Fault Tolerant Multi-Layer Perceptron Networks

George Bolt[1]

James Austin, Gary Morgan

Technical Report: YCS 180

July 1992

Advanced Computer Architecture Group

Department of Computer Science

University of York

Heslington, York, YO1 5DD, U.K.

Tel: +44-904-432771 Fax:+44-904-432767

Abstract

This report examines the fault tolerance of multi-layer perceptron networks. First, the operation of a single perceptron unit is analysed, and it is found that they are highly fault tolerant. This suggests that neural networks composed from these units could in theory be extremely reliable. The multi-layer perceptron network was then examined, but surprisingly was found to be non-fault tolerant. This result lead to further research into techniques to embed fault tolerance into such a neural network. It was found that injecting a few weight faults during training produced a MLP network which was fault tolerant. Further, it would tolerate more faults than the number injected during training. The trained network was then extensively analysed to locate the source of this fault tolerance. It was found that the magnitude of weight vectors was greatly increased in such networks, and from this it was discovered that the loss of potential fault tolerance in a MLP is due to training with back-error propagation algorithm. Finally, it is shown that the lengthy and computationally expensive training sessions in which faults are injected are not needed, since either binary thresholded units can be used, or else the trained networks' weight vectors can be scalar multiplied to produce a fault tolerant classification system.

---

[1] E-mail: george@minster.york.ac.uk

This work was supported by SERC and also by a CASE sponsorship with British Aerospace Brough, MAL.

1. Introduction

Perceptrons were devised by McCullogh and Pitts in 1943 [1] as a crude model of the brain. They are very simple computational devices which can perform binary classification on linearly separable sets of data. A binary input vector is sampled by a number of fixed predicate functions, whose weighted binary outputs are fed into a threshold logic unit. There exists a training algorithm (Perceptron Learning Rule [1]) for linearly separable problems which is guaranteed to find the required weights that are applied to each predicate output.

Due to the limited capabilities of the perceptron unit, an obvious advance was to connect layers of perceptrons together. The perceptron units were simplified by only allowing the first layer to have predicate functions sampling the input, if at all. This architecture became known as a multi-layer perceptron network (MLP). However, it was not clear how to train it since the original perceptron learning rule relied on knowing the correct response for every unit given some input. For the internal units of a multi-layer perceptron network this is not possible. This problem of spatial credit assignment was a major stumbling block to neural network research in the late 60's. The publication of Minsky and Papert's book [2] which comprehensively analysed perceptron units and single layer networks composed from them in 1969 discouraged many researchers who were trying to develop learning algorithms for neural networks composed of many layers of perceptron units. However, in 1974 Werbos [3] gave an algorithm which could train such a network, though continuous activation functions were used instead of the original binary decision threshold. It was subsequently rediscovered by other researchers including the Parallel Distributed Processing (PDP) research group [4] in 1986 who termed it Back-Error Propagation (BP).

This new learning algorithm has become almost synonymous with multi-layer perceptron networks to such an extent that a clear distinction between the architecture and learning algorithm has been lost in many cases. Back-error propagation is only one particular method for configuring the weights in a MLP so that it solves a task. The work given in this report leads to the conclusion that the back-error propagation algorithm is inherently flawed with respect to developing neural networks exhibiting fault tolerance due to the weight configurations which it finds. However, it will be seen later that it is possible to derive a set of weights which do lead to fault tolerance.

2. Construction of Training Sets

To analyse the computational fault tolerance of MLP's, realistic training sets were constructed artificially rather than using a real data source. This allowed many training sessions to be quickly performed, and more importantly, the characteristics of the dataset to be fully known.

An algorithm was devised which would produce a number of classes (c) with a number of examples (c_p) drawn from each class in a n-dimensional bipolar $\{-1,+1\}^n$ or binary $\{0,1\}^n$ space. Each class centre was chosen randomly, but with the constraint that it was a certain minimum distance from any other centre. These class centres can be viewed as pattern exemplars. The output patterns associated with the inputs were of type 1 in c, i.e. 00010 would represent inputs sampled from the second of 5 pattern exemplars.

The actual selection criteria for accepting a set of class centres was if the inequality ${}^{n}\mathrm{C}_{\frac{1}{2}d} > 2p$ held. It accepts any class set in which at least twice the number (p) of examples required from each class could be found in the space owned by a particular class exemplar. This space extends one half of the minimum interclass distance d.

The example patterns drawn from each class were based on the class exemplar with components randomly reversed with probability

$$\Pr() = \frac{\frac{1}{2}\min\left(\left\|c_i - c_j\right\| \cdot \forall c_i \neq c_j\right)}{n}$$

This method selects pattern examples with high probability from the space owned by a class exemplar, though it also allows for possible class overlap.

A seed value was specified for the pseudorandom number generator so that training sets could be repeatedly reproduced.

3. Perceptron Units

The operation of the simplified perceptron units used in multi-layer networks can be described by the following equation

$$output = \sigma\left(\sum_{k=1}^{n} i_k w_k - \theta\right) = \sigma\left(\underline{i} \cdot \underline{w} - \theta\right) \qquad (1)$$

where i_k is the k^{th} input component, and w_k is the weight on the connection from that input. The function σ applied to the result of the summation (activation) generally maps it into a limited range *[a,b]*, and hence is often called a squashing function. The constant θ offsets the activation, and is normally termed the bias.

The function of a perceptron unit is to classify its inputs into two classes, possibly with some notion of certainty added. This is a crude model of the behaviour of neurons in the brain which fire given certain stimuli, generally in bursts with frequency relating to the closeness of the input stimulus to its exemplar [5].

There are three main classes of squashing functions σ which have been developed and used in perceptron units:

Binary: The output of units is hard-limited to binary *{0,1}* or bipolar *{-1,+1}* values.

Linear: The squashing function maps x to x. Generally the output represents two classes based on the sign of the output, and the absolute magnitude the certainty of response.

Non-Linear: The activation is mapped to a limited range as with the binary units, though here the mapping is continuous. In accordance with the notion of a perceptron unit representing two classes, the function tends to be monotonically increasing. This is the class of units employed in MLP networks.

3.1. Fault Tolerance of Perceptron Units

This section examines the fault tolerance arising from the perceptron unit's style of computation. First though, a simple fault model will be constructed. From equation 1 it can be seen that the majority of entities in a unit occur in the sum of weight and input components. So long as the bias θ is small, it will be masked by the summation terms. If this is not the case, then since the bias is often considered as being a weight from a unit with fixed output -1, it could be included without special provision as an extra summation term. The squashing function can be ignored for similar scaling reasons.

Inputs i_k for classification problems are generally binary *{0,1}* or bipolar *{-1,+1}*, and so the dominating term in the computation performed by a perceptron unit, with respect to its fault tolerance, is a sum of weights w_k. It is the result of this sum which is then classified by comparison with the bias θ. For now, faults affecting weights will be considered to have the effect of forcing its value w_k to zero, which can also be viewed as removing the connection between the unit and input component i_k. The fault model will be discussed in more rigourous detail later when considering multi-layer perceptron networks.

Notice that the consequence of faults affecting weights in this way is to reduce the relative difference between the unit's activation and its bias value, i.e. the unit will move closer to the point at which an input is misclassified and failure occurs.

Since a single perceptron unit can only correctly classify linearly separable patterns, the two classes can be viewed as non-intersecting regions in n-dimensional space. The optimal separating hyperplane for maximising fault tolerance is the vector perpendicular to the bisection of the line connecting their centroids[2] (see figure 1). This is since its associated weight vector maximises the distance of every input pattern from the separating hyperplane and hence also the possibility of misclassification.

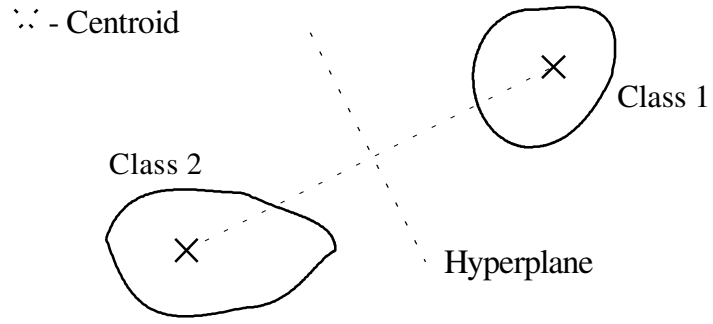


Figure 1 Separating hyperplane for maximal fault tolerance

More formally, if class C_k has n members $\underline{c}_i$, each with associated weighting p_i which indicates the chance of $\underline{c}_i$ occurring as an input, then its centroid $\underline{c}_k^*$ is defined as

$$\underline{c}_k^* = \frac{1}{n} \sum_{i=1}^{n} p_i \cdot \underline{c}_i \qquad \text{where } \Sigma p_i = 1$$

The separating hyperplane which optimises fault tolerance is specified by the weight vector $\underline{w}$ and bias value θ as follows

$$\underline{w} = \underline{c}_2^* - \underline{c}_1^* \qquad \text{and} \quad \theta = \frac{\left(\underline{c}_2^* - \underline{c}_1^*\right) \cdot \left(\underline{c}_1^* + \frac{1}{2}\underline{w}\right)}{|\underline{w}|} \qquad (2)$$

---

[2] Defined as average member of class where every member is weighted by its likelihood of occurring.

This can be shown by considering that the following function must be maximised to optimise fault tolerance

$$\mathrm{F} = \sum_{i=1}^{n} p_i \cdot d\left(\underline{c}_i, H\left(\underline{w}, \theta\right)\right) \qquad (3)$$

where $H\left(\underline{w}, \theta\right)$ defines the separating hyperplane, and function d gives the distance of input $\underline{c}_i$ from this hyperplane measured positive in the direction towards the class t_i which $\underline{c}_i$ belongs to. For bipolar representations this function is

$$\mathrm{F} = \sum_{i=1}^{n} (\underline{w} \cdot \underline{c}_i - \theta).t_i \qquad (4a)$$

whilst for binary representations

$$\mathrm{F} = \sum_{i=1}^{n} \left(\underline{w} \cdot \underline{c}_i - \theta\right).(2t_i - 1) \qquad (4b)$$

Taking the case for bipolar representations, the method for binary being similar, maximising F requires that

$$\frac{d\mathrm{F}}{dH} = 0$$

Note that function F has no minimum since the separating hyperplane could placed infinitely far away from either of the two classes.

The differentiation of F can be simplified by incorporating the bias as an extra weight on a connection from a unit which always outputs -1. This has the effect of moving the separating hyperplane to pass through the origin. Notating the new weight vector as $\underline{w}^*$

$$\begin{aligned} \frac{d\mathrm{F}}{d\underline{w}^*} &= \sum_{i=1}^{n} p_i \cdot \underline{c}_i \cdot t_i \\ &= \sum_{t_i=1} p_i \cdot \underline{c}_i \; - \sum_{t_i=-1} p_i \cdot \underline{c}_i = 0 \end{aligned}$$

This shows that maximum fault tolerance is achieved when the class centroids are equidistant from each other, in this case about the origin. Hence the separating hyperplane must be such that it perpendicularly bisects the line between the class centroids as required. Note that this result also emphasises the need to incorporate a bias into a perceptron unit.

Given that the components of the centroids of the two input classes $\underline{c}_1^*$ and $\underline{c}_2^*$ are defined to be c_i^1 and c_i^2 respectively, then a suitable measure of the distance between them is supplied by

$$\mathrm{D}\left(\underline{c}_1^*, \underline{c}_2^*\right) = \sum_{i=1}^{n} \mathrm{abs}\left(c_i^1 - c_i^2\right)$$

This measure copes with both the binary and bipolar representations being considered here. On average, the distance of a particular input from any of the two classes will be $\frac{1}{2}\mathrm{D}\left(\underline{c}_1^*, \underline{c}_2^*\right)$

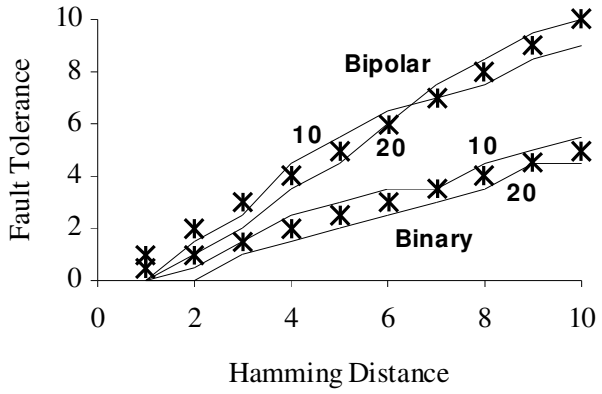
due to the position of the separating hyperplane. Since the fault tolerance of a perceptron unit can be considered as the sum of weighted input components, this implies that $\frac{1}{2}\mathrm{D}\left(\underline{c}_1^*, \underline{c}_2^*\right)$ weight faults could be tolerated before failure (i.e. misclassification) would occur.

This also indicates that a bipolar representation will lead to improved fault tolerance in perceptron units since the distance between class centroids in bipolar space supplied by function D will be twice that for a binary representation. It can be viewed that 0 valued input components in a binary representation do not actively provide information in computing the output of a unit unlike their counterparts in a bipolar representation.

3.2. Empirical Analysis

To test this theory, a simulation was run training a single perceptron unit to distinguish between two pattern classes. The two class centres were randomly chosen, and the Hamming distance between their centres varied between 1 and 10. The training set was then constructed by selecting 5 examples of each class (see section 1), and then the back-error propagation algorithm was used to find a weight vector solving the problem. This particular learning algorithm was used instead of the simpler (but sufficient) perceptron learning rule for consistency with later experiments.

For every training set, the perceptron unit was trained until the mean error was less than 0.1. Both 10 input and 20 input perceptron units were used. Then weights were randomly chosen and removed (i.e. setting *w* to zero) and the unit tested for failure. The definition of failure used was inability to distinguish the two classes. Each experiment was carried out many times until the standard deviation of the fault tolerance exhibited fell below 1.0.



Graph 1 Binary .vs. Bipolar Representation in Perceptron Unit

Graph 1 above shows the results of these experiments. The value for fault tolerance given on the y-axis is the mean maximum number of weights/connections that can be removed without failure occurring. It can be seen that the data collected closely matches the theoretical predictions (marked with stars). Also, it clearly shows that bipolar representations lead to improved fault tolerance as stated above.

3.3. Alternative Visualisation of a Perceptron's Function

The predominant technique for understanding the operation of a perceptron unit is by viewing that it classifies patterns based on a dichotomy of its input space formed by a hyperplane which is normal to the weight vector $\underline{w}$ and distance θ from the origin. An alternative understanding of a perceptron unit's computation is proposed in this section. Although both visualisations precisely describe the operation of a perceptron unit, hyperplane separation does not naturally extend to allow intuitive insight into visualising the effect of faults, as was seen in the previous section.

The alternative concept proposed here for visualising a perceptron unit's computation starts from considering the scalar value of the vector projection of the input $\underline{x}$ onto the weight vector $\underline{w}$ compared to the bias θ of the unit. The weight vector defines the feature which the perceptron unit represents in a subset of its input space. A subset is specified since it has been found that not all the weights on connections feeding a unit are used, some decay to near zero during training and play no significant part in the units operation[3]. Note that by the term *feature* used above, it is not meant that a unit's weight vector corresponds to some semantic object in the problem domain.

The bias then represents by what degree the feature which the weight vector represents has to be present in the input $\underline{x}$. If there is enough evidence, i.e. $\underline{w} \cdot \underline{x} > \theta$, then it will cause the unit to "fire". The squashing function then saturates the unit's activation as appropriate.

This alternative visualisation of the operation of a perceptron unit has various advantages over that of hyperplane separation. The effect of removing weights on a hyperplane is difficult to visualise, whereas for feature recognition it is clear that information is lost and the projection of the input onto the weight vector will be less precise.

Also, the notion of distribution of information storage in neural networks becomes more obvious since it can be viewed that the feature which a unit represents consists of many components, not all of which have to be present for a pattern match to be performed. These

[3] This is the basis for the various pruning algorithms which have been developed.

components could either be inputs fed to the network, or also the outputs of previous units so combining multiple features to form more complex ones. As stated above, it is not intended that these features should be viewed as corresponding to any semantic item.

4. Multi-Layer Perceptrons

For ease of description later in this report, the MLP neural network will now be defined and its associated training algorithm, back-error propagation, also given. The architecture of a MLP is shown in figure 2 which shows how units are arranged in layers, with full connectivity between the units in neighbouring layers. This is the standard pattern of connectivity commonly used, though others such as having connections between units and layers past its immediate neighbour are possible.

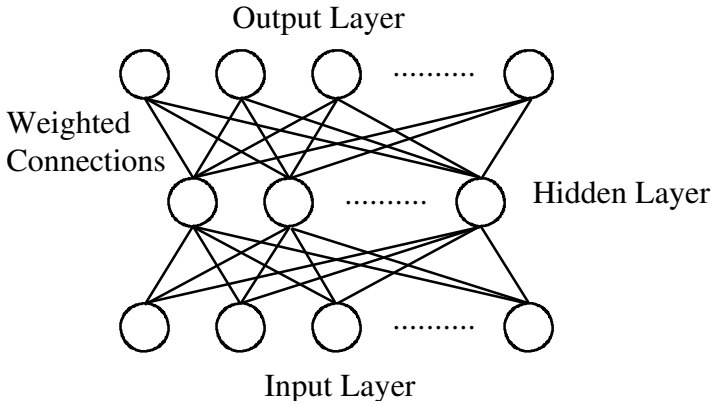


Figure 2 Multi-Layer Perceptron Neural Network

Each unit computes the following function based on its inputs from feeding units:

$$o_i = f_i\left(\sum_k o_j w_{ij}\right) \tag{5}$$

Note that an ordering of a MLP units is specified since feeding units j must have already been evaluated. Also, the bias θ has been incorporated as a special weight link as described previously. The activation or squashing function f_i can be any differentiable monotonically increasing function. The input units merely take on the value of their corresponding component in the input pattern.

4.1. Back-Error Propagation

The back-error propagation learning algorithm [4] supplies a weight change for every connection in the MLP network given an input vector $\underline{i}$ and its associated target output vector $\underline{t}$. The weight change for each weight is

$$\Delta w_{ij} = \eta \delta_i o_j \tag{6}$$

where for output units

$$\delta_i = (t_i - o_i) f_i'\left(\sum_k w_{ik} o_k\right) \tag{7}$$

and for hidden units

$$\delta_i = f_i'\left(\sum_k w_{ik} o_k\right) \sum_l \delta_l w_{il} \qquad (8)$$

This last equation shows how the error for unit *i,* δ_i, is propagated back to units in previous layers to solve the credit assignment problem.

4.2. Fault Model for MLP's

Before a study of the fault tolerance of multi-layer perceptron networks can be made, a fault model must be constructed for them. A fault model lists which components in a system can become defective, and also the nature of their defect. Note that such components need not physically exist in an implementation; they can be an abstract object. In general a fault model should attempt to satisfy the following conditions:

- The abstract faults described in the model should adequately cover the effects of the physical faults which would occur in an implementation.
- The computation requirements for simulation should be satisfiable.
- The fault model should be conceptually simple and easy to use.
- It should provide an insight into introducing fault tolerance in a design.

Note that these requirements often conflict which each other and a compromise must be found. For instance, simplicity, which leads to lower computational requirements, may result in an inaccurate model if carried to excess.

The development of fault models from an *abstract* description of a neural network is extensively dealt with in [6]. To summarise, the methodology for producing such a fault model is as follows:

1. The atomic entities within the system viewed at the conceptual level at which its fault tolerance is being examined must be extracted.
2. Discard from these entities any which would not have a significant effect on the reliability of the system. This may be due to the number of such entities in the overall system being very small as compared to other entities selected in step 1.
3. For each entity, the manifestation of the faults affecting it can be defined by applying the principle of causing maximum damage to the system's computation, restricted by considering certain implementation details.

For a multi-layer perceptron network viewed from the abstract level as defined above, the various atomic entities during operational use are the weights, a unit's activation, and the squashing function. Due to the massive number of weights compared to entities associated with units, only the weights need be considered in a multi-layer perceptron.

The manifestation of weight faults in a multi-layer perceptron must now be defined. To cause maximum harm, a weight should be multiplied by $-\infty$ (see [6] for more details). However, in any realistic implementation, a potentially infinite valued weight would be highly unlikely. Instead it is probable that weights will be constrained to fall in a range $[-W,+W]$, and so a weight fault should cause its value to become the opposite extreme value. The loss of a connection can be modelled by a weight value becoming 0. For simplification, only the latter fault mode was considered in this report.

Note that an actual unit becoming faulty is *not* considered eligible for the fault model since the concept of a unit entity exists at a much higher visualisation level than that taken here. Such an abstract definition of a neural network would not be particularly useful since it hides far too much of the underlying computation of the system, and so would not provide beneficial information on the fault tolerance of multi-layer perceptron networks. This is especially true if results obtained on fault tolerance were used in the development of a physical implementation.

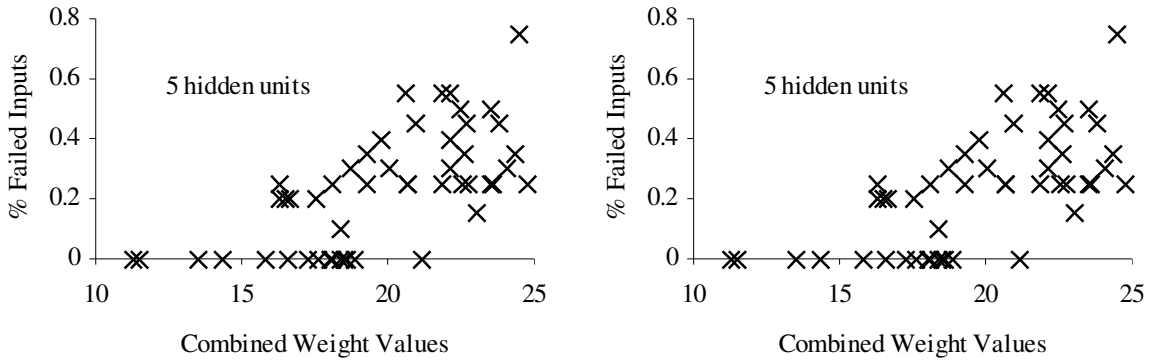
4.3. Fault Tolerance of MLP's

Many studies of the fault tolerance of multi-layer perceptron networks have been carried out [8,7,9], a survey can be found in [10]. However, nothing approaching a comprehensive analysis of the nature of the fault tolerance in MLP's is known to exist. In the rest of this report this task will be approached, and in part met. Clearly the results looking at a single perceptron unit as described already will be of great use in undertaking this.

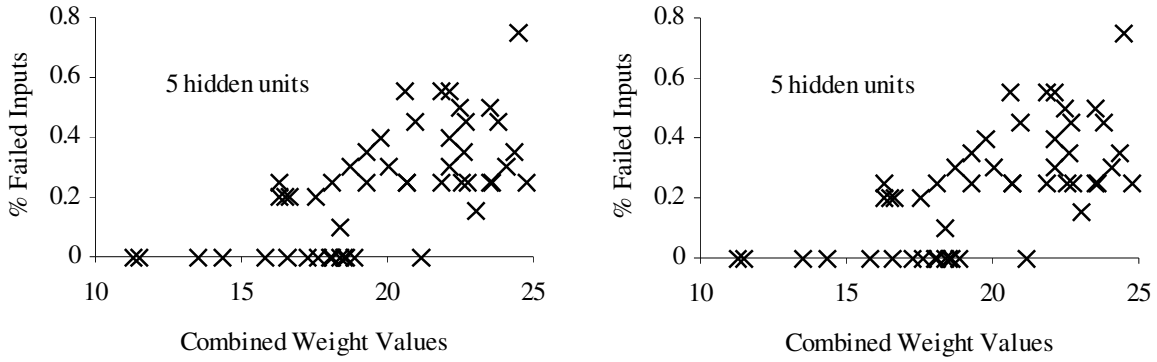
Given that a single perceptron unit seems to be very reliable, a simulation was run to gauge the effect of faults in a multi-layer network. A complex training set was used following the method described in section 2. Four class exemplars were randomly chosen in a 10-dimensional bipolar space, with 5 pattern examples selected from each making a training set of 20 vector associations. A MLP network was then trained to solve this classification problem using the back-error propagation algorithm until the maximum output unit error diminished to 0.05. Two training sessions were run, the first on a MLP network having 5 hidden units, and the second for 10 hidden units.

The trained MLP network was then subjected to fault tolerance analysis. This consisted of randomly selecting approximately 10% of the weights in each network, and forcing their values to 0 (see section 4.2). The proportion of patterns in the training set that were then misclassified (i.e. the maximum output unit error was over 1.0) was used as a measure of the damage inflicted on the MLP network.

The surprising result was that so few weight faults (8 in the case of the 10-5-4 network containing 79 weights) would cause a considerable proportion of the input set to fail, whilst the recognition of the remaining input patterns would not be appreciably degraded. It was also found that even certain individual weights would cause failure to occur. Graph 2 below shows how the percentage of input patterns incorrectly classified in the training set varies with the total absolute magnitude of the faulted weights. It clearly illustrates that weights which contribute most towards the feature represented by a unit (i.e. sum of faulted weights is large) cause an appreciable percentage of the training set to be incorrectly classified when faulted. Graph 3 below shows the maximum unit error over all training patterns. It further



Graph 2 Proportion of failed patterns due to 10% weight faults



Graph 3 Maximum output unit error due to 10% weight faults

reinforces the result that significant weights exist in the classification of particular input patterns.

This result contradicts many remarks made by previous work that multi-layer perceptron networks are fault tolerant. It also brings into question the view that they store information in a distributed manner since the destruction of only a few weights causes a non-trivial failure among many stored associations.

4.4. Distribution of Information in MLP's

The traditional view of information distribution in neural networks, and multi-layer perceptrons in particular, is by analogy to holographic storage; no single storage element (normally taken to be a weight, or occasionally a unit) in a neural network stores a particular pattern. Instead, patterns are stored distributed across all of the weights in a neural network. The argument for fault tolerance which has been relied on is that, as for a hologram, each weight in a neural network is unimportant globally, and so its loss will not seriously impair the operation of the network. However, it is doubtful whether this argument is valid for MLP's given the above results which showed that for a small number of weight faults, a large proportion of the training set is misclassified. For a single perceptron unit though, it has been shown that a certain number of weights can be viewed as being redundant in this fashion.

For MLP networks, it is considered more appropriate to view each layer transforming patterns into a different space, such that in the last hidden layer a representation is developed which is linearly separable to produce the required output. This process can be viewed as distributing the complex task of classification into several simpler steps at each hidden layer. Each layer of perceptron units though, can be viewed as being distributed in the sense given in the previous paragraph. Fault tolerance will arise jointly from the reliability of each layer of perceptron units and the redundancy in each hidden layer representation.

4.5. Training for Fault Tolerance

Given that a MLP trained using back-error propagation is not as fault tolerant as might have been concluded from the results obtained above by examining the reliability of a single perceptron unit, various studies were undertaken into producing a technique which produced a fault tolerant neural network based on the MLP. These included:

- Limited interconnectivity

- Local feedback at hidden and/or output layers
- Training with weight faults injected

However, only the technique of injecting weight faults during training produced clear results with respect to developing a MLP network which exhibits fault tolerance.

4.5.1. Training with Weight Faults

This method is similar to that used by Clay and Sequin which produces a fault tolerant MLP network by injecting unit faults during training [11]. However, in section 4.2 it was shown that the basic functional entities in a MLP network which should be considered when examining its fault tolerance are the weights on connections between units rather than the actual units. Due to this, weights were randomly set to 0 during training so that specific fault tolerance to weight faults would be introduced. A training session would consist of the following steps:

1. Randomly choose a fixed number of weights and fault them.
2. Apply back-error propagation algorithm for all patterns in training set.
3. Restore fault weights and repeat from step 1 until the maximum output unit error diminishes to an acceptable value.

Generally only a single weight was faulted during each training step, though simulations were also carried out faulting multiple weights. However, the increase in possible faulted weight combinations increases combinatorily, and so training rapidly becomes prohibitively expensive.

4.5.2. Comparison with Clay and Sequin's Technique

Superficially, there seems little difference to injecting weight faults during training injecting weight faults as against units being faulted. However, the argument for training with faults is to imbue a neural network with resistance to those particular faults, and since the construction of a fault model for a MLP (section 4.2) showed that only weight faults are important in a MLP system, then it seems more reasonable to train injecting weight faults. Unit faults are too abstract and unlikely to be representative of the effect of physical faults in an implemented MLP. Due to this, it is expected that training with weight faults will lead to better overall fault tolerance.

Also, the technique of injecting weight faults during back-error propagation training to promote fault tolerance in a MLP network is *not* the major work described in this chapter. Instead, this chapter concentrates on analysing the MLP networks produced by fault injection training and shows that the back-error propagation algorithm inherently produces non-fault tolerant classification systems. The results of this analysis, combined with the previous analysis of the fault tolerance of a single perceptron unit, is used to show how a fault tolerant MLP network can be constructed after normal back-error propagation training. This is used to show how a fault tolerant MLP network can be constructed after normal back-error propagation training. This is a great advantage since the extremely long training times required when training with faults injected in each learning cycle will not be needed to produce a similarly fault tolerant MLP.

4.6. Analysis of Trained MLP

The MLP network trained using the method described above has been demonstrated to form fault tolerant systems, and several reasons have been proposed to explain why this should be so. Similar reasons can be applied when training with unit faults.

The first line of reasoning views the faulted MLP network during training as a sub-network due to the loss of a unit/weight. These sub-networks are then individually trained to solve the problem, and their individual solutions converge such that global agreement between them is reached. Once fully trained, the loss of a single weight can easily be tolerated, and tolerance to more than one weight is due to distribution over the sub-networks.

An alternative view is that a distributed representation is formed in the MLP [11], i.e. the hidden layer representation is different to that normally found by plain back-error propagation. This representation is redundant in some way and so produces fault tolerance.

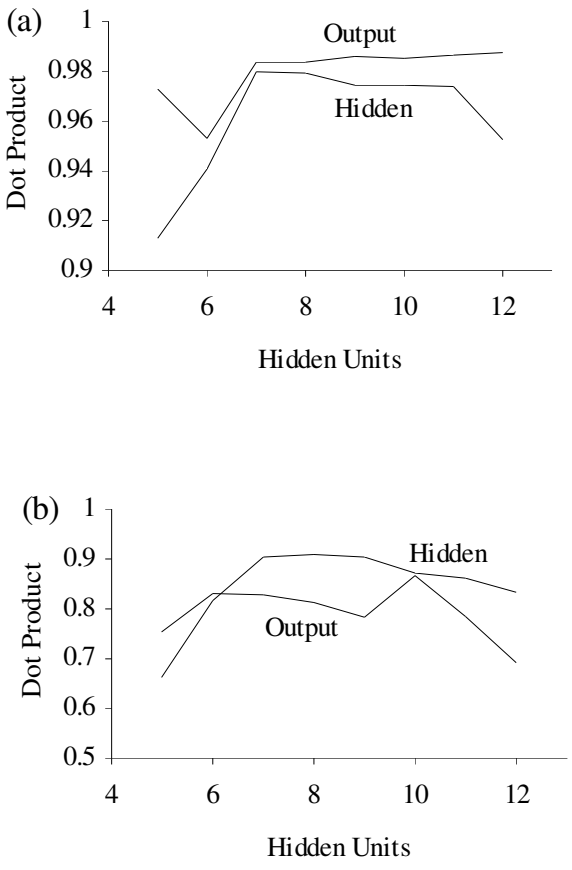
However, it will be shown in this section that neither of these two lines of reasoning are correct, and from the results, it is shown how to produce a fault tolerant MLP, in the style of the networks produced by training with faults, though with little extra computational expense over basic back-error propagation training.

4.6.1. Analysis of Fault Injection Training

To identify the difference between a MLP trained with plain back-error propagation and one with fault injection, various sized MLP's were trained using both methods and the resulting networks compared. A complex training set was constructed using the algorithm given in

section 2 consisting of 4 class exemplars with 5 input patterns drawn from each producing a training set of 20 associated pairs. The dimension of the input space was 10.

The first area examined was the internal representation developed for each of the four class exemplars. It was found that all hidden units had a value of near -1 or +1 (a bipolar representation was used) for every input pattern. Further, comparing the hidden representations of matching MLP network configurations trained using the two methods, it was found that they were identical in every case. The comparison took into account the possibility of a fixed permutation of the hidden units. This result implies that the second of the two reasons given above explaining the fault tolerance induced by training with faults is incorrect.

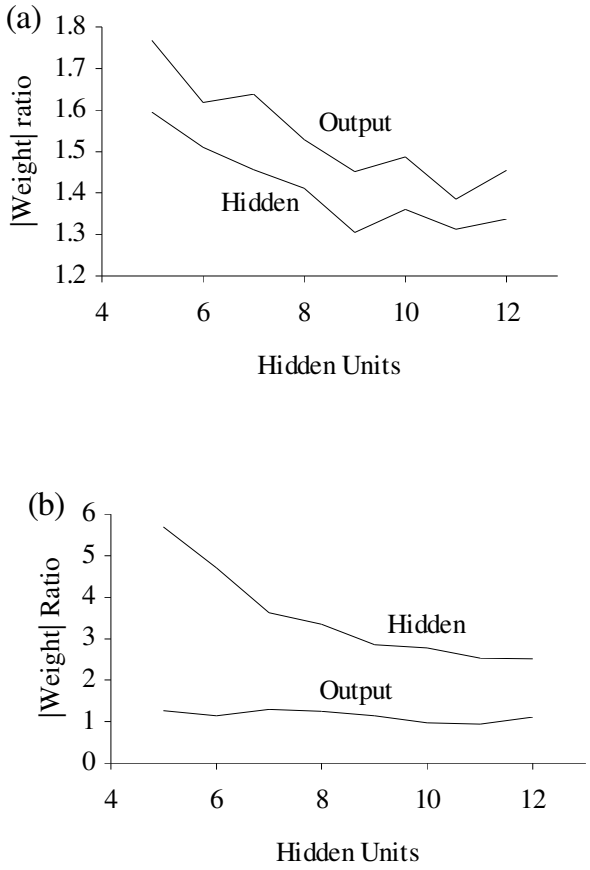


Graph 4 Comparison of weight vector directions in MLP's trained with weight faults, a) single fault injection, and b) double fault injection

The next comparison performed was between the vector direction of the weights feeding every unit in each network. As above, the possibility of a fixed permutation in the hidden layer units of both networks was allowed for. Graphs 4 above show the average dot product

between the weight vectors of matching hidden and output units in MLP networks trained with and without injected faults. The number of hidden units in each network varied between 5 and 12. Once again, it can be seen that no significant difference exists between the various pairs of matching networks. This meant that not only are the hidden representations the identical, but the dichotomies formed by all units in their input space are also almost exactly the same.

Finally, the length of weight vectors for matching units was compared between the two sets of trained MLP networks, where the length of a weight vector was found using the Euclidean measure. Graph 5 show the average ratio of the length of weight vectors from a MLP trained with faults injected to that of the corresponding weight vector when plain back-error propagation is used. It can be seen that in the MLP networks trained with faults injected, the length of weight vectors is greater than in the original network. When two faults are injected on each training step, this ratio is even more accentuated for hidden units.



Graph 5 Comparison of weight vector lengths in MLP's trained with weight faults, a) single fault injection, and b) double fault injection

4.6.2. Back-Error Propagation

The results described above can be explained if the operation of a perceptron unit is considered using the alternative visualisation described in section 3.3. The projection of an input $\underline{x}$ onto its weight vector $\underline{w}'$ which suffers a fault in component f can be described as follows

$$s = \underline{w}' \cdot \underline{x} = \sum_{i=1}^{n} w_i \cdot x_i \; - \; w_f \cdot x_f$$

This scalar value s is now compared against the unit's bias θ to see if the degree by which input $\underline{x}$ matches the feature $\underline{w}$ is sufficient to activate the unit. Looking at the absolute difference between s and θ

$$\begin{aligned} |\underline{w}' \cdot \underline{x} - \theta| &= \left| \sum_{i=1}^{n} w_i x_i \; - \; w_f x_f \; - \; \theta \right| \\ &= \left| \sum_{i=1}^{n} w_i x_i - \theta \right| - w_f x_f \\ &= |\underline{w} \cdot \underline{x} - \theta| - w_f x_f \end{aligned} \qquad (9)$$

It can be seen that the absolute difference between the fault-free projection and θ is decreased. If this value becomes negative, failure will result since the unit will then misclassify its input.

Although this describes the effect of a weight fault, it does not explain why only a few faults generally cause such a dramatic failure in a multi-layer perceptron network for a subset of the training set. It will now be shown how the back-error propagation algorithm used to train the MLP network causes this lack of fault tolerance. The common multiplicative term in the weight update $\Delta w_{ij} = \eta \delta_i o_j$ (equation 6) is

$$f_i'\left(\sum_k w_{ik} o_k \right) \cdot o_j = f_i'\left(\sum_k w_{ik} o_k \right) \cdot f_j\left(\sum_l w_{jl} o_l \right) \qquad (10)$$

by examination of equations 7 and 8. If it is assumed that the same squashing function f is used for all units (as is generally the case), then this term is the multiple of the derivative of f and f itself. A plot is shown in figure 3 below using the sigmoid function (bipolar representation) for f:

$$f(act) = \frac{2.0}{1.0 + \exp(-act)} - 1.0$$

It can be seen that for values outside the range *[-p,+p]*, this term is very small for large unit activation values. This means that the change Δw_{ij} applied to the weights on the connections feeding into a unit will also become very small as the unit's activation increases.

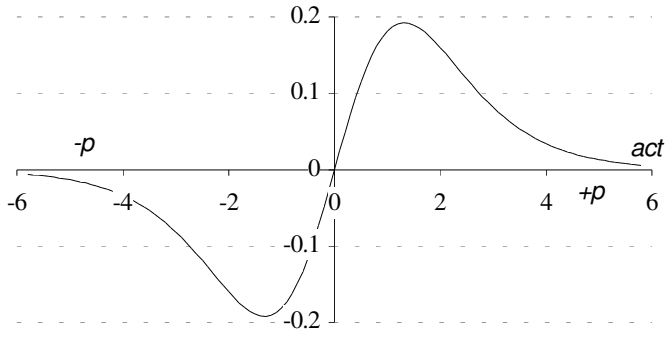


Figure 3 Plot of common multiplicative term in BP algorithm

When training a MLP network, the weight vectors move towards a stable point, and so the weight changes must decrease towards zero. In figure 3 above it can be seen that there are three points where this occurs, and are when a unit's activation tends outwards from $\pm p$ or towards 0. However, a unit having zero activation is very unstable since a slight disturbance causes a rapid rise in the weight change, and so this case is considered most unlikely to occur. This means that units in a back-error propagation trained MLP network will have activation values clustered around $\pm p$ (see figure 4). The simulation result in section 4.6.1 that showed hidden units tend to output their extreme values supports this.

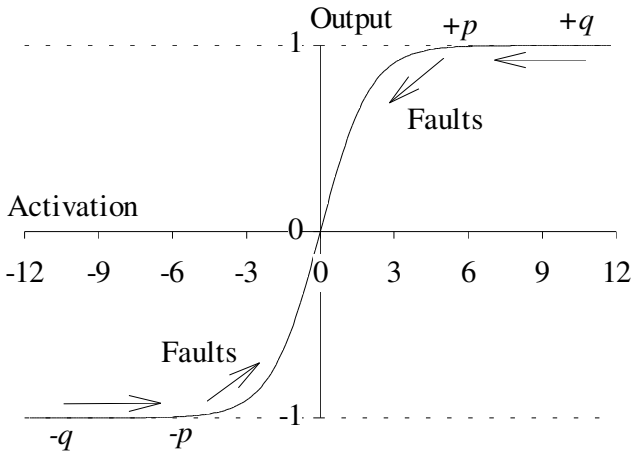


Figure 4 Clustering of units' activations around +/- p

Given this, it becomes clear why a back-error propagation trained MLP is not fault tolerant despite being composed of reliable perceptron units. A single weight fault (either forcing its value to 0 or the opposite extreme value) will decrease the projection of the input onto the unit's weight vector, and so move the activation towards 0 (equation 9). Since the unit's

activation was already close to the point where the squashing function rapidly moves away from its asymptotes, this causes a large error in the unit's output, and so greatly increases the likelihood of overall system failure.

4.6.3. New Technique for Fault Tolerant MLP's

It will now be shown how to overcome the consequence that by training a MLP using the back-error propagation algorithm a non-fault tolerant neural network is formed. In figure 4 above, it can be seen that in the asymptote region of the activation function $\pm q$, a weight fault will not cause an error in a unit's output, so avoiding possible failure. To achieve this, the weight vector of a unit can be scalar multiplied by some suitable constant ζ which will cause the activation of a unit to be likewise increased:

$$\begin{aligned} act &= \left(\zeta \underline{w}\right) \cdot \underline{x} \\ &= \zeta\left(\underline{w} \cdot \underline{x}\right) \end{aligned}$$

This will produce a unit which will tolerate weight faults since the output of the unit will not be erroneous, even though its absolute activation will decrease. If every unit's weight vector is processed in this way, the entire MLP network will tolerate a number of weight faults before failure occurs. This result is supported by the previous analysis in section 4.6.1 of MLP networks trained with faults injected where it was found that the magnitude of weight vectors was greater than those in a normal MLP network.

The feature of neural networks that they show an indication of approaching failure due to graceful degradation [12] will still be exhibited since as more weight faults affect a unit, its absolute activation will decrease into the region of the squashing function's codomain where the function transitions between asymptotic values. This will cause the output of the unit to gradually become increasingly erroneous, and so failure will not be a sudden discrete event.

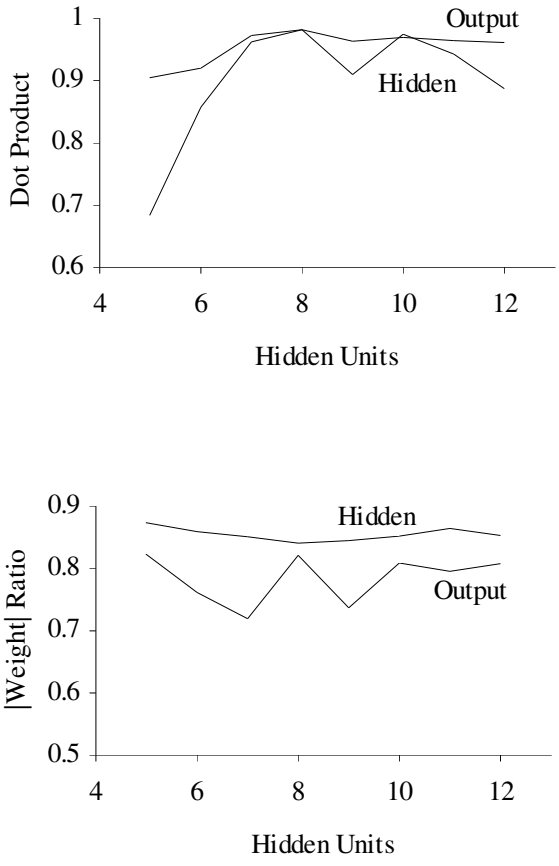
Note that as $\zeta \to \infty$, a unit will behave as if it were hard thresholded (c.f. section 3.1) and provide failure free service until the number of faults equals the Hamming distance between the centroids of its input classes. However, at this point failure will be abrupt since the change in activation caused by each weight fault will not be mirrored by a gradual increase in the error of a unit's output as above. It can be seen that a trade-off exists between the degree of graceful degradation required and the fault tolerance, depending on the value ζ.

The enormous advantage of this method in producing a fault tolerant MLP network is that the training time is essentially only that required for plain back-error propagation. This is a

great improvement over the long training times required to produce essentially the same MLP network configuration when injecting faults during the training session.

4.6.4. Comparison with MLP trained injecting unit faults

For comparison with the above results, simulations were also performed examining the nature of MLP networks developed when training with unit faults injected. The parameters of the simulation were similar in all other respects with its counterparts above which analysed the weight vectors produced when training with weight faults. Graph 6 below compares the MLP networks produced by training with a single weight fault injected to that when a single unit fault is injected.

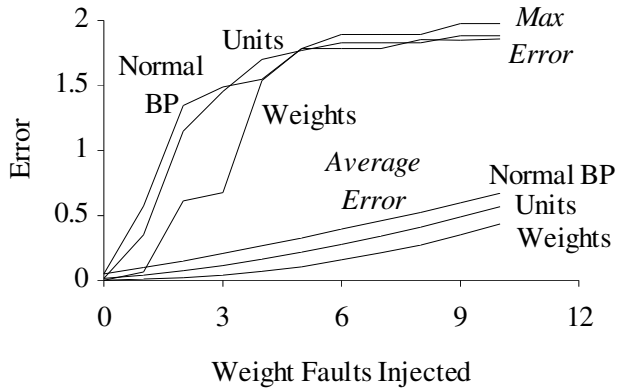


Graph 6 Comparing training with weight faults and unit faults

It can be seen that the direction of the weight vectors in both the hidden and output layers of both MLP networks are almost identical. However, the length of weight vectors in the MLP trained with unit faults injected are less than in the corresponding weight injected MLP. It

will now be shown that this leads to a less fault tolerant MLP network, as was expected in section 4.5.2.

To compare the two fault injection training techniques, a simulation was run training a MLP network on the training set used previously. Graph 7 below shows the results for a MLP network with 8 hidden units. It can be seen that training with weight faults gives improved fault tolerance over unit fault injection training. However, both fault injection training methods do produce a MLP network which is more fault tolerant than if simply trained using back-error propagation.



Graph 7 Comparison of Weight Injection and Unit Injection Training

4.7. Results of Scaled MLP Fault Tolerance

Simulations were performed to examine empirically the fault tolerance of MLP networks with scaled weight vectors. The same training set as used in previous simulations was used so that comparison with their results could be made. The number of hidden units in the simulations ranged from 5 to 12. Note that the MLP networks were trained using the normal back-error propagation algorithm. However, the final weight vectors feeding into the hidden units were then scaled by a factor ζ_h, and similarly ζ_o for the output units, to produce a fault tolerant MLP network.

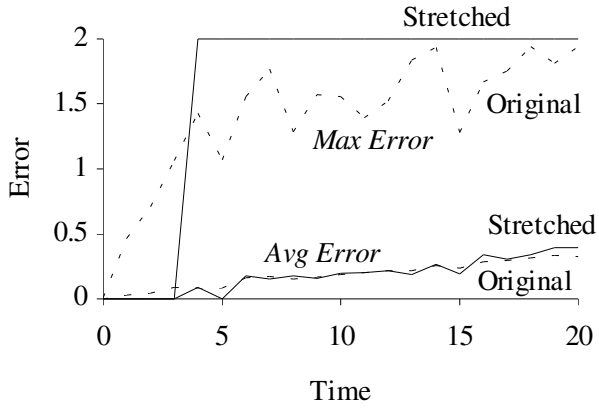
To allow results from MLP networks with various numbers of hidden units to be directly compared, the service degradation method [13] was used to collect reliability data. This requires each fault to be assigned a constant failure rate λ, which together with the equation below, probabilistically models the occurrence of the fault type at time *t*

$$\Pr\left(\text{fault occurs}\right) = 1 - e^{-\lambda t} \qquad (11)$$

A simulation is then started from time t_0, and at each time step every weight is marked faulty according to equation 11. The degree of failure of the MLP network is then measured by some means, and the process repeated for the next time increment.

The measure of failure employed can either be discrete, or as is more appropriate for neural networks [13], a continuous assessment of the system's reliability. The measure used in these simulations was the proportion of inputs in the training set which were misclassified. This can be related to the probability of failure at time t if the selection of input patterns is uniformly distributed.

Graph 8 below shows the results of the service degradation simulations on a MLP with 8 hidden units. Plots labelled *original* are of a normal MLP network, those labelled *stretched* are the results obtained from the same network but with factors $\zeta_h = 1.4$ and $\zeta_o = 100$. It can be seen that the maximum output unit error of the modified MLP network is far less than the original network at initial times $t<4$. At later times, $t>4$, the maximum error in both networks is over 1.0, and hence failure due to misclassification occurs.



Graph 8 Output error of MLP with 8 hidden units over time

However, during this latter period, the average output unit error was approximately the same for both MLP networks. This shows that the fault tolerant network is not sacrificing classification ability to achieve its fault tolerance; this arises purely by allowing the inherent fault tolerance of a perceptron unit to be apparent in the MLP networks' units by increasing their absolute activation levels.

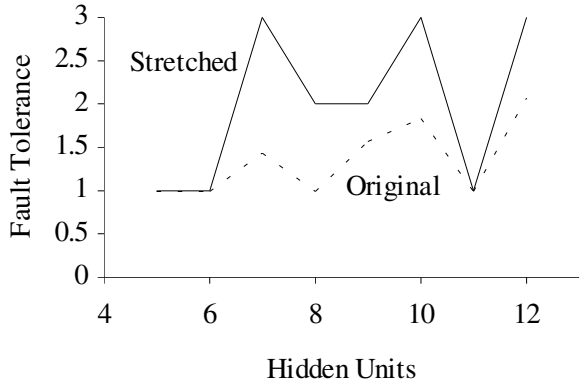
The plots in graph 8 are termed failure curves since they depict the probability of failure in the system due to faults defined in the fault model. A measure for a system's fault tolerance

can be defined as the area bounded by the failure curve until it rises to a point at which system failure occurs. Since a bipolar representation was used in the simulations here, this is when the maximum output unit error reaches 1.0.

$$\text{FT}= \int_{t=0}^{t=t_f} 1.0 - Error(t)\,\mathrm{d}t \quad \text{where } Error\left(t_f\right) = 1.0$$

Note that the area above the failure curve is measured so that increasing values of FT imply a more fault tolerant system.

Using this measure, graph 9 below shows how the fault tolerance of networks trained with the previous weight scaling parameters ζ changes as more hidden units are added to the MLP network. The fault tolerance of the original MLP network is also shown for comparison. It can be seen that the fault tolerance increases as more hidden units are added for both the original trained network and the modified network. As expected though, the fault tolerance of the latter MLP networks is higher than the original.



Graph 9 Fault Tolerance of MLP for various numbers of hidden units

5. Conclusions

This report has analysed the fault tolerance of perceptron units, and concluded that individually they are extremely reliable. However, it was found that a MLP network was not as fault tolerant as might be expected given this result. It has shown that training with weight faults develops a fault tolerant multi-layer perceptron network in a similar fashion as injecting unit faults described in [11]. The trained fault tolerant MLP networks were extensively analysed to locate the mechanism which lead to their robustness. It was found that the both the hidden representations and the directions of weights vectors were not significantly different to a MLP network trained with normal back-error propagation. The only discrepancy was in the magnitude of the weight vectors.

Separate analysis of the effect of faults in a MLP, and the activation of units in a trained MLP, showed how the back-error propagation algorithm results in individual units not being fault tolerant due to insufficient unit activation levels. It was then shown that by scalar multiplying every weight vector by factor ζ, each unit in the MLP would then be capable of exhibiting fault tolerance as suggested by the initial analysis of a single perceptron unit. This leads to better overall fault tolerance in the entire MLP. Finally, simulations were carried out which matched these results.

In conclusion, this report has shown how to allow a MLP network to use the inherent fault tolerance of perceptron units to produce an overall fault tolerant system. As discussed in section 4.4, this is only one area of distributed processing which results in fault tolerance being exhibited by a MLP. The other is to force the development of redundant representations in each hidden layer. Although the simulations above show that as more hidden units are added the fault tolerance of a MLP does increase, it is unlikely that the maximum fault tolerance possible is achieved.

References

1. McCullogh, W.S. and Pitts, A., "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics* **5**, pp.115-133 (1943).
2. Minsky, M. and Papert, S., *Perceptrons: An introduction to computational geometry*, MIT Press, (1969).
3. Werbos, P.J., "Beyond regression: New tools for prediction and analysis in the behavioural sciences" (1974).
4. Rumelhart, D.E., Hinton, G.E. and Williams, R.J., "Learning Internal Representations by Error Propagation" pp. 318-362 in *Parallel Distributed Processing*, ed. Rumelhart, D.E. and McClelland, J.L. (Eds), MIT Press (1986).
5. von der Malsburg, C., "Self-Organization of Orientation Sensitive Cells in the Striate Cortex", *Kybernetik* **14**, pp.85-100 (1973).
6. Bolt, G.R., "Fault Models for Artificial Neural Networks", *IJCNN-91, Singapore* **3**, pp.1918-1923 (November 1991).
7. Damarla, T.R. and Bhagat, P.K., "Fault Tolerance in Neural Networks", *Southeastcon '89 Proceedings: Energy and Information Technologies in the S.E.* **1**, pp.328-31 (1989).
8. Bedworth, M.D. and Lowe, D., *Fault Tolerance in Multi-Layer Perceptrons: a preliminary study*, RSRE: Pattern Processing and Machine Intelligence Division (July 1988).
9. Tanaka, H., "A Study of a High Reliable System against Electric Noises and Element Failures", *Proceedings of the 1989 International Symposium on Noise and Clutter Rejection in Radars and Imaging Sensors*, pp.415-20 (1989).
10. Bolt, G.R., "Investigating Fault Tolerance in Artificial Neural Networks", *YCS 154*, University of York, UK (March 1991).
11. Clay, R.D. and Sequin, C.H., "Fault Tolerance Training Improves Generalisation and Robustness", *IJCNN-92*, *Baltimore* **1**, pp.769-774 (1992).
12. Bolt, G.R., "Fault Tolerance and Robustness in Neural Networks", *IJCNN-91, Seattle* **2**, pp.A-986 (July 1991).
13. Bolt, G.R., "Assessing the Reliability of Artificial Neural Networks", *IJCNN-91, Singapore* **1**, pp.578-583 (November 1991).