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 4 471 5328
Fax: +64 4 495 5232
Internet: office@mcs.vuw.ac.nz

Biologically Plausible Back-Propagation

Tim Robinson

Supervisor: Marcus Frean

October 28, 2000

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

Abstract

Backpropagation was the first algorithm to enable general feed-forward neural networks to learn complex tasks. However, some aspects of the algorithm are not biologically plausible. This project investigates a new algorithm that approximates backpropagation in a more biologically plausible manner. This algorithm performs similarly to backpropagation on real world problems, and scales well to networks with a large number of hidden units.

Contents

1 Introduction

artificial adj 1: contrived by art rather than nature

intelligence n 1: the ability to comprehend; to understand and profit from experience

– Wordnet 1.6, ©1997 Princeton University

When learning about something, it is natural to look for parallels in nature. Such is the case in artificial intelligence. Ultimately, we seek to endow a machine with the seemings, if not the actual ability, to behave as a human. Machine learning is a branch of artificial intelligence that attempts to give a machine the capability to learn, to 'understand and profit from experience'. As ever, we can attempt to simulate nature's methods. Artificial Neural Networks (ANNs) are the result of researchers attempting to model the structure of the human brain — with its neurons and synapses — in machine form. We are particularly interested in the brain as a collection of a vast number of parallel units with localised operations [1].

In the brain, a neuron is either 'off' or 'on'. The process of turning 'on' is known as 'firing'. Neurons are connected to other neurons by way of one-way synapses of variable strength. These synapses can influence the behaviour of the neurons they connect to. When a neuron 'fires', this fact is registered by any neuron which the fired neuron has a synapse going *to*. If enough of the neurons that are connected *to* a particular neuron fire at once, and their 'strength' is large enough, then that neuron will fire also. The value of 'enough' varies from neuron to neuron, and can be thought of as the 'firing threshold' for that neuron.

The first explicit formulation of an *artificial* neural network was Rosenblatt's perceptron[2]. Rosenblatt proposed a system which was self-organising (to a degree), and in which random factors played a part. As originally proposed, the perceptron was more brain-like than the standard model of subsequent years. This formulation, put forward by Widrow and Hoff[3] involves a network of logical threshold units with variable connection strengths or 'weights'. Each unit performs a sum of the inputs times their synaptic weight, plus a bias input[1]. If the sum is greater than zero, the output value is 1. If the sum is zero or less, the output is -1.

Research into these simple logical units continued, but interest in the field began to wane in the mid-sixties, as limitations of the threshold model became obvious. The most famous limitation was that of exclusive logical or (XOR)[2]. The problem could be solved using a very simple network, but no known algorithm could learn the problem from scratch. Minsky and Papert summed up the general feeling in their 1969 book *Perceptrons* [4], and concluded that they felt that there never would be such an algorithm.

Interest in the field continued to wane until in 1986, Rumelhart, Hinton and Williams derived an algorithm named Backpropagation[5] that addressed some of Minsky and Papert's criticisms. It was a gradient descent algorithm that worked by approximating the linear threshold of each unit with a non-linear activation function (see figure 1).

We begin with a brief discussion of the backpropagation algorithm, introduce a new algorithm to overcome some of backpropagation's deficiencies, and compare the two algorithms. We then look at the mathematical basis for the new algorithm, before moving on to explore some of its interesting features.

[1] A bias unit always has output value 1, and is connected to all non-input units. Adjusting the weight on the connection from the bias is the equivalent of adjusting the unit threshold.

[2] Two binary inputs and one binary output. 0 0 maps to 0, 0 1 maps to 1, 1 0 maps to 1, 1 1 maps to 0

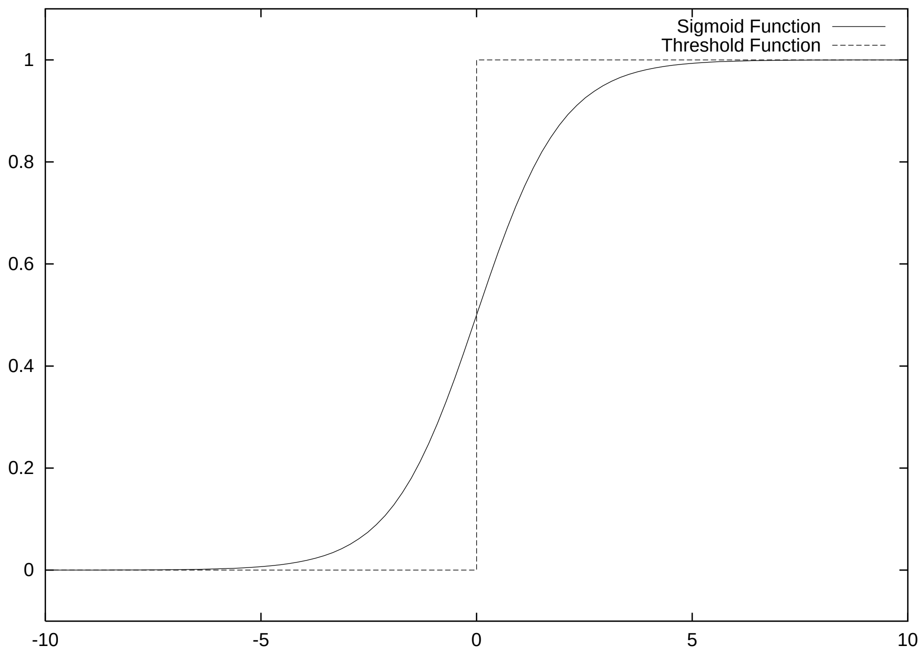

Figure 1: A threshold function and the sigmoidal approximation.

2 A Problem and a Solution

2.1 Gradient Descent

The goal of doing neural network learning (or *any* learning, for that matter) is to get the appropriate output for any given input. As seems logical, the difference between the desired output and the actual output is the *error* . For every set of weights that the network can be set to, there is a corresponding error. If we plot each weight on one axis of a graph, and the corresponding error as another axis, then we have a surface in n-space where there are n weights.

Gradient descent methods calculate the gradient of this error surface. A small step can then be taken down this gradient, and a new gradient calculation performed, repeating until no lower point can be reached. This is not guaranteed to find the global minimum, because of the possibility of 'local minima' where the gradient is zero but the error is above the global minimum.

2.2 Back Propagation of Errors

The basic problem when changing the weights in an ANN (ie learning) is to calculate the extent to which each individual neuron influences the error in the final output of the network[3]. Backpropagation[5] was the first algorithm to provide a method for general feed-forward ANNs to calculate this influence. It involves explicitly calculating the error at the output node k, and passing this error back to each unit feeding into k, with the *same weight* as is on the forward connection. The total error caused by a particular node in a network can be arrived at by summing that node's contribution to the errors further forward in the network. This is accomplished by a weighted sum over the errors at each node that the current node

[3]There are various ways of measuring the total error over the entire training set. A common error measure is the *Mean Squared Error* , the average of the squared absolute errors for each training pattern.

outputs to. The fact that the weighted connection must function in either direction with the same value is a crucial – but biologically implausible – aspect of the algorithm.

The basic idea in ANN research is to use our knowledge about the structure of the brain to accomplish useful computation. Conversely, we would like to be able to take ideas from ANNs and apply them to our understanding of the brain. When we look at back propagation in this context, we find that while being a good machine learning algorithm, it falls down as a model for learning in the brain. Synapses in the brain are strictly one-way, and while feedback in the brain is common, having two one-way synapses with the same weight is implausible. Also, under backpropagation, as the error signal arrives at each node, it is multiplied by the derivative of the activation function of that node. This is also biologically implausible.

2.3 Biologically Plausible Backpropagation

This report investigates an algorithm[6] which approximates the gradient descent performed by backpropagation in a way that is more biologically plausible. This algorithm is called **BPB** (for **B**iologically **P**lausible **B**ack-propagation).

The main idea behind BPB is that the influence of each unit on the output of the network can be estimated empirically, instead of being calculated analytically by the backpropagation algorithm.

The influence of a unit can be estimated by adding some noise to the output value of that unit, and comparing the added noise to the perturbation in the output of the network. This can be used to arrive at a good estimate of the true influence, provided the activation function used in the neurons is smooth. In that case, local linearity implies that

$$f(\phi + n) = f(\phi) + \alpha n \tag{1}$$

where n is the added noise and α is the influence on the output, namely the gradient $\frac{\partial f}{\partial \phi}$. Details of how this gives rise to BPB are given in section 4.

To allow each hidden unit to see the effect of the noise that it introduces, a backward connection is added from each output unit to each hidden unit (see figure 2). These backward links are not used in any forward pass and they do not affect the output of any hidden unit. On a forward pass through the network, each hidden unit will add a small amount of random noise to its output, then use these backward connections to estimate its contribution to the overall error in the network. The noise added at each hidden node is not correlated with noise added at any other node.

The steps involved in training a neural network using BPB are as follows (further detail can be found in section 4).

1. Present the training pattern at the inputs.

2. Perform a forward pass through the network — with no noise added — to obtain the output values.

3. Perform a forward pass through the network with each hidden unit adding noise.

4. Let each unit update the weights on its backward links according to the correlation between the noise it produced and the change in the corresponding output.

5. Repeat steps 3 and 4 a number of times in order to get a better estimate, collecting this estimate of the influence as the weight of the backward link.

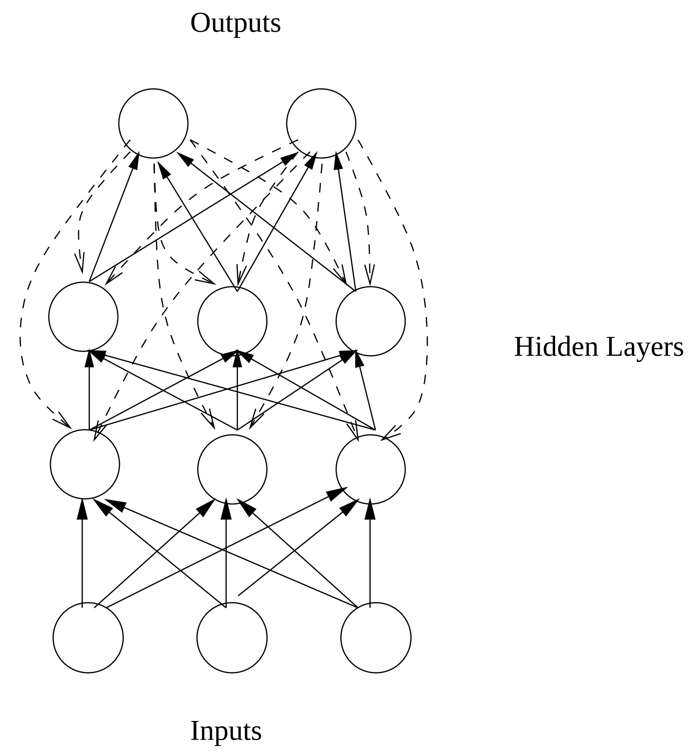

Figure 2: BPB Network Architecture — Dotted Lines are backward links.

6. At each hidden node, sum over all the backward links to get the estimate for the gradient at that node

7. Use the estimated gradient to update the regular weights in the network.

Thus, we arrive at an approximation of the true gradient, in a more biologically plausible manner than for backpropagation.

2.3.1 Parameters particular to the BPB algorithm

As an approximation to backpropagation, BPB shares many of the same parameters. However, there are two parameters particular to BPB. They are

τ The number of samples to take when accumulating the weights on backward links.

β The variance of the distribution from which the added noise is taken.

With higher τ (more samples), the estimate gets closer to the true gradient. As $\tau \to \infty$, the estimate tends to the true gradient α. This is proven in section 4

There are many distributions from which random noise values can be taken. Any distribution can be used, as long as the expected value of any sample from that distribution is zero, and the variance of the distribution is known. The simplest distribution is where noise is either $+\beta$ or $-\beta$ (this distribution is the one used in all experiments presented here), but any zero mean distribution can be used. For instance, uniform random noise between $+\beta$ and $-\beta$ could be used, or noise from a Gaussian distribution with variance β.

3 Comparing BPB to the Backpropagation algorithm

It is not intended that BPB provide better performance than back-propagation in the conventional sense (shorter training time during learning), but rather that it be a more biologically plausible learning algorithm.

3.1 Similarities between BPB and Backpropagation

Since what is being estimated is, in fact, an approximation to the gradient calculated by back propagation, the behaviour of the two algorithms is similar in many ways. They share the following properties

- Random initialisation of the weights in the network. These weights can affect how fast the network learns[7], and have the same effect for either type of network.
- A learning rate η which specifies the size of the step down the gradient.
- An activation function $f(i)$. The most common activation function is the standard sigmoid[4] (see figure 1). Another popular choice is tanh, the tangens hyperbolicus.

Various techniques for speeding up the backpropagation algorithm can equally well be applied to BPB. Not only could simple improvements such as momentum and weight decay be implemented, more complicated algorithms that use gradients such as the quickprop[8] algorithm or the conjugate gradient method could also be implemented. Any improvement that uses the gradients calculated by backpropagation can use the gradients estimated by BPB. There are some BPB-specific optimisations discussed in section 6.

Although these networks are parallel architectures, simulated learning takes place in a serial environment. Thus, in each learning phase, some units must have their input weights updated before other units. Under some implementations of backpropagation, these new weights are used to calculate the changes to weights further on in the network. While this does work, it is not mathematically correct. Under BPB, the changes to be made for every unit's input weights are calculated before any updating is performed, so BPB corresponds to the mathematically correct version of backpropagation.

3.2 BPB's advantages over Backpropagation

Learning in BPB avoids the two major biological implausibilities present in backpropagation (that is, passing errors backward between units using the weights on the forward connection, and the differentiation of activation functions), and so is more brain-like than in back propagation. BPB has some other advantages over conventional backpropagation as well.

Firstly, learning in BPB is Hebbian[9]: the more influence a unit has on the output of the network, the larger the weight on the backward-link will be. It does not involve differentiating a unit signal (as backpropagation does), and learning is instead achieved through much simpler mathematical operations. This may mean that BPB can provide us with more insight to learning in the brain than backpropagation can. It also means that BPB may be simpler than backpropagation to hard-wire in silicon.

Secondly, backpropagation is very sensitive to the initial weights in the network. If all the weights are zero, then there is no gradient to descend, nothing to propagate, and no learning happens. If some weights have the same value, backpropagation cannot break the symmetry,

[4] Which is used in all cases in the work presented here.

and learning is stymied. If the gradient is zero at any point, under backpropagation learning will stop. The randomness inherent in BPB serves to break any symmetry, and means that if ever the gradient is zero (including the case where all the weights are zero) BPB does the equivalent of a random walk around the zero gradient portion of the error surface. This is a much better procedure than just stopping dead.

Thirdly, to calculate the error in a backpropagation network, we must know the activation function for each neuron, and also its derivative. BPB requires no knowledge of the activation function. It can be used in cases where backpropagation cannot. For example, when using genetic algorithms or genetic programming for learning at each hidden unit (as in [10]), a good choice for the fitness function of a GA at a hidden node would be the error in the output of the network caused by that particular hidden node. As there is no activation function available to differentiate, we could not use backpropagation to find this error, but we could use BPB to find a good estimate. This fact also contributes to the possibility of hard-wiring BPB on a silicon chip.

3.3 Backpropagation's advantages over BPB

As stated above, BPB is not an attempt to come up with a "better learning algorithm". It is an attempt to approximate the process performed by the back propagation algorithm in a more biologically plausible manner.

- BPB adds τ noisy feed-forward passes for each training pattern. Thus, for a given number of training epochs[5], BPB uses $\tau+1$ times the number of feed-forward passes that backpropagation uses.
- Backpropagation finds the true gradient exactly.

Thus, training a network using back-propagation will take less time than training it using BPB.

3.4 Performance

Somewhat surprisingly, training a network using BPB takes on average only slightly more epochs than training it using backpropagation, even for small values of τ. Because of BPB's inherent randomness, it can sometimes reach the specified error *before* backpropagation does. Figure 3 shows a typical training run.

Throughout this project I used the datasets in the PROBEN1 benchmark set[6] collated by Lutz Prechelt[4]. This training setup uses the cancer data[7] from that collection.

The network used had two hidden layers, containing four and two units respectively, with all feed-forward connections between adjacent layers. Weights were randomly initialised to a value between -0.5 and 0.5. The learning rate was 0.01, τ 4, and noise was ± 0.001. The network was trained until the MSE on the training data was less than 0.04.

	Average of 50 Training Runs	Standard Deviation
Backpropagation	146.71	21.14
BPB	147.08	21.05

[5] An 'epoch' is one presentation of each pattern in the training set

[6] I also used the XOR problem in some cases

[7] Use of this data requires the acknowledgement that it was collected at University of Wisconsin Hospitals, Madison by Dr. William H. Wolberg

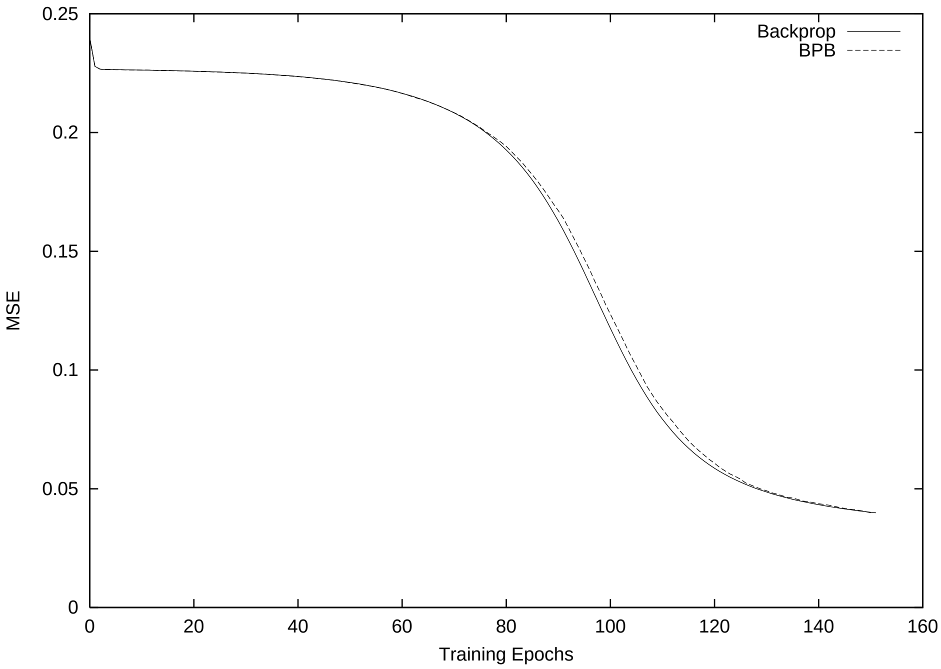

Figure 3: Mean Squared Error (MSE) of output on training set for backpropagation and BPB

In this case $\tau = 4$, so the total number of feed-forward passes required for BPB is about 5 times that of backpropagation. Performance for different values of τ is a central issue in BPB, and is returned to in sections 5.2 and 8. We first present the mathematical ideas behind BPB.

4 Mathematical Basis of BPB

4.1 Derivation of the Update Rule

For simplicity we deal with a single output unit, whose output value is Y. Backpropagation is essentially a method for calculating the gradient of Y with respect to ϕ_i (the weighted sum of activity into the i^{th} neuron). Call this (true) gradient

$$\alpha_i = \frac{\partial Y}{\partial \phi_i}$$

We approximate this gradient by adding noise n_i to the input ϕ_i, and observing the change in the output ΔY. Suppose we estimate α_i by

$$\alpha_i \approx \frac{\Delta Y}{n_i}$$

This becomes exact in the limit of $n \to 0$ since

$$\frac{\Delta Y}{n} = \frac{f(\phi + n) - f(\phi)}{n}$$

and using our local linearity equation 1 above, this is

$$\frac{f(\phi) + \alpha n - f(\phi)}{n} = \frac{\alpha n}{n} = \alpha$$

and we can rewrite it as

$$\alpha_i \;=\; c\, n_i \, \Delta Y \tag{2}$$

where $c = \frac{1}{n_i^2}$. This is an interesting form because it is just like a Hebbian update of a weight between the output unit Y and unit i, with c playing the role of a learning rate.

One could take each neuron in turn, adding noise to the units one by one (allowing each unit to estimate its influence exactly), but this sacrifices the locality that we are trying to achieve. If all hidden units add noise all the time, the effects of other units must be averaged out to get the effect of i alone.

If we average equation 2 over τ forward passes, each with random noise, we have an estimate for α_i of

$$B_i \;=\; c \sum_{s=1}^{\tau} n_i^s \, \Delta^s Y \tag{3}$$

where $\Delta^s Y$ is the change in the output Y caused by noise from every node in the network, not just i. This estimate is labeled B_i, as it is the weight on the **b**ackward link from the output Y to the unit i. Note that s is an index counting the samples, rather than an exponent. The correct value for c turns out to be

$$\frac{1}{var(n)\ \tau}$$

as shown in section 4.2.

4.2 Expected value of B

Here we confirm that the expected value of B_i is the true gradient α.
As before, by assuming the input-output mapping is locally linear we can write

$$\Delta^s Y = \sum_j \alpha_j n_j^s \tag{4}$$

That is, for any sample s, the change in the output of the network is equal to the noise added multiplied by some constant, and that constant is the gradient. Substituting this into equation 3 and rearranging gives

$$B_i = c \ \sum_j \alpha_j \sum_{s=1}^{\tau} n_i^s n_j^s$$

The expected value of this is

$$E[B_i] = c \ \sum_j \alpha_j \sum_{s=1}^{\tau} E[n_i^s n_j^s] \tag{5}$$

The expected value of B_i depends on the noise at unit i times the noise at unit j, at each sample s. The noise produced by i is uncorrelated with the noise produced by j, so

$$E[n_i^s n_j^s] = 0, \forall i \neq j$$

The only case remaining is $i = j$ (when i and j refer to the same unit), where the expected value is just the variance of the noise, $var(n)$. Equation 5 becomes

$$E[B_i] = c \ \ \alpha_i \sum_{s=1}^{\tau} \sigma^2 \ \ = c \ \ \alpha_i \ \tau \ var(n)$$

and if

$$c = \frac{1}{var(n) \ \ \tau}$$

we get α_i. So the expected value of the back-weight is indeed the true gradient.

4.3 Variance of B

The form that the variance of this estimate takes is important, as it indicates how well the algorithm scales for different parameters and architectures. The variance of B_i is

$$\text{Var}(B_i) = E[B_i^2] - (E[B_i])^2 \tag{6}$$

The second term is α_i^2 as above, making the equation

$$\text{Var}(B_i) = E[B_i^2] - \alpha_i^2$$

The first term is

$$B_i^2 = \left(\frac{1}{var(n) \ \tau} \sum_{s=1}^{\tau} n_i^s \ \Delta^s y \right)^2$$

which is

$$B_i^2 = \frac{1}{var(n)^2\ \tau^2} \sum_s \sum_{s'} n_i^s n_i^{s'} \Delta Y^s \Delta Y^{s'}$$

Substituting in equation 4 (using $\Delta^s Y = \sum_j \alpha_j n_j^s$ and $\Delta^{s'} Y = \sum_k \alpha_j n_k^{s'}$) gives

$$B_i^2 = \frac{1}{var(n)^2\ \tau^2} \sum_s \sum_{s'} n_i^s n_i^{s'} \sum_j \alpha_j n_j^s \sum_k \alpha_k n_k^{s'}$$

After rearranging the sums we get

$$B_i^2 = \frac{1}{var(n)^2\ \tau^2} \sum_{j,k} \alpha_j \alpha_k \sum_{s,s'} n_i^s n_j^s n_i^{s'} n_k^{s'}$$

The expectation over all those noise terms is

$$E[B_i^2] = \frac{1}{var(n)^2 \tau^2} \sum_{j,k} \alpha_j \alpha_k \sum_{s,s'} E\left[n_i^s n_j^s n_i^{s'} n_j^{s'}\right]$$

Again, there are two cases here, namely those where $s \neq s'$ and those where $s = s'$

- When $s \neq s'$ (ie, two different samples are being compared), the expectation is zero if *any* of the four terms remains uncorrelated, which is the case unless $i = j = k$, leaving

  $$E[(n_i^s)^2 (n_i^{s'})^2]\ \ \delta_{i=j=k}$$

  where $\delta_{i=j=k}$ is the Kronecker delta function (1 if $i = j = k$ and 0 otherwise). The expectations are done at different samples s, so we get $var(n)^2$, summed up $\tau(1-\tau)$ times[8]. The sum over j, k is just α_i^2 due to the Kronecker delta, so this comes to

  $$\left(\frac{\tau - 1}{\tau}\right)\ \alpha_i^2$$

- When $s = s'$, the expectation is zero unless $j = k$, but we no longer require that $i = j$. The inner expectation is now $var(n)^2 \delta_{j=k}$. Since $s = s'$ this only gets summed up τ times, and the j, k sum becomes a sum over α_j^2 because $j = k$, leaving

  $$\frac{1}{\tau} \sum_j \alpha_j^2$$

Equation 6 now reads as follows:

$$\mathrm{Var}(B_i) \ = \ \left(\frac{\tau - 1}{\tau}\right) \alpha_i^2 \ + \ \frac{1}{\tau} \sum_j \alpha_j^2 \ - \ \alpha_i^2$$

Further, if the sum is restricted to $j \neq i$, all i terms are accounted for. The very simple result is that

$$\mathrm{Var}(B_i) = \frac{1}{\tau} \sum_{j \neq i} \alpha_j^2 \tag{7}$$

The variance from the true gradient is $\frac{1}{\tau}$ multiplied by the sum of the squares of the influences at each hidden node. Properties of the algorithm arising from this equation are discussed in sections 5.2 and 7.

[8] ie. all samples s, multiplied once by all other samples.

4.4 Using the back weights to update the feed-forward weights

The important result shown above is that an estimate of the gradient α_i at some unit i can be arrived at through the following update equation for the back-weights

$$B_i \ += \ c \ n_i^s \ \Delta^s Y \tag{8}$$

where the learning rate for this Hebbian weight change is

$$c = \frac{1}{var(n)} \ \tau$$

That is, the noise n_i^s added by *this* unit i, multiplied by the change in the corresponding output $\Delta^s Y$ for this sample s. We divide by τ to take the average, and by the variance of the noise distribution to eliminate the proportionality to the gradient. An estimate of the total error from multiple outputs is then arrived at by

$$\epsilon_i \approx \sum_1^k (T_k - Y_k) \times B_{ik}$$

Which is the sum over all output units k of the error at k multiplied by the estimate of unit i's influence on unit k. The weights of the connections leading into unit i can be updated using the standard update rule

$$w_{ij} = \eta \ \ \epsilon_i \ \ y_j$$

where y_j is the output value of unit j.

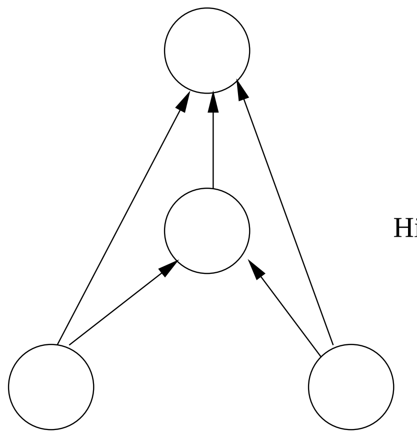

Figure 4: The XOR network used in these experiments (bias units and their connections not shown).

5 Basic Properties of BPB

5.1 Small β is better than large β

The one major assumption made under BPB (and under conventional backpropagation) is that the activation function of each unit is locally linear. This means that the non-linear effects of this function can be disregarded when trying to discern the effect of a particular unit upon the output. If too much noise is added the activation function is less and less linear over the noise distance ($f(i+n)$ gets further and further away from $f(i) + \alpha n$), and the gradient estimates get worse and worse.

Figure 4 shows the simplest network that can solve the XOR problem. It has one hidden unit, so under BPB there is only one unit to perturb the output, and as there will be no interference between hidden nodes, the single hidden unit will be able to exactly estimate its influence on the output of the network.

Figures 5 through 8 show the gradient calculated by BPB at the hidden unit in the network described, as training proceeds, for different values of β.

The variability of the estimate can be clearly seen, and becomes larger as we add more noise to the system. Also note that though systems with large β still learn, it takes longer for these systems to train to the same accuracy as systems with smaller values of β.

So when training with BPB, one should use noise that is as small as possible, in order to achieve good gradient estimates.

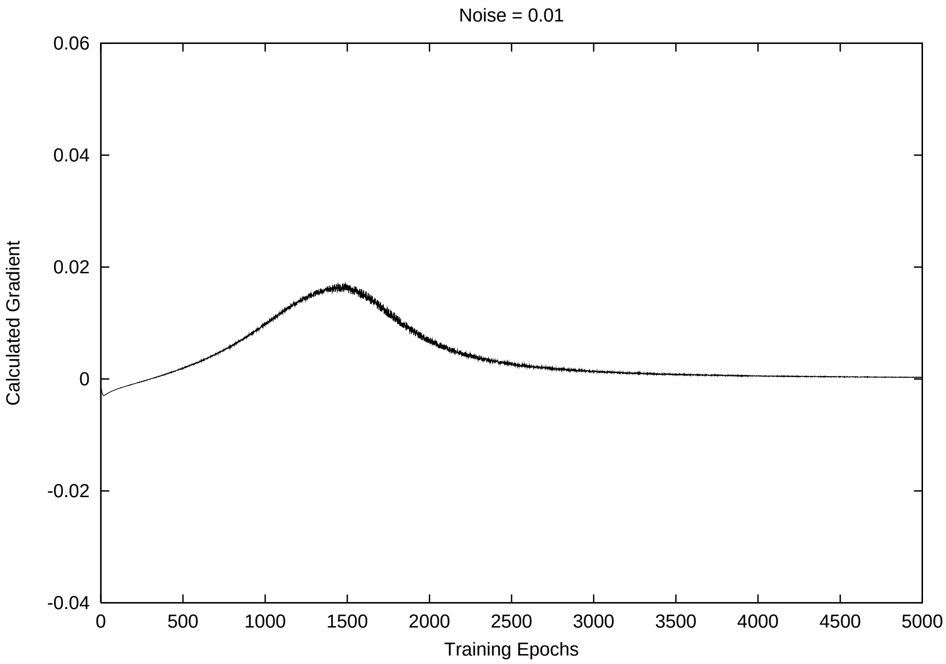

Figure 5: The gradient estimated at the hidden node for each training epoch with β= 0.01

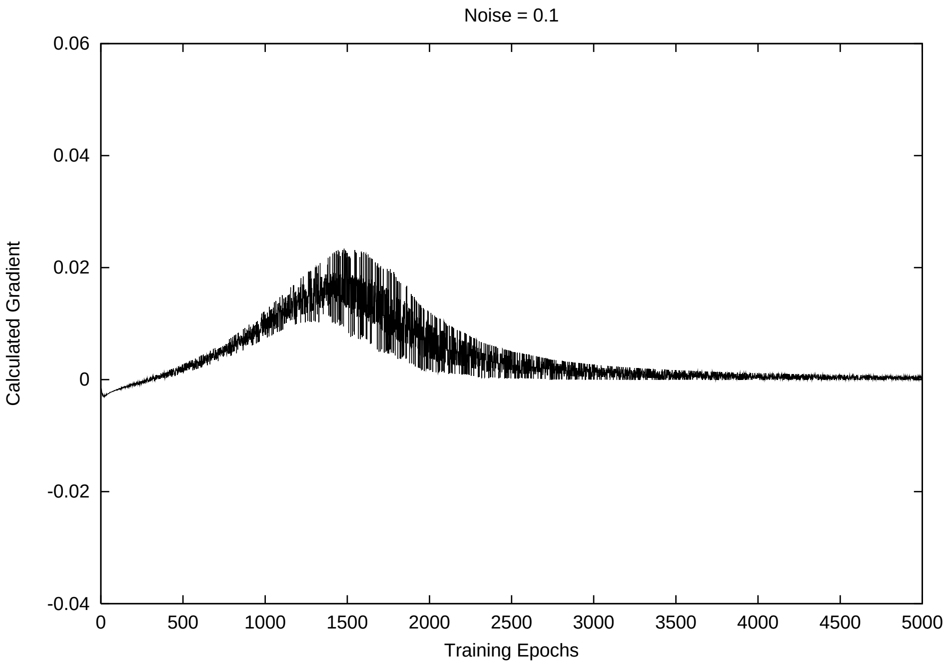

Figure 6: The gradient estimated at the hidden node for each training epoch with β= 0.1

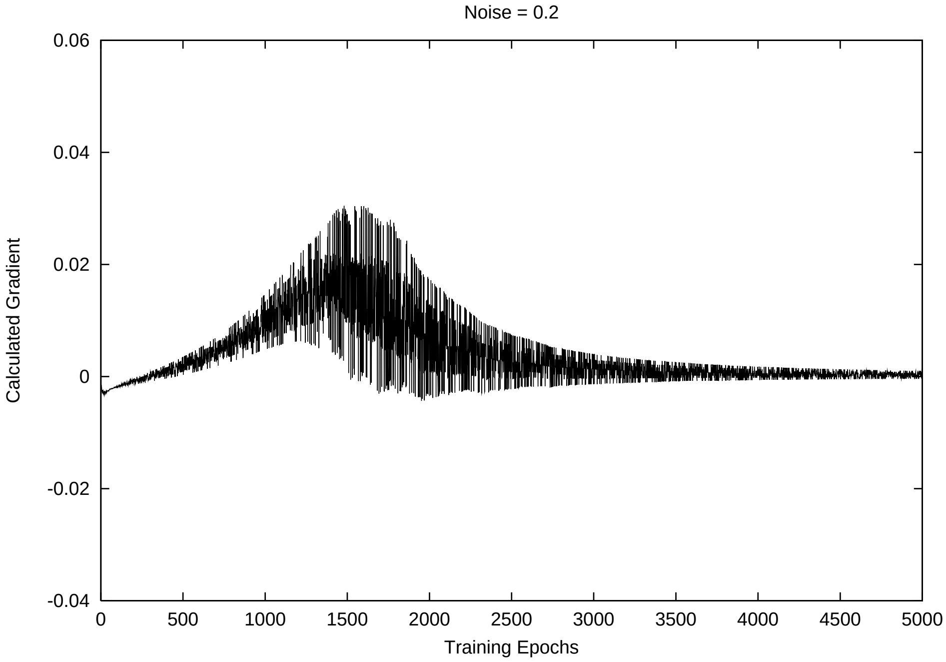

Figure 7: The gradient estimated at the hidden node for each training epoch with β= 0.2

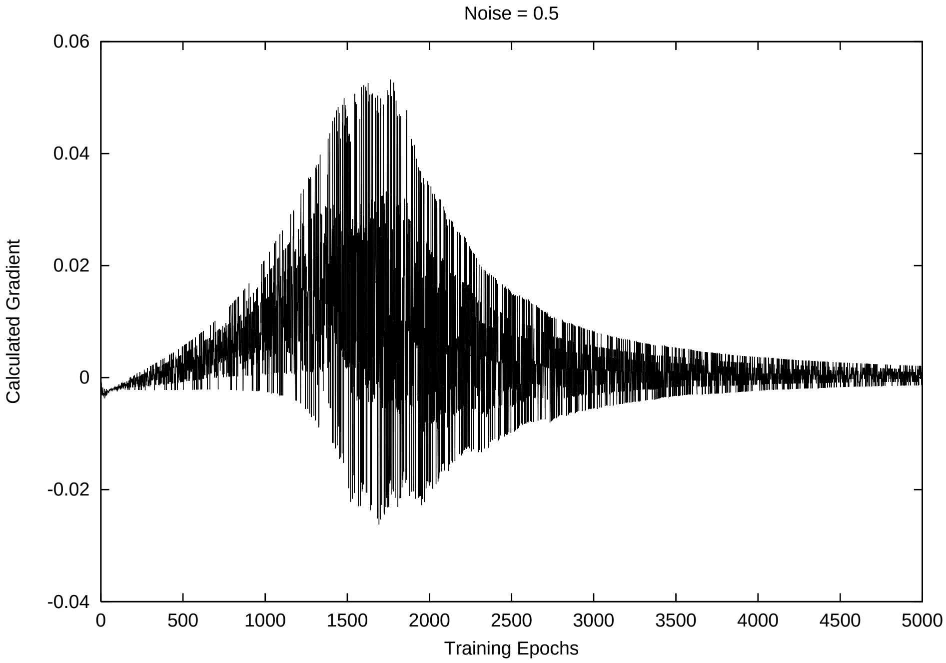

Figure 8: The gradient estimated at the hidden node for each training epoch with β= 0.5

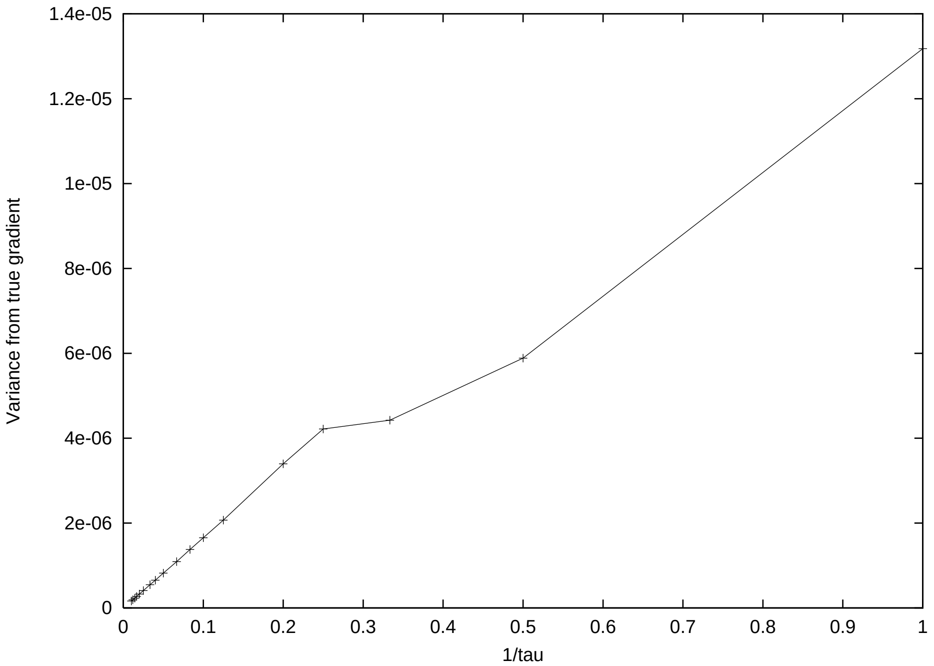

Figure 9: Error vs $\frac{1}{\tau}$ for 18 values of τbetween 1 and 100

5.2 Large τ is better than small τ

The estimate of the gradient produced will tend to the actual gradient as $\tau \rightarrow \infty$. When τ is low, we cannot get a very good estimate of the gradient, but as the number of samples gets larger and larger, the estimate gets better and better. In fact, the variance of the estimate is proportional to $\frac{1}{\tau}$ as in equation 7.

Figure 9 shows the variance from the true gradient against $\frac{1}{\tau}$. This data comes from 50 training runs on the XOR problem for each of 18 values of τ. The network used had one hidden layer, containing eight hidden nodes, in order to give a reasonable amount of interference between noise generated at each hidden unit. The network was fully connected between adjacent layers, with no shortcut connections. Weights were randomly initialised to a value between 1 and -1. The learning rate was 0.01, and noise was ± 0.001. Each run was smoothed, and the highest value of the variance over the whole training period was recorded. The 50 variances produced were then averaged to generate this plot.

In order to get the variance from the true gradient, we calculate the true gradient at each node and find the squared difference of this and the estimated gradient, so what is plotted is the total variance of the entire network.

Figure 9 is a straight line for all but the smallest values of τ. This confirms in practice the theory that the variance of the gradient estimates is proportional to $\frac{1}{\tau}$.

6 Optimisations for BPB

6.1 What to do when the input pattern changes

There are several things that can be done to the weights on the backward links when the input pattern changes. The simplest thing to do is reset them to zero. This means that B_i must be estimated from scratch for each different training pattern. It throws away everything learnt about the influence of the node at the previous pattern. If we modify equation 8, we can include some information past values of B_i. Whereas previously all samples were treated as equal, and we simply averaged over the number of samples, we now treat more recent samples as more valuable, incorporating the new sample in with the old ones as we go. This is a 'running average' rather than an actual average. Call this approach BPB_γ. We use the modified update equation

$$B_i(t+1) = (1-\gamma)B_i(t) + \gamma \, \frac{n_i(t)\;\Delta Y}{var(n)}$$

for appropriately small γ. The first term is simply a proportion of the old back-weight, and the second term is the change calculated in the update equation 8, with $c = \frac{1}{var(n)}$. Dividing by τ is no longer needed, as multiplying by γ properly proportions this sample with respect to the others.

If $\frac{1}{\tau}$ is used for γ, the back-weight will fade to the actual mean, as each sample counts for the same as it did under the old system, though its effect hangs around for longer.

Under this method, different patterns could have differing values of τ. One can imagine a situation in which some patterns were presented for longer than others. Under the resetting case, the network would have a bad estimate of the gradient to be descended for those patterns which were only presented for a small number of time-steps (τ), and a much better estimate for those patterns that were presented for longer. Under this method, previous gradient information can help better estimate the effect of every pattern.

Figure 10 shows a training run for BPB_γ, as well as for regular BPB. There is little difference in the number of epochs taken, but the BPB_γ estimate of the gradient is much better, as figure 11 shows. Note that when training is nearly complete, the variance of the gradient estimate fluctuates around some constant as the gradient approaches zero. This is because when the gradient is zero, BPB reverts to a random walk, and can never find the exact bottom of the minimum.

Dataset cancer1

α 0.25

The network used had two hidden layers, containing four and two units respectively, with all feed-forward connections between adjacent layers. Weights were randomly initialised to a value between -1 and 1. The learning rate was 0.01, τ 4, and noise was ± 0.001. The network was trained on the cancer data until the MSE on the training data was less than 0.04. For BPB_γ, α is 0.25, $\frac{1}{\tau}$.

6.2 What to do with the perturbed output value

Previously, changes to the back-weights were discussed in terms of the change in the output. This would suggest that we take the difference between the clean output and the noisy output,

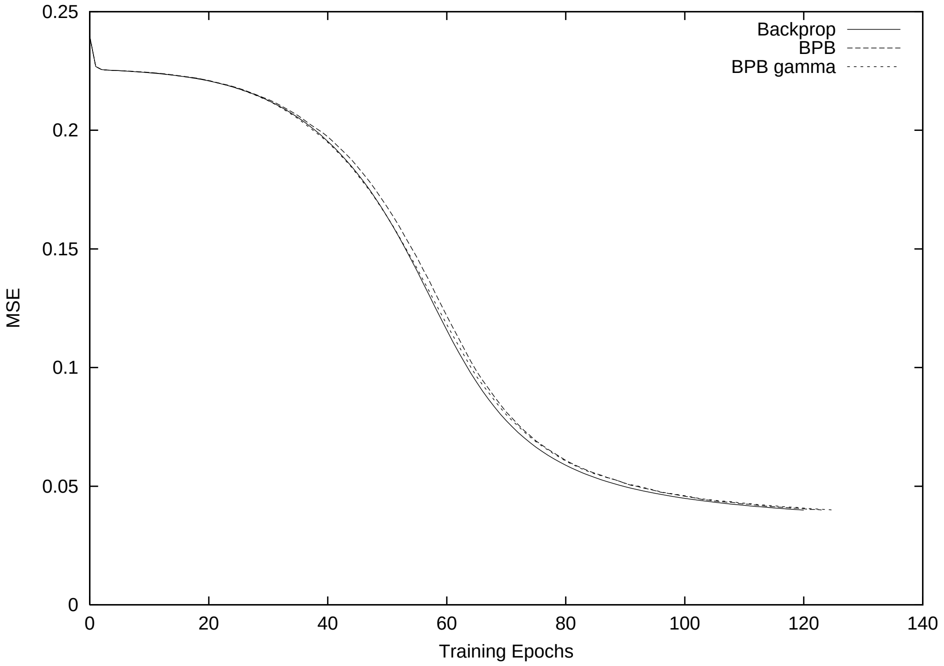

Figure 10: Training MSE vs Epochs

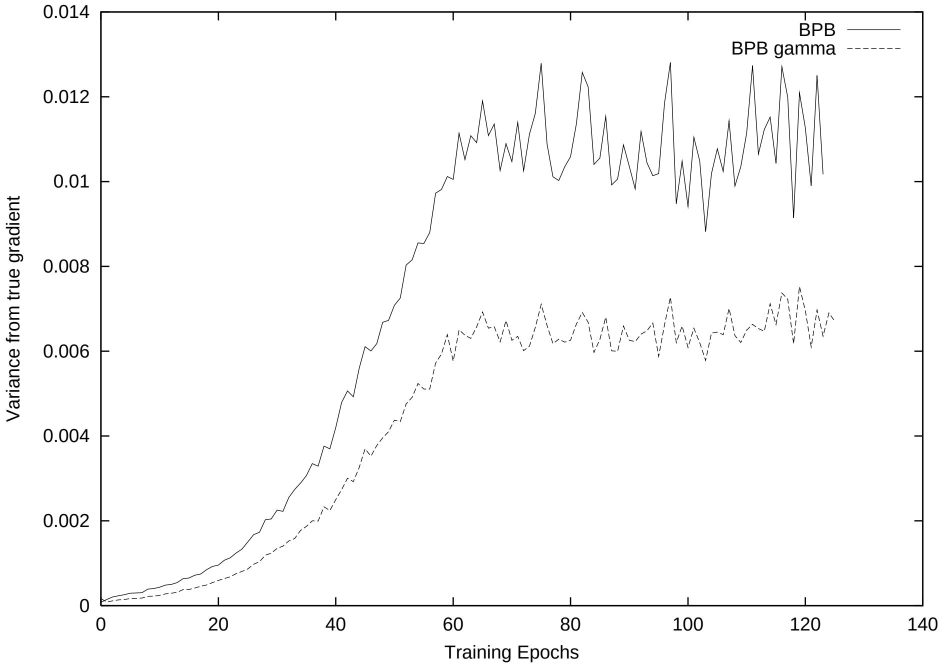

Figure 11: Variance from backpropagation vs Epochs

and use this value to update the back-weights. A more brain-like approach would be to just use the raw noisy output of the network to update the back-weights, as over multiple τ the variance in the output will enable the back-weights to be updated in using exactly the same rules. Call this approach BPB^*.

If we substitute $\Delta Y + Y$ for ΔY in equation 3 above, we get the following:

$$\begin{aligned} B_i &= c \sum_{s=1}^{\tau} n_i^s \ (\Delta Y + Y) \\ &= c \sum_{s=1}^{\tau} (n_i^s \ \Delta Y + \ n_i^s \ Y) \\ &= c \, (\sum_{s=1}^{\tau} n_i^s \ \Delta Y \ \ + \ \ Y \sum_{s=1}^{\tau} n_i^s) \end{aligned}$$

Since the expected value of the noise — $E[n]$ — is zero, the second term disappears, and we are back to equation 3.

In practice, to produce reasonable estimates of the gradient using this method, we need either large τ or large β. If τ is low, then not enough samples are taken to remove the effect of the second term. If β is not large enough, then the added noise gets swamped by the second term. Unless a reasonable value is chosen for β, variance can be orders of magnitude worse than for regular BPB.

A bad estimate of the gradient does not necessarily preclude a good training run. Typically, both of these methods can learn a given task to a specified error in around the same number of epochs. BPB^* suffers more from bad estimates of the gradient than any other variation of BPB, in that if β is too low, training can actually fail quite spectacularly, with no learning taking place at all.

Figure 12 shows an *atypical* training run. In this case, when training begins, the network is at a *very* flat point on the error surface. BPB^* produces a particularly bad estimate of the gradient, and it takes longer for the network to find its way off the plain. Figure 13 shows that the variance of the estimate of the BPB^* network *throughout* the training run is comparable to the variance of the regular BPB network at the end of training.

The network used had two hidden layers, containing four and two units respectively, with all feed-forward connections between adjacent layers. Weights were randomly initialised to a value between -1 and 1. The learning rate was 0.01, τ 4, and noise was ± 0.8. The network was trained on the cancer data until the MSE on the training data was less than 0.035.

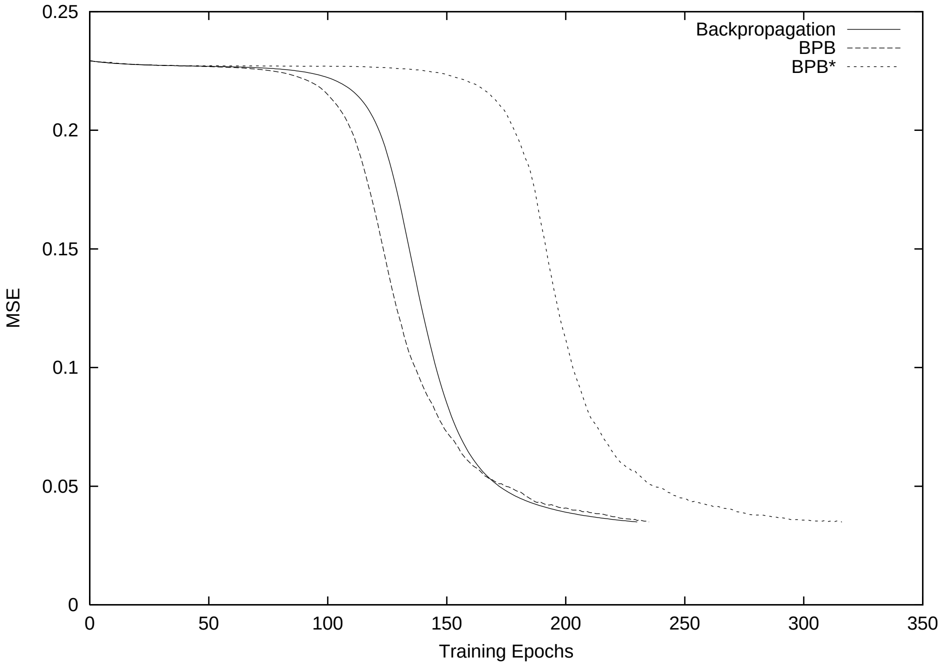

Figure 12: Training MSE vs Epochs, BPB^* and regular BPB

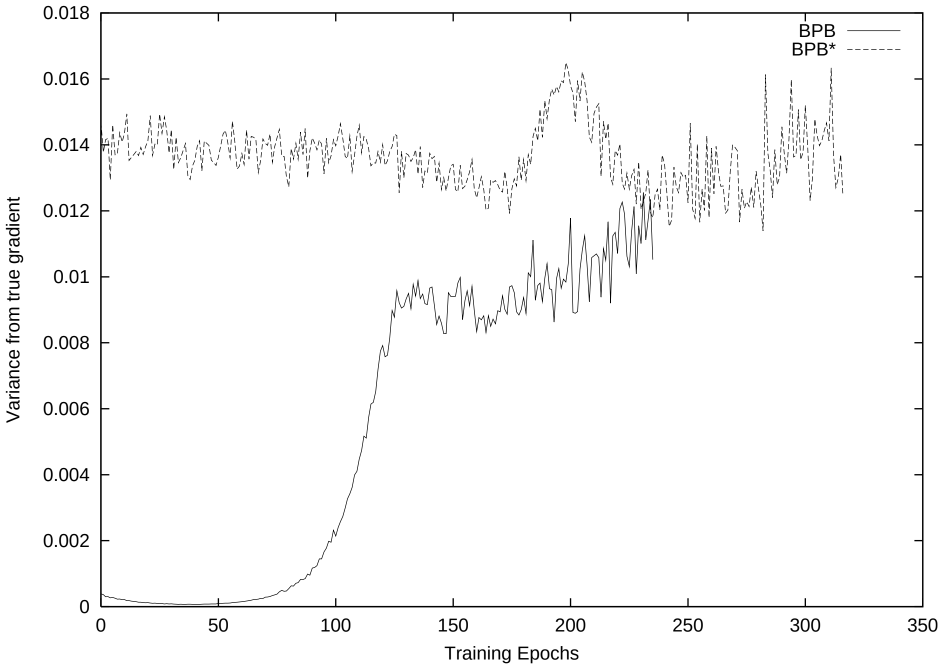

Figure 13: Variance from the true gradient vs Epochs, BPB^* and regular BPB

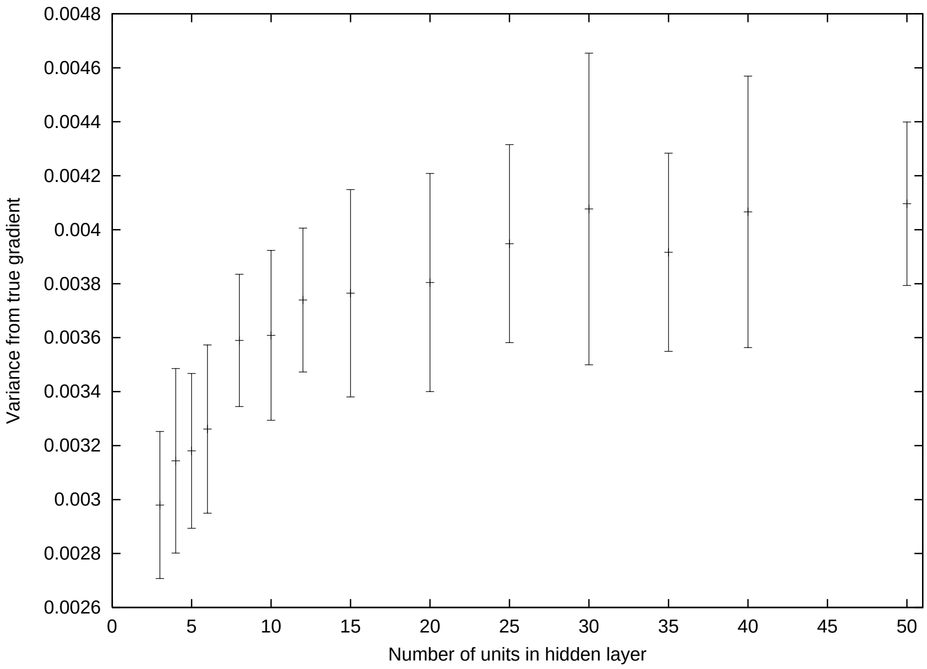

Figure 14: Variance for networks with different numbers of hidden units

7 Variance and the size of a network

Equation 7 states that the variance of an estimate is the sum of the squared gradients at all the hidden units in the network. In the worst case, the size of the variance should increase linearly as the number of hidden units goes up. This depends on the gradients staying the same. In practice, when training a large network to do the same job as a smaller one, the gradients at each node in the network will be proportionally smaller, as each unit has less of an influence on the ultimate output of the network. Thus, the total variance in the network does not rise linearly, but instead levels off as more hidden units are added.

Figure 14 shows the variance from the true gradient for different numbers of hidden units close to the end of training. The error bars are the standard deviation over 20 samples.

One might have expected that BPB would not scale well to use in large networks, because of the interference between more and more noise sources. However because the variance does not rise linearly (here it looks to have plateaued at around 25 hidden units) the gradient estimates in relatively large networks are not significantly worse than those in smaller ones.

The network used had one hidden layer, containing a varying number of units, with all feed-forward connections between adjacent layers. Weights were randomly initialised to a value between -1 and 1. The learning rate was 0.01, τ 4, and noise was ± 0.001. The network was trained on the cancer data until the MSE on the training data was less than 0.05.

8 Accuracy vs Speed

It seems natural that a better gradient estimate means that it takes relatively fewer epochs to train to a particular error level. This is not necessarily the case. In section 5.2 it was shown that the variance of the gradient estimate produced by BPB is proportional to $\frac{1}{\tau}$. Figure 15 shows the number of epochs needed to train networks of varying τ to a particular error value on the cancer data from the PROBEN1 collection.

One of the most interesting things encountered during the investigation of the BPB algorithm is that good results can be obtained even with extremely low τ. In fact, there is little difference in the speed of learning of different τ, even the variance of the training time is the same. Although the gradient estimate produced by a network with small τ is much worse than for a network which takes many more samples, they will both arrive at the specified error in the same time.

It seems that a noisy estimate of the gradient is as good as a non-noisy value. It doesn't really matter that we step directly down the gradient, as long as we are going in that general direction. This makes the extra feed-forward passes spent sampling to get better estimates of the gradient unnecessary.

There were two approaches to testing this postulate:

1. Use the directly calculated backpropagation gradients with some random noise added, to see what effect this has on learning.

2. Find a harder problem to test the algorithm on, where getting a good estimate of the gradient *does* matter.

Initially, the first approach sounds simple. The critical question is how much noise to add. Though we can easily add noise proportional to the value of τ that we are simulating, the variance in the estimate of the gradient changes as training progresses. To add the correct amount of noise to the calculated gradient, we would need to record the variance of the estimated gradient, and use this changing variance to add noise to the calculated one. Finding the correct variance to add is more difficult than was initially thought, so the other approach was used.

Additional testing was performed on the heart[9], horse, card and soybean data in the PROBEN1 benchmark set. These problems produced data similar to that shown in figure 15.

It would seem that for these problem sets, a noisy estimate of the gradient is just as able to learn a classification as an exact calculation. This would tend to suggest that these problems are too easy to show the kind of behaviour that we might expect. However, these datasets have been compiled from real problems, the sort which are encountered in the real world. If a noisy gradient estimate can be used to correctly classify these datasets, then a noisy gradient estimate can be used to classify other real datasets accurately. Though it seems likely that there is some problem for which the gradient is so important that we need to use large τ to get decent training performance, the difference between the accuracy of the gradient estimates produced by low τ and high τ becomes a theoretical, rather than a practical problem, and we can use BPB with low τ to accurately learn real world problems.

It was initially thought that there would be some optimal value of τ which produced the best results in the smallest amount of time. Thinking about this for a moment, we can see that

[9] Use of this data requires the acknowledgement that it was originally collected at the Hungarian Institute of Cardiology, Budapest by Andras Janosi, M.D., University Hospital, Zurich, Switzerland by William Steinbrunn, M.D., University Hospital, Basel, Switzerland by Matthias Pfisterer, M.D., V.A. Medical Center, Long Beach and Cleveland Clinic Foundation by Robert Detrano, M.D., Ph.D.

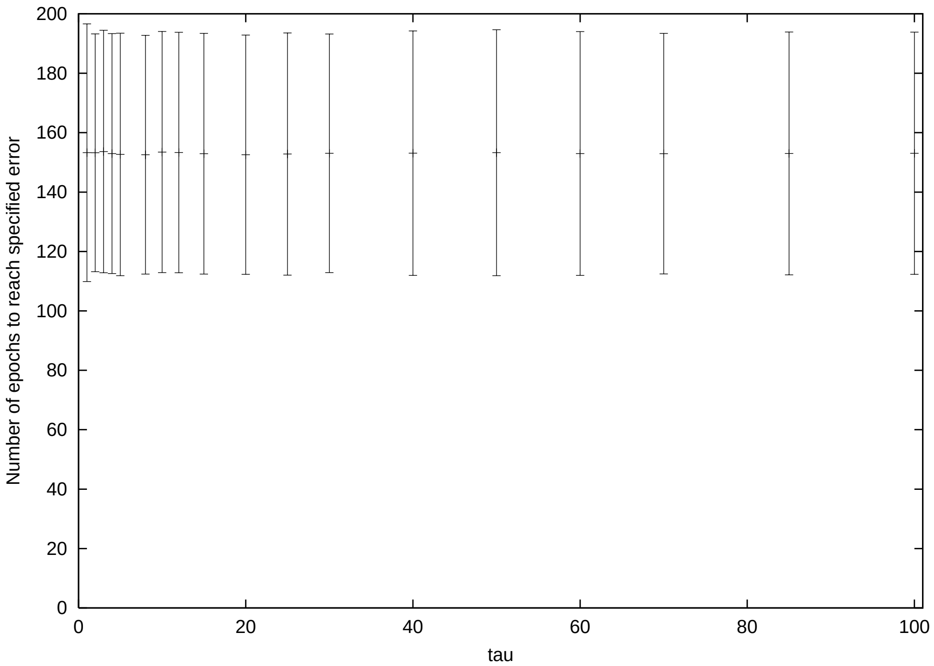

Figure 15: Graph of Number of training epochs needed to reach specified error vs. tau

the problem will have to be very difficult in order for this optimal τ to be a large one. For a larger τ to be better, it must train considerably faster. For $\tau = 2$ to have fewer feed-forward passes than $\tau = 1$, it must train in half as many epochs. For $\tau = 3$ to have fewer feed-forward passes than $\tau = 2$, it must cut training time by more than a third.

The network used had two hidden layers, containing four and two units respectively, with all feed-forward connections between adjacent layers. Weights were randomly initialised to a value between -1 and 1. The learning rate was 0.01 and noise was ± 0.001. The network was trained on the cancer data until the MSE on the training data was less than 0.035.

9 Conclusion

This project has investigated BPB, an algorithm for learning in neural networks which models backpropagation in a more biologically plausible manner. This algorithm is robust, mathematically correct, and has some additional advantages over conventional backpropagation. Two particularly encouraging results are that τ does not have to be large to get a good estimate, thus the algorithm does not need to do too many more feed-forward passes than backpropagation, and that the algorithm scales up to large networks without a major loss of performance. Neither of these results were anticipated at the outset of the project.

9.1 Reflections on the biological plausibility of BPB

Many of the biologically implausible aspects of backpropagation have been addressed and solved. In the spirit of the original conception of BPB, we can propose a *most* biologically plausible variation of BPB.

We use both BPB_γ and BPB^*, creating another variant, BPB^*_γ. We use τ of 1, and thus don't need to keep track of the input patterns, simply updating all the weights once each time-step. If the pattern stays around for more than one time-step, the running average of the back-weights means that the influence does not have to be estimated anew. No clean feed-forward passes are needed, meaning that the number of feed-forward passes performs is simply one per input pattern. Training is not only accomplished with a similar number of epochs to backpropagation, but also with a similar number of feed-forward passes.

We have an algorithm that trains with similar efficiency to backpropagation, and is more biologically plausible.

9.2 Directions of future research

In [11], Bartlett and Baxter introduce a reinforcement learning algorithm they call Direct Reinforcement Learning (DRL). Part of the algorithm needs an element of randomness to work. It is possible that the randomness inherent in BPB could be put to dual use under DRL, and a hybrid algorithm produced.

In section 3.2, the example was made of a neural network in which genetic algorithms are used to decide the output of each node. It would be interesting to do some trials using BPB to determine the fitness of the algorithm at each node.

In section 3.2, it was also conjectured that the BPB algorithm could be hardwired on a silicon chip. This possibility seems particularly promising, and investigation would be well worthwhile.

9.3 Acknowledgments

I would like to thank my supervisor Dr. Marcus Frean for the bulk of section 4, and for spending many hours leading me by the hand through forests of sums and derivatives. It all looks so much simpler from this side ...

References

[1] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, Massachusetts, 1969.

[2] F. Rosenblatt. The perceptron: a probablistic model for information storage and organization in the brain. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing : Foundations of Research*, pages 89–114. MIT Press, Cambridge, Massachusetts, 1988.

[3] B. Widrow and M. E. Hoff. Adaptive switching circuits. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing : Foundations of Research*, pages 123–137. MIT Press, Cambridge, Massachusetts, 1988.

[4] Lutz Prechelt. PROBEN1 - A Set of Neural Network Benchmarking Problems and Benchmarking Rules. Technical Report 21/4, Universität Karlsruhe, 7128 Karlsruhe, Germany, September 1994.

[5] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Internal Representations by Error Propagation. In David E. Rumelhart and James L. McLelland, editors, *Parallel Distributed Processing : Explorations in the Microstrucure of Cognition Volume 1 : Foundations.* MIT Press, Cambridge, Massachusetts, 1986.

[6] Marcus R. Frean. Local Learning of Error Gradients. Department of Computer Science and Electrical Engineering, University of Queensland, 1998.

[7] Georg Thimm, Perry Moerland, and Emile Fiesler. The Interchangeability of Learning Rate and Gain in Backpropagation Neural Networks. *Neural Computation*, 8(2), February 1996.

[8] Scott E. Fahlman. An empirical study of learning speed. Technical Report CMU-CS-88-162, Carnegie Mellon University, 1988.

[9] Donald O. Hebb. The Organisation of Behaviour. In James A. Anderson and Edward Rosenfeld, editors, *Neurocomputing : Foundations of Research*, chapter Introduction and Chapter Four, pages 45–56. MIT Press, Cambridge, Massachusetts, 1988.

[10] X. Yao. Evolutionary artificial neural networks. *International Journal of Neural Systems*, 4(3):203–222, September 1993.

[11] Peter L. Bartlett and Jonathan Baxter. Hebbian Synaptic Modifications in Spiking Neurons that Learn. Technical report, Australian National University, Canberra ACT 0200, Australia, November 1999.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Prentice-Hall, Upper Saddle River, New Jersey 07458, 1st edition, 1995.

[13] Warren S. Sarle. comp.ai.neural-nets FAQ. ftp://ftp.sas.com/pub/neural/FAQ.html, 2000.

[14] Donald R. Tveter. Backpropagator's Review. http://www.dontveter.com/bpr/bpr.html, 1998.

[15] Silicon Recognition Inc. ZISC Zero Instruction Set Computer Technology Reference Guide. http://www.silirec.com/download/zisc_refguide.pdf, April 2000.

[16] O. L. Mangasarian and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News*, 23(5):1 and 18, September 1990.

[17] William H. Wolberg and O.L. Mangasarian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the National Academy of Sciences, U.S.A.*, 87:9193–9196, December 1990.

[18] O. L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis. In Thomas F. Coleman and Yuying Li, editors, *Large-scale numerical optimization*, pages 22–30. SIAM Publications, 1990.