International Journal of Neural Systems, Vol. 16, No. 5 (2006) 321–327

World Scientific
www.worldscientific.com

IMPLEMENTING GAUSSIAN PROCESS INFERENCE WITH NEURAL NETWORKS

MARCUS FREAN*, MATT LILLEY and PHILLIP BOYLE
Victoria University of Wellington, P.O. Box 600,
Wellington, New Zealand
**marcus@mcs.vuw.ac.nz*

Gaussian processes compare favourably with backpropagation neural networks as a tool for regression, and Bayesian neural networks have Gaussian process behaviour when the number of hidden neurons tends to infinity. We describe a simple recurrent neural network with connection weights trained by one-shot Hebbian learning. This network amounts to a dynamical system which relaxes to a stable state in which it generates predictions identical to those of Gaussian process regression. In effect an infinite number of hidden units in a feed-forward architecture can be replaced by a merely finite number, together with recurrent connections.

1. Introduction

Feed-forward neural networks are powerful models of broad applicability, able to capture non-linear surfaces of essentially arbitrary complexity.[1] Bayesian neural networks[2,3] provide a principled way to deal with model uncertainty in the predictions made by such networks, by explicitly treating uncertainty in model parameters (weights). In particular, Neal has shown that predictions made by neural networks approach those made by Gaussian processes as the number of hidden units tends to infinity.[4] In many respects Gaussian processes provide a replacement for supervised neural networks.[5]

In this paper we connect these two approaches in a different way by showing that finite but recurrent neural networks can implement Gaussian process regression directly. While we don't claim this is a sensible way to carry out regression in an engineering context, it is interesting that such a process could conceivably be implemented by biological neurons.

1.1. *Gaussian processes*

Suppose we are given training data D consisting of input patterns $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$, each of which is a vector, paired with their associated scalar output values $\mathbf{y} = \{\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_n}\}$. Here we very briefly review the Gaussian process approach to predicting y_{n+1} given some novel input $\mathbf{x_{n+1}}$, omitting all details other than the resulting algorithm (see Refs. 3, 5–7 for details).

Gaussian processes make predictions in a way that is fundamentally different to feed-forward networks. Rather than capturing regularities in the training data via a set of representative weights, they apply Bayesian inference to explicitly compute a posterior distribution over possible output values t given all the data *and* the new input x. This process involves $\mathbf{C}$, a covariance matrix generated using a covariance function $Cov(x, x'; \Theta)$ where Θ are hyperparameters. Although a variety of other alternatives are possible,[7] a typical form for the covariance function is

$$Cov(\mathbf{x}, \mathbf{x}') = \theta_\mathbf{1} \exp\left(-\frac{(\mathbf{x}-\mathbf{x}')^\mathbf{2}}{\mathbf{2}\theta_\mathbf{2}^\mathbf{2}}\right) + \theta_\mathbf{3}\, \delta_{\mathbf{x},\mathbf{x}'}$$

θ_1 determines the relative scale of the noise in comparison with the data. θ_2 characterises the distance in x over which y is expected to vary significantly.

θ_3 models white noise in measurements and δ is the delta function.[a]

The covariance matrix determines the scale and orientation of a Gaussian distribution amongst the variables $\mathbf{y}$. The task of regression is to find the distribution $P(y|D, \mathbf{x}, \mathbf{C}, \Theta)$, conditioning on the n input-output pairs corresponding to the training data (D), together with the new input $\mathbf{x}$. For a Gaussian process this conditioning process can be done analytically, resulting in a 1-dimensional Gaussian distribution characterised by the following mean m and variance $\sigma^{\mathbf{2}}$ (see e.g. Ref. 6 for a derivation):

$$m = \mathbf{k^T C^{-1} y} \quad \sigma^{\mathbf{2}} = \kappa - \mathbf{k^T C^{-1} k}. \tag{1}$$

Here $\mathbf{C_{ij}} = Cov(\mathbf{x_i}, \mathbf{x_j})$ and $\mathbf{k}$ is the vector of individual covariances $\mathrm{k_j} = Cov(\mathbf{x_j}, \mathbf{x})$ between the new input $\mathbf{x}$ and each of those in the data set. κ is $Cov(x, x)$, a constant for stationary convariance functions. Here, $\kappa = \theta_1 + \theta_3$.

2. The Daugman Algorithm

Notice that both m and σ^2 involve the vector $\mathbf{k^T C^{-1}}$: matrix inversion is a fundamental part of Gaussian process inference. Accordingly we begin by describing an existing approach to matrix inversion in a neural network. In later sections we adapt the resulting network architecture significantly and apply it to our specific case.

Consider a layer of neurons with real valued activity levels $\mathbf{g} = \{\mathbf{g_i} : \mathbf{i} = \mathbf{1} \ldots \mathbf{m}\}$, which projects to a second layer of neurons $\mathbf{k} = \{\mathbf{k_j} : \mathbf{j} = \mathbf{1} \ldots \mathbf{n}\}$ via a weight matrix $\mathbf{W}$, so that $\mathbf{k} = \mathbf{Wg}$. Now suppose that we know $\mathbf{k}$ and wish to infer the vector $\mathbf{g}$ most likely to have generated it. Clearly $\mathbf{g} = \mathbf{W^{-1} k}$, but the process of inverting a weights matrix is computationally intensive. Daugman[8] has described a method by which simple linear neurons could find $\mathbf{g}$ *without* explicitly inverting $\mathbf{W}$. Essentially the idea is that we can take an initial guess at $\mathbf{g}$ and improve it iteratively. Any given $\mathbf{g}$ generates a "prediction" $\mathbf{k}^\star = \mathbf{Wg}$. If $\mathbf{k}^\star \neq \mathbf{k}$ we have an error and can consider its quadratic cost

$$E = \frac{1}{2}(\mathbf{k} - \mathbf{k}^\star)^{\mathbf{T}} \cdot (\mathbf{k} - \mathbf{k}^\star)$$

The gradient and curvature of this with respect to $\mathbf{g}$ are

$$\nabla_g E = -\mathbf{W^T}(\mathbf{k} - \mathbf{k}^\star) \quad \nabla^{\mathbf{2}}_{\mathbf{g}} \mathbf{E} = \mathbf{W^T W}$$

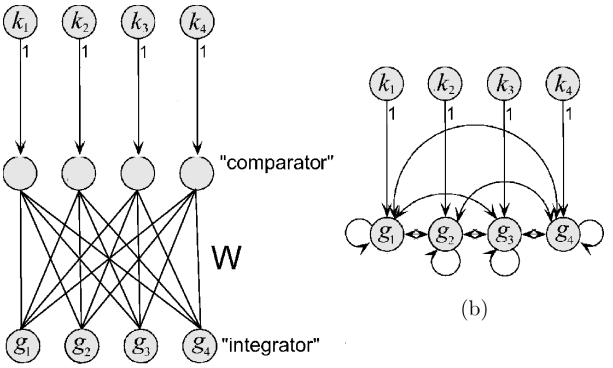

Fig. 1. Network architectures for matrix inversion. (a) Shows the network structure suggested by Daugman (with $\mathbf{A} = \mathbf{W^T}$). (b) Shows the even simpler architecture that is possible where $\mathbf{k}$ and $\mathbf{g}$ have the same dimensionality. In both cases the recurrent weights from g have an additive but inhibitory effect whereas the feed-forward input from k is additive and excitatory. Both networks "relax" into a stable state in which $\mathbf{g} = \mathbf{k^T W^{-1}}$.

So for example if we require that $W_{ij} > 0$ for all (i, j) the curvature with respect to $\mathbf{g}$ is positive everywhere, implying a single minimum. Gradient descent can therefore be used to find this minimum for a suitable choice of η (discussed further in Sec. 1) via the following update rule

$$\Delta\mathbf{g} = \eta \mathbf{W^T}(\mathbf{k} - \mathbf{Wg}) \tag{2}$$

We shall refer to this as the Daugman algorithm. It was originally proposed as a neural architecture for faithful recoding of sensory input data by the visual cortex[8,?]: $\mathbf{k}$ is the "input" to the system (such as activity in thalamus) and $\mathbf{g}$ is an alternative representation of that input (such as activity in area V1). Figure 1(a) shows the neural architecture of such a network. Implementing Eq. (2) requires that $\mathbf{k}$ provide fixed input to a "comparator" layer that finds $\mathbf{k} - \mathbf{Wg}$ and in turn provides this as input via a weights matrix $\mathbf{W^T}$ to an "integrator" layer that accumulates $\mathbf{g}$. The dynamics improve the match between $\mathbf{k}^\star$ and $\mathbf{k}$, until eventually $\mathbf{k} - \mathbf{k}^\star = \mathbf{0}$ and no more corrections are fed forward to $\mathbf{g}$.

2.1. *A special case*

A drawback of the above algorithm in terms of biological plausibility is that it requires the weights

[a]Requiring that θ_3 be non-zero also effectively prevents $\mathbf{C}$ being ill-conditioned.

matrix $\mathbf{W}$ to be used "in both directions", since the difference $\mathbf{k} - \mathbf{k}^\star$ is transformed back through $\mathbf{W^T}$ to arrive at an update for $\mathbf{g}$. However if $m = n$ this can be avoided, as follows. Consider the alternative update rule

$$\Delta \mathbf{g} = \eta(\mathbf{k} - \mathbf{W}\mathbf{g}) \tag{3}$$

which has the same fixed point as Eq. (2), namely $\mathbf{g} = \mathbf{W^{-1}k}$. Integrating the negative of this gives the corresponding cost function[6]

$$E' = \mathbf{g^T} \cdot \left(\mathbf{k} - \frac{\mathbf{1}}{\mathbf{2}}\mathbf{W}\mathbf{g}\right)$$

for which

$$\nabla_g E' = -(\mathbf{k} - \mathbf{k}^\star) \quad \nabla^{\mathbf{2}}_{\mathbf{g}}\mathbf{E}' = \mathbf{W}$$

Hence the gradient of E' is zero at the solution, and again its curvature is positive everywhere for $W_{ij} \geq 0$. One way to implement such changes is to feed the difference calculated in the "comparator" layer of Fig. 1(a) directly to layer g via the identity matrix instead of $\mathbf{W^T}$. However an interesting feature of this choice is that it permits us to avoid using the comparator layer of neurons altogether. The architecture shown in Fig. 1(b) achieves the same computation by combining comparator and integrator in a single layer with inhibitory recurrent connections $\mathbf{W}$.

For the remainder of this paper we restrict our discussion to the algorithm given by Eq. (3).

2.2. *Convergence for the special case*

Here we confirm that Eq. (3) converges on the exact solution. In doing so we are also able to derive an optimal value for η. Assume $\mathbf{W}$ is symmetric and consists of positive real numbers, and is therefore positive definite.

Assume that $\mathbf{g}$ at time zero is $\mathbf{0}$. At time t, $\mathbf{g_t} = \mathbf{g_{t-1}} + \mathbf{k^T} - \mathbf{W}\mathbf{g_{t-1}}$ which is $\mathbf{g_{t-1}}(\mathbf{I} - \mathbf{W}) + \mathbf{k^T}$. The closed form for $\mathbf{g}$ at time t is then

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

Multiplying both sides by $(\mathbf{I} - \mathbf{W})$, subtracting from $\mathbf{g(t)}$, and right-multipling by $\mathbf{W^{-1}}$ yields (see Ref. 14 for details)

$$\mathbf{g(t)} = \mathbf{k^T}\big(\mathbf{I} - (\mathbf{I} - \mathbf{W})^{\mathbf{t}}\big)\mathbf{W}^{-1} \tag{4}$$

Consider the term $(\mathbf{I} - \mathbf{W})^{\mathbf{t}}$ in the limit as $t \to \infty$. Making use of the eigen-decomposition theorem[10] we can rewrite $\mathbf{I} - \mathbf{W}$ in terms of the matrix D which has the eigenvalues of $\mathbf{I} - \mathbf{W}$ along its diagonal so that $\mathbf{I} - \mathbf{W} = \mathbf{P^{-1}\,DP}$, and since $(P^{-1}\,DP)^z = P^{-1}\,D^z\,P$ all that remains is to show that

$$\lim_{z \to \infty} P^{-1}\,D^z\,P \tag{5}$$

is defined and finite. Because D is diagonal, $[D^z]_{ij} = [D]^z_{ij}$ and so we conclude that if all eigenvalues λ_i of $\mathbf{I} - \mathbf{W}$ have absolute magnitude less than one then the above limit is simply the zero matrix. Otherwise, the limit is infinite, and therefore the algorithm fails. To summarise, provided $|\lambda_i| < 1$ for all i, we have $\lim_{z\to\infty}(\mathbf{I} - \mathbf{W})^{\mathbf{z}} = \mathbf{0}$, and $\mathbf{g(t)}$ converges to $\mathbf{g(t)} = \mathbf{k^T W^{-1}}$ as $t \to \infty$.

We can enforce the condition $|\lambda_i| < 1$ by introducing a parameter, η, as follows. If $\mathbf{g_t} = \mathbf{g_{t-1}} + \eta\mathbf{k^T} - \eta\mathbf{W}\mathbf{g_{t-1}}$ then by a similar process to which equation 4 was derived, we have

$$\mathbf{g(t)} = \eta\mathbf{k^T}\big(\mathbf{I} - (\mathbf{I} - \eta\mathbf{W})^{\mathbf{t}}\big)(\eta\mathbf{W})^{-\mathbf{1}} \tag{6}$$

If we choose η such that the eigenvalues of $\mathbf{I} - \eta\mathbf{W}$ are of magnitude less than one, then Eq. (5) will converge, and therefore ultimately Eq. (6) will converge also. It is an identity that the eigenvalues λ of $\mathbf{I} + \eta\mathbf{M}$ are $1 + \eta\lambda$,[11] and the eigenvalues of a positive definite matrix are all positive or zero,[12] therefore by letting η be $-\frac{1}{\max \lambda}$, we guarantee that all eigenvalues of $\mathbf{I} - \mathbf{W}$ are of magnitude equal to or less than one.

3. Solving the Regression Problem Using a Dynamical System

We can use the architecture shown in Fig. 1(b) to solve the Gaussian process prediction Eq. (1). Firstly, notice that the vector $\mathbf{k}$ can be thought of as the output of a layer of radial basis function (RBF) units, given input pattern $\mathbf{x}$. In the RBF networks literature there are a number of fairly standard ways to set the number of hidden units and to place them in the input space. One popular option is to have n of them (one for every input pattern in the training set), and to place their centers on the corresponding input patterns. A very common choice for the receptive field in RBF networks is the spherical Gaussian:

$$k_i(\mathbf{x}) = \alpha \exp\left(-\frac{(\mathbf{x} - \mathbf{x^{(i)}})^{\mathbf{2}}}{\mathbf{2}\beta^{\mathbf{2}}}\right)$$

where $\mathbf{x}$ is the novel input pattern and $\mathbf{x^{(i)}}$ is the i-th input vector. Thus we can think of the vector $\mathbf{k}$ required for Gaussian process inference as being the output of a particular choice of RBF units, with $\alpha = \theta_1$ and $\beta = \theta_2$.

At this point one could come up with an algorithm such as gradient descent for mapping $\mathbf{k}(\mathbf{x})$ onto outputs via a layer of weighted connections, exactly as is done in RBF networks. Indeed the mean output predicted by a Gaussian process model is $\mathbf{k^T C^{-1} y}$, and so if the output weights are set to $\mathbf{C^{-1} y}$ such a network will output the correct GP prediction for any input. The question then becomes: how might such weights be arrived at? The primary task appears at first to be inversion of $\mathbf{C}$, but it is sufficient to find $\mathbf{k^T C^{-1}}$ and then to take the dot product with targets $\mathbf{y}$. The first part of this can be done by the dynamics of the network described in the previous section, simply by inserting $\mathbf{C}$ for the weight matrix $\mathbf{W}$.

To recap: connections from $\mathbf{k}$ to $\mathbf{g}$ have weights of 1, recurrent connections have weights $-\mathbf{C_{ij}}$, and those from $\mathbf{g}$ to the output m have weights $\mathbf{y}$. The $\mathbf{g}$ neurons take a linear sum of their input and add it to their current level of activity, which can start at any level.

Once this network converges we have only to take the dot product with the vector of targets (Eq. (1)), which is easily achieved via a second layer of weights whose values are set to their respective target outputs. Figure 2 shows the whole system.

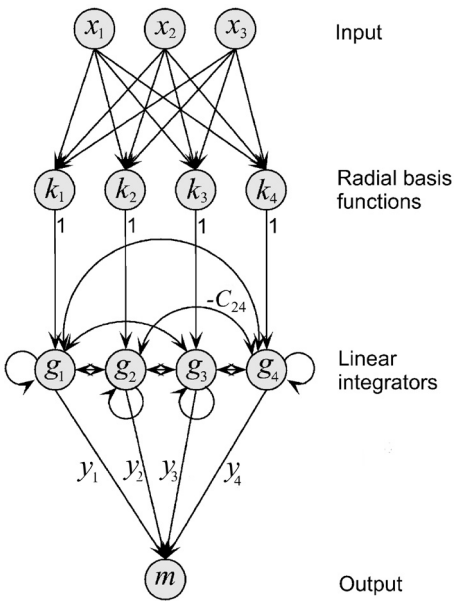

Fig. 2. A network architecture that implements Gaussian process regression. It converges on the mean m of the predicted output distribution, given input $\mathbf{x}$. In this particular case the input is a 3-dimensional vector and inference is carried out on the basis of 4 input-output pairs.

3.1. *Ensuring convergence*

Earlier we showed that convergence is guaranteed provided $\eta \leq \frac{1}{\lambda_{\max}}$ where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{I} - \eta\mathbf{C}$. The largest eigenvalue of $\mathbf{C}$ is strictly less than the maximal row sum (for a symmetric matrix),[13] which in turn is bounded by $n(\theta_1 + \theta_3)$, for a matrix having n columns.

$$\eta_{estimate} = |N(\theta_1 + \theta_3) + 1|^{-1} \tag{7}$$

Equation (7) gives a tractable way to approximate an appropriate value of η. Empirical evidence suggests that using the estimate described in Eq. (7) indeed has similar performance to using the inverse of the largest eigenvalue of $\mathbf{I} - \mathbf{C}$, which appears to be optimal.[14]

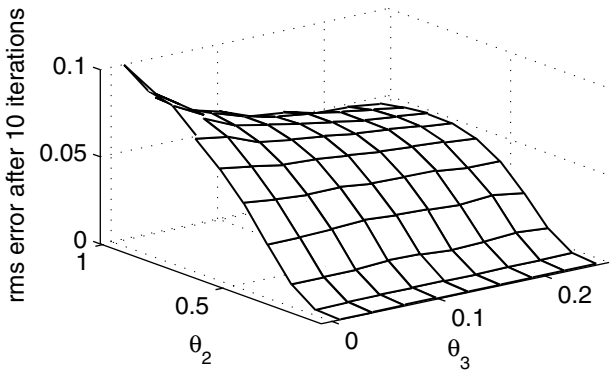

Fig. 3. The rms error between g and $\mathrm{k^T C^{-1}}$ after 10 iterations of the dynamics. We used 100 data points chosen at random from within a 10-dimensional hypercube. θ_1 was fixed at 2.0 and the convergence rate for various values of θ_2 and θ_3 explored. Each vertex is an average over 100 independent runs. The rate parameter η was set to the value derived in the text.

The rate of convergence depends on the choices for hyperparameters, as shown in Fig. 3 for a representative range of outcomes. θ_2 plays a crucial role, as it effectively determines the expected number of datapoints involved in each prediction. If θ_2 is small

then the new input is unlikely to be close to any previous data and therefore $\mathbf{k} \approx \mathbf{0}$, or it may be close to just one previous input, in which case only one element of $\mathbf{k}$ is significantly non-zero. Larger values of θ_2 make both $\mathbf{k}$ and $\mathbf{C}$ less sparse, and intuitively one can see that this will require more iterations to take account of the corresponding interrelationships.

3.2. *Uncertainty in predictions*

The variance of the prediction is given by the expression $\mathbf{k^T C^{-1} k}$. Part of this ($\mathbf{k^T C^{-1}}$) has already been calculated by the network which determines the mean, so we can reuse the previous calculation and essentially get the variance for (almost) no added cost. All that remains is to find the dot product of the $\mathbf{g}$ vector with $\mathbf{k}$. Initially this seems like a trivial problem, but from a biological perspective it poses some difficulty. A possible mechanism is suggested by the process known as *shunting inhibition*, proposed as a possible mechanism for neurons to divide numbers[15] in which the output of one neuron inhibits the transmission of charge between two other neurons. As the elements of $\mathbf{k}$ are between 0 and 1, computing $\mathbf{k^T C^{-1} k}$ can be considered to be scaling the elements of $\mathbf{k^T C^{-1}}$ by the elements of $\mathbf{k}$, a task to which shunting inhibition seems ideally suited. Against this, some precise wiring is now required, as the i-th $\mathbf{k}$ neuron must gate the effect of the i-th $\mathbf{g}$ neuron on the output.

Figure 4 shows a small 1-dimensional example for illustrative purposes.

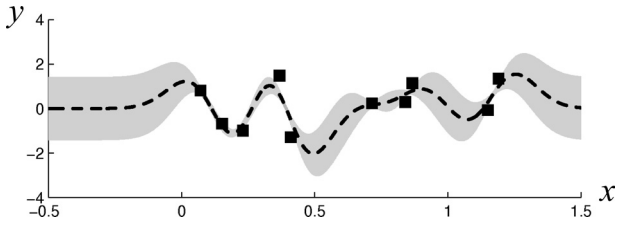

Fig. 4. Example of Gaussian process inference. Predictions for y were then made by running the dynamics for 100 iterations, for a range of inputs, x. There are 10 data points (squares). The dashed line is the mean prediction calculated as described in the text, and vertical bars indicate one standard deviation of the uncertainty.

4. Construction and Learning

The various connections in this system need to be set to particular values in order for this procedure to work. Firstly, the RBF units must each be centered on a unique input pattern. One can imagine a constructive process in which a novel input pattern $\mathbf{x_{n+1}}$ triggers the recruitment of a new cell whose output is, and remains, maximal for that pattern. By thinking of the network in this constructive manner it is also clear how the other connections might be set: another new cell g_{n+1} is similarly recruited, and receives input from $\mathbf{x_{n+1}}$ with a synaptic weight of one. Its weights both to and from any other $\mathbf{g}$ cell, say g_i, need to be $-Cov(\mathbf{x_i}, \mathbf{x_{n+1}})$, which is simply the value taken by k_i for the current input. Indeed this is locally available as the instantaneous[b] value of g_i, amounting to a form of (anti) Hebbian learning.[16]

Note that the self-weights need to be set slightly higher, to $\theta_1+\theta_3$, whereas the above procedure would make them just θ_1. The additive term is required to ensure that the weight incorporates the correct diagonal term as given in Eq. (1).

Finally the synaptic weight from g_{n+1} to the "output" must be set to y_{n+1}, which we may assume is the output cell's current value. In this way a network is both constructed and "learned" by local mechanisms as input-output pairs are presented to it.

5. Discussion

Traditional feed-forward neural networks might be characterised as slow to learn but fast to react. Learning amounts to a complex non-linear recoding of the input-output mapping indicated by the training data into the weight values, and typically takes many iterations of a weight-updating algorithm. One consequence of this is the threat of "catastrophic forgetting"[17,18] if online learning is attempted: if weights are updated in the light of a new training example, the changes are liable to wreak havoc on the existing mapping and distort the network's response to earlier training items. In many cases one is forced to retrain on the entire expanded set of examples despite the obvious appeal of online learning. on the other hand, once trained feed-forward neural nets can calculate the expected output y given any novel

[b] i.e. the value g_i takes, prior to being perturbed by the recurrent neural dynamics.

input vector $\mathbf{x}$ very quickly. Loosely speaking, when a novel input arrives, all the important work involved in prediction has already been done via the learning process.

By contrast, the recurrent neural network described here is fast to train but slow to react. So-called "training" amounts only to the setting of connection weights by one-shot Hebbian learning[16] carried out at the moment the training example is presented. No iterative learning process is carried out. Notice that its incremental construction requires no adjustment to earlier weights when a new training example is incorportated. In this way catastrophic forgetting is automatically avoided. Against this, such an architecture is slow in making predictions due to the iterative process by which it arrives at the solution.

There are two reasons why this is less of a problem than might be imagined. Firstly, it can be shown accuracy improves exponentially with time[14]: early results are rough, but improve rapidly and eventually become exact. Secondly, there are at least two ways to use results from the "slow" iterative algorithm to generate targets with which to learn a second "fast" network, as follows.

Firstly notice that $\mathbf{g}$ is a linear function of $\mathbf{k}$, so we can compute $\mathbf{g} = \mathbf{k^T C^{-1}}$ from $\mathbf{k}$ as indicated and use this as a target for a second set of weights between layers k and g, thus learning weights corresponding to $\mathbf{C^{-1}}$. Given $\mathbf{k}$ this secondary network can then directly compute $\mathbf{k^T C^{-1}}$ in a single pass instead of by iteration — see Fig. 5(a). We have shown experimentally and analytically that this process converges exponentially quickly to the correct solution.[14]

Alternatively, since we have a method for generating Gaussian process predictions m given arbitrary input $\mathbf{x}$, unlimited "training items" consisting of random $\mathbf{x}$'s and their predicted outputs t can be generated. The mapping from $\mathbf{k}$ to m is linear, and so a layer of linear weights directly from the k layer to m can be learned, as indicated in Fig. 5(b).

We have adapted an algorithm suggested by Daugman, using it to as a way to produce on-the-fly "inversions" of the covariance matrix, in effect. The addition of radial basis function nodes as inputs, and output weights set to the original targets, permits us to generate the same outputs as the expected values of a Gaussian process. Although some of the appeal of this idea is lost due to the exact wiring required in estimating the *uncertainty* in such predictions, the dynamical system embodied by our network is no less plausible as a biological metaphor than other commonly cited mechanisms, such as the backpropagation algorithm. Our method has the added benefits of a sound theoretical basis in Bayesian statistics, rapid (one-shot) learning using the Hebb rule, and guaranteed convergence.

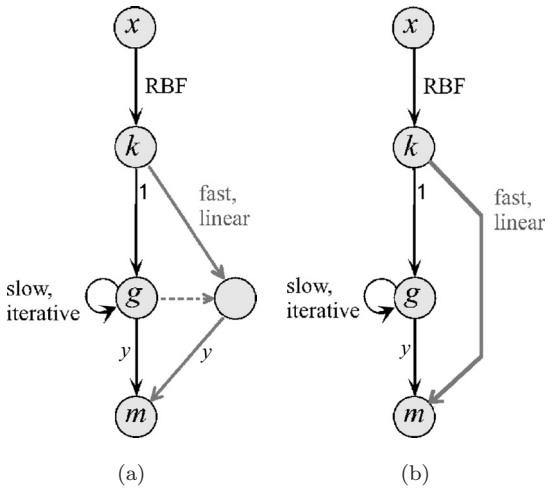

Fig. 5. Two possible ways to generate very fast responses that learn to match those given by the iterative algorithm. Solid lines schematically represent entire weights matrices, namely those shown in full in Fig. 2. (a) Converged values of $\mathbf{g}$ can be considered as "targets" conveyed (dashed line) to new neurons wired "in parallel" to the g layer. These then project to the output with the same outputs weights as before (namely the targets y). (b) Alternatively a direct mapping from $\mathbf{k}$ to the output can be learned, by using targets generated by the dynamical network's responses to *random* inputs.

References

1. S. Haykin, *Neural Networks: A Comprehensive Foundation* (Prentice Hall; 2nd edn., 1998).
2. R. M. Neal, *Bayesian Learning for Neural Networks* (Lecture Notes in Statistics No. 118, New York: Springer-Verlag, 1996).
3. D. MacKay, *Information Theory, Inference, and Learning Algorithms* (Cambridge University Press, 2003, ch. 34).
4. R. M. Neal, Priors for infinite networks, Tech. rep., University of Toronto (1994).
5. D. J. MacKay, Gaussian processes — a replacement for supervised neural networks?, Lecture notes for a tutorial at NIPS (1997). http://www.inference.phy.cam.ac.uk/mackay/gpB.pdf

6. M. N. Gibbs, Bayesian Gaussian processes for regression and classification, PhD thesis, University of Cambridge (1997).
7. C. K. I. Williams and C. E. Rasmussen, Gaussian processes for regression, *Advances in Neural Information Processing Systems*, **8**(1996) 514–520.
8. J. Daugman, Complete Discrete 2-D Gabor Transforms by Neural Networks for Image Analysis and Compression, *IEEE Trans. ASSP*, **36**(7) (1988) 1169–1179
9. A. E. C. Pece, Redundancy reduction of a Gabor representation: a possible computational role for feedback from primary visual cortex to lateral geniculate nucleus, Unpublished manuscript (1993).
10. E. W. Weisstein, Eigen decomposition theorem. http://mathworld.wolfram.com/EigenDecompositionTheorem.html.
11. K. Petersen, The matrix cookbook, Technical University of Denmark (2004). http://2302.dk/uni/matrixcookbook.html
12. E. W. Weisstein, Positive definite matrix. http://mathworld.wolfram.com/PositiveDefiniteMatrix.html.
13. H. V. McIntosh, *Linear Cellular Automata*. (Universidad Autonoma de Puebla, 1987, ch. 9.4).
14. M. Lilley, Gaussian processes as neural networks, Honours thesis, Victoria University of Wellington (2004). Available from http://www.mcs.vuw.ac.nz/people/Marcus-Frean
15. P. Dayan and L. Abbott, *Theoretical Neuroscience*. (Massachusetts Institute of Technology, 2001), p. 189.
16. D. O. Hebb, *The Organisation of Behaviour* (Wiley, New York, 1949).
17. M. McCloskey and N. J. Cohen, Catastrophic Interference in conectionist networks: the sequential learning problem, *The Psychology of Learning and Motivation*, **23** (ed. G. H. Bower) (New York: Academic) (1989), pp. 109–164.
18. M. R. Frean, and A. V. Robins, Catastrophic forgetting in simple networks: An analysis of the pseudorehearsal solution, *Network: Computation in Neural Systems* **10** (1999) 227–236.
19. J. J. Hopfield, Neurons with graded response have collective computational properties like two-state neurons, in *Proc. Natl. Acad. Sci.*, **81** (1984) 3088–3092.
20. D. W. Tank and J. J. Hopfield, Simple 'neural' optimization networks: An A/D converter, signal decision circuit, and a linear programming circuit, *IEEE Trans. on Circuits and Systems*, **33** (1986) pp. 533–541.
21. Y. S. Foo and Y. Takefuji, Integer linear programming neural networks for job-shop scheduling, in *Proc. IEEE Intl. Conf. on Neural Networks*, **II**, (1988) pp. 341–348.