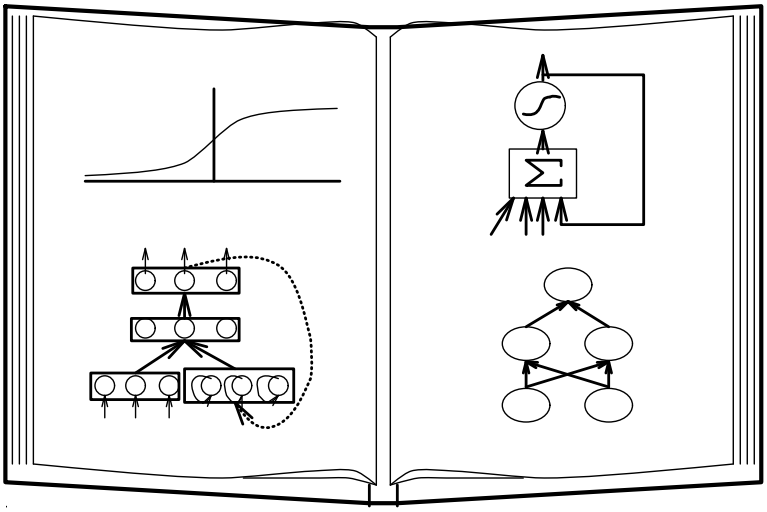

Back Propagation Family Album

jondarr c g h 2 ♡ gibb
Department of Computing
Macquarie University
NSW 2109

Technical Report C/TR96-05

August, 1996
Version 1.0

Abstract

Training of multi-layer feed-forward artificial neural networks via the backpropagation algorithm has a popularity and general appeal exceeded only by the simplicity of its implementation and the way in which the algorithm may be improved upon or specialised for any given problem. These factors have given rise to the current situation where there are so many algorithms which trace their heritage to backpropagation that it is almost impossible for any one person to be aware of all of the developments, and it is even more difficult for someone new to the area of feed-forward networks to choose an appropriate member of the backpropagation family for the task envisaged.

This is a survey of backpropagation learning algorithms and techniques. It brings together members of the backpropagation family which have been developed away from their siblings over the years, from its conception to modern modifications to the algorithm.

In this taxonomy, the family members are broken into five major branches of the tree : *Algorithms*, *Heuristics*, *Regularisers*, *Network Techniques* and *Generative Networks*. Each branch has grown away from the trunk in a different direction, and only rarely have the limbs touched, generally only where two minor branches have come together. This is a place where many branches meet.

It is my belief that in bringing together some of these distant relations, new combinations of techniques can be investigated which marry the particular advantages of disparate branches in the family and breed new methods carrying the best attributes of their predecessors.

There is no intention for this document to investigate issues of speed of training, or any performance measures, on the algorithms presented, but merely to summarise the details of the variations on backpropagation that have been developed.

Contents

List of Figures

Chapter 1

Error Backpropagation in Time

Backpropagation, as a development of the *LMS algorithm* [100] [74] applied to feed-forward multi-layer perceptron networks, was proposed by Werbos [99] and Parker [69] independently. The former suffered from the after-effects of Minsky's attacks on the validity of the Perceptron [61] and the latter gained little publicity compared to the classic book of the PDP Research Group [77], and particularly the formalisation of the delta-rule by Rumelhart, Hinton and Williams [76] therein. It is this last document which is considered to show the full potential of the learning algorithm. The algorithm itself is discussed fully in Section 2.1 and derived in Appendix A.

This report is a survey of modifications to that original, rather simple, algorithm proposed by Rumelhart, et al. Since 1986, there has been a flood of modifications, improvements, heuristics and black magic attached to backpropagation, and much has been written to show how each successive progeny is 'better' than its siblings, or at least an improvement on the parent algorithm.

In this document I will try to bring together as many different relations as I can for a major family reunion and introduce many backpropagation techniques to each other that have not been in the same document before.

1.1 Groups within the Family

The family of backpropagation algorithms is far from coherent. Not all of the techniques applied to Rumelhart's delta-rule can be considered as strictly changes to the algorithm, but certainly do improve some aspect of the learning, such as speed of training, the ability to learn a solution to a particular problem and learn with consistency, or generalisation. These are issues that all neural network family members are intended to improve – through evolution (LeMarckian and Darwinian) we improve the feasability of the species.

I have divided the family into five major branches to show different directions of change that have occurred.

Algorithmic Changes: by this, I mean a change to the original algorithm or a definition of what is being optimised by the network (normally error with respect to weights), or the manner in which the error is minimised. This includes additions (or deletions) of terms which change some aspect of the learning (such as a momentum term), but do not

change the nature of the network itself, merely the operations within it.

Heuristics applied to Algorithms: these techniques and heuristics change the manner in which (almost) any algorithm may be applied. They are generally independent of the optimisation being performed, although some tend to be algorithmic in nature (such as *Descending Epsilon*). Other heuristics are changes which apply to the network based more on experimental results rather than any theoretical basis.

Regularisers: these changes affect the error function by adding some cost term on the weights, trying to minimise the network architecture while solving the problem. The earliest example of this is *Weight Decay*.

Techniques applied to Networks: anything which changes the network operation without changing the problem definition is a technique applied to a network – in general, these can be applied to any problem to produce similar effects. They tend to change the shape of the error surface being optimised (such as different activation functions).

Generative Techniques: networks which have the ability to grow or diminish are separate from Network Techniques because they often involve an Heuristic or Algorithm to change the size of the network, such as *Cascade Correlation*.

The following lists the members of each branch of the backpropagation family with the primary investigators / discoverers of that offspring.

Algorithmic Methods

Delta Rule (Rumelhart, et al [76]), (Parker [69])
Momentum (Rumelhart, et al [76])
Second Order Methods
– Second Order Momentum (Pearlmutter [70])
– Quickprop (Fahlman [24])
– Newton, Quasi- and Pseudo-Newton Methods (Watrous and Shastri [97])
– Le Cun's Technique [55]
Gain Variation (Plaut, et al [71])
Conjugate Gradient Techniques (Johansson [46])
– Scaled Conjugate Gradient Method (Møller [62])
Backpropagating Desired States (Plaut, et al [71])
Complementary Reinforcement Backpropagation (Ackley and Littman [1])
Learning Rate Adaptation
– Angle-Driven (Chan and Fallside [12])
– by Sign Change (Silva and Almeida [84])
– SuperSAB (Tollenaere [88])
– Delta-Delta (Sutton [6])
– Delta-Bar-Delta (Jacobs [45])
– Extended Delta-Bar-Delta (Minai and Williams [60])
– Marquardt-Levenberg method (Hagan and Menhaj [33])

– Vogl's method (Vogl, et al [93])
– Resilient Backprop (Riedmiller [73])

Heuristics on Algorithms

Weight Heuristics
– Initial Weight Range (Hecht-Nielsen [38])
– Initial Weight Geometry (Hamey)
Learning Rate Calculation (Eaton and Olivier [19])
Decreasing Learning Rate (Darken and Moody [16])
Update Schemas
– Epoch-based updates (Plaut, et al [71])
– Descending Epsilon (Yu and Simmons [104])
– Selective Updates (Huang and Huang [43])
Restarting the Training (Fahlman [24])
Varying η with Fan-In (Plaut, et al [71])
– Varying η with $\sqrt{\text{Fan-In}}$ (Becker and Le Cun [10])
Noise Introduction (Plaut, et al [71])
Non-Linear Error Functions (Fahlman [24])
Cross Validation (Amari [2])

Regularisers

Weight Decay (Hinton [39])
Weight Costing (Rumelhart [75])
Network Energy (Chauvin [13])
Constrained Backpropagation (Chauvin [14])
Competitive Units (Kruschke [53])

Techniques on Networks

Different Neurons
– Threshold Nodes, Heavyside/Sign Nodes
– Sigmoid Prime Offset (Fahlman [24])
– Symmetric Nodes (Stornetta and Huberman [87])
– Coupled Neurons (Fukumi [30])
– Simple Squashing Function (Elliot [20])
Multi-Valued Networks
– Complex Nodes (Kim [49]) (Leung [58])
– Complex Weights (Little [59])
Unit Splitting (Plaut, et al [71])
Weight Sharing (Rumelhart, et al [77], Nowlan and Hinton [68])
Recurrent Nodes (Jordan [47]) (Elman [21])
Recurrent Networks (Plaut, et al [71])
Iterative Networks (Rumelhart, et al [76])

Extra Output Nodes (Yu and Simmons [105])
Extra Hidden Layers (Sontag [86])
– Filtering/Preprocessing Input Layer
Piecewise Linear Structure (Batruni [7])

Generative Networks

Cascade Correlation (Fahlman and Lebiere [26])
– Recurrent Cascade Correlation (Fahlman [25])
Skeletonisation / Pruning (Sietsma and Dow [83], Le Cun, et al [55])
Node Sensitivity (Mozer [65])
Dynamic Node Creation (Ash [3])

1.2 Notation

The notation described below will be used throughout the report.

t	unit of time/epoch number
O	$(O_1, O_2, ...O_n)$ actual network output vector
D	$(d_1, d_2, ...d_n)$ desired network output vector
I	$(I_1, I_2, ...I_n)$ network input vector
y_i	output of neuron i
x_{ij}	input to neuron i from neuron j
z_i	sum of inputs to neuron i (before activation function applied)
δ_i	error derivative term at node i
θ	node bias
$E(W)$	error of the network with weight vector W
$E'(W)$	derivative of network error with respect to weight vector W
w_{ij}	weight of path from unit i to unit j
$C(W)$	cost of a network with weight vector W

f	(sigmoid) activation function
f'	derivative of activation function
η	learning rate (constant or varied)
α	momentum constant
h	weight decay multiplier
g	activation function gain
σ	sigmoid prime offset
μ	quickprop maximal weight growth factor

Chapter 2

Algorithms in Backpropagation

Algorithms are mainly derived formulae and rules which mathematically describe the weight update method. They may apply to other parameters of the network, such as the learning rate or activation gain term.

2.1 The PDP Research Group

The original backpropagation algorithm of the PDP Research group [76] is derived fully in Appendix A. There, the algorithm obtained implements weight changes proportional to the error in the value fed forward by the weight (credit distribution), and in the opposite direction. This performs gradient descent in error-weight space, seeking local minima in the hope that a global minimum will be reached. The amount of change is calculated in a different manner for the output and hidden layer nodes.

When a presentation (p) is given at time t, with network outputs of $O_{p,i}$, each node's weight w is changed such that:

$$\begin{aligned} w(t) &= w(t-1) - \eta\delta_i y_j \\ \delta_i &= \begin{cases} f'(z_i)(O_{p,i} - d_{p,i}) & \text{(output layer)} \\ f'(z_i)\sum_h \delta_h w_{hi} & \text{(hidden layers)} \end{cases} \end{aligned} \tag{2.1}$$

where η is the learning rate, which Rumelhart held constant, but others have varied. The activation function used by Rumelhart (and generally used since then) has a sigmoid shape (see Figure 2.1):

$$f(z) = \frac{1}{1 + e^{-(z-\theta)}}$$

where z is the sum of all weighted inputs and θ is the activation bias for a given node. Alternatively, a bias may be implemented as an extra node of binary value 1 permanently connected to each node, and θ is the weight connecting the bias to the node.

The derivative of this activation function in the case where the bias is incorporated in the sum, z, is given by:

$$f'(z) = f(z).(1 - f(z))$$

Alternatives to the sigmoidal activation function are reviewed in Section 5.1.

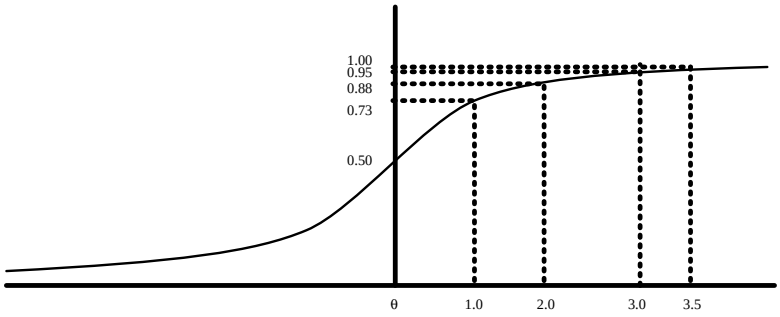

Figure 2.1: Sigmoid Activation Function

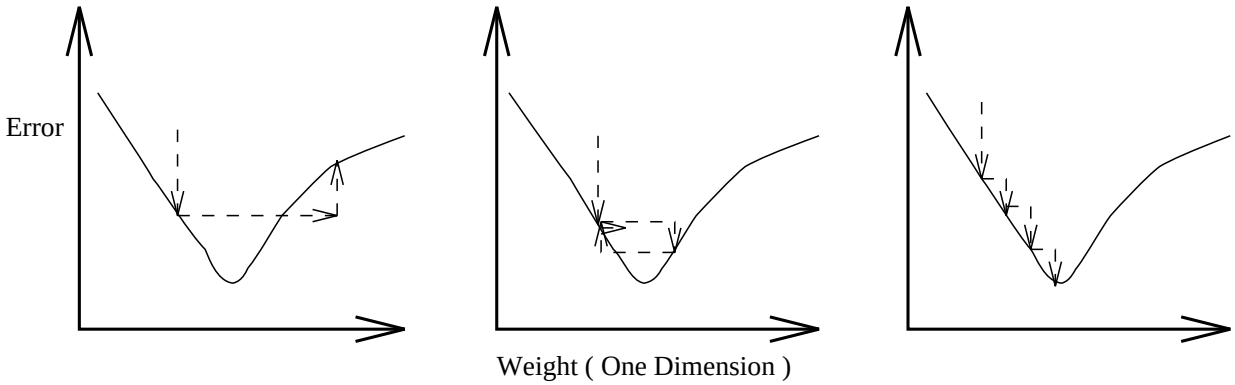

Figure 2.2: Effects of Changing Eta: Instability, Oscillation or Slow Gradient Descent

A momentum term may be added to Equation 2.1 which tends to propel the weights in the same direction as their previous change by a factor of α, the momentum, giving:

$$w(t) = w(t-1) - \eta\delta_i y_j + \alpha(w(t-1) - w(t-2)) \tag{2.2}$$

A value for η defines the learning rate, or the proportion of error at this presentation taken into account in the current weight change. The momentum of the node, α, is the effect of previous weight changes on the current one and has a stabilising effect. This is usually only useful if the direction of change is relatively correct, and learning continues toward the minimum. Momementum allows larger values of η to be employed because of this stabilising effect when learning is in the right direction.

Figure 2.2, taken from Sietsma [82], shows examples of some of the detrimental effects of varying η. If the value is too large, then the network may be unstable and overshoot the minimum in weight space, or else a network may oscillate around a minimum without any further learning. If η is too small, then the gradient descent will be slow, taking many epochs to reach the minimum.

Rumelhart, et al, [76] used values of 0 (no effect) or a range of 0.5 up to 1.0 for α and values for η ranging from quite small, 0.25, up to 0.5. Rumelhart claims little improvement in learning for a network which has momentum over an equivalent network with a lower learning rate. Obviously there is a great speed difference. Other implementors, such as Fahlman [24] used values up to 2.0 for η, without momentum.

This original backpropagation algorithm will henceforth be referred to as BP (with or without momentum).

2.2 Newton's Method

All second-order techniques described in the following sections are based on *Newton's Method* of using the first three terms of the Taylor expansion at the current point to describe the shape of the weight space and predict the minimum. The method, in itself, is not used in any practical application, but changes to this basic idea have given birth to Quickprop (Section 2.5), Delta-Delta (Section 2.7) and its associates, and Conjugate Gradient Techniques (Section 2.16). The following description of Newton's method comes from Battiti [8].

The Taylor expansion at the current point in weight space, $W(t)$ (shown simply as W hereafter), given a model (m_t) for the error E in any new direction $W + S$ is given by:

$$E(W+S) \approx m_t(W+S) \stackrel{def}{=} E(W) + \nabla E(W)^T S + \frac{1}{2} S^T \nabla^2 E(W) S$$

this is then solved to find a zero gradient:

$$\nabla m_t(W + S^N) = 0$$

for some final step N, where S^N is the Newton direction. The linear system to be solved can also be expressed as:

$$\nabla^2 E(W) S^N = -\nabla E(W) \tag{2.3}$$

If the model is a good one, and the Hessian matrix $(\nabla^2 E)$ is positive definite, then one iteration will reach the minimum. However, this iteration will require solving Equation 2.3, which will be of order $O(N^3)$.

If the Hessian is not positive definite, then an *infinite* step may be required in certain directions to minimise the model in Equation 2.3. This would indicate possibly a local minimum in weight-space which is really a saddle-point – in this case a negative curvature would lead away from the minimum if the gradient was close to zero. However, this method cannot easily deal with that situation.

Ways of approximating the Hessian matrix form the basis of other second-order techniques.

2.3 Marquardt-Levenberg

If computation complexity is not an issue, then the Marquardt-Levenberg method, as discussed by Hagan and Menhaj [33] is an improvement on the Gauss-Newton method of finding the minimum in error-weight space through its use of a variable learning rate.

If the weight update rule from Equation 2.3 is rewritten thus (where $S \approx 0$, and $W(t)$ is represented by W for clarity):

$$\Delta W = -[\nabla^2 E(W)]^{-1} \nabla E(W)$$

then the modification is given by:

$$\Delta W = -[\nabla^2 E(W) + \mu I]^{-1} \nabla E(W)$$

where μ is multiplied by a factor β when $E(W)$ would increase, or is divided by β when $E(W)$ would decrease. Initial values used were $\mu = 0.1$ initially, and $\beta = 10$.

As μ becomes large, this algorithm approaches gradient descent.

2.4 Second-Order Momentum

A simple method of applying a second-order momentum term to the back propagation algorithm was introduced by Pearlmutter [70]. The modification to the standard back-propagation rule, including the decay of previous weight changes is given by

$$\Delta w(t) = -\eta \frac{dE}{dw} + \alpha \Delta w(t-1) + \beta \Delta w(t-2)$$

The author recommend a value of $\beta = \frac{\alpha - 1}{3}$. This paper also recommends ideal values for α and η for a given problem.

2.5 Quickprop

Of the speed-up modifications to BP, Fahlman's *Quickprop Algorithm* [24] is one of the most well-known. As a second-order technique, the simple approximation employed allows easy calculation at each epoch/presentation, and the speed-up achieved by using the technique means an overall improvement in learning speed by a factor of up to 10. The loss in learnability, or proportion of learned networks, is generally small.

The main result of Fahlman's algorithm describes a new equation for weight change ($\Delta w(t)$):

$$\Delta w(t) = -\eta \frac{dE}{dw} + \frac{\frac{dE}{dw}(t)}{\frac{dE}{dw}(t-1) - \frac{dE}{dw}(t)} * \Delta w(t-1)$$

The change in a weight is the same as the change defined in BP (see Equation 2.1) plus a term which approximates the second order derivative of error with respect to weight, times the previous change in weight.

The gradient descent term is essential for learning where the current derivative or previous weight change is small. Fahlman also introduces a *maximal growth factor* (μ) which is used as an upper limit of the weight change, to avoid oscillation. It is used as an indicator of local minima which may cause weights to grow rapidly without additional learning. A value for μ of 1.75 or 2.25, depending on the problem and the learning rate, is reasonable to avoid excessive oscillation.

Veitch and Holmes [92] claim that this maximal growth factor is superfluous and that their experiments show that the *Training Restart* heuristic described in Section 3.7 takes care of the problem.

The pseudo-code which Veitch and Holmes use to implement their version of the quick-prop algorithm is shown in full in Appendix B.

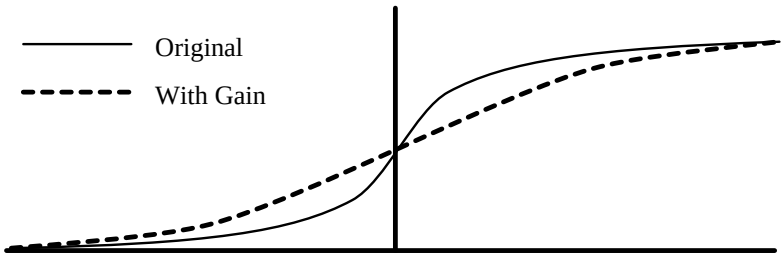

Figure 2.3: Effects of Gain Variation: changing the slope of the sigmoid function

The momentum term is dropped from Fahlman's algorithm as its function is taken over by the second order term which will follow the curvature of weight space instead of its direction. Adding a momentum term has little or no effect on learning speed.

2.6 Gain Variation

Dynamically varying the gain in the nodes of a network is a method of modifying the problem being solved. This does not modify the network physically, and has the effect of changing the algorithm used in solving the problem by varying the gain relative to the error.

Gain variation is used to 'stretch' the sigmoid function so that:

a) the slope is smaller, therefore the derivative of the function, and change in weights (see Equation 2.1), becomes smaller; and

b) because of (a), an input sum of greater absolute value is required before the output of the node approaches either binary 0 or 1.

The overall effect is that learning is *slowed down* somewhat over time, and solutions are less likely to be missed due to oscillation or overshoot when the network is far from the minimum, or conversely, learning can become faster if the sigmoid approaches the step function.

Plaut, Nowlan and Hinton [71] and Kruschke and Movellan [53] have investigated the effects of gain variation. The basic principle is to use a slightly modified sigmoid function:

$$f(z) = \frac{1}{1 + e^{-gz}}$$

where g is the gain term which is varied from some initial value, generally 1, by gradient descent similar to the variation of weights described in Section 2.1 and derived in Appendix A.

The gain variation is then described by:

$$g(t) = g(t-1) - \eta_g(t)\frac{\partial E}{\partial g}$$

This introduces the new parameter η_g which defines the amount by which the error in the gain term is taken into account in updating the term. This correlates to η, the learning rate, when applied to modifying the weights.

The equation simplifies to incorporate the derivation of δ in Equation 2.1 for each node i with output y_i:

$$g_i(t) = g_i(t-1) - \eta_g(t)\delta_i y_i$$

The effect of the gain term can be seen in Figure 2.3: the original sigmoid function (solid line) is stretched to a new shape (dotted line) to give a smoother transition from 0 to 1. This means that, as the sigmoid becomes flatter, the output is decreased for inputs greater than, and increased for inputs less than the threshold value.

Likely values for η_g provided by Plaut are 0.0 (no change) up to 0.06.

2.7 Delta-Delta

Delta-Delta is a backpropagation derived algorithm which varies the learning rate. It was introduced by Sutton [6] as an improvement which applies a gradient descent-like effect to the learning rate.

The learning rate is varied at each weight in the network. Weights are updated as they are in BP, but effort is applied to *train* the learning rate at each epoch as well.

If $G(t)$ is the error function which is to be minimised through learning rate update (as opposed $E(t)$, the error with respect to the weight vector), then gradient descent implies that the learning rate for node i be updated by:

$$\Delta\eta_i(t) = \gamma\frac{\partial G(t)}{\partial \eta_i(t)}$$

and $\frac{\partial G(t)}{\partial \eta_i(t)}$ can be simplified, so that:

$$\Delta\eta_i(t) = \gamma\frac{\partial E(t)}{\partial w_i(t)}\frac{\partial E(t-1)}{\partial w_i(t-1)}$$

where γ is a small positive step size parameter held constant and performing a similar function in this rule to the learning rate in BP.

According to Jacobs [45], Sutton's algorithm, does not handle weight spaces where the first and second derivatives are small (shallow sloped and flat) or the second derivative is very large (steeply curved).

2.8 Delta-Bar-Delta

A second-order technique which is based on the *Delta-Delta* algorithm (Section 2.7) is Jacobs' *Delta-Bar-Delta* [45] learning rule.

In the Delta-Bar-Delta algorithm, the new update rule for the learning rate in unit i is

$$\Delta\eta_i(t) = \begin{cases} \kappa & \text{if } \overline{\delta}(t-1)\delta(t) > 0 \\ -\Phi\eta(t) & \text{if } \overline{\delta}(t-1)\delta(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

$$\begin{aligned} \text{where } \delta(t) &= \frac{\partial E(t)}{\partial w(t)} \\ \overline{\delta}(t) &= (1-\theta)\delta(t) + \theta\overline{\delta}(t-1) \end{aligned}$$

Here, $\delta(t)$ is the partial derivative of error with respect to weights and $\overline{\delta}(t)$ is an exponential average of the current and past derivatives. Jacobs uses values for κ, the linear growth rate, varying from 0.35 for complex tasks up to 5.0 for simple quadratic problems; ϕ, the exponential decay rate, ranges from 0.1 up to 0.35, and θ, the proportionality of the previous directions in the calculation, should have a value of around 0.7.

An interesting and useful effect of the Delta-Bar-Delta method is that the learning rate will increase linearly (by a fixed value of κ) when the slope of weight space is in the same direction as the collective slopes of previous weight-points, but will decrease at an exponential rate (see Equation 2.4) when the slope is in the opposite direction to previous slopes. This gives a rapid reaction to changes in the shape of weight-space and avoids most oscillation and overshoot problems associated with BP. A momentum term was used by Jacobs in conjunction with Delta-Bar-Delta to give increased speed without loss of learnability.

2.9 Extended Delta-Bar-Delta

An extension on Jacobs' *Delta-Bar-Delta* (Section 2.8) is Minai and Williams' [60] improved version, which incorporates changes to learning rate as well as momentum (which *Delta-Bar-Delta* does not handle at all). This modification also includes a ceiling value for the increase in parameters. There is now an exponentially decreasing function for changes to learning rate and momentum.

An added heuristic is 'memory', whereby if the error increases beyond some λ times the lowest error met so far, then the network is restored to that point with different learning rate and momentum (to avoid following that same path).

The update rule for the learning rate for weight i, j is given by:

$$\eta_{ij}(t+1) = \min[\eta_{max}, \eta_{ij}(t) + \Delta\eta_{ij}(t)] \tag{2.5}$$

where

$$\Delta\eta_{ij}(t) = \begin{cases} \kappa_l \exp(-\gamma_l|\overline{\delta_{ij}}(t)|) & \text{if } \overline{\delta}(t-1)\delta(t) > 0 \\ -\Phi_l\eta_{ij}(t) & \text{if } \overline{\delta}(t-1)\delta(t) < 0 \\ 0 & \text{otherwise} \end{cases}$$

The update rule for the momentum for weight i, j is given by:

$$\alpha_{ij}(t+1) = \min[\alpha_{max}, \alpha_{ij}(t) + \Delta\alpha_{ij}(t)] \tag{2.6}$$

where

$$\Delta\alpha_{ij}(t) = \begin{cases} \kappa_m \exp(-\gamma_m|\overline{\delta_{ij}}(t)|) & \text{if } \overline{\delta}(t-1)\delta(t) > 0 \\ -\Phi_m\alpha_{ij}(t) & \text{if } \overline{\delta}(t-1)\delta(t) < 0 \\ 0 & \text{otherwise} \end{cases}$$

The definitions for $\delta_{ij}(t), \overline{\delta_{ij}}(t)$ are as given in Section 2.8. Values of $\kappa_l = 0.095, \Phi_l = 0.1, \gamma_l = 1, \eta_{max} = 10.0, \kappa_m = 0.1, \Phi_m = 0.3, \gamma_m = 1, \alpha_{max} = 0.9$ were used by the authors with some success.

2.10 Angle Driven Learning Rate Adaptation

Another method of adapting the learning rate is given by Chan and Fallside [12]. If we define the error with respect to the set of all patterns thus:

$$\nabla E = \sum_p \frac{\partial E_p}{\partial w_{ij}}$$

for all patterns, p, and all nodes i and j, then we can use the angle, θ, between ∇E and $\Delta w(t-1)$, the change in a weight at epoch t, to adjust the learning rate. The idea is to keep this angle at 90°; if it is larger, decrease it, if smaller, increase it.

Given:

$$\cos\theta(t) = \frac{-\nabla E(t).\Delta w(t-1)}{|\nabla E(t)| * |\Delta w(t-1)|}$$

adapt the learning rate such that

$$\eta(t) = \eta(t-1) * (1 + \kappa * \cos\theta(t))$$

and adapt the momentum such that

$$\alpha(t) = \alpha(0) * \frac{|\nabla E(t)|}{|\Delta w(t-1)|}$$

The value of κ given by Chan is 0.5, but Schiffmann, et al [80] claim that 0.1 is more useful for larger problems.

2.11 Learning Rate Adaptation through Sign Changes

Silva and Almeida [84] employ an algorithm for changing the learning rates for each weighted path from node i to j dynamically. Based on the signs of the last two gradients of error, each learning rate, η_{ij}, starting from some initial value, $\eta_{ij}(0)$, is increased while there is no sign change, and decreased when there is.

$$\eta_{ij} = \begin{cases} \eta_{ij}(t-1) * u & \text{if } \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\ \eta_{ij}(t-1) * d & \text{otherwise} \end{cases} \tag{2.7}$$

where u and d should keep the relationship $u \approx 1/d$. The values used for u were 1.1 – 1.3 and d had values of 0.7 – 0.9.

There is a back-tracking strategy which restarts an update if the total error increases, halving all learning rates.

2.12 SuperSAB

Based on the work of Silva and Almeida (see Section 2.11), Tollenaere's SuperSAB algorithm [88] changes the update rule so that weight changes which change the sign are

undone.

$$\begin{aligned} \text{if} \quad & \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\ \text{then} \quad & \Delta w(t) = -\eta_{ij}(t) * \frac{\partial E}{\partial w_{ij}} + \alpha * \Delta w_{ij}(t-1) \\ \text{else} \quad & w_{ij}(t+1) = w_{ij}(t-1), \Delta w_{ij}(t) = 0 \end{aligned}$$

Values used by Tollenaere for u and d were 1.05 and 0.5 respectively (the relationship described in Section 2.11 no longer applies).

Schiffmann, et al [80] modify this to add in a weight decay term, h.

$$\begin{aligned} \text{if} \quad & \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\ \text{then} \quad & \Delta w(t) = -\eta_{ij}(t) * \frac{\partial E}{\partial w_{ij}} + \alpha * \Delta w_{ij}(t-1) - h * w_{ij}(t) \\ \text{else} \quad & w_{ij}(t+1) = -w_{ij}(t), \Delta w_{ij}(t) = 0 \end{aligned}$$

They also tried modifying Equation 2.7 such that the learning rate is increased only when $\frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0$ and $\eta_{ij}(t-1) \leq \eta_{max}$ for some suitable η_{max}.

2.13 Vogl's Method

The work of Vogl, et al [93] accelerates learning (and adds some stabilisation) through the modification of both learning rate and momentum terms in batch training of a network.

There are two new terms involved, $\phi > 1$, the learning rate increase, and $\beta < 1$, the learning rate decrease. If error is reduced over an epoch of training, then the learning rate is increased by a factor of ϕ in the next step. If the error increases (by 1–5 per cent), then the learning rate is decreased by a factor of β, and the momentum term is set to 0. The weights are not updated, but the epoch is done over until error is decreased. After the next successful step, the momentum is returned to its original value.

Although this description is similar to the Marquardt-Levenberg method discussed in Section 2.3, Vogl claims that this method is less heuristically oriented. This method tends to avoid over-shooting often associated with momentum; but the implementation implies that, where the error-weight space is particularly corrugated, it might take several repetitions of one epoch to find the correct direction to travel in.

2.14 Resilient Backpropagation

The claim made by Riedmiller [72, 73] for the *RPROP* algorithm is that it is resilient to target overshoot when adapting the weights of the network. The size of a weight's change is given independently for each weight as a variable $\Delta_{ij}(t)$ at any given time. This weight update value is used to determine the direction of change for the weight, thus:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t) & \text{if } \frac{\partial E(t)}{\partial w_{ij}} > 0 \\ +\Delta_{ij}(t) & \text{if } \frac{\partial E(t)}{\partial w_{ij}} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

The way to update the value for $\Delta_{ij}(t)$ is similar to the method whereby a learning rate might be adapted (see Sections 2.12, 2.8).

$$\Delta w_{ij}(t) = \begin{cases} \eta^+ * \Delta_{ij}(t-1) & \text{if } \frac{\partial E(t-1)}{\partial w_{ij}} * \frac{\partial E(t)}{\partial w_{ij}} > 0 \\ \eta^- * \Delta_{ij}(t-1) & \text{if } \frac{\partial E(t-1)}{\partial w_{ij}} * \frac{\partial E(t)}{\partial w_{ij}} > 0 \\ \Delta_{ij}(t-1) & \text{otherwise} \end{cases} \tag{2.9}$$

$$\text{where } 0 < \eta^- < 1 < \eta^+$$

Values given by Riedmiller are $\eta^+ = 1.2$ and $\eta^- = 0.5$. These were obtained experimentally.

When a partial derivative changes sign (the last weight update caused the network to overshoot its target), then the update value is decreased; otherwise, if the partial derivative is in the same direction, then the weight update value will increase (for better convergence).

2.15 Backpropagating Desired States

As an alternative to backpropagating the error of the network through the nodes to modify weights (see Appendix A), Plaut, Nowlan and Hinton proposed *Backpropagating Desired States* [71]. This epoch-based algorithm incorporates a different formula for weight updates.

The amount by which a node contributes to an incorrect output is more or less ignored – the important part is the desired value of output nodes and the contribution that any hidden layer node makes to an incorrect output. Starting with the penultimate layer (connected to the outputs themselves), tally the number of output nodes which an incorrect value on that node would affect. There is also a value calculated to indicate the concurrence of these output nodes, so that if they are all wrong, then a definite change is required in the hidden node, but if half are wrong and half right, then no change should be made at this time.

The desired state of a hidden unit, i, is defined as:

$$d_i = \begin{cases} 1 & \text{if } \sum_j w_{ji}(2d_j - 1)c_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

The criticality factor for a unit is given by

$$c_i = \frac{|\sum_j w_{ji}(2d_j - 1)c_j|}{\sum_j |w_{ji}(2d_j - 1)c_j|}$$

The sign of $w_{ji}(2d_j - 1)$ determines the direction, c_j is the *criticality* and determines the amount of influence on unit i by output unit j.

Each output unit has a fixed criticality of 1 for calculations and a desired state provided by the problem definition. Layers other than the penultimate use the criticality and desired state values calculated in their own weight updates, moving backwards through the layers.

The new weight update rule is:

$$w_{ij}(t) = \eta(d_i - O_i)c_i y_j$$

Weight changes are carried out at the end of each epoch (full cycle of presentations). Momentum may be added to the weight update rule if desired.

The purpose of the development of this algorithm was to find a variation of backpropagation which scaled well and could handle situations where the error gradient is small, yet the difference in desired states is large. Plaut tested this algorithm on tasks such as the 4-2-4 encoder and random binary associations with results indicating that this algorithm was not as good as BP. However, for a test in which backpropagation fails, a 1-10×1-1 encoder (see Section 5.8), backpropagating desired states succeeds. In similarly structured networks of larger size, backpropagating desired states improves faster when adding more units to a hidden layer than BP.

2.16 Conjugate Gradient Techniques

A conjugate gradient technique is an algorithm in which the error is minimised along successive lines conjugate to all previous directions searched in weight space, gradually reaching a minimum in weight space. These are described in Johansson et al [46], Battiti and Masulli [9].

Any two weight vectors a and b are conjugate if $aHb = 0$, where H is the Hessian matrix with components being the second derivatives of the error with respect to the weights.

$$H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$$

Conjugate methods take a linear combination of the steepest descent vector with the previous weight-change vectors such that the new vector is conjugate to all previous vectors. A linear search is performed in this direction to obtain the error minimum along that line in weight space.

As the second derivatives are often very time-consuming to calculate (for larger networks, giving larger matrices), an approximation based on the difference between successive first derivatives is used. For $G = \frac{\partial E}{\partial W}$, the direction of weight change, V, at time t is given by

$$V(t) = -G(t) + z(t, t-1)V(t-1)$$

where z is a constraint for approximating successive search directions.

There are two alternatives for calculating z:

$$\text{Fletcher-Reeves:} \quad z(t, t-1) = \frac{|G(t)|^2}{|G(t-1)|^2}$$

$$\text{Polak-Ribiere:} \quad z(t, t-1) = \frac{|G(t)|^2 - G(t) \cdot G(t-1)|}{|G(t-1)|^2}$$

After this, the change in weight is given by $\Delta W = \lambda V(t)$, where λ is determined by the linear search to find the minimum of $E(W(t-1)+\lambda V)$. There are at least three evaluations of the output of the network (and error) required with different weights, depending on the accuracy desired in the search. The method relies on evaluating the whole network to obtain the value of z, which is dependent on the norm of the error gradient vector, however, fewer iterations are required for a solution by a factor of at least 10.

2.17 Scaled Conjugate Gradient Method

An improvement on the conjugate gradient method was proposed by Møller [62], who removes the need for a line search in each successive direction by scaling the step size. This method still requires twice as much calculation as BP at each point, but is as fast as other conjugate gradient methods. The complete algorithm is described in Appendix C.

The strategy for finding the minimum in weight space is to use an iterative function in the neighbourhood of the current point. An approximation of the function uses the first or second order Taylor expansion. The next point is determined relative to the current point by a *search direction* and *step size*. If the search direction is set to the negative gradient with a constant step size, then we have backpropagation (which uses a first order approximation of the Taylor expansion). If, however, the second order Taylor expansion is used to get both the search direction and step size, one step may be sufficient to get to the minimum in weight space.

Chapter 3

Algorithm Heuristics

Heuristics may be algorithmic in nature but they tend to lie in the realms of experimentally workable, rather than theoretically proven. Many of these methods are algorithm-independent.

3.1 Network Weight Initialisation

Although the method of determining weight changes is specified in BP, there is no strict initialisation method employed, except to note that weights should start with some non-zero value (unlike Hebbian learning, where weights increase from zero to show learning), and preferably small (say, less than 1.0) to avoid an initial configuration which is difficult to unlearn during training.

The effect of weight initialisation was not realised until the work of Hecht-Nielson [37] showed that local minima were possible when looking for a solution with a backpropagation network, and that certain weight configurations would lead to local minima, while others lead (eventually) to solutions. Kolen and Pollack [50] also investigated these effects. Techniques for getting out of local minima are rare, and so methods of avoiding them are sought after. A good weight initialisation algorithm is one such method.

An initial weight range parameter, ρ, is usually employed which specifies that weights should be initialised within the range $[-\rho, +\rho]$. Most applications use a value for ρ of around 1.0. This is dependent on whether *Weight Decay* is employed (see Section 4.1).

As well as the range of weights, there is also the manner in which the weights should be selected within that range. Some methods of allocating weights include:

Totally random : Each weight is initialised to some random value within the range, regardless of the value of any other weight.

Weight groupings : Each node or group of nodes has its own range or sub-range from which the values of the weights may be (randomly) initialised.

Shaped weight vectors : The weight vector for the network or layer of nodes may take on a *shape* in n-dimensional space. This is achieved by several techniques suggested by Hamey.

a) The weight vector for the network/group may be a point in weight-space within a small hypercube at a corner of the hypercube of side 2ρ. In this case, each weight value will be 'near' $+\rho$ or $-\rho$.

b) The weight vector sits on the edge of a hypersphere of radius ρ - ie, the weight vector is normalised.

c) The weight vectors for the nodes in each hidden layer are mutually orthogonal, having at least one component different to every other node. This implies that feature detection is separated within each layer. This technique is not necessarily as useful to the output layer, where the function of the node is dependent on the desired outputs.

3.2 Learning Rate Calculation

Although many techniques suggest ways of varying learning rates, they don't often describe a method of choosing a start point. Eaton and Olivier [19] describe a method whereby a training set consisting of similar patterns might suggest a learning rate on the assumption that similar patterns would suggest similar gradients. If the training set can be divided into subsets of similar patterns, then the number of elements in each subset can be given by N_1, N_2, $\ldots N_m$. Therefore the learning rate can be given by:

$$\eta = \frac{1.5}{\sqrt{N_1^2 + N_2^2 + \ldots + N_m^2}}$$

They also set the momentum, α to a constant, 0.9. According to Schiffmann, et al [80], this does not produce useful results in some cases.

3.3 Varying the Learning Rate with Fan-In

Varying the learning rate parameter, η, has already been described as the *Gain Variation Algorithm* in Section 2.6. There are other useful, and simpler, techniques for changing the rate of learning. Varying the learning rate with respect to the Fan-In, or the number of weighted pathways leading to a node, is one such heuristic. This, like Eaton and Olivier's technique (see Section 3.2), is done once, before training begins.

Plaut, Nowlan and Hinton [71] describe this technique in conjunction with *Unit Splitting* (see Section 5.8 and Figure 5.4). Similarly, Becker and Le Cun [10] describe the worth of dividing the learning rate within a node by the square-root of the number of weighted paths leading into it.

Implementing this simple change achieves a speed-up by a factor of 4 in some cases ([71]). The reason being that the sum total of weights into any one node will be of the same order as any other node regardless of the number of pathways to it. (It is only relevant when the number of nodes in one layer of a fully interconnected network exceeds the number in the previous layer.) Therefore, no one node will have an unnecessarily high error due to a larger number of inputs, and each will tend to backpropagate an accumulated error equivalent to a node with only one pathway leading into it.

3.4 Decreasing the Learning Rate

Darken and Moody [16] describe a method whereby the learning rate is adjusted downwards dynamically from a high value on a pattern by pattern basis. If the intention is to reduce the learning rate from an intial value $\eta(0)$ to some $\eta(t)$ at training epoch t.

$$\eta(t) = \frac{\eta(0)}{(1 + \frac{t}{r})}$$

where r is used to adjust the learning rate with respect to the total training period. After r steps, the learning rate should be halved. The value for r can therefore only be achieved by trial and error.

As this method is not affected by the learning itself, it is more heuristic than algorithmic in nature.

3.5 Descending Epsilon Technique

Yu and Simmons [104] developed an algorithm, or heuristic, within which backpropagation may be applied. Called *Descending Epsilon*, the technique modifies the problem being solved by starting with a relatively simple goal for error reduction, and slowly increasing the required correctness of the network until it reaches a state where the implementor is satisfied that the problem has been learned.

Unlike BP, which is generally epoch-based, Descending Epsilon is a pattern-based technique, that is, for each pattern in the problem there is a presentation and the error is calculated at each node. If any node is in error, then backpropagation is performed for this pattern. This goes on until all patterns produce an error less than the current threshold. The threshold is then reduced, and the sequence starts again. The pseudo-code for the algorithm is given in Appendix D.

The advantage of this algorithm is that large changes in any one weight are less common due to the less stringent error threshold at any one time, and the effects of one weight change on another weight are diminished as all weights with an error beyond the given threshold will be changed during one backpropagation until they are all beneath that error threshold.

The disadvantage of the technique is in speed. Obviously this algorithm can cause backpropagation learning to slow down considerably when solving a problem with fixed intermediate error steps. However, because of this, the learnability of a problem may increase and good generalisation may occur.

Parameters to this algorithm are the initial and final error thresholds and the threshold stepsize.

3.6 Selective Updates

The *Selective Update* algorithm of Huang and Huang [42, 43] is an update schema which changes only the weights attached to nodes which satisfy certain criteria. There is also a slightly modified objective error function to be minimised here.

There are two criteria to be satisfied before a node's weights may be considered for update. Define a pattern, I_p to be classified correctly if its value is within 0.5 of the correct binary value, and misclassified otherwise.

$$|O_j(I_p) - d_j(I_p)| \begin{cases} \geq 0.5 \text{ implies } I_p \text{ is misclassified} \\ < 0.5 \text{ implies } I_p \text{ is classified} \end{cases}$$

Define a pattern to be within the insignificant update region if it is within some tolerance, ϵ, of an error of 1. This tolerance could be the machine's own limitation. Combine these two criterion to give the set of valid update data V_j for the jth output neuron satisfying:

$$0.5 \leq |O_j(I_p) - d_j(I_p)| < 1 - \epsilon$$

The new objective function to replace Equation A.2 is

$$E(W) = \frac{1}{2} \sum_j \sum_{I \in V_j} [O_j(I_p) - d_j(I_p)]^2 \tag{3.1}$$

This does not in any way change the rest of the derivation of the weight update rule described in Appendix A, however Huang and Huang claim that local minima are avoided using this technique. Updates are performed only when a node is learning, whereas BP will update regardless.

3.7 Restarting the Training

If a network cannot learn a solution 100% of the time (being possibly trapped in local minima due to bad weight initialisation), but generally does succeed, then there is some point in the training after which further effort is really wasted. Fahlman [24] introduces the idea of restarting the training of a network which has taken 'too long' to learn, but does not specify how long that should be. The investigations of Hamey [34, 35] quantify some models which show how to measure where the best point to restart learning is. This information gives an heuristic for finding the point for restarting a network's training based on data from previous trials.

3.8 Noise Introduction

Generalisation in solving a problem is defined as the ease with which a network can give a correct output for patterns not presented during the learning phase (or the ability of a network to do any such extension from the learning set). Noise introduction is the simplest heuristic to improve generalisation.

Noise is introduced by adding a small perturbation to the value of each presentation at the input layer. The nature of this perturbation could be one of the following:

Random : each noise value is chosen at random within a range $[-\rho_{noise}, +\rho_{noise}]$.

Normal : noise is distributed normally with mean (μ_{noise}) and standard deviation (σ_{noise}).

Plaut, Nowlan and Hinton [71] are attributed with being the first to use *artificial* noisy inputs when classifying signals. They showed that adding noise during training allowed the network to recognise similarly noisy signals during the testing stage. Noise is a normal part of the problem which those involved in speech recognition have always had to contend with, and so it was an obvious idea to experimenters like Waibel [95] that generalisation would come about with noise added to synthetic problems. Holmström [41] also investigated additive noise.

3.9 Non-Linear Error Functions

Fahlman [24] describes many problems in BP, and one of his main complaints is with the operation of the neuron. Looking at the backpropagation derivation in Appendix A, we can see that the error propagated from the output layer for a normal sigmoidal neuron (see Section 2.1) is described by:

$$\begin{aligned} \Delta w_{ij} &= -\eta \delta_i y_j \\ \delta_i &= f'(z_i)(O_i - d_i) \\ f'(z_i) &= O_i(1 - O_i) \end{aligned}$$

The weight change is proportional to the difference between the desired and actual outputs, and also the product of the output O_i with $(1 - O_i)$. Learning will stop as O_i approaches d_i, or where O_i approaches 0 or 1, that is, as the correct response is learned, or else the actual response approaches either of the two extremes.

The first problem can be avoided if, instead of the difference between desired and actual outputs, the hyperbolic arctangent of that difference is used, taking on values between 1 and -1. The hyperbolic arctangent is defined up to $\pm\infty$, but Fahlman approximates this by using a value of -17.0 below -0.9999999 and +17.0 above +0.9999999.

This is not really a change to the algorithm – it is closer to an heuristic than a technique, but it is a way of changing the function of the output layer neurons without having any other effect on the network.

The second problem mentioned, where the derivative of the sigmoid function ($f'(z_i)$) approaches 0, is discussed in section 5.6.

3.10 Thresholds, or *Don't Care* Regions

It is often desirable to allow the network to ignore some training patterns when they are 'near enough' to the desired result, so as to concentrate on modifying the network to handle patterns which are quite distinctly wrong. One technique for doing this is to use a threshold below which a low binary value is assumed, and a margin above that threshold which indicates that a value is neither high nor low (giving a lower bound for high values). This technique is discussed by Fahlman [24] and Bakker [4, 5]. In the case of Hamey [35], the threshold is given as τ_c, with is the upper bound on low binary values, and the lower bound on high values is $1 - \tau_c$ (with a value of $\tau_c = 0.4$).

This concept seems to be related to Plaut, Nowlan and Hinton's *Backpropagating Desired States* [71] as discussed in section 2.15, approaching it in extreme cases where the

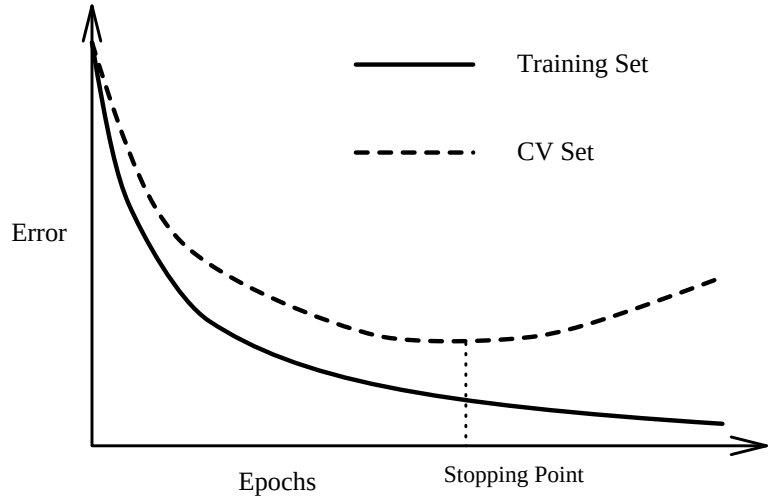

Figure 3.1: Learning Curve on Training and Cross-Validation Sets

margins are very small. However, that algorithmic technique is a major variation on the back propagation algorithm, whereas *Don't Care* regions are a simple modification.

3.11 Repeat Until Bored

Munro [66] suggests a method of selecting training patterns in such a way as to (possibly) decrease the learning time required. The basic idea is that a new parameter, β is used as an error criterion value, and a randomly selected pattern is presented to the network until the error (for that pattern) falls below β. At this point, another random pattern is selected, and training continues in this manner until all patterns have been learned by the network.

The choice of the boredom parameter, β is critical: if too small, no convergence will take place on the first pattern; if too large, no benefit will be gained over simple random selection of patterns.

3.11.1 Cross Validation

Although used as a statistical technique to avoid over-fitting, cross validation has been formalised and used specifically in the context of neural networks for some time, as discussed in Sarle [78], and Sjöberg and Ljung [85]. The basic idea is that the data must be divided into three sets (at least): the training set (the largest component), the test set (which should be of the same nature as the training set, and is used to test for generalisation), and a cross validation set (CV set). The composition of the CV set should be independent of the other two, and the size of the set has been investigated by both Amari, et al [2] and Jackel, et al [44].

Although the process of cross validation may slow down training significantly, in that the CV set must be presented to the network every k training epochs, it is used as a

stopping condition on training, and may therefore avoid unnecessary training. Plotting the training set and CV set error together will, in a log-like learning curve, indicate when training should stop (see figure 3.1). When the CV curve rises, then it is most likely that no further training will improve the generalisation error. The technique therefore requires that a copy of the network weights at the point of lowest CV error must be kept.

In cases where data is short, the average of a combination of runs of cross validation training might be used to determine the correct point to stop training. For instance, for a fixed data set, some portion must be used for testing, and the rest might be divided into portions of one tenth of the set each. One portion is used in each of ten runs as the CV set while the rest is used for training. The best stopping time for training the network is the average stopping time of the ten runs.

A discussion of the uses of cross validation can be found in Schaffer [79]. This shows the wide-ranging benefits of such a method on back propagation and also some statistical classification methods.

Chapter 4

Regularisers

Regularisers are cost terms added to the error function being minimised. BP minimises the difference between desired and actual outputs (error, E_{net}) with respect to the values of the weights in the network, W, (see also, Equation A.1):

$$E_{net}(W) = \frac{1}{2}(O - D)^2 = \frac{1}{2}\sum_{i=1,m}(O_i - d_i)^2 \tag{4.1}$$

where O is the output vector of the network and D is the desired output, with i running over all m output nodes. The factor of $\frac{1}{2}$ is not always used, but it simplifies the derivation (see Appendix A).

Changes to this definition are referred to as alternative error functions. An example is the non-linear error function used by Fahlman [24] (see Section 3.9), using the hyperbolic arctangent of the difference between desired and actual output.

Additional terms intended to constrain the network while solving the problem are regularisers. As well as minimising the error defined in Equation 4.1, implementations minimise some other function, such as the sum of weights in the network.

Most of these regularising techniques are discussed in Crespo [15].

4.1 Weight Decay

Plaut, Nowlan and Hinton [71] introduced the most well-known regulariser in *Weight Decay*. The basic principle is to minimise the sum of squared weights with the error, thus preventing them from growing too large and encouraging redundancy of hidden units. Using E_{net} from Equation 4.1, we use the cost with respect to weights (C_{decay}) to define the total cost of the network (C_{total}) as:

$$\begin{aligned} C_{decay}(W) &= \sum_{i=1,n} w_i^2 \\ C_{total} &= E_{net} + C_{decay} \end{aligned} \tag{4.2}$$

where there are n nodes in the network.

Taking the derivative of this equation gives a slightly different weight update rule for minimising the cost of the network, based on Equation A.8:

$$w_{ij}(t) = (1-h)w_{ij}(t-1) - \eta\delta_i y_j \tag{4.3}$$

where h is the weight decay term, a small constant which Plaut sets to 0.00001 – sufficient to drive small weights closer to 0 over time, but not enough to hinder learning. The value of δ_i is as defined in Appendix A. Momentum can be added to Equation 4.3 without changing the problem.

This technique was also investigated by Krogh and Hertz [51]. Notably, Plaut, et al, did not specify whether the decay term should be applied to the weight after it has been changed by the update rule, or, as Krogh and Hertz did (shown in equation 4.3), before updating the weight.

4.2 Rumelhart's Regularisers

Some regularisers were suggested by a lecture given by Rumelhart [75] describing possible directions of research in limiting the interactions between nodes or the size of the network to create better generalisation and consistency.

> "The simplest, most robust network which accounts for the data set is, on average, the network which will generalise best."

Rumelhart introduced two different measures of network complexity. C_{weight} measures the complexity of all of the weights in a network:

$$C_{weight} = \frac{w_{ij}^2}{1 + w_{ij}^2} \tag{4.4}$$

C_{unit} measures the weights in each unit, encouraging a node to be cut off from its neighbours.

$$C_{unit} = \frac{w_{ij}^2}{1 + \sum_k w_{ik}^2} \tag{4.5}$$

There is also a factor, λ, representing the proportionality taken into account when costing the network in terms of error and complexity:

$$Cost = \lambda Error(W) + (1-\lambda)Complexity(W) \tag{4.6}$$

Weigand, Huberman and Rumelhart [98] use Equation 4.4 to get a relationship:

$$\frac{\partial C}{\partial w_{ij}} = \frac{2w_{ij}}{(1 + w_{ij}^2)^2}$$

In this case, Equation 4.6 is implemented with λ starting at 1 and being increased until performance declines, that is, the network's complexity is taken more into account until it dominates the error term.

In some of these regularising techniques, training takes longer because of the need to remove nodes (skeletonisation) at some point in the learning, and then put in more effort to train the remaining nodes to solve the problem only partly solved.

4.3 Weight Costing

Within a year of Rumelhart's lecture [75], four parties undertook the work to develop his ideas. Hanson and Pratt [36] tried two different cost functions (which they called Biases) for each of the alternatives described in Section 4.2 (Equations 4.4, 4.5), with minor changes to the weight update rule in each case.

Firstly, to replace Equation 4.4, a simple quadratic weight cost, ($C = w^2$), is introduced to the network cost, giving a new update rule of:

$$\Delta w_{ij}(t) = \eta \left(-\frac{\partial E}{\partial w_{ij}} - 2w_{ij}(t-1) \right)$$

This indicates that the learning parameter, η is also a part of the weight decay term, and a value of less than $\frac{1}{2}$ is necessary for learning to occur.

This gives uniform decay for large and small weights alike. The intention is to decay the small weights to zero, allowing the larger weights to stay at some reasonable level. A better cost function is therefore $C = \frac{w_{ij}^2}{1+w_{ij}^2}$ (as given by Rumelhart), which gives an update rule of:

$$\Delta w_{ij}(t) = \eta \left(-\frac{\partial E}{\partial w_{ij}} - \frac{2w_{ij}(t-1)}{(1 + w_{ij}(t-1)^2)^2} \right)$$

This formula will tend to move all small weights to zero much faster than a simple decay technique.

When considering the cost of weights within a single unit, there are two alternatives for Equation 4.5 employed by Hanson and Pratt which group weights in a better manner. Given that the sum of weights to a hidden unit i may be given by:

$$v_i = \sum_j |w_{ij}|$$

Using

$$E_{total} = E_{weights} + C_{weights}$$

and an error derivative of:

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E_{weights}}{\partial w_{ij}} + \frac{\partial C_{weights}}{\partial w_{ij}}$$

define a hyperbolic cost function which uses the weight group in the place of individual weights, such that:

$$\begin{aligned} C &= \frac{v_i}{1 + \lambda v_i} \\ \frac{\partial C}{\partial w_{ij}} &= \frac{\lambda sign(w_{ij})}{(1 + v_i)^2} \end{aligned}$$

or define an exponential cost function, hoping for a smoother change of weights towards zero, such that:

$$\begin{aligned} C &= 1 - e^{-\lambda v_i} \\ \frac{\partial C}{\partial w_{ij}} &= \frac{sign(w_{ij})}{e^{\lambda v_i}} \end{aligned}$$

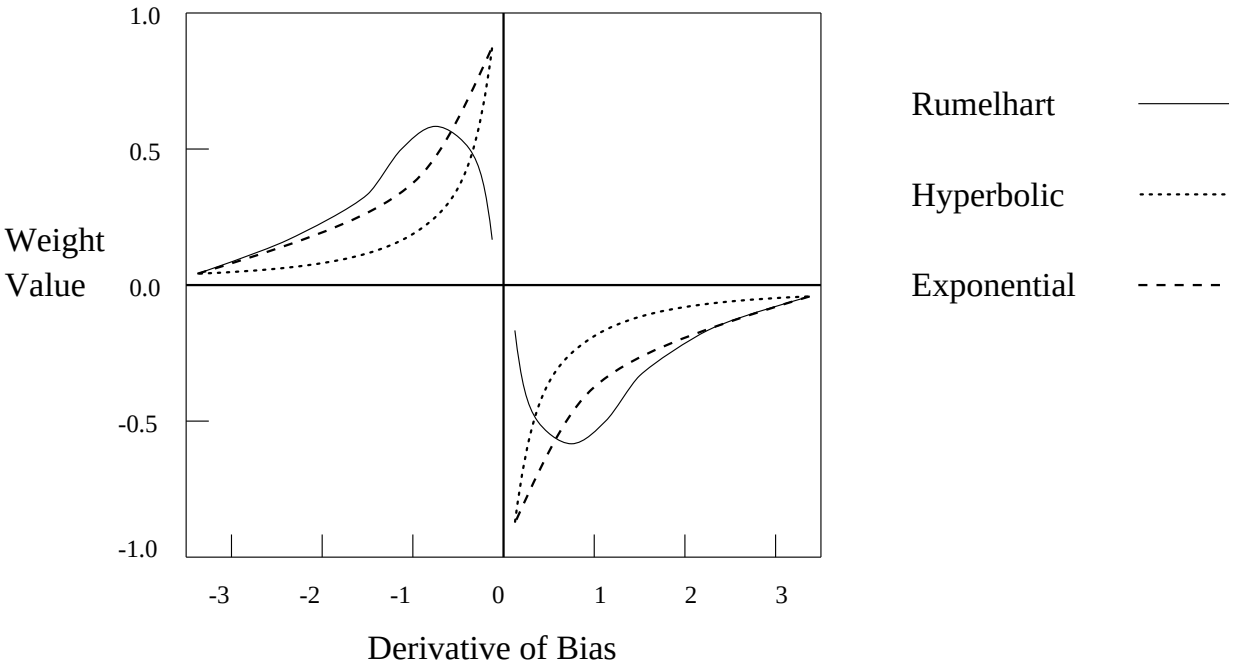

Figure 4.1: Bias function behaviour.

Hanson and Pratt claim that these two methods follow Rumelhart's intentions more closely. This is born out by Figure 4.1 (which comes from [36]). The shape of the two functions used follow Rumelhart's closely, adding a monotonic flavour. The intention of using these other Bias functions is to show the effectiveness of varying Biases when minimising network complexity.

4.4 Constrained Back-Propagation

Another author to base work on Rumelhart's lecture on generalisation [75] was Chauvin. He introduced an *Energy Function* [13] which could be applied to the weights of a network, and then modified this to produce *Constrained Backpropagation* [14].

The cost function which Chauvin first implements is:

$$\begin{aligned} C &= \lambda E_{error} + (1-\lambda) E_{energy} \\ &= \lambda \sum_p \sum_i (d_{ip} - O_{ip})^2 + (1-\lambda) \sum_p \sum_h e(y_{hp}^2) \end{aligned} \tag{4.7}$$

where p runs over all patterns, i is every node in the output layer, and h is (a subset of) the nodes in the hidden layers, each node having an output of y. The energy function, e is linear.

The combined weight update rule incorporating the minimisation of this energy term is:

$$w_{ij}(t) = w_{ij}(t-1) - \eta y_j \delta_i^*$$

where δ_i^* includes the accumulative effect of all of the layers, k, preceding the current one:

$$\delta_i^* = f'(z_i) \sum_k \delta_k^* w_{ki}$$

The energy function used by Chauvin has a derivative defined by:

$$e' = \frac{\partial e(o^2)}{\partial o_i^2} = \frac{1}{(1 + o_i^2)^n}$$

where the value for n determines the shape of the function.

In *Constrained Backpropagation*, the new cost function takes into account not only the error used previously, and the sum of weights, but also restricts the sum of outputs of hidden layer nodes. This means that as few as possible of the hidden units will be used to separate the problem space for a solution, thus enforcing simplicity.

The overall cost of the network (replacing Equation 4.1) is therefore given by:

$$C = \lambda \sum_p \sum_i (d_{ip} - O_{ip})^2 + \chi \sum_p \sum_h \frac{o_{hp}^2}{1 + o_{hp}^2} + (1 - \lambda - \chi) \sum_k \sum_j \frac{w_{kj}^2}{1 + w_{kj}^2}$$

where p is each pattern, i is each node in the output layer, h is each node in the hidden layers, and j and k run across all nodes in the network. Both λ and χ are proportionality factors which sum to less than 1.

In practice, the energy of the network will decrease rapidly in early learning (for most boolean problems), will stabilise, then again decrease in energy as weights are discarded. In tests on simple problems such as parity, a minimal network was always achieved, regardless of the number of hidden units or layers in the initial network.

4.5 Minimised Weight Product

Instead of minimising the sum of the weights at any given unit, an alternative developed by Niida, Tani, Hirobe and Koshijima [67] is to minimise the product of weights.

$$C_{weights} = \sum_i |\prod_j w_{ij}|$$

giving a relationship of:

$$\frac{\partial C}{\partial w_{ij}} = |\prod_{k \neq j} w_{ik}|$$

This particular network was used in a textiles industry application to predict the quality of felt given inputs of raw materials and chemical treatments. The initial network configuration was skeletonised using this technique to create a solution with only 15% of the complexity of the original.

4.6 Competitive Units

Kruschke [52] introduces a method of removing complexity which does not use a regulariser in the normal sense of the word. Units must compete to increase their weights in any given direction in global weight space. Every node's weights have a direction (normalised weight vector), W^*, and *gain factor* (magnitude), g, which determines the usual vector of weights used by a node, W. There may be many zero-valued weights for unused directions in a node.

Nodes will therefore tend to become better feature detectors in the hidden layers.

After each backpropagation pass, the change in a gain factor, g, for the weight of a node, i, is:

$$\Delta g_i = -\gamma \sum_{i \neq j} (W_i^* . W_j^*)^2 g_i$$

where

$$W_i = g_i W_i^*$$

Chapter 5

Techniques on Networks

Techniques which change the network in some way (but not dynamically) are very different to heuristics applied to an algorithm, as some part of the problem has changed, usually through modifying an element of the network.

5.1 Non-Sigmoidal Neurons

An obvious technique which can be applied to a network is to replace the usual sigmoidal neuron used by Rumelhart [76] (see Section 2.1, Figure 2.1) with something similar to gain some advantageous properties.

Some of these neurons have been developed as special purpose responses to various problems seen in the original algorithm or network, while some are based on ideas used in other (non-backpropagation) networks which are also found to be applicable here.

5.2 Hyperbolic Arctangent Activation

An alternative to the sigmoidal function which has a similar shape and simplicity for its derivative is the *hyperbolic arctangent* function.

$$f(z) = \mathrm{a}rctanh(z)$$

which has a derivative defined by

$$\begin{aligned} f'(z) &= 1 - [\mathrm{a}rctanh(z)]^2 \\ &= 1 - f(z)^2 \end{aligned}$$

5.3 Symmetric Neurons

The usual sigmoid activation function gives a positive value for all summed inputs to the neurons. An alternative to this is a node which gives a negative value for a negative sum.

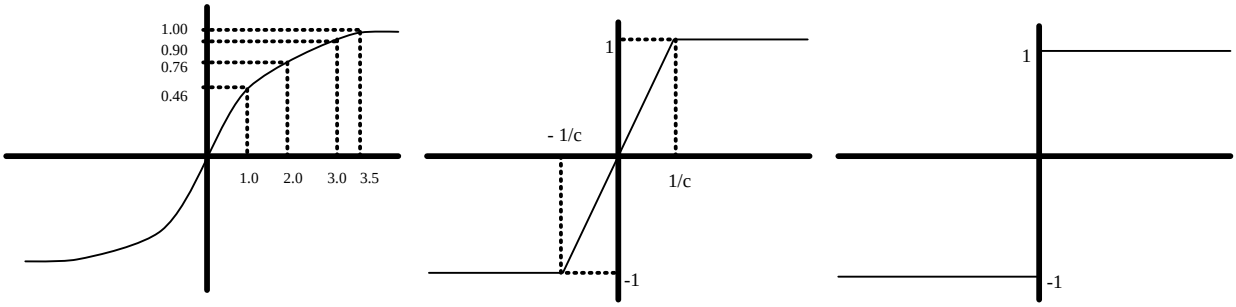

Figure 5.1: Symmetric, Linear Threshold and Signed Activation Functions

This is a symmetric sigmoidal function, as discussed in Stornetta and Huberman [87].

$$\begin{aligned} f(z) &= \tanh(z) \\ &= 2\left(\frac{1}{1+e^{-z}} - 0.5\right) \end{aligned}$$

Where z is defined as the sum of weighted inputs (see also Appendix A). This activation function has asymptotes of $f = -1$ and $f = 1$. The derivative is now:

$$f'(z) = \frac{1 - f(z)^2}{2}$$

Some symmetric problems, such as N-parity, have been learned faster or more consistently using symmetric neurons (see Fahlman [24]).

Symmetric neurons don't solve the problems described in Section 3.9, they only change them by shifting the danger area to output values at the new asymptotes of $\frac{-1}{2}$ and $\frac{1}{2}$. A *Sigmoid Prime Offset* (see Section 5.6) was also employed here by Fahlman.

The stability of symmetric nodes was also investigated by Yang [103].

5.4 Linear Threshold and Signed/Heavyside Neurons

Signed and *Linear* neurons had been used before the sigmoidal type in single layer perceptron models. These multi-linear functions make computing the output very simple, in both cases having a single value beyond certain thresholds. (See Figure 5.1)

The derivatives of the function are 0 over a large region of possible summed inputs which causes problems when backpropagating errors outside of the range $[-1/c, 1/c]$ for the linear function. The signed function has a derivative of 0 over almost all of its range (infinite at $z = 0$). A threshold neuron is a signed neuron with zero output where the summed inputs is negative.

$$threshold(z) = \frac{sign(z) + 1}{2}$$

Therefore none of these three neuron types are well suited to BP, which requires a non-zero derivative to propagate the errors, but they may be used in conjunction with other types, such as in *Coupled Neurons* in Section 5.7.

A multi-layer network made up of any of these neuron types has no advantage over a single layer.

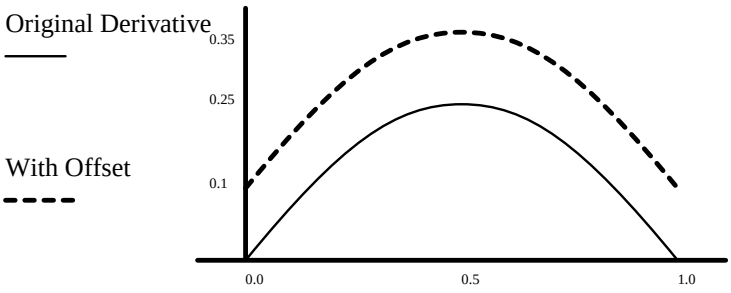

Figure 5.2: Derivative of the Asymmetric Activation Function

5.5 Simplified Squashing Function

Although most activation functions seem to be easy to implement, the use of exponentials can be compute-intensive. An alternative proposed by Elliot [20] is to use the following:

$$f(z) = \frac{z}{1 + |z|}$$

which has a derivative defined by:

$$\begin{aligned} f'(z) &= \frac{1}{(1 + |z|)^2} \\ &= (1 - |f(z)|)^2 \end{aligned}$$

It is claimed that the *operation count* (computation complexity) involved is a magnitude less than the usual sqashing functions such as the sigmoidal (see Section 2.1) and the hyperbolic tangent (see Section 5.2).

5.6 Sigmoid Prime Offset

As discussed in Section 3.9, the derivative of the sigmoid function approaches 0 as the activation of the node approaches 0 or 1 (the extremes of its output). At these points there will be little or no learning.

Fahlman's solution [24] is to add a small offset to the derivative, which he calls the *Sigmoid Prime Offset*, so that it never reaches 0. He attributes this technique to Franzini [27]. The new derivative for a sigmoid function is:

$$f'(z_i) = y_j(1 - y_j) + \sigma_{offset}$$

The value generally used for σ_{offset} by Fahlman is 0.1. It will improve the learning speed when the network is close to the solution, or farthest away (assuming binary-valued outputs). However, it may cause learning to move in a direction which increases error.

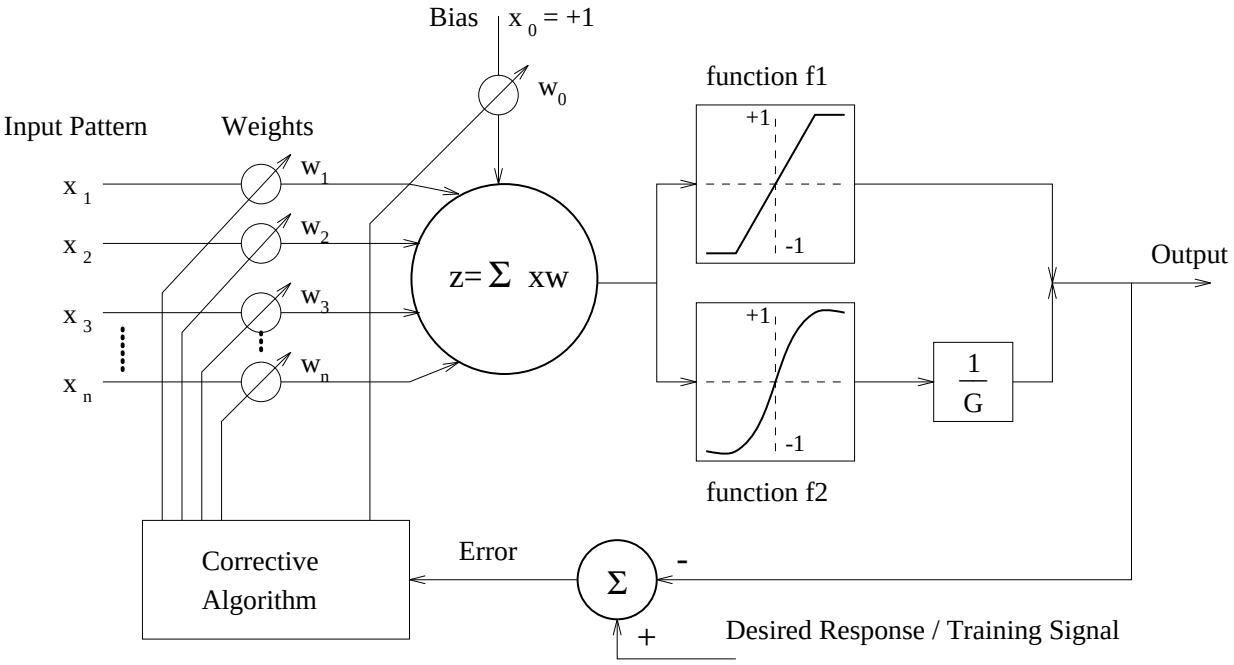

Figure 5.3: The Saturating Linear Coupled Neuron

5.7 Coupled Neurons

Many different linear and non-linear functions have been used in multilayer networks, such as the signed and threshold-based neurons (as in Widrow's ADALINE network [100, 102] and also Section 5.4), and the sigmoid (Section 2.1) to name a few. Fukumi and Omatu were the first to use two different functions, creating the *Coupled Neuron* [30] or sl-CONE (saturating linear *CO*upled *NE*uron). This is based on analog [31] and digital [28, 29] *CONE* solutions previously implemented.

The output of the neuron for summed inputs, z, is a combination of two functions $f_1(z)$ and $f_2(z)$ as shown in Figure 5.3 (from [30]). The linear function, f_1, is given by:

$$f_1(z) = \begin{cases} +1 & \text{if } z \geq \frac{1}{c_1} \\ c_1 * z & \text{if } -\frac{1}{c_1} < |z| < \frac{1}{c_1} \\ -1 & \text{if } z \leq -\frac{1}{c_1} \end{cases}$$

and the sigmoid function, f_2, is:

$$f_2(z) = \frac{2}{1 + e^{-c_2 * z}} - 1$$

combining these, and adding in a large divisor G to the sigmoidal output gives the output of the neuron, y, as:

$$y(z) = f_1(z) + f_2(z)/G \approx f_1(z)$$

The values for c_1 and c_2 scale the functions similar to a fixed gain term (see also Section 2.6).

The derivatives used in backpropagating the errors are:

$$y'(z) \quad = \quad f_1'(z) + f_2'(z)/G$$

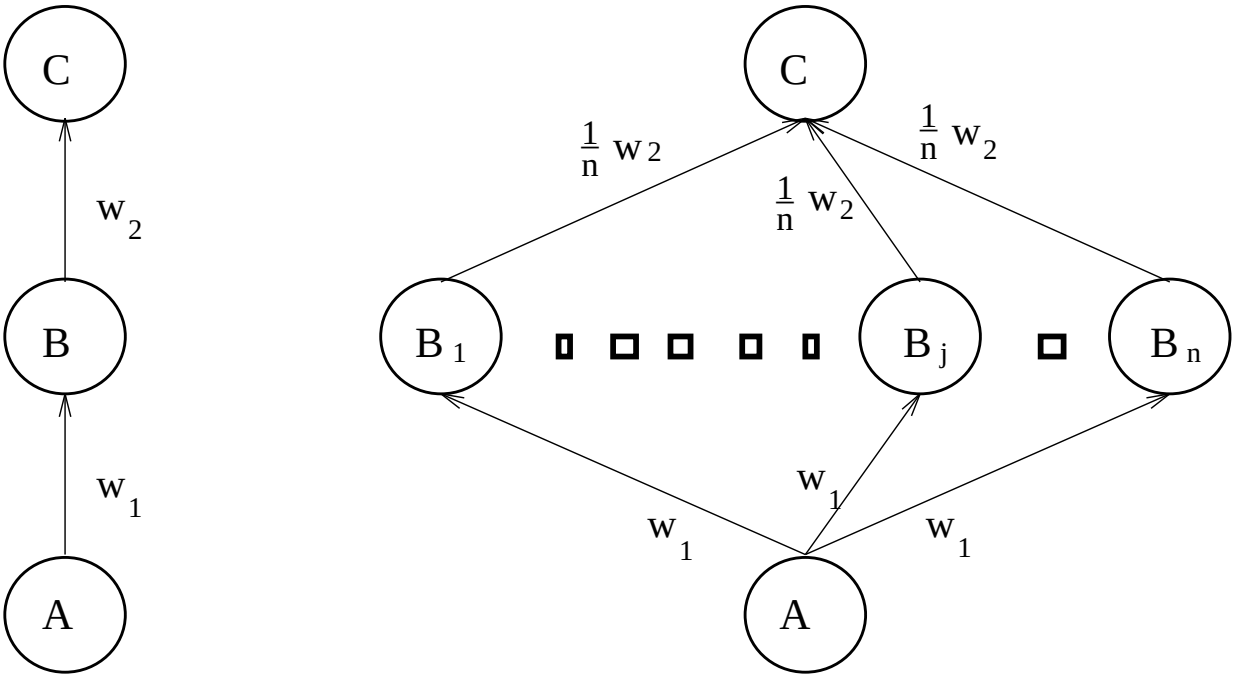

Figure 5.4: The effects of Unit-Splitting and Varying η with Fan-In

$$f_2'(z) \quad = \quad (1 - f_2(z)^2) * \frac{c_2}{2}$$

which must incorporate separate learning rates for each function to become

$$y'(z) = \eta_1 f_1'(z) + \eta_2 f_2'(z)/G$$

because $f_2'(z)/G$ is negligible and $f_1'(z)$ is 0 outside of the range $[-\frac{1}{c_1}, \frac{1}{c_1}]$. The factor of $\frac{1}{G}$ could be incorporated into η_2.

The derivative will appear to be that normally obtained with the linear function plus a scalar term added in. The algorithm that Fukumi uses to train the network is the same as BP from this point.

5.8 Unit Splitting

Unit or *Node Splitting*, as described by Plaut, Nowlan and Hinton [71] is a method by which more complexity can be added to a network generally without affecting its performance. It was developed in the hope of increasing the understanding of problems encountered when scaling a network. In conjunction with *Varying η by Fan-in* (see Section 3.3), breaking a hidden layer node up into n component nodes has the same effective input/output characteristic as a single node, where all weight changes on paths from the new split node to the next higher layer are also divided by n (the fan-in) as in Figure 5.4. The weight leading into the split node remains the same for all component nodes.

A similar result can be obtained by multiplying the number of nodes employed by the proposed unit splitting factor, and using this network instead. Although it would have the

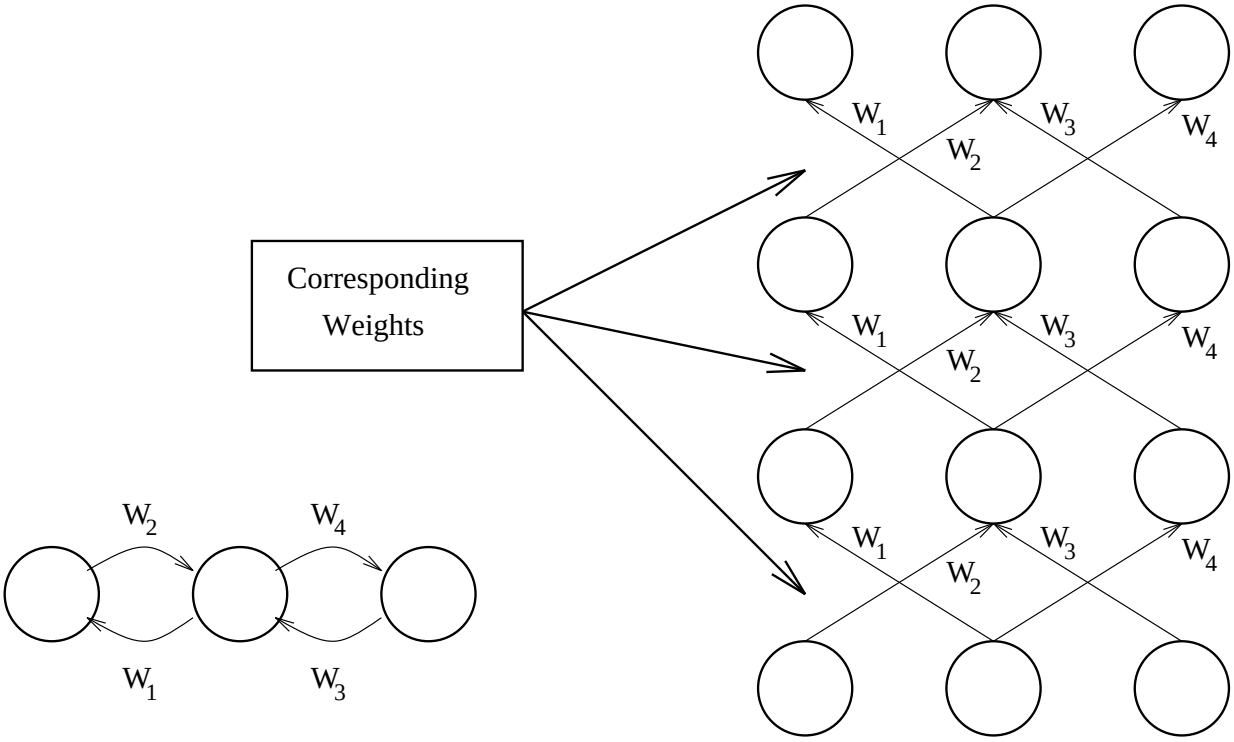

Figure 5.5: An iterative network with the layered equivalent

same complexity as a network obtained by unit splitting, it would not have the inherent structure of shared weights, even if fan-in was employed. The intention of Plaut, though, was to minimise the interaction between the weights.

The comparison shown by Plaut of these two ways of improving the complexity did not conclusively indicate that unit splitting caused faster learning. The point of interest here is that *scaling* is taking place without any loss of performance.

5.9 Iterative Networks

Backpropagation may be applied to iterative networks without losing performance. Rumelhart, Hinton and Williams [77] show that for a recurrent network, where the current state of a node relies directly on its previous state, a layered network may be built with one layer corresponding to each time slice, as in Figure 5.5 (from [71]). Plaut, Nowlan and Hinton [71] also used iterative nets with *Gain Variation* (see Section 2.6).

Computing error derivatives in the iterative net requires storing the derivatives over time at each node, which is equivalent to backpropagating through the layered net for the change in weights. This could be viewed as a 'growing' technique, as one layer is needed for each time slice, but the number of slices is generally determined in advance, and there is no 'algorithm' for adding more layers.

Iterative nets are an alternative to *Recurrent Nodes* (see also Section 5.11) when classifying sequences.

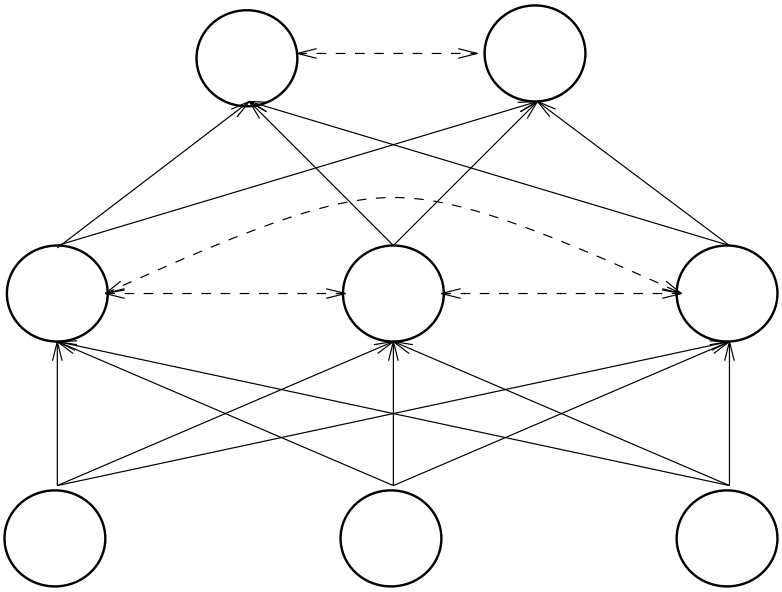

Figure 5.6: A fully-connected network

5.10 Recurrent Networks/Layers

Plaut, Nowlan and Hinton [71] use fully connected (recurrent) layers as a means of improving learning. The theory behind this is that within each layer, negative weights will be employed between the nodes to avoid the case where two nodes respond in the same way to the same data. These networks can be considered recurrent because at time t each node is affected by the outputs of the nodes on the same layer at time $(t-1)$ which were affected by the output of the node itself from time $(t-2)$. This is not to be confused with a recurrent node, which is a node which is affected by its own value directly from any number of preceding time points (See Section 5.11).

In the hidden layer(s), different neurons will tend to detect different features. This will tend to diminish the *herd effect* described by Fahlman [26].

In the output layer, the differentiation between correct output patterns will be sped up, because a small value on an output node can be forced down to zero faster as one of the other output nodes gets a higher (closer to binary 1) output. Therefore the output nodes will tend to settle into their binary patterns faster.

Figure 5.6 shows a sample fully-interconnected network with normal pathways displayed as unbroken lines, and bidirectional pathways between nodes of a layer shown in dashed lines. The bidirectional paths represent two independent connections between the two nodes which should inhibit co-firing of the nodes in the hidden layer, thus assisting feature detection.

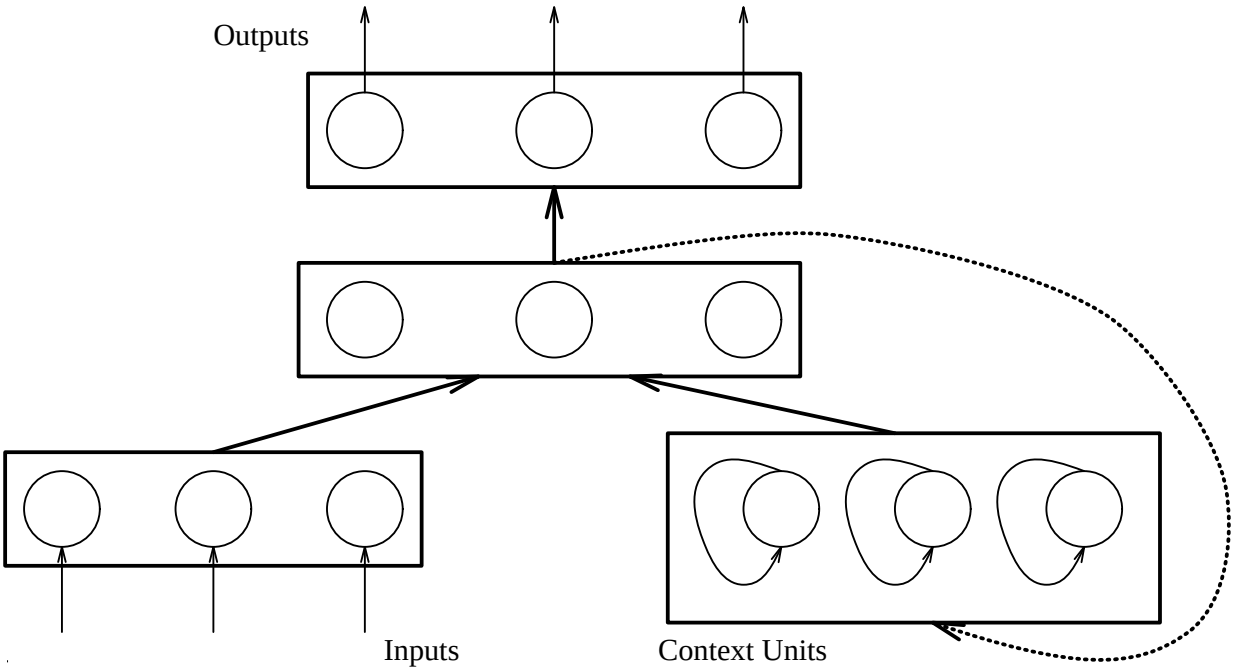

Figure 5.7: An Elman Network - fixed weights are dotted lines

5.11 Recurrent Nodes

Recurrent nodes were developed by Elman [22, 23] as an extension from the work of Jordan [47] to add a time dimension to networks, giving them the benefit of recognising a series of input patterns made up of words. Waibel [54, 94, 96] produced something similar for the same purpose.

Elman and Jordan both add a new hidden layer made up of recurrent nodes, referred to as *state* or *context* units. Each keeps a copy of their own output from the previous time unit (See also Figure 6.2, from [22]).

In Jordan's networks the number of state units is the same as the number of outputs, and they are connected on a one-to-one basis. Elman builds his networks such that there is one context unit for each hidden unit (See Figure 5.7). There are always fixed pathways of weight 1 leading into a context unit. Each context node is then connected to all nodes of the hidden layer of a fully interconnected network. Initially all context units have output of 0.5.

Recurrent nodes are also described in Dolson [17].

The *Autoregressive Algorithm* of Leighton and Conrath [56] had recurrent nodes which kept track of multiple epochs, thus giving a node true memory (to the extent built-in by the network implementor).

5.12 Complex Weights

As multi-valued networks go, the easiest implementation is merely to add the ability to use *Complex Weights* in an otherwise real-valued network. Little, Gustafson and Senn [59]

show that their modification, based on the *Complex LMS* algorithm of Widrow, McCool and Ball [101], gives a great improvement of learning speed with little additional complexity to the computation.

The derivation is parallel to the algorithm for BP, as in Appendix A, ending up with an alternative to Equation A.8, having separate real and imaginary parts

$$\begin{aligned} w_{ij}^r(t) &= w_{ij}^r(t-1) - \eta\delta_i^r y_j \\ \delta_i^r &= \begin{cases} 2z_i^r f'(z_i^r)(O_{p,i} - d_{p,i}) & \text{(output layer)} \\ 2z_i^r f'(z_i^r)\sum_h(\delta_h^r w_{ih}^r + \delta_h^i w_{ih}^i) & \text{(hidden layers)} \end{cases} \\ w_{ij}^i(t) &= w_{ij}^i(t-1) - \eta\delta_i^i y_j \\ \delta_i^i &= \begin{cases} 2z_i^i f'(z_i^i)(O_{p,i} - d_{p,i}) & \text{(output layer)} \\ 2z_i^i f'(z_i^i)\sum_h(\delta_h^i w_{ih}^i + \delta_h^i w_{ih}^i) & \text{(hidden layers)} \end{cases} \end{aligned} \tag{5.1}$$

where r and i superscripts refer to *real* and *imaginary* parts respectively.

Another point of note is the activation function, $f(z)$, used. Little claims that a linear function may be employed (such as $f(z) = z$), but instead applies the sigmoid function ($f(z) = \frac{1}{1+e^{-z}}$) thusly:

$$y_i = f(z_i^r + z_i^i)^2$$

where z^r and z^i are the real and imaginary summed inputs, and y_i is the output of the ith node, making it real-valued. After differentiating this, the factors $2z_i^r$ and $2z_i^i$ appear in Equation 5.1.

Little's purpose for developing this technique was specifically in the application of optical and real-valued engineering problems.

Benvenuto and Piazzi [11] apply a different activation function in their algorithm, but in that case the data throughout are complex-valued. See Section 5.13 for more details.

5.13 Complex Nodes

Complex backpropagation algorithms have come about through the partnerships of Kim and Guest [49] and Leung and Haykin [58] separately, and investigated by Benvenuto and Piazzi [11]. These are mostly based on the Complex variation to the *LMS Algorithm* described by Widrow, McCool and Ball [101]. The inputs and all values within a complex network have real and imaginary parts, not just the weights as described in Section 5.12. Complex Domain Backpropagation (CDBP) was also investigated by Georgiou and Koutsougeras [32].

The new weight update rule derived is:

$$\begin{aligned} w_{ij}(t+1) &= w_{ij}(t) + \eta\delta_i y_j^* \\ \delta_i &= e_i^r f'(z_i^r) + ie_i^i f'(z_i^i) \\ e &= \begin{cases} (d_i^r - O_i^r) + i(d_i^i - O_i^i) & \text{(output layer)} \\ \sum_j[(\delta_j^r w_{ij}^r + \delta_j^i w_{ij}^i) + i(-\delta_j^r w_{ij}^i + \delta_j^i w_{ij}^r)] & \text{(hidden layers)} \end{cases} \end{aligned} \tag{5.2}$$

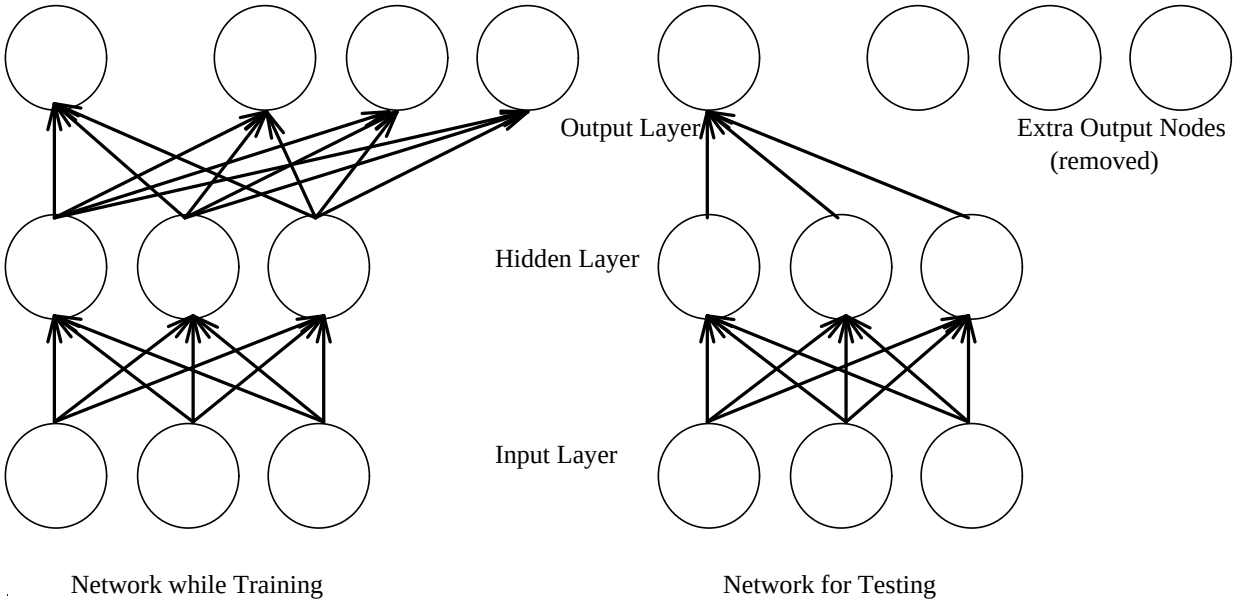

Figure 5.8: Parity check a) with extra output nodes to count high inputs; b) after removal of extra nodes

where y^* is a complex-valued output and e is a complex-valued indication of error (difference between desired and actual outputs for the node). The superscripts of r and i refer to the real and imaginary parts respectively.

5.14 Extra Output Nodes

The work of Yu and Simmons [105] using networks constructed with extra output nodes tends to suggest that a problem is usually not defined as well as it could be in terms of inputs and outputs.

The lack of definition in the hidden layers suggests that there are many different combinations of weights which will solve a given problem defined only by the inputs and outputs of that problem, so extra nodes are added in the output layer to introduce a second, related, problem to be solved by the same network with the same number of hidden units. The second problem should not hinder the main problem, and the extra outputs can be removed after the network has been trained.

Examples given include the parity problem, whose associated secondary problem might be a count of binary 1's (see Figure 5.8a). Due to the $N - N - 1$ structure of the network, there are many local minima as N gets larger, so it is useful to have another function which grows with the network.

After removal of the extra output units (Figure 5.8b), no further training is required (nothing has been removed from the hidden layers). Not only is there an improvement in generalisation, but also in learning speed and the percentage of patterns learned.

The technique was also applied to English sentence recognition.

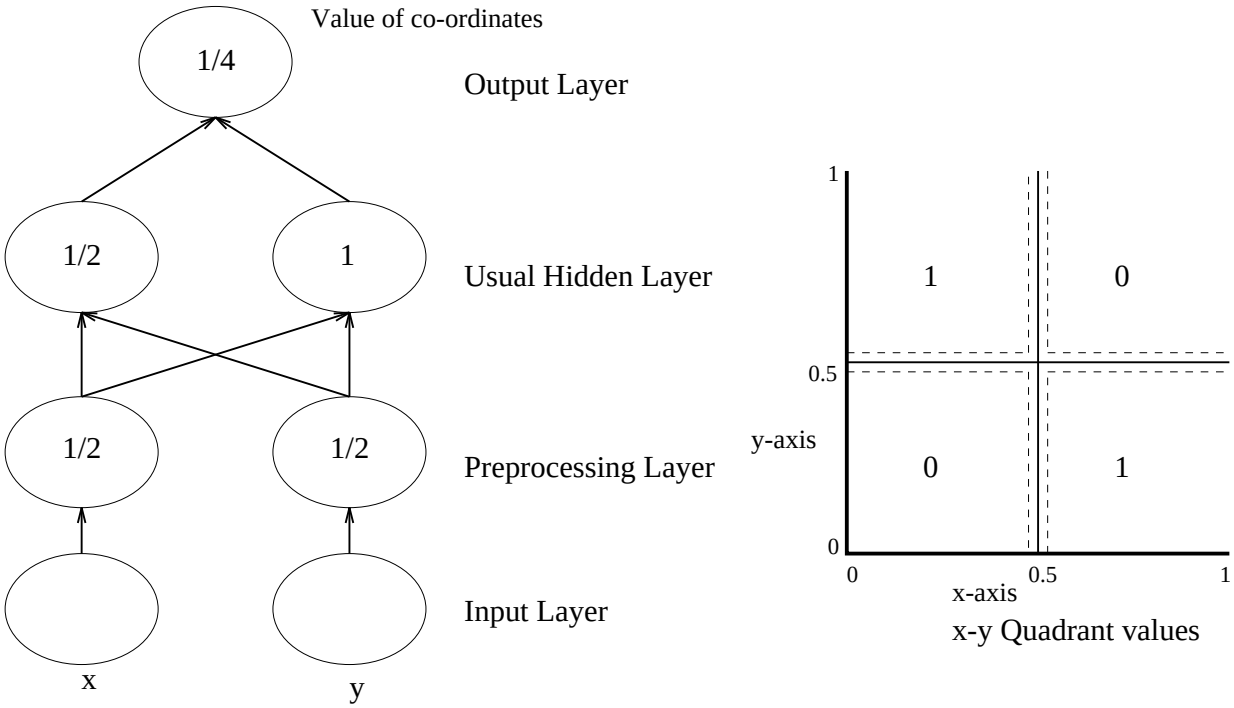

Figure 5.9: a) A network with Preprocessing Layer to solve b), dividing the unit square into quarters

5.15 Preprocessing Layer

It is often the case when teaching a network that the input data are in a format which is not conducive to learning rapidly or fully. Instead of leaving it up to the network to decide this and have it learn a better data representation dynamically using the first hidden layer (if there are more than one), it is often better to design into the system a preprocessing layer on top of the expected number of layers normally used.

This new layer can be connected to the inputs in a way in which the trainer knows or expects will be useful. This is best demonstrated through example. Figure 5.9a shows a network which is designed to solve the problem of splitting the unit square into quarters (Figure 5.9b). A simple network with one hidden layer will not solve this problem, because it will attempt to draw two curves to divide the input space. A trainer knows better, and simply by preprocessing the input layer the problem becomes much simpler. If the new layer is trained to distinguish *x-inputs* of value greater than $\frac{1}{2}$ and similarly *y-inputs*, then the binary 0/1 output from this layer can be passed on to the original hidden layer such that the problem degenerates into a simple *XOR* network.

Chapter 6

Generative Networks

Network structures may change dynamically in solving a problem by growing extra nodes or losing non-essential or unused nodes and/or layers. Although these are network techniques, to some extent (see also Chapter 5), their ability to change the network structure for themselves makes them a very different concept in learning, requiring some built-in guiding heuristic or algorithm to govern their decisions.

6.1 Cascade Correlation Architecture

The most well-known growing network using BP is Fahlman's *Cascade Correlation Architecture* [26] (see also Section 6.2). This network architecture, with associated algorithm, tries to build a solution to the given problem by adding hidden nodes one at a time to create a minimal network, minimising the residual error at each building step. There is no chance of the herd effect which is usually seen in backpropagation when all of the hidden units try to solve one portion of the network problem at the same time.

The number of inputs and outputs is defined by the problem, and a network is built similar to Figure 6.1(a) (this diagram comes from [26]). All weights at this point are adjustable (marked with a $\times$), and they are trained with BP or quickprop (see Section 2.5) until further training would not produce further learning. One extra training run through all patterns gives the current network error, E.

As the structure at this point is a single layer of perceptrons, Minsky's work [61] shows that the type of patterns that could be solved at this point would be quite limited. In general, the error, E, will be substantial.

Each hidden unit is added to the network connected to the inputs and any other hidden units (as in Figure 6.1b). It is then trained to maximise S, the correlation between the output of the node, y_p for pattern p, and the error calculated above, E.

$$S = \sum_o \left| \sum_p (y_p - \overline{y})(E_{p,o} - \overline{E_o}) \right|$$

where o runs over all output nodes, and p is each pattern. $\overline{E_o}$ and $\overline{y}$ are averaged error and candidate node output, respectively.

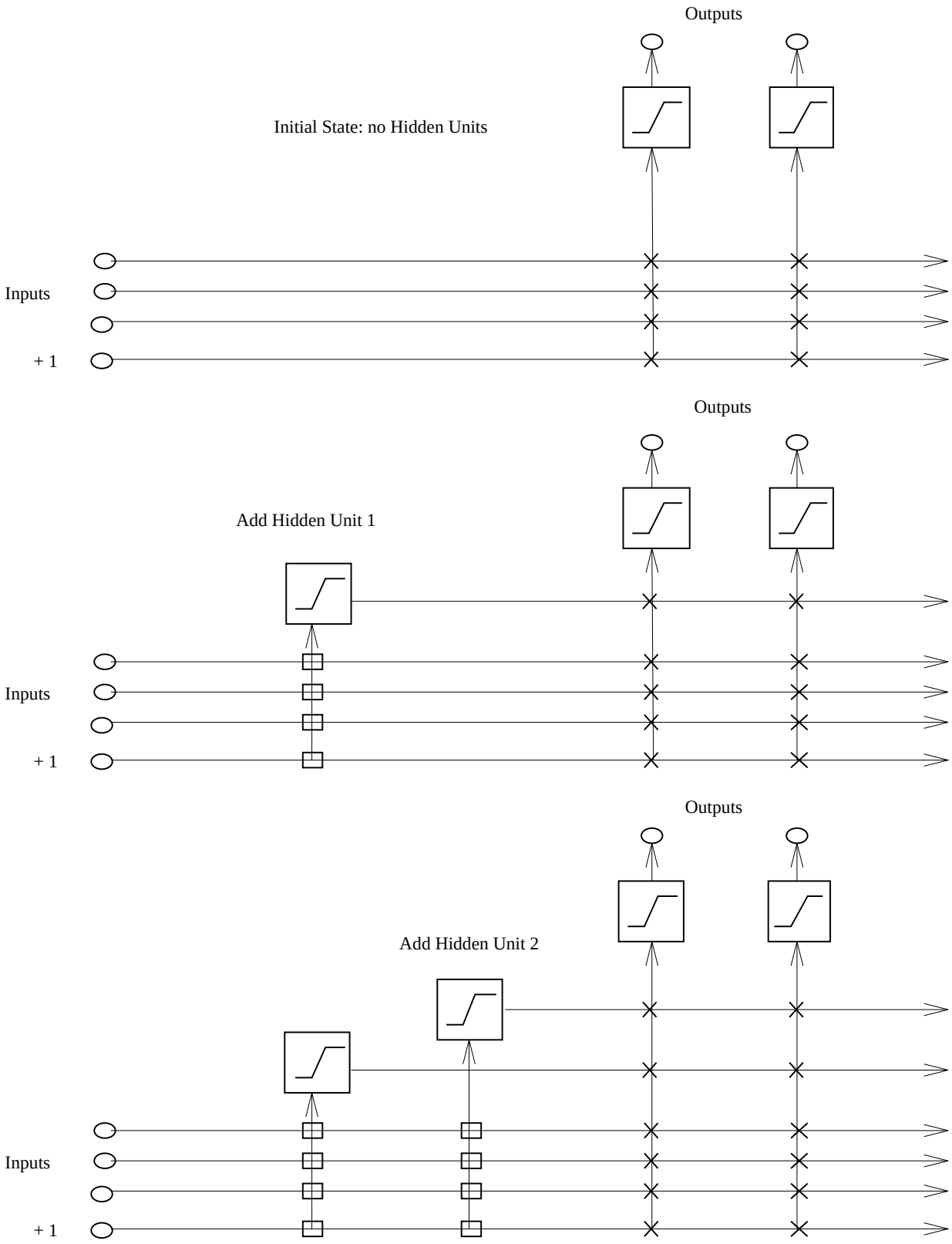

Figure 6.1: Cascade Correlation network: a) Initially, b) After adding a hidden unit, c) After adding a second hidden unit

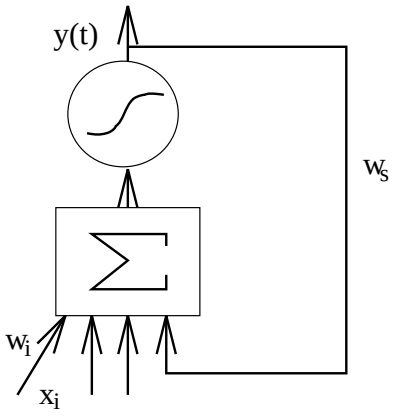

Figure 6.2: Recurrent Node

Maximising the value of S is by gradient descent similar to the algorithm used to minimise error in quickprop (see Section 2.5 and Appendix B), computing the partial derivative of S with respect to each incoming weight w_i:

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p x_{i,p}$$

where $x_{i,p}$ is the value from input/hidden node i for presentation p, f' is the derivative of the activation function, and σ_o takes on a value to indicate the sign of the correlation between the output of the candidate node and the error in output of the network.

When the node has been trained, it is placed in the network such that the connections from the inputs and other hidden nodes are fixed ($\Box$), while the connections to the output nodes are kept as variable ($\times$) as shown in Figure 6.1(c).

Nodes are added to the network until the residual error E after adding a node is within some prescribed tolerance.

Variations on this basic algorithm include the use of a *pool* of *candidate* nodes to be placed in the network. Each of these candidates will have a different initial configuration of weights, learning rates or such. Candidates may be trained simultaneously, and the one which correlates to the residual error the closest will be the one finally placed in the network.

Candidate nodes avoid the problem of training any given node into a local minimum, making them useless to the network, and can also mean that a parallel implementation could search weight-space from different initial states simultaneously. There is a great speed advantage over BP because only one node is being trained (or several in parallel) at any one time.

For the results of experiments with this algorithm, see Hoehfeld and Fahlman [40].

6.2 Recurrent Cascade Correlation

Fahlman married the speed of the *Cascade Correlation Architecture* [26] (see Section 6.1) with Elman's *Recurrent Neurons* [22] (see Section 5.11) to create *Recurrent Cascade Cor-*

relation [25]. The result of this is somewhat similar to Mozer's *Focused Back-Propagation* algorithm [64].

The output of each recurrent neuron becomes:

$$y(t) = f(\sum_i x_i(t)w_i + y(t-1)w_s)$$

where i ranges over the inputs to the node, $x_i(t)$ are those inputs and w_s is the weight of the path leading back from the output of the node (see Figure 6.2) all at time t.

The derivative of the output with respect to the weights is now:

$$\begin{aligned}\frac{\partial y(t)}{\partial w_i} &= f'(t)\left(x_i(t) + w_s\frac{\partial y(t-1)}{\partial w_i}\right)\\ \frac{\partial y(t)}{\partial w_s} &= f'(t)\left(y(t-1) + w_s\frac{\partial y(t-1)}{\partial w_s}\right)\end{aligned}$$

(assume that time $t = 0$ gives derivatives and output values of 0)

This algorithm is particularly useful for learning sequences. Fahlman claims that the resulting network is equivalent to a finite state machine and uses it to solve a finite-state (Reber) grammar, giving good generalisation with a minimal network.

6.3 Dynamic Node Creation

It is often easiest to start with a small network and build it up by adding nodes only when necessary. However, the manner in which decisions are made with regards to this growth can be very complex, like *Cascade Correlation* (Section 6.1) or can be simple, as is the case with Ash's *Dynamic Node Creation* [3] method.

The rate at which a network learns will drop off as the limit of the network's ability to learn the problem is reached. At some point the lack of learning, coupled with the knowledge that the problem has not yet been learned, should trigger the addition of a new node.

The following can be used as an indicator:

$$\frac{|E(t) - E(t-\delta)|}{E(t_0)} < \Delta_T \text{ where } (t \geq t_0 + \delta)$$

where $E(t)$ is the error of the network at time t, t_0 is the time at which the last neuron was added to the network, δ is the interval over which the slope of the error curve is calculated, and Δ_T is the trigger slope for node creation.

The whole network is retrained with the current set of weights after a new neuron is added with its inputs connected to all input nodes and its outputs connected to the output nodes.

This technique adds nodes to a single hidden layer until the desired complexity is reached, unlike *Cascade Correlation*, where each new node takes an input from previous nodes created.

A point of note is that computation time never exceeded 40% more than that described in the PDP group experiments [77], and sometimes less, due to the lesser complexity of the network during the initial stages of learning.

6.4 Network Pruning

For a fixed network structure, there is always the question of how large the network should be — how many hidden layers and how many nodes within each layer — as only the input and output are defined by the problem. Obviously it is better to overestimate than underestimate, because having too small a network will never solve the problem (unless it grows), whereas having too large a network will usually learn a solution, even if some nodes repeat the functions of others or are not used at all. Nets that are excessively large may generalise poorly.

The work of Sietsma and Dow [81, 83] shows a way of starting with a (possibly) large network and pruning off the unnecessary nodes.

The first pruning stage consists of looking at the outputs of the nodes in any given layer. If a node has consistently low or high outputs, then it is not being used to differentiate between patterns, and can be discarded. If two nodes have the same output for every pattern, then one of them can be discarded without great loss of learning. A little extra learning at this point should restore any information lost to the network by this process.

The second pruning stage requires 'intelligent' selection of any nodes which are not essential to the separation of the inputs. On an individual basis each node may be scrutinised for its effect on the network solution, and is discarded if it adds nothing intrinsic.

Removing a whole layer is possible if that layer does not differentiate the inputs any differently to its predecessor. At this point, a small amount of added training may be required because the values represented at each of the two layers may have been different (for instance the layer may have caused an inversion of the patterns).

The methods by which each stage of this pruning takes place could be compute intensive, requiring that patterns be presented to the network and outputs at each node in each layer be compared in some *non-deterministic* sense to gauge the usefulness or effectiveness of each component. This does not have the mathematical coherence of Mozer's work, described in Section 6.5.

Sietsma and Dow claim that it is not necessarily possible to teach the minimal network a solution that the excessive network finds, and therefore a network of minimal size might be expected to get stuck in local minima in more cases than a larger network. This idea is directly in line with Plaut, Nowlan and Hinton's claims [71] that added complexity in hidden layers aids learning.

This is similar to *Task Based Pruning* as described by Dunne, Campbell and Kiiveri [18] - where it is assumed that each node in the hidden layers performs some individual task, which can be separated out.

6.5 Skeletonisation

An algorithm for finding the usefulness of each node in a network was proposed by Mozer and Smolensky [65]. This algorithm is applied in a skeletonisation process to remove the nodes non-essential to the functionality of the network.

The relevance of a unit i represented by ρ_i is relative to its effects on the layer(s) it is connected to. If it has an output y_i leading into node j, then redefine node j's output to

be:

$$y_j = f(\sum_i w_{ji}\kappa_i y_i)$$

where f is a sigmoid activation function, w_{ji} is the weight from node i to node j, and κ_i is a new coefficient, the attentional strength, the amount by which node i's output is important to node j. This may take on values from 0, totally unimportant, to 1, as in BP.

The relevance, ρ_i can then be described as the difference between the error in the network when node i is either in the network or unused.

$$\rho_i = E_{\kappa_i=0} - E_{\kappa_i=1}$$

The value for ρ_i is approximated using the derivative of the error with respect to κ_i.

$$\lim_{\gamma\to 0} \frac{E_{\kappa_i=\gamma} - E_{\kappa_i=1}}{\gamma - 1} = \left.\frac{\partial E}{\partial \kappa_i}\right|_{\kappa_i=1}$$

with the assumption that the equality is valid at $\gamma = 0$.

$$\frac{E_{\kappa_i=0} - E_{\kappa_i=1}}{-1} \approx \left.\frac{\partial E}{\partial \kappa_i}\right|_{\kappa_i=1} \text{ or } -\rho_i \approx \left.\frac{\partial E}{\partial \kappa_i}\right|_{\kappa_i=1}$$

The approximation of ρ_i is therefore $\hat{\rho}_i = -\frac{\partial E}{\partial \kappa_i}$.

However, Mozer claims that a better approximation is an exponentially decaying series:

$$\hat{\rho}_i(t+1) = 0.8\hat{\rho}_i(t) + 0.2\frac{\partial E(t)}{\partial \kappa_i}$$

Another point of implementational note is that the error objective function used is not the sum of squared errors, but the sum of absolute errors, $\sum_p |D_p - O_p|$. This linear error, the authors claim, is better suited to the above approximations.

In the algorithm provided for trimming a network, backpropagation training is done until learning is sufficient, and then the relevance coefficients are calculated for each node. The node with the lowest relevance is discarded and learning continues. There is no specified number of nodes to trim, nor any other manner of defining the cessation of trimming operations.

6.6 Node Sensitivity

Mozer and Smolensky's technique is very similar to that used by Karnin [48], who employs a measure of the sensitivity of the network to the removal of a node.

The sensitivity, S, can be written:

$$\begin{aligned} S_{ij} &= E(w_{ij} = 0) - E(w_{ij} = w_{ij}^f) \\ &= -\frac{E(w_{ij}^f) - E(0)}{w_{ij}^f - 0} w_{ij}^f \\ &\approx -\frac{E(w_{ij}^f) - E(w_{ij}^i)}{w_{ij}^f - w_{ij}^i} w_{ij}^f \end{aligned} \tag{6.1}$$

where w_{ij}^f is the value of the weight after training (final value), w_{ij}^i is the initial (random) value of the weight. The approximation in the third line is required because there is generally no evaluation of $E(0)$ available during training.

Equation 6.1 must be approximated to take into account the fact that there are many weights in the system, each one changing in a different direction.

$$E(w_{ij} = w_{ij}^f) - E(w_{ij} = 0) \approx \int_{init}^{fin} \frac{\partial E(w_{00}, \ldots w_{ij}, \ldots w_{nm})}{\partial w_{ij}} dw_{ij}$$

and this can be approximated by a discrete sum over N epochs:

$$\hat{S} = -\sum_{n=0}^{N-1} \frac{\partial E}{\partial w_{ij}}(n) \Delta w_{ij}(n) \frac{w_{ij}^f}{w_{ij}^f - w_{ij}^i} \tag{6.2}$$

The summation in Equation 6.2 may be implemented using an accumulative array keeping the appropriate terms. Due to the update rule in backpropagation, this equation may be simplified to become:

$$\hat{S} = -\sum_{n=0}^{N-1} [\Delta w_{ij}(n)]^2 \frac{w_{ij}^f}{w_{ij}^f - w_{ij}^i}$$

Weights with the smallest sensitivity will be pruned after training. A small amount of retraining may be required.

Appendix A

Backpropagation Derivation

The backpropagation algorithm is described in Rumelhart, et al [76]. The notation used follows Chapter 1.2.

For a network of n input and m output neurons, the error at the output of the network for pattern p is defined as:

$$E_p(W) \stackrel{def}{=} \frac{1}{2}(O_p - D_p)^2 = \frac{1}{2}\sum_{i=1,m}(O_{p,i} - d_{p,i})^2 \tag{A.1}$$

Therefore the total error in the network for the training set is:

$$E(W) = \sum_p E_p(W) \tag{A.2}$$

Gradient descent is performed on E to minimise error, and each weight w_{ij} between nodes i and j is modified after each presentation p in a direction opposite to the error gradient.

$$w_{ij}(t) = w_{ij}(t-1) - \eta\frac{\partial E_p}{\partial w_{ij}} \tag{A.3}$$

If we sum up the total input to any neuron i across all neurons, j connected to it, we have:

$$\begin{aligned} z_i &= \sum_j w_{ij}x_j \\ y_i &= f(z_i) \end{aligned}$$

where x_j may be I_j (input datum) or y_j, the output of neuron from the preceding layer. The function f is usually the sigmoidal function. So the proportion of sensitivity each E_p has to a weight w_{ij} is

$$\begin{aligned} \frac{\partial E_p}{\partial w_{ij}} &= \frac{\partial E_p}{\partial z_i}\frac{dz_i}{dw_{ij}} \\ \frac{dz_i}{dw_{ij}} &= \frac{\partial(\sum_j w_{ij}y_j)}{\partial w_{ij}} = y_j \end{aligned}$$

with j ranging over the neurons in the layer preceding i, their outputs, y_j (which are the inputs to this neuron, x_j), do not depend on the weights w_{ij}. Sensitivity is therefore given by:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial z_i} y_j$$

substituting δ_i for $\frac{\partial E_p}{\partial z_i}$,

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_i y_j$$

which modifies Equation A.3 to become:

$$w_{ij}(t) = w_{ij}(t-1) - \eta \delta_i y_j \tag{A.4}$$

For any given neuron i in the output layer, only $O_{p,i}$ depends on z_i.

$$\delta_i = \frac{\partial[\frac{1}{2}\sum_p (O_{p,i} - d_{p,i})^2]}{\partial z_i} = (O_{p,i} - d_{p,i}) \frac{\partial O_{p,i}}{\partial z_i}$$

as $O_{p,i} = f(z_i)$:

$$\delta_i = (O_{p,i} - d_{p,i}) f'(z_i) \tag{A.5}$$

In the hidden layer(s):

$$\delta_i = \sum_h \frac{\partial E_p}{\partial z_h} \frac{\partial z_h}{\partial z_i} = \sum_h \delta_h \frac{\partial z_h}{\partial z_i}$$

where h ranges over the neurons to which i is connected. The other inputs to z_h are independent of z_i, so:

$$\delta_i = \sum_h \delta_h \frac{\partial z_h}{\partial y_i} \frac{\partial y_i}{\partial z_i}$$

With an index p over the neurons connected to h, these neurons are in the same layer as i, and therefore their outputs y_p are independent of y_i where $p \neq i$, giving:

$$\frac{\partial z_h}{\partial y_i} = \frac{\partial[\sum_p w_{hp} y_p]}{\partial y_i} = w_{hi}$$

but, since $y_i = f(z_i)$, we obtain:

$$\delta_i = f'(z_i) \sum_h \delta_h w_{hi} \tag{A.6}$$

Here is the complete rule for weight update, joining Equations A.4, A.5 and A.6, at time t:

$$w_{ij}(t) = w_{ij}(t-1) - \eta \delta_i y_j \tag{A.7}$$

$$\delta_i = \begin{cases} f'(z_i)(O_{p,i} - d_{p,i}) & \text{(output layer)} \\ f'(z_i) \sum_h \delta_h w_{hi} & \text{(hidden layers)} \end{cases}$$

Appendix B

Quickprop Algorithm

The algorithm described below comes from Veitch and Holmes [92], and is a slight modification to that described by Fahlman [24], but is presented in full.

FOR each weight w_i
  IF $\Delta w_{i-1} > 0$ **THEN**
    IF $\frac{\delta E}{\delta w_i} > 0$ **THEN**
      $\Delta w_i := \eta * \frac{\delta E}{\delta w_i}$
    IF $\frac{\delta E}{\delta w_i} > \frac{\mu}{1+\mu} * \frac{\delta E}{\delta w_{i-1}}$ **THEN**
      $\Delta w_i := \Delta w_i + \mu * \Delta w_{i-1}$
    ELSE
      $\Delta w_i := \Delta w_i + \frac{\frac{\delta E}{\delta w_i} * \Delta w_{i-1}}{\frac{\delta E}{\delta w_{i-1}} - \frac{\delta E}{\delta w_i}}$
  ELSIF $\Delta w_{i-1} < 0$ **THEN**
    IF $\frac{\delta E}{\delta w_i} < 0$ **THEN**
      $\Delta w_i := \eta * \frac{\delta E}{\delta w_i}$
    IF $\frac{\delta E}{\delta w_i} < \frac{\mu}{1+\mu} * \frac{\delta E}{\delta w_{i-1}}$ **THEN**
      $\Delta w_i := \Delta w_i + \mu * \Delta w_{i-1}$
    ELSE
      $\Delta w_i := \Delta w_i + \frac{\frac{\delta E}{\delta w_i} * \Delta w_{i-1}}{\frac{\delta E}{\delta w_{i-1}} - \frac{\delta E}{\delta w_i}}$
  ELSE
    $\Delta w_i := \eta * \frac{\delta E}{\delta w_i}$
  $w_i := w_i + \Delta w_i$
  IF $w_i >$ *weight_limit* **THEN**
    restart_learning
END

Fahlman's description of the algorithm is not detailed, and the *jump-starting* technique, Veitch paralleled in the final 'ELSE' clause is merely mentioned by Fahlman, never specified. The restarting of the system is based on an epoch limit in Fahlman's version.

Appendix C

Scaled Conjugate Gradient Algorithm

The development of this algorithm is described fully in Møller [62].

Extra notation used :

p_k	previous steepest descent direction
δ_k	approx. of $p_k^T E"(W_k)p_k$ - definiteness
μ_k	step size
α_k	scaled step size
β_k	conjugate direction
λ_k	matrix 'indefiniteness' scaling factor
r_k	steepest descent direction $-E'(W_k)$
σ_k	reduction factor for second order, $\sigma / \mid p \mid$
s_k	approx. of $E''(W_k)p_k = \frac{E'(W_k+\sigma_k p_k)-E'(W_k)}{\sigma_k}$
	$0 < \sigma_k \ll 1$

1. /* choose W_1 and scalars $\sigma > 0, \lambda_1 > 0$ and $\overline{\lambda_1} > 0$. */
   $p_1 := r_1 := -E'(W_1); \quad k := 1; \quad success :=$ **TRUE**
2. **IF** *success* **THEN**
   /* calculate second order information */
   $\sigma_k := \frac{\sigma}{|p_k|}$
   $s_k := \frac{E'(W_k+\sigma_k p_k)-E'(W_k)}{\sigma_k}$
   $\delta_k := p_k^T s_k$
3. $s_k := s_k + (\lambda_k - \overline{\lambda}_k)p_k$
   $\delta_k := \delta_k + (\lambda_k - \overline{\lambda}_k) \mid p_k \mid^2$
4. **IF** $\delta_k \leq 0$ **THEN**
   /* make the Hessian positive definite */
   $s_k := s_k + (\lambda_k - 2\frac{\delta_k}{|p_k|^2})p_k$
   $\overline{\lambda}_k := \overline{\lambda}_k + 2(\lambda_k - \frac{\delta_k}{|p_k|^2})$
   $\delta_k := -\delta_k + \lambda \mid p_k \mid^2; \lambda_k := \overline{\lambda}_k$
5. /* calculate step size */
   $\mu_k := p_k^T r_k; \quad \alpha_k := \frac{\mu_k}{\delta_k}$

6. /* calculate comparison factor */
   $\Delta_k := \frac{2\delta_k[E(W_k)-E(W_k+\alpha_k p_k)]}{\mu_k^2}$
7. **IF** $\Delta_k \geq 0$ **THEN**
   /* a successful reduction in error is possible */
   $W_{k+1} := W_k + \alpha_k p_k$
   $r_{k+1} := -E'(W_{k+1})$
   $\overline{\lambda}_k := 0$; *success* := **TRUE**
   IF $k \bmod N = 0$ **THEN**
   /* restart algorithm */ $p_{k+1} := r_{k+1}$
   ELSE
   /* **create new direction** */
   $\beta_k := \frac{|r_{k+1}|^2 - r_{k+1} r_k}{\mu_k}$
   $\beta_{k+1} := r_{k+1} + \beta_k p_k$
   IF $\Delta_k >= 0.75$ **THEN**
   $\lambda_k := \lambda_k / 2$
   ELSE
   /* **reduction in error not possible** */
   $\overline{\lambda}_k := \lambda_k$; ***success* := FALSE**
8. **IF** $\Delta_k < 0.25$ **THEN**
   $\lambda_k := 4\lambda_k$
9. **IF** steepest descent direction $r_k \neq 0$ **THEN**
   $k := k + 1$
   GOTO 2
   ELSE
   RETURN W_{k+1}

Appendix D

Descending Epsilon Algorithm

The Descending Epsilon Algorithm is described in Yu and Simmons [104].

$\epsilon := initial\_\epsilon$
REPEAT
    REPEAT
        $correct := 0$
        FOR each pattern p
            $no\_error :=$ **TRUE**
            FOR each output_unit i
                IF $\mid t_{p_i} - o_{p_i} \mid > \epsilon$ **THEN**
                $error_i := t_{p_i} - o_{p_i}$
                $no\_error :=$ **FALSE**
                ELSE
                $error_i := 0$
            END
            IF no_error **THEN**
                $correct := correct + 1$
            ELSE
                back_propagate
        END
    UNTIL $correct = number\_of\_patterns$
    $\epsilon := \epsilon - step\_\epsilon$
UNTIL $\epsilon < final\_\epsilon$

This algorithm describes an environment or method by which any backpropagation algorithm may be applied, via the *back_propagate* procedure.

The inputs to the algorithm are $initial\_\epsilon$, $final\_\epsilon$ and $step\_\epsilon$, the values for initial and final error tolerance, and the step size used to get from the former to the latter. Values used by the author were 0.5, 0.1, and 0.1 respectively.

Bibliography

[1] David H. Ackley and Michael L. Littman. Generalization and scaling in reinforcement learning. In Touretzky [90], pages 550–557.

[2] S. Amari, N. Murata, K.-R. Müller, M. Finke, and H. Yang. Asymptotic staistical theory of overtraining and cross-validation. Technical Report METR 95-06, Dept Mathematical Engineering and Information, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan, August 1995.

[3] Timur Ash. Dynamic node creation in back-propagation networks. Technical Report 8901, Institute for Cognitive Sciences, University of California, San Diego, CA 92093-0126, 1989.

[4] Paul Bakker. Don't care margins help backpropagation learn exceptions. In A. Adams and L. Sterling, editors, *Proceedings of the Fifth Australian Joint Conference on Artificial Intelligence*, Singapore, 1992. World Scientific.

[5] Paul Bakker. Exception learning by backpropagation: A new error function. In Philip Leong and Marwan Jabri, editors, *Proceedings of Fourth Australian Conference on Neural Networks*, pages 118–121, Sydney University, Sydney, 1993. Sydney University Electrical Engineering.

[6] Andrew G. Barto and Richard S. Sutton. Goal seeking components for adaptive intelligence. Technical Report AFWAL-TR-81-1070, Wright-Patterson Air Force Base, Ohio, 1981.

[7] Roy Batruni. A multilayer neural network with piecewise-linear structure and back-propagation learning. *IEEE Transactions on Neural Networks*, 3(5):395–403, 1992.

[8] Roberto Battiti. First and second-order methods for learning: between steepest descent and Newton's method. Technical report, Department of Mathematics, University of Trento, 38350 Povo (Trento), Italy, September 1991.

[9] Roberto Battiti and F. Masulli. BFGS for faster and automated supervised learning. In *INNC 90, Paris, International Neural Network Conference*, pages 757–760, 1990.

[10] Sue Becker and Yann Le Cun. Improving the convergence of back-propagation learning with second-order methods. In Touretzky et al. [91].

[11] N. Benvenuto and F. Piazza. On the complex backpropagation algorithm. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 40(4):967–969, April 1992.

[12] L. W. Chan and F. Fallside. An adaptive training algorithm for backpropagation networks. *Computer Speech and Language*, 2:205–218, 1987.

[13] Yves Chauvin. A back-propagation algorithm with optimal use of hidden units. In Touretzky [89], pages 519–526.

[14] Yves Chauvin. Dynamic behaviour of constrained back-propagation networks. In Touretzky [90], pages 642–649.

[15] J. L. Crespo and E. Mora. Tests of different regularization terms in small networks. Technical report, Applied Mathematics and Computer Science Department, University of Cantabria, Avida. Los Castros, 39005 Santander, Spain, 1993.

[16] Christian Darken, Joseph Chang, and John Moody. Learning rate schedules for faster stochastic gradient search. Technical report, Yale Departments of Computer Science and Statistics, PO Box 2158, New Haven, CT 06520, August 1992. also appears in *Neural Networks for Signal Processing 2*.

[17] M. Dolson. Machine tongues XII: Neural networks. *Computer Music Journal*, 13(3):28–40, 1989.

[18] R. A. Dunne, N. A. Campbell, and H. T. Kiiveri. Task based pruning. In Leong and Jabri [57], pages 166–169.

[19] Harry A. C. Eaton and Tracy L. Olivier. Learning coefficient dependence on training set size. *Neural Networks*, 5:283–288, 1992.

[20] David L. Elliot. A better activation function for artifical neural networks. Technical Report TR 93-8, Institute for Systems Research, University of Maryland, 1993.

[21] Jeffrey L. Elman. Finding structure in time. Technical Report CRL 8801, Centre for Research in Language, University of California, San Diego, CA 92093-0126, 1988.

[22] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, pages 179–211, 1988.

[23] Jeffrey L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225, 1991.

[24] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie Mellon University, Pittsburgh, PA 15213, September 1988.

[25] Scott E. Fahlman. The Recurrent Cascade-Correlation architecture. Technical Report CMU-CS-91-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[26] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, February 1990.

[27] Michael A. Franzini. Speech recognition with back propagation. In *Proceedings, Ninth International Conference of IEEE Engineering in Medicine and Biology Society*, 1987.

[28] Minoru Fukumi and Sigeru Omatu. A new back-propagation algorithm with coupled neuron. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, page 611, 1989.

[29] Minoru Fukumi and Sigeru Omatu. A new back-propagation algorithm with coupled neuron and its application to pattern recognition. In *Proceedings of the 28th SICE Annual Conference*, pages 1327–1330, 1989.

[30] Minoru Fukumi and Sigeru Omatu. A new back-propagation algorithm with coupled neurons. *IEEE Transactions on Neural Networks*, 3(5):535–538, 1992.

[31] Minoru Fukumi, Sigeru Omatu, and M. Teranishi. A new neuron model "CONE" with fast convergence and its application to pattern recognition. In *Transactions of the IEICE of Japan*, volume J73-D-II, pages 648–653, 1990. (in Japanese).

[32] George M. Georgiou and Cris Koutsougeras. Complex Domain backpropagation. Technical report, Tulane University, New Orleans, LA 70118, February 1992. to appear in IEEE Transactions on Circuits and Systems.

[33] Martin T. Hagan and Mohammad B. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, November 1994.

[34] Leonard G. C. Hamey. Modelling and measuring performance of feed-forward neural networks. Technical report, Department of Computing, Macquarie University, PO Box 2109, NSW, Australia, July 1991.

[35] Leonard G. C. Hamey. Benchmarking feed-forward neural networks: Models and measures. In Moody et al. [63], pages 1167–1174.

[36] Stephen José Hanson and Lorien Y. Pratt. Comparing biases for minimal network construction with back-propagation. In Touretzky [89], pages 177–185.

[37] Robert Hecht-Nielsen. Theory of the back-propagation neural networks. In *First Annual INNS Meeting*, pages 445–, 1988.

[38] Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, Reading, Mass, 1990.

[39] Geoffrey E. Hinton. Learning translation invariant recognition in a massively parallel network. In G. Goos and J. Hartmanis, editors, *Parallel Architectures and Languages Europe - Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, Berlin, 1987.

[40] Markus Hoehfeld and Scott E. Fahlman. Learning with limited numerical precision using the Cascade-Correlation algorithm. Technical Report CMU-CS-91-130, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[41] Lasse Hölstrom and Petri Koistinen. Using additive noise in back-propagation training. *IEEE Transactions on Neural Networks*, 3(1):24–38, 1992.

[42] Shih-Chi Huang and Yih-Fang Huang. Back-propagation with selective updates. In *Proceedings of the Conference on Information Sciences and Systems*, pages 584–589, 1989.

[43] Shih-Chi Huang and Yih-Fang Huang. Learning algorithms for perceptrons using back-propagation with selective updates. *IEEE Computer Systems Magazine*, pages 56–61, April 1990.

[44] Larry Jackel, et al. Practical applications of capacity control to neural net learning. Invited Presentation, ACNN95, February 1995.

[45] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1988.

[46] E. M. Johansson, F. U. Dowla, and D. M. Goodman. Backpropagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. Technical Report UCRL-JC-104850, Lawrence Livermore National Laboratory, 1990.

[47] Michael I. Jordan. Serial order: A parallel distributed processing approach. Technical Report 8604, Institute for Cognitive Sciences, University of California, San Diego, CA 92093-0126, 1988.

[48] Ehud D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990.

[49] M. S. Kim and C. G. Guest. Modification of back-propagation for complex-valued signal processing in frequency domains. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 27–31, June 1990.

[50] John F. Kolen and Jordan B. Pollack. Back-propagation is sensitive to initial conditions. Technical Report 90-JK-BPSIC, Computer and Information Systems Department, Ohio State University, Columbus, Ohio 43210, 1990.

[51] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. Technical report, Neils Bohr Institute / Nordita, Blegdamsvej 17, DK-2100 Copenhagen, Denmark, 1990.

[52] John K. Kruschke. Creating local and distributed bottlenecks in hidden layers of back-propagation networks. In Touretzky et al. [91], pages 120–126.

[53] John K. Kruschke and Javier R. Movellan. Benefits of gain: Speed learning and minimal hidden layers in back-propagation networks. *IEEE Transactions on Systems, Man and Cybernetics*, 21(1):273–280, 1991.

[54] Kevin J. Lang, Alex H. Waibel, and Geoffrey E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3, 1990.

[55] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In Touretzky [89], pages 598–605.

[56] Russell R. Leighton and Bartley C. Conrath. The Autoregressive backpropagation algorithm. Technical report, The MITRE Corp, 7525 Colshire Dr, McLean, VA 22102, 1990.

[57] Philip Leong and Marwan Jabri, editors. *Proceedings of the Third Australian Conference on Neural Networks*, Sydney University, Sydney, 1992. Sydney University Electrical Engineering.

[58] Henry Leung and Simon Haykin. The Complex Backpropagation algorithm. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 39(9):2101–2104, September 1991.

[59] Gordon R. Little, Steven C. Gustafson, and Robert A. Senn. Generalization of the backpropagation neural network learning algorithm to permit complex weights. *Applied Optics*, 29(11):1591–1592, April 1990.

[60] Ali A. Minai and Ronald D. Williams. Acceleration of back-propagation through learning rate and momentum adaptation. In *International Joint Conference on Neural Networks (winter Meeting)*, pages 676–679, 1990.

[61] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[62] Martin F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *IEEE Transactions on Systems, Man and Cybernetics*, 21:272–280, 1991.

[63] John E. Moody, Steven J. Hanson, and Richard P. Lippmann, editors. *Advances in Neural Information Processing Systems 4*, San Mateo, CA, 1992. Morgan Kaufmann.

[64] Michael C. Mozer. A focused back-propagation algorithm for temporal pattern recognition. Technical Report CRG-TR-88-3, Department of Psychology and Computer Science, University of Toronto, Canada, 1988.

[65] Michael C. Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky [89], pages 107–115.

[66] P. W. Munro. A dual back-propagation scheme for scalar reinforcement learning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Seattle, WA, 1987.

[67] K. Niida, J. Tani, T. Hirobe, and I. Koshijima. Application of neural network to rule extraction from operation data. In *American Institute of Chemical Engineers Annual Meeting*, San Francisco, CA, 1989.

[68] Steven J. Nowlan and Geoffrey E. Hinton. Soft weight-sharing. Technical report, Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A4, 1991.

[69] David B. Parker. Learning logic. Invention Report 581-64, File 1, March 1982.

[70] Barak Pearlmutter. Gradient descent: Second order momentum and saturating error. In Moody et al. [63], pages 887–894.

[71] David C. Plaut, Steven J. Nowlan, and Geoffrey E. Hinton. Experiments on learning by back propagation. Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1986.

[72] Martin Riedmiller. Rprop – description and implementation details. Technical report, Institut für Logik, Komplexität und Deduktionsysteme, University of Karlsruhe, W-76128 Karlsruhe FRG, January 1994.

[73] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster back-propagation learning: The rprop algorithm. In *Proceedings of the IEEE International Conference oN Neural Networks*, University of Karlsruhe, W-76128 Karlsruhe FRG, April 1994.

[74] Frank Rosenblatt. The perceptron:a probabilisitic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[75] David Rumelhart. Parallel distributed processing. Plenary Lecture : Second IEEE Conference on Neural Networks, 1988.

[76] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In Rumelhart and McClelland [77], chapter 8, pages 318–362.

[77] David E. Rumelhart and James L. McClelland, editors. *Parallel Distributed Processing.* MIT Press, Cambridge, MA, 1986.

[78] Warren S. Sarle. Tweakless training: Neural network training benchmarks using sas software. draft, SAS Institute Inc, SAS Campus Dr, Cary, NC 27513, January 1994.

[79] Cullen Schaffer. Selecting a classification method by cross-validation. Technical report, Dept Computer Science, CUNY/Hunter College, 695 Park Ave New York, NY 10021, March 1993. email: schaffer@roz.hunter.cuny.edu.

[80] W. Schiffmann, M. Joost, and R. Werner. Synthesis and performance analysis of multilayer neural network architectures. Technical Report 16/1992, University of Koblenz Institute für Physics, Rheinau 3-4, D-5400 Koblenz, 1992. email: schiff@infko.uni-koblenz.de.

[81] Jocelyn Sietsma. The effect of pruning a back-propagation network. Technical report, DSTO Materials Research Laboratory, Ascot Vale, 3032 VIC, 1990. expanded version of poster from ACNN90.

[82] Jocelyn Sietsma. A computational overview of artificial neural networks. Technical Report 13/91, La Trobe University, Melbourne, July 1991.

[83] Jocelyn Sietsma and R. J. F. Dow. Neural net pruning – how and why. In *Proceedings of the IEEEConference on Neural Networks*, pages 325–333, PO Box 50 Ascot Vale 3032, 1988.

[84] Fernando M. Silva and Luis B. Almeida. Speeding up backpropagation. In R. Eckmiller, editor, *Connectionism in Perspective*, pages 151–158. Elsevier Science Publishers, North Holland, 1990.

[85] Jonas Sjöberg and Lennart Ljung. Overtraining, regularization, and searching for minimum in neural networks. Technical Report LiTH-ISY-I-1297, Department of Electrical Engineering, Liköping University, Linköping, Sweden, May 1992.

[86] Eduardo D. Sontag. Feedback stabilization using two-hidden-layer nets. Technical Report SYCON-90-11, Department of Mathematics, Rutgers University, New Brunswick, NJ 08903, October 1990.

[87] W. S. Stornetta and B. A. Huberman. An improved three-layer back-propagation algorithm. In *Proceedings IEEE International Conference on Neural Networks*, pages 637–644, 1987.

[88] Tom Tollenaere. SuperSAB: Fast adaptive backpropagation with good scaling properties. *Neural Networks*, 3:561–573, 1990.

[89] David A. Touretzky, editor. *Advances in Neural Information Processing Systems 1*, San Mateo, CA, 1989. Morgan Kaufmann.

[90] David A. Touretzky, editor. *Advances in Neural Information Processing Systems 2*, San Mateo, CA, 1990. Morgan Kaufmann.

[91] David A. Touretzky, Georffrey E. Hinton, and Terence J. Sejnowski, editors. *Proceedings of the 1988 Summer School*. Morgan Kaufmann, San Mateo, CA, 1988.

[92] Andrew C. Veitch and G. Holmes. Benchmarking and fast learning in neural networks: Backprop. In *Proceedings of the Conference on Neural Networks*, pages 167–171, 1990.

[93] T. P. Vogl, J. K. Mangis, O. K. Rigler, W. T. Zink, and D. L. Alkon. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59:257–263, 1988.

[94] Alex Waibel. A connectionist glue:modular design of neural speech system. In Touretzky et al. [91], pages 120–126.

[95] Alex Waibel. Constant recognition by modular construction of large phonemic time-delay neural-networks. In Touretzky [89], pages 215–223.

[96] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 37(3):328–339, March 1989.

[97] Raymond L. Watrous and Lokendra Shastri. Learning phonetic features using connectionist networks. In *IEEE First International Conference on Neural Networks*, volume 4, pages 381–388, 1987.

[98] A. S. Weigand, B. A. Huberman, and David E. Rumelhart. Predicting the future: A connectionist approach. Technical Report PDP-90-01, Stanford University, 1990. to appear in International Journal of Neural Systems.

[99] Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[100] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE Wescon Conference*, pages 96–104, New York, 1960. IRE.

[101] Bernard Widrow, John McCool, and Michael Ball. The complex LMS algorithm. *IEEE Proceedings*, pages 719–720, April 1975.

[102] Bernard Widrow, R. G. Winter, and R. A. Baxter. Layer neural nets for pattern recognition. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 36:1109–1118, July 1988.

[103] Hua Yang. Stability and oscillation of asymmetric neural networks. In Leong and Jabri [57], pages 270–273.

[104] Yeong-Ho Yu and Robert F. Simmons. Descending Epsilon in back-propagaton: A technique for better generalization. Technical Report AI90-130, Department of Computer Science, University Austin, Texas, 78712-1188, March 1990.

[105] Yeong-Ho Yu and Robert F. Simmons. Extra output biased learning. Technical Report AI90-128, Department of Computer Science, University Austin, Texas, 78712-1188, March 1990.

Reading List

James A. Anderson and Edward Rosenfeld, editors. *Neurocomputing : Foundations of Research.* MIT Press, Cambridge, MA, 1988.

Russel Beale and Tom Jackson. *Neural Computing : an Introduction.* Adam Hilger, Bristol, England, 1991.

Francisco A. Camargo. Learning algorithms in neural networks. Technical Report CUCS-062-90, DCC Laboratory, Department of Computer Science, Columbia University, New York, NY 10027, December 1990. Draft.

Eric Davalo and Patrick Naïm. *Neural Networks.* MacMillan, London, 1991.

Joachim Diederich. Connectionist recruitment learning. In *Proceedings of 8th European Conference on Artificial Intelligence*, pages 351–356. Pitman Pub. Ltd, 1988.

Joachim Diederich, editor. *Artificial Neural Networks : Concept Learning.* IEEE Transactions on Computers Press, Los Alamitos, CA, 1990.

Russell C. Eberhart and Roy W. Dobbins, editors. *Neural Network PC Tools.* Academic Press, San Diego, CA 92101, 1990.

John Fulcher. Neural networks - a survey. Technical Report 89/7, Dept of Computer Science, University of Wollongong, PO Box 1144 Wollongong 2500, November 1989.

Stephen Grossberg, editor. *Neural Networks and Natural Intelligence.* MIT Press, Cambridge, MA, 1988.

Geoffrey E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1):143–150, 1989.

Teuvo Kohonen, Kai Mäkisara, Olli Simula, and Jari Kangas, editors. *Artificial Neural Networks Volume 2.* North-Holland, Amsterdam, The Netherlands, 1991.

Clifford Lau, editor. *Neural Networks Theoretical Foundations and Analysis.* IEEE Press, Piscataway, NJ 08855-1331, 1992.

Tsu-Chang Lee. *Structure Level Adaptation for Artificial Neural Networks.* Kluwer Academic Publishers, Boston, MA, USA, 1991.

Richard P. Lippmann. An introduction to computing with neural networks. *IEEE Transactions on Accoustics, Speech and Signal Processing*, pages 4–22, April 1987.

Berndt Müller and Joachim Reinhardt. *Neural Networks an Introduction*. Springer-Verlag, Heidelberg, Germany, 1990.

Edgar Sánchez-Sinencio and Clifford Lau, editors. *Artificial Neural Networks Paradigms, Applications and Hardware Implementations*. IEEE Press, Piscataway, NJ 08855-1331, 1992.

Drew van Camp. Amateur scientist. *Scientific American*, pages 125–127, September 1992.

V. Vemuri, editor. *Artificial Neural Networks : Theoretical Concepts*. IEEE Transactions on Computers Press, Washington, DC, 1988.

Phillip D. Wasserman. *Neural Computing : Theory and Practice*. Van Nostrand Reinhold, New York, 1989.

Bernard Widrow and Michael A. Lehr. 30 years of adaptive neural networks: Perceptron, Madaline, Backpropagation. In *Proceedings of the IEEE*, pages 1415–1442, September 1990.