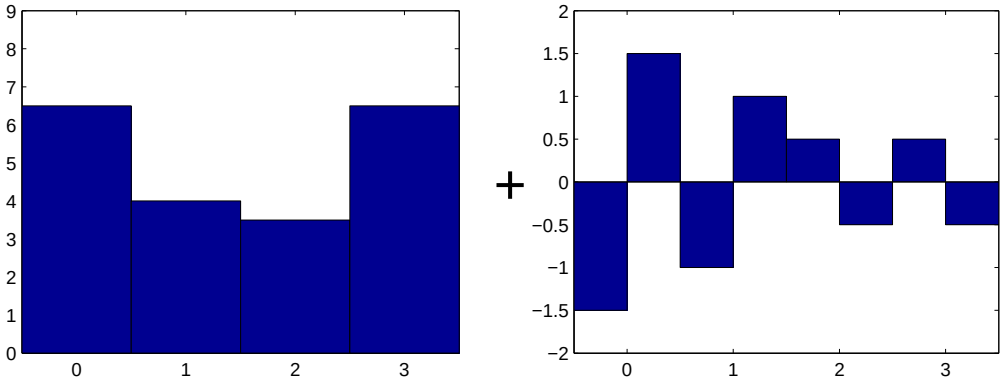

FIGURE 1.4.3. Scaling and Wavelet Composition of Original Signal

The scaling coefficients for (1.4.13) can be found using equation (1.4.7):

$$\langle x, \phi_{-1,0} \rangle = \sum_n h_n \langle x, \phi_{0,n} \rangle = h_0 \langle x, \phi_{0,0} \rangle + h_1 \langle x, \phi_{0,1} \rangle = \frac{1}{\sqrt{2}}.5 + \frac{1}{\sqrt{2}}.8 = \frac{13}{\sqrt{2}}$$

$$\langle x, \phi_{-1,1} \rangle = \sum_n h_{n-2} \langle x, \phi_{0,n} \rangle = h_0 \langle x, \phi_{0,2} \rangle + h_1 \langle x, \phi_{0,3} \rangle = \frac{1}{\sqrt{2}}.3 + \frac{1}{\sqrt{2}}.5 = \frac{8}{\sqrt{2}}$$

$$\langle x, \phi_{-1,2} \rangle = \sum_n h_{n-4} \langle x, \phi_{0,n} \rangle = h_0 \langle x, \phi_{0,4} \rangle + h_1 \langle x, \phi_{0,5} \rangle = \frac{1}{\sqrt{2}}.4 + \frac{1}{\sqrt{2}}.3 = \frac{7}{\sqrt{2}}$$

$$\langle x, \phi_{-1,3} \rangle = \sum_n h_{n-6} \langle x, \phi_{0,n} \rangle = h_0 \langle x, \phi_{0,6} \rangle + h_1 \langle x, \phi_{0,7} \rangle = \frac{1}{\sqrt{2}}.7 + \frac{1}{\sqrt{2}}.6 = \frac{13}{\sqrt{2}}$$

The wavelet coefficients for (1.4.13) can be found using equation (1.4.5):

$$\langle x, \psi_{-1,0}\rangle = \sum_n g_n \langle x, \phi_{0,n}\rangle = g_0\langle x, \phi_{0,0}\rangle + g_1\langle x, \phi_{0,1}\rangle = \frac{1}{\sqrt{2}}.5 - \frac{1}{\sqrt{2}}.8 = -\frac{3}{\sqrt{2}}$$

$$\langle x, \psi_{-1,1}\rangle = \sum_n g_{n-2} \langle x, \phi_{0,n}\rangle = g_0\langle x, \phi_{0,2}\rangle + g_1\langle x, \phi_{0,3}\rangle = \frac{1}{\sqrt{2}}.3 - \frac{1}{\sqrt{2}}.5 = -\frac{2}{\sqrt{2}}$$

$$\langle x, \psi_{-1,2}\rangle = \sum_n g_{n-4} \langle x, \phi_{0,n}\rangle = g_0\langle x, \phi_{0,4}\rangle + g_1\langle x, \phi_{0,5}\rangle = \frac{1}{\sqrt{2}}.4 - \frac{1}{\sqrt{2}}.3 = \frac{1}{\sqrt{2}}$$

$$\langle x, \psi_{-1,3}\rangle = \sum_n g_{n-6} \langle x, \phi_{0,n}\rangle = g_0\langle x, \phi_{0,6}\rangle + g_1\langle x, \phi_{0,7}\rangle = \frac{1}{\sqrt{2}}.7 - \frac{1}{\sqrt{2}}.6 = \frac{1}{\sqrt{2}}$$

Therefore, after the first DWT:

$$\begin{aligned} x(t) = &\frac{13}{2}\phi(\frac{t}{2}) + \frac{8}{2}\phi(\frac{t}{2}-1) + \frac{7}{2}\phi(\frac{t}{2}-2) + \frac{13}{2}\phi(\frac{t}{2}-3) \\ &- \frac{3}{2}\psi(\frac{t}{2}) - \frac{2}{2}\psi(\frac{t}{2}-1) + \frac{1}{2}\psi(\frac{t}{2}-2) + \frac{1}{2}\psi(\frac{t}{2}-3) \end{aligned}$$

1.5. Inverse Discrete Wavelet Transform

If we know the scaling and wavelet coefficients ($\langle x, \phi_{-K,k}\rangle$ and $\langle x, \psi_{-j,k}\rangle$ from Equation (1.4.9)) from the DWT, we can reconstruct the original signal $x(t)$. This is known as the Inverse Discrete Wavelet Transform (IDWT). The signal is reconstructed iteratively: first the coefficients of level K are used to calculate the level $K-1$ scaling coefficients (we already know the level $K-1$ wavelet coefficients). The $K-1$ coefficients are then used to calculate the $K-2$ scaling coefficients, and so on until we have the level 0 scaling coefficients. i.e. the $\langle x, \phi_{0,k}\rangle$.

The IDWT for level $K \to (K-1)$ is performed by solving equations (1.4.5) and (1.4.7) for each $\langle x, \phi_{-(K-1),n}\rangle$. Since most of the h_k and g_k are usually equal to 0, this usually leads to a simple relation.

For the Haar wavelet, the IDWT is defined as follows:

$$\langle x, \phi_{-(K-1),2n}\rangle = \frac{1}{\sqrt{2}}\langle x, \phi_{-K,n}\rangle + \frac{1}{\sqrt{2}}\langle x, \psi_{-K,n}\rangle \quad \text{for } n = 0, 1, \ldots, N_K - 1 \tag{1.5.1}$$

$$\langle x, \phi_{-(K-1),2n+1}\rangle = \frac{1}{\sqrt{2}}\langle x, \phi_{-K,n}\rangle - \frac{1}{\sqrt{2}}\langle x, \psi_{-K,n}\rangle \quad \text{for } n = 0, 1, \ldots, N_K - 1 \tag{1.5.2}$$

The full derivation of the Haar IDWT is given in Section 3.4 of [**9**].

Matlab code that will perform the IDWT is given in Appendix A.2. It inverse transforms the matrix of wavelet coefficients and the vector of scaling coefficients that resulted from the DWT.

One of the uses of the wavelet transform is to remove unwanted elements, like noise, from a signal. We can do this by transforming the signal, setting the required level of

wavelet coefficients to zero, and inverse transforming. For example, we can remove high-frequency noise by zeroing the 'level 1' wavelet coefficients, as shown in Figure 1.5.1.

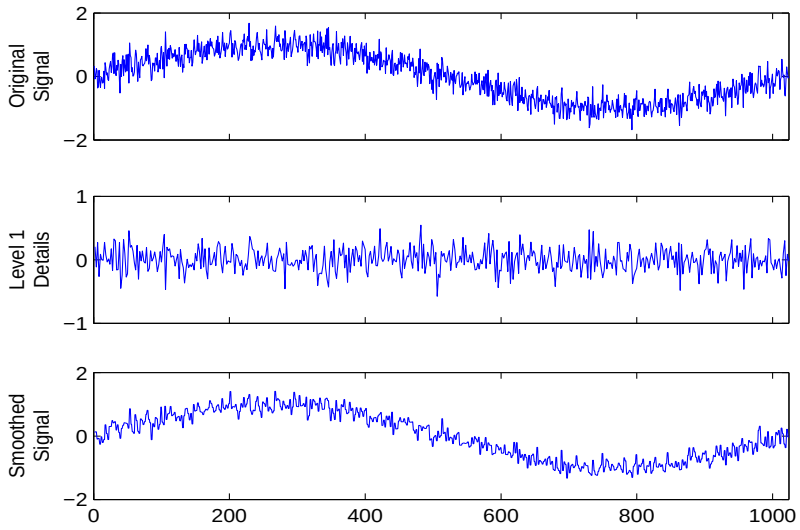

FIGURE 1.5.1. High-Frequency Noise Removal of Signal

1.6. Daubechies Discrete Wavelets

We have seen in Section 1.2 the three properties that a function must satisfy to be classed as a wavelet. For a wavelet, and its scaling function, to be useful in the DWT, there are more conditions that must be satisfied:

$$\sum_k h_k = \sqrt{2} \tag{1.6.1}$$

$$\sum_k (-1)^k k^m h_k = 0, \quad \text{for } m = 0, 1, \ldots, \frac{N}{2} - 1 \tag{1.6.2}$$

$$\sum_k h_k h_{k+2m} = \begin{cases} 0 & \text{for } m = 1, 2, \ldots, \frac{N}{2} - 1 \\ 1 & \text{for } m = 0 \end{cases} \tag{1.6.3}$$

Ingrid Daubechies discovered a class of wavelets, which are characterised by orthonormal basis functions. That is, the mother wavelet is orthonormal to each function obtained by shifting it by multiples of 2^j and dilating it by a factor of 2^j (where $j \in \mathbb{Z}$).

The Haar wavelet was a two-term member of the class of discrete Daubechies wavelets. We can easily check that the above conditions are satisfied for the Haar wavelet, remembering that $h_0 = h_1 = \dfrac{1}{\sqrt{2}}$.

The Daubechies D(4) wavelet is a four-term member of the same class. The four scaling function coefficients, which solve the above simultaneous equations for $N = 4$, are specified

as follows:

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}} \qquad h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}$$
$$h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}} \qquad h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \tag{1.6.4}$$

The scaling function for the D(4) wavelet can be built up recursively from these coefficients (see Figure 1.6.1 [2]). The wavelet function can be built from the coefficients g_k, which are found using the relation $g_k = (-1)^k h_{4-k-1}$.

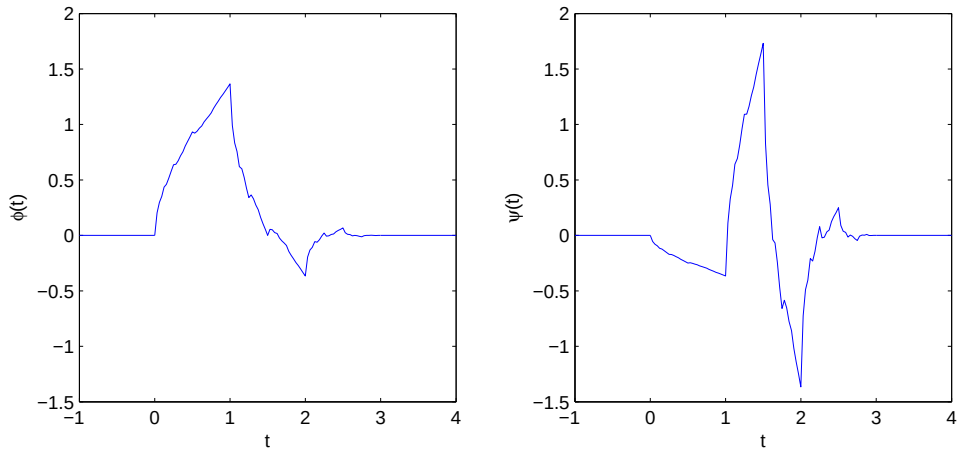

FIGURE 1.6.1. Daubechies D(4) Scaling and Wavelet Functions

The Daubechies orthonormal wavelets of up to 20 coefficients (even numbered only) are commonly used in wavelet analysis. They become smoother and more oscillatory as the number of coefficients increases (see Figure 1.6.2 for the D(20) wavelet).

A specific Daubechies wavelet will be chosen for each different wavelet analysis task, depending upon the nature of the signal being analysed. If the signal is not well represented by one Daubechies wavelet, it may still be efficiently represented by another. The selection of the correct wavelet for the task is important for efficiently achieving the desired results. In general, wavelets with short widths (such as the Haar or D(4) wavelets) pick out fine levels of detail, but can introduce undesirable effects into the resulting wavelet analysis. The higher level details can appear blocky and unrealistic. Wavelets with larger widths can give a better representation of the general characteristics of the signal. They do however result in increased computation and result in decreased localisation of features (such as discontinuities). A reasonable choice is to use the smallest wavelet that gives satisfactory results.

For further details and the coefficient values for the other wavelets in the Daubechies orthonormal wavelet family see Chapter 6.4 in [**3**] and Section 4.4.5 in [**8**].

[2]Created using matlab code from the SEMD website [**17**]

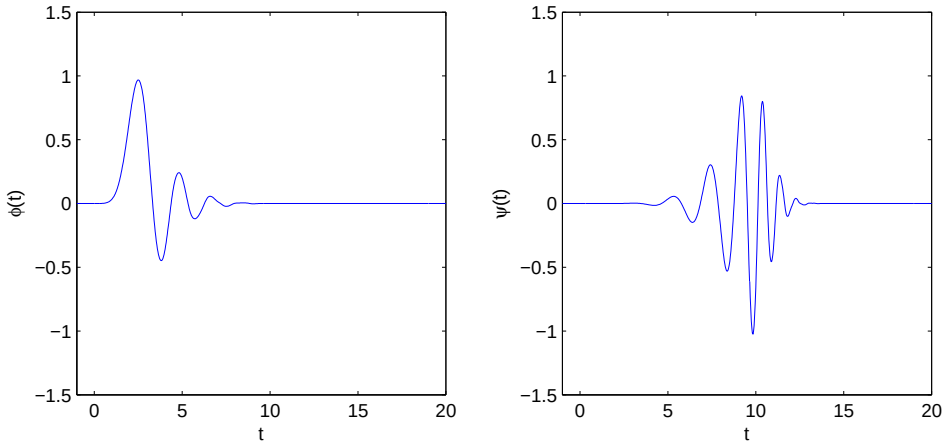

FIGURE 1.6.2. Daubechies D(20) Scaling and Wavelet Functions

1.7. Other Wavelet Definitions

Some common wavelets that will be used later in this document are described below.

1.7.1. Gaussian Derivative. The Gaussian function is local in both time and frequency domains and is a $C^\infty(\mathbb{R})$ function. Therefore, any derivative of the Gaussian function can be used as the basis for a wavelet transform.

The Gaussian first derivative wavelet, shown in Figure 1.7.1, is defined as:

$$\psi(u) = -u \exp(-\frac{u^2}{2}) \tag{1.7.1}$$

The Mexican hat function is the second derivative of the Gaussian function. If we normalise it so that it satisfies the second wavelet property (1.2.2), then we obtain

$$\psi(u) = \frac{2}{\sqrt{3}}\pi^{-1/4}(1-u^2)\exp(-\frac{u^2}{2}). \tag{1.7.2}$$

The Gaussian derivative wavelets do not have associated scaling functions.

1.7.2. Battle-Lémarie Wavelets. The Battle-Lémarie family of wavelets are associated with the multiresolution analysis of spline function spaces. The scaling functions are created by taking a B-spline with knots at the integers. An example scaling function is formed from quadratic B-splines,

$$\phi(u) = \begin{cases} \frac{1}{2}(u+1)^2 & \text{for } -1 \le u < 0 \\ \frac{3}{4} - (u-\frac{1}{2})^2 & \text{for } 0 \le u < 1 \\ \frac{1}{2}(u-2)^2 & \text{for } 1 \le u \le 2 \\ 0 & \text{else} \end{cases} \tag{1.7.3}$$

and is plotted in Figure 1.7.3.

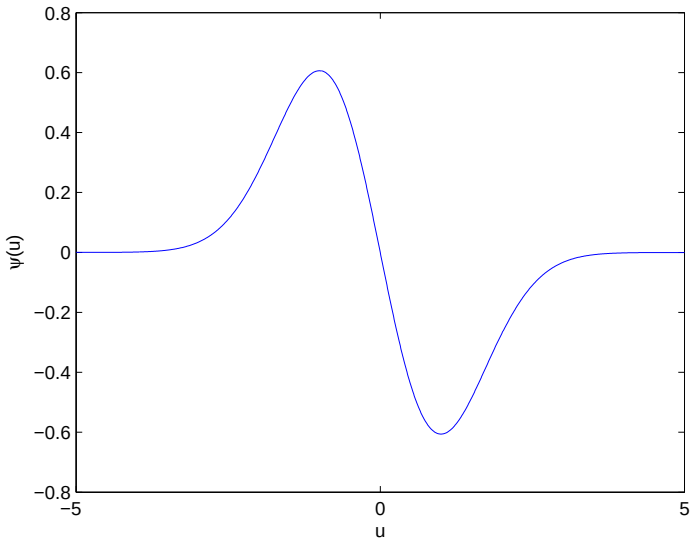

FIGURE 1.7.1. The Gaussian Derivative Wavelet Function

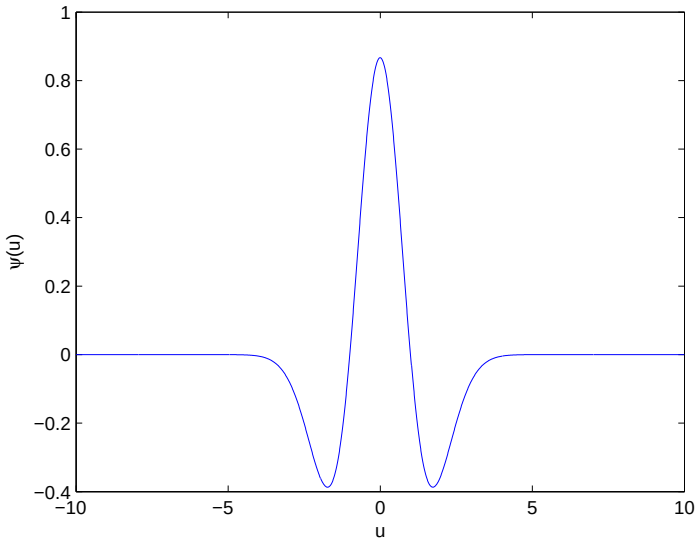

FIGURE 1.7.2. The Mexican Hat Wavelet

This scaling function satisfies Equation (1.4.1) as follows:

$$\phi(u) = \frac{1}{4}\phi(2u+1) + \frac{3}{4}\phi(2u) + \frac{3}{4}\phi(2u-1) + \frac{1}{4}\phi(2u-2)$$

For further information see Section 5.4 of [**3**].

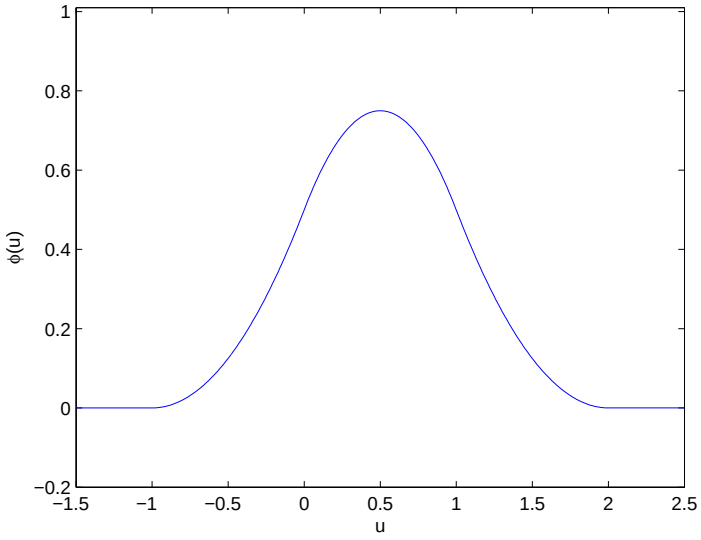

FIGURE 1.7.3. The Battle-Lémarie Scaling Function

1.8. Wavelet-Based Signal Estimation

Signal Estimation is the process of estimating a signal, that is hidden by noise, from an observed time series. To do this using wavelets we modify the wavelet transform, so that when we perform the inverse transform, an estimate of the underlying signal can be realised. The theory given in this chapter is further developed in Section 10 of [**19**].

1.8.1. Signal Representation Methods. Let the vector D represent a deterministic signal consisting of N elements. We would like to be able to efficiently represent this signal in fewer than N terms. There are several ways in which this is potentially possible, depending upon the nature of D, but we are particularly interested in how well the DWT performs.

To quantify how well a particular orthonormal transform, such as the DWT, performs in capturing the key elements of D in a small number of terms we can use the notion of normalised partial energy sequence (NPES).

DEFINITION 2. ***Normalised Partial Energy Sequence (NPES) (see Section 4.10 in*** [**19**]***)***

For a sequence of real or complex valued variables $\{U_t : t = 0, \ldots, M-1\}$*, a NPES is formed as follows:*

(1) *Form the squared magnitudes* $|U_t|^2$*, and order them such that*

$$|U_{(0)}|^2 \geq |U_{(1)}|^2 \geq \cdots \geq |U_{(M-1)}|^2.$$

(2) *The NPES is defined as*

$$C_n = \frac{\sum_{u=0}^{n} |U_{(u)}|^2}{\sum_{u=0}^{M-1} |U_{(u)}|^2} \quad \textit{for } n = 0, 1, \ldots, M-1. \tag{1.8.1}$$

We can see that the NPES $\{C_n\}$ is a nondecreasing sequence with $0 < C_n \leq 1$ for all n. If a particular orthonormal transform is capable of representing the signal in few coefficients, then we would expect C_n to become close to unity for relatively small n.

If we define $O = \mathcal{O}D$ to be the vector of orthonormal transform coefficients, for the signal D using the $N \times N$ transform matrix $\mathcal{O}$, then we can apply the NPES method to O and obtain the set of C_n values.

For certain types of signals the DWT outperforms other orthonormal transforms such as the orthonormal discrete Fourier transform (ODFT). To see this, let us look at the following three signals (shown in the left-hand column of Figure 1.8.1). Each signal D_j is defined for $t = 0, \ldots, 127$.

(1) $D_{1,t} = \frac{1}{2}\sin(\frac{2\pi t}{16})$

(2) $D_{2,t} = \begin{cases} D_{1,t} & \text{for } t = 64, \ldots, 72 \\ 0 & \text{else} \end{cases}$

(3) $D_{3,t} = \frac{1}{8}D_{1,t} + D_{2,t}$

D_1 is said to be a 'frequency domain' signal, because it can be fully represented in the frequency domain by two non-zero coefficients. D_2 is said to be a 'time domain' signal, because it is represented in the time domain by only nine non-zero coefficients. Signal D_3 is a mixture of the two domains. The right-hand column of Figure 1.8.1 shows the NPESs (see Appendix A.3 for the matlab code used to create the plots) for three different orthonormal transformations of each of the signals D_j: the identity transform (shown by the solid line), the ODFT (dotted line) and the DWT (dashed line). The Daubechies D(4) wavelet was used for the DWT and it was calculated to four transform levels.

We can see the DWT was outperformed by the ODFT and the identity transform for the 'frequency domain' and 'time domain' signals respectively. However, when the signal is a mixture of the two domains, the DWT produces superior results. This suggests that the DWT will perform well for signals containing both time domain (transient events) and frequency domain (broadband and narrowband features) characteristics. This is more representative of naturally occurring deterministic signals.

1.8.2. Signal Estimation via Thresholding. In practice, a deterministic signal will also contain some unwanted noise. Let us look at a time series modelled as $X = D + \epsilon$, where D is the deterministic signal of interest (unknown to the observer) and ϵ is the stochastic noise.

If we define $O = \mathcal{O}X$ to be the N dimensional vector of orthonormal transform coefficients, then:

$$O \equiv \mathcal{O}X = \mathcal{O}D + \mathcal{O}\epsilon \equiv d + e \tag{1.8.2}$$

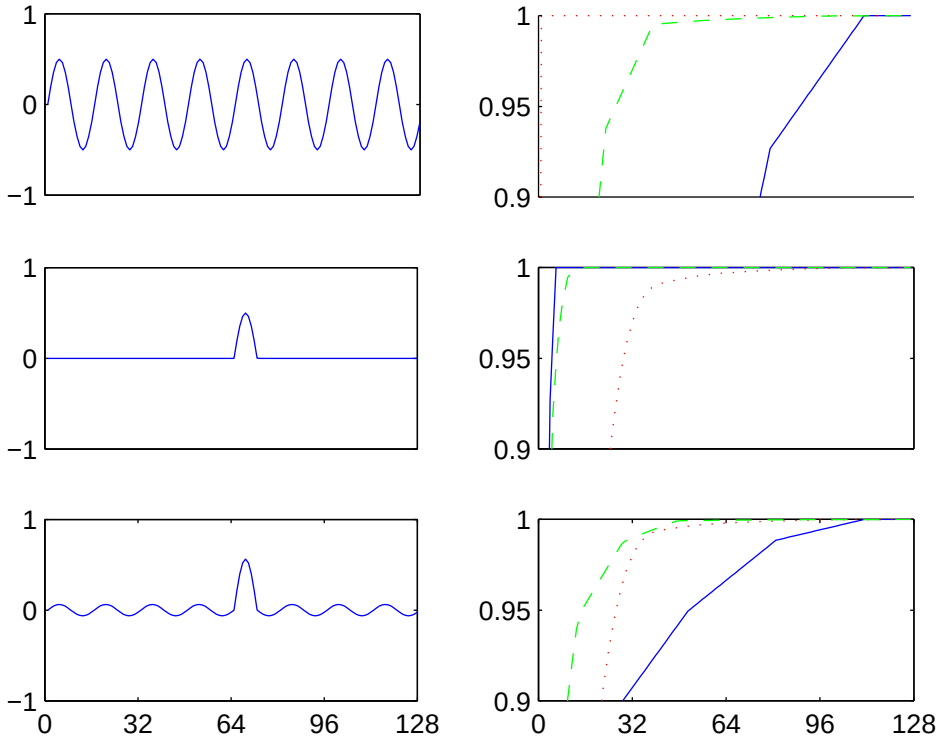

FIGURE 1.8.1. The signals D_j (left-hand column) and their corresponding NPES (right-hand column). The NPES is shown for the signal (solid line), the ODFT of the signal (dotted line) and the DWT using the D(4) wavelet (dashed line).

Hence, $O_l = d_l + e_l$ for $l = 0, \ldots, N-1$. We can define a signal-to-noise ratio for this model as: $||D||^2/E\{||\epsilon||^2\} = ||d||^2/E\{||e||^2\}$. If this ratio is large and the transform $\mathcal{O}$ successfully isolates the signal from the noise, then O should consist of a few large values (relating to the signal) and many small values (relating to the noise).

Let us define M to be the (unknown) number of large coefficients relating to the signal and $\mathcal{I}_M$ to be an $N \times N$ matrix that extracts these coefficients from O. To estimate the signal, we need to find M and hence $\mathcal{I}_M$, then use the relation:

$$\hat{D}_M \equiv \mathcal{O}^T \mathcal{I}_M O. \tag{1.8.3}$$

where, due to the orthonormality of the transform, $\mathcal{O}^T$ is the inverse transform of $\mathcal{O}$.

To find the best estimate $\hat{D}$ we can minimise

$$\gamma_m \equiv ||X - \hat{D}_m||^2 + m\delta^2 \tag{1.8.4}$$

over $m = 0, 1, \ldots, N$ and over all possible $\mathcal{I}_m$ for each m, where $\delta^2 > 0$ is constant. The first term (the *fidelity* condition) in (1.8.4) ensures that D never strays too far from the

observed data, while the second term (the *penalty* condition) penalises the inclusion of a large number of coefficients.

δ in Equation (1.8.4) is the threshold value (see Section 1.8.3), it is used to determine M and the matrix $\mathcal{I}_M$. After choosing δ, M is defined to be the number of transform coefficients O_l satisfying

$$|O_l|^2 > \delta^2 \tag{1.8.5}$$

and hence $\mathcal{I}_M$ is the matrix which will extract these coefficients from O. We can see that this $\mathcal{I}_M$ matrix will lead to the desired signal estimate $\hat{D}_M$ that minimises (1.8.4) by looking at the following:

$$\begin{aligned}\gamma_m = ||X - \hat{D}_m||^2 + m\delta^2 &= ||\mathcal{O}O - \mathcal{O}\mathcal{I}_m O||^2 + m\delta^2 \\ &= ||(I_N - \mathcal{I}_m)O||^2 + m\delta^2 = \sum_{l \notin \mathcal{J}_m} |O_l|^2 + \sum_{l \in \mathcal{J}_m} \delta^2\end{aligned}$$

where $\mathcal{J}$ is the set of m indices $l \in [1, N]$ such that the lth diagonal element of $\mathcal{I}_m$ equals 1. Thus, γ_m is minimised by putting all the ls satisfying (1.8.5) into $\mathcal{J}_m$.

1.8.3. Thresholding Methods. There are many different variations of thresholding of which *soft*, *hard* and *firm* will be discussed here. The basic premise of thresholding is that if a value is below the threshold value $\delta > 0$ then it is set to 0 and set to some non-zero value otherwise.

Thus, a thresholding scheme for estimating D will consist of the following three steps:

(1) Compute the transform coefficients $O \equiv \mathcal{O}X$.
(2) Perform thresholding on the vector O to produce $O^{(t)}$ defined as follows:

$$O_l^{(t)} = \begin{cases} 0 & \text{if } |O_l| \leq \delta \\ \text{some non-zero value} & \text{else} \end{cases}$$

for $l = 0, \ldots, N-1$ (the nonzero values are yet to be determined).
(3) Estimate D using the variation of Equation (1.8.3): $\hat{D}^{(t)} \equiv \mathcal{O}^T O^{(t)}$.

When a coefficient is greater than δ there are several choices of non-zero values to choose.

1.8.3.1. *Hard Thresholding.* In *hard thresholding* the coefficients that exceed δ are left unchanged.

$$O_l^{(ht)} = \begin{cases} 0 & \text{if } |O_l| \leq \delta \\ O_l & \text{else} \end{cases} \tag{1.8.6}$$

This is the simplest method of thresholding but the resulting thresholded values are not now defined upon the whole real line (see solid line in Figure 1.8.2).

1.8.3.2. *Soft Thresholding.* In *soft thresholding* the coefficients exceeding δ are pushed towards zero by the value of δ.

$$O_l^{(st)} = \text{sign}\{O_l\}(|O_l| - \delta)_+ \tag{1.8.7}$$

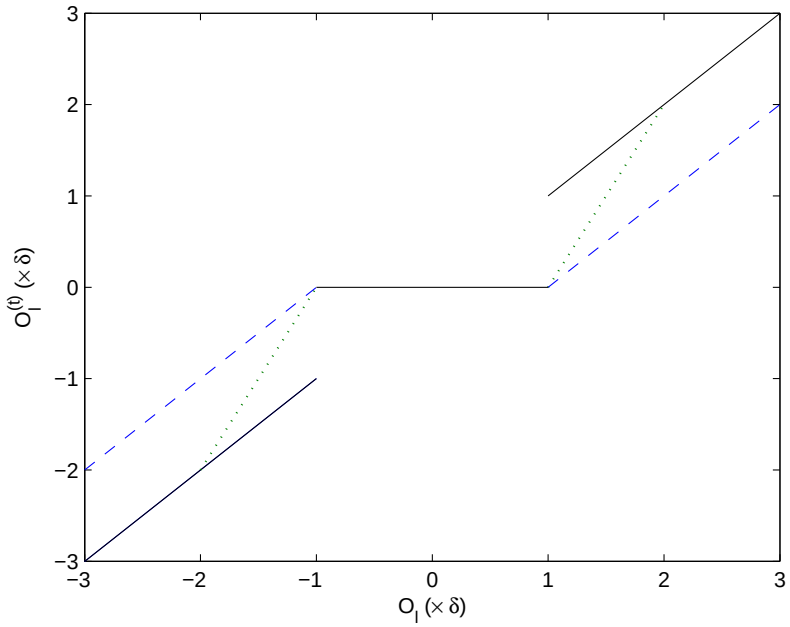

FIGURE 1.8.2. Mapping from O_l to $O_l^{(t)}$ for hard (solid line) soft (dashed line) and firm (dotted line) thresholding

This method of thresholding produces results that are defined over the whole real axis, but there may have been some ill effects from pushing the values towards zero (see dashed line in Figure 1.8.2).

1.8.3.3. *Firm Thresholding. Firm thresholding* is a compromise between hard and soft thresholding. It is defined in terms of two parameters δ and δ'. It acts like hard thresholding for $|O_l| \notin (\delta, \delta']$ and interpolates between soft and hard for $|O_l| \in (\delta, \delta']$.

$$O_l^{(ft)} = \begin{cases} 0 & \text{if } |O_l| \leq \delta \\ \text{sign}\{O_l\} \frac{\delta'(|O_l| - \delta)}{\delta' - \delta} & \text{if } \delta < |O_l| \leq \delta' \\ O_l & \text{if } |O_l| > \delta' \end{cases} \tag{1.8.8}$$

The dotted line in Figure 1.8.2 shows the mapping from O_l to $O_l^{(ft)}$ with $\delta' = 2\delta$.

1.8.3.4. *Universal Threshold Value.* We have so far discussed thresholding a vector using the threshold level of δ, but we don't know how to choose a satisfactory value of δ.

One way to derive δ was given by Donoho and Johnstone [**7**] for the case where the IID noise ϵ is Gaussian distributed. That is, ϵ is a multivariate normal random variable with mean of zero and covariance $\sigma_\epsilon^2 I_N$. The form of δ was given as:

$$\delta^{(u)} \equiv \sqrt{2\sigma_\epsilon^2 \log_e(N)} \tag{1.8.9}$$

which is known as the *universal threshold*.

1.8.4. Signal Estimation using Wavelets. We are particularly interested in signal estimation using the *wavelet* orthonormal transformation. The signal of interest is modelled

as $X = D + \epsilon$, where D is a deterministic signal. By applying the DWT we get

$$W = \mathcal{W}D + \mathcal{W}\epsilon = d + e$$

which is a special case of Equation (1.8.2). As discussed in Section 1.8.2 we can estimate the underlying signal D using thresholding. Additionally if the noise ϵ is IID Gaussian with a common variance of σ_ϵ^2 we can use the 'universal threshold' level (see Section 1.8.3.4).

If we use a level j_0 DWT (as recommended by Donoho and Johnstone [**7**]), then the resulting transform W will consist of $W_1, \ldots, W_{j_0}$ and V_{j_0}. Only the wavelet coefficients $W_1, \ldots, W_{j_0}$ are subject to the thresholding.

The thresholding algorithm is as follows:

(1) Perform a level j_0 DWT to obtain the vector coefficients $W_1, \ldots, W_{j_0}$ and V_{j_0}.
(2) Determine the threshold level δ. If the variance σ_ϵ^2 is not known, calculate the estimate $\sigma_{(mad)}^2$ using

$$\sigma_{(mad)}^2 \equiv \frac{\text{median}\{|W_{1,0}|, |W_{1,1}|, \ldots, |W_{1,\frac{N}{2}-1}|\}}{0.6745}. \tag{1.8.10}$$

(method based upon MAD, see Section , using just the level $j = 1$ wavelet coefficients). $\delta = \delta^{(u)}$ can then be calculated as in Equation (1.8.9), using either the true variance σ_ϵ^2 if known, or its estimate $\sigma_{(mad)}^2$.
(3) For each $W_{j,t}$ coefficient, where $j = 1, \ldots, j_0$ and $t = 0, \ldots, N_j - 1$, apply a thresholding rule (see Section 1.8.3), to obtain the thresholded coefficients $W_{j,t}^{(t)}$, which make up the $W^{(t)}$.
(4) Estimate the signal D via $\hat{D}^{(t)}$, calculated by inverse transforming $W_1^{(t)}, \ldots, W_{j_0}^{(t)}$ and V_{j_0}.

1.8.4.1. *Example.* Let us look at the deterministic signal with 10% added Gaussian noise

$$X = \cos\left(\frac{2\pi t}{128}\right) + \epsilon \quad \text{where} \begin{cases} t = 0, 1, \ldots, 511 \\ \epsilon \sim N(0, 0.225^2) \end{cases} \tag{1.8.11}$$

The matlab function 'threshold.m' (see Appendix A.4) will

(1) Perform the DWT, for a given wavelet transform matrix, to a specified level j_0.
(2) Hard threshold the wavelet coefficients.
(3) Inverse transform the thresholded DWT to produce an estimate of the underlying signal.

Figure 1.8.3 shows the 'observed' signal with the thresholding signal estimate below. We can see that most of the Gaussian noise has been removed and the smoothed signal is very close to the underlying cosine wave.

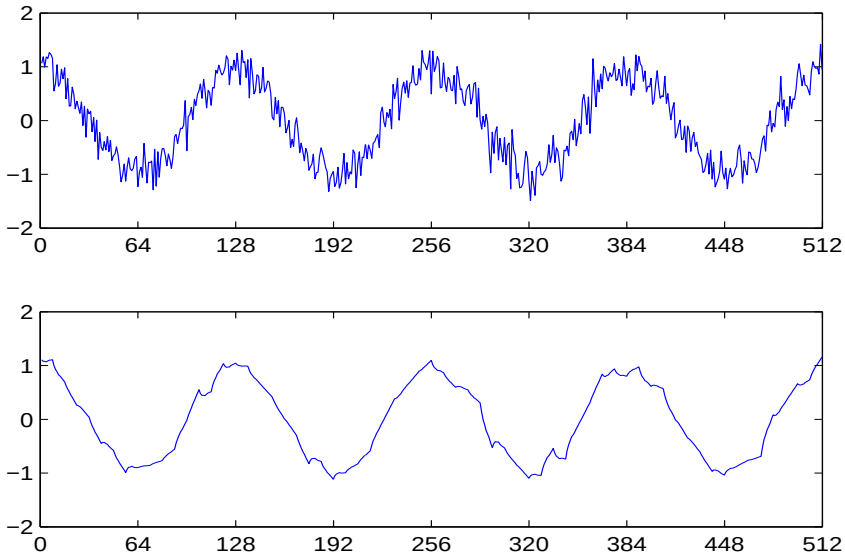

FIGURE 1.8.3. Thresholding signal estimate (lower plot) of the deterministic signal (1.8.11) (upper plot) using Daubechies D(4) wavelet transform to level $j_0 = 3$ and hard thresholding.

CHAPTER 2

Neural Networks

2.1. Introduction - What is a Neural Network?

An *Artificial Neural Network* (ANN) is a highly parallel distributed network of connected processing units called neurons. It is motivated by the human cognitive process: the human brain is a highly complex, nonlinear and parallel computer. The network has a series of external inputs and outputs which take or supply information to the surrounding environment. Inter-neuron connections are called synapses which have associated synaptic weights. These weights are used to store knowledge which is acquired from the environment. Learning is achieved by adjusting these weights in accordance with a learning algorithm. It is also possible for neurons to evolve by modifying their own topology, which is motivated by the fact that neurons in the human brain can die and new synapses can grow.

One of the primary aims of an ANN is to generalise its acquired knowledge to similar but unseen input patterns.

Two other advantages of biological neural systems are the relative speed with which they perform computations and their robustness in the face of environmental and/or internal degradation. Thus *damage* to a part of an ANN usually has little impace on its computational capacity as a whole. This also means that ANNs are able to cope with the corruption of incoming signals (for example: due to background noise).

2.2. The Human Brain

We can view the human nervous system as a three-stage system, as shown in Figure 2.2.1. Central to the nervous system is the brain, which is shown as a neural network. The arrows pointing from left to right indicate the forward transmission of information through the system. The arrows pointing from right to left indicate the process of feedback in the system. The receptors convert stimuli from the external environment into electrical impulses that convey information to the neural network. The effectors convert electrical impulses from the neural network into responses to the environment.

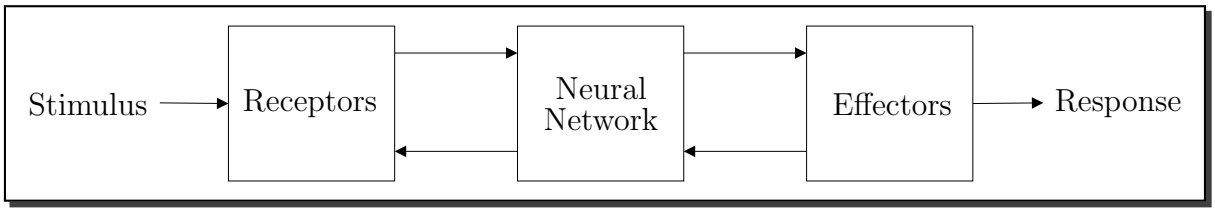

FIGURE 2.2.1. Representation of the Human Nervous System

2.3. Mathematical Model of a Neuron

A *neuron* is an information-processing unit that is fundamental to the operation of a neural network. Figure 2.3.1 shows the structure of a neuron, which will form the basis for a neural network.

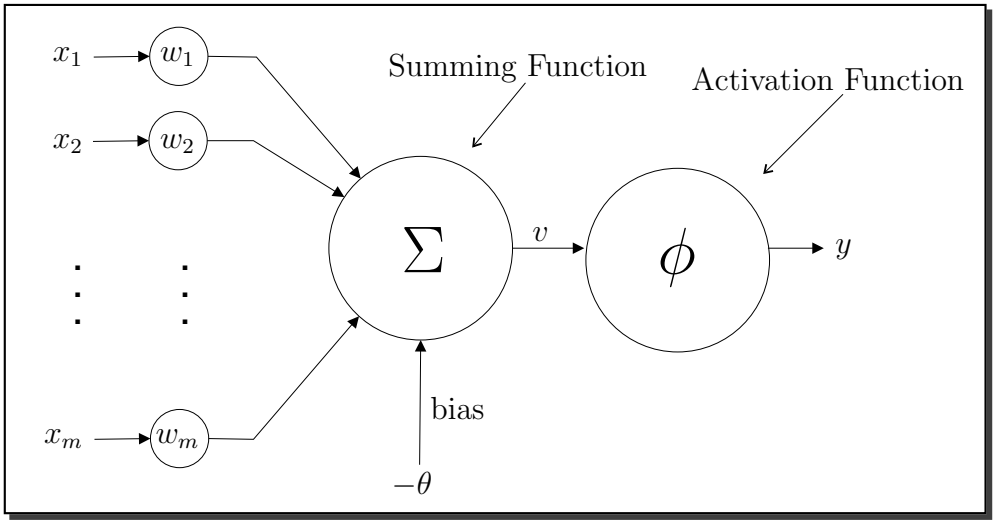

FIGURE 2.3.1. Mathematical Model of a Nonlinear Neuron

Mathematically we describe the neuron by the following equations:

$$\begin{aligned} v &= \left(\sum_{i=1}^{m} w_i x_i\right) - \theta \\ &= w \cdot x - \theta \\ &= \tilde{w} \cdot \tilde{x} \qquad \text{where} \quad \begin{cases} \tilde{w} = (-\theta, w_1, \ldots, w_m) \\ \tilde{x} = (1, x_1, \ldots, x_m) \end{cases} \end{aligned} \tag{2.3.1}$$

$$\begin{aligned} y &= \phi(v) \\ &= \phi(\tilde{w} \cdot \tilde{x}) \end{aligned} \tag{2.3.2}$$

2.3.1. Activation Function. The activation function, denoted $\phi(v)$, defines the output of the neuron in terms of the local field v. Three basic types of activation functions are as follows:

(1) *Threshold Function* (or *Heaviside Function*): A neuron employing this type of activation function is normally referred to as a *McCulloch-Pitts model* [**13**]. The model has an *all-or-none* property.

$$\phi(v) = \begin{cases} 1 & \text{if} \quad v \geq 0 \\ 0 & \text{if} \quad v < 0 \end{cases}$$

(2) *Piecewise-Linear Function*: This form of activation function may be viewed as an approximation to a non-linear amplifier. The following definition assumes the amplification factor inside the linear region is unity.

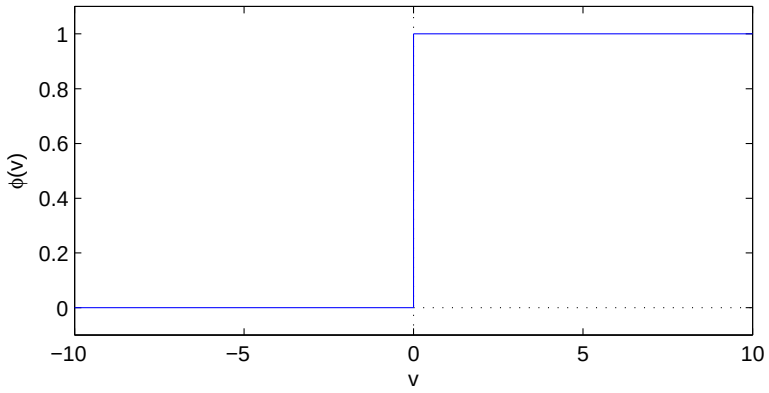

FIGURE 2.3.2. Threshold Function

$$\phi(v) = \begin{cases} 1 & \text{if} \quad v \geq \frac{1}{2} \\ v & \text{if} \quad -\frac{1}{2} < v < \frac{1}{2} \\ 0 & \text{if} \quad v \leq -\frac{1}{2} \end{cases}$$

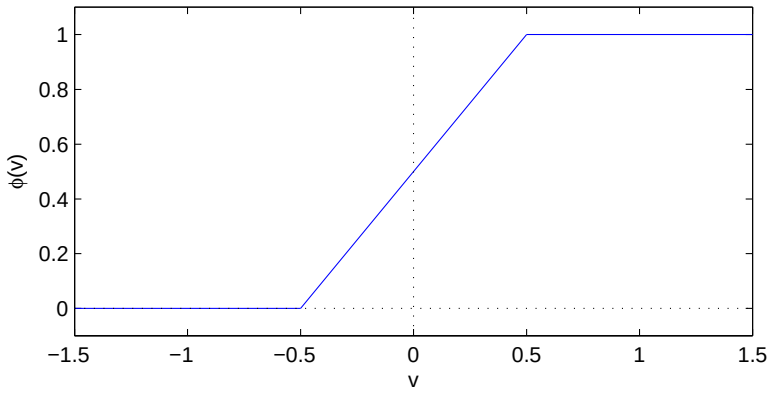

FIGURE 2.3.3. Piecewise-Linear Function

(3) *Sigmoid Function*: This is the most common form of activation function used in artificial neural networks. An example of a sigmoid function is the *logistic function*, defined by:

$$\phi(v) = \frac{1}{1 + \exp(-av)}$$

where $a > 0$ is the *slope parameter*. In the limit as $a \to \infty$, the sigmoid function simply becomes the threshold function. However, unlike the threshold function, the sigmoid function is continuously differentiable (differentiability is an important feature when it comes to network learning).

2.4. Architectures of Neural Networks

There are three fundamentally different classes of network architectures:

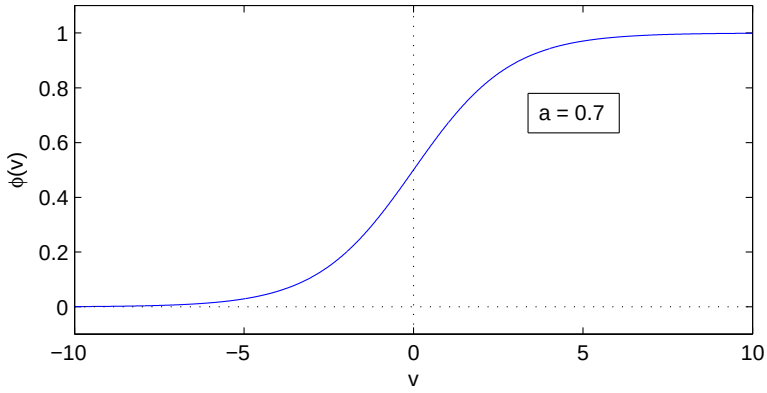

FIGURE 2.3.4. Sigmoid Function with $a = 0.7$

(1) *Single-Layer Feed-Forward Networks*

The simplest form of a layered network, consisting of an input layer of source nodes that project onto an output layer of neurons. The network is strictly feed-forward, no cycles of the information are allowed. Figure 2.4.1 shows an example of this type of network. The designation of single-layer refers to the output layer of neurons, the input layer is not counted since no computation is performed there.

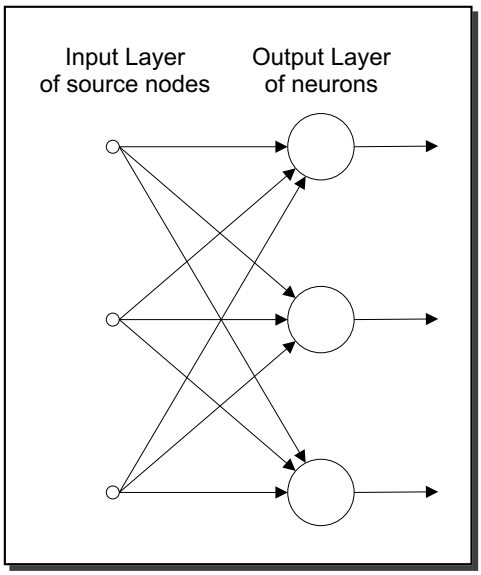

FIGURE 2.4.1. Single-Layer Feed-Forward Neural Network

(2) *Multi-Layer Feed-Forward Networks*

This class of feed-forward neural networks contains one or more *hidden layers*, whose computation nodes are correspondingly called *hidden neurons*. The hidden neurons intervene between the input and output layers, enabling the network to extract higher order statistics. Typically the neurons in each layer of the network

have as their inputs the output signals of the neurons in the preceding layer only. Figure 2.4.2 shows an example with one hidden layer. It is referred to as a 3-3-2 network for simplicity, since it has 3 source nodes, 3 hidden neurons (in the first hidden layer) and 2 output neurons. This network is said to be *fully connected* since every node in a particular layer is forward connected to every node in the subsequent layer.

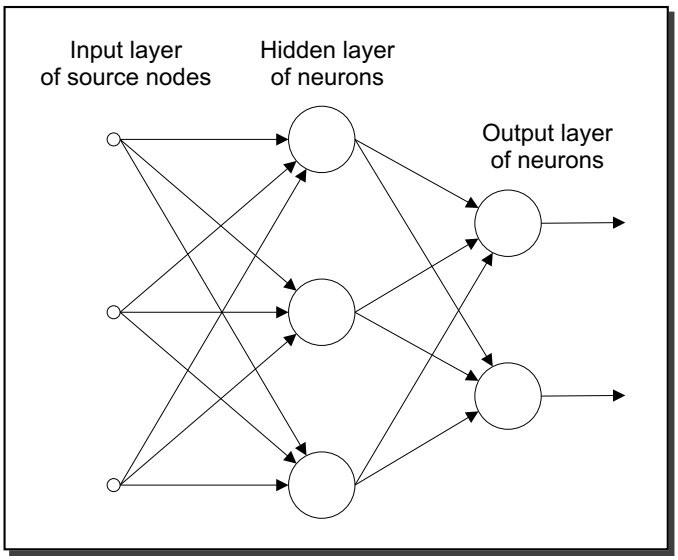

FIGURE 2.4.2. Multi-Layer Feed-Forward Neural Network

(3) *Recurrent Networks*

A recurrent neural network has a similar architecture to that of a multi-layer feed-forward neural network, but contains at least one *feedback* loop. This could be self-feedback, a situation where the output of a neuron is fed-back into its own input, or the output of a neuron could be feed to the inputs of one or more neurons on the same or preceding layers.

Feed-forward neural networks are simpler to implement and computationally less expensive to run than recurrent networks. However, the process of feedback is needed for a neural network to acquire a state representation, which enables it to model a dynamical system, as we shall see in Section 2.7.

2.5. The Perceptron

The simplest form of ANN is the perceptron, which consists of one single neuron (see Figure 2.5.1). The perceptron is built around a nonlinear neuron, namely, the McCulloch-Pitts model of a neuron.

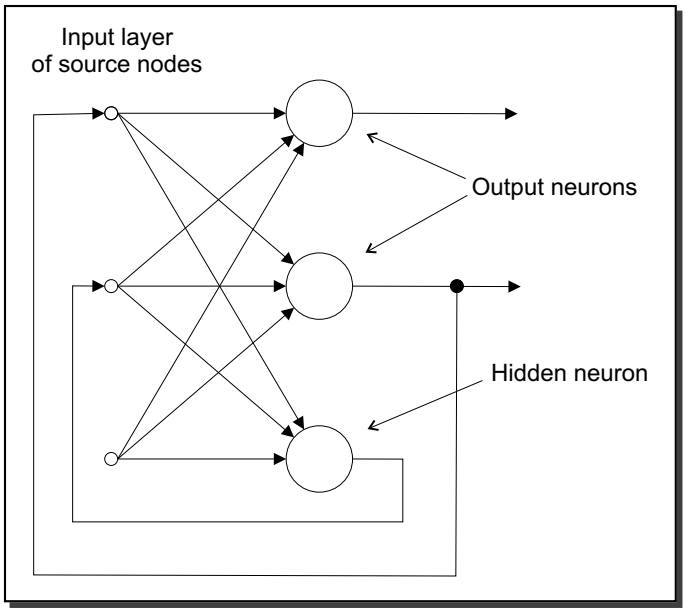

FIGURE 2.4.3. Recurrent Neural Network

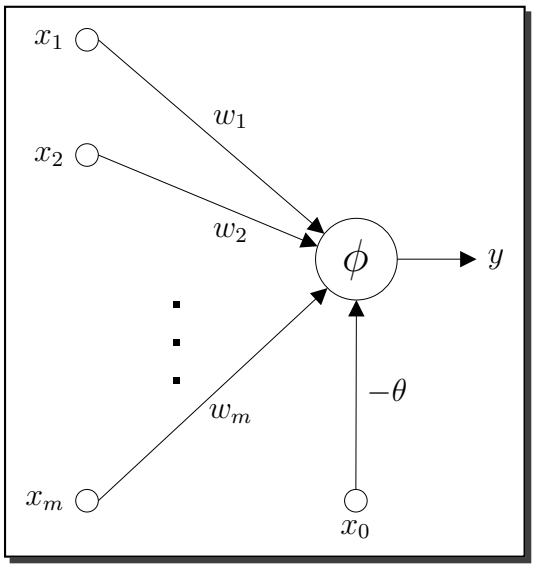

FIGURE 2.5.1. Perceptron with m inputs

The activation function ϕ of the perceptron is defined to be the *Heaviside step function* (see Section 2.3.1 for the definition) and the output is defined to be:

$$y = \phi(\tilde{w} \cdot \tilde{x}) \qquad \begin{cases} \tilde{w} = (-\theta, w_1, \ldots, w_m) \\ \tilde{x} = (1, x_1, \ldots, x_m) \end{cases} \tag{2.5.1}$$

The goal of the perceptron is to correctly classify the set of externally applied stimuli $x_1, x_2, \ldots, x_m$ to one of two classes $\mathcal{C}_1$ or $\mathcal{C}_2$. The point x is assigned to class $\mathcal{C}_1$ if the perceptron output y is 1 and to $\mathcal{C}_2$ if the output is 0.

The perceptron is remarkably powerful, in that it can compute most of the binary Boolean logic functions, where the output classes $\mathcal{C}_1$ and $\mathcal{C}_2$ represent *true* and *false* respectively. If we look at the binary Boolean logic functions, a perceptron with 2 inputs can compute 14 out of the possible 16 functions (Section 2.5.3 demonstrates why this is the case).

The perceptron theory is developed further in Sections 3.8 and 3.9 of [**5**].

2.5.1. Supervised Learning of the Perceptron. Given an initially arbitrary synaptic weight vector w, the perceptron can be trained to be able to calculate a specific *target function* t. The weights are adjusted in accordance to a *learning algorithm*, in response to some classified training examples, with the state of the network converging to the correct one. The perceptron is thus made to learn from experience.

Let us define a *labelled example* for the target function t to be $(x, t(x))$, where x is the input vector. The perceptron is given a *training sample* s, a sequence of labelled examples which constitute its *experience*. i.e:

$$s = (x_1, t(x_1)), (x_2, t(x_2)), \ldots, (x_m, t(x_m))$$

The weight vector w is altered, in accordance with the following learning algorithm, after each of the labelled examples is performed.

2.5.1.1. *Perceptron Learning Algorithm* [**14**]. For any *learning constant* $v > 0$, the weight vector w is updated at each stage in the following manner:

$$w' = w + v(t(x) - h_w(x))x$$

where:

- h_w is the output computed by the perceptron using the current weight vector w.
- $t(x)$ is the expected output of the perceptron.

2.5.2. Implementation in Java of the Perceptron Learning Algorithm. The Java code in Appendix B.1 implements the learning process described in Section 2.5.1 for a perceptron with 2 Boolean inputs. i.e. The perceptron attempts to learn any of the 16 binary Boolean logic functions specified by the user.

Figures 2.5.2 and 2.5.3 show the output, for different Boolean values of x and y, of the binary Boolean logic function $x \wedge \neg y$.

The Java program will correctly learn the 14 perceptron computable binary Boolean logic functions. For the other two, XOR and XNOR, it will fail to converge to a correct input mapping, as expected.

2.5.3. Linear Separability Problem. The reason that the perceptron can compute most of the binary Boolean logic functions is due to linear separability. In the case of the binary input perceptron and a given binary Boolean function, the external input points $x_1, x_2, \ldots, x_m$ are assigned to one of two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. If these two classes of points can

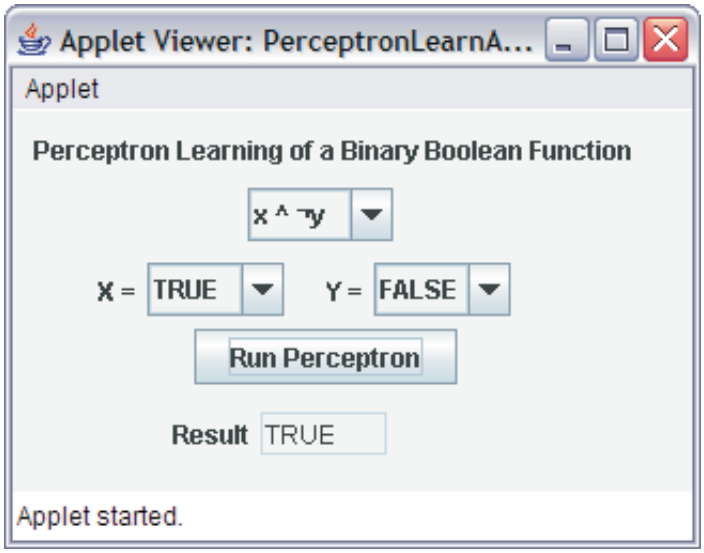

FIGURE 2.5.2. Perceptron Learning Applet: Sample Output 1

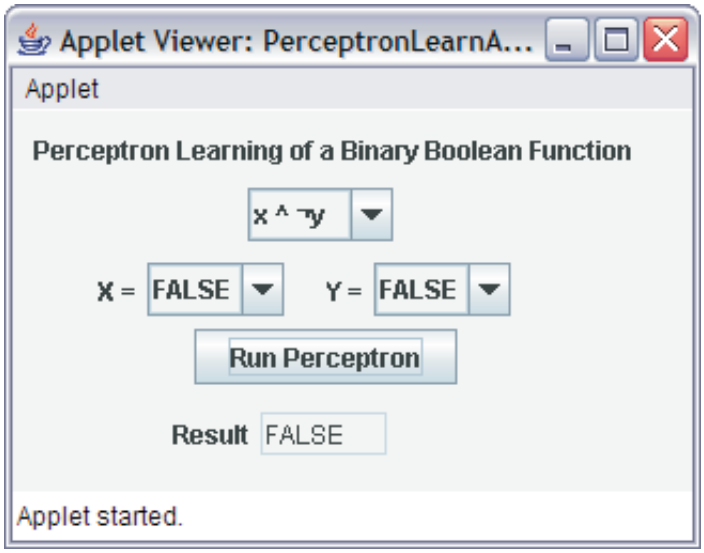

FIGURE 2.5.3. Perceptron Learning Applet: Sample Output 2

be separated by a straight line, then the Boolean function is said to be a linearly separable problem, and therefore perceptron computable.

For example, Figure 2.5.4 shows the input space for the AND binary Boolean function. We can see that the set of 'false' points can be separated from the 'true' points by the straight line $x_2 = \frac{3}{2} - x_1$. For the perceptron to compute the AND function, it needs to

output a '1' for all inputs in the shaded region:

$$x_2 \geq \frac{3}{2} - x_1$$

Rearranging this we can see the perceptron will output a '1' when:

$$x_1 + x_2 - \frac{3}{2} \geq 0$$

Therefore, the perceptrons weights and bias are set as follows:

$$\theta = -\frac{3}{2}$$
$$w_1 = 1$$
$$w_2 = 1$$

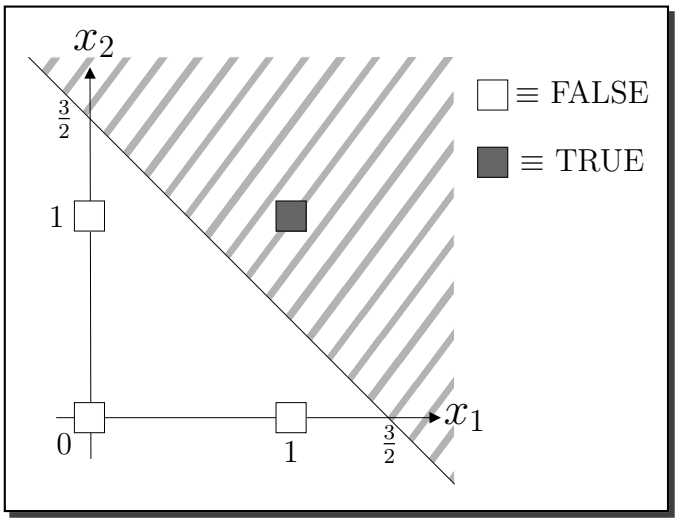

FIGURE 2.5.4. Input space for the AND binary Boolean logic function

Figure 2.5.5 shows the input space for the XOR function. Intuitively we cannot fit a single straight line to this diagram that will separate the 'true' values from the 'false' ones. Therefore, it is not a linearly separable problem and cannot be solved by a perceptron.

PROOF. Suppose XOR is computable by a perceptron and that w_1, w_2 and θ are its weights and bias. Then let us look at the perceptron output for various inputs:

$$(x_1, x_2) = (0,0) \to F \Rightarrow -\theta < 0 \qquad \Rightarrow \theta > 0 \tag{1}$$
$$(0,1) \to T \Rightarrow w_2 - \theta \geq 0 \qquad \Rightarrow w_2 \geq \theta > 0 \tag{2}$$
$$(1,0) \to T \Rightarrow w_1 - \theta \geq 0 \qquad \Rightarrow w_2 \geq \theta > 0 \tag{3}$$
$$(1,1) \to F \Rightarrow w_1 + w_2 - \theta < 0 \Rightarrow w_1 + w + 2 < \theta \tag{4}$$

Statement (4) is a contradiction to statements (1) to (3). Therefore the function XOR is *not* perceptron computable. □

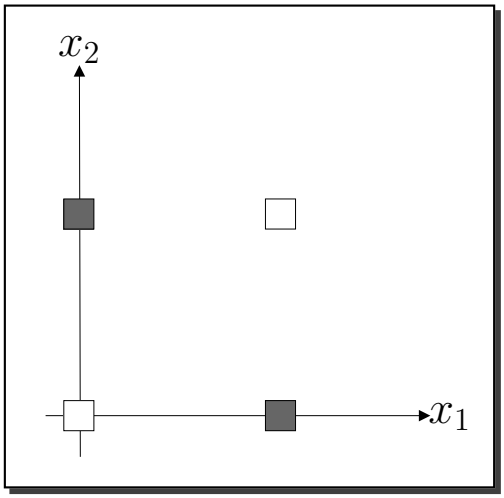

FIGURE 2.5.5. Input space for the XOR binary Boolean logic function

2.6. Radial-Basis Function Networks

We have seen that the perceptron is capable of solving most, but not all, of the binary Boolean logic functions. In fact, if we have an adequate preprocessing of its input signals, the perceptron would be able to approximate any boundary function.

One way of doing this is to have multiple layers of hidden neurons performing this preprocessing, as in multi-layer feed-forward neural networks.

An alternative way, one which produces results that are more transparent to the user, is the *Radial-Basis Function Network* (RBFN). The underlying idea is to make each hidden neuron represent a given region of the input space. When a new input signal is received, the neuron representing the closest region of input space, will activate a decisional path inside the network leading to the final result.

More precisely, the hidden neurons are defined by *Radial-Basis Functions* (RBFs), which express the similarity between any input pattern and the neurons assigned centre point by means of a distance measure.

2.6.1. What is a Radial-Basis Function? A RBF, ϕ, is one whose output is radially symmetric around an associated *centre point*, μ_c. That is, $\phi_c(x) = \phi(||x - \mu_c||)$, where $||\cdot||$ is a vector norm, usually the Euclidean norm. A set of RBFs can serve as a *basis* for representing a wide class of functions that are expressible as linear combinations of the chosen RBFs:

$$F(x) = \sum_{i=1}^{\infty} w_i \phi(||x - \mu_i||) \tag{2.6.1}$$

The following RBFs are of particular interest in the study of RBFNs:

(1) *Multi-quadratics:*

$$\phi(r) = (r^2 + c^2)^{1/2} \qquad \text{for some } c > 0 \text{ and } r \in \mathbb{R}$$

(2) *Inverse Multi-quadratics:*

$$\phi(r) = \frac{1}{(r^2 + c^2)^{1/2}} \qquad \text{for some } c > 0 \text{ and } r \in \mathbb{R}$$

(3) *Gaussian Functions:*

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right) \qquad \text{for some } \sigma > 0 \text{ and } r \in \mathbb{R}$$

The characteristic of RBFs is that their response decreases (or increases) monotonically with distance from a central point, as shown in Figure 2.6.1.

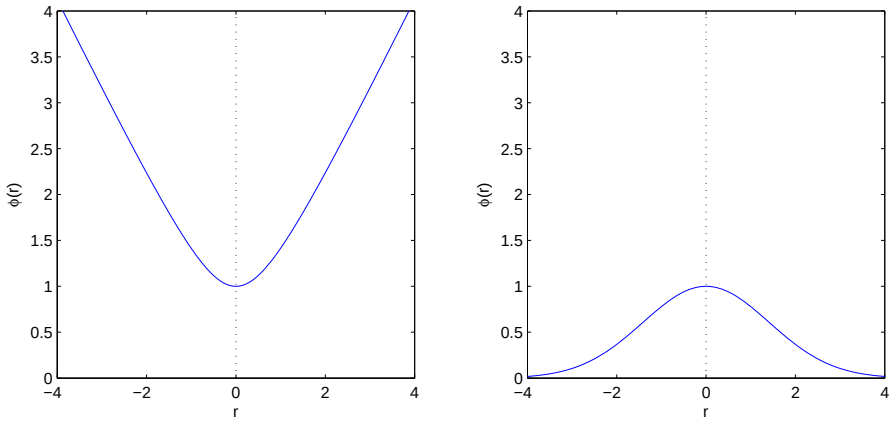

FIGURE 2.6.1. Multi-quadratic (left) and Gaussian Radial-Basis functions.

The Gaussian functions are also characterised by a *width* parameter, σ, which is also true of many RBFs. This can be tweaked to determine how quickly the function drops-off as we move away from the centre point.

2.6.2. Cover's Theorem on the Separability of Patterns. According to an early paper by Cover (1965), a pattern-classification task is more likely to be linearly separable in a high-dimensional rather than low-dimensional space.

When a radial-basis function network is used to perform a complex pattern-classification task, the problem is solved by nonlinearly transforming it into higher dimensions. The justification for this is found in *Cover's Theorem* on the *separability of patterns*:

THEOREM 1. ***Cover 1965*** [**2**]
A complex pattern-classification problem cast in a high-dimensional space nonlinearly is more likely to be linearly separable than in a low-dimensional space.

So once we have linearly separable patterns, the classification problem is relatively easy to solve.

2.6.3. Interpolation Problem. The input-output mapping of an RBF Network is produced by a *nonlinear mapping* from the input space to the hidden space, followed by a *linear mapping* from the hidden space to the output space. That is a mapping s from m_0-dimensional input space to one-dimensional output space:

$$s : \mathbb{R}^{m_0} \to \mathbb{R}$$

The learning procedure of the true mapping essentially amounts to *multivariate interpolation* in high-dimensional space.

DEFINITION 3. ***Interpolation Problem (see Section 5.3 in*** [5]***)***
Given a set of N different points $\{x_i \in \mathbb{R}^{m_0} | i = 1, 2, \ldots, N\}$ and a corresponding set of N real numbers $\{d_i \in \mathbb{R} | i = 1, 2, \ldots, N\}$, find a function $F : \mathbb{R}^{m_0} \to \mathbb{R}$ that satisfies the interpolation condition:

$$F(x_i) = d_i, \qquad i = 1, 2, \ldots, N \tag{2.6.2}$$

The *radial-basis functions* technique consists of choosing a function F of the form:

$$F(x) = \sum_{i=1}^{N} w_i \phi(||x - \mu_i||) \tag{2.6.3}$$

where $\{\phi(||x - \mu_i||) | i = 1, 2, \ldots, N\}$ is a set of N arbitrary functions, known as *basis functions*. The known data points $\mu_i \in \mathbb{R}$, $i = 1, 2, \ldots, N$ are taken to be the *centres* of these basis functions.

Inserting the interpolation conditions of Equation (2.6.2) into (2.6.3) gives a set of simultaneous equations. These can be written in the following matrix form:

$$\Phi w = d \qquad \text{where} \quad \begin{cases} d = [d_1, d_2, \ldots, d_N]^T \\ w = [w_1, w_2, \ldots, w_N]^T \\ \Phi = \{\phi_{ji} | i, j = 1, 2, \ldots, N\} \\ \phi_{ji} = \phi(||x_j - x_i||). \end{cases} \tag{2.6.4}$$

Assuming that Φ, the *interpolation matrix*, is nonsingular and therefore invertible, we can solve Equation (2.6.4) for the weight vector w:

$$w = \Phi^{-1} d. \tag{2.6.5}$$

2.6.4. Micchelli's Theorem. The following theorem, shows that for a large number of RBFs and certain other conditions, the interpolation matrix Φ from Equation (2.6.5) is invertible, as we require.

THEOREM 2. ***Micchelli 1986*** [**11**]
Let $\{x_i\}_{i=1}^N$ be a set of distinct points in $\mathbb{R}^{m_0}$. Then the N-by-N interpolation matrix Φ, whose ji^{th} element is $\phi_{ji} = \phi(||x_j - x_i||)$, is nonsingular.

So by Micchelli, the only condition needed for the interpolation matrix, defined using the RBFs from Section 2.6.1, to be nonsingular is that the basis function centre points $\{x_i\}_{i=1}^N$ must all be distinct.

2.6.5. Radial-Basis Function Network Technique. To solve the interpolation problem, RBFNs divide the input space into a number of sub-spaces and each subspace is generally only represented by a few hidden RBF units. There is a preprocessing layer, which activates those RBF units whose centre is sufficiently 'close' to the input signal. The output layer, consisting of one or more perceptrons, linearly combines the output of these RBF units.

2.6.6. Radial-Basis Function Networks Implementation. The construction of a RBFN involves three layers, each with a different role:

- The input layer, made up of source nodes, that connects the network to the environment.
- A hidden layer, which applies a nonlinear transformation from the input space to the hidden space. The hidden space is generally of higher dimension to the input space.
- The output layer which is linear. It supplies the response of the network to the signal applied to the input layer.

At the input of each hidden neuron, the *Euclidean distance* between the input vector and the neuron centre is calculated. This scalar value is then fed into that neurons RBF. For example, using the *Gaussian function*:

$$\phi(||x - \mu_i||) = \exp\left(-\frac{1}{2\sigma^2}||x - \mu_i||^2\right) \tag{2.6.6}$$

where $\begin{cases} x \text{ is the input vector.} \\ \mu_i \text{ is the centre of } i^{th} \text{ hidden neuron.} \\ \sigma \text{ is the width of the basis feature} \end{cases}$

The effect is that the response of the i^{th} hidden neuron is a maximum if the input stimulus vector x is centred at μ_i . If the input vector is not at the centre of the receptive field of the neuron, then the response is decreased according to how far it is away. The speed at which the response falls off in a Gaussian RBF is set by σ.

A RBFN may be singular-output, as shown in Figure 2.6.2, or multi-output, as shown in Figure 2.6.3. Each output is formed by a weighted sum, using the weights w_i calculated in Equation (2.6.5), of the neuron outputs and the unity bias.

The theory of RBFN is further developed in Sections 5 of [**5**] and 8.5 of [**4**]. There is a working example of a RBFN, showing its use in function approximation, in [**6**].

2.7. Recurrent Networks

Recurrent networks are neural networks with one or more *feedback loops*. The feedback can be of a *local* or *global* kind. The application of feedback enables recurrent networks to acquire *state* representations. The use of global feedback has the potential of reducing the memory requirement significantly over that of feed-forward neural networks.

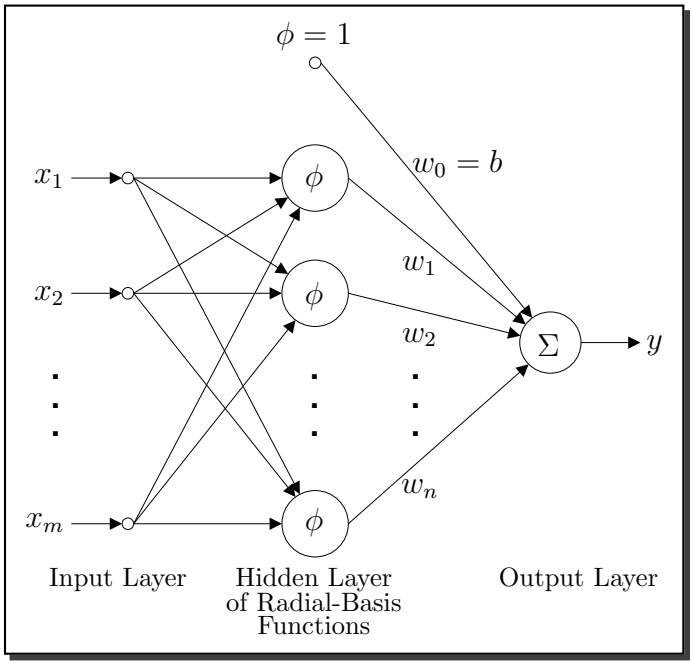

FIGURE 2.6.2. Radial-Basis Function Network with One Output

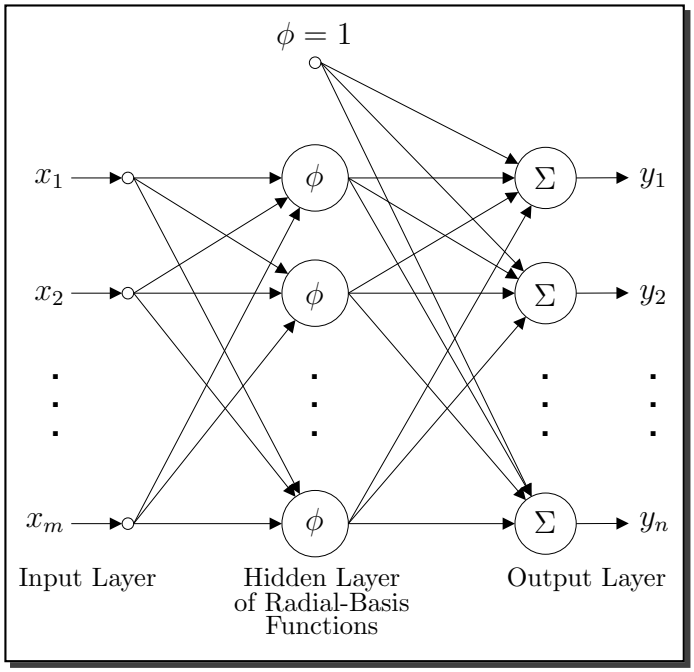

FIGURE 2.6.3. Radial-Basis Function Network with Multiple Outputs

2.7.1. Recurrent Network Architectures. The application of feedback can take various forms. We may have feedback from the output layer of the multi-layer neural

network to the input layer, or from the hidden layer to the input layer. If there are several hidden layers then the number of possible forms of global feedback is greatly increased. So there is a rich variety of architectural layouts to recurrent networks. Four such architectures are described here.

2.7.1.1. *Input-Output Recurrent Model.* Figure 2.7.1 show the generic architecture of a recurrent network. It has a single input, which is fed along a tapped-delay-line memory of p units. The structure of the multilayer network is arbitrary. The single output is fed-back to the input via another tapped-delay-line memory of q units. The current input value is $x(n)$ and the corresponding output, one time step ahead, is $y(n+1)$.

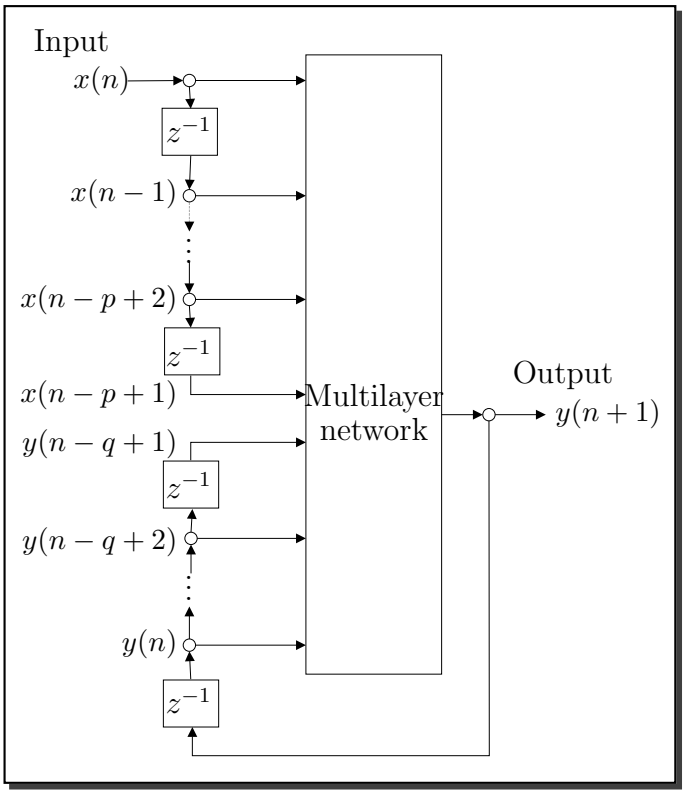

FIGURE 2.7.1. Nonlinear Autoregressive with Exogenous (NARX) Inputs Model

The input to the the first layer of the recurrent network consists of the following:

- Present and time-delayed input values, $x(n), \ldots, x(n-p+1)$, which represent the *exogenous* inputs. i.e. those originating from outside the system.
- Time-delayed output values fed-back, $y(n), \ldots, y(n-q+1)$, on which the model output $y(n+1)$ is *regressed*.

This type of network is more commonly known as a *nonlinear autoregressive with exogenous (NARX) input network*. The dynamics of the NARX model are described by:

$$y(n+1) = f(x(n-p+1), \ldots, x(n), y(n-q+1), \ldots, y(n)) \tag{2.7.1}$$

where f is a nonlinear function.

2.7.1.2. *State-Space Model.* Figure 2.7.2 shows the block diagram of the *state-space model* of a recurrent neural network. The neurons in the hidden layer describe the *state* of the network. The model is multi-input and multi-output. The output of the hidden layer is fed-back to the input layer via a layer of *context units*, which is a bank of unit delays. These context units store the outputs of the hidden neurons for one time step. The hidden neurons thus have some knowledge of their prior activations, which enables the network to perform learning tasks over time. The number of unit delays in the context layer determines the *order* of the model.

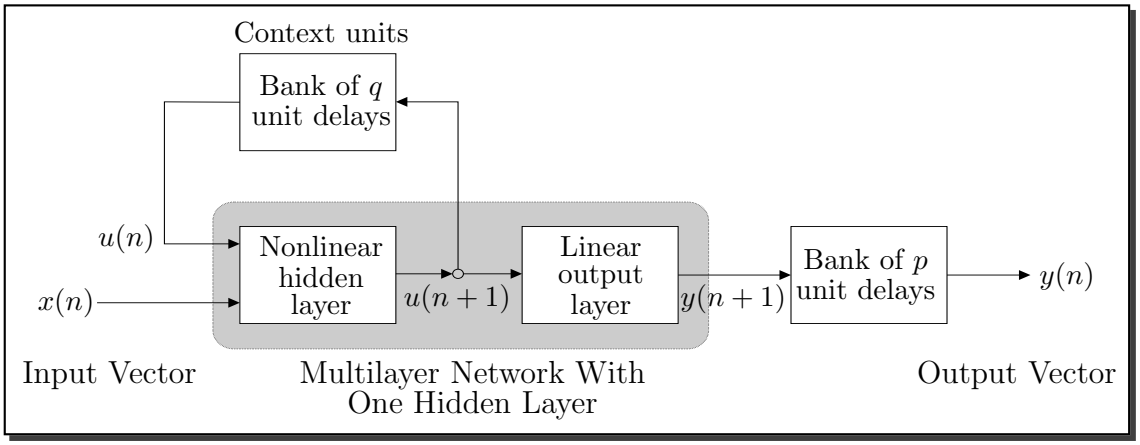

FIGURE 2.7.2. State-Space Model

The dynamical behaviour of this model can be described by the following pair of coupled equations:

$$u(n+1) = f(x(n), u(n)) \tag{2.7.2}$$

$$y(n) = Cu(n) \tag{2.7.3}$$

where $f(\cdot, \cdot)$ is the nonlinear function characterising the hidden layer, and C is the matrix of synaptic weights characterising the output layer.

2.7.1.3. *Recurrent Multilayer Neural Network.* A *recurrent multilayer neural network* (see Figure 2.7.3) contains one or more hidden layers, which generally makes it more effective than the single layer model from the previous section. Each layer of neurons has a feedback from its output to its input. In the diagram, $y_i(n)$ denotes the output of the i^{th} hidden layer, and $y_{out}(n)$ denotes the output of the output layer.

The dynamical behaviour of this model can be described by the following system of coupled equations:

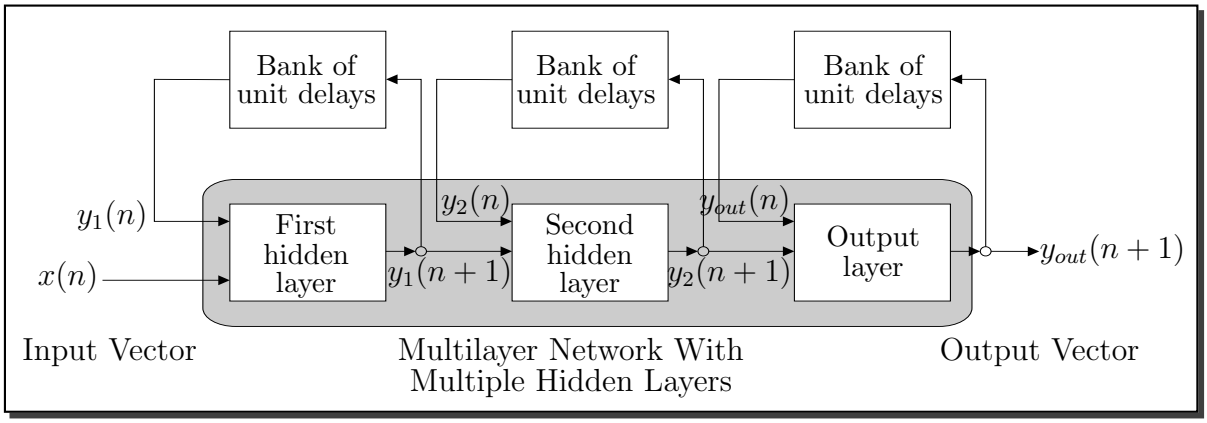

FIGURE 2.7.3. Recurrent Multilayer Neural Network

$$\begin{aligned} y_1(n+1) &= \phi_1(x(n), y_1(n)) \\ y_2(n+1) &= \phi_2(y_1(n+1), y_2(n)) \\ &\vdots \\ y_{out}(n+1) &= \phi_{out}(y_K(n+1), y_{out}(n)) \end{aligned} \tag{2.7.4}$$

where $\phi_1(\cdot,\cdot)$, $\phi_2(\cdot,\cdot)$ and $\phi_{out}(\cdot,\cdot)$ denote the activation functions of the first hidden layer, second hidden layer and the output layer respectively. K denotes the number of hidden layers in the network.

2.7.1.4. *Second Order Recurrent Network.* The term "order" can be used to refer to the way in which the induced local field of a neuron is defined. A typical induced field v_k for a neuron k in a multilayer neural network is defined by:

$$v_k = \sum_i w_{a,ki} x_i + \sum_j w_{b,kj} y_j \tag{2.7.5}$$

where x is the input signal, y is the feedback signal, w_a represents the synaptic weights for the input signal and w_b represents the synaptic weights for the fed-back signal. This type of neuron is referred to as a *first-order neuron*

When the induced local field v_k is defined as:

$$v_k = \sum_i \sum_j w_{kij} x_i y_j \tag{2.7.6}$$

the neuron is referred to as a *second-order neuron.* i.e. When the input signal and the fed-back signal are combined using multiplications. A single weight w_{kij} is used for a neuron k that is connected to input nodes i and j.

Figure 2.7.4 shows a *second-order neural network* which is a network made up of second-order neurons.

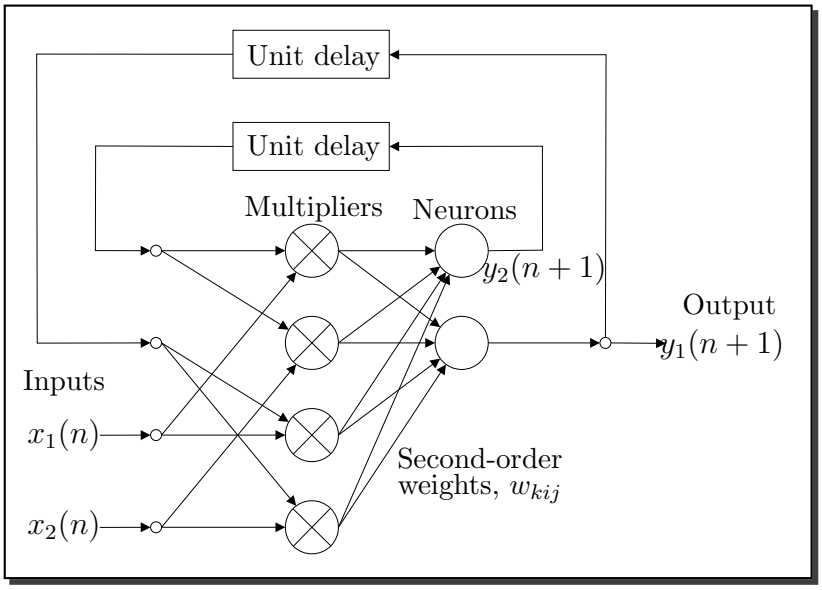

FIGURE 2.7.4. Second-Order Recurrent Network

The dynamical behaviour of this model is described by the following coupled equations:

$$v_k(n) = b_k + \sum_i \sum_j w_{kij} x_i(n) y_j(n) \tag{2.7.7}$$

$$y_k(n+1) = \phi(v_k(n)) \tag{2.7.8}$$

where $v_k(n)$ is the induced local field of neuron k, with associated bias b_k.

2.7.2. Modelling Dynamical Behaviour. In *static* neural networks, the output vector is always a direct consequence of the input vector. No memory of previous input patterns is kept. This is sufficient for solving static association problems such as classification, where the goal is to produce only one output pattern for each input pattern. However, the evolution of an input pattern may be more interesting than its final state.

A *dynamic* neural network is one that can be taught to recognise sequences of input vectors and generate the corresponding output. The application of feedback enables recurrent networks to acquire *state* representations, which makes them suitable for studying nonlinear dynamical systems. They can perform mappings that are functions of time or space and converge to one of a number of limit points. As a result, they are capable of performing more complex computations than feed-forward networks.

The subject of neural networks viewed as nonlinear dynamical systems is referred to as *neurodynamics*. There is no universally agreed upon definition of what we mean by neurodynamics, but the systems of interest possess the following characteristics:

(1) *A large number of degrees of freedom*: The human brain is estimated to contain about 10 billion neurons, each modelled by a state variable. It is this sheer number or neurons that gives the brain highly complex calculation and fault-tolerant capability.

(2) *Nonlinearity*: Nonlinearity is essential for creating a universal computing machine.
(3) *Dissipation*: This is characterised by the reduction in dimension of the state-space volume.
(4) *Noise*: This is an intrinsic characteristic of neurodynamical systems.

When the number of neurons, N, in a recurrent network is large, the neurodynamical model it describes possesses the characteristics outlined above. Such a neurodynamical model can have complicated attractor structures and therefore exhibit useful computational capabilities.

2.7.2.1. *Associative Memory System.* Associative memory networks are simple one or two-layer networks that store patterns for subsequent retrieval. They simulate one of the simplest forms of human learning, that of memorisation: storing patterns in memory with little or no inferring involved. Neural networks can act as associative memories where some P different patterns are stored for subsequent recall. When an input pattern is presented to a network with stored patterns, the pattern associated with it is output. Associative memory neurodynamical systems have the form:

$$\tau_j \dot{y}_j(t) = -y_j(t) + \phi\left(\sum_i w_{ji} y_i(t)\right) + I_j, \quad j = 1, 2, \ldots, N \tag{2.7.9}$$

where y_j represents the state of the jth neuron, τ_j is the relaxation time[1] for neuron j and w_{ji} is the synaptic weight between neurons j and i.

The outputs $y_1(t), y_2(t), \ldots, y_N(t)$ of the individual neurons constitute the state vector of the system.

The *Hopfield Network* is an example of a recurrent associative network. The weight matrix W of such a network is symmetric, with diagonal elements set to zero. Figure 2.7.5 shows the architecture of the Hopfield network. Hopfield networks can store some P prototype patterns $\pi^1, \pi^2, \ldots, \pi^P$ called *fixed-point attractors*. The locations of the attractors in the input space are determined by the weight matrix W. These stored patterns may be computed directly or learnt by the network.

To recall a pattern π^k, the network recursively feeds the output signals back into the inputs at each time step, until the network output stabilises.

For discrete-time systems the outputs of the network are defined to be:

$$y_i(t+1) = \operatorname{sgn}\left(\sum_j w_{ij} y_j(t) - \theta\right) \qquad \text{for } i = 1, 2, \ldots, N \tag{2.7.10}$$

where $\operatorname{sgn}(\cdot)$ (*signum* function) is a bipolar activation function for each neuron, defined as:

$$\operatorname{sgn}(v) = \begin{cases} +1 & \text{if } v > 0 \\ -1 & \text{if } v < 0 \end{cases}$$

If $v = 0$, then the output of the signum function is arbitrary and, by convention, the neuron output will remain unchanged from the previous time step.

[1]The relaxation times τ_j allow neurons to run at different speeds.

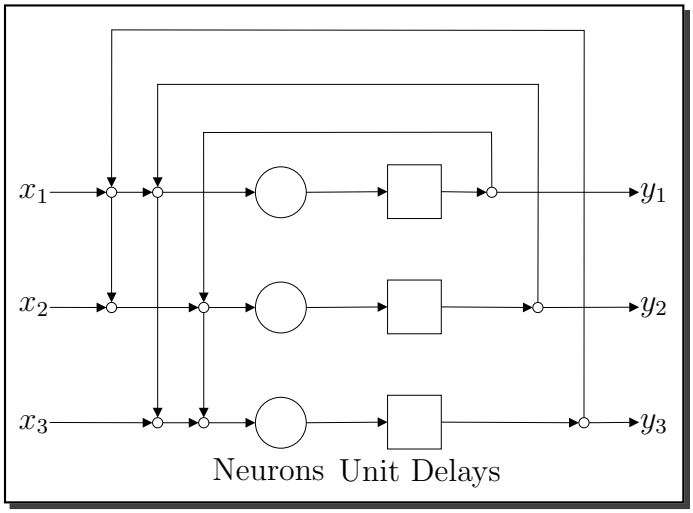

FIGURE 2.7.5. Hopfield Network consisting of $N = 3$ neurons

The continuous version is governed by the following set of differential equations:

$$\tau_i \dot{y}_i = -y_i + \phi\left(\sum_j w_{ij} y_j\right) \qquad \text{for } i = 1, 2, \ldots, N \tag{2.7.11}$$

where τ_i is the relaxation time and $\phi(\cdot)$ is the nonlinear activation function:

$$\phi(v) = \frac{1}{1 + \exp(-v)}$$

Starting with an initial input vector $x(0)$ at time $t = 0$, all neuron outputs are computed *simultaneously* before being fed back into the inputs. No further external inputs are applied to the network. This process is repeated until the network stabilises on a fixed point, corresponding to a prototype pattern.

To simulate an associative memory, the network should converge to the fixed point π^j that is closest to the input vector $x(0)$ after some finite number of iterations.

For further reading on associative memory systems see Sections 14 of [**5**] and 5 of [**12**].

2.7.2.2. *Input-Output Mapping System.* In a mapping network, the input space is mapped onto the output space. For this type of system, recurrent networks respond *temporally* to an externally applied input signal. The architecture of this type of recurrent network is generally more complex than that of an associative memory model. Examples of the types of architectures were given in Section 2.7.1.

In general, the mapping system is governed by coupled differential equations of the form:

$$\tau_i \dot{y}_i(t) = -y_i(t) + \phi\left(\sum_j w_{ij} y_j + x_i\right) \qquad \text{for } i = 1, 2, \ldots, N \tag{2.7.12}$$

where y_i represents the state of the ith neuron, τ_i is the relaxation time, w_{ij} is the synaptic weight from neuron j to i and x_i is an external input to neuron i.

The weight matrix W of such a network is asymmetric and convergence of the system is not always assured. Because of their complexity, they can exhibit more exotic dynamical behaviour than that of associative networks. The system can evolve in one of four ways:

- Convergence to a stable fixed point.
- Settle down to a periodic oscillation or stable limit cycle.
- Tend towards quasi-periodic behaviour.
- Exhibit chaotic behaviour.

To run the network the input nodes are first of all clamped to a specified input vector $x(0)$. The network is then run and the data flows through the network, depending upon the topology. The activations of the neurons are then computed and recomputed, until the network stabilises (assuming it does stabilise). The output vector $y(t)$ can then be read from the output neurons.

For further reading on input-output mapping systems see Section 15 of [**5**].

CHAPTER 3

Wavelet Neural Networks

3.1. Introduction

Wavelet neural networks combine the theory of wavelets and neural networks into one. A wavelet neural network generally consists of a feed-forward neural network, with one hidden layer, whose activation functions are drawn from an orthonormal wavelet family.

One applications of wavelet neural networks is that of function estimation. Given a series of observed values of a function, a wavelet network can be trained to learn the composition of that function, and hence calculate an expected value for a given input.

3.2. What is a Wavelet Neural Network?

The structure of a wavelet neural network is very similar to that of a (1+ 1/2) layer neural network. That is, a feed-forward neural network, taking one or more inputs, with one hidden layer and whose output layer consists of one or more linear combiners or *summers* (see Figure 3.2.1). The hidden layer consists of neurons, whose activation functions are drawn from a wavelet basis. These wavelet neurons are usually referred to as *wavelons*.

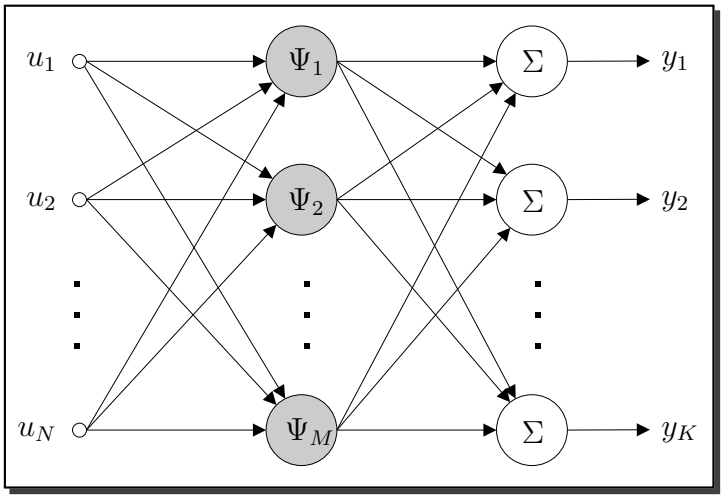

FIGURE 3.2.1. Structure of a Wavelet Neural Network

There are two main approaches to creating wavelet neural networks.

- In the first the wavelet and the neural network processing are performed separately. The input signal is first decomposed using some wavelet basis by the neurons in the hidden layer. The wavelet coefficients are then output to one or more summers whose input weights are modified in accordance with some learning algorithm.
- The second type combines the two theories. In this case the translation and dilation of the wavelets along with the summer weights are modified in accordance with some learning algorithm.

In general, when the first approach is used, only dyadic dilations and translations of the mother wavelet form the wavelet basis. This type of wavelet neural network is usually referred to as a *wavenet*. We will refer to the second type as a *wavelet network*.

3.2.1. One-Dimensional Wavelet Neural Network. The simplest form of wavelet neural network is one with a single input and a single output. The hidden layer of neurons consist of wavelons, whose input parameters (possibly fixed) include the wavelet dilation and translation coefficients. These wavelons produce a non-zero output when the input lies within a small area of the input domain. The output of a wavelet neural network is a linear weighted combination of the wavelet activation functions.

Figure 3.2.2 shows the form of a single-input wavelon. The output is defined as:

$$\psi_{\lambda,t}(u) = \psi\left(\frac{u-t}{\lambda}\right) \tag{3.2.1}$$

where λ and t are the dilation and translation parameters respectively.

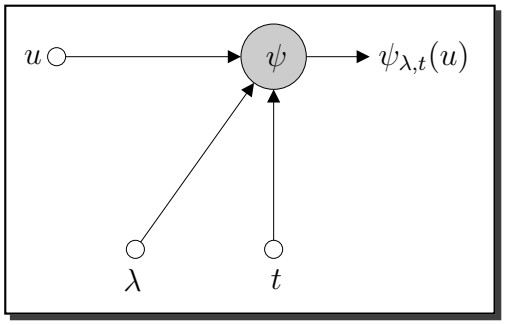

FIGURE 3.2.2. A Wavelet Neuron

3.2.1.1. *Wavelet Network.* The architecture of a single input single output wavelet network is shown in Figure 3.2.3. The hidden layer consists of M wavelons. The output neuron is a *summer*. It outputs a weighted sum of the wavelon outputs.

$$y(u) = \sum_{i=1}^{M} w_i \psi_{\lambda_i,t_i}(u) + \bar{y} \tag{3.2.2}$$

The addition of the $\bar{y}$ value is to deal with functions whose mean is nonzero (since the wavelet function $\psi(u)$ is zero mean). The $\bar{y}$ value is a substitution for the scaling function $\phi(u)$, at the largest scale, from wavelet multiresolution analysis (see Section 1.3.3).

In a wavelet network all parameters $\bar{y}$, w_i, t_i and λ_i are adjustable by some learning procedure (see Section 3.3).

3.2.1.2. *Wavenet.* The architecture for a wavenet is the same as for a wavelet network (see Figure 3.2.3), but the t_i and λ_i parameters are fixed at initialisation and not altered by any learning procedure.

One of the main motivations for this restriction comes from wavelet analysis. That is, a function $f(\cdot)$ can be approximated to an arbitrary level of detail by selecting a sufficiently large L such that

$$f(u) \approx \sum_{k} \langle f, \phi_{L,k} \rangle \phi_{L,k}(u) \tag{3.2.3}$$

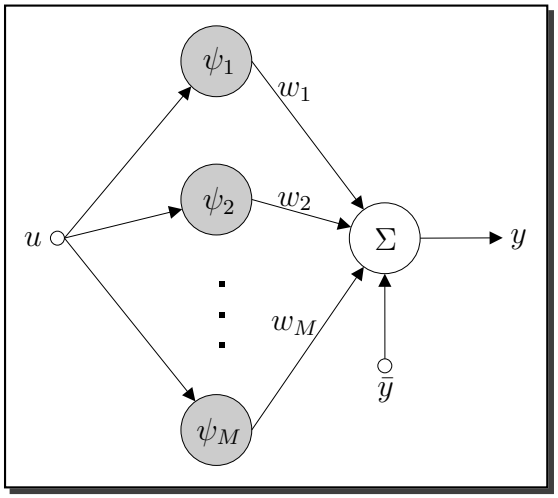

FIGURE 3.2.3. A Wavelet Neural Network

where $\phi_{L,k}(u) = 2^{L/2}\phi(2^L u - k)$ is a scaling function dilated by 2^L and translated by dyadic intervals 2^{-L}.

The output of a wavenet is therefore

$$y(u) = \sum_{i=1}^{M} w_i \phi_{\lambda_i,t_i}(u) \tag{3.2.4}$$

where M is sufficiently large to cover the domain of the function we are analysing. Note that an adjustment of $\bar{y}$ is not needed since the mean value of a scaling function is nonzero.

3.2.2. Multidimensional Wavelet Neural Network. The input in this case is a multidimensional vector and the wavelons consist of multidimensional wavelet activation functions. They will produce a non-zero output when the input vector lies within a small area of the multidimensional input space. The output of the wavelet neural network is one or more linear combinations of these multidimensional wavelets.

Figure 3.2.4 shows the form of a wavelon. The output is defined as:

$$\Psi(u_1, \ldots, u_N) = \prod_{n=1}^{N} \psi_{\lambda_n,t_n}(u_n) \tag{3.2.5}$$

This wavelon is in effect equivalent to a multidimensional wavelet.

The architecture of a multidimensional wavelet neural network was shown in Figure 3.2.1. The hidden layer consists of M wavelons. The output layer consists of K summers. The output of the network is defined as

$$y_j = \sum_{i=1}^{M} w_{ij} \Psi_i(u_1, \ldots, u_N) + \bar{y}_j \quad \text{for } j = 1, \ldots, K \tag{3.2.6}$$

where the $\bar{y}_j$ is needed to deal with functions of nonzero mean.

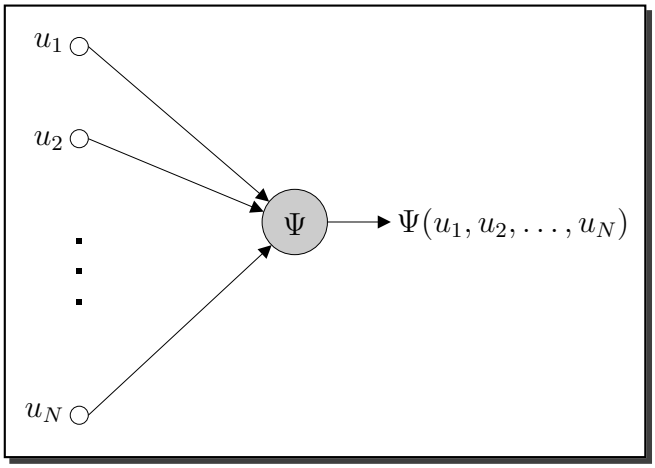

FIGURE 3.2.4. A Wavelet Neuron with a Multidimensional Wavelet Activation Function

Therefore the input-output mapping of the network is defined as:

$$\mathbf{y}(\mathbf{u}) = \sum_{i=1}^{M} \mathbf{w_i} \Psi_i(\mathbf{u}) + \bar{\mathbf{y}} \quad \text{where} \quad \begin{cases} \mathbf{y} = (y_1, \ldots, y_K) \\ \mathbf{w_i} = (w_{i1}, \ldots, w_{iK}) \\ \mathbf{u} = (u_1, \ldots, u_N) \\ \bar{\mathbf{y}} = (\bar{y}_1, \ldots, \bar{y}_K) \end{cases} \tag{3.2.7}$$

3.3. Learning Algorithm

One application of wavelet neural networks is function approximation. Zhang and Benveniste [**1**] proposed an algorithm for adjusting the network parameters for this application. We will concentrate here on the one-dimensional case, and look at both types of wavelet neural networks described in Section 3.2.

Learning is performed from a random sample of observed input-output pairs $\{u, f(u) = g(u) + \epsilon\}$ where $g(u)$ is the function to be approximated and ϵ is the measurement noise. Zhang and Benveniste suggested the use of a stochastic gradient type algorithm for the learning.

3.3.1. Stochastic Gradient Algorithm for the Wavelet Network. The parameters $\bar{y}$, w_i's, t_i's and λ_i's should be formed into one vector θ. Now $y_\theta(u)$ refers to the wavelet network, defined by (3.2.2) (shown below for convenience), with parameter vector θ.

$$y_\theta(u) = \sum_{i=1}^{M} w_i \psi \left(\frac{u - t_i}{\lambda_i} \right) + \bar{y}$$

The objective function to be minimised is then

$$C(\theta) = \frac{1}{2} \mathrm{E}\{(y_\theta(u) - f(u))^2\}. \tag{3.3.1}$$

The minimisation is performed using a *stochastic gradient algorithm*. This recursively modifies θ, after each sample pair $\{u_k, f(u_k)\}$, in the opposite direction of the gradient of

$$c(\theta, u_k, f(u_k)) = \frac{1}{2}(y_\theta(u_k) - f(u_k))^2. \tag{3.3.2}$$

The gradient for each parameter of θ can be found by calculating the partial derivatives of $c(\theta, u_k, f(u_k))$ as follows:

$$\begin{aligned}
\frac{\partial c}{\partial \bar{y}} &= e_k \\
\frac{\partial c}{\partial w_i} &= e_k \psi(z_{ki}) \\
\frac{\partial c}{\partial t_i} &= -e_k w_i \frac{1}{\lambda_i} \psi'(z_{ki}) \\
\frac{\partial c}{\partial \lambda_i} &= -e_k w_i \left(\frac{u_k - t_i}{\lambda_i^2} \right) \psi'(z_{ki})
\end{aligned}$$

where $e_k = y_\theta(u_k) - f(u_k)$, $z_{ki} = \dfrac{u_k - t_i}{\lambda_i}$ and $\psi'(z) = \dfrac{d\psi(z)}{dz}$.

To implement this algorithm, a *learning rate* value and the number of *learning iterations* need to be chosen. The learning rate $\gamma \in (0, 1]$ determines how fast the algorithm attempts to converge. The gradients for each parameter are multiplied by γ before being used to modify that parameter. The learning iterations determine how many times the training data should be fed through the learning process. The larger this value is, the closer the convergence of the network to the function should be, but the computation time will increase.

3.3.2. Stochastic Gradient Algorithm for the Wavenet. As for the wavelet network, we can group the parameters $\bar{y}$, w_i's, t_i's and λ_i's together into one vector θ. In the wavenet model, however, the t_i and λ_i parameters are fixed at initialisation of the network. The function $y_\theta(u)$ now refers to the wavenet defined by (3.2.4) (shown below for convenience), with parameter vector θ.

$$y_\theta(u) = \sum_{i=1}^{M} w_i \sqrt{\lambda_i} \phi(\lambda_i u - t_i)$$

The objective function to be minimised is as (3.3.1), and this is performed using a *stochastic gradient algorithm*. After each $\{u_k, f(u_k)\}$ the w_i's in θ are modified in the opposite direction of the gradient of $c(\theta, u_k, f(u_k))$ (see Equation (3.3.2)). This gradient is found by calculating the partial derivative

$$\frac{\partial c}{\partial w_i} = e_k \sqrt{\lambda_i} \phi(z_{ki})$$

where $e_k = y_\theta(u_k) - f(u_k)$ and $z_{ki} = \lambda_i u_k - t_i$.

As for the wavelet network, a *learning rate* and the number of *learning iterations* need to be chosen to implement this algorithm (see Section 3.3.1).

3.3.3. Constraints on the Adjustable Parameters (Wavelet Network only). Zhang and Benveniste proposed to set constraints on the adjustable parameters to help prevent the stochastic gradient algorithm from diverging. If we let $f : \mathcal{D} \to \mathbb{R}$ be the function we are trying to approximate, where $\mathcal{D} \in \mathbb{R}$ is the domain of interest, then:

(1) The wavelets should be kept in or near the domain $\mathcal{D}$. To achieve this, choose another domain $\mathcal{E}$ such that $\mathcal{D} \subset \mathcal{E} \subset \mathbb{R}$. Then require

$$t_i \in \mathcal{E} \quad \forall i. \tag{3.3.3}$$

(2) The wavelets should not be compressed beyond a certain limit, so we select $\epsilon > 0$ and require

$$\lambda_i > \epsilon \quad \forall i. \tag{3.3.4}$$

Further constraints are proposed for multidimensional wavelet network parameters in [**1**].

3.3.4. Initialising the Adjustable Parameters for the Wavelet Network. We want to be able to approximate $f(u)$ over the domain $\mathcal{D} = [a, b]$ using the wavelet network defined as (3.2.2). This network is single input and single output, with M hidden wavelons. Before we can run the network, the adjustable parameters $\bar{y}$, w_i, t_i and λ_i need to be initialised.

- $\bar{y}$ should be estimated by taking the average of the available observations.
- The w_i's should be set to zero.
- To set the t_i's and λ_i's select a point p within the interval $[a, b]$ and set

$$t_1 = p \quad \text{and} \quad \lambda_1 = \mathcal{E}(b - a)$$

where $\mathcal{E} > 0$ is typically set to 0.5. We now repeat this initialisation: taking the intervals $[a, p]$ and $[p, b]$, and setting t_2, λ_2 and t_3, λ_3 respectively. This is recursively repeated until every wavelon is initialised. This procedure applies when the number of wavelons M is of the form $2^L - 1$, for some $L \in \mathbb{Z}^+$. If this is not the case, this procedure is applied up until the remaining number of uninitialised wavelons cannot cover the next resolution level. At this point these remaining wavelons are set to random translations within this next resolution level.

3.3.5. Initialising the Parameters for the Wavenet. We want to be able to approximate $f(u)$ over the domain $\mathcal{D} = [a, b]$ using the wavelet network defined as (3.2.4). Before we can run the network for this purpose, we need to initialise the (adjustable and non-adjustable) parameters w_i, t_i and λ_i.

- The w_i's should be set to zero.
- To set the t_i's and λ_i's we first need to choose a resolution level L. The λ_i's are then set to 2^L. The t_i's are set to intervals of 2^{-L} and all satisfy $t_i \in \mathcal{E}$, where $\mathcal{E}$ is a domain satisfying $\mathcal{D} \subset \mathcal{E} \subset \mathbb{R}$.

3.4. Java Program

The Java source code in Appendix C.1 implements both a wavelet network and a wavenet. It does this by modelling the wavelons and the wavelet neural network as objects (called classes in Java):

- Each wavelon class (in file 'Wavelon.java') is assigned a translation and a dilation parameter and includes methods for changing and retrieving these parameters. The wavelon class also has a method for firing the chosen wavelet activation function.
- The wavelet neural network class (in file 'WNN.java') stores the wavelons, along with the network weights. It includes methods for adding wavelons at initialisation and retrieving the network weights and wavelon parameters. For the wavelet neural network to be complete, methods to initialise, perform the learning and to run the network need to be defined. These are specific to the type of wavelet neural network that is needed, so they are defined by extending this class.
- The wavelet network class (in file 'WaveletNet.java') extends the wavelet neural network class. It implements the wavelet network where the wavelet parameters as well as the network weights can be modified by the learning algorithm. The 'Gaussian derivative' function (see Section 1.7.1) was chosen to be the mother wavelet for the basis of wavelet activation functions.
- The wavenet class (in file 'Wavenet.java') extends the wavelet neural network class. It implements the wavenet where only the network weights are subject to change by the learning algorithm. The 'Battle-Lémarie' scaling function (see Section 1.7.2) was chosen for the basis of activation functions.

When the program is run, the training data stored in the file 'training.txt' needs to be in the same directory as the program package 'waveletNN'. The file must consist of the lines 'u_k $f(u_k)$', where $\{u_k, f(u_k)\}$ is a training sample, with the first line of the file containing the number of samples in the file. The Matlab file 'export2file.m' (in Appendix C.1.6 will output a set of training data to the file 'training.txt' in the correct format.

The user is given the choice whether to use the 'Wavelet Network' or 'Dyadic Wavenet' implementation. The program performs the corresponding stochastic gradient learning algorithm, based upon the learning parameters input by the user. The wavelet parameters and the network weights are then output to the file 'coeffs.txt' in the same directory as the program package.

These parameters can then be used by a program such as Matlab to provide a functional estimate of the training data. The Matlab files 'gaussian.m' and 'lemarie.m', in Appendix C.1, are such functions for the parameters output by the 'Wavelet Network' and 'Dyadic Wavenet' respectively.

3.4.1. Wavelet Network Implementation. The following pseudo code describes the implementation of the learning algorithm and parameter initialisation, described in Section 3.3, for the wavelet network.

The function 'initialise' is a public method in the class 'WaveletNet'.

method initialise(a, b)
Set $\mathcal{E}$, the open neighbourhood of the domain $\mathcal{D}$.
Set ϵ, the minimum dilation value.
Calculate n, the number of complete resolution levels
Call initComplete(a, b, n)
while There are uninitialised wavelons **do**
    Set t_i randomly within the interval $[a, b]$
    Set λ_i to be the highest resolution
    Initialise a wavelon with t_i and λ_i values
end while

The function 'initComplete' is a private method in the class 'WaveletNet'. It recursively initialises the wavelons up to resolution level 'n'.

method initComplete(a, b, n)
Set $t_i = \dfrac{a+b}{2}$ {Let p be the midpoint of $[a, b]$}
Set $\lambda_i = \dfrac{b-a}{2}$ {Let $\mathcal{E}$ equal 0.5}
Initialise a wavelon with t_i and λ_i
if $(n \leq 1)$ **then**
    Return
else
    initComplete$(a, t_i, n-1)$
    initComplete$(t_i, b, n-1)$
end if

The function 'learn' is a public method in the class 'WaveletNet'.

method learn$(Training_Data, Iterations, Rate)$
Calculate $\bar{y}$ from the training data
for $j \in 1, \ldots, Iterations$ **do**
    for $k \in 1, \ldots, NumTrainingSamples$ **do**
        Adjust $\bar{y}$ by $Rate \times \dfrac{\partial c}{\partial \bar{y}}$
        for $i \in 1, \ldots, NumWavelons$ **do**
            Adjust Weights w_i by $Rate \times \dfrac{\partial c}{\partial w_i}$
            Adjust Translations t_i by $Rate \times \dfrac{\partial c}{\partial t_i}$ {Ensuring t_i stays within domain $\mathcal{E}$}
            Adjust Dilations λ_i by $Rate \times \dfrac{\partial c}{\partial \lambda_i}$ {Ensuring $\lambda_i > \epsilon$}
        end for
    end for
end for

3.4.2. Wavenet Implementation. The following pseudo code describes the implementation of the learning algorithm and parameter initialisation, described in Section 3.3, for the wavenet.

The function 'initialise' is a public method in the class 'Wavenet'.

method initialise($Resolution$)
for $i \in 1, \ldots, NumWavelons$ **do**
  Calculate the dyadic position t_i of wavelon within the given Resolution
  Set $\lambda_i = 2^{Resolution}$
  Initialise a wavelon with t_i and λ_i values
end for

The function 'learn' is a public method in the class 'Wavenet'.

method learn($Training\_Data, Iterations, Rate$)
for $j \in 1, \ldots, Iterations$ **do**
  for $k \in 1, \ldots, NumTrainingSamples$ **do**
    for $i \in 1, \ldots, NumWavelons$ **do**
      Adjust Weights w_i by $Rate \times \dfrac{\partial c}{\partial w_i}$
    end for
  end for
end for

3.5. Function Estimation Example

In the following example we will try to estimate a function that is polluted with Gaussian noise. The function we will look at is defined as follows:

$$f(t) = 10 \sin\left(\frac{2\pi t}{32}\right) \quad \text{for } 0 \leq t \leq 64. \tag{3.5.1}$$

This sine wave with 10% added noise is shown in the left-hand plot of Figure 3.5.1. A random sample of 250 points $\{(t_k, f(t_k)) : k = 1, \ldots, 250\}$ from this signal is shown in the right-hand plot.

Figure 3.5.2 shows the output from two different wavelet networks. In each case the 'learning iterations' and 'learning rate' were set to 500 and 0.05 respectively. The left-hand plot shows the estimate produced by a wavelet network consisting of 64 neurons. We can see that it still contains some of the noise. The right-hand plot shows the estimate produced by a 16 wavelon wavelet network. This produces a much better estimate of the true signal. The mean squared error (MSE), between the true and estimate signals, is 28.98 for the first estimate and 19.09 for the second. The reduction in the number of wavelons has removed the high frequency wavelets from the network, so that it will pick up the overall trend of the data better, rather than the noise.

Figure 3.5.3 shows the output from two different wavenets. As before the 'learning iterations' and 'learning rate' were set to 500 and 0.05 respectively. In the first wavenet, the dyadic scale was set to 0, meaning that the wavelons were centred at unit (or 2^0)

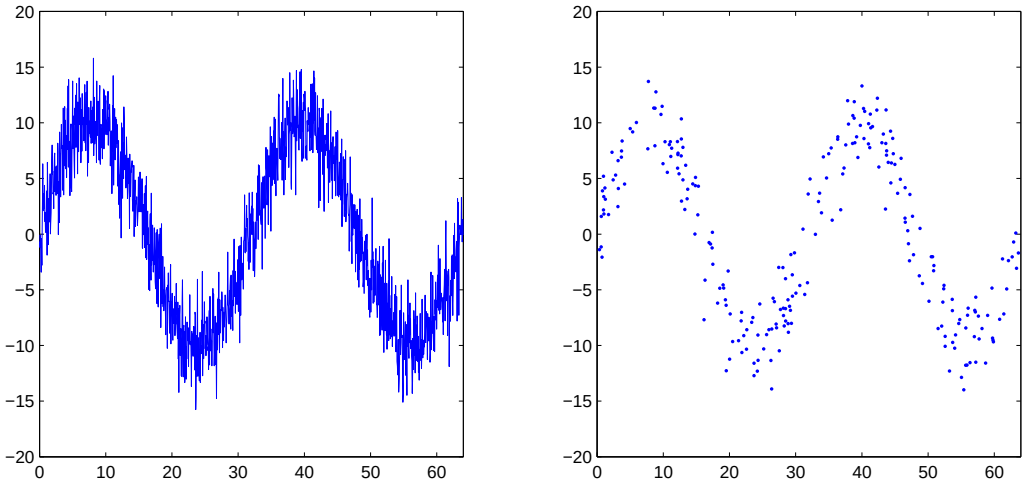

FIGURE 3.5.1. A noisy sine wave (left-hand plot) and a random sample of 250 points from it (right-hand plot).

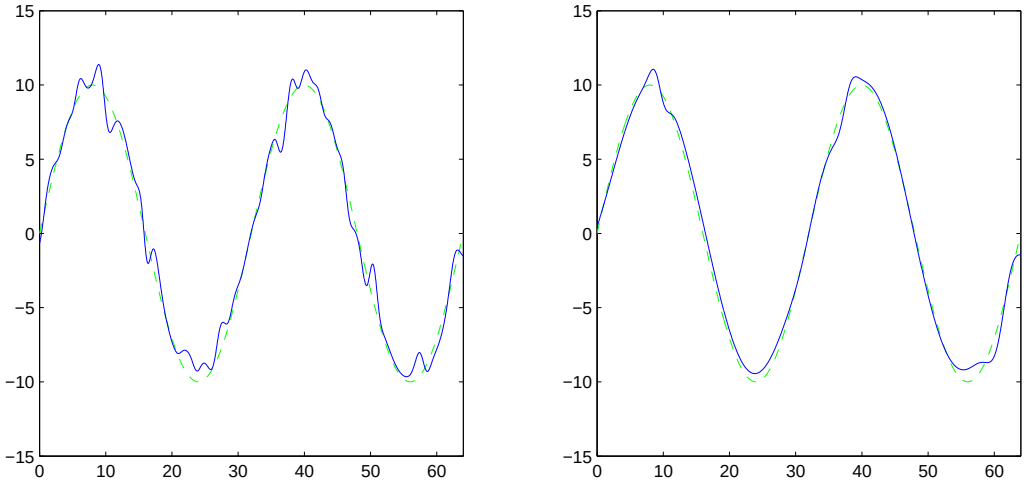

FIGURE 3.5.2. Function estimate using wavelet networks with 'Gaussian' activation functions. The wavelet networks used contained 64 wavelons (left-hand plot) and 16 wavelons (right-hand plot). The underlying function is shown as a dashed line

intervals and with unit dilations. In the second wavenet, the dyadic scale was set to -2, so that the wavelons were centred at $2^2 = 4\times$ unit intervals, with dilations of $2^{-2} = \frac{1}{4}$. We can see that the second wavenet has removed more of the noise, due to it consisting

of lower frequency wavelets. The MSE for the first estimate is 41.37 and for the second estimate is 11.45.

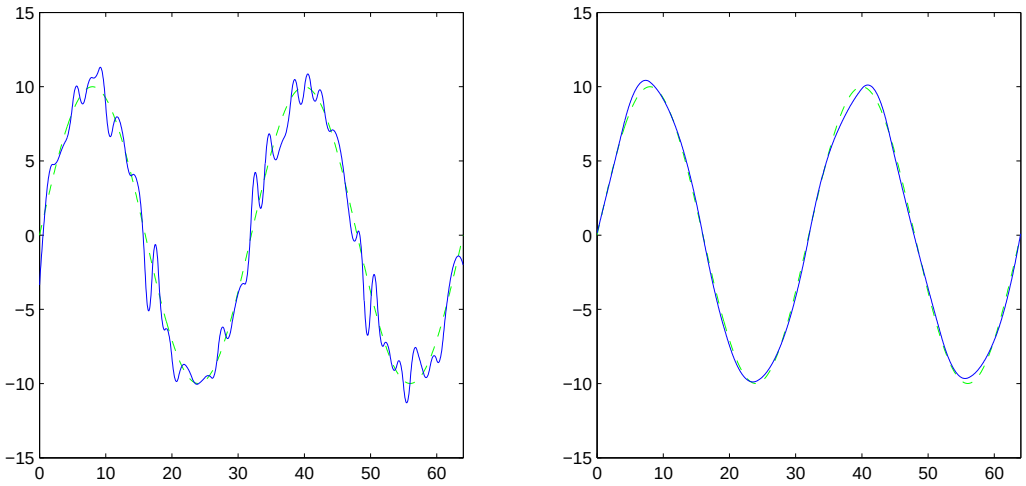

FIGURE 3.5.3. Function estimate using wavenets with 'Battle-Lémarie wavelet' activation functions. The wavenets used contained 67 wavelons (left-hand plot) and 19 wavelons (right-hand plot). The underlying function is shown as a dashed line

For this example the wavenet proved to be more accurate at estimating the true signal. Also, for both networks, the configurations with the fewer wavelons worked better. The wavenet, with its fixed wavelet parameters, is less computationally expensive than the wavelet network, so it seems to be the better choice for function estimation.

3.6. Missing Sample Data

It would be useful for many applications to be able to reconstruct portions of missing data from a sampled time-series. One way of doing this is to use a wavenet to learn from the available data, then to interpolate new data for the missing time values.

Figure 3.6.1 shows this interpolation for the Lorenz 'X' time-series. A portion of 35 data points were missing from the wavenet training data . The dashed line shows the reconstruction from the wavenet function estimate. It is fairly accurate, with the general oscillatory pattern being kept.

Figure 3.6.2 shows the interpolation for a different portion of 35 data points. The dashed line, in this case, does not reconstruct the true values of the time series.

A wavenet is therefore capable in some circumstances to reconstruct a portion of missing data, but in general the reconstruction cannot be relied upon.

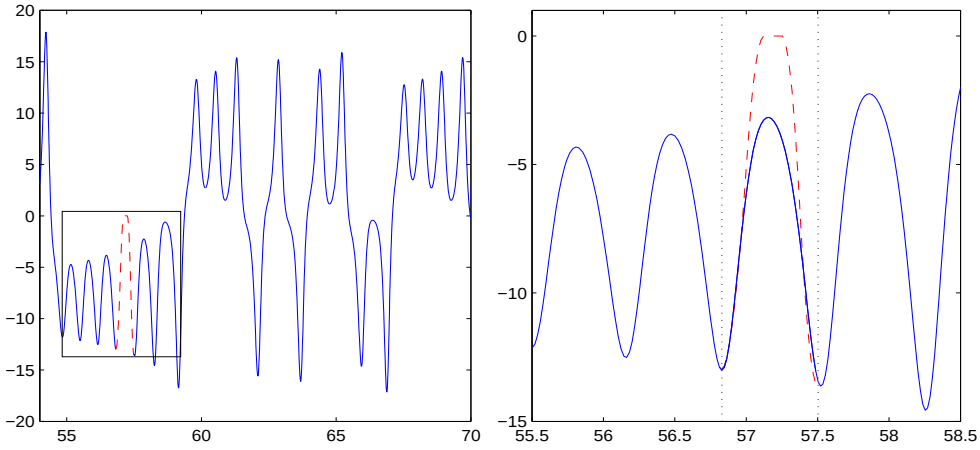

FIGURE 3.6.1. First reconstruction of missing data from the Lorenz 'X' time-series, using a wavenet (right-hand plot is an enlargement of the left-hand plot).

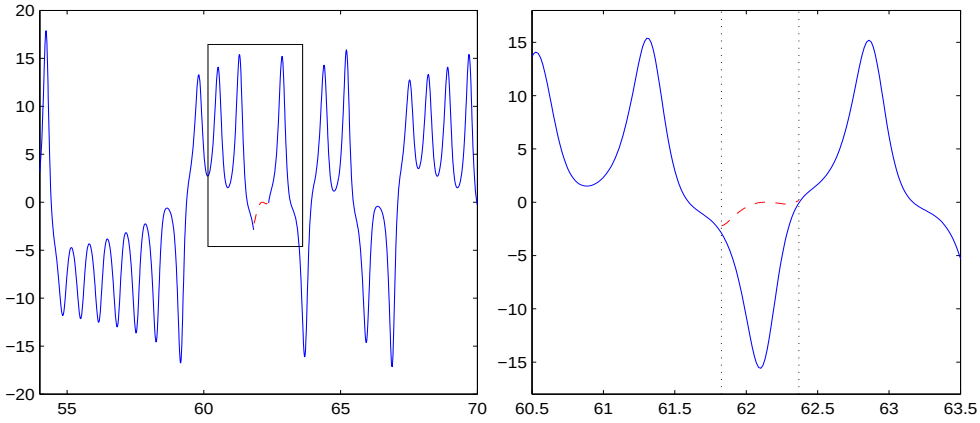

FIGURE 3.6.2. Second reconstruction of missing data from the Lorenz 'X' time-series, using a wavenet (right-hand plot is an enlargement of the left-hand plot).

3.7. Enhanced Prediction using Data Interpolation

For discretely sampled time-series, it is useful to be able to accurately interpolate between the data points to produce more information about the underlying model. In the case of a chaotic model, such as the Lorenz map, if we have more information then we should be able to make more accurate predictions.

We can perform data interpolation using the wavenet. The sampled time-series data is used to train the wavenet to produce a functional estimate of it. Interpolations of the time-series can be produced by passing interpolations of the 'time' values through the trained wavenet.

Testing of various down-sampling intervals $\tau\Delta_0$ for the Lorenz 'X' time-series, numerically integrated at $\Delta_0 = \frac{1}{130}$ time unit intervals, and then up-sampling using a wavenet was performed. It was found that the time-series sampled at intervals below $10\Delta_0$ could be accurately up-sampled again using the wavenet. For intervals above $10\Delta_0$, the wavenet would fail to accurately reconstruct the original time-series where there were any sharp peaks in the data. The wavenet was also able to cope with non-equisampled data, for example with sampling intervals between $\tau_1\Delta_0$ and $\tau_2\Delta_0$, provided again that the maximum interval $\tau_2\Delta_0$ was below $10\Delta_0$.

The middle plot in Figure 3.7.1 shows a down-sampling by $\tau = 6$ of the Lorenz 'X' time-series. Predicted values of this time series (using the prediction algorithm described in Section 3.7.1) are shown for times $t \in [134, 138]$. The left hand plot in Figure 3.7.2 shows the error between this prediction and the actual values of the Lorenz 'X' time-series. We can see that the prediction quickly diverges away from the actual time-series.

The bottom plot in Figure 3.7.1 shows the interpolation of the above time series by $\tau = 6$ using the wavenet function estimate, along with a new prediction for $t \in [134, 138]$. We can see from the right hand plot in Figure 3.7.2, that this new prediction is accurate for alot longer. The prediction has been performed using the same period of the time-series but more information about the nature of the model has been obtained by interpolating the data points.

3.7.1. Prediction using Delay Coordinate Embedding. The file 'predict.m' in Appendix C.2 performs the following delay coordinate embedding prediction algorithm. For given sampled data $\{x_1, x_2, \ldots, x_N\}$, the algorithm will predict the point x_{N+k}.

Algorithm:

- Reconstruct the underlying attractor using 'delay coordinate embedding' (DCE, see Section 3.7.2).

$$\{S_t\}_{t=(\Delta-1)\tau+1}^{N} \ : \ S_t = (x_{t-(\Delta-1)\tau}, \ldots, x_{t-\tau}, x_t)$$

- Neglecting the points $S_{N-k+1}, \ldots, S_{N-2}, S_N$:
  - Calculate ϵ_{min}, the distance of the closest point to S_N.
  - Choose an $\epsilon > \epsilon_{min}$.
  - Find all points within ϵ of S_N and call this set $\tilde{S} = \{S_{t_1}, S_{t_2}, \ldots, S_{t_{n_0}}\}$.
- Find the points k steps along the trajectories starting at each S_{t_i}.

$$\hat{S} = \{S_{t_1+k}, S_{t_2+k}, \ldots, S_{t_{n_0}+k}\}$$

- Then $\hat{x}_{N+k}$, a prediction of x_{N+k}, is the average of the last component of each of these points.

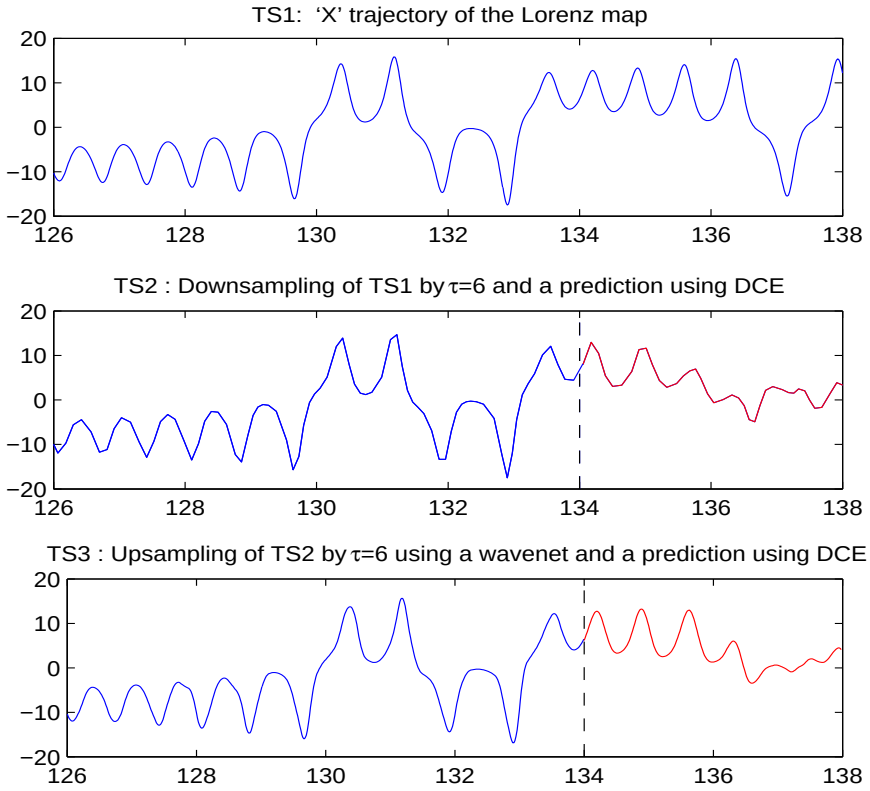

FIGURE 3.7.1. Prediction of the Lorenz map before (TS2) and after (TS3) up-sampling using a wavenet function estimate.

3.7.2. Delay Coordinate Embedding. Given a single finite time-series $\{x_1, x_2, \ldots, x_N\}$, we can reconstruct the systems multidimensional state space, if we make the assumption that the data is stationary. These reconstructed dynamics are topologically equivalent to true dynamics of the system.

Algorithm:

- Given the finite time-series $\{x_1, x_2, \ldots, x_N\}$.
- Let $\tau \in \mathbb{N}$, the delay parameter, and $\Delta \in \mathbb{N}$, the embedding dimension, be fixed.
- Form the (Δ, τ)-embedding by defining the set of Δ-dimension vectors S_t as follows:

$$S_t = (x_{t-(\Delta-1)\tau}, \ldots, x_{t-\tau}, x_t) \quad \text{for } t = (\Delta - 1)\tau + 1, \ldots, N.$$

- This defines an orbit $S_t \in \mathbb{R}^{\Delta}$.

For details about how to find the delay parameter and embedding dimension see Chapters 3 and 9 of [**16**].

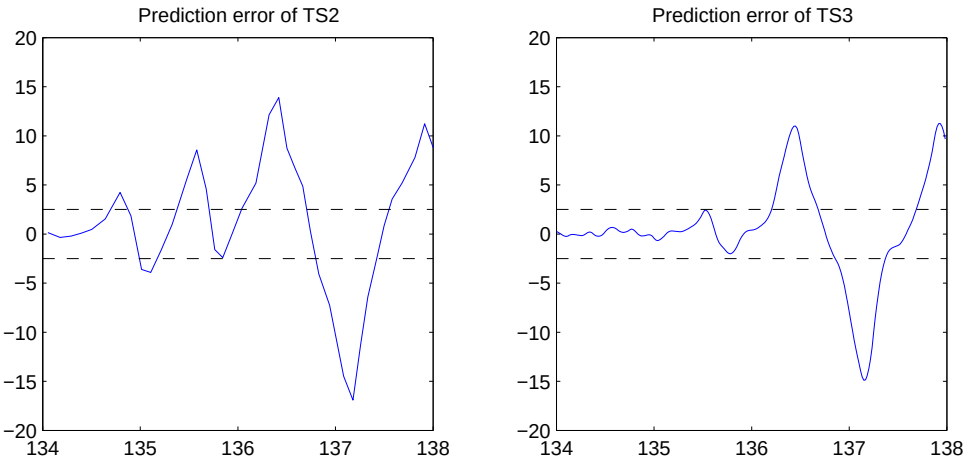

FIGURE 3.7.2. Errors in the predictions of TS2 and TS3 compared with actual values of TS1. A threshold of ±2.5 is shown

3.8. Predicting a Chaotic Time-Series

Given a chaotic time-series $\{x_t\}$, we can reconstruct the underlying attractor of the system using DCE (as described in Section 3.7.2) to give us the Δ-dimensional orbit $\{X_t\}$. Takens' embedding theorem [**18**] then tells us there exists a mapping

$$X_{t+1} = F(X_t) \quad \text{for } t = 1, 2, \ldots \tag{3.8.1}$$

If we denote the attractor by A, then $F : A \to A$. So we only need to estimate the function F in the domain A. We should be able to estimate this function, as already shown, using a wavenet.

Let us consider the one-dimensional case, by trying to predict the Logistic map. Figure 3.8.1 shows the Logistic map, together with a function estimate $\hat{F}$ of the systems attractor. The function $\hat{F}$ was produced using a wavenet, and the training data was taken to be the first 1000 values of the Logistic map for $x_0 = 0.1$.

The next 50 values $\hat{x}_{1001}, \ldots, \hat{x}_{1050}$ were predicted using F and compared with the actual values $x_{1001}, \ldots, x_{1050}$ (shown in Figure 3.8.2).

This is a good result, since the 'sensitive dependence on initial conditions' nature of chaos usually ensures any predictions will diverge away from the actual values very quickly.

This method of prediction can be extended to the multidimensional case, such as for predicting the Lorenz map, by using a multidimensional wavenet. For further information about this see Chapter 8 of [**15**].

3.9. Nonlinear Noise Reduction

A similar application to prediction is noise reduction. Instead of predicting future values of a time-series, we want to predict accurate time-series values from the noisy sampled values. That is, we want to decompose the sampled time-series into two components, one

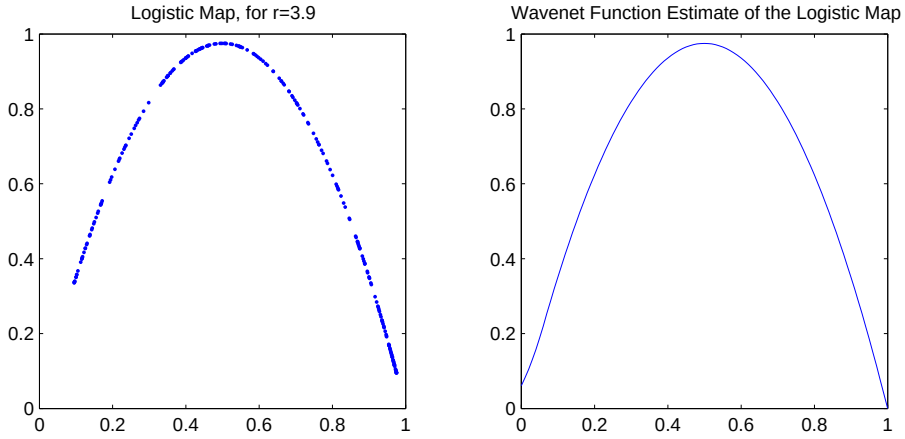

FIGURE 3.8.1. The Logistic map and its wavenet function estimate.

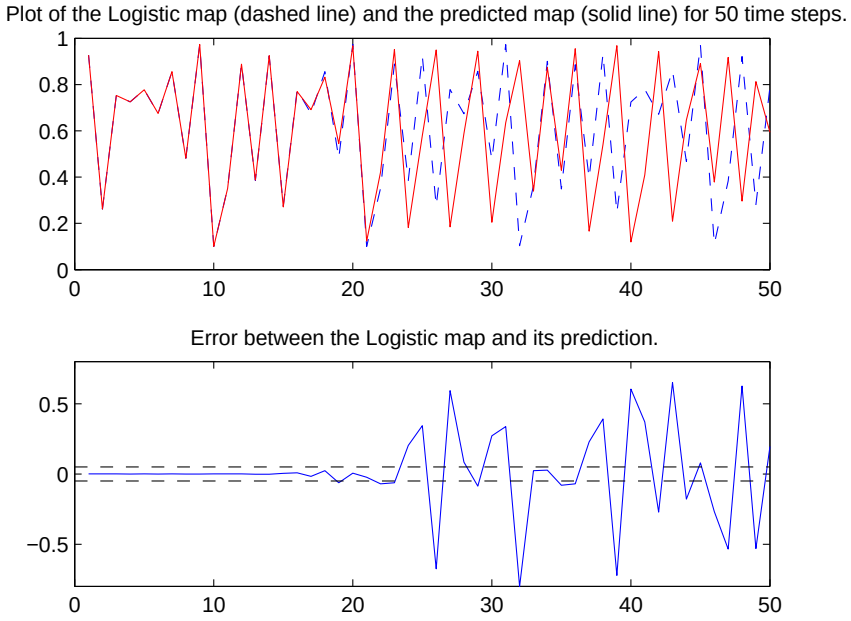

FIGURE 3.8.2. Prediction of the Logistic map using a wavenet.

containing the true values and the other containing the noise contamination. The normal way to do this is by analysing the power spectrum. Random noise has a broad spectrum, whereas periodic or quasi-periodic signals have sharp peaks in the power spectrum, making them easy to distinguish. This approach fails for a deterministic signal, as that too will have a broad spectrum and be indistinguishable from the noise.

If we look at a chaotic deterministic time-series $\{x_n\}$, defined by a deterministic mapping F, then a sampled time series $\{s_n\}$ will be defined in the following way:

$$s_n = x_n + \epsilon_n \quad \text{where} \quad \begin{cases} x_n = F(x_{n-1}, \ldots, x_{n-1-(\Delta-1)\tau}) \\ \epsilon_n \text{ is the random noise} \end{cases} \tag{3.9.1}$$

If we concentrate on the one-dimensional case, then we can estimate the function F using a wavenet, as in Section 3.8. Figures 3.9.1 & 3.9.2 show the Logistic map with 1% added noise, along with a function estimate $\hat{F}$ of the true Logistic map. The MSE between $\{s_n\}$ and $\{x_n\}$ is 2.54 for this example.

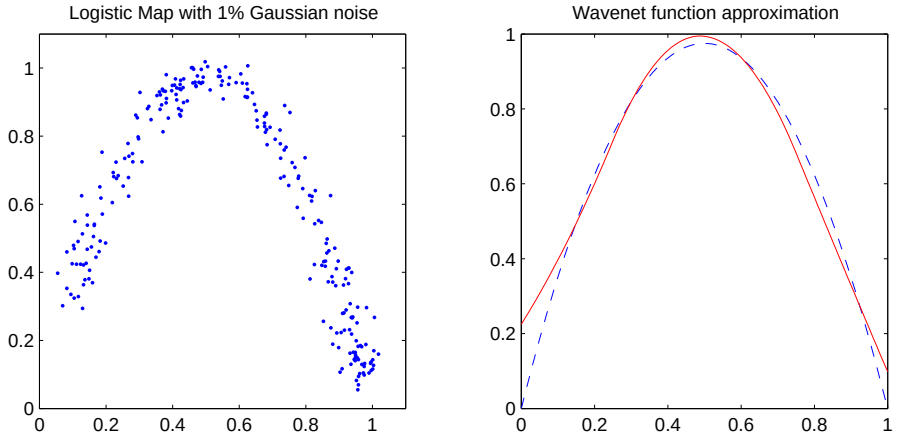

FIGURE 3.9.1. Wavenet function approximation to the Logistic map with 1% Gaussian noise added

This function estimate $\hat{F}$ seems to be all we need to remove the noise from the data, i.e. setting $\hat{x}_{n+1} = \hat{F}(s_n)$ for $n = 1, 2, \ldots, N-1$. If we do this, then we are predicting from noisy values, and the chaotic nature of the system means that these predictions will be further away from the true values. So we need an alternative method for deterministic systems.

We cannot hope to remove all of the noise, so want to construct a new time-series with reduced noise of the form

$$\hat{x}_{n+1} = \hat{F}(\hat{x}_n) + \epsilon'_{n+1} \quad \text{for } n = 1, \ldots, N-1 \tag{3.9.2}$$

where the ϵ'_n is the remaining noise in the system.

We can do this by minimising the MSE in our constructed time-series:

$$e^2 = \sum_{n=1}^{N-1} (\hat{x}_{n+1} - \hat{F}(\hat{x}_n))^2 \tag{3.9.3}$$

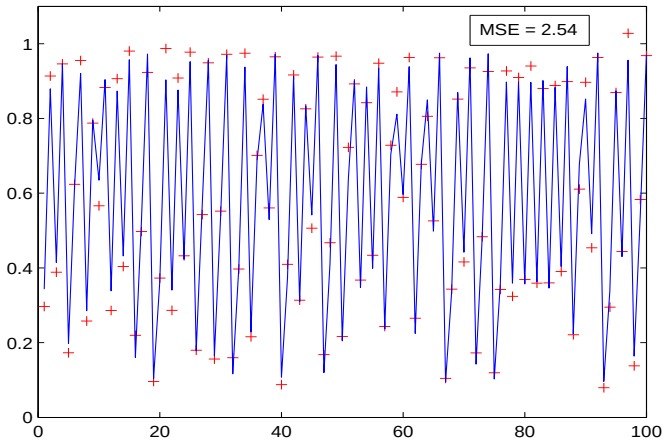

FIGURE 3.9.2. Time series output of the Logistic map $\{x_n\}$ (solid line) and the Logistic map with 1% added Gaussian noise $\{s_n\}$ ('+' points)

The simplest way to solve this problem numerically is by gradient descent

$$\begin{aligned}\hat{x}_n &= s_n - \frac{\alpha}{2}\frac{\partial e^2}{\partial s_n} \\ &= (1-\alpha)s_n + \alpha(\hat{F}(s_{n-1}) + \hat{F}'(s_n)(s_{n+1} - \hat{F}(s_n)) \end{aligned} \tag{3.9.4}$$

where the step size α is a small constant. The two end points are set as follows

$$\begin{aligned}\hat{x}_1 &= s_1 + \hat{F}'(s_1)(s_2 - \hat{F}(s_1)) \\ \hat{x}_N &= \hat{F}(s_{N-1})\end{aligned}$$

This procedure can be performed iteratively, estimating new values $\{\hat{\hat{x}}_n\}$ from the previous estimates $\{\hat{x}_n\}$. Information from the past and future, relative to the point x_n, is used by this algorithm, removing the worry about divergence of near trajectories in a chaotic system.

Our example with the Logistic map uses a wavenet function estimate for $\hat{F}$. In general, this function will not be differentiable (for example, there are no explicit formulae for the Daubechies wavelets), but we can use the following, from the fundamental theorem of calculus,

$$f'(x) = lim_{h\to 0}\frac{f(x+h) - f(x)}{h}$$

and select a sufficiently small value of h to calculate an estimate of $\hat{F}'$ (Note, the derivative would not make sense if we were to use the Haar wavelet, since it is not continuous).

The Matlab file 'noisereduction.m' (in Appendix C.3) performs this gradient descent algorithm. Figure 3.9.3 shows the time-series $\{\hat{x}_n\}$ along with the true time-series $\{x_n\}$. The MSE is now down by almost 50% to 1.29.

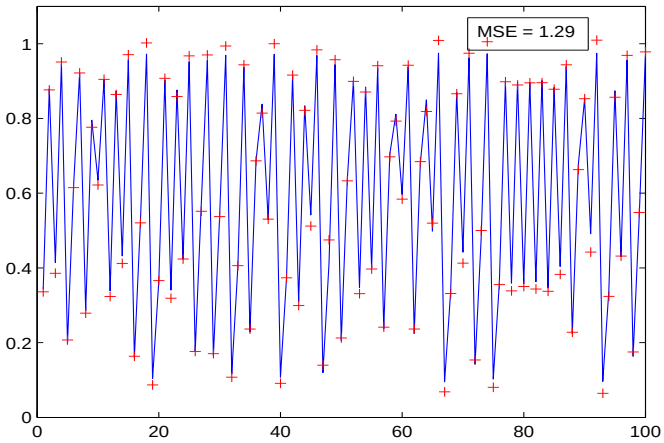

FIGURE 3.9.3. Logistic map time-series (solid line) and its estimate ('+') from noisy data using nonlinear noise reduction

For further reading on nonlinear noise reduction see Chapter 10 of [**16**].

3.10. Discussion

We have shown, through the implementation in the one-dimensional case, that the wavelet neural network is a very good method for approximating an unknown function. It has also been shown to be robust against noise of up to 10% for stationary signals and up to 1% for deterministic chaotic time series.

In the application of dynamical systems this has enabled us to accurately predict and remove noise from chaotic time series.

In the case of prediction, we have shown an improvement in the prediction capabilities of the delay coordinate embedding algorithm (Section 3.7) by interpolating new time series values between the observed values using our wavelet neural network. The wavelet neural network has also been able to accurately predict the Logistic map by estimating the underlying attractor of the system (Section 3.8), the prediction was very close for a relatively long period of time given that the mapping was chaotic.

When applied to nonlinear noise removal the wavelet neural network accurately estimated the underlying attractor of the Logistic map, given only noisy data points, enabling the noise reduction algorithm to be performed successfully.

These wavelet neural network techniques can be extended to the multidimensional case, as described in Section 3.2.2. This would enable the prediction and noise removal for the Lorenz system under discrete sampling, by estimating the function describing the Lorenz attractor for which, unlike the Logistic map, the true function is not known.

APPENDIX A

Wavelets - Matlab Source Code

A.1. The Discrete Wavelet Transform using the Haar Wavelet

A.1.1. haardwt.m.

```
%*************************************************************
%                       File : haardwt.m                     *
%*************************************************************
% Calculates the DWT using the haar wavelet, to the          *
% specified number of 'levels'.                              *
% X is the input signal, if its length is not a power of 2   *
% then it is padded out with zeros until it is.              *
%                                                            *
% The wavelet coefficients are plotted for levels 1 to       *
% 'levels'.                                                  *
% The scaling coefficients for the final level are plotted.  *
%*************************************************************
%                  Author : David C Veitch                   *
%*************************************************************

function [W, S ] = haardwt( X, levels )

  % The length of the input signal must be a power of 2.
  % If it isn't then pad it out with zeros.
  N = length(X);
  A = log2(N);
  B = int32(A);
  if (A ~= B)
    if (A > B)
      B = B+1;
    end;
    X(N+1:2^B)= 0;
    disp('length of signal is not a power of 2!');
    N = length(X);
  end;

  % Wavelet Coefficients
  W = zeros(levels,N/2);
  % Scaling Coefficients
  S = zeros(1,N/(2^levels));
  S_tmp = zeros(levels+1,N);
  S_tmp(1,:) = X;

  % Initialise the output plots.
  hold on;
  suptitle('Discrete Wavelet Transform using the Haar Wavelet');
  subplot(levels+2,1,levels+2);

  % Plot the original signal 'X'.
  plot(X);
  set(gca,'XLim',[0 N]);
  set(gca,'XTick',[0:N/8:N]);
  set(gca,'YLabel',text('String',{'Original';'Signal'}));

  % Plot the wavelet coefficients up to scales 1 to 'levels'.
  for j=1:levels
    N_j= N/2^j;
    % Perform the dwt using the haar wavelet.
    [W(j,1:N_j) S_tmp(j+1,1:N_j)] = haar(S_tmp(j,1:2*N_j));
    % Calculate the times associated with the new
    % wavelet coefficients.
    t = 2^(j-1)-1/2:2^j:(2*N_j - 1)*2^(j-1)-1/2;
    subplot(levels+2,1,j);
    plot(t ,W(j,1:N_j));
    set(gca,'XLim',[0,N]);
    set(gca,'XTickLabel',[]);
```

```
    set(gca,'YLabel',text('String',{'Level';j}));
  end;

  S = S_tmp(levels+1, 1:N_j);

  % Plot the remaining scaling coefficients
  subplot(levels+2,1,levels+1);
  plot(t,S);
  set(gca,'XLim',[0,N]);
  set(gca,'XTickLabel',[]);
  set(gca,'YLabel',text('String','Approx'));
```

A.1.2. haar.m.

```
%************************************************************
%                      File : haar.m                        *
%************************************************************
% Outputs the wavelet and scaling coefficients for one level *
% level of the 'Haar' wavelet transform.                     *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [W,S]=haar(signal)

  % W : Wavelet Coefficients
  % S : Scaling Coefficients

  N=length(signal);

  for i=1:N/2
    W(i)=(signal(2*i-1)-signal(2*i))/2;
    S(i)=(signal(2*i-1)+signal(2*i))/2;
  end
```

A.2. The Inverse Discrete Wavelet Transform using the Haar Wavelet

A.2.1. haaridwt.m.

```
%************************************************************
%                    File : haaridwt.m                      *
%************************************************************
% Performs the Inverse DWT using the haar wavelet, on the    *
% given wavelet and scaling coefficient matrices.            *
% The IDWT is performed for as many iterations needed to     *
% return the original signal.                                *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [ signal ] = haaridwt( W, S )

  % The row dimension 'r' of the matrix 'W' specifies the number
  % of wavelet transformations applied to the original signal.
  [r c]=size(W);
  N = length(S);
  signal = zeros(1,2*c);
  signal(1:N) = S;

  % Perform the inverse transform 'r' times to reconstruct the
  % original signal.
  for i=1:r
    signal = haarinv(W(r+1-i,1:N),signal(1:N));
    N = N*2;
  end;
```

A.2.2. haarinv.m.

```
%************************************************************
%                     File : haarinv.m                      *
%************************************************************
% Outputs the original signal, given the wavelet and scaling *
% coefficients of the 'Haar' wavelet transform.              *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [signal]=haarinv(W, S)

  % W : Wavelet Coefficients
  % S : Scaling Coefficients

  N=length(W);

  for i=1:N
    signal(2*i-1)=S(i) + W(i);
```

```
   signal(2*i)  =S(i) - W(i);
 end;
```

A.3. Normalised Partial Energy Sequence

A.3.1. npes.m.

```
%************************************************************
%                      File : npes.m                        *
%************************************************************
% Function to calculate the normalised power energy sequence *
% for the given signal 'X'.                                  *
%************************************************************
%                   Author : David C Veitch                 *
%************************************************************

function [ C ] = npes( X )

  N = length(X);

  O = zeros(1,N);
  C = zeros(1,N);

  % Form the squared magnitudes and order them.
  O = abs(X).^2;
  O = sort(O,'descend');

  % Calculate the NPES for each element.
  O_sum = sum(O);
  for i=1:N
    C(i) = sum(O(1:i)) / O_sum;
  end;
```

A.3.2. odft.m.

```
%************************************************************
%                      File : odft.m                        *
%************************************************************
% Function to perform the orthonormal discrete fourier       *
% transform.                                                 *
%************************************************************
%                   Author : David C Veitch                 *
%************************************************************

function [ F ] = odft( X )

  N = length(X);

  F = zeros(1,N);

  for k=0:N-1
    F(k+1) = X(1:N)* exp((0:N-1).*(-i*2*pi*k/N))'/sqrt(N);
  end;
```

A.4. Thresholding Signal Estimation

A.4.1. threshold.m.

```
%************************************************************
%                    File : threshold.m                     *
%************************************************************
% Function to perform signal estimation via thresholding     *
% using the wavelet orthogonal transform.                    *
% Inputs : Observed signal 'X'.                              *
%          Level to perform transform to 'j0'.               *
%          Wavelet transform matrix.                         *
% Output : The thresholded and inverse tranformed signal.    *
%************************************************************
%                   Author : David C Veitch                 *
%************************************************************

function [ X_t ] = threshold( X, j0, waveletmatrix )

  N = length(X);
  % length of signal must be a power of 2 to perform
  % the discrete wavelet transform
  A = log2(N);
  B = int32(A);
  if (A ~= B)
    if (A > B)
      B = B+1;
    end;
    X(N+1:2^B)= 0;
    disp('length of signal is not a power of 2!');
    N = length(X);
  end;
```

```
D = X;
X_t = zeros(1,N);

%Perform the DWT up to level 'j0'
for j = 1:j0
  N_j = N/(2^(j-1));
  W = waveletmatrix(N_j);
  D(1:N_j) = D(1:N_j)*W';
end;

% Copy the scaling coefficients directly.
% They are not subject to thresholding.
X_t(1:N_j/2) = D(1:N_j/2);

% Variance is unknown, so estimate it using
% a variance of the 'MAD' method.
var = median(abs(D(N/2+1:N)))/0.6745

% Calculate the threshold level
delta = sqrt(2*var*log(N))

% Perform hard thresholding on the transformed signal
for i = N_j/2+1:N
  if abs(D(i)) <= delta
    X_t(i) = 0;
  else
    X_t(i) = D(i);
  end;
end;

%Perform the IDWT
for j = 1:j0
  N_j = N/(2^(j0-j));
  W = waveletmatrix(N_j);
  X_t(1:N_j) = X_t(1:N_j)*W;
end;
```

A.4.2. d4matrix.m.

```
%************************************************************
%                   File : d4matrix.m                       *
%************************************************************
% Function to produce the wavelet matrix to dimension 'N'    *
% for the Daubechies D(4) wavelet.                           *
%************************************************************
%                   Author : David C Veitch                 *
%************************************************************

function [ W ] = d4matrix( N )

  if N<4
    disp('error:  matrix dimension is too small');
    return;
  else if mod(N,2) ~= 0
      disp('error:  matrix dimension must be even');
      return;
    end;
  end;

  % Set the Scaling function coefficients
  h = [(1+sqrt(3)),( 3+sqrt(3)),
       (3-sqrt(3)),( 1-sqrt(3))]./(4*sqrt(2));
  % Set the Wavelet function coefficients
  g = [(1-sqrt(3)),(-3+sqrt(3)),
       (3+sqrt(3)),(-1-sqrt(3))]./(4*sqrt(2));

  % Set the Transform matrix
  % The top N/2 rows contain the scaling coefficients
  % The bottom N/2 rows contain the wavelet coefficients
  W = zeros(N,N);
  for i = 1:N/2-1
    W(i,    2*i-1:2*i+2) = h;
    W(i+N/2,2*i-1:2*i+2) = g;
  end;
  % Wrap around the coefficients on the final row
  W(N/2,[N-1 N 1 2]) = h;
  W(N,  [N-1 N 1 2]) = g;
```

APPENDIX B

Neural Networks - Java Source Code

B.1. Implementation of the Perceptron Learning Algorithm

B.1.1. PerceptronLearnApplet.java.

```
/************************************************************
 *            File : PerceptronLearnApplet.java             *
 ************************************************************
 * A Java Applet that will attempt to learn any one of the   *
 * 16 binary boolean functions specified by the user.  It    *
 * will then output the specified values from that           *
 * functions truth table                                     *
 ************************************************************
 *                Author : David C Veitch                    *
 ************************************************************/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PerceptronLearnApplet extends JApplet
                                   implements ActionListener
{
  // Possible training inputs
                // x0, x1, x2
  double S[][] = {{1, 1, 1},
                  {1, 1, 0},
                  {1, 0, 1},
                  {1, 0, 0}};
  /* Expected results for inputs S depending upon which
      of the 16 binary boolean functions is chosen*/
  double t[][] = {{0, 0, 0, 0},
    {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1},
    {1, 1, 0, 0}, {1, 0, 1, 0}, {1, 0, 0, 1},
    {0, 1, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1},
    {1, 1, 1, 0}, {1, 1, 0, 1}, {1, 0, 1, 1}, {0, 1, 1, 1},
    {1, 1, 1, 1}};

  /* Names of the 16 binary boolean functions specified
      in the same order as the test data 't' */
  String names[] = {"FALSE",
                    "AND", "x ^ y", "x ^ y", "x ^ y",
                    "x", "y", "XNOR",
                    "XOR", "y", "x",
                    "x v y", "x v y", "x v y", "NAND",
                    "TRUE"};

  // Boolean input and output values
  String booloptions[] = {"FALSE", "TRUE"};

  // Synaptic weights 'w', input vector 'x' and output 'y'
  double  w[] = {0, 0, 0},
          x[] = {1, 0, 0},
          y = 0;

  // McCulloch-Pitts Neuron
  McCullochPittsNeuron perceptron =
      new McCullochPittsNeuron(3, w);

  // Display objects for the applet
  JLabel outlabel, title, xlabel, ylabel, implieslabel;
  JTextField outtext;
  JButton runbutton;
  JComboBox boolExp, xbox, ybox;
```

```
int converged = 1;

public void init()
{
  Container container = getContentPane();
  SpringLayout layout = new SpringLayout();
  container.setLayout(layout);

  // Initialise the display object
  title = new JLabel("Perceptron Learning of a Binary
                      Boolean Function");
  // Add it to the display
  container.add(title);
  // Set the positioning of the object
  layout.putConstraint(SpringLayout.WEST, title, 10,
                       SpringLayout.WEST, container);
  layout.putConstraint(SpringLayout.NORTH, title, 10,
                       SpringLayout.NORTH, container);

  boolExp = new JComboBox(names);
  container.add(boolExp);
  layout.putConstraint(SpringLayout.WEST, boolExp, 110,
                       SpringLayout.WEST, container);
  layout.putConstraint(SpringLayout.NORTH, boolExp, 10,
                       SpringLayout.SOUTH, title);

  xlabel = new JLabel("X =");
  container.add(xlabel);
  layout.putConstraint(SpringLayout.WEST, xlabel, 40,
                       SpringLayout.WEST, container);
  layout.putConstraiNt(SpringLayout.NORTH, xlabel, 15,
                       SpringLayout.SOUTH, boolExp);

  xbox = new JComboBox(booloptions);
  container.add(xbox);
  layout.putConstraint(SpringLayout.WEST, xbox, 5,
                       SpringLayout.EAST, xlabel);
  layout.putConstraint(SpringLayout.NORTH, xbox, 10,
                       SpringLayout.SOUTH, boolExp);

  ylabel = new JLabel("Y =");
  container.add(ylabel);
  layout.putConstraint(SpringLayout.WEST, ylabel, 20,
                       SpringLayout.EAST, xbox);
  layout.putConstraint(SpringLayout.NORTH, ylabel, 15,
                       SpringLayout.SOUTH, boolExp);
  ybox = new JComboBox(booloptions);

  container.add(ybox);
  layout.putConstraint(SpringLayout.WEST, ybox, 5,
                       SpringLayout.EAST, ylabel);
  layout.putConstraint(SpringLayout.NORTH, ybox, 10,
                       SpringLayout.SOUTH, boolExp);
  implieslabel = new JLabel("=>");

  runbutton = new JButton ("Run Perceptron");
  runbutton.addActionListener(this);
  container.add(runbutton);
  layout.putConstraint(SpringLayout.NORTH, runbutton, 10,
                       SpringLayout.SOUTH, xlabel);
  layout.putConstraint(SpringLayout.WEST, runbutton, 85,
                       SpringLayout.WEST, container);

  outlabel = new JLabel("Result");
  container.add(outlabel);
  layout.putConstraint(SpringLayout.WEST, outlabel, 75,
                       SpringLayout.WEST, container);
  layout.putConstraint(SpringLayout.NORTH, outlabel, 15,
                       SpringLayout.SOUTH, runbutton);

  outtext = new JTextField(5);
  outtext.setEditable(false);
  container.add(outtext);
  layout.putConstraint(SpringLayout.WEST, outtext, 5,
                       SpringLayout.EAST, outlabel);
  layout.putConstraint(SpringLayout.NORTH, outtext, 13,
                       SpringLayout.SOUTH, runbutton);
}

// Run this when the ‘runbutton’ has been pressed
public void actionPerformed(ActionEvent actionEvent)
{
```

```
    /* Perform the perceptron learning algorithm on the
        binary boolean function specified by the user */
    perceptron.perceptronLearn(
        S, t[boolExp.getSelectedIndex()], 0.01, 25);

    // Get the input vector 'x' from the screen
    x[1] = xbox.getSelectedIndex();
    x[2] = ybox.getSelectedIndex();

    //  Run the McCulloch-Pitts neuron
    y = perceptron.runNeuronHeaviside(x);

    // Output the boolean result to the screen
    outtext.setText(booloptions[(int) y]);
  }

}
```

B.1.2. McCullochPittsNeuron.java.

```
/*************************************************************
 *           File : McCullochPittsNeuron.java                *
 *************************************************************
 * A Class representing the McCulloch-Pitts Neuron.          *
 * Methods to initialise, change synaptic weights run the    *
 * neuron and perform a perceptron learning algorithm.       *
 *************************************************************
 *               Author : David C Veitch                     *
 *************************************************************/

public class McCullochPittsNeuron {
  // # of inputs to the neuron (including the bias)
  private int m;
  // Synaptic weights of the neuron
  private double w[];

  public McCullochPittsNeuron (int m, double w[]) {
    this.m = m;
    this.w = w;
  }

  public void changeWeights (double w[]) {
    this.w = w;
  }

  public double runNeuronHeaviside (double x[]) {
    int i;
    double v = 0.0,
           y = 0.0;

    /* Perform the dot-product between the synaptic
     * weights 'w' and the input vector 'x' */
    for (i=0; i<this.m; i++) {
      v += this.w[i]*x[i];
    }

    // Heaviside Step Function
    if (v >= 0) {
      y = 1.0;
    }

    return y;
  }

  public void perceptronLearn(double S[][], double t[],
                              double lconst, double runtimes)
  {
    double wPrime[] = new double[m+1];
    int i, j, k;
    double h;
    int possOutputs = (m-1)*(m-1);

    //  lconst must be positive
    if (lconst <= 0)
        lconst = 1;

    for(k=0; k<runtimes; k++){
      /* for each of the possible training outputs,
       * assuming boolean values */
      for(j=0; j<possOutputs; j++){
        /* Run the neuron with the current synaptic
         * weight values */
        h = this.runNeuronHeaviside(S[j]);
        for(i=0; i<this.m; i++){
```

```
          /* adjust the synaptic weights in proportion
           * to the error between the current output
           * 'h' and the expected training output 't' */
          this.w[i] += lconst*(t[j] - h)*S[j][i];
        }
      }
    }
  }
}
```

APPENDIX C

Wavelet Neural Networks - Source Code

C.1. Function Approximation using a Wavelet Neural Network

C.1.1. FunctionApproximator.java.

```
/************************************************************
 *              File : FunctionApproximator.java            *
 ************************************************************
 * A Java Program that will approximate an unknown          *
 * function from a series of sample inputs and outputs for  *
 * that function, read from the file 'training.txt'.        *
 * It achieves this via wavelet network learning.           *
 * It outputs the wavelet coefficients to the file          *
 * 'coeffs.txt'.                                            *
 ************************************************************
 *                 Author : David C Veitch                  *
 ************************************************************/

package waveletNN;

import java.io.*;
import java.util.StringTokenizer;
import java.lang.Math;
import javax.swing.*;

public class FunctionApproximator {

  public static void main(String[] args) throws IOException {

    int dyadic, N, M, wavelons;
    int S = 300;
    double gamma;

    double[][] data;
    int samples = 0;
    double domain_low = 0, domain_high = 0;
    int t_low, t_high;
    String line;
    StringTokenizer tokens;
    String[] buttons = {"Wavelet Network", "Dyadic Wavenet"};
    BufferedReader fileInput =
      new BufferedReader(new FileReader("training.txt"));

    /* Read the number of training samples from the first
     * line of file 'training.txt' */
    if ((line = fileInput.readLine()) != null){
      S = Integer.parseInt(line);
    }
    else{
      JOptionPane.showMessageDialog(null,
          "Error reading the number of training samples",
          "File Read Error",
          JOptionPane.ERROR_MESSAGE);
      System.exit(-1);
    }

    data = new double[S][2];

    /* Read the file 'training.txt',
     * for a maximum of 'S' training samples. */
    while ((line = fileInput.readLine()) != null
            && samples < S){
      tokens = new StringTokenizer(line, " ");
      /* Each line is of the form 'u_k f(u_k)' */
      if (tokens.countTokens() != 2){
        JOptionPane.showMessageDialog(null,
            "Error on line " + (samples+1),
```

```
        "File Read Error",
        JOptionPane.ERROR_MESSAGE);
    System.exit(-1);
  }
  /* The first value will be the sample input 'u_k' */
  data[samples][0] = Double.parseDouble(tokens.nextToken());
  /* The second value will be the sample output 'f(u_k)' */
  data[samples][1] = Double.parseDouble(tokens.nextToken());
  /* Initialise the domain ranges from the first lines
   * values. */
  if (samples == 0){
    domain_low = data[samples][0];
    domain_high = data[samples][0];
  }
  else
  /* If necessary, adjust the domain of the function to
   * be estimated. */
  if (data[samples][0] < domain_low) {
    domain_low = data[samples][0];
  }
  else if (data[samples][0] > domain_high) {
    domain_high = data[samples][0];
  }

  samples++;
}

/* Prompt the user for the type of network to use. */
dyadic = JOptionPane.showOptionDialog(null,
              "Select the type of WNN",
              "WNN Selection",
              JOptionPane.DEFAULT_OPTION,
              JOptionPane.QUESTION_MESSAGE,
              null,
              buttons,
              buttons[0]);

/* Prompt the user for the following learning constants */
N = Integer.parseInt(JOptionPane.showInputDialog(
        "Enter the number of learning iterations:"));
gamma = Double.parseDouble(JOptionPane.showInputDialog(
        "Enter the learning rate:"));

if (dyadic == 1){
  M = Integer.parseInt(JOptionPane.showInputDialog(
      "Enter the dyadic resolution:"));
  /* Calculate the range of the wavelet centres covering a
   * neighbourhood of the sample domain */
  t_low = (int)((domain_low-1)*Math.pow(2,M) - 1);
  t_high = (int)((domain_high + 1)*Math.pow(2,M) + 1);
  /* Instantiate the wavenet */
  Wavenet wnn = new Wavenet(t_low, t_high);
  /* Initialise the wavenet for the given resolution 'M' */
  wnn.initialise(M);
  /* Perform the learning of the sampled data */
  wnn.learn(data, samples, N, gamma);
  /* Output the learned wavelet parameters to the
   * specified file */
  wnn.outputParameters("coeffs.txt");
  /* Mark for garbage collection */
  wnn = null;
}
else{
  wavelons = Integer.parseInt(JOptionPane.showInputDialog(
             "Enter the number of wavelons:"));
  /* Instantiate the wavelet network */
  WaveletNet wnn = new WaveletNet(wavelons);
  /* Initialise the wavelet network for the given sample
   * domain */
  wnn.initialise(domain_low, domain_high);
  /* Perform the learning of the samples data */
  wnn.learn(data, samples, N, gamma);
  /* Output the learned wavelet parameters to the
   * specified file */
  wnn.outputParameters("coeffs.txt");
  /* Mark for garbage collection */
  wnn = null;
}

System.gc();
System.exit(0);
}
```

```
}
```

C.1.2. WNN.java.

```
/************************************************************
 *                     File : WNN.java                      *
 ************************************************************
 * Contains the WNN superclass.                             *
 * This implements the methods needed to set and retrieve   *
 * the network weights and wavelet coefficients.            *
 ************************************************************
 *                 Author : David C Veitch                  *
 ************************************************************/

package waveletNN;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import javax.swing.JOptionPane;

public class WNN {
  protected int wavelonCount;

  protected double y_bar = 0.0;
  protected double[] weights;

  protected Wavelon[] wavelons;

  protected int count = 0;

  /* Constructor to set up the network */
  public WNN(int wavelonCount){
    this.wavelonCount = wavelonCount;
    this.wavelons = new Wavelon[wavelonCount];
    this.weights = new double[wavelonCount];
  }

  /* Method to initialise a wavelon,
   * if there is one uninitialised */
  protected void addWavelon(double w, double t, double l){
    if (this.count < this.wavelonCount){
      this.wavelons[this.count] = new Wavelon(t, l);
      this.weights[this.count] = w;
      this.count++;
    }
    else{
      JOptionPane.showMessageDialog(null,
          "Number of wavelons has been exceeded!",
          "Initialisation Error",
          JOptionPane.ERROR_MESSAGE);
      System.exit(-1);
    }
  }

  /* Methods to return the network parameters */
  public double getYBar(){
    return this.y_bar;
  }
  public double[] getWeights(){
    return this.weights;
  }
  public double[] getTranslations(){
    int i;
    double[] trans = new double[this.wavelonCount];
    for(i=0; i<this.wavelonCount; i++){
      trans[i] = this.wavelons[i].getTranslation();
    }
    return trans;
  }
  public double[] getDilations(){
    int i;
    double[] dils = new double[this.wavelonCount];
    for(i=0; i<this.wavelonCount; i++){
      dils[i] = this.wavelons[i].getDilation();
    }
    return dils;
  }

  /* Method to print the network parameters to the
   * specified file */
  public void outputParameters(String filename)
```

```
      throws IOException{
    int i;
    PrintWriter fileOutput =
      new PrintWriter(new FileWriter(filename));
    double[] translations = this.getTranslations();
    double[] dilations = this.getDilations();

    fileOutput.println(this.y_bar);
    for(i=0; i<this.wavelonCount; i++){
      fileOutput.println(this.weights[i] + " "
                         + translations[i] + " "
                         + dilations[i]);
    }
    fileOutput.close();
  }
}
```

C.1.3. WaveletNet.java.

```
/************************************************************
 *                  File : WaveletNet.java                  *
 ************************************************************
 * Contains a Wavelet Network subclass, which extends the   *
 * WNN superclass.                                          *
 * This Wavelet Network adjusts its wavelet coefficients by *
 * learning from a set of training data.                    *
 ************************************************************
 *                  Author : David C Veitch                 *
 ************************************************************/

package waveletNN;

public class WaveletNet extends WNN{

  private double trans_min;
  private double trans_max;
  private double dil_min;

  public WaveletNet(int wavelonCount) {
    super(wavelonCount);
  }

  /* Method to initialise the network */
  public void initialise(double a, double b){
    /* 'n' is the number of complete resolution levels that
     * can be initialised for the given wavelon count. */
    int n = (int) (Math.log(super.wavelonCount)/Math.log(2));
    double t_i, l_i;

    /* Set the range of the translation parameters
     * to be 20% larger than D=[a,b] */
    this.trans_min = a - 0.1*(b-a);
    this.trans_max = b + 0.1*(b-a);
    /* Set the minimum dilation value */
    this.dil_min = 0.01*(b-a);

    /* Initialise the wavelons within the complete
     * resolution levels. */
    this.initComplete(a,b,n);

    /* Initialise the remaining wavelons at random within the
     * highest resolution level. */
    while(super.count < super.wavelonCount)
    {
      t_i = a+(b-a)*Math.random();
      l_i = 0.5*(b-a)*Math.pow(2,-n);

      super.addWavelon(0.0, t_i, l_i);
    }
  }

  /* Recursive method to initialise the wavelons within the
   * complete resolution levels*/
  private void initComplete(double u, double v, int level){
    double t_i = 0.5*(u+v);
    double l_i = 0.5*(v-u);

    super.addWavelon(0.0,t_i,l_i);

    if(level<=1){
      return;
    }
    else{
      this.initComplete(u,  t_i, level-1);
```

```
    this.initComplete(t_i,  v, level-1);
  }
}

/* Method to perform the stochastic gradient learning
 * algorithm on the training data for the given
 * 'learning iterations' and 'learning rate' (gamma)
 * constants. */
public void learn(double[][] training, int samples,
              int iterations, double gamma){
  int i, j, k;
  double sum = 0;
  double u_k, f_u_k, e_k,
         psi, dpsi_du,
         w_i, l_i, t_i,
         dc_dt, dc_dl;

  /* y_bar is set to be the mean of the training data. */
  for(i=0;i<samples;i++){
    sum += training[i][1];
  }
  super.y_bar = sum/samples;

  for(j=0;j<iterations;j++)
  {
    for(k=0;k<samples;k++)
    {
      /* For each training sample, calculate the
       * current 'error' in the network, and then
       * update the network weights according to
       * the stochastic gradient procedure. */
      u_k = training[k][0];
      f_u_k = training[k][1];
      e_k = this.run(u_k) - f_u_k;

      super.y_bar -= gamma * e_k;

      for(i=0;i<super.wavelonCount;i++)
      {
        psi = super.wavelons[i].fireGD(u_k);
        dpsi_du = super.wavelons[i].derivGD(u_k);
        w_i = super.weights[i];
        t_i = super.wavelons[i].getTranslation();
        l_i = super.wavelons[i].getDilation();
        dc_dt = gamma * e_k*w_i*Math.pow(l_i,-1)*dpsi_du;
        dc_dl =
          gamma * e_k*w_i*(u_k-t_i)*Math.pow(l_i,-2)*dpsi_du;

        super.weights[i] -= gamma * e_k*psi;

        /* Apply the constraints to the adjustable parameters*/
        if (t_i + dc_dt < this.trans_min){
          super.wavelons[i].setTranslation(this.trans_min);
        }
        else if (t_i + dc_dt > this.trans_max){
          super.wavelons[i].setTranslation(this.trans_max);
        }
        else{
          super.wavelons[i].setTranslation(t_i + dc_dt);
        }
        if (l_i + dc_dl < this.dil_min){
          super.wavelons[i].setDilation(dil_min);
        }
        else{
          super.wavelons[i].setDilation(l_i + dc_dl);
        }
      }
    }
    System.out.println(j);
  }
}

/* Method to run the wavelet network
 * in its current configuration */
public double run(double input){
  int i;
  double output = super.y_bar;

  for(i=0;i<super.wavelonCount;i++)
  {
    output += super.weights[i]
            * super.wavelons[i].fireGD(input);
```

```
    }

    return output;
  }
}
```

C.1.4. Wavenet.java.

```
/************************************************************
 *                  File : Wavenet.java                     *
 ************************************************************
 * Contains a Wavenet subclass, which extends the WNN       *
 * superclass.                                              *
 * The wavelet coefficients are dyadic for the wavenet.     *
 * The learning algorithm adjusts the network weights only. *
 * The wavelet coefficients are fixed at initialisation.    *
 ************************************************************
 *                Author : David C Veitch                   *
 ************************************************************/

package waveletNN;

public class Wavenet extends WNN{

  private int t_0;
  private int t_K;

  /* Constructor to set the number of wavelons and
   * the ranges of the translation parameter. */
  public Wavenet(int t_low, int t_high){
    /* Number of wavelons equals the number of integers
     * in the interval [t_low, t_high]. */
    super(t_high - t_low + 1);
    this.t_0 = t_low;
    this.t_K = t_high;
  }

  /* Method to initialise the wavelons. */
  public void initialise(int M){
    int t_i;

    for(t_i=this.t_0;t_i<=this.t_K;t_i++){
      super.addWavelon(0.0, t_i, Math.pow(2,M) );
    }
  }
}

/* Method to perform the stochastic gradient learning
 * algorithm on the training data for the given
 * 'learning iterations' and 'learning rate' (gamma)
 * constants. */
public void learn(double[][] training, int samples,
                  int iterations, double gamma){
  int i, j, k;
  double u_k, f_u_k, e_k, phi;

  for(j=0;j<iterations;j++)
  {
    for(k=0;k<samples;k++)
    {
      /* For each training sample, calculate the
       * current 'error' in the network, and then
       * update the network weights according to
       * the stochastic gradient procedure. */
      u_k = training[k][0];
      f_u_k = training[k][1];
      e_k = this.run(u_k) - f_u_k;

      for(i=0;i<super.wavelonCount;i++)
      {
        phi = super.wavelons[i].fireLemarie(u_k);
        /* The normalisation factor from the formula
         * is taken care of within 'psi'. */
        super.weights[i] -= gamma * e_k * phi;
      }
    }
    System.out.println(j);
  }
}

/* Method to run the wavenet in its
 * current configuration */
public double run(double input){
  int i;
  double output = 0.0;
```

```
    for(i=0;i<super.wavelonCount;i++)
    {
      output += super.weights[i]
              * super.wavelons[i].fireLemarie(input);
    }

    return output;
  }
}
```

C.1.5. Wavelon.java.

```
/************************************************************
 *                    File : Wavelon.java                   *
 ************************************************************
 * Contains a 'Wavelon' object.                             *
 * This 'Wavelon' has associated 'translation' and          *
 * 'dilation' parameters.  There are methods to adjust and  *
 * return these parameters.                                 *
 * There are two choices of activation function, the        *
 * 'Gaussian Derivative' or the 'Battle-Lemarie' wavelet.   *
 ************************************************************
 *                 Author : David C Veitch                  *
 ************************************************************/

package waveletNN;

public class Wavelon{

  private double translation;
  private double dilation;

  /* Constructor to initialise the private variables */
  public Wavelon(double translation, double dilation){
    this.translation = translation;
    this.dilation = dilation;
  }

  /* Methods to change the private variables */
  public void setTranslation(double translation){
    this.translation = translation;
  }
public void setDilation(double dilation){
  this.dilation = dilation;
}

/* Methods to return the private variables */
public double getTranslation(){
  return this.translation;
}
public double getDilation(){
  return this.dilation;
}

/* Method to calculate the 'Gaussian Derivative' */
public double fireGD(double input){
  double u = (input - this.translation)/this.dilation;

  return -u * Math.exp( -0.5*Math.pow(u,2));
}

/* Method to calculate the 'Gaussian 2nd Derivative'
 * used by the wavelet network learning algorithm */
public double derivGD(double input){
  double u = (input - this.translation)/this.dilation;

  return Math.exp( -0.5*Math.pow(u,2) )*( Math.pow(u,2) - 1);
}

/* Method to calculate the 'Battle-Lemarie' wavelet */
public double fireLemarie(double input){
  double u = this.dilation * input - this.translation;
  double y = 0.0;

  if (u>=-1 && u<0){
    y = 0.5 * Math.pow(u+1, 2);
  }
  else if (u>=0 && u<1){
    y = 0.75 - Math.pow(u-0.5, 2);
  }
  else if (u>=1 && u<2){
    y = 0.5*Math.pow(u-2, 2);
  }
```

```
    else y = 0.0;

    return Math.pow(this.dilation, 0.5) * y;
  }
}
```

C.1.6. export2file.m.

```
%************************************************************
%                   File : export2file.m                    *
%************************************************************
% Function to export the sampled training data pairs        *
% (x(k), f_u(k)) to the file 'training.txt'.                 *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [ ] = export2file( u, f_u )

  if size(u) ~= size(f_u)
    disp('Error:  Sizes of Input and Output training data do not match');
    return;
  end;

  n = length(u);
  t = zeros(n,2);

  t(:,1) = u;
  t(:,2) = f_u;

  dlmwrite('training.txt', n);
  dlmwrite('training.txt', t, 'delimiter', ' ', '-append');
```

C.1.7. gaussian.m.

```
%************************************************************
%                   File : gaussian.m                       *
%************************************************************
% Outputs the wavelet network function estimate, using the  *
% 'gaussian' wavelet, for the data points in 'u'.           *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [ y ] = gaussian( u, coeffs, mean)

  size(coeffs);
  w = coeffs(:,1);
  t = coeffs(:,2);
  l = coeffs(:,3);

  wavelonCount = length(w);
  m = length(u);

  y = zeros(1,m);

  % For each data point in 'u'.
  for i = 1:m
    % Output the wavelet network estimation of the function
    % using the 'gaussian' wavelet.
    y(i) = mean;
    for j=1:wavelonCount
      x = (u(i) - t(j)) / l(j);
      y(i) = y(i) + w(j) * -x * exp(-0.5*x^2);
    end;
  end;
```

C.1.8. lemarie.m.

```
%************************************************************
%                   File : lemarie.m                        *
%************************************************************
% Outputs the wavenet function estimate, using the          *
% 'Battle-Lemarie' wavelet, for the data points in 'u'.     *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [y] = lemarie(u,coeffs)

  w = coeffs(:,1);
  t = coeffs(:,2);
  l = coeffs(:,3);

  wavelonCount = length(w);
  m = length(u);
```

```
  y = zeros(1,m);

  % For each data point in 'u'.
  for i=1:m
    % Output the wavenet estimation of the function
    % using the 'Battle-Lemarie' wavelet.
    for j=1:wavelonCount
      x = l(j)*u(i) - t(j);
      if x>=-1 && x<0
        y(i) = y(i) + w(j) * 1/2*(x+1)^2;
      else if x>=0 && x<1
          y(i) = y(i) + w(j) * (3/4 - (x-1/2)^2);
        else if x>=1 && x<2
            y(i) = y(i) + w(j) * 1/2*(x-2)^2;
          end;
        end;
      end;
    end;
  end;
```

C.2. Prediction using Delay Coordinate Embedding

C.2.1. predict.m.

```
%*************************************************************
%                    File : predict.m                        *
%*************************************************************
% Input: X - time series input                               *
%        K - number of steps ahead to predict                *
%        tau - delay coordinate                              *
%        delta - embedding dimension                         *
%        alpha - positive increment for 'epsilon min'        *
% Output: Z = {K predicted points of X}                      *
%*************************************************************
%                  Author : David C Veitch                   *
%*************************************************************

function [Z] = predict(X, K, tau, delta, alpha)

  % Perform the delay coordinate embedding.
  N = length(X);
  L = (delta-1)*tau;
  M = N-L;

  S = zeros(N,3);
  for t=L+1:N
    for d = 1 : delta
      S(t-L,d) = X(t - (delta-d)*tau);
    end;
  end;

  Z = zeros(K,1);

  % Stores points close to last element in SX
  I = [];
  ptr = 1;

  % Find the point closest to the last element in S
  [min_pt,eps_min] = dsearchn(S(1:M-K,:),S(M,:));
  disp(eps_min)
  % Choose an epsilon > eps_min
  if delta > 0
    eps = eps_min + alpha;
  else
    disp('delta must be positive!');
  end;

  % Neglecting points (M-K+1) ... M
  % Find all points within eps of S(M)
  for i = 1:M-K
    if sqrt(sum((S(i,:)-S(M,:)).^2)) < eps
      % Store the index of the point
      I = [I,i];
    end;
  end;

  disp(length(I))
  for k = 1:K
    pts = [];
    % Increment points in 'I' by 'k' steps
    for j = 1:length(I)
      pts = [pts ; S(I(j)+k,1) ];
    end;
    % Prediction 'k' steps ahead is average of these points
    Z(k) = mean(pts);
```

```
  end;
```

C.3. Nonlinear Noise Reduction

C.3.1. noisereduction.m.

```
%************************************************************
%                  File : noisereduction.m                  *
%************************************************************
% Iteratively applies the gradient descent algorithm to the  *
% time series 's'.  The mapping 'F' is estimated using the   *
% wavenet with 'Battle-Lemarie' wavelet activation function. *
%************************************************************
%                  Author : David C Veitch                  *
%************************************************************

function [y] = noisereduction( s, alpha, h, w, iter )

  N =length(s);
  y = zeros(size(s));

  for j=1:iter
    % Perform the gradient descent algorithm
    for n=2:N-1
      y(n) = (1-alpha)*s(n) + alpha*(F(s(n-1),w) + Fprime(s(n),h,w)*(s(n+1)-F(s(n),w)));
    end;
    % The first and last values are special cases.
    y(1) = s(1) + Fprime(s(1),h,w)*(s(2)-F(s(1),w));
    y(N) = F(s(N-1),w);
    s = y;
  end;

% The mapping function is the wavenet function estimate
function [v] = F(u,w)
  v = lemarie(u,w);

% Derivative of the wavenet function estimate
function [V] = Fprime(U,h,w)
  V = (F(U+h,w)-F(U,w))/h;
```

Bibliography

1. Q. Zhang & A. Benveniste, *Wavelet Networks*, IEEE Transactions on Neural Networks **3** (1992), no. 6, 889 – 899.
2. T. M. Cover, *Geometric and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition*, IEEE Transactions on Electronic Compututers **14** (1965), 326 – 334.
3. I. Daubechies, *Ten Lectures on Wavelets*, SIAM, 1992.
4. M. Berthold & D. Hand, *Intelligent Data Analysis*, 2nd ed., Springer, 2003.
5. S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., Prentice Hall, 1999.
6. *Function Approximation Capabilities of a RBFN*, http://diwww.epfl.ch/mantra/tutorial/english/rbf/html/.
7. D. L. Donoho & I. M. Johnstone, *Ideal Spatial Adaptation by Wavelet Shrinkage*, Biometrika **81** (1994), no. 3, 425 – 455.
8. H. L. Resnikoff & R. W. Wells Jr., *Wavelet Analysis: The Scalable Structure of Information*, Springer, 1998.
9. A. Jensen & A. la Cour-Harbo, *Ripples in Mathematics: The Discrete Wavelet Transform*, Springer, 2001.
10. S. G. Mallat, *A Theory for Multiresolution Signal Decomposition: The Wavelet Representation*, IEEE Transactions on Pattern Analysis and Machine Intelligence **11** (1989), no. 7, 674 – 693.
11. C. A. Micchelli, *Interpolation of Scattered Data: Distance, Matrices and Conditionally Positive Definite Functions*, Constructive Approximations 2 **2** (1986), no. 1, 11 – 22.
12. D. W. Patterson, *Artificial Neural Networks: Theory and Applications*, Prentice Hall, 1996.
13. W. S. McCulloch & W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, Bulletin of Mathematical Biophysics **5** (1943), 115 – 133.
14. F. Rosenblatt, *Two Theorems of Statistical Separability in the Perceptron*, Mechanisation of Thought Processes **1** (1959), 421 – 456.
15. E. C. Cho & Vir V. Phoha S. Sitharama Iyengar, *Foundations of Wavelet Networks and Applications*, Chapman & Hall/CRC, 2002.
16. H. Kantz & T. Schreiber, *Nonlinear Time Series Analysis*, Cambridge University Press, 1997.
17. *Science/Educational Matlab Database*, http://matlabdb.mathematik.uni-stuttgart.de/.
18. F. Takens, *Detecting Strange Attractors in Turbulence*, Springer Lecture Notes in Mathematics **898** (1981), 366 – 381.
19. D. B. Percival & A. T. Walden, *Wavelet Methods for Time Series Analysis*, Cambridge University Press, 2000.