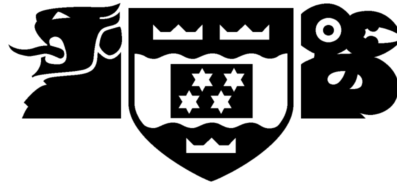
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Gaussian Processes as Neural Networks

Matthew Lilley

Supervisor: Marcus Frean

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

An implementation of a Gaussian process prediction mechanism using biologically plausible processes in the framework of a neural network.

Acknowledgments

I would like to thank my family for their support throughout the year, Marcus for his guidance and help (especially in early October) and finally Ellen, for the love and support that has kept me going throughout the year

Contents

1 Introduction **1**

2 Background **2**
2.1 A Brief Review of Bayesian Inference 2
2.2 Neural Networks 2
2.3 Biological Plausibility 3

3 Gaussian Processes **5**
3.1 A Tool for Regression 5
3.2 Evaluating the mean and variance of the joint distribution 7
3.3 Relation to Forward Feed, Back Propagation Neural Networks 9

4 The Inversion Problem **11**
4.1 Objective Functions for the two Algorithms 11
4.2 Making the Algorithms Covariant 14
4.3 Convergance Speed 14
4.3.1 Relative speed of the two algorithms 14
4.4 Proof of Correctness 15
4.4.1 Speed Relative to Orthodox Matrix Inversion, and the Effect of η . . . 17
4.5 A Space Optimisation 18

5 Biological Background **20**
5.1 Grandmother Cells 20
5.2 Biological Parallels 20
5.3 Shunting Inhibitions and Variance 21

6 Data Recall and Sleeping **22**

7 Faster Predictions: Instinct **23**

8 Determining Hyper-hyper Parameters **24**
8.1 Gaussian Process Hyper-Parameters 24
8.2 Hyper-hyper Parameters 24
8.3 Learning the Hyper-Parameters 25
8.3.1 Orthodox Approach: Optimising Hyper-Parameters 25
8.3.2 Bayesian Approach: Integrating the Hyper-Parameters Out 25
8.4 The Exact Method 25
8.5 The Approximated Likelihood 26
8.6 The Log of the Approximated Likelihood 26
8.7 The Normalisation Problem 27

9 Visualisation **28**
9.1 A One Dimensional Example 28
9.2 Learning The Inverse 28

10 Future Work **35**
10.1 Better Sampling 35
10.2 Reducing the Matrix: Naïve Relevance Determination 35

11 Conclusions **37**

A The Inversion Convergance can be Approximated as Exponential **39**

Figures

3.1 Conditional two dimensional Gaussian distribution 6

4.1 Gibbs Inversion Network 12
4.2 Daugman Inversion Network 12
4.3 Comparison of convergance speed between Gibbs' and Daugman's algorithms 15
4.4 Dependance of Inversion Speed on η 18
4.5 Space Optimised Network 19

9.1 Simple 1d example distribution 29
9.2 1d example distribution with a smaller training set 30
9.3 1d example distribution with a underestimated length scale 31
9.4 1d example distribution with a overestimated length scale 32
9.5 Error in instinct network after n iterations 33
9.6 Magnification of the last 500 iterations of figure 9.5 34

Chapter 1

Introduction

Algorithms employed in computer science can usually be classified into two categories. First, there are algorithms inspired from statistics and mathematics. These attempt to solve goals by employing complex, abstract procedures with a grounding in mathematical theory. These alogrithms are often very good at solving problems which humans have difficulty in performing themselves. However, it is clear that the process which leads to our intellectual thought processes is fundamentally different in that we do not have cells capable of performing complex abstract mathematics. Instead, we have only primitive, and slow hardware which is prone to failure. Yet we are able to discover the very theory that the aforementioned algorithms are derived from.

The second category consists of algorithms inspired by nature. These include neural networks and genetic programming. The principle behind neural networks is that humans are capable of complex thought without requiring operations which are complex at the base level. Therefore, an abstracted neural network, with sufficient parallelism should be able to perform complex tasks.

A very common statistical procedure is inference. Given a set of data, we wish to make predictions about unknown regions in the dataspace. This is done by formulating a hypothesis about the data from the known points, and then inferring unknown points from this hypothesis. The primary difficulty then becomes selecting a hypothesis space which is rich enough to allow us to express prior beliefs in particular models, and has sufficient flexibility to contain models which we believe are acceptable.

Gaussian processes are a means of deriving a prior distribution over an infinite space of functions, so that we may perform Bayesian inference as describe above on a flexible and parameterisable set of hypotheses.

Calculating a prior distribution from a Gaussian process involves biologically intractible mathematics, specifically inverting and multiplying large matrices. However, it has been suggested that this method of regression far outstrips nerual networks on two counts - first, it is considerably faster to *train*, and secondly it has an intrinsic uncertainty model built in. While techniques for obtaining uncertainty from a neural network exist (XXX) they are additions to the architecture, whereas Gaussian processes have uncertainty as a fundamental component.

The work I have undertaken attempts to reconcile the mathematically inspired process of inference using Gaussian processes with the biologically inspired process of neural networks by creating a neural network that employs a Gaussian process to produce predictions from data.

Chapter 2

Background

2.1 A Brief Review of Bayesian Inference

The equation now known as Bayes' theorem is derived from a simple application of the product rule on the joint probability $P(A, B)$. This can be decomposed as either $P(A)P(B|A)$ or as $P(B)P(A|B)$. Because these are equivalent, we may write

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

This can be used to make an inference by letting A represent a hypothesis about the cause of some data, and B being a set of observations about the data. This leads to the establishment of some important nomenclature. If we consider a hypothsis H and a data set D, then we can write

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{2.2}$$

The term $P(H|D)$ is commonly referred to as the *posterior* distribution. $P(D|H)$ is referred to as the *likelihood* of the data given the hypothesis. $P(H)$ is referred to as the *prior* distribution for the hypothsis, and $P(D)$ is called the *evidence* of the data.

The prior distribution reflects the belief that H is correct *even before any data is observed*. Orthodox statisticians regard this as absurd – how is it possible to have a prior *belief* in a hypothesis before you have any idea what kind of surface is being modelled? Bayesian statisticians counter that to do any inference you must make prior assumptions about the surface (affectionately known as the *No Free Lunch theorem*), and Bayesian inference is preferable to orthodox inference because these assumptions are made explicit.

As the data set becomes larger, the effect of the prior becomes weaker. In the limit where we have data for every point we are attempting to infer, the inference depends only on the data, and not the prior.

2.2 Neural Networks

Neural networks evolved from the realisation that the human brain is very good at solving problems that computers are very poor at solving, despite being many orders of magnitude faster, more complex, and more precise (albeit less parallel). The brain is comprised primarily of richly interconnected cells called neurons. Each neuron is comprised of a *soma* (the body) and an *axon* – a long, thin organelle which connects to other neurons by a region called a *synapse*.

Electrical impulses travel along the axons and are acculumated at the soma. Once an intrinsic threshold is reached, the soma releases the accumulated charge in a single pulse, causing the synapse to release a *neurotransmitter* into the synaptic gap. This in turn causes an excitation in the neurons connected to the receptor on the synaptic gap. This is referred to as the *integrate and fire* model for neural activity. While this is one of the most common models for neural behaviour, there are other models, such as voltage dependent models. [14] See also section 5.3.

Hebbian learning is the (at least partly) theoretical process under which synaptic strength is adjusted based on the difference in charge between two neurons. This is considered the primary mechanism behind memory and learning in biological neural systems.

The abstraction of this concept is a neural network. A neural network is a directed graph, where the vertices represent somata and the edges represent synapses. Each edge is associated with a weight which models the varying connectivity that a neuron can have with a neighbour.

While there are many kinds of neural networks (single layer, recursive and temporal are three commonly employed to solve problems in computers), the most common kind is multilayer forward-feed, backward-propagate networks. Designed to overcome the lack of richness arising from the low complexity in a single layer network, multi-layer networks are able to model complex smooth surfaces. They traditionally comprise of three layers – the input layer, the hidden layer and the output layer. Every vertex is connected to all the vertices in the layer below it, and the edges are initially weighted randomly.

Data is fed into the input neurons, which 'fire' producing an input to the hidden layer neurons. These in turn 'fire', outputting a function of the weighted sum of their inputs, and the final result is read out at the output neuron(s). In the training phase, the output is compared with the known value, and the weights are adjusted incrementally by propagating the error backwards through the network (hence back-progagation).

The weights for the input-hidden connection are calculated after each iteration as follows:

$$w_{ij} = w_{ij} + \alpha \times input_i \times \frac{\partial \sigma}{\partial x_i}(\sum_i x_i w_{ij})(\sum_k w_{ji} \frac{\partial \sigma}{\partial x_j}(\sum_j (x_j \times w_{jk})) \times Err_k) \qquad (2.3)$$

where x_i is the value at neuron i, and w_{ij} is the weight between neurons i and j. Err_k is the error at output neuron k.

While this is similar to Hebbian learning, the important difference is that the error must propagate backwards across all the synapses to make the adjustments throughout the network, a process which is unrealistic given the cell physiology.

Although computers are able to compute the error and thus the changes to the weights very quickly, making neural networks a valuable tool for performing regression, the actual operation that occurs in the brain when modelling the environment must be different, and this is one of the primary motivations for my research.

2.3 Biological Plausibility

It is relatively straightforward to implement Gaussian processes directly using matrices and vectors, at least for small data sets (techniques for dealing with large data sets and efficient inversion of sparse matrices are being studied [4]). The problem with using a neural network archicture for implementing Gaussian processes arises primarily from the lack of complexity in biological neurons – they are capable of only simple unary functions on the sum of their inputs. The functions which I have allowed for neurons in the networks are accumulation

(long term integration of information), short term acculumation (where the neuron fires after receiving inputs from its neighbours) and exponentiation if the output is less than 1, which is conceivable as a gating process.

I have designed the network with these principles in mind, and attempted to solve all the problems which arose without resorting to arbitrarily complex processes. This required both approximations and iterative techniques to be employed in order to evaluate complex functions. Optimisation became extremely important in keeping the accuracy high without increasing the time required to process the data.

Chapter 3

Gaussian Processes

In contrast with neural networks, Gaussian processes learn each new example instantaneously - there is no *training* required. However, the process of inverting the matrix is slow in comparison with the process of feeding an input through a neural network. In short, Gaussian processes are fast to learn and slow to predict, while neural networks behave in (typically) the opposite manner.

Additionally, Gaussian processes have the advantage of intrinsically incorporating uncertainty into predictions. This alone has been cited [9] as cause for abandoning neural networks in favour of Gaussian processes for classification and regression tasks, which are typically dominated by uncertainty. As demonstrated in section 3.3, a Gaussian process model is equivalent to a neural network with an infinite number of hidden layer neurons. This has obvious advantages as far as the complexity of functions it can represent.

3.1 A Tool for Regression

A Gaussian process is a collection of random variables (the *targets*) $t = \{t(x_1), t(x_2), ...\}$ which have a joint distribution that is Gaussian. That is, for a collection of inputs $\mathbf{x}_n$, the form of the joint distribution is:

$$P(t|C, \mathbf{x}_n) = \frac{1}{Z} \exp\left(-\frac{1}{2}(t-\mu)^T \mathbf{C}^{-1}(t-\mu)\right) \tag{3.1}$$

C is a covariance matrix encoding a covariance function $C(x, x'; \Theta)$ where Θ is a set of hyperparameters for the covariance function. The covariance function itself is subject only to the constraint that for all x, x′, $C(x, x'; \Theta) \geq 0$, to ensure that the resulting distribution is normalisable.

Essentially, the covariance matrix determines the scale and orientation of a Gaussian distribution amongst the variables $t_1 ... t_n$. By fixing k of these, an $n-k$ dimensional conditional Gaussian is produced, which can be used to determine the most probable values for the remaining $n-k$ variables given the k values chosen. As a simple example, consider the two variable distribution illustrated in figure 3.1.

The contours are the Gaussian probability density of the two points t1 and t2. The orientation and scale is determined by **C**. Given a value t′ for t1, the distribution for t2 can be calculated, and the most probable value of t2 determined, as well as the uncertainty.

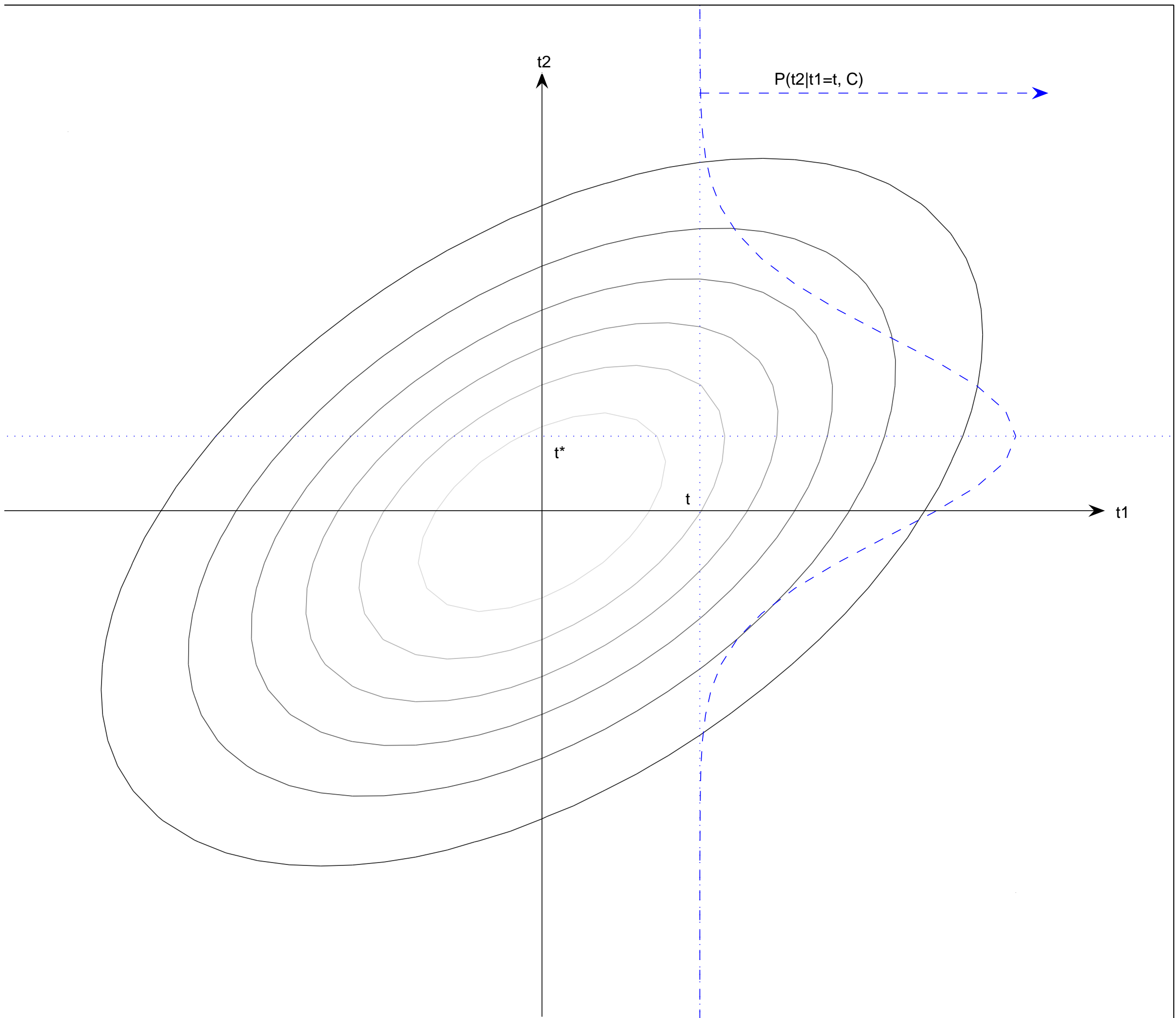


Figure 3.1: Conditional distribution of two variables. If t1 is fixed at t, then we can obtain a distribution as shown for the possible values of t2. The most likely value is shown as t*

3.2 Evaluating the mean and variance of the joint distribution

The task of regression requires one to find the distribution $P(t_{n+1}|\mathbf{t}_n)$. Applying the product rule for probability yields

$$P(t_{n+1}|\mathbf{t}_n) = \frac{P(t_{n+1}, \mathbf{t}_n)}{\mathbf{t}_n}$$

Since $P(t_{n+1}, \mathbf{t}_n)$ is really just $P(\mathbf{t}_{n+1})$, we arrive at the following distribution:

$$P(t_{n+1}|\mathbf{t}_n) = \frac{Z_{n+1}}{Z_n} \exp\left(-\frac{1}{2}\left(\mathbf{t}_{n+1}^T \mathbf{C}^{-1} \mathbf{t}_{n+1} - \mathbf{t}_n^T \mathbf{C}^{-1} \mathbf{t}_n\right)\right) \tag{3.2}$$

We must now consider the form of $\mathbf{C}_{n+1}^{-1}$. Suppose that we write it as:

$$\begin{bmatrix} \begin{bmatrix} \\ \mathbf{M}_n \\ \\ \end{bmatrix} & \begin{bmatrix} \mathbf{m}_{n+1} \end{bmatrix} \\ \begin{bmatrix} \mathbf{m}_{n+1}^T \end{bmatrix} & [\mu] \end{bmatrix}$$

Then we may write $\mathbf{C}_{n+1}$ as:

$$\begin{bmatrix} \begin{bmatrix} \\ \mathbf{C}_n \\ \\ \end{bmatrix} & \begin{bmatrix} \mathbf{k}_{n+1} \end{bmatrix} \\ \begin{bmatrix} \mathbf{k}_{n+1}^T \end{bmatrix} & [\kappa] \end{bmatrix}$$

Since $\mathbf{C}\mathbf{C}^{-1} = \mathbb{I}$, we may write down:

$$\mathbf{C}_n \mathbf{M}_n + \mathbf{k}_{n+1} \mathbf{m}_{n+1}^T = \mathbb{I} \tag{3.3}$$

$$\mathbf{k}_{n+1}^T \mathbf{M}_n + \kappa \mathbf{m}_{n+1}^T = \mathbf{0}^T \tag{3.4}$$

$$\mathbf{C}_n \mathbf{m}_{n+1} + \mu \mathbf{k}_{n+1} = \mathbf{0} \tag{3.5}$$

We can solve equation 3.3 for $\mathbf{M}_n$:

$$\mathbf{M}_n = \mathbf{C}_n^{-1} - \mathbf{C}_n^{-1} \mathbf{k}_{n+1} \mathbf{m}_{n+1}^T \tag{3.6}$$

Substituting equation 3.6 into equation 3.4 gives

$$\mathbf{k}_{n+1}^T \left(\mathbf{C}_n^{-1} - \mathbf{C}_n^{-1} \mathbf{k}_{n+1} \mathbf{m}_{n+1}^T\right) + \kappa \mathbf{m}_{n+1}^T = \mathbf{0}^T \tag{3.7}$$

Expanding gives

$$-\mathbf{k}_{n+1}^T \mathbf{C}_n^{-1} + \mathbf{k}_{n+1}^T \mathbf{C}_n^{-1} \mathbf{k}_{n+1} \mathbf{m}_{n+1}^T = \kappa \mathbf{m}_{n+1}^T \tag{3.8}$$

Solving for $\mathbf{m}_{n+1}^T$ gives

$$-\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1} = \kappa\mathbf{m}_{n+1}^T - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1}\mathbf{m}_{n+1}^T \tag{3.9}$$

$$-\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1} = \mathbf{m}_{n+1}^T\left(\kappa - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1}\right) \tag{3.10}$$

$$\mathbf{m}_{n+1}^T = \frac{-\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}}{\kappa - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1}} \tag{3.11}$$

Substituting this into equation 3.5 yields

$$\mathbf{C}_n\left(\frac{-\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}}{\kappa - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1}}\right) = -\mu\mathbf{k}_{n+1} \tag{3.12}$$

Now, since $\mathbf{C}_n$ is symmetric, so is $\mathbf{C}_n^{-1}$, and so $\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1} = \left(\mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\right)^T = \mathbf{C}_n^{-1}\mathbf{k}_{n+1}$ which lets one simplify to

$$\mu = \frac{1}{\kappa - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1}} \tag{3.13}$$

We now return to equation 3.2, and rewrite the terms of $\mathbf{t}_{n+1}^T\mathbf{C}^{-1}\mathbf{t}_{n+1} - \mathbf{t}_n^T\mathbf{C}^{-1}\mathbf{t}_n$ which depend on t_{n+1} in terms of $\mathbf{m}$, $\mathbf{M}$ and μ:

$$\begin{aligned}
\mathbf{t}_{n+1}^T\mathbf{C}^{-1}\mathbf{t}_{n+1} - \mathbf{t}_n^T\mathbf{C}^{-1}\mathbf{t}_n &= \left(\mathbf{t}_{n+1}^T\mathbf{M}_n + t_{n+1}\mathbf{m}^T + \mathbf{t}_n^T\mathbf{m}_{n+1} + t_{n+1}\mu\right) - \mathbf{t}_n^T\mathbf{C}^{-1}\mathbf{t}_n \\
&= \mathbf{t}_n^T\mathbf{M}_n\mathbf{t}_n + t_{n+1}\mathbf{m}_{n+1}^T\mathbf{t}_n + \mathbf{t}_n^T t_{n+1} + t_{n+1}^2\mu - \mathbf{t}_n^T\mathbf{C}^{-1}\mathbf{t}_n \\
&= t_{n+1}\left(\mathbf{m}_{n+1}^T\mathbf{t}_n + \mathbf{t}_n^T\mathbf{m}_{n+1}\right) + \mu t_{n+1}^2 + constant \\
&= \mu\left(\frac{2\mathbf{t}_n^T\mathbf{m}_{n+1}}{\mu}t_{n+1} + t_{n+1}^2\right) + constant
\end{aligned}$$

Completing the square, we get

$$\mu\left(\frac{\mathbf{t}_n^T\mathbf{m}_{n+1}}{\mu} + t_{n+1}\right)^2 + constant' \tag{3.14}$$

Which means that the distribution in equation 3.2 is really just the Gaussian:

$$P(t_{n+1}|\mathbf{t}_n) = \frac{1}{Z}\exp\left(-\frac{1}{2}\left(\mu\left(\frac{\mathbf{t}_n^T\mathbf{m}_{n+1}}{\mu} + t_{n+1}\right)^2\right)\right) \tag{3.15}$$

which shows that the mean value for t_{n+1} given $\mathbf{t}_n$ is given by

$$mean = -\frac{\mathbf{t}_n^T\mathbf{m}_{n+1}}{\mu}$$

Making use of equations 3.11 and 3.13 we find

$$mean = \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{t}_n \tag{3.16}$$

Similarly, the variance is simply $\frac{1}{\mu}$ which equation 3.13 tells us is

$$variance = \kappa - \mathbf{k}_{n+1}^T\mathbf{C}_n^{-1}\mathbf{k}_{n+1} \tag{3.17}$$

Regression using Gaussian processes can be accomplished by evaluating equations 3.16 and 3.17.

3.3 Relation to Forward Feed, Back Propagation Neural Networks

Neal showed in his thesis that a multilayer perceptron neural network has Gaussian process behaviour when the number of hidden neurons tends to infinity, provided weight decay is employed [12]. The covariance function is then a function of the priors for the weights and the activation functions of the neurons in the network. Indeed, it was this work which was fundamental to the development of Gaussian processes as a tool for regression.

Consider a network with a set of inputs, I, a single output O, and a number of hidden neurons H, with hyperbolic tangent activation functions. (This can easily be extended to multiple outputs as the outputs are independent of each other). Let the layers be fully connected, and all the hidden neurons and the output neuron having a bias input as well. Then the output will be

$$O = \sum_H h(x)w_h + b_0 \tag{3.18}$$

and the output of the hidden layer, $h(x)$, will be

$$h(x) = tanh\left(\sum_j x_j u_{jh} + a_h\right) \tag{3.19}$$

These weights and biases are chosen from priors. Let all the priors have zero mean – this is equivalent to weight decay. Let the standard deviation for the hidden to output weights (w_i) be σ_w, for the bias, σ_b, for the input to hidden weights(u_{ih}) be σ_u and for the hidden biases (a_h) be σ_a.

Now, if we consider the distribution of the output given a particular input x^i, we can make the following statements:

- The effect of the hidden neurons on the output has a zero mean. If we consider the expected value of the output:

$$\begin{aligned}\left\langle\sum_H h(x)w_h + b_0\right\rangle &= \sum_H \langle h(x)w_h\rangle + \langle b_0\rangle\\ &= \sum_H \langle h(x)\rangle\langle w_h\rangle + \langle b_0\rangle\\ &= \sum_H \langle h(x)\rangle\, 0 + \langle b_0\rangle\\ &= 0 + 0\end{aligned}$$

- The variance of the the output is finite, regardless of how many hidden neurons there are. Variance is defined as $\left\langle(x - \langle x\rangle)^2\right\rangle$, which in this case is $\left\langle(\sum_h w_h h(x^i) + b_0 - 0)^2\right\rangle$ since we just saw that $\langle\sum_h h(x)w_h + b_0\rangle = 0$.

$$\begin{aligned}\sigma_O^2 &= \left\langle(\sum_h w_i h(x^i) + b_0)^2\right\rangle\\ &= \langle b_0^2\rangle + \langle 2b_0\rangle\left\langle\sum_{j,k} w_h h(x^i) w_j j(x^i)\right\rangle + \left\langle(\sum_h w_h h(x^i))^2\right\rangle\\ &= \sigma_b^2 + 0 + \left\langle\sum_h (w_h h(x^i))^2\right\rangle\end{aligned}$$

Now, since the weights from the hidden layer are independent of the values at the hidden layer, we may further write

$$\begin{aligned}\sigma_O^2 &= \sigma_b^2 + \langle w_h^2 \rangle \left\langle \sum_h h(x^i)^2 \right\rangle \\ &= \sigma_b^2 + \sigma_w^2 \sum_h \langle h(x^i)^2 \rangle\end{aligned}$$

We now need to make use of the central limit theorem, to arrive at

$$\begin{aligned}\sigma_O^2 &= \sigma_b^2 + \langle w_h^2 \rangle \sum_h \langle h(x^i)^2 \rangle \\ &= \sigma_b^2 + \sigma_w^2 H \langle h(x^i)^2 \rangle\end{aligned}$$

We can use a very similar argument to find the covariance for two arbitrary inputs x^i, x^j is given by

$$covariance_O(x^i, x^j) = \sigma_b^2 + \sigma_w^2 H \langle h(x^i) h(x^j) \rangle \tag{3.20}$$

If we let $\alpha = \sigma_w \sqrt{H}$ then we can rewrite this as

$$covariance_O(x^i, x^j) = \sigma_b^2 + \alpha^2 \langle h(x^i) h(x^j) \rangle \tag{3.21}$$

Which means that for any two variables, we have an output which has zero mean and a covariance of

$$\sigma_b^2 + \alpha^2 covariance(x^i, x^j) \tag{3.22}$$

Thus it has been shown that the limit of a neural network with infinite hidden neurons has the critical property that any set of function values has a Gaussian joint distribution, the defining property of a Gaussian process.

Chapter 4

The Inversion Problem

The central problem in implementing Gaussian processes in a distributed neural environment is to invert the covariance matrix. However, it is not strictly necessary to calculate the inverse directly, instead we can settle for the product of k with the inverse. The problem can be reformulated as follows:

Given a matrix $\mathbf{C}$ and a vector $\mathbf{k}$, we wish to find $\mathbf{C}^{-1}\mathbf{k}$. Supposing that $\mathbf{C}^{-1}\mathbf{k} = \mathbf{x}$, then we can convert this into an equation with a single unknown rather than two by premultiplying by $\mathbf{C}$: $\mathbf{k} = \mathbf{Cx}$. The problem then reduces to iteratively improving x until the difference between $\mathbf{Cx}$ and $\mathbf{k}$ is sufficiently small.

Three approaches were identified. The first was Gauss-Siedel iteration, a numerical method which converged quickly but proved to be too complex for a realistic biologically plausible implementation.

The second was proposed by Daugman [2] and then futher refined by Pece [13]. The original context for this algorithm was in determining the inverse of a Gabor basis (internal representation) to pixel (direct visual stimulus) transformation which occurs in the brain. The algorithm is as follows: Determine the error ξ as $\mathbf{k}$ - $\mathbf{Cx}$. Then increment $\mathbf{x}$ by $\eta(\mathbf{C}^T\xi)$ where η is sufficiently small that the algorithm will converge. (See section 4.4) The algorithm can be considered to be a gradient descent with η as the step size, although it is useful to consider the iterations as an infinite series for the purposes of determining an optimal η. Figure 4.4.1 shows the relative performance of the algorithm with different values of η, section 4.4 gives some insight into the mathematical basis behind the algorithm.

The third approach was proposed by Gibbs[5]. This algorithm is similar to Daugman's, excepting that $\mathbf{x}$ is adjusted only by $\eta\xi$ in each iteration.

The implementation of the Gibbs and Daugman algorithms in neural networks are given in 4.1 and 4.2 respectively. Figure 4.2 bears close resemblence to the model which Pece has devised for visual model processing. The neurons labelled k are equivalent to the retinal ganglion cells, those labelled E are equivalent to the LGN cells, and the g neurons represent the cells in the primary visual cortex. [13]

4.1 Objective Functions for the two Algorithms

The convergance of both algorithms employed (Gibbs and Daugman) is guaranteed only for certain matrices, however it is possible to make adjustments to well-behaved covariance matrices such that they converge. (See 4.4 By well-behaved, I mean that the matrices must be positive definite and diagonally dominant). It should be noted that any covariance matrix is necessarily symmetric by virtue of the symmetry of the covariance relation, and so the difference between the equations is essentially that Gibbs multiplies ξ by $\mathbf{C}$ while Daugman

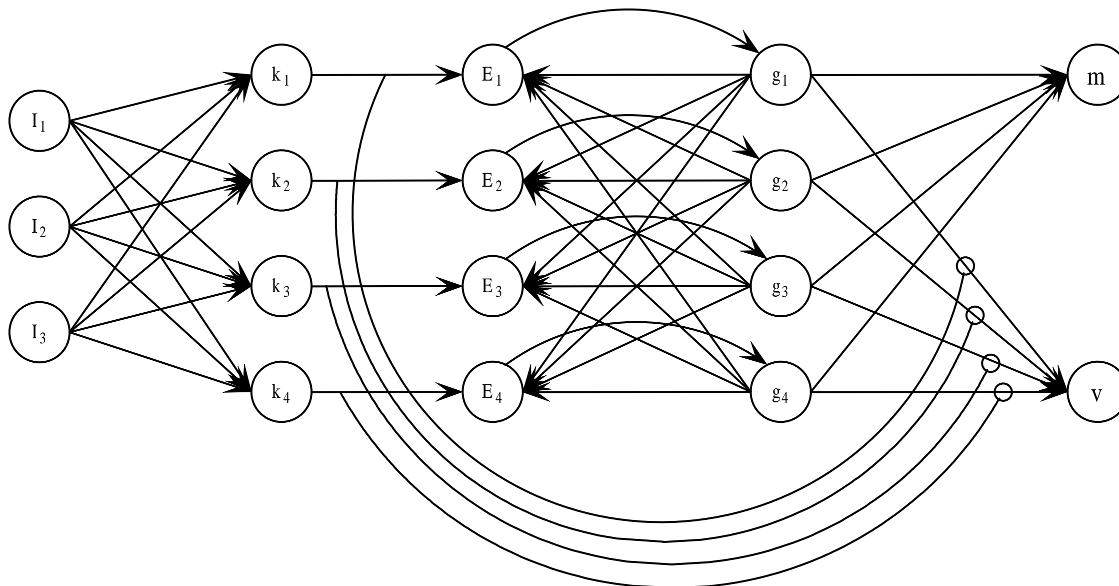


Figure 4.1: Gibbs Inversion Network

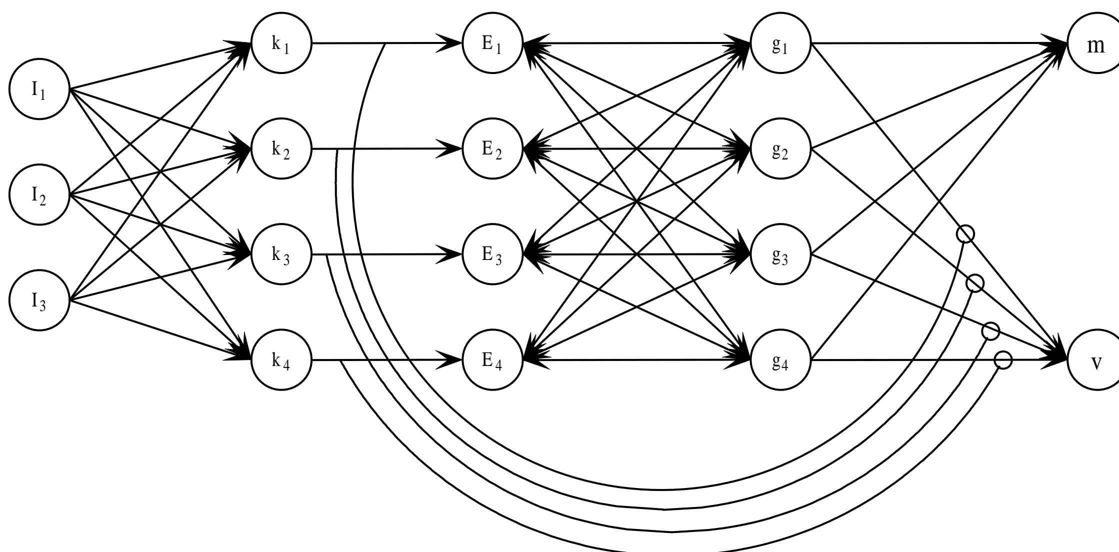


Figure 4.2: Daugman Inversion Network

doesn't. This point is discussed further in section 4.2.

To improve the approximation at each step, we require a method of determining how close the current value of **x** is to the true value of **x**. The dot product $Q = \frac{1}{2}(x - C^{-1}k).(x - C^{-1}k)^T$ is an excellent candidate, but requires a priori knowledge of C^{-1}. Instead, it is useful to have a *pseudo-error* Q which is indicative of, but not necessarily equal to, the true error. The problem then becomes minimising the pseudo-error. Daugman proposes the following:

$$Q^{Daugman} = -\frac{1}{2}(\mathbf{x} - \mathbf{Ck})^{\mathbf{T}}.(\mathbf{x} - \mathbf{Ck}) \tag{4.1}$$

Clearly, if $x = C^{-1}k$ then $Cx - k = 0$ and so $Q^{Daugman}$ has a zero exactly where the Q^{true} has its zero. This is quadratic in the true error multiplied by **C**. This can be expanded to

$$Q^{Daugman} = -\frac{1}{2}(\mathbf{k^T k} - \mathbf{k^T Cx} - \mathbf{Cx^T k} + \mathbf{Cx^T x}) \tag{4.2}$$

by virtue of C being symmetric, and then the gradient with respect to x is given by

$$\nabla_x Q^{Daugman} = \mathbf{C}(\mathbf{k} - \mathbf{Cx}) \tag{4.3}$$

Meanwhile, Gibbs explores the following function for the pseudo error:

$$Q^{Gibbs} = \mathbf{x^T}(\mathbf{k} - \frac{\mathbf{1}}{\mathbf{2}}\mathbf{Cx}) \tag{4.4}$$

The rationale for this choice is purely that the derivative is zero at the solution. The derivative of the objective function with respect to x is given by

$$\nabla_x Q^{Gibbs} = (\mathbf{k} - \mathbf{Cx}) \tag{4.5}$$

The process of improving **x** is then simply to take a small step in the direction of $\nabla x Q$ - that is,

$$\Delta x = \eta \nabla_x Q \tag{4.6}$$

A salient observation is that for C = $-\mathbb{I}$, $\nabla_x Q^{Daugman} = -\nabla_x Q^{Gibbs}$. This seems to be indicating that given a particular **x** vector, the two update rules make opposite adjustments to improve the value! In fact that is precisely what is happening - for all matrices **C**, $Q^{Daugman}$ is a negative parabaloid, meaning that the process of reducing the pseudo-error is equatable to gradient ascent. On the other hand, Gibbs' approach doesn't require that we find a minimum, just a stationary point. A local minimum or even a maximum will yield the inverted matrix. This means that sometimes (such as when C = $-\mathbb{I}$) the process of inversion is actually gradient ascent, not descent. This is reflected in the gradient of the gradient:

$$\nabla\nabla_x Q^{Daugman} = -(C^2) \tag{4.7}$$

The square of any positive definite matrix is positive definite, so this is always negative definite

$$\nabla\nabla_x Q^{Gibbs} = -C \tag{4.8}$$

This is negative definite if C is positive definite, positive definite otherwise.

4.2 Making the Algorithms Covariant

MacKay has made an interesting observation about gradient descent.[8] Suppose we have some (dimensionless) objective function L, and are attempting to adjust a set of parameters θ based on the derivative of L with respect to θ_i. The equation used to update the parameters is typically given as:

$$\Delta\theta_i = \eta \frac{\partial L}{\partial \theta_i}$$

But this is dimensionally incosistent. The left hand side has units of θ, whereas the right hand side has units of $\frac{1}{\theta}$. This is not an extremely serious problem, as evidenced by the great success of the algorithm in many parameter optimisation algorithms. In fact, if η is scaled with 1/n for online learning tasks, then the algorithm will asymptotically converge by the Munro-Robbins theorem [8]. However, the algorithm is not *covariant*, meaning that it isn't guaranteed to give the same result independent of the units used to measure the parameters and surface [7]. This can be corrected if the update rule had the form:

$$\Delta\theta_i = \eta \sum_{i'} M_{ii'} \frac{\partial L}{\partial \theta_i}$$

with a positive definite matrix M whose (i,i') element has dimension of $w_i w_{i'}$. The most natural choice for this matrix arises from the curvature of the objective function.

$$A = -\frac{\partial^2 L}{\partial \theta^2}$$

Then we can let M be A^{-1} and the algorithm will be covariant [8]. Furthermore, this alogirthm is now the Newton algorithm, and as such will cope better with ill-conditioned objective functions.

In the case of the Gibbs algorithm,

$$-\nabla\nabla Q = 1$$

So, the algorithm is already covariant. However, with Daugman,

$$-\nabla\nabla Q = C$$

This means we should premultiply the Daugman update rule with C^{-1} to make it covariant. This gives

$$\mathbf{C^{-1}}\nabla_x Q^{Daugman} = \mathbf{C^{-1}C}(\mathbf{k} - \mathbf{Cx})$$
$$\mathbf{C^{-1}}\nabla_x Q^{Daugman} = (\mathbf{k} - \mathbf{Cx})$$

Which is now exactly the same as the update rule for Gibbs.

4.3 Convergance Speed

4.3.1 Relative speed of the two algorithms

Gibbs' algorithm converges typically an order of magnitude faster than Daugman's. This decrease in inversion time, along with the demonstration in 4.2 that Gibbs' algorithm is just an improved version of Daugman's has led me to continue only with the Gibbs model. Daugman's approach is also less biologically plausible (the error needs to be propagated backward through the synapse).

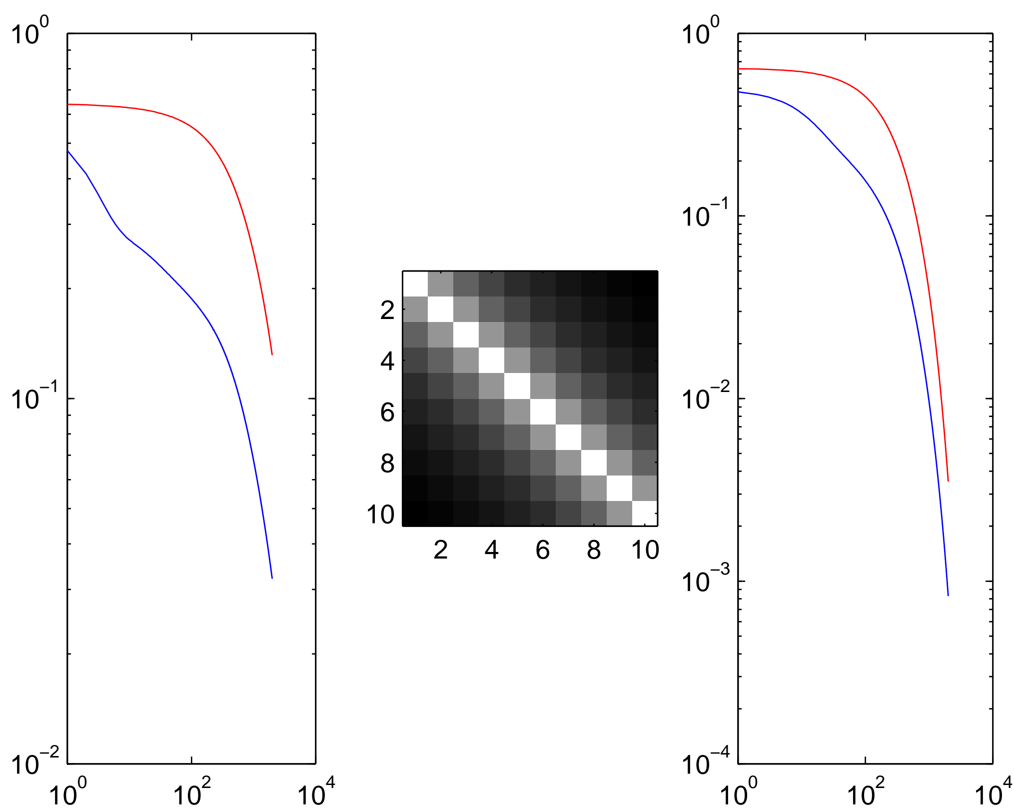


Figure 4.3: Error remaining after n cycles of Daugman's algorithm(left) and Gibbs' algorithm(right). In each case the blue curve represents the actual error and the red curve the pseudoerror. The matrix in the centre shows the covariance matrix - white represents 1 and black 0

4.4 Proof of Correctness

It is easy to show that Gibbs' algorithm is correct. We start by assuming the value of **g** at time zero is **0**. Then, at time t,

$$\mathbf{g}(t) = \mathbf{g}_{t-1} + k^T - \mathbf{C}\mathbf{g}_{t-1}$$

$$\mathbf{g}(t) = \mathbf{g}_{t-1}(\mathbb{I} - \mathbf{C}) + k^T$$

The closed form for **g** at time t is then

$$\mathbf{g}(t) = k^T \sum_{i=0}^{t-1} (\mathbb{I} - \mathbf{C})^i$$

Multiply both sides by $(\mathbb{I} - \mathbf{C})$

$$\mathbf{g}(t)(\mathbb{I} - \mathbf{C}) = k^T \sum_{i=1}^{t} (\mathbb{I} - \mathbf{C})^i$$

Subtract the preceeding two equations to give

$$\begin{aligned}\mathbf{g}(\infty)(\mathbb{I}-(\mathbb{I}-\mathbf{C})) =& k^T\left(\mathbb{I}-(\mathbb{I}-\mathbf{C})^t\right)\\ \mathbf{g}(\infty)(\mathbf{C}) =& k^T\left(\mathbb{I}-(\mathbb{I}-\mathbf{C})^t\right)\end{aligned}$$

Multiplying on the right by C^{-1} yields

$$\mathbf{g}(t) = k^T\left(\mathbb{I}-(\mathbb{I}-\mathbf{C})^t\right)\mathbf{C}^{-1}$$

Finally, taking the limit as t tends to infinity

$$\mathbf{g}(t) = k^T(\mathbb{I}-0)\mathbf{C}^{-1}$$

$$\mathbf{g}(t) = k^T\mathbf{C}^{-1}$$

As required. This makes some important assumptions about C, which require attention. First, we must assume that C is nonsingular (which it is, or else the covariance function is malformed). Second, we assume that the sum

$$\sum_0^\infty(\mathbb{I}-C)^n$$

converges and that specifically

$$\lim_{n\to\infty}(\mathbb{I}-C)^n = 0$$

The definition of convergence means that if we can prove the latter, the former is trivially true (as zero both exists and is finite). Making use of the eigen decomoposition theorem [16] we can rewrite $\mathbb{I}-C$ in terms of the matrix D which has the eigenvalues of $\mathbb{I}-C$ along its diagonal:

$$\mathbb{I}-C = P^{-1}DP$$

So we now have to show finiteness of

$$\lim_{n\to\infty}(P^{-1}DP)^n$$

This is much easier, since

$$P^{-1}DP^2 = P^{-1}DPP^{-1}DP = P{-}1D^2P$$

and in general, we have

$$(P^{-1}DP)^n = P{-}1D^nP$$

which means that all that remains is to show that

$$\lim_{n\to\infty}P^{-1}D^nP \tag{4.9}$$

is defined and finite. Because D is diagonal, $[D^n]_{ij} = [D]^n_{ij}$ and so we conclude that if for all eigenvalues λ_i of $\mathbb{I}-C$, $|\lambda_i| < 1$ then this is simply the zero matrix, which is defined, and finite. Otherwise, the limit is infinite, and therefore the algorithm fails. In general, $|\lambda_i| \not< 1$, but we can sometimes force the condition by introducing a new parameter, η, as follows: If

$$\mathbf{g}(t) = \mathbf{g}_{t-1} + \eta k^T - \eta\mathbf{C}\mathbf{g}_{t-1}$$

Then we have

$$\mathbf{g}(t) = \eta k^T\sum_{i=0}^{t-1}(\mathbb{I}-\eta\mathbf{C})^i$$

so it follows that

$$\mathbf{g}(t) = \eta k^T \left(\mathbb{I} - (\mathbb{I} - \eta\mathbf{C})^t\right) \eta^{-1}\mathbf{C}^{-1}$$

And the effect of η vanishes except in the limit:

$$\mathbf{g}(t) = k^T \left(\mathbb{I} - (\mathbb{I} - \eta\mathbf{C})^t\right) \mathbf{C}^{-1}$$

Now since we are free to choose η, we try to select an appropriate value so that the limit in equation 4.9 converges. In order for this to work, we must choose η such that the eigenvalues of $\mathbb{I}-\eta\mathbf{C}$ are of magnitude less than one. Such an η always exists for a positive definite matrix because:

- It is an identity that the eigenvalues λ of $\mathbb{I} + \alpha M$ are $1 + \alpha\lambda$ [15]
- the eigenvalues of a positive definite matrix are all positive or zero [17]
- therefore by letting α be $-\frac{1}{\max\lambda}$, we guarantee that all eigenvalues of $\mathbb{I} - \mathbf{C}$ are of magnitude equal to or less than one
- while the inequality is in reality strict, in practise if the diagonal elements are 1, the matrix is very ill-conditioned and the inverse can be very large. As the matrix size increases, finding the inverse can become increasingly difficult. Fortunately, if the diagonal elements are even slightly larger than 1, the inversion is much easier. For this reason, it is sensible to impose the restriction $\theta_3 \geq 0.01$.

As the algorithm converges only for sufficiently small values of η, we need a means of approximating the largest eigenvalue of **C**. There are numerous ways to approximate the eigenvalue of a matrix if we already have a good guess at the associated eigenvector, such as the Rayleigh quotient[1], but it is critical to find an upper bound for the largest eigenvalue to be sure the algorithm converges. To do this, we use the fact that the largest eigenvalue is strictly less than the maximal row sum in a symmetric matrix [10], which in turn is bounded by $N(\theta_1 + \theta_2) + \theta_3$, for a matrix having N columns.

$$\eta_{estimate} = \left|\frac{1}{N(\theta_1 + \theta_2) + \theta_3 + 1}\right| \tag{4.10}$$

4.4.1 Speed Relative to Orthodox Matrix Inversion, and the Effect of η

Inverting a general (non-singular) matrix is considered to be an $O(n^3)$ operation. For large data sets, the performance of the inversion algorithm is critical to the plausibility of using Gaussian processes for regression. Excepting the size of the covariance matrix, the value of η is the most relevant parameter when determining the convergence speed. Experimental data shows that using this has similar performance to using the inverse of the largest eigenvalue of $\mathbb{I} - C$.

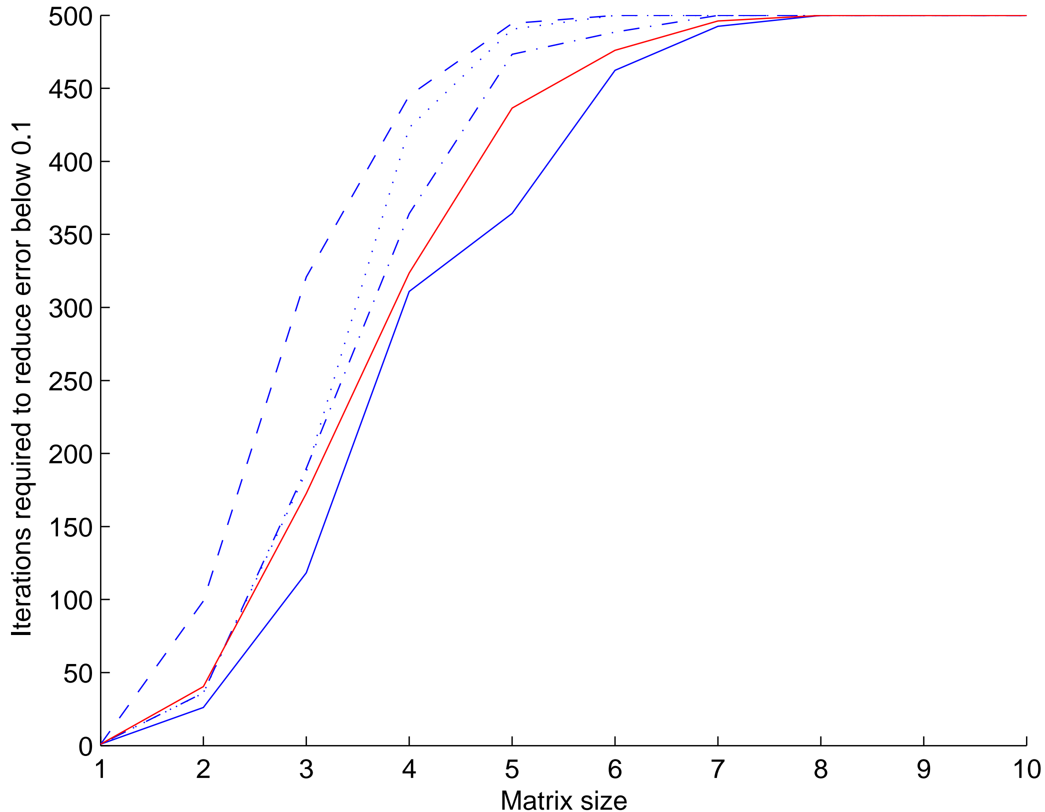


Figure 4.4: Number of iterations required to achieve a root mean square error of less than 0.1 for matrices of differing sizes. The curve in red uses the inverse of the largest eigenvalue of $\mathbb{I} - C$ for η, and the blue curves have η as 1/(sn + 1) where s is: [dashed=10, dash-dotted=1, solid=0.1, dotted=0.01]

4.5 A Space Optimisation

Finally, the use of Gibbs's algorithm allows for a slight space optimisation. Because the change to each component of $\mathbf{x}$ is a linear function of $\mathbf{k}$ and $\mathbf{Cx}$, this can be computed without the intermediate step of explicitly determining ξ, removing an entire layer from the network, and increasing overall connectivity amongst the inversion neurons. Furthermore, the neurons representing the $\mathbf{x}$ vector are now behaving as accumulators, creating a more biologically realistic system.

This optimised network is represented in figure 4.5.

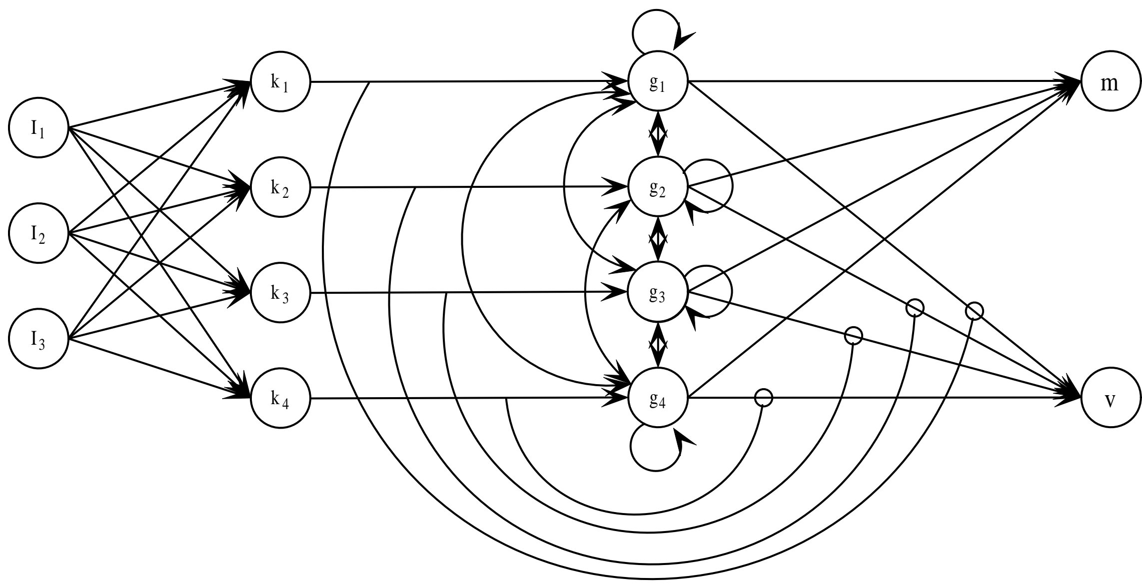


Figure 4.5: Space Optimised Network

Chapter 5

Biological Background

5.1 Grandmother Cells

One reason for allowing exponetiation as a valid neural process is the concept of *Grandmother cells* – neurons which respond very strongly to a particular stimulus, but weakly to others. The name, coined by Jerry Lettvin, comes from the concept of highly specialised neurons postulated to be present in the brain which are trained to specifically recongise the features describing the organism's grandmother [6]. Since the original proposal (which was actually relating to frogs having fly detectors), it has become accepted that we do not have detectors for features for every object we can recognise (this was humourously put by Harris in 1980 in terms of *Yellow Volkswagen cells*). However, neurons with grandmother-cell-*like* behaviour have indeed been observed, especially in the visual system. These cells respond preferentially to images of faces, or hands. Numerous studies have been done in this field (for example [3]).

An obvious analogy to Gaussian processes can now be drawn. The covariance matrix is similar in composition to a set of Grandmother cells. Each grandmother cell is trained to respond strongly when shown the data it was trained on. When the data is exactly the same, the cell produces the target value it was trained on. Otherwise, it produces an expontially weaker value. When a prediction is required for a new example, the grandmother cells all see the input and the sum of how strongly they react is accumulated as the mean.

The structures present which perform this job are henceforth referred to as preprocessors, as they perform the task of converting the raw input data into the **k** vector.

5.2 Biological Parallels

Transmission of information from the eye to the various parts of the brain is done in several stages. One of the first is the transmission from the Lateral Geniculate Nucleus to the primary Visual Cortex. In the process, the information is converted from a dense representation at the LGN to a sparse one at the V1. However, approximately 60% of the synapses in the LGN are in fact signals coming from the cortex to the LGN, that is, feedback from the cortex. Little is known about the function of these synapses. One suggestion is that the brain to eye signals are describing templates (hypotheses about what might be visible), and the signals travelling back in the expected direction are carrying residuals – errors between the templates and what is actually observed.[11].

The first parallel is the conversion of a stimulus into the responses of a large number of 'grandmother cells', most of which are negligible. When the input is first received, it is raw data with dimension equal to the surface being modelled. Once it reaches the inversion

part of the network, it has been converted into a sparse vector with dimension equal to the number of examples which have been observed, typically much greater than the dimension of the surface.

The second parallel is the feedback which occurs while inverting the matrix. In the uncompressed Gibbs network, a substantial proportion ($\frac{n}{n+1}$) of the signals are travelling in this 'wrong' direction, and indeed the signals travelling in the 'right' direction are conveying error signals.

In fact, the work undertaken by Daugman and Pece [2, 13] was originally to form a model of how images were represented in the brain, and the need to invert a matrix arose out of the requirement to invert a linear transformation from internal models to the physical observations. MacKay suggested in 1962 that the brain is actively trying to predict the physical world, and only the predictions which are not accurate require changes to the internal model. This requires the inversion of the prediction procedure to alter the internal model – a complicated procedure, but when the model is reasonably good, it reduces the amount of learning that needs to be done.

5.3 Shunting Inhibitions and Variance

The variance of the prediction is given by the expression $\mathbf{k}^T\mathbf{C}^{-1}\mathbf{k}$. Part of this has already been calculated by the network which determines the mean, specifically $\mathbf{k}^T\mathbf{C}^{-1}$. All that remains is to find the dot product of this with $\mathbf{k}$. Initialy this seems like a trivial problem, but from a biological perspective it poses some difficulty, as neurons are unable to multiply inputs – rather, they can only accumulate them. One possibility involves a complex network for each element in $\mathbf{k}$ using logs to reduce the multiplication to addition, but a more feasible possibility is suggested by the process known as *shunting inhibition*.

Shunting inhibition has been proposed as a possible mechanism for neurons to divide numbers [14]. The actual process involves the output of one neuron inhibiting the transmission of charge between two other neurons. In this way, the transfer between the latter two neurons can be moderated by a seperate process. As the elements of $\mathbf{k}$ are between 0 and 1, computing $\mathbf{k}^T\mathbf{C}^{-1}\mathbf{k}$ can be considered to be scaling the elements of $\mathbf{k}^T\mathbf{C}^{-1}$ by the elements of $\mathbf{k}$, a task to which shunting inhibitions seem ideally suited. The scaled output is the accumulated in the variance output node.

Chapter 6

Data Recall and Sleeping

With many biologically inspired regression techniques, the training set needs to be considered more than once. However, most techniques require the data to be processed an implausibly large number of times. For example, a typical neural network might need to see a single example several hundred times in different training epochs before it can correctly make predictions based on the example. One advantage of a Gaussian process model is that the training data need only be observed once - once the covariance matrix is augmented, the example is 'remembered' forever.

Unfortunately, for reasons discussed in section 8.4, determining the hyperparameters without the original training set is very difficult. For this reason, each training example is *encoded* into the network by adding a set of synapses leading back from the second preprocessor neuron (the one which performs accumulation) to the input neurons, with weights equal to the input which that preprocessor was trained on. Additionally, a synapse from this same neuron to the mean neuron with a weight of the negation of the target value is added.

These neurons have a special behaviour. During normal operation they have a weight of 0 - contributing nothing when the preprocessor fires. However, in another mode of operation, these synapses become active and take the weights they were trained on again. To extend the biological metaphor, another neural process would have to initiate this secondary behaviour. This could be chemical in nature or take the form of shunting inhibition (see 5.3) from other neurons.

In this state, by randomly sending a spike of 1.0 to one of the second preprocessors, the stimulus that part of the network was trained on reappears at the input. Further, because this stimulus is the one which will cause that particular part of the network to be strongly excited, the input will be continually refreshed when the second preprocessor fires in each subsequent cycle.

Because of the nature of this behaviour, which is essentially examining past events in some (possibly random) order and then attempting to judge how well they fit with the internal model which has been constructed, I have referred to this secondary neural behaviour as *sleeping* (or dreaming). While sleeping and dreaming are considerably more complex than the process described, little is known about their purposes and effects, although it has long been believed that the purpose relates to reconciling models based on previous events. Again this seems analagous to the process of correcting templates seen at the end of section 5.2

Chapter 7

Faster Predictions: Instinct

One of the advantages of Gaussian processes over forward-feed backward-propagating neural networks is their convergance speed. FFBP neural networks are very slow to train – considerably slower than biological systems. However, once trained they are able to produce their predictions almost instantaneously, considerably *faster* than biological systems.

Compare this behaviour to the mechanism described in 4. This is very fast to learn, but because of the iterative nature of inverting the matrix, is very slow to make predictions. The time taken before a prediction can be made is dependent on two factors. First, a larger number of training examples will lead to a larger matrix to be inverted. The time taken to invert a matrix by any technique is dependent on the size. Secondly, the time taken to find the inverse of a matrix of a given size so that it is sufficiently accurate (that is, $\mathbf{Cx}$ - $\mathbf{k}$) is still considerable.

We return to the fundamental mismatch which both neural networks and Gaussian processes have with nature – either they are too slow to predict, or too slow to train. This suggests that a hybrid between the approaches may be effective. The idea has lead me to develop a secondary single layer forward-feed, backward-propagate network mechanism which runs silently in parallel with the Gaussian process mechanism and essentially learns C^{-1}. Because it is single-layered, this network is less biologically implausible than the multi-layered networks – a single layer network learns by Hebbian learning, which is an established model for neural learning, rather than propagating the error across multiple synaptic gaps. This secondary network can then directly compute $\mathbf{k^T C^{-1}}$.

Ideally, this should be used as the initial value for the inversion algorithm. However, it is difficult to imagine a process which fires once, but continually receives input and adjusts its weights when data is avaiable – when should it fire next? Instead, I have added another two outputs to the network, and connected these to this alternative inversion network. The output from these is also interpreted as a mean and variance, but are more of an *instinctive* prediction. Given a set of inputs, the network can immediately produce a poor estimate of the mean and variance using this, but if given time to crunch the numbers and do the manual inversion, a more accurate version can be produced. Once an input has been seen sufficient times (providing no new examples have been added) the output of the alternative network is quite accurate. See section 9.2 for details on the learning rate.

Chapter 8

Determining Hyper-hyper Parameters

8.1 Gaussian Process Hyper-Parameters

The effectiveness of the Gaussian process model depends critically on the covariance function. Stationary covariance functions are usually parameterised in N+3 parameters for an N dimensional input surface. Specifically, there are three global *hyperparameters*, Θ_1, Θ_2 and Θ_3, and a set of N *length scales*. Θ_1 determines the relative scale of the noise in comparison with the data, and typically must be greater than one for the covariance matrix to be nonsingular (or at least positive definite). Θ_2 determines the long term behaviour of the predictions away from data points. At points sufficiently far from training examples (sufficiently far here is a function of the hyerparameters themselves), the prediction will be a function of Θ_2. For zero mean data, this hyperparameter should be left at zero. Θ_3 is a component of the noise model, and is effectively representative of how much white noise is present in the data set. The covariance function for Gaussian white noise is a single spike at zero, and zero everywhere else. It follows that adding $\delta_{i,j}\Theta_3$ is equvialent to adding in this Gaussian white noise model.

The length scales, r_i, indicate the relative importance of each input. If the length scale is very small for a particular dimension, then slight deviations in that dimension from the training examples will contribute a large variance, whereas another dimension having a larger length scale may have an equivalent deviation from the training example with little increase in overall variance. Isotropic Gaussian processes are those in which all the length scales are equivalent, and these are the examples I have concentrated on for the purposes of training the networks.

8.2 Hyper-hyper Parameters

In a neural network, the outputs are determined by means of a set of parameters (specifically the weights). If these weights are themselves determined by a set of parameters, these are called the hyper-parameters. In the case of the networks I have studied, the hyper-parameters are the rows of the covariance matrix **C**. The rows of **C** are themselves determined by the parameters of the covariance function as described in 8.1. Extending the nomenclature, the length scales and the set of three Θ values should correctly be referred to as hyper-hyper-parameters. However, for ease of understanding, they will be referred to henceforth as hyper-parameters where the meaning is clear from context.

8.3 Learning the Hyper-Parameters

8.3.1 Orthodox Approach: Optimising Hyper-Parameters

The hyper-parameters are difficult to adjust because of the complex dependence of the weights of the network on the values the hyper-parameters had during training. For example, the length scales are implemented as the weights on the neurons between the two neurons in the preprocessing stage. Adjusting them after training would require the recomputation of the covariance matrix, and hence the complex and non-linear adjustment of all the weights between the grandmother cells.

It is conceivable that in a seperate mode of operation the network might actually recalculate all the weights that represent the covariance matrix. To do this, after adjusting a hyper-parameter, the examples would all have to be redisplayed, and the weights would be able to be adjusted using the delta rule. This was my first approach, and lead to the replay system described in chapter 6

8.3.2 Bayesian Approach: Integrating the Hyper-Parameters Out

The Bayesian statistician takes a slightly different approach to the problem of optimisation. Instead of trying to improve the value presently available, they integrate over all possible values and determine the output based on which values perform best. To this end, I have implemented a *network of networks*, and developed a mechanism for determining how good each network is given its hyperparameters. The initial values for the hyper-parameters are chosen at random from a hyper-prior, which must be programmed in after the data has been inspected. With sufficient networks, the data will overwhelm this prior making the selection unimportant. However, in practical terms, having sufficiently many networks is computationally infeasible, and so a suitable hyperprior must be encoded by the programmer.

8.4 The Exact Method

I believe this to be biologically implausible. The log likelihood for a Gaussian proccess is

$$L = -\frac{1}{2}log(det(\mathbf{C_N})) - \frac{1}{2}\mathbf{t}_N^T\mathbf{C}_N^{-1}\mathbf{t}_N - \frac{N}{2}log(2\pi). \quad (8.1)$$

To optimise a hyperparameter θ by gradient ascent, we must evaluate

$$\frac{\partial L}{\partial \theta} = -\frac{1}{2}trace(\mathbf{C^{-1}}_N\frac{\partial \mathbf{C}_N}{\partial \theta}) + \frac{1}{2}\mathbf{t}_N^T\mathbf{C}_N^{-1}\frac{\partial \mathbf{C}_N}{\partial \theta}\mathbf{C}_N^{-1}\mathbf{t}_N - \frac{N}{2}log(2\pi). \quad (8.2)$$

Although computing the trace is relatively straightforward, the remainder of the operations seem insurmountably complex to implement in a neural network. Multiplying matrices requires a large number of carefully wired neurons. Worse yet, we must know $\mathbf{C}^{-1}$ to evaluate this task, which is never found directly (only kC^{-1} is computed by the network). Furthermore, the derivative of $\mathbf{C}$ with respect to θ is complex to find as it essentially requires a second covariance matrix to be calculated using the derivative of the covariance matrix. This could be performed in parallel with the formation of the original covariance matrix, but the other challeneges remain.

There are methods of approximating (large) matrix multiplication, however these are typically even more implausible than the direct multiplcation. I abandoned the method for directly computing the likelihood and turned to an approximation.

8.5 The Approximated Likelihood

For any prediction mechanism, the probability of the parameters H, given some data D, can be determined by applying Bayes' rule:

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{8.3}$$

In the case of the network, P(H) is taken care of by selecting the initial set of hyperparameters from the hyperprior. The evidence P(D) is considered constant across all hypotheses (that is, all possible values for the hyper-parameters), and can be ignored provided we agree to treat the results of the calculations no longer as probabilities of the parameters given the data, but as degrees of belief in each parameter, given the data. This leads to some normalisation complications, as discussed in 8.7.

Essentially, the equation reduces to $P(H|D) = P(D|H)$ if H is appropriately sampled. If we take a naïeve approach and assume that all the data points are independent, then it follows that:

$$P(H|D) = \prod_i P(H|D_i) \tag{8.4}$$

But, as mentioned in section 5.3, neurons are not good at multiplying inputs. Instead, it is helpful to consider $log(P(H|D))$ which may be expressed as a sum:

$$log(P(H|D)) = \sum_i log(P(H|D_i)) \tag{8.5}$$

We now turn to a method for evaluating the log likelihood of the data for given network.

8.6 The Log of the Approximated Likelihood

The first attempt at this was to pick a random number r, between 1 and the length of k, and set k_i to be $\delta_{r,i}$. The effect was to get the initial output being the target value which the rth g neuron was trained on, and from there progress towards the output that the whole network agreed on, the difference between these being proportional to the error in the hyperparameters.

In fact, the theorem in A shows that we don't even need to wait till the algorithm has converged - the rate at which the output is changing is linearly proportional to the difference between the target value and the converged output. However, this has a fatal problem. Because we only consider the training examples, the error is smallest precisely on those networks which are closer to the training examples. Even with a suitably chosen hyperprior, it is easy to see that the network of networks will be, in the limit of infinite networks, not the network with the most useful hyperparameters, but the one which goes through precisely all the training examples. This network will necessarily be the one with length scales set to zero. The problem is then that the network is infinitely over-trained, and predictions away from these points are completely useless.

A better approach is to employ cross validation. In this mechanism, one of the g neurons is turned off completely - this is achieved by having it immediately leak all the input charge (from the soma, not down the synapses) rather than accumulate it and transmit it. This effectively reduces the network to one which has never seen this example. The associated k neuron is then switched on, the stimulus is recreated (See chapter 6) and the network makes a prediction about the point.

This prediction is collected, along with the negation of the target value for the 'sleeping' g neuron, in the mean neuron, and in turn passed into a new neuron - the likelihood neuron. The process is the repeated with other g neurons. (Presumably this could be done either at random, or in a newest-first fashion in a biological network, rather than in strict reverse age order). The output of the likelihood is then considered the log likelihood for the network.

The network network collects all the errors, and normalises their inverses (See section 8.7). Then, a set of weights from each of the smaller networks is adjusted to give a grand total for the mean and variance, weighted so that the network with the largest inverse error contributes most to the overall output.

8.7 The Normalisation Problem

Unfortunately, because measuring the evidence is so difficult, rather than probabilities to be summed over, we have beliefs which must be carefully normalised to give probabilities. Doing this with neurons seems to end up quite unappealing; we need two phases of behaviour – one where the beliefs are calculated, and a second where we sum all these up and divide by the total. The division might still be accomplished by some sort of shunting inhibition, but the entire process seems somewhat unbelievable.

One possible alternative is to sample from networks using Markov Chain Monte Carlo methods, such as Metropolis-Hastings, or Gibbs sampling. See future work (chapter 10)

Chapter 9

Visualisation

9.1 A One Dimensional Example

It is useful to study a one dimensional 'toy' example. Data can be cheaply generated, and the dimensionless nature of the covariance matrix means that there is little difference between a one dimensional regression problem and a 100 dimensional problem, but a one dimensional example is possible to visualise. I have selected a smooth function as the process which generated the training data, and then produced a series of graphs showing the predictions and variances for a number of unseen points.

The function used to generate the data is

$$10 \exp^{-|3x|}(-x^4 + 3x^2 + x); \tag{9.1}$$

Figure 9.1 shows the result of using a network of five networks, with $\theta_1 = 1, \theta_2 = 0, \theta_3 = 0.01$ and the length scales chosen from a log normal prior. If we remove the training data, we see that the variance increases to a maximum and the mean prediction tends to zero. This is demonstrated in 9.2

Next it is useful to see the effects the hyperparameters have. Consider figures 9.3 and 9.4. Figure 9.3 has a predetermined length scale of 0.01, which is too small for the data. Notice how the predictions quickly drop away from the known points towards zero (which is the prior value of the function), and the uncertainty quickly grows to a maximum away from data. In contrast, 9.4 has a length scale of 0.5. This is clearly too big – the uncertainty is too high in regions without data. Figure 9.2 shows the result of a 10-network cross-validation simulation with the length scales drawn from a prior. The performance is clearly better. Based on the data given, the network is able to determine how the data varies with time, and determined an appropriate rate at which to become uncertain about the prediction as we move further from the known points.

9.2 Learning The Inverse

Figure 9.5 shows the error after having seen n random inputs. While the convergance is reasonably slow initially, it converges exponentially. Eventually it becomes faster than the iterative process, while achieving a tolerable level of error. (The error in the figure is simply Euclidean distance). Figure 9.6 shows a magnification of the second half of the graph. Both figures are for learning a 30x30 inverse matrix in parallel with the iterative inversion.

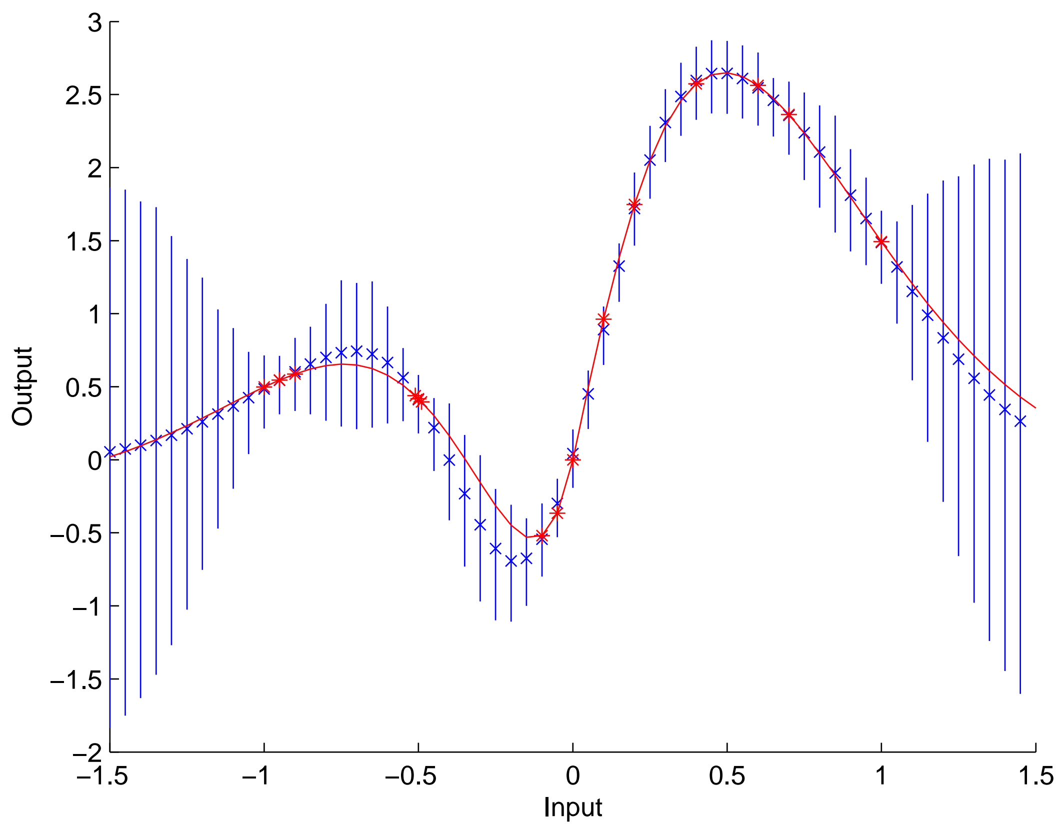


Figure 9.1: Sample prediction from a 1d function (red). The mean output is shown as a blue cross, and two standard deviations of the variance. The network was trained on the points labelled with red asterisks ([-1,-0.95,-0.9,-0.49,-0.5,-0.51,-0.1,-0.05,0,0.1,0.2,0.4,0.6,0.7,1])

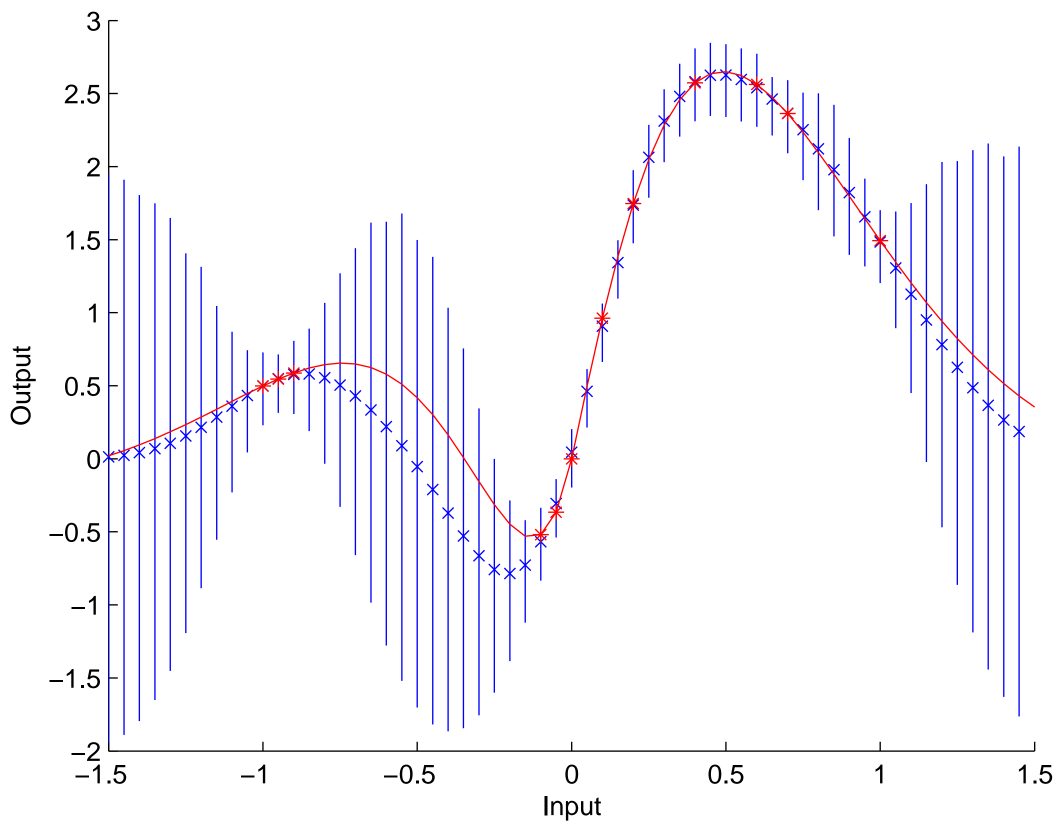


Figure 9.2: Sample prediction from the same 1d function as figure 9.1. Training examples were given at [-1,-0.95,-0.9,-0.1,-0.05,0,0.1,0.2,0.4,0.6,0.7,1] (shown as red asterisks). Notice that in the range -0.9 to -0.1 the variance increases as there are no known points

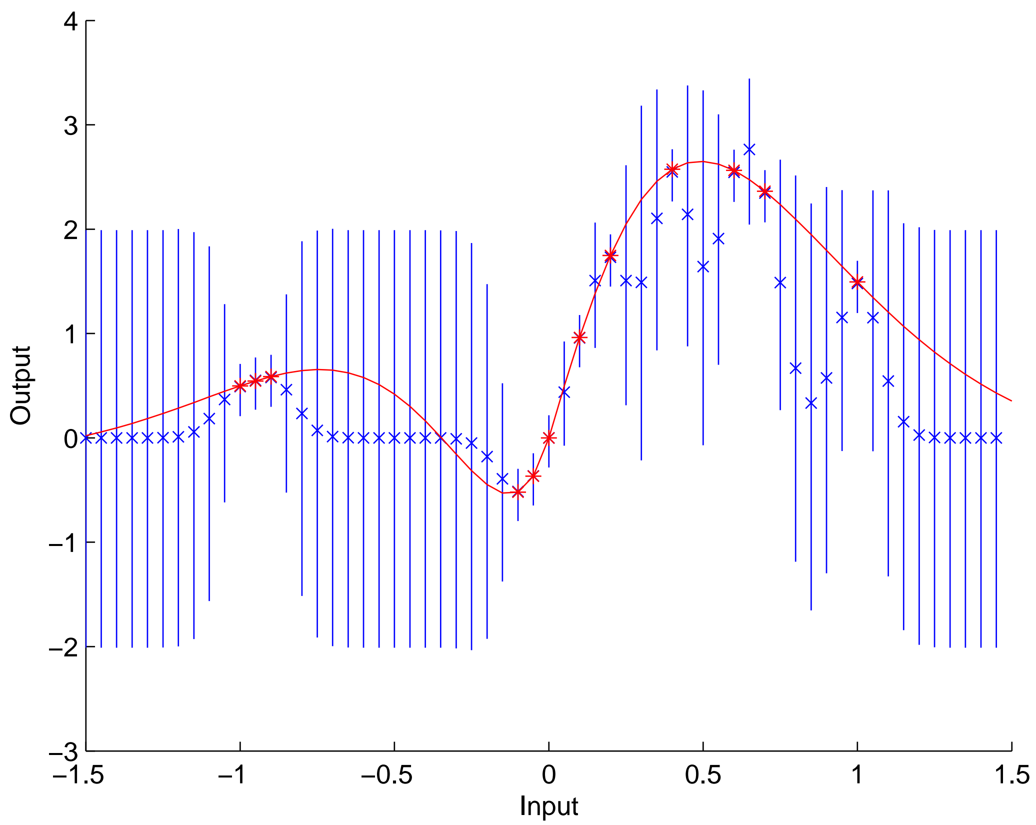


Figure 9.3: Sample prediction from the same data set as figure 9.2. The length scale was set at 0.01, which is an underestimation for the data

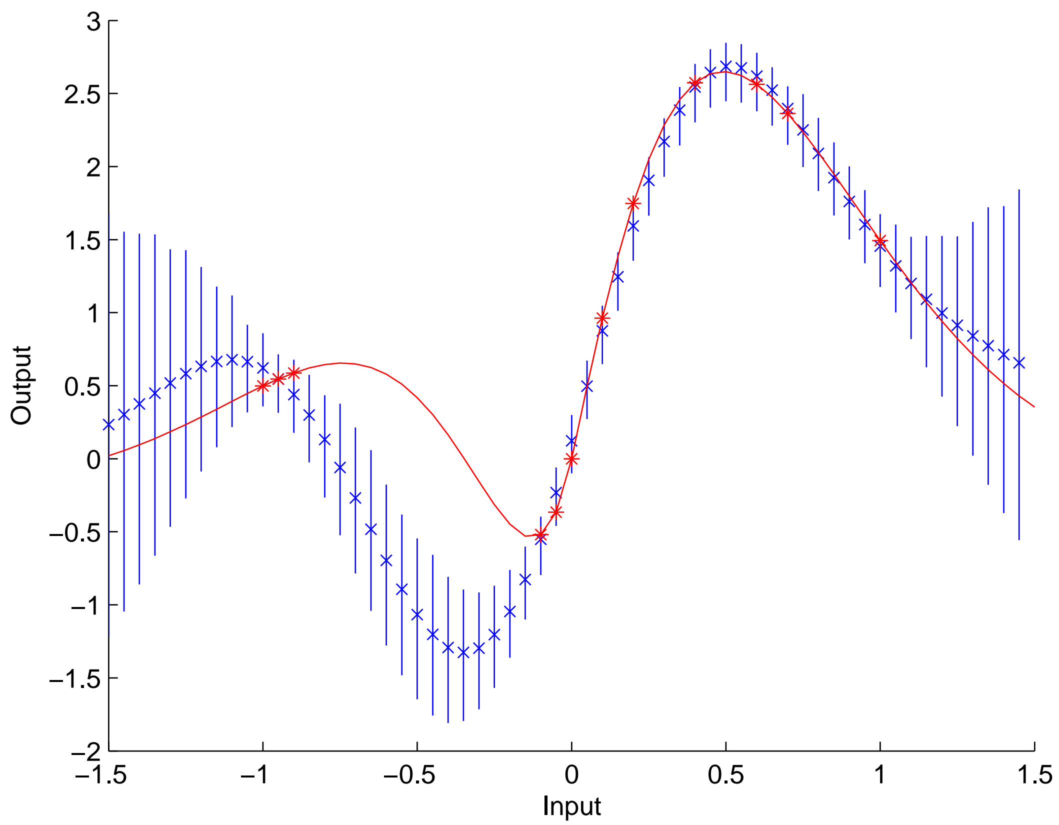


Figure 9.4: Sample prediction from the same data set as figure 9.2. The length scale was set at 0.5, which is an overestimation for the data

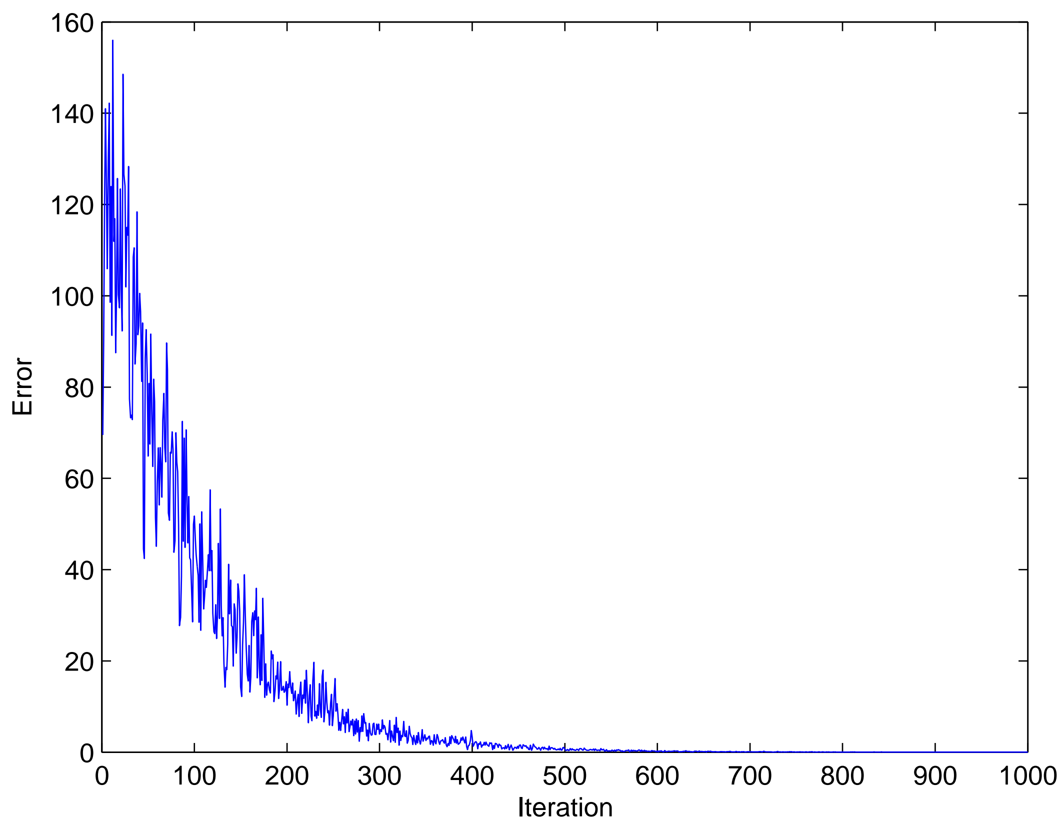


Figure 9.5: Error after n iterations for predictions made by the alternative mechanism. The matrix was randomly generated and of size 30x30

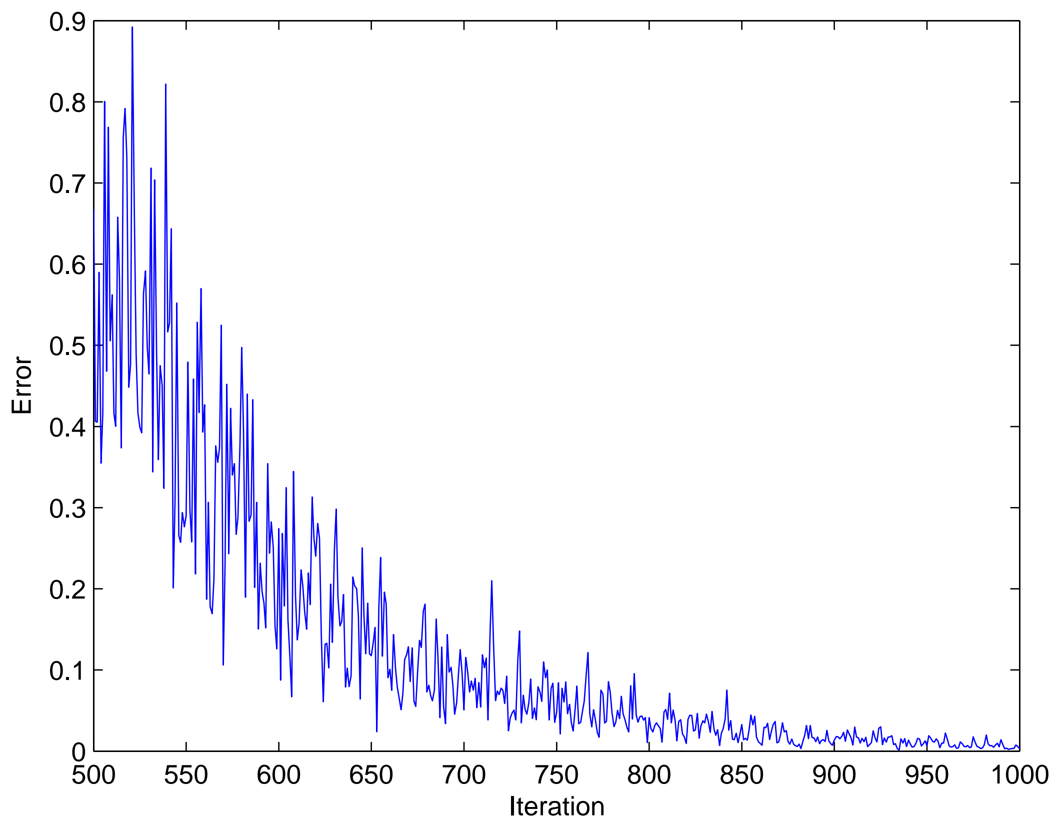


Figure 9.6: Magnification of the last 500 iterations of figure 9.5

Chapter 10

Future Work

10.1 Better Sampling

The most unrealistic part of the prediction procedure arises from the way in which the hyperparameters are sampled. One possible improvement, at least on a theoretical level, would be to use Markov Chain Monte Carlo sampling rather than Maximum A Posterori sampling to determine the 'optimal' hyperparamters.

While this alleviates the normalisation problem (see section 8.7) I am unsure of the suitability of neural networks as an architecture for sampling via such an abstract technique. However, the concept of detailed balance is prevalent in biological systems, and it is conceivable that an implementation using methods such as Gibbs or Metropolis-Hastings sampling could improve performance at the same time as increasing the overall plausibility of the system as a biological system.

10.2 Reducing the Matrix: Naïve Relevance Determination

One of the main drawbacks with using a Gaussian process prediction model is that each training example is represented explicitly in the model. A training set can easily have hundreds or thousands of data points, which can make predictions based on these training sets intolerably slow. In traditional implementations, the space requirement is in the form of the covariance matrix. While the space occupied on disk by the matrix is relatively small, the time taken to invert the matrix scales with N^3, so adding new examples on-line can be very costly without some form of optimisation.

There are several techniques for inverting sparse matrices which can help reduce the time required to invert a matrix, and techniques for dealing with ill-conditioned matrices (if the ratio of largest to smallest eigenvalue is large, the inversion process becomes sensitive to numerical inaccuacy).

Approaching the problem from a different viewpoint, there are techniques being researched for keeping the covariance matrix at a manageable size by treating the matrix as sparse and only recording important information required to approximate the inverse. [4] Generally, these involve complex computations outside the ability of a simple neural network. However, it is easy to see that training examples which are very similar do not contribute much to the predictive power or accuracy of the network. I propose instead the following simple algorithm to reduce the size (and therefore processing time) of the covariance matrix at the expense of accuracy. The algorithm is as follows:

1. Rather than immediately adding a network component for a new example, make a prediction for the target given the inputs

2. If the variance is high, add a network component for the example, finish.
3. If the variance is low and the target is within a predetermined distance α of the prediction, then finish.
4. Examine all the preprocessors. Remove the network components with secondary preprocessors with an output greater than a predetermined threshold β. Add a network component for the new example, finish.

The most obvious problem with this approach is that a large number of examples may be discarded if a new, erroneous example is presented. A better approach would be to only delete the strongly excited network components if the sum of all network components is below a third threshold γ. γ is trivial to calculate - it is simply the sum of all the secondary preprocessors.

Chapter 11

Conclusions

It is possible to implement Gaussian processes in a neural network architecture. I have outlined a technique for determining the hyperparameters of an isotropic Gaussian process, and suggested a possible means of improving the plausibility of the approach.

Do I really need this? Do I need more?

Appendix A

The Inversion Convergance can be Approximated as Exponential

Suppose that the initial value in the g vector is β. We know that the value of g at time x+1 is given by the old value (that is, g(x)) plus the approximated error, given by k - Cg.

$$g(x+1) = g(x) + k - Cg(x) \tag{A.1}$$

$$g(x+1) = (1-C)g(x) + k \tag{A.2}$$

The closed form of g(x) is then

$$g(x) = k\sum_{i=0}^{x-1}((1-C)^i) + \beta(1-C)^x \tag{A.3}$$

And we wish to find the gradient with respect to x of g(x). There is an immediate problem with this because a gradient is continuous while g(x) is most certainly discrete in x (and is not smooth - in fact the gradient changes sign at every discrete point). Possibly the best approach is to use a Fourier transform or similar smooth approximation. However, we can approximate a gradient as follows:

$$(\nabla g)(x) \approx g(x+1) - g(x) \tag{A.4}$$

Now, given A.3, we can give g(x+1) as

$$g(x+1) = k\sum_{i=0}^{x}((1-C)^i) + \beta(1-C)^{x+1} \tag{A.5}$$

$$= k(1-C)^x + k\sum_{i=0}^{x-1}((1-C)^i) + \beta(1-C)^{x+1} \tag{A.6}$$

$$= k(1-C)^x + (g(x) - \beta(1-C)^x) + \beta(1-C)^{x+1} \tag{A.7}$$

Giving the the difference g(x+1) - g(x) as

$$\Delta g = k(1-C)^x - \beta(1-C)^x + \beta(1-C)^{x+1} \tag{A.8}$$

Now we consider the long term behaviour. At x = ∞, the value of g is constant (assuming the algorithm has converged). This means that k - Cg = 0. Let this long-term value of g be γ. We can now write $k = \gamma C$, and substituting this into A.8 gives

$$\Delta g = \gamma C(1-C)^x - \beta(1-C)^x + \beta(1-C)^{x+1} \tag{A.9}$$

$$\Delta g = \gamma C(1-C)^x - \beta(1-C)^x + \beta(1-C)(1-C)^x \quad (A.10)$$

$$\Delta g = (\gamma C - \beta + \beta(1-C))(1-C)^x \quad (A.11)$$

$$\Delta g = (\gamma C - \beta + \beta - \beta C)(1-C)^x \quad (A.12)$$

$$\Delta g = (\gamma - \beta)C(1-C)^x \quad (A.13)$$

Thus we have that the "gradient" is proportional to the difference between the initial and long term values of g(x). Suppose now that we choose to evaluate the gradient not over a single time step, but over n steps to try and smooth the values a bit. In this case, we'll have

$$\Delta g = \left(\gamma C\left(\sum_{j=0}^{n-1}(1-C)^j\right) + \beta(1-C)^n - \beta\right)(1-C)^x \quad (A.14)$$

$$\Delta g = \left(\gamma C\left(\sum_{j=0}^{n-1}(1-C)^j\right) + \beta((1-C)^n - 1)\right)(1-C)^x \quad (A.15)$$

Now we need a quick lemma to show that this is really the same as before

$$Lemma : -x\sum_{i=0}^{n-1}(1-x)^i = (1-x)^n - 1 \quad (A.16)$$

Proceed by induction. The base case with n = 1 has x = x. The step is as follows: Assume true for n = m;

$$-x\sum_{i=0}^{m-1}(1-x)^i = (1-x)^m - 1 \quad (A.17)$$

Add $-x(1-x)^m$ to both sides, giving

$$-x\sum_{i=0}^{m-1}(1-x)^i - x(1-x)^m = (1-x)^m - 1 - x(1-x)^m \quad (A.18)$$

Factor the left hand side

$$-x\sum_{i=0}^{m}(1-x)^i = (1-x)^m - 1 - x(1-x)^m \quad (A.19)$$

Factor the right hand side

$$-x\sum_{i=0}^{m}(1-x)^i = (1-x)(1-x)^m - 1 \quad (A.20)$$

This is the m+1 case, so we're finished. Back to the proof, we get

$$\Delta g = ((-\gamma(1-C)^n - 1)) + \beta((1-C)^n - 1))(1-C)^x \quad (A.21)$$

After some rearrangement:

$$\Delta g = (\gamma - \beta)(1 - (1-C)^n)(1-C)^x \quad (A.22)$$

The gradient is approximately

$$\nabla g \approx \frac{1}{n}(\gamma - \beta)(1 - (1-C)^n)(1-C)^x \quad (A.23)$$

And we see that again, the gradient is proportional to $\gamma - \beta$.

Bibliography

[1] BERGHEN, F. V. *Constrained, Non-linear, Derivative-Free, Parallel Optimization of Continuous, High-Computing Load, Noisy Objective Functions*. PhD thesis, IRIDIA, ULB, 2004.

[2] DAUGMAN, J. Complete discrete 2-d gabor transforma by neural networks for image analysis and compression. *IEEE Trans. ASSP*, 7 (July 1988), 1169–1179.

[3] DESIMONE, R. Face-selective cells in the temporal cortex of monkeys. *Journal of Cognitive Neuroscience* (1998).

[4] GIBBS, M., AND MACKAY, D. Efficient implementation of gaussian processes.

[5] GIBBS, M. N. *Bayesian Gaussian Processes for Regression and Classification*. PhD thesis, University of Cambridge, 1997.

[6] J.Y. LETTVIN, H.R. MATURANA, W. M., AND PITTS, W. *The Mind: Biological Approaches to its Functions*. Interscience Publishers, 1968, ch. 7, pp. 233–258.

[7] KNUTH, D. E. *The Art of Computer Programming*, vol. 1. Addison Wesley, 1968.

[8] MACKAY, D. *Information Theory, Inference, and Learning Algorithms*. University Press, 2003, ch. 34, pp. 441,442.

[9] MACKAY, D. J. Introduction to gaussian processes. Tech. rep., Department of Physics, Cambridge University, 1998.

[10] MCINTOSH, H. V. *Linear Cellular Automata*. Universidad Autonoma de Puebla, 1987, ch. 9.4.

[11] MUMFORD, D. On the computational role of cortico-cortical loops. *Biological Cybernetics 66* (1992), 241–251.

[12] NEAL, R. Priors for infinite networks. Tech. rep., University of Toronto, 1994.

[13] PECE, A. Redundancy reduction of a gabor representation: a possible computational role for feedback from primary visual cortex to lateral geniculate nucleus. Physiological Laboratatry, Cambridge University.

[14] PETER DAYAN, L. A. *Theoretical Neuroscience*. Massachusetts Institute of Technology, 2001, ch. 5.9, p. 189.

[15] PETERSEN, K. The matrix cookbook. Technical University of Denmark, 2004.

[16] WEISSTEIN, E. W. Eigen decomposition theorem. `http://mathworld.wolfram.com/EigenDecompositionTheorem.html`. From MathWorld–A Wolfram Web Resource.

[17] WEISSTEIN, E. W. Positive definite matrix. http://mathworld.wolfram.com/PositiveDefiniteMatrix.html. From MathWorld–A Wolfram Web Resource.