Presented to the Staple Inn Actuarial Society

on 9th December 1997

NEURAL NETWORKS
STATISTICS FOR PROFESSIONALS?

by

Sally Bridgeland, Martyn Dorey

and

Jon Palin

Presented to the
Channel Islands Actuarial Society, 24th November 1997

Contents

Glossary *The glossary explains some of the jargon used.*

References *References are indicated by numbers in square brackets.*

1. Introduction

"If we want things to stay as they are, things will have to change."

Giuseppe di Lampedusa

1.1 Big bang

Artificial neural networks are mathematical models. The motivation behind this paper is to introduce the actuarial profession to the modelling technique which is used to create neural networks. By using neural networks for familiar tasks we hope to encourage actuaries to make the most of neural networks in the future.

This section of the paper outlines:

- the features of neural networks and the modelling technique
- why we have chosen the examples used in the rest of this paper to demonstrate the technique's potential
- why using this technique can enable the actuarial profession to extend the scope of the modelling work it can do in the future

1.2 Background

Artificial neural networks are a mathematical representation of the way that neurons (brain cells) link together and fire up electrical activity when humans make decisions [1]. They originated in the 1940s but suffered a major setback when further research pointed to some apparent limitations [2]. In the last decade, these limitations have been overcome through the development of "learning" algorithms [3] and the availability of the computer power necessary to process those algorithms.

Today, neural networks are widely used. Applications include classifying sales information from supermarket loyalty cards, developing new strategies for playing backgammon [4] and producing lower risk investment strategies [5].

1.3 Features

In traditional modelling techniques, a great deal of human time and thought goes into analysing information, inferring relationships and creating a model of those relationships. In an expert system, the modeller develops and programs the logical steps in a chain of decision-making rules. In contrast, the learning algorithms used to create neural networks automate the process of inferring relationships and developing rules.

This automation means that a modeller can create a neural network without having any understanding or prior knowledge of the underlying relationships. The neural network will infer the relationships purely from the information available to it and not from the laws of physics, economics or actuarial science. Those who are familiar with techniques used in lateral thinking will recognise that this freedom from constraints opens the way to more creative solutions [6, 7].

However, the modeller needs to spend time checking a new neural network to make sure that it makes sense. We don't test human understanding of a problem by unravelling the brain's circuits. Similarly, neural network tests involve setting some exams which suit the nature of the problem and the information the neural network has been given, rather than unravelling the model's formulae.

Using the examples in this paper we hope to demonstrate how a knowledge of the relationships being modelled and an understanding of statistics can lead to more reliable neural network models.

1. Introduction (continued)

1.4 Science fact

This paper is not about fashionable mathematics for its own sake. Two of the three authors knew very little about neural networks until they started working on the paper, and both were determined not to be seduced by the glamorous science fiction image. In this paper we have chosen to bring the technique back down to earth by applying neural networks to some traditional actuarial problems.

1.5 Unknown process - mortality

The first example we have chosen (in section 2) is to model the unknown process of death. We have used a fairly simple neural network and Microsoft Excel to produce a smooth curve which fits raw mortality data. The aim is to introduce you to the process, mathematics and jargon of neural networks using a familiar task.

1.6 Complex formula - MFR

The second example (in section 3) tests the power of a neural network model to approximate an actuarial function - that of the Minimum Funding Requirement (MFR) minimum transfer value. The aims of this work were:

1. To create a benchmark for an appropriate network design for pensions applications. We are not aware of any previous work to fit a neural network to actuarial functions, so this is necessary before moving on to more complex pensions problems.
2. To see how using our knowledge of the problem helped produce a more reliable network. Commentators suggest that the more knowledge you have, the better network you can produce [8]. Our experienced neural network modeller had no knowledge of the formulae underlying the MFR, but we drip-fed him with information when he needed it to improve the results.
3. To develop a rigorous process for designing a network. Those who are new to neural network modelling may be surprised to learn that in doing this we were relying on the very latest developments in this branch of mathematics [9]. Our basic process for designing and testing a network and our IQ measure for assessing a network's intelligence both break new ground, so we hope even readers who already have experience of neural network modelling will find our research of interest.

1.7 Extension of GLM

Our final example (in section 4) is about Generalised Linear Modelling (GLM). Whether or not you have some experience of using GLM, our presentation of a GLM model as a particular design of neural network will enable you to compare the modelling techniques. We hope that this will enable GLM users to spot the similarities and use neural networks models and techniques to supplement GLM. In the future, we believe that this will help produce better models because:

- Linear models have their limits: using linear approximations on real life problems can lead to anomalies and uncertainty principles [10]
- GLM has risks: it involves transforming variables to make the input data linear. In problems of any complexity, it requires skill and persistence to understand the relationships between all the variables [11]

Instead, neural networks use data to determine what shape the model should have. Actuaries can then use their knowledge of the problem to create better neural network models.

1.8 Statistics for amateurs

Neural networks have an image problem which may make actuaries wary of embracing the modelling technique. They have been described as "statistics for amateurs" because those with no knowledge of statistics can buy inexpensive commercial software and create neural network models [12]. We feel that actuaries have the mathematical skills to add the rigour necessary to transform the technique into statistics for professionals.

1.9 Natural selection

Actuaries are the natural providers of neural network services. Our profession's motto is "certainty out of uncertainty". We analyse problems so that we can model and quantify risks and help manage them. However, the value of our advice is limited by our understanding and knowledge. Neural networks can help us pick out new patterns and relationships from what we previously might have assumed to be random variations in the data. As a result, our models can fit complex problems better and our advice can be more valuable.

1.10 There's something out there

We believe that the power and flexibility of neural network techniques gives them great potential. We are not alone: governments and private companies around the world are already investing in designing and using neural networks.

We are not too late to grasp this new tool and evolve so that it fits into our current actuarial toolbox. In the final section of this paper (section 5) we provide some pointers for the future and outline other problems which suit neural network techniques.

2. Mortality

"Call no man happy before he dies, he is at best but fortunate.

Solon

2.1 Background

Our first example of using the neural network modelling technique on an actuarial problem was prompted by work done for a previous paper [13]. That paper looked at fitting English mortality tables to mortality data for the Guernsey population.

For this paper, the aim of the work was to find out whether a neural network could be used to produce a mortality table from raw data. The work was carried out in a fairly unsophisticated way by another of the authors who hadn't even heard of a neural network before April 1997. The calculations were all done in Excel rather than using specialised neural network software.

2.2 Traditional techniques

The traditional techniques that actuaries would use on this problem would involve one of the following:

- drawing a smooth curve by hand
- piecing together several likely-looking functions
- adapting an existing table

then testing how well the resulting curve fits the data using a chi-squared test [14].

2.3 Define the problem

The first step in designing a neural network is to be clear about what you want the model to achieve. Here, we have chosen to produce an actuarial mortality table starting from observed deaths and an exposed-to-risk for a population. We are looking for a table of q_x values which is smooth and which fits the data well. We will use the chi-squared test as our measure of fit so that the problem mimics the traditional actuarial one.

We are not going to draw on information from an existing mortality table. The neural network modelling technique involves fitting a function to data so resembles the second one given in the previous paragraph. However, the function that we will use is so flexible that the practical issues are similar to the first technique, ie drawing a smooth curve through the data.

2.4 Examine the data

To draw a curve we would first examine the data. The two main questions we would ask are:

a) Do we have enough data? If we have more data, our brain has more clues when it is trying to visualise the curve and instruct the hand to draw it.

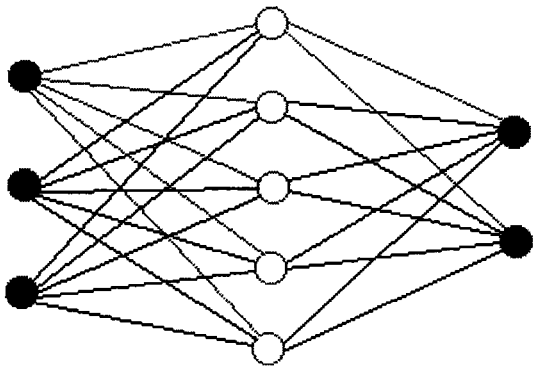
b) How reliable is our data? It will be easier to draw a smooth curve if we have quality data which has minimal noise or random variation.

For our neural network, we have chosen to use the mortality data used to create the male ELT 14 rate [15]. This means that we have underlying population data (number of deaths and exposed to risk) to calculate a raw observed mortality rate at each age over the whole of the range of ages. Given the size of the population, our data is of good quality, except perhaps at the extreme ages.

2.5 The human neural network

The next step is to design the network: the explanation of the design will require you to grapple with neural network terms, based on the jargon used in neurobiology.

Humans make decisions when electrical signals fire round a large and complex network of brain cells. Within this network there are sensory cells which receive inputs; association cells, which combine the signals from the sensory cells; and response cells which pass on the messages from the brain to the rest of the body. The cells are linked by synapses which transmit and amplify (or inhibit) the signals.



sensory cells → association cells → response cells

2.6 The artificial neural network

An artificial neural network is based on layers of "cells" of simple mathematical functions. A typical neural network is a nested function with the following structure:

1. A layer of sensory (or input) cells which could be the inputs or raw data items.
2. A synapse linking each cell in the sensory layer to each cell in the association layer, multiplying the output from the sensory layer's cell by a "weight". The weighted outputs from all the cells in the first layer and are summed together with a "bias". At this stage the function is known as a "linear predictor".
3. A layer of association cells which apply functions ("activation functions") to the linear predictors.
4. A synapse linking each cell in the association layer to each cell in the output layer, multiplying it by a weight. The weighted outputs from all the cells in the second layer and are combined and a "bias" added to this sum.
5. A layer of response (or output) cells, which apply functions (more "activation functions") to the outputs from the association layer.

The mathematics of this structure is given in section 2.7.

This design of network is usually called a "three-layer network" because there are three layers of cells. (The association layer is also known as the "hidden layer" because its inputs and outputs are not visible as direct inputs and outputs.)

Confusingly, some authors prefer to call this a two-layer network, perhaps because there are two layers of synapses and therefore two sets of weights and biases, the parameters in our mathematical function.

2.7 The formula

Expressing this as a formula:

x_i is the input transmitted by the i^{th} sensory cell

a_{ij} is the weight used to multiply x_i before it reaches the j^{th} cell in the association layer

a_j is the bias added to sum of the weighted inputs on the way to the j^{th} association cell

u_j is the linear predictor to the j^{th} association cell

$$u_j = \sum_i x_i a_{ij} + a_j$$

f_j is the activation function used in the j^{th} association cell, so that its output is

$$y_j = f_j(u_j)$$

b_{jk} is the weight used to multiply y_j before it reaches the k^{th} response cell

b_k is the bias added to sum of the weighted inputs on the way to the k^{th} response cell

v_k is the input to the k^{th} response cell

$$v_k = \sum_j y_j b_{jk} + b_k$$

g_k is the activation function used in the k^{th} response cell, so that the final output from the network is

$$z_k = g_k(v_k)$$

Expanding out the overall function gives the output from the k^{th} response cell as a function of the inputs x_i

$$z_k = g_k\left(\sum_j\left[f_j\left(\sum_i x_i a_{ij} + a_j\right) b_{jk}\right] + b_k\right)$$

2.8 Design the network

The designer of the neural network chooses the number of cells, the way in which they are arranged and the activation functions they use. For our problem, we have one input (age), one output (mortality rate) and will start simply with one cell in the association layer. This makes the process and mathematics as transparent as possible.

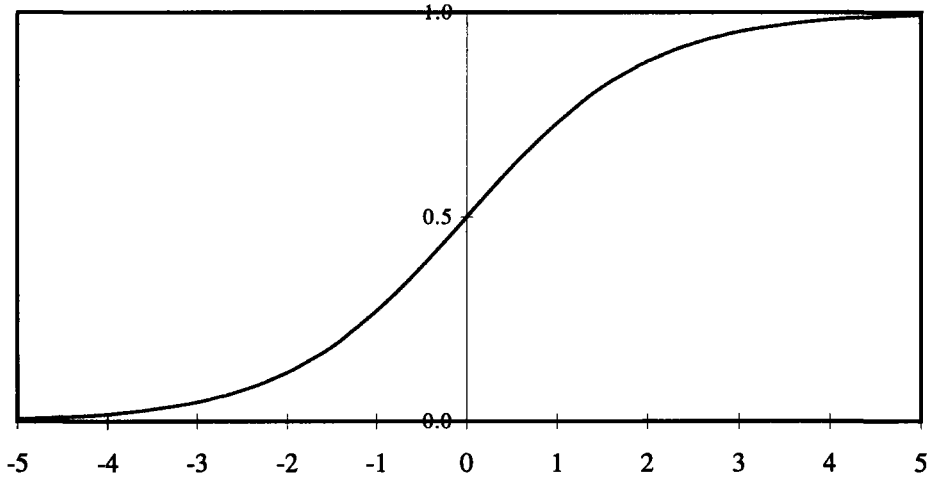
Having chosen something simple for the number of cells, we can be more adventurous with the activation function. The traditional neural function comes from the "sigmoid" (or s-shaped) family. This tradition has been built on a proof that a neural network which uses two layers of sigmoid functions (as the activation functions in the second and third layers) can model any function [16]. Indeed, Kolmogorov's Mapping Existence Theorem (1957) [17] showed that a neural network with a single layer of sigmoid functions could model any continuous function to any required degree of accuracy. However, to do this, you may need to combine a very large number of sigmoid functions and cells in the neural network's hidden layer.

2.9 The sigmoid activation function

We have chosen a sigmoid function for the cell in the association layer. It takes the form:

$$s(u) = \frac{1}{1+e^{-u}}$$

A graph showing the shape of this function is shown below.



In our model we have only one input so the hidden layer receives:

$$u_1 = a_{11}x + a_1$$

where x is the age. The weight is a_{11} and our bias is a_1. The output from the hidden layer is $y_1 = s(u_1)$.

Drawing y_1 as a function of x would give a gentle slope if the weight were small, increasing to something close to a step function as the weight gets larger. Changing the bias changes the point at which the function y_1 crosses the y axis and the horizontal position of the steepest part of the curve.

To keep the mathematics simple, we will do nothing more to the output than use a linear transformation in the final layer, so our final output is:

$$z_1 = v_1 = b_{11}y_1 + b_1$$

This means that our modelling function for the mortality rates is:

$$q_x = \frac{b_{11}}{1+e^{-(a_{11}x+a_1)}} + b_1$$

2.10 Learning

As our brain develops, the synapses develop and change so that the links between cells change, and we learn. Memories are created when the links are well-formed and the trigger of a sight, sound, smell or exam question is enough to set the brain working along a familiar pattern.

The network's learning process involves finding the best values for the parameters, ie the two weights a_{11} and b_{11} and the two biases a_1 and b_1. The "best values" are determined with reference to a "cost function" which measures how close the network's output is to its target function. In this example, our cost function is the chi-squared test.

2.11 Educating our network

The training process we have used involves the following steps:

1. Give the network some examples of the target function (the "training set"). Here we used all the available data for our training set. This makes it as close as possible to the traditional actuarial approach.
2. Pick some initial values of the weights and biases.
3. Calculate the cost function over the data in the training set. Here we set up all the data and the network formulae in an Excel model. We calculated the chi-squared error by comparing the network's output and the raw mortality rate at all the ages.
4. Use a learning algorithm to change the values of the weights and biases and return to 3.

A bit more jargon before we continue:

- Each cycle around the loop is an "epoch"
- A graph showing the value of the cost function at each epoch is the "learning curve"

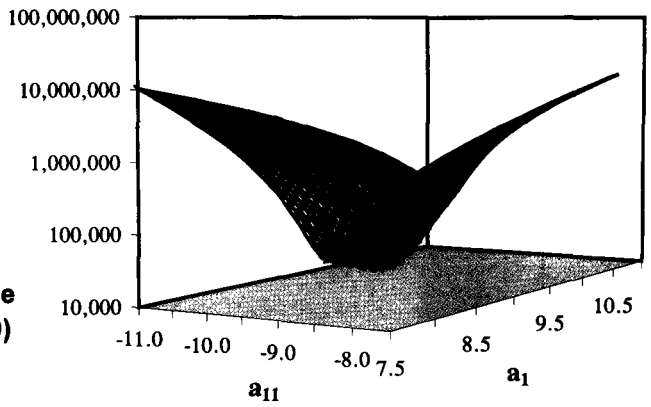
So, if the training time is long enough and the method of adjusting the weights and biases is reliable, we would expect the learning curve to go down and then level off before the end of training.

2.12 Initialising the weights

We only have one hidden cell and the final layer is linear. So we can set likely starting values for the weight and bias by plotting the "error surface". This shows graphically how the cost function varies with the different weights. The training process is a minimisation problem, aiming to find the values of the weights and biases which give the lowest point on the surface.

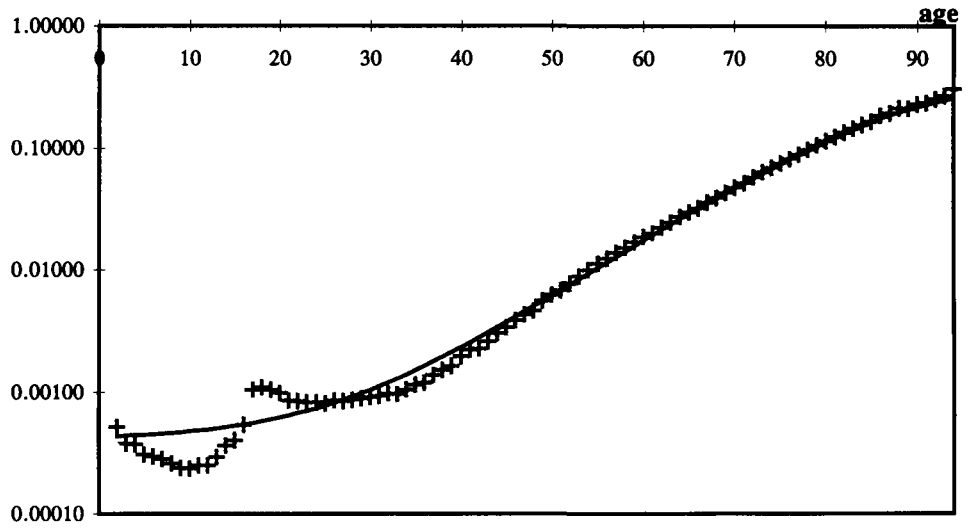
The following graph shows the error surface for our mortality problem. Close to its only minimum point it is a well defined valley. Looking at this error surface we were able to pick some likely starting values for the weight and bias in the hidden layer.

Error surface (b_{11}=1; b_1=0)



2.13 Training

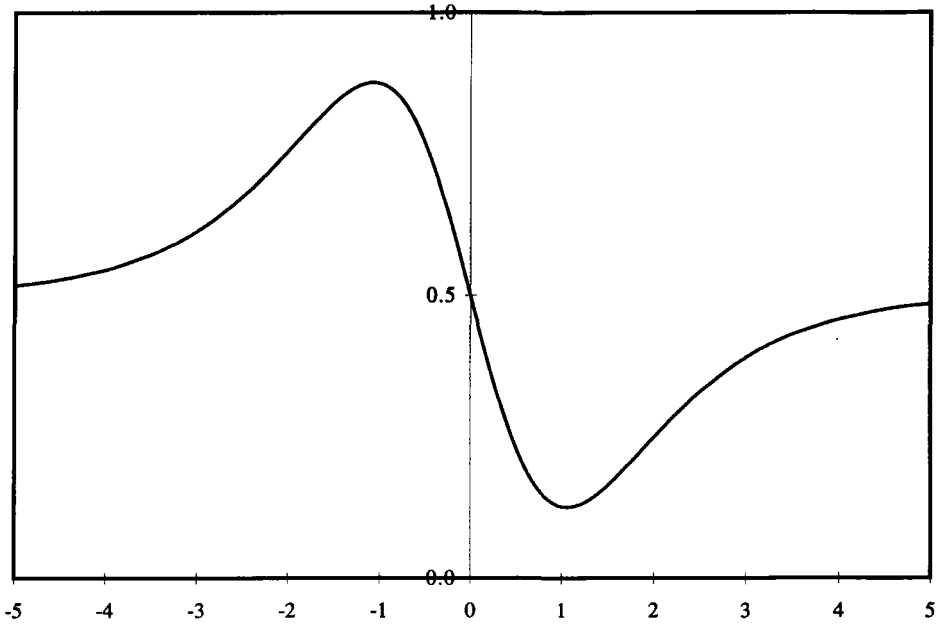
Having picked reasonably good starting values we used the Solver Tool in Excel as out learning algorithm to minimise the chi-squared error by varying our four parameters. The model this produced is shown by the line on the following graph (with the raw data shown by crosses at each age):



2.14 Redesign

In actuarial terms, our one-celled model is overgraduated, ie it is smooth but its adherence to the data is poor.

With only one s-shape to work with, we cannot hope for anything better. By adding cells we can add bumps. The sum of two sigmoid functions is shown below, with weights of +1 and -1 and biases 0 and 1 respectively.



The second layer of synapses in our network transforms the activation functions from each cell, to give them each a different slope and setting the combined position.

2.15 More cells

With more cells it is not possible to display the error surface graphically as there are too many parameters to represent, so we needed to find another way of initialising the weights and biases. We tried picking the initial values randomly, but the Excel Solver did not return good results from these starting points.

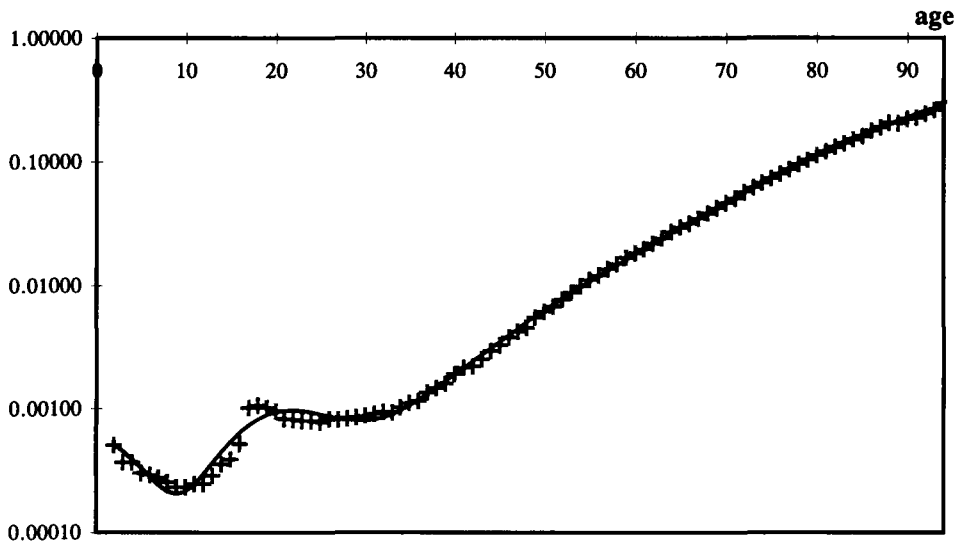
Most neural network models are trained using more sophisticated algorithms than those used in the Solver, although they are built on the same basic principles. As in other methods used to find the minimum of a function, information about how the error changes as the weights change is used to estimate how to change the weights to reduce the error.

However, in a neural network, having two layers of parameters complicates the algorithm. Changing any parameter will have an impact on the error. The algorithm is known as backpropagation of errors because changes in the ultimate error are fed back to produce the changes necessary in the second and then the first layers of weights. Appendix B looks at learning algorithms in more detail.

We found that continuing with more cells and using Excel Solver was possible, but only by using our knowledge of the problem. This involved looking at the shape of the data and creating a caricature of its shape by adding in cells with weights and biases which would give curves roughly in the right place. Excel Solver then finished off the job.

2.16 Results for more cells

The graph below shows the neural network solution with three cells. This has a much better shape.



2.17 Dangers

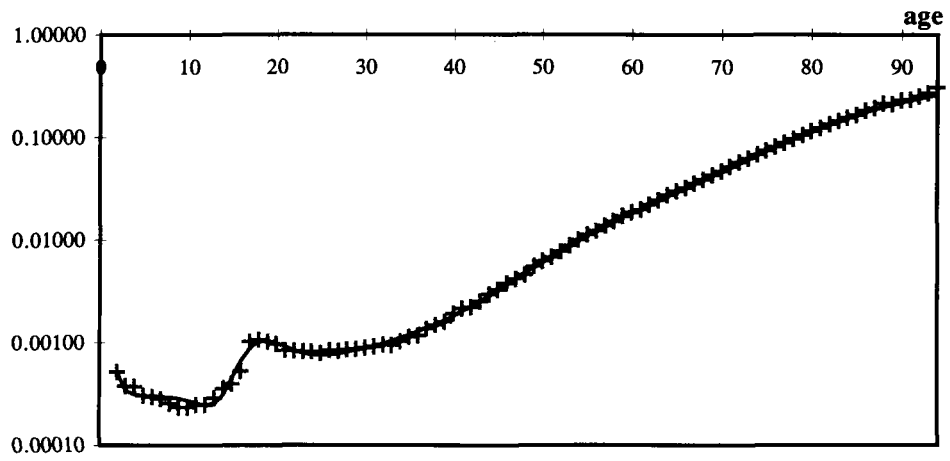
It is possible for the network to work too hard. There is a risk of this happening if:

1. The data is "noisy". If there are random errors in the data that the neural network is trying to model, the network may model the noise as well as the underlying function. Apart from at the extreme ages, we had reasonably good quality data with low levels of noise. If there had been more noise the network wouldn't be able to distinguish between noise (which we do not want to model) and the "accident hump" at around age 20 (which we do).
2. The network has more than the minimum number of cells that it needs to solve a problem. If we had one cell for each age we could produce a network which fitted each data point perfectly but which stepped fairly steeply between each point in the training data set, or a network which drew a gently curving line between each point. The actuary can choose between these by applying a range of standard tests. Alternatively by choosing a suitable network design or training method we can encourage the network to produce the preferred result.

2.18 Too many cells?

Here we tried to make sure that the network didn't get too smart by keeping the number of cells low. This, combined with the chi-squared test, forces it to draw a curve which actuaries would accept as smooth. The model wisely uses its limited brainpower to look for a pattern, rather than be too clever by learning the data by heart.

The results below show the output from a six-celled network model. The chi-squared test would tell us to accept this model as it fits the data well, but the shape differs from what we might expect between ages five and fifteen. Is this something that we should reflect in a mortality table, or might the data at young ages be too noisy to be reliable?



The traditional way to detect when a neural network is modelling the noise as well as the underlying pattern is to split the data into two sets. One set is used to train the model. If the neural network has "overfitted" the training set it will have modelled the noise in the target function for that data set. As the noise is random, the noise in the other set of data (the "test set") will lead to target data points which are not well fitted by the neural network's output. This lack of fit is measured by the cost function: the error for the test set of data will be much higher than the error for the training set.

2.19 Related problems

From our results it seems likely that you only need a few neurons and basic technology to come up with a neural network model for English male mortality. The rest of this section of the paper illustrates how a change in approach might be necessary to use neural network techniques for a couple of related problems:

- deriving a mortality table when data is sparse or noisy
- predicting future trends in mortality

2.20 Life is different in Guernsey

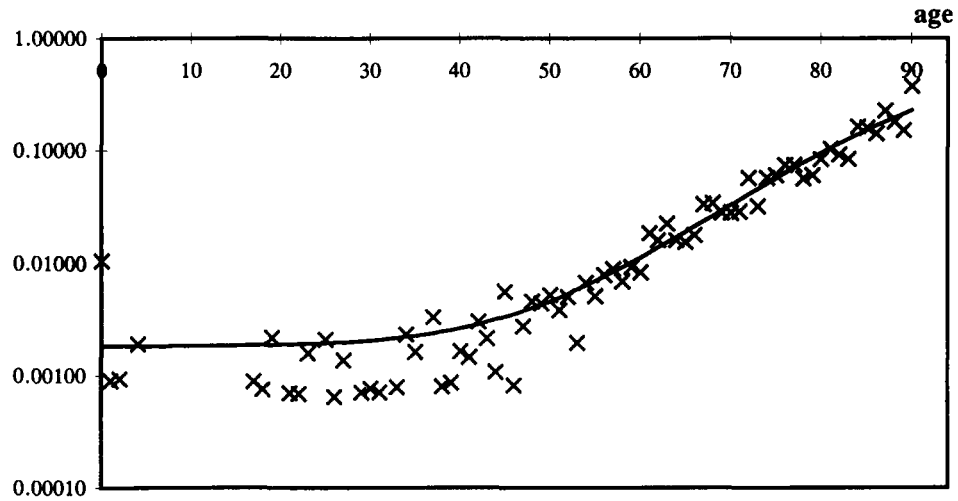
Two of the authors were born in Guernsey and have forsaken that safe island for the dangers of the mainland. In previous work [13], neural networks had been used to derive a Guernsey Life Table (GLT) by using ELT tables. However, previous research had shown that mortality experience on the island is different from the mainland [18, 19]. Could a neural network come up with something from scratch, or would there be problems arising from the sparse data?

Guernsey is a wild and charming island with a population of around 60,000 [20]. We obtained the data from the public records office at Greffe and from censuses [21]. At each age, the exposed to risk is not much more than 1,000 lives: it is significantly lower at the higher ages. In our data, the numbers of deaths at each age is small, sometimes zero.

2.21 Coping with sparse data

We used the same approach as with the mainland data: one sigmoid cell in one hidden layer, building up the cells as necessary to reflect significant bumps in the data.

With one cell the curve looked reasonable and the chi-squared result was good, apart from a large contribution from a sinister number of infant deaths where our single s-shape could not cope. In the following graph, the network's model is shown as a line. Data points are shown as crosses: a logarithmic scale is used for q_x , so zero values are not shown.



2.22 Same result

To improve the fit, we increased the number of cells. Three had been enough to pick up the curves in the mainland data: we tried the same for GLT.

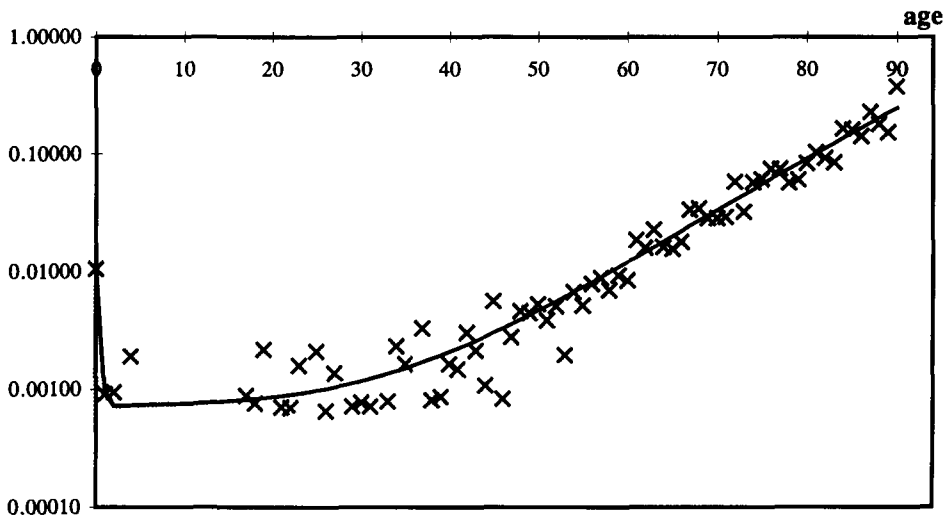
We had to come up with a reasonable way of initialising the weights. We tried two approaches:

1. By hand. As with the ELT model, we positioned bumps in suitable places (in particular to pick up the infant mortality).
2. Using the weights derived from the three-celled ELT model.

The two resulting models are illustrated in the graphs below and on the next page. They show the value of knowledge of the problem, and the dangers of creating a neural network solution using something like a chi-squared test alone. Both models had chi-squared values around 85, but had quite different shapes.

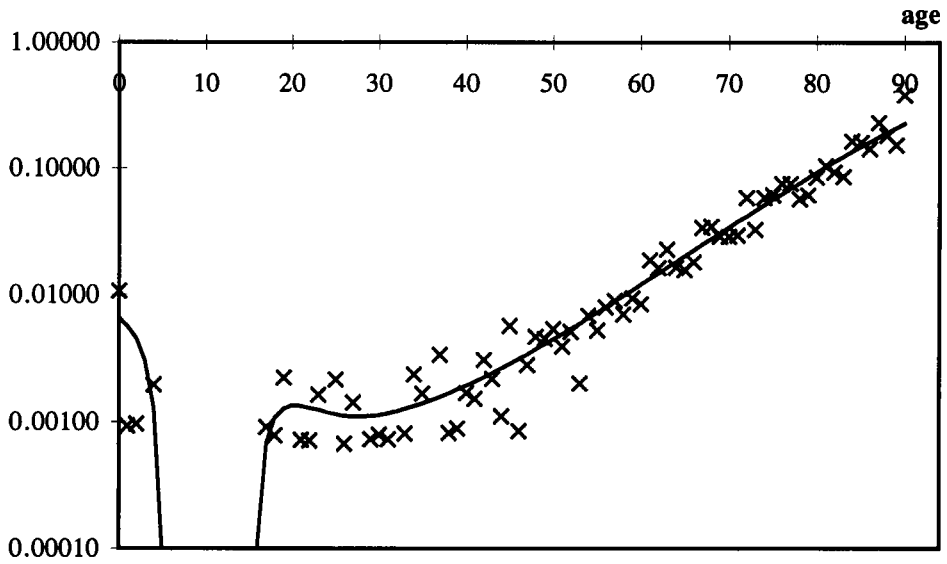
2.23 Different models

Our hand-started solution reflects the likely patterns in the data.



Using the ELT model as a starting point, we get a different model (illustrated on the next page) which mimics the actual patterns in the noisy (sparse) data. The two extra s-shapes create a steep-sided trough for the ten-year age-range over which there were no recorded deaths. This gave good values for the chi-squared error. However, to compensate for weaknesses in the data and reflect the features of the problem you need to supplement a chi-squared test with something more specific. Or give the network some clues.

2.23 (continued)

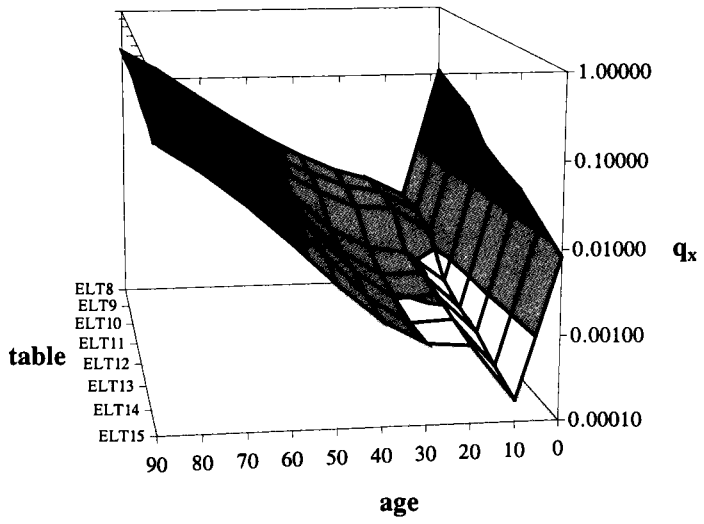


2.24 Predictions

Having modelled ELT data, another problem prompted by a recent article in The Actuary [22], would be to predict mortality of the future. Can a neural network model the function ELT(x,t) where x is age and t is time? We have shown how it can interpolate and smooth between data points, can it extrapolate? We have a sample of data points for this function, illustrated in the graph which follows, where improving mortality is shown as sloping down towards us.

Currently the preparation of an ELT depends only on recent experience. It may be possible to obtain better results by referring to previous data as well. A neural network seems ideally suited to this.

There are other designs of neural network which are better suited to modelling time series. This problem may also need more sophisticated software. However it would make an interesting project for further research.



2.25 Conclusions

Although the idea of a neural network is very simple, the modelling technique needs our assistance to produce a model which would meet all our requirements for a mortality table. Producing a good result is far from easy without specialist software. Even for a fairly simple system with three association cells there are ten parameters to manipulate when we are looking for the minimum of our cost function. Simply picking a random starting point and letting a package like Excel work away will not in general produce good results. However, doing this with many different starting points for the weights is an improvement.

Computer power and complicated algorithms can help the optimisation problem, but we are still faced with a non-linear system that is highly dependent on initial conditions. The modeller needs to take action to control the chaos which could ensue.

2.26 Defining the best solution

In all but a few very simple cases it is not possible to claim that we have the best solution. This is not surprising: there is more than one way to arrange and transform a series of functions to get another function, particularly when given plenty of different basic building block functions to play with. Although this seems unsatisfactory, it is only one of the many questions in neural network modelling which are answered either by vague rules of thumb, or not answered at all:

- Is there a way of designing and creating a neural network to do any given job in the best way?
- Is there a way of presenting the data to the network which will improve the learning process?
- Is there an optimum number of cells? Can we do the same job with fewer cells?
- What cost function is the best measure for testing the accuracy of the network?
- Are there qualities or characteristics of the weights and biases which make for more robust networks?

In the next section of this paper, we develop some answers to these questions.

3. Minimum Funding Requirement

"Mankind always sets itself only such problems as it can solve; since, looking at the matter more closely, it will always be found that the task itself arises only when the material conditions for its solution already exist or are at least in the process of formation."

Karl Marx

3.1 Actuarial networks

Research has shown that a suitably structured neural network can be used to model any formula [16,17]. However, to our knowledge, a neural network has never been used to model an actuarial present value formula, with the complications of mortality, interest and benefit increases. The Minimum Funding Requirement (MFR) calculation seemed an attractive first target for a neural network: it is new and reasonably complicated.

3.2 New minimum

Pension administrators around the country have been busy over the last few months programming and testing their new transfer value calculations. As always, the calculations should follow the instructions certified by the actuary. Since 6 April 1997, the calculations must not fall below the minimum transfer value calculated on the MFR basis [23, 24].

By using a neural network for this purpose we are not trying to suggest that actuaries should be using neural networks to replace spreadsheet models or pensions administration systems. Although, if we had clients who were hooked on neural networks, we could help them produce reliable models if we had a benchmark neural network for this problem. In practice, the commercial advantages of using a neural network come from its speed and its ability to cope well without complete data.

3.3 Faster

Once a neural network has learned to model a problem it comes up with an output from the various inputs virtually instantaneously, by processing data in parallel rather than in series. For a given set of inputs the trained network will always produce the same result. It can do this as a spreadsheet function. The neural network could be trained to cope over a wide range of cases. It could be used as a quick independent check of the calculations produced by the administration system.

3.4 Filling in the gaps

The neural network model could also be trained to "understand" enough about the complex actuarial formula for it to be able to generalise and cope with missing data and data errors. The commercial applications would be to perform a "back of an envelope" MFR valuation of a whole scheme on sparse data, or cope with calculations where information is unreliable or missing eg for the personal pension loss assessment.

3.5 SafetyNet

Our aim was to produce a neural network ("SafetyNet") which could calculate the MFR transfer value for a member of a particular design of scheme, given some raw membership data.

Defining the problem more closely, we decided that our measure for success would be to be within 1% of the correct transfer value amount consistently for all cases within a realistic range. This was one of the targets used when the Institute and Faculty of Actuaries were discussing the professional guidance with the DSS. The idea was that the MFR basis needed to be specified sufficiently in the guidance so that if two actuaries followed the guidance then they should not come up with answers which were more than 2% apart.

3.6 Simplifications

We made the following simplifications:

a) Only one benefit structure. This would be similar to an administration system where calculations are only performed for any one scheme. Different scheme designs can be catered for by bolting on another network which classifies different scheme designs according to the value of their benefit structure and adjusts the basic calculations.

b) Excluding elements of the calculation which relate to market values. This meant excluding the market adjustment factors, but calculating separately the elements which have the different adjustments applied to them.

c) Doing all the calculations at one date (6 April 1997). We felt that learning different rates of revaluations for Guaranteed Minimum Pensions (GMPs) would be an unnecessary distraction from the primary problem and were an extension which we could build in later.

All the networks we refer to in this section of the paper involve these simplifications.

3.7 Inputs and outputs

To start with, we took real data from a pension scheme with 5,000 members.

For each member, we used as our 25 inputs: sex, date of birth, date joined scheme, date joined company, final salary, final taxable earnings, and National Insurance (NI) contributions (or band earnings) for each tax year covering the period from 6 April 1978 to 5 April 1997.

For our outputs, we split the transfer value into two parts so that we could apply different market value adjustments for those close to retirement and complete the calculations. We also decided that we would give the network a third output equal to the sum of the first two.

3.8 Statistical model

We trained a neural network on this data, got some encouraging results, and then thought again. We were worried that we may not have enough data for the network to learn the problem to the accuracy we required.

More importantly, it wasn't the right data for the problem: what we wanted to be able to do with the network was to calculate the liability for any member of a scheme with the same benefit design. Trained on this data, the neural network would learn patterns and relationships within the data and create a statistical model of our real scheme, not model the MFR transfer value formula. It would have been fine for similar schemes with similar membership profiles, but would be at its best with "typical" members.

3.9 Data creation

So, at the risk of creating some fairly atypical members, we generated some random data. This data incorporated some reasonable constraints (dates of birth were randomly spread within a range of likely ages; for each member date of birth and date of joining the scheme were at least 20 years apart; and NI contributions were consistent with the member's service and salary).

In this way we produced just over 32,000 members ("data sets") being the maximum that our computer memory would allow us to handle. We then calculated accurately the correct transfer values for each of these data sets. So there was no noise in the target data.

3.10 Scale the data

The next step was to scale the input data to make each data item a similar order of magnitude. The rationale behind doing this is that if you have data items which are small mixed in with data items which are an order of magnitude larger, the network output and error will be more sensitive to the same percentage change in the larger data item than the others. The training process will be distracted and will not focus on the input items equally.

3.11 Test run

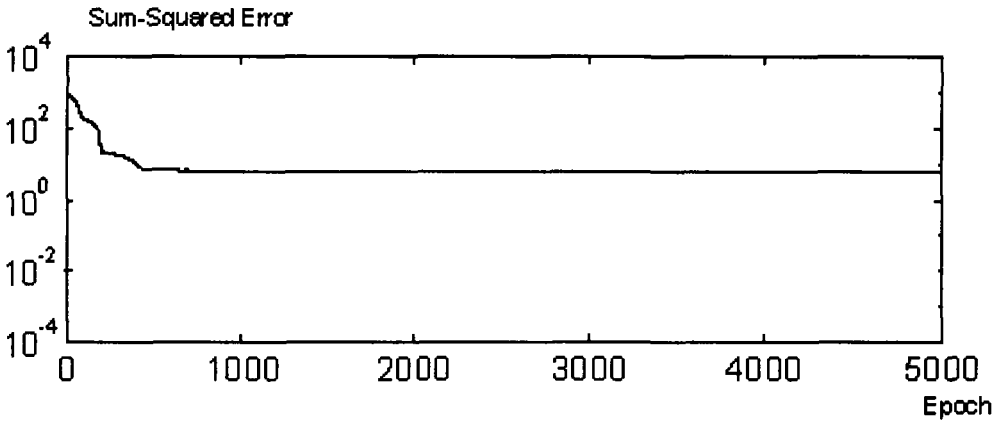
The first test run used sigmoid functions in the two cells in the association layer. We used sum of squared errors as the cost function, to appraise the neural network during training.

We used MATLAB® software to set up and train our network. We started off with randomly-chosen weights and biases. For each epoch or training cycle, the network came up with its 32,000 attempts at matching the target function for each data set and calculated the sum squared error. The backpropagation algorithm used in MATLAB® adjusted the weights after each epoch to move closer to a minimum position on the error surface.

3.12 Learning curve

The progression of the error with each calculation is shown on the next page. It shows that over 5,000 steps, the error is settling down very quickly, but doesn't make much progress after 1,000 steps. The shape of the curve is also interesting. If it were closer to a straight line, the network would be coming up with a fairly simple and essentially linear model. Having initialised the weights randomly, we are giving the model a non-linear starting point from which it made good use of the sigmoid functions in the two cells in the hidden layer. As a result, the learning curve is lumpy.

3.12 (continued)

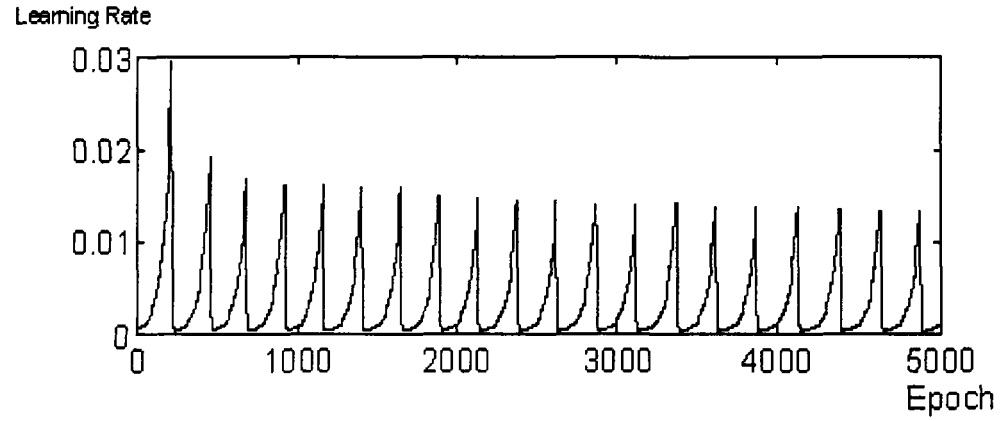


3.13 Adaptive learning rules

The next graph shows the learning rate - the size of the changes in the weights produced after each epoch. The learning rate was set to start at a small amount. The algorithm then used an adaptive learning rate rule to increase the step size:

1. If the error is reducing, the step size increases, gradually building up confidence and taking larger strides.
2. If the error then starts increasing, it quickly takes the hint and dramatically cuts back the step size. This is because it has reached a point where the algorithm is getting nowhere or going backwards. This may be because it is stepping over a minimum on the error surface and can only descend further by taking smaller steps.

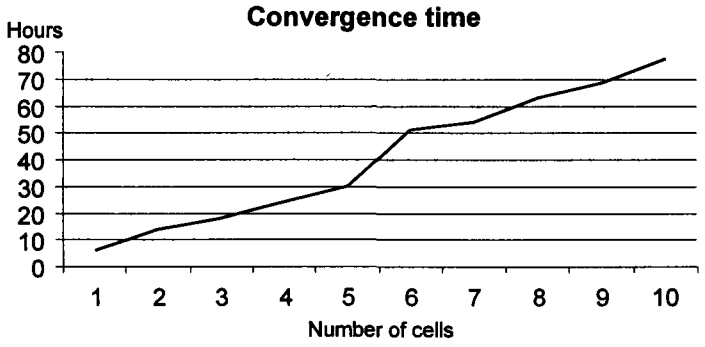
After 1,000 epochs, the peaks show a slight downward trend, but essentially this pattern shows that we are stuck in a minimum and not getting out of it. But with just two cells, it is likely that this is the only minimum so is as far as we can go.



With more cells, the pattern is usually less regular, but can provide useful information on how to change the parameters used in the adaptive learning rate rule. For example, it may need to take bigger steps to avoid local minima. More sophisticated learning rules can be used which change as the learning progresses, taking smaller steps as the change in error reduces, to help at the end of training.

3.14 Increasing number of cells

Although the two-celled model was picking up the non-linearities, it was not getting the error down to a satisfactory level. The next training runs involved increasing the number of cells by one each time and looking at the results. To the uninitiated, it may come as a surprise that backpropagation takes some time: the chart shows the time it took for the model to converge. So training time is a practical constraint.

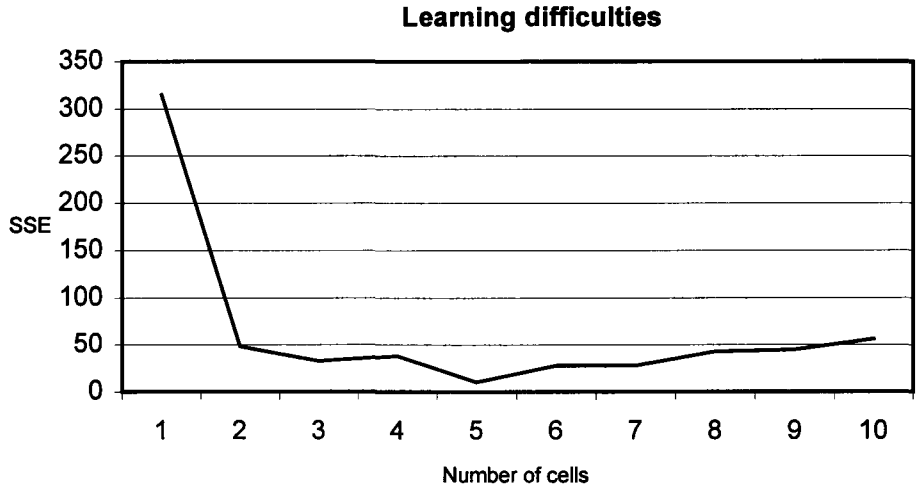


3.15 Patience

But was it worth the wait? The following graph illustrates how the final error changed as the number of cells increased. So, to begin with, yes. But later on, no. This is disappointing, and appears illogical. With more cells, you should be able to model the function better and the error should carry on going down.

But there is a logical explanation: as the number of cells increases, the model may get more bumpy. This is fine if you are trying to model an accident hump, but not good when you are trying to model a smooth function like the MFR. A bumpy model and a smooth target function will mean that the error surface will also be bumpy so the backpropagation algorithm may get permanently stuck in a local minimum [25,26,27]. From our knowledge of the MFR formula, you need more than five neurons to solve this problem.

The traditional way to improve performance involves using more cells, a more complicated algorithm and a consequentially longer training time. But before we tried this, our main concern was that we were still far away from producing a model that came close to passing our tests.

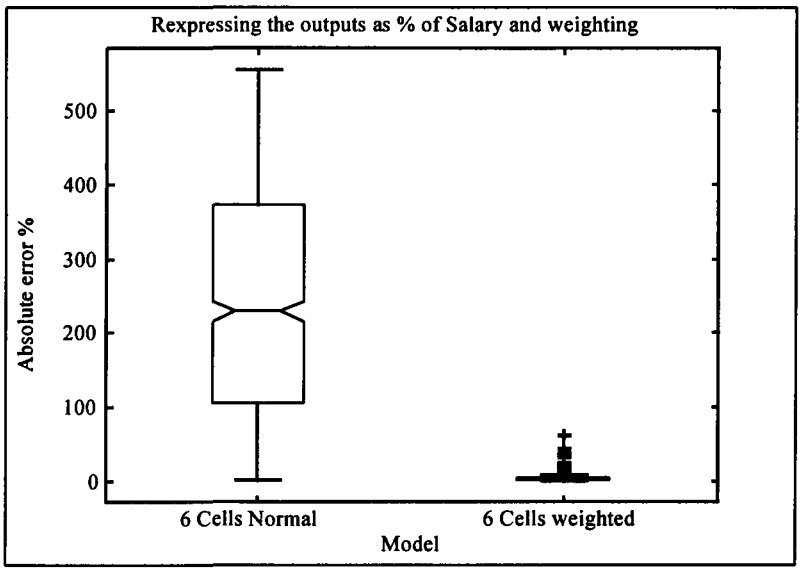


3.16 A little bit of knowledge

To reduce the error, we started using our knowledge about the problem. Instead of having the parts of the transfer value as absolute amounts we decided to re-express them as a percentage of salary. We hoped that, by making the relationship simpler, this would lead to lower errors and a lower range of errors.

The box and whisker plots below show the error distribution for two six-celled networks trained in the same way, but with the second one using the re-scaled outputs. It shows the error distribution for the 2,000 randomly-chosen test cases.

The centre of the box shows the median error. The top and bottom of each box show the second and third quartiles and the whiskers cover the first and last quartiles. Crosses represent the outliers for which the neural network is performing poorly. Not only is the average error significantly lower, but the spread of the errors became dramatically narrower with the re-scaled outputs. Our cost function - the sum squared error - also reduced from 40 down to 15.



3.17 Inputs - network IQ

Before we continued, we also decided to take a closer look at the neural network's workings. We have defined a measure which we have called the network's "IQ". It measures how well the network is using the information from the various inputs.

The first step in calculating the IQ is to calculate each input's "variable contribution" to the accuracy of the network's output for a particular example [5]:

$$\frac{\left|\left(target - output_{average}\right)\right|}{\left|\left(target - output_{actual}\right)\right|}$$ where:

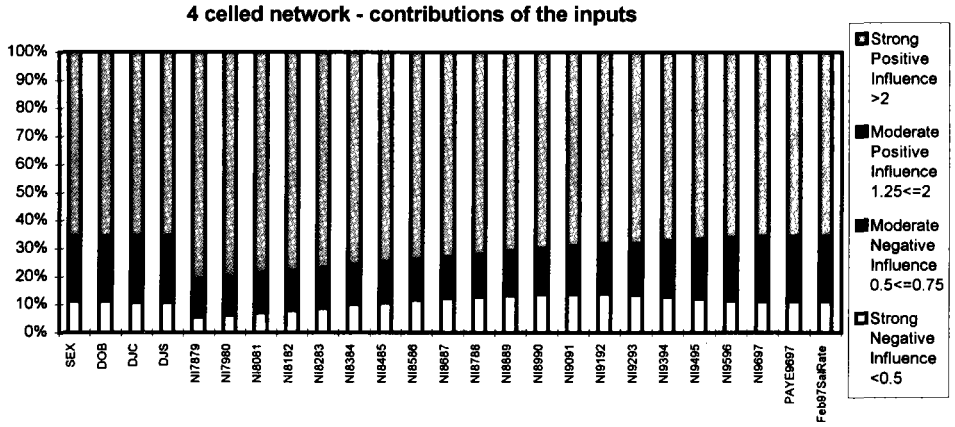
target is the correct output for that example

$output_{average}$ is the output from the network if one input is replaced by the average value over all the data sets

$output_{actual}$ is the output from the network if it is given the actual input

3.17 (continued) The higher this value, the more the network is relying on having the full data to get close to the correct target value for that example. Where this measure is greater than 2, we have defined that as the input having a strong positive influence on the results. Where the variable contribution is less than 0.5, the network seems to be doing better with an average value than the correct value for the input so we have defined this as a strong negative influence.

This is illustrated in the following chart which shows, for each input, the distribution of the variable contributions over the whole data set. To emphasise the positive and negative variable contributions, the chart ignores variable contributions between 0.75 and 1.25.



Conceptually, this approach is similar to looking at the correlation of the inputs with the results.

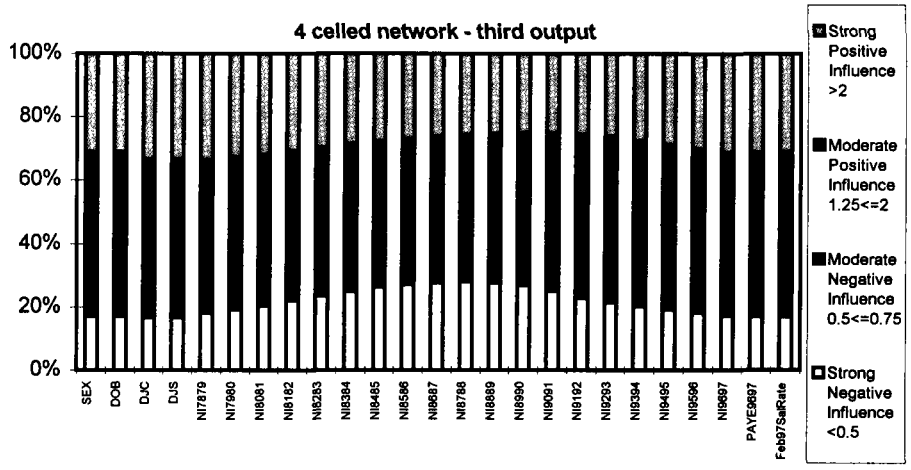
For a neural network, IQ is defined as being the percentage of times over all the examples and all the inputs that the input variables are having a positive influence on the results (which we have defined as being those over 1.25). From the above chart it can be seen that the positive outweigh the negative variable contributions. For this network, the IQ is over 80%.

3.18 Distractions From the chart, the network seems to be treating all the NI contributions in a similar way but placing more importance on the older NI contributions. This is reassuring and ties in with our knowledge of the problem. A change in the older NI contributions has a greater effect because of the impact of compound interest (revaluations).

At this point we were also struck by the similarity of the profiles for the two input items "PAYE9697" and "Feb97SalRate"; and for "DJC" and "DJS". Indeed, we had generated the same data for each pair of inputs. Interestingly, the network had not ignored the duplication, but had treated each input in exactly the same way. Rather than waste its learning power, we decided to cut out the duplicate inputs for future training runs. The lesson to be learned for the future was to look for high correlation in the inputs before running the model to see if we can eliminate any inputs and save time.

The network was also having difficulty with the one of the three outputs. The chart below shows the variable contributions from the inputs for this third output.

3.18 (continued)



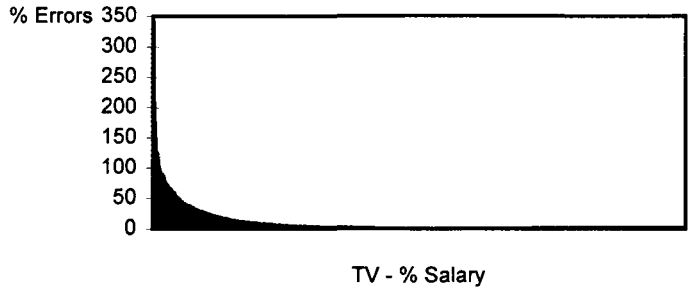
This shows that the neural network is making less use of the inputs - there is a suspiciously random looking distribution of the variable contributions, suggesting that the network is not being too clever. It has a much lower IQ value (33%) for this output.

One reason may be that this output is very different from the other two outputs - and is sometimes zero. With this number of cells, the network finds the other values easier to learn because they have a similar shape. As we can get this output by taking the difference between the other two, we decided to abandon the troublesome output at this stage.

3.19 Error pattern

We then turned to looking at the errors in more detail. We had used a sum squared error over the whole training set for our cost function. We went on to test the network using 2,000 randomly chosen test cases and working out the percentage error - to see how this compared to our 1% target.

By looking at the errors we can also see if they are random. If they show a pattern we can try and eliminate that pattern and get closer to our target. From the graph shown below (for a six-celled network), the smaller the transfer value (as a percentage of salary) the higher the error. This is consistent with having used sum squared error as the cost function: the higher amounts of error will have been more influential in the minimisation process. As our real criterion is based on percentage error we need to use a cost function that reflects this better.



3.20 New cost function

The most obvious cost function would be to use absolute percentage errors for training, ie $|target - output|/target$

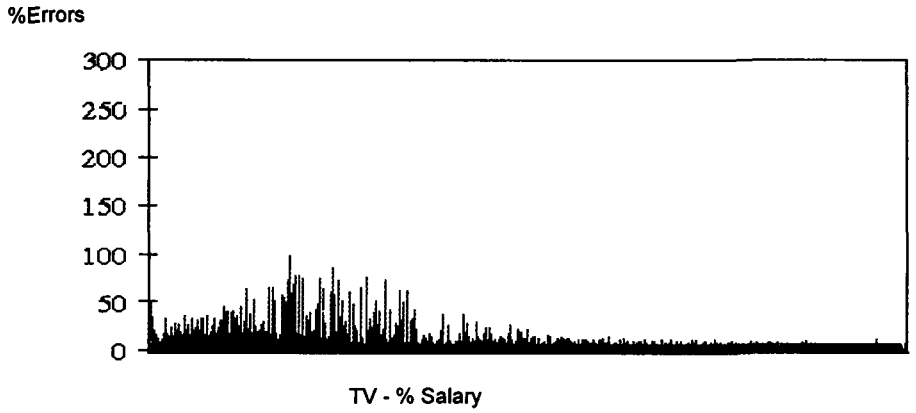
However, this might give values of the cost function which are close to infinity for small target values, which might cause the backpropagation algorithm to crash. A better approach would be to use a cost function of say:

$|target - output|/(target + 1)$

Indeed, the pattern of the errors above suggests a cost function of:

$|target - output|/e^{target}$

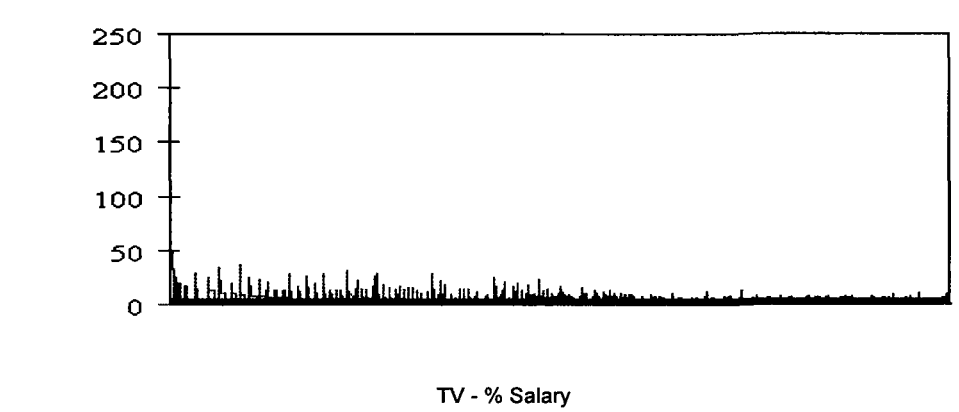
In both cases, the algorithm needs to be reprogrammed (because the cost function and its partial derivatives with respect to the weights and biases has changed). The error surface will look different so the other parameters used in the learning process (eg the adaptive learning rate rule) will also need to change. The best way to do this is to carry out test runs with a reasonably small number of cells and compare the results with those for the same number of cells on the original error measure. The results for the "target+1" cost function are shown below. Disappointingly, there is still a pattern in the error.



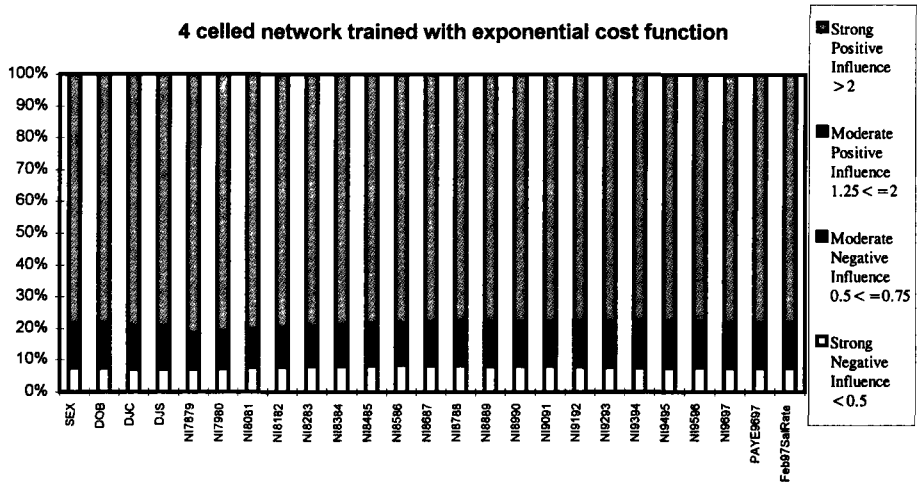
3.21 Exponential cost function

Thankfully, the exponentially weighted cost function gives a fairly random error distribution (shown below). Encouragingly, the network trained on this measure also has a higher IQ measure with a beautifully consistent use of all the input items, illustrated in the chart on the next page.

However, there are still some unacceptably large errors. To proceed, we needed to overcome the problems we had previously when we increased the number of cells (section 3.15).



3.21 (continued)



3.22 Weight decay

As we noticed in the work on mortality, a model produced from s-shaped functions will be bumpy if the weights are large. This will mean that the model will not interpolate and generalise well. It also brings the associated problems of a lumpy or ridged error surface with local minima. The traditional solution is known as weight decay and is used on the weights in both layers of synapses.

Interestingly, there is a physiological equivalent of regulating a neural network's weights, known as sleep. During sleep, the brain does a bit of housekeeping to try and keep the full range of synapses in good working order. Using random electrical impulses it prunes out paths that lead nowhere and adds more links so that the network is not overly reliant on any particular path. If one path swamps others the train of thought may become too narrow-minded or obsessive and not receptive to learning. Any awareness we have of this activity will be a dream, which is why dreams tend to be closely related to what we've been thinking about recently - or related to nothing at all.

3.23 Weight decay formula

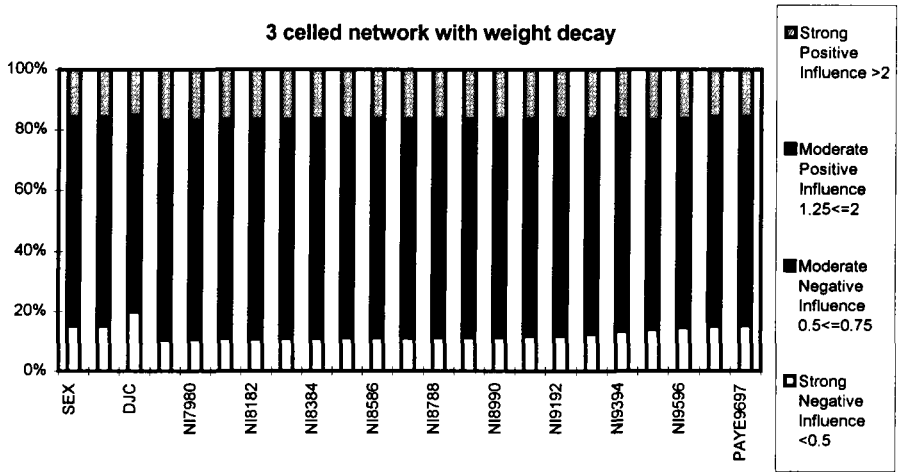
When the network is training, weight decay is achieved by knocking a bit off the weight a_t every time it is updated by the backpropagation algorithm (after each epoch) [28]: $a_t = a_{t-1} - \delta a_{t-1} +$ update

This will cause the weights to decay exponentially, having the greatest effect on the largest weights. Traditionally, weight decay would be achieved by using a cost function which includes not only the sum squared error, but a penalty for large weights, eg based on: $\frac{1}{2}(output - target)^2 + \frac{1}{2}\delta w^2$ where w is the sum of all the weights in the network.

Differentiating this cost function with respect to the individual weights gives the formula above for updating the weights - where "update" is the update using the original cost function alone. However, in MATLAB®, when we are using our adapted error functions, it is necessary to have the weight decay as an explicit part of the formula for changing the weights, rather than including it in the cost function, so it is helpful to express it as in the first equation.

3 23 (continued) We observed that when using a constant decay parameter (δ) which reduced all the hidden layer's weights by the same proportion, the network was able to compensate for this by scaling up the weights in the response layer. To improve the processing power of the network, we extended weight decay to include a random adjustment to each of the weights. This forces the network to combine the weights in the hidden layer in a more imaginative way.

3.24 Results with weight decay We tried using weight decay with a three-celled network (with the inputs and outputs both simplified as described previously). We also returned to the sum squared cost function so that we could compare the results with those obtained in our first three-celled neural network. From the IQ chart below we can see that this network is no genius, even though we have made the inputs and outputs easier.



This is because we have added constraints to the weights. The cells are forced to share the work with other cells. Effectively there are less parameters to work with. As a result, we need more cells with more independent parameters to get the same performance as a neural network which hasn't been trained with weight decay. And more cells will mean that the training time gets rather long again, so we need ways of cutting down the training time.

3.25 Bootstrapping One way of reducing the training time is to use less data (than the full set of over 32,000 cases). However, we have to be sure that we are using enough data to represent the problem. As this depends on the complexity of the problem, there are no rules of thumb. We decided to use an approach known as bootstrapping [29]:

- Train the neural network with sample of N data sets taken (with replacement) from whole population
- Compare the errors with those for a test set of data using an identical cost function (our test set had 1,000 data sets)

3.26 Stopped training

To find the minimum training set size N we drew on some of the theory of "stopped training". This aims to avoid overfitting by stopping the training when the model seems to be getting too complex [30]. The model is considered to be too complex when the error in the test set increases over an epoch. This indicates that the neural network has started to model the noise in the training set, rather than the true pattern.

For this problem we know that there is no noise because our formula is accurate. If the error in the training set decreases and the error in the test set increases as training progresses, the model is only capable of fitting the problem for the data in the training set. This means that the training set is not large enough for the neural network to produce a model which represents the whole problem.

From this we found that we needed 10,000 cases in the sample set to get a reasonable generalisation ability. This also told us the training time we needed to use for that sample size. Although the training time was lower than that indicated in section 3.14, it was worth looking at other ways of simplifying the problem.

3.27 Linearities

The next thing we tried was to perform a linear regression on the problem to extract the linear relationships between the inputs and the outputs. The idea here was to use the neural network to model only the non-linearities in the problem. This looked promising as the linear part of the problem accounted for 84% of the relationship between the inputs and outputs.

We used a 20-celled network and a sample of 10,000 data sets to model the non-linearities. We used the stopped training technique to check that our model was not overfitting the problem. We initialised the weights, calculated the error on the training set, adjusted the weights and recalculated the error. When the test set error increased, we moved on to another sample to continue the learning.

This approach was unsuccessful: each time the network adjusted the weights from their initial values the error in the test set increased so the network moved on to another sample. The learning started again, but stopped after one step [31].

Material released this July on the Internet [32] suggested that stopped training will only work for problems with small curvature. Our experience endorses this view. In the early stages of learning, there is likely to be significant variation in the performance of the model on the test set. That variation alone may trigger the halt of the training. Stopped training works fine on linear problems and those with small curvature, but not on high curvature ones. Here we stripped out the linear relationships from our problem, so stopped training was no good.

3.28 If at first ...

Having relied on stopped training to pick the sample size with the linear model, our main worry now was that 10,000 may not be a reasonable sample size to model purely the non-linearities. With a 20-celled network and a sample size of 10,000 the training was still taking over 100 hours.

The next idea was to use a different backpropagation algorithm.

3.28 (continued) For the previous models we had used the gradient descent method. Instead we decided to use the Levenberg-Marquardt method, which is a hybrid method that uses second order differences in a similar way to the Newton Raphson method for finding the minimum of a function. On messier parts of the error surface the Levenberg-Marquardt method uses the more robust approach of gradient descent, switching to the more accurate Newton Raphson approach to speed things along when the error surface is clearer.

The downside of using this approach is that it is very hungry on memory: it needs the eigenvectors and eigenvalues of the error surface for every input data set. Using all 23 inputs to the model, the most we could sample was 1,000 data sets which we knew wouldn't be enough to represent the problem. So we used four samples of 1,000 data sets, calculated the weight update for each using the Levenberg-Marquardt method, and then averaged these to calculate the weight update [25, 33]. This gave speedier results than gradient descent alone, which averages the weight update across the whole training set and tends to be slow, even if reasonably sure.

But the errors were still not low enough and the training times too long.

3.29 Dimensionality We then decided to draw on our knowledge to speed up the process. We knew that by using the pre 1988 and post 1988 Guaranteed Minimum Pensions (GMPs) instead of the raw NI data we should be able to produce the same model, but with only six inputs instead of 23. The memory this released allowed us to use larger samples.

We were unsure about the sample size we needed to use to model the problem with the linearities stripped out so we decided to put the linearities back in again. This had the added advantage of meaning that we didn't need to transform the data, avoiding complications in measuring the neural network's error [34, 35].

3.30 Design summary We used the following design:

- 40 cells in the hidden layer
- Four training sets of 4,000 each
- Levenberg-Marquardt to provide four weight updates after each epoch from the four training sets, then averaging these to get the overall weight update
- Cost function based on sum of squared errors
- Weight updates including weight decay and a cap on the weights of 3 in both the hidden layer and the output layer

3.31 ... and trying again

With this network, the errors on the test set still showed a pattern, but the sum squared error over the 1,000 test cases was 3.6 which was significantly lower than that for the other neural networks produced earlier in the training. We went on to try 70 cells to see if the error reached the right level. But, even with 100 cells, the sum squared error did not reduce below 3.5. The table below illustrates the distribution of the absolute errors: this network is "Net1" and only 12% of the errors were within our 1% target.

We then set up a network with a smaller number of cells, but using the outputs from the previous 70-celled net as two additional inputs. The rationale for doing this was to use the smaller network to improve on the performance of the larger network, rather than starting from scratch. This improved the performance significantly, but only 25% of errors were within the 1% target (shown in the column labelled Net2 in the table below).

At this stage we tried two further modifications:

- transforming the inputs so that they fell in the range (-1,1)
- transforming the outputs so that they had a shape closer to the function used in the response layer, ie linear (by taking logarithms)

The resulting network (Net 3) was an improvement, with 34% of the errors being within our 1% target. But 7% of the errors were still greater than 5%.

Network design Error tolerance	Net1 %	Net2 %	Net3 %
<1%	12	25	34
<3%	34	66	81
<5%	49	83	93
<10%	70	92	98
<15%	81	95	99
>50%	2	0.009	0

3.32 SafetyNet

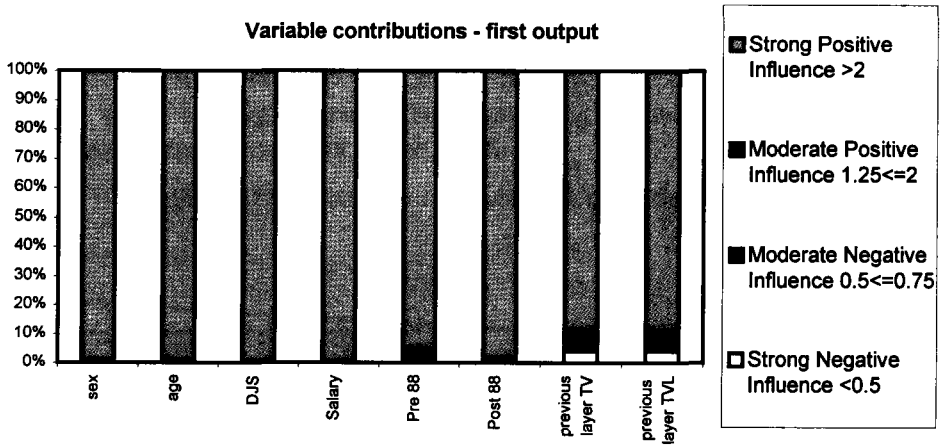
To reduce the number of large errors, we developed the idea of using a chain of networks where the aim of adding each extra neural network was to reduce the error produced by the previous network in the chain. Instead of using sigmoid functions throughout each network's hidden layer, we introduced two cells which had linear activation functions. These two cells allowed the two outputs from the previous network to go straight through the network.

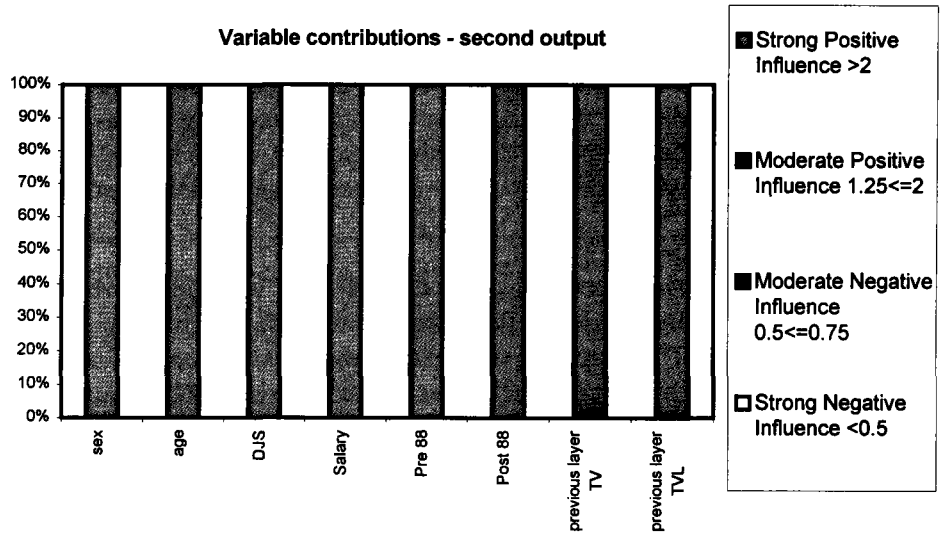
The final network which we will describe in this paper was made up of four neural networks feeding into each other in a sequence. The first three neural networks each had ten cells; the fourth neural network 20 cells. We used the same transformed inputs and outputs described in section 3.31 for Net3. However, we also located a low memory version of the Levenberg-Marquardt algorithm on the worldwide web which meant that we could go back to using the full training set of 32,000.

3.32 (continued) The absolute error improved by adding each extra network until the fourth network: adding a fifth network only improved the error marginaliy. We felt that the only way to improve performance further was by focusing on the training data used.

3.33 IQ The good news was that the final network was nothing short of a genius when measured by its IQ (see section 3.17 for the formula).

As illustrated in the charts below, the overall IQ for the first output is 98.4% and the overall IQ for the second output is very close to 100%. This implies that the network is making full use of all the data that is being presented to it. Our transformations of the inputs and outputs have helped performance significantly.

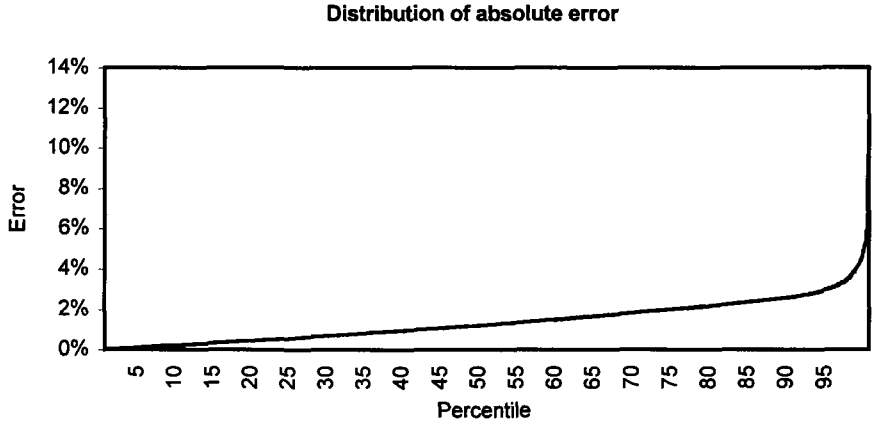




3.34 Errors

The bad news was that this network was not a significant improvement on Net3 when looking at the percentage of the absolute errors which fell below our 1% target. However, it had reduced the number of large errors with over 99% of absolute errors being under 5%.

The graph on the next page illustrates the distribution of the absolute errors. The errors have been sorted and plotted as percentiles of the data in the test set. The smooth progression shows consistent treatment of the inputs up to a limit where the neural network's experience is insufficient in producing a suitable output for a few exceptional cases. This would suggest looking at the data for the troublesome calculations more closely and providing the network with more of these difficult examples as part of its training set.



For many applications this error distribution would be acceptable. Unfortunately it does not pass our test. In order to achieve less than 1% absolute error for 100% of cases, the neural network would benefit from more data. Generating data systematically rather than randomly across the range of input values might help improve the errors across the whole range, but particularly at the input limits where the network is producing some exceptional errors.

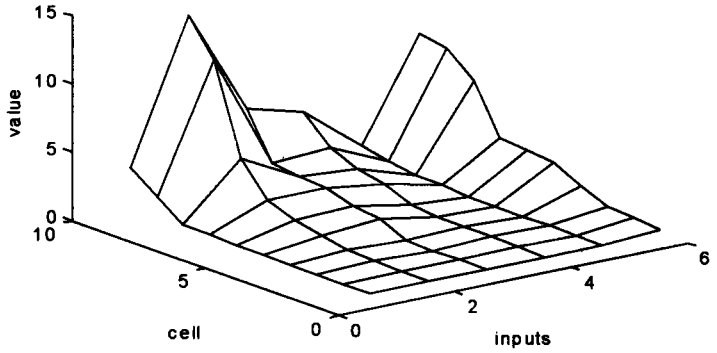
3.35 Weights

Are there any other changes that we could have made to improve performance? By looking at the distribution of the weights, we can spot any learning difficulties. The graph on the next page shows the weights used in the activation functions in the first network layer, sorted by size for each input.

This distribution is not ideal because the shape is quite uneven for the second and sixth inputs (age and post 1988 GMP). This means that the cells are not sharing the calculation work evenly. By applying weight capping this could have been avoided. However, there is a good spread of values, reflecting the weight decay and capping used to regulate the weights in training.

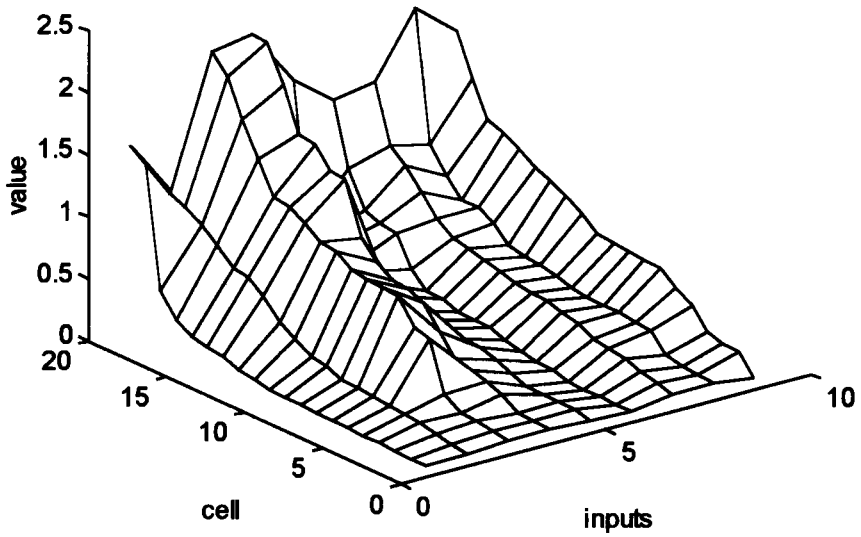
3.35 (continued)

Weight distribution - first network's hidden layer



In the fourth neural network in the chain, the distribution of weights is much improved. (This has eight inputs: the original six inputs and the two outputs from the previous neural network in the chain.)

Weight distribution - fourth network's hidden layer



3.36 Conclusions

We have found a suitable methodology to train this design of neural network on continuous actuarial functions. Although our neural network was reliable to within a 5% error tolerance, 32,000 random data sets do not represent the problem well enough to produce errors consistently lower than 1%.

Between us, we had a reasonable knowledge of statistics and neural network techniques but still had difficulties building a neural network which got close to solving the problem. It seems reasonable to conclude that someone using basic neural network software would not have been able to come up with a better neural network solution without:

- knowledge of the problem
- an understanding of the underlying mathematics
- perseverance and the ability to think laterally

One of the obstacles to progress is the limit of computer power. But it will only be a matter of time before the barriers of processing speed and memory capacity will fade.

In the meantime, the actuarial profession should continue to practise and develop neural network modelling skills. In Appendix C we have documented the process that we should have followed to produce our model in the most efficient way. Together with our model's design, this can form the starting point for actuaries addressing similar modelling problems in the future.

4. Generalised Linear Modelling

"General notions are generally wrong."

Lady Mary Wortley Montagu

4.1 Introductions

Generalised Linear Modelling (GLM) [36] is a modelling technique which has been used by actuaries for many years, particularly for general insurance claims modelling and reserving. GLM involves a significant amount of time and persistence in analysing the raw data items and inferring relationships between the available data and the desired target function. To produce a reliable model the modeller must:

1. Find the most promising inputs.
2. Transform the inputs either alone or together so that the relationship between the transformed inputs and the target function is likely to be linear.
3. Express the relationship using parameters and use a sum squared error measure used to optimise the parameters.

This section of the paper compares the structure of neural network and GLM models and suggests how GLM modellers might be able to improve their models by drawing on ideas from neural network techniques.

4.2 Link functions

GLM relies on being able to transform the inputs and target function (using what is usually known as a "link function") so that there is a linear relationship between the transformed data. The link functions are chosen by the modeller - but they should be differentiable, monotonic and from the exponential family of functions.

For example, if we were modelling a function which had an exponential shape, the link function would be the natural logarithm:

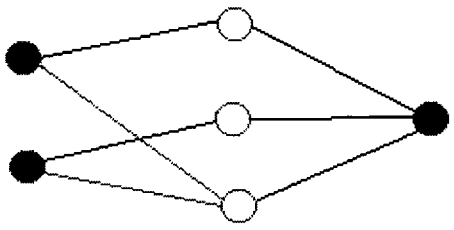
if $y = e^{(ax+b)}$ $log_e(y)$ is a linear function of the input x.

GLM then uses a sum squared error measure and linear regression to optimise the parameters a and b.

4.3 Layers

We can translate a GLM model into a network which has the structure [29]:

- inputs in the sensory layer
- limited connections between the sensory and association cells
- link functions in the association layer
- linear functions in the response layer



sensory cells → association cells → response cells

4.4 Formula

Expressing this as a formula:

x_i is the input transmitted by the i^{th} sensory cell

a_{ij} is the weight used to multiply x_i before it reaches the j^{th} association cell. The weights are fixed at either 1 or 0. In the simplest GLM models, there are the same number of association cells as sensory cells and $a_{ij}=1$ when $i=j$ and 0 otherwise.

In more complex GLM models, additional association cells use link functions of the combination of two or more of the inputs. The diagram in section 4.3 illustrates the cells and links in a model with two inputs and one output: the activation functions in the first two association cells are functions only of single inputs; the third cell is an activation function of both of the inputs.

$u_j = x_j$ in a simple GLM model - in more complex models it will be a sum of combinations of the inputs

f_j is the link function appropriate for the j^{th} input (or combination of inputs) - used by the corresponding association cell, so that its output is

$$y_j = f_j(u_j)$$

b_{jk} is the weight used to multiply y_j before it reaches the k^{th} response cell

b_k is the bias added to sum of the weighted outputs from the association cells, giving an input to the k^{th} cell in the response layer of

$$v_k = \sum_j y_j b_{jk} + b_k$$

g_k to continue the comparison, the formula used in the k^{th} response cell is the identity function so that the final output from the network is

$$z_k = v_k$$

For a simple GLM model, the overall function gives the output from the k^{th} response cell as a function of the inputs x_i

$$z_k = \sum_j \left[f_j(x_j)\, b_{jk} \right] + b_k$$

4.5 Simplicity

The simplicity of this formula explains why the testing and optimisation process is less involved than the equivalent in a neural network. There is only one layer of parameters and the model is linear, so a sum squared error (the cost function used in linear regression) will give a unique solution for:

- a given set of inputs (and combinations of those inputs)
- a given set of link functions f_j

These are chosen by the modeller. But how does the modeller know that these are correct - or good enough to model the problem?

4.6 Start simply

The usual GLM approach is to start simply and add inputs one by one, testing at each stage whether each has a significant effect (using what is known as scaled deviance). This keeps the number of parameters to a minimum. However, the order in which you add the different inputs may determine the choice of inputs.

There is also the risk of discarding an input in a simple model which may become significant if the model is extended to a more complex model where the interrelationships between the inputs become important. For rigour, every possible combination of inputs and their interactions should be tested.

4.7 Links

More importantly, the analytical ability of the modeller is of critical importance. Picking the right transformation functions from the library of those available will make the difference between a model that is accurate only over a particular range or one which can extrapolate and generalise well.

In practice, picking the link functions involves plotting inputs and outputs and trying out a variety of functions on a computer (or on paper) to see which look most promising. A neural network fanatic would find this rather quaint: it is not really making the most of the available computer power.

What can a GLM modeller learn from neural network procedures and how can a neural network be used to extend a GLM model?

4.8 Improved procedure

The same tests and procedures used for neural networks can equally be applied to GLM models. These involve not using the whole data set for building the model but instead reserving some of the data for testing the model. This is useful in making sure that the GLM model will be able to cope well with situations beyond those included in the initial data set, ie the model will be able to generalise.

4.9 Improved model

Using the structure in section 4.3, it is possible to translate a GLM model into a neural network model. We can then go on to train this neural network and see whether the structure changes significantly during training - if it does, we probably need to re-examine the structure in the original GLM model. Even if the structure remains the same, if the neural network model is performing significantly better, this may indicate that the problem is better suited to a non-linear model.

4.10 Combined models

We could also ask a neural network to model the errors between the GLM model and the desired output. This error network should be retrained as more data becomes available. Significant changes in the neural network will then show that the GLM model is getting out of date. Using the inputs and outputs from the GLM model as inputs for a stand-alone neural network will also invariably reduce the training time.

4.11 But don't forget

Non-linear models have their weaknesses. The suggestions we have made above aim to improve GLM models, not to replace them. When you use neural network modelling, you should still follow the process outlined in Appendix C to come up with a complete and rigorous solution.

5. What next?

"Knowledge advances by steps, and not by leaps."

Thomas Babington

5.1 Tip of the iceberg

In this paper we have provided only a glimpse of the potential of neural networks. This final section gives a few examples of other uses of neural networks. It also suggests what the actuarial profession might do next.

5.2 Good for something

In the examples in this paper we have illustrated how, subject to the practical considerations of time and computer memory, neural networks can be used to model problems where:

- the laws and relationships between the inputs and outputs are not fully known or understood
- the data is noisy, sparse or unwieldy
- there is a need to simplify the calculations or relationships
- existing models are no longer sufficiently accurate

5.3 ... and something else

There are two other main areas where neural networks have been used:

1. Predicting the future.

   The networks we have used in this paper are all "feedforward" networks where the information flows from the inputs through to the outputs. For modelling time series, "recurrent" networks are more promising. In these, the outputs from previous calculations are fed back in as inputs to the next calculations to set the model off into the future.

2. Analysing data into homogenous classes.

   For "classification" problems networks are given a number of bell-shaped "radial basis" functions instead of sigmoid functions. The modelling process fits these functions over the items of information we have for each data item, to group them together. By normalising the functions and applying Bayesian probability theory, the trained neural network can be used to identify which class a new item of data fits into - and the probability that the network is right.

Both of these types of application have an appeal relevant to actuarial work. Predicting the future is relevant for setting assumptions and investment strategies; analysing data is relevant for insurance premium rating or selecting investment managers. Predicting the contributions needed to meet the MFR liabilities for a particular pension scheme might involve elements of both types of application.

5.4 Making money

By way of a simple example, we were intrigued by an article which appeared in the Financial Times earlier this year which looked at three investment strategies:

1. Invest entirely in US treasury bills: $1 invested in 1926 would grow to $14 in 1996.
2. Invest entirely in the US stock market: $1 invested in 1926 would grow to $1,400 in 1996.
3. Invest each month in the better of the two (with perfect clairvoyance): $1 invested in 1926 would grow to $2,300,000,000 in 1996.

Students of compound interest can explain the differences in terms of annual rates of return over a long period, but the value of perfect clairvoyance ($2,299,998,600) is apparent.

Can neural networks spot patterns to help make better investment decisions?

5.5 Making more money

We looked at a similar problem - deciding at the start of each month whether to invest in the FT-SE All Share Index or put our pennies under the mattress. We used a simple one-celled network with a hard limit neuron - one which outputs either one or zero, to correspond to "invest" or "don't invest".

We assumed that the only data we would have available to us would be the previous one, three and twelve-month returns on the index. We used patterns of returns over five years to pick promising models and then tested them over a fourteen-year period.

The networks which gave returns better than the All Share Index in training all performed better than the index in testing. The most promising network from the results in training had a strong tendency to invest (92% of the time) over the fourteen-year period. The best results would have been obtained if it had invested just 65% of the time. However, the losses that it avoided when it didn't invest were over twice as large as the gains that it missed out on, so it seemed to have picked out a useful pattern.

It is easy to see how a more sophisticated neural network model might be able to choose its investments more accurately.

5.6 Existing research

To come up with better models, we do not need to start with a blank sheet of paper. There has already been a considerable amount of research on the subject of financial modelling and we can build on this by adding our own knowledge and a rigorous approach.

The collection of examples in "Neural Network Solutions for Trading in Financial Markets" [5] builds on the introduction to neural networks given in this paper. The authors take an objective look at various possible applications, comparing them to other models and theories and examining whether neural networks overcome the weaknesses of existing models.

5.6 (continued)

To whet your appetite, the examples include using networks to:

- detect non-linear relationships in time series of financial data to make robust predictions
- predict tax receipts
- derive a profitable trading strategy from derivative market information
- perform technical analysis of individual stocks and beat other stock market prediction models
- provide a portfolio with a higher return and a lower risk than those used by professional fund managers when allocating funds to five major international markets
- combine quantitative and qualitative inputs to assess credit risk and predict corporate failure

In Appendix A we give details of this book, and others which we recommend as further reading.

5.7 Software

To come up with better models, we do not need to start with a blank spreadsheet.

We used the neural network toolbox in MATLAB® for the work on the MFR problem in section 3. This software is reasonably versatile. The demonstration programs can be edited to create your own models and are useful to help explain how the backpropagation algorithms work. (The graphics facilities are superb, too.)

We have also been offered a trial of AcuSTAR® which uses classification networks to help categorise data. This is sold to retailers and other organisations as a "data mining" tool: to use customer data to make decisions on pricing, marketing and product placement. We plan to try this out on pension scheme, insurance and investment data in the near future.

Scientific Computers also have a NeuralWare product range which contains some of the most up-to-date and user-friendly packages that we have seen. Their NeuralWorks Predict™ runs off Excel and covers both classification and prediction networks.

5.8 Government interest

In 1993 the DTI carried out research into companies who were using neural networks [37]. As a result, they drew up some guidelines on neural network design and a directory of services, products and commercial suppliers using neural computing. Although much of the information is now rather out of date, the conclusions of the survey remain interesting:

- Neural networks were being used most in industry, in applications related to control systems and quality control
- Retailers were the next largest users, using neural networks to analyse consumer patterns and target direct mail
- The financial sector was in third place, with applications including fraud detection, economic modelling and financial predictions

5.8 (continued)

- Most projects were brought about by hiring a consultant or employing new graduates with neural network knowledge.
- On average companies spent £0.5m developing their first neural network: after establishing that initial knowledge base, the costs went down dramatically
- The biggest problem companies faced in developing reliable neural networks was a lack of data with which to train their models

The government uses neural networks in military applications. The police also use neural networks for doing intelligent searches of criminal databases.

5.9 Networking actuaries

Do you know the joke about an actuary being a computer, but without the personality?

If the actuarial profession does not grasp neural network techniques and add them to our toolbox, then that joke may get too close to home too quickly. Will insurance companies and pension funds need actuaries to assess and manage risks in the future if they have a computer and enough data?

Traders and fund managers are already feeling the pressure: In 1993, the Economist reported on John Deere (an American manufacturer of farming equipment) [8]. The company had handed over $100m (about 10%) of its pension fund assets to one of its employees, an engineer with a new PhD and a computer. With only his neural network model and market data to help him, he buys and sells shares: he does not believe in tinkering with what the model tells him to do. When the report was written, he had already been investing for a few years and had not been tempted back to the world of combine harvesters.

The profession's future will be secure if we combine our knowledge of the laws of economics and statistics with a rigorous professional approach to using neural networks.

5.10 Professional conduct

To develop neural networks it has taken a combination of biology, physiology, psychology, computer science, physics, mathematics and statistics. To apply them all you need is statistics. Actuaries are perfectly placed to embrace neural network techniques and use them to create sophisticated and reliable financial models which meet our exacting demands.

University courses teach neural network techniques, but they needs professional backing to promote neural networks as a practical scientific tool for financial modelling in the business world. More importantly, the actuarial profession needs to build up a knowledge base. The most immediate way of doing this is to make sure that the techniques are acknowledged in the course of reading for the professional exams. The profession has a lot to gain by taking larger steps and sponsoring further research. And actuaries are beginning to take neural networks more seriously [38].

We believe that the further reading we have recommended will add to the professional development of both student and qualified actuaries. A deeper knowledge of the techniques will help them to question the use of neural networks objectively and come up with better ways of creating certainty out of uncertainty in the future.

5.11 Parting thoughts

To conclude, our personal thoughts (limited to 100 words each) on neural networks:

"Life isn't linear: every London road has its potholes. The power of metaphors illustrates the additional understanding that our brains glean by drawing on apparently unrelated information to understand new concepts.

Neural networks translate the cerebral power of parallel processing, of recognising patterns and similarities, into a mathematical model. The models are limited only by the data available to them, the skills of the modeller responsible for educating the network and computer power.

Actuaries can gather data and learn the skills. Will computers based on traditional logic be able to support the neural networks of the future? I hope so."

"This report achieves its goal of benchmarking, and popularisation.

However, there remain practical hurdles to be overcome, and I hope that a grasp of this paper's experience will save time for future statistical modellers.

The future will be best served by more actuaries presenting papers and popularising information, this is the challenge for the profession, and I hope that enough people take it up for it to be self sustaining.

Probability modelling, and Bayesian techniques are likely to be the areas that will touch us all, so here is the ocean, wade in and climb aboard for a nonlinear ride."

"A neural network is a powerful tool. So is a chainsaw. But it's dangerous and inappropriate to cut your toenails with it. If you can get a good answer by a simpler method then do so. But if you need a neural network then use one. Wisely. With caution. Be patient. Give it lots of data. Test it. Refine it. Try again. And again. And again. It can work wonders. But it's not intelligent. It's not a precise science. It's not an alternative to human thought but a complement. A tough problem is tough to solve however you do it."

Thanks

Thanks to everyone who helped us develop and write this paper. To Richard Chapman, Nick Fitzpatrick and Alan Phillips for supporting our work. To Jeremy Farr, Andrew Inchley, Kevin Skinner, Andrew Smith and Shirleen Stibbe who listened to our thoughts, challenged our early ideas and shaped the approach we used. To Steve Smallwood and Anne Freeman for providing data. To Raj Mody for incisive comments on drafts of the paper. To Professor Keith Rennolls for guidance. To Angela at the Bailiff's Chambers in Guernsey for the snippets of magic. And to Wendy and Mike for their patience.

We remain entirely responsible for any errors.

Appendix A Further reading

"Read not to contradict and confute, nor to believe and take for granted, nor to find talk and discourse, but to weigh and consider."

Francis Bacon

About neural networks

G E Hinton [1992]: "How Neural Networks Learn from Experience" Scientific American 267.

Murray Smith [1993]: "Neural Networks for Statistical Modelling" International Thompson Computer Press.

Christopher M Bishop [1995]: "Neural Networks for Pattern Recognition" Oxford University Press.

B D Ripley [1996]: "Pattern Recognition and Neural Networks" Cambridge University Press.

Martin T Hagan, Howard B Demuth and Mark Beale [1996]: "Neural Network Design" PWS Publishing Company.

E Gately [1996]: "Neural Networks for Financial Forecasting" John Wiley and Sons, Inc.

D E Rumelhart and J L McClelland [1986]: "Parallel Distributed Processing: Explorations in the Microstructure of Cognition" (volumes 1 & 2) The MIT Press.

David E Rumelhart, Geoffrey E Hinton and Ronald J Williams [1986]: "Learning representations by back-propagating errors" Nature 323.

Dirk-Emma Baestaens, Willem Max Van den Bergh and Douglas Wood [1994] "Neural Network Solutions for Trading in Financial Markets" Financial Times/Pitman.

The Economist [9 October 1993]: "The Frontiers of Finance" survey now published as a chapter in "Going Digital" [1996] The Economist/Profile Books.

Hubert L Dreyfus "What computers still can't do: a critique of artificial reason".

Other books

Scott Adams [1996]: "The Dilbert Principle" Boxtree Limited.

Jean-Dominique Bauby [1997]: "The Diving-Bell and the Butterfly" Fourth Estate.

Bill Bryson [1995]: "Notes from a Small Island" Black Swan.

Albert Camus [1942]: "The Outsider" Penguin.

Roger Hargreaves [1976]: "Mr Impossible" Thurman Publishing.

Herge [1968]: "The Adventures of Tintin 'Destination Moon'" Mammoth.

G B Edwards [1982]: "The book of Ebenezer le Page" Penguin.

Ricardo Semler [1993]: " Maverick!" Century.

Clifford Stoll [1995]: "Silicon Snake Oil" Macmillan.

Cyberspace

Dr Leslie Smith's introduction to neural networks, with examples and diagrams, at http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html.

Donald Tveter's "Backpropagator's Review" contains answers to FAQs and an annotated neural net bibliography at http://www.mcs.com/~drt/bprefs.html.

DACS Technical Report Summary: Artificial Neural Networks Technology at http://www.utica.kaman.com/techs/neural/neural.html.

Usenet groups comp.ai.neural-nets and comp.theory.self-org-sys include FAQs which give a comprehensive list of reading material, journals, conferences and web sites.

To experiment with neural networks you can draw on data provided on web sites, for example:

- The neural-bench Benchmark collection at http://www.boltz.cs.cmu.edu/. The data sets include nettalk, two spirals, protein structure prediction, vowel recognition, sonar signal classification and a few others
- Lloyd Lubet's Financial, Business, and Economic Data Warehouse at http://www.rt66.com/~llubet. The database consists of 23 tables containing 896 monthly indicators, indices and statistics for over 21 years. 7 tables deal directly with historic stock prices and market indices

Appendix B Learning algorithms

"Who with a natural instinct to discern what knowledge can perform, is diligent to learn."

William Wordsworth

Aim

"Learning" is the process by which the parameters in a neural network are changed so that the network's performance improves. The performance is measured by a cost or error function.

For a simple problem, such as fitting a straight line to some observed data points and testing the fit with a sum squared error function, this is straightforward. The mathematics is tractable and an exact solution can be obtained algebraically.

For a neural network of any complexity this is not possible. A small change in any one of the parameters (weights and biases) can have a considerable impact on the overall performance of the neural network. There may also be difficulties if the cost function is complicated.

The only way to fit the parameters is by using an iterative process. Rather than performing some algebra and then finding the best answer we are left with the process of choosing initial values and then improving.

Practice

By way of a simple example, consider a model with a single parameter x; we want to minimise C(x), where C is the cost function. We need to specify a_0 and f so that if $a_{n+1}=f(a_n)$ then the sequence (a_n) has a limit and so that the limiting value of the sequence (a) gives a locally minimum value of the cost function C.

So, what do we choose for f?

If we are looking for a minimum it makes sense to consider the derivative of C. If $C'(a_n)$ is negative then a small increase in a_n will decrease C as required. Similarly if $C'(a_n)$ is positive then a small decrease in a_n will decrease C. And if $C'(a_n)$ is zero then we are at an extremum.

A reasonable choice for f would therefore be $f(a_n)=a_n - \alpha\, C'(a_n)$ where α is a small positive number. This approach is useful as long as α is well chosen. It is also important to pick the right initial value for a_0.

Example

For example, if our cost function is $C(x)=4x^2-9x+11$ we can solve algebraically to find that $a=1.125$.

Using the algorithm with $a_0=1$:

- For $\alpha=0$, $a_n=1$ for all n - no convergence
- For $\alpha=0.0001$, progress is slow ($a_{500}=1.12275$) but it does converge eventually
- For $\alpha=0.1$, convergence is quicker and $a_{10}=1.125000$
- For $\alpha=0.2$, $a_{10}=1.125000$ again
- For $\alpha=0.5$, the sequence diverges

Example (continued)

It is possible to analyse the process and see that:

for $0<\alpha<0.125$ convergence is achieved and $a_n<1.125$ for all n

for $\alpha=0.125$ convergence is achieved after just one step

for $0.125<\alpha<0.25$ convergence is achieved and a_n is alternately higher and lower than 1.125

for $\alpha=0.25$ a_n oscillates between the two values of 1 and 1.25

for $\alpha>0.25$ the sequence diverges

Problems

This simple case illustrates the difficulty in finding a suitable value of α. Too high and convergence is not achieved; too low and progress is slow. Here it was possible to analyse the problem algebraically but in general we have little choice but to try some likely values and see which work well.

More sophisticated approaches use a variable α rather than just a fixed value. By changing α according to some features of the cost function the algorithm can make quick progress in certain areas and slower steadier improvements near the minimum.

Implementation

The neural network architecture offers some advantages in implementing this algorithm. Although the overall functions dealt with are extremely convoluted, the simplicity of the individual activation functions helps.

To use the backpropagation algorithm we need the derivatives of the cost function with respect to all of the weights and biases. For the weights in the final layer this is easy to calculate. For those in earlier layers we are able to apply the chain rule for partial derivatives. This makes it possible to calculate the derivatives in each layer easily from those in the following layer. This explains the origin of the name backpropagation: the changes required in the parameters in the final layer are calculated first and these are then used to calculate the changes in earlier layers.

Further Problems

For a cost function with two or more minima there is a danger of finding a local minimum that has an error considerably greater than that at the global minimum. One way to avoid this problem is to operate the algorithm many times with widely differing starting values for a_0.

Alternatively we can introduce "momentum". The idea is to prevent the algorithm from settling too quickly in a minimum. By giving it a nudge it will be disturbed from its equilibrium and may settle in a different, lower, minimum.

Other algorithms

There are many useful algorithms, nearly all of which are derived from the idea of moving in the opposite direction to the gradient. The ongoing research in this area aims to reach a minimum efficiently but without becoming stuck in a poor local minimum on the error surface [29].

An alternative plan of attack is to make sure that the error surface is not too bumpy to begin with. By forcing the network to use many small weights rather than a few large ones, the error surface is made smoother and the algorithms are able to work more efficiently.

"It is only those who do nothing that make no mistakes, I suppose."

Joseph Conrad

Define the problem

The neural network is a formula which approximates or models a problem. We need to have a clear definition of what we are trying to achieve before we gather the data we need and design and test our model.

Purpose

What is the neural network's mathematical model designed to do?

We can:

- approximate a known formula
- model an unknown formula
- derive new relationships
- make a prediction

Success

How will we define whether or not the neural network is doing the job?

We need to measure success, ie to define whether or not our network is accurate enough and has solved our problem. We should set this test in advance so that it depends only on the problem, not on the data we have used or the design of the neural network.

Gather information

The data we have available to train the network affects the neural network's design and the training process.

Number of inputs

Are the inputs highly correlated?

Highly correlated inputs make the model more sensitive to the peculiarities of the particular sample. If two inputs are correlated the network spends time learning about them both when it really only needs to spend time learning about how the outputs differ when the two inputs differ. It will mean that the network is less able to generalise. It may be better to leave out one of the highly correlated inputs.

Representing the outputs

How do we represent the outputs?

Representing the outputs as a proportion as one of the inputs tends to be a good strategy as opposed to using absolute values.

Ideally, the shape of our output function should also resemble the shape of the activation functions in the response layer.

Number of data sets

How much data do we have available from which the network can learn?

The amount of data will have implications for the number of parameters that we can optimise through backpropagation, and for the number of cells that we can realistically have in the neural network.

Range of data sets

Is the data evenly spread about the possible range of input values or will it have some characteristics which depend on the source of the data?

If the data is randomly and evenly spread over the range of input values the network will learn the pure relationship between inputs and outputs. This will mean that it is better at working out the corresponding output for any input, ie it will be better at generalising.

If the data is taken from a particular source and does not reflect the whole possible range of inputs, the network will also be learning some of the characteristics of the data. It will be producing a statistical model of the likely problem rather than just approximating a function.

What range of the possible input values does the data cover?

The network will only be able to learn the problem within the range covered by the examples it uses to learn. Similarly, if an input can take a thousand possible values, the likely range of possibilities needs to be represented in our training data.

Known relationships

Do we know something about the problem?

If we have tried linear or GLM techniques we can use these to transform the data before presenting it to the network. This approach can be used for transforming either the inputs or the outputs, or both. Helping the network in this way brings two benefits:

- the learning process will be quicker because the network doesn't waste time learning something that we already know
- the network can go on to achieve even greater accuracy and find features in the data that we have missed. By providing the network with the obvious relationships, we give it a chance to find less obvious relationships before it starts getting too clever for its own good and overfitting

However, we must be careful about constraining the network too much. It must still have the flexibility to be able to ignore our initial ideas, in case these were misguided.

Size of inputs

Are all the inputs similar orders of magnitude?

If not, scaling the inputs may reduce the learning time. All the inputs go through the same mathematical formula in the backpropagation process: this includes taking the partial derivatives of the inputs to the hidden cells by the weights which come into the hidden cells. As this partial derivative is the size of the input, the network will automatically focus on the larger sized inputs in the early stages of learning.

Transforming the inputs to make them a similar size and centred on zero will make learning more efficient and effective. Two possible approaches are:

- normalising (deducting the mean and dividing the result by the standard deviation)
- linear transformation over a range centred on zero, eg (-2,2)

Type of data

Are the inputs and outputs a mixture of quantitative and qualitative variables?

Depending on the problem to be solved, it may help the network's learning process to reformat the inputs. This might entail splitting ranges of the inputs into separate inputs or outputs and expressing each possible class input as a separate binary input.

For example, an old man could be represented by:

- (1,0,1,0) for (male, female, old, young)
- a single number in the range 1 to 4 to represent the four possible outcomes

The second representation would require fewer parameters in the network.

Design a network

Knowing the data, the problem and the results of any previous training sessions, we can design the network architecture. This involves setting the following items:

Network type

What sort of network should we use?

Feedforward networks are more established and the theory is more transparent so this design is preferable. Feedback networks (where some of the outputs are fed through as inputs within the network) are widely used for modelling time series.

Number of output cells

How many outputs will we have?

Sometimes it is useful to train the network to produce a range of values, or to produce values for outputs which are dependent. This will give a "committee view" of the likely outputs or an indication of the errors produced by the network. Other times, particularly if different inputs are needed for different outputs, it is better to minimise the number of outputs so that the network is not distracted by the more difficult outputs.

Number of hidden layers

Is there any reason why we will need more than one hidden layer?

If one hidden layer fails to model the problem an extra layer may help.

In some cases two hidden layers can model solutions better and with fewer parameters.

If there are discontinuities in the target function then we will need two hidden layers in the network.

Cell formulae

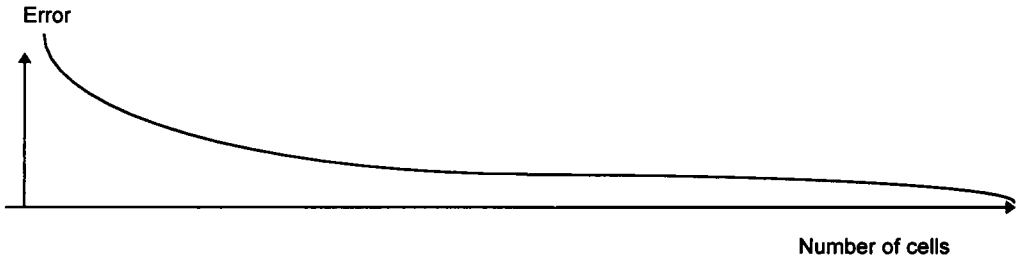
What formulae should we use in the cells?

The inputs are transformed linearly by the weights in the input layer. From the work by Kolmogorov and Cybenko, one hidden layer of sigmoid functions will be enough to model any continuous functions and two layers of sigmoids will model functions which have discontinuities. If the range of the target function is outside the output range of the sigmoid function, usually (0,1), there will need to be another linear layer at the output layer.

Number of cells

How many cells should we use in the hidden layer?

To avoid the possible danger of overfitting the problem, one way to proceed is to start with a small number of cells and add cells until the network is successful. So first we need to decide the minimum number of cells that it is worthwhile trying. This will depend on the complexity of the problem. Are there any similar successful networks which we can use as a benchmark or starting point for the number of cells? If not, start at one cell and increase the number of cells according to how the error seems to be changing - for example, doubling the number of cells each time. A typical error progression is shown below.



We need to check that this number does not exceed the maximum reasonable number of cells. This depends on the amount of data: to train the network we need at least as many data sets as the number of parameters in the network's formulae; and ideally the same amount again for validation and testing. For a network with one hidden layer, and where each cell has one bias, the number of parameters is:

(number of sensory cells + 1) x number of association cells

+ (number of association cells + 1) x number of response cells

Let the network analyse the information and infer relationships

We are now nearly ready to train the network. First we have to make some more decisions. These depend on the network design that we are using and the results of any previous tests.

Data

What data will we use to train the network?

Before we start training we need to decide how to use the available data to train the network, validate our results and test the network's ability. The training set is used to determine the weight updates, the validation set can be used to check that the model is not overfitting and provide more information during training. To test performance we consider a third set. One traditional approach is to split the data randomly into thirds, but this does not make the best use of the data available and quite often the validation set can be used as a test set.

One more advanced technique is to use what is known as a "bootstrap" technique where a sample is taken (with replacement) from the data for training and validation. The size of the sample depends on the curvature of the problem. As a rough guide, our MFR model required 10,000 data sets. After training, the network is tested using the whole population. This technique finds a quick initial minimum for small sample sizes and it makes full use of the data set. Occasionally it may sample more difficult sets than easy ones, which may enable the neural network to work its way out of a local minimum.

Cost function

How should we set the cost function?

Should this be the same as the formula we will use to test the model or are there characteristics of the network design that mean that we need to set additional constraints?

For the first run, we can use a simple cost function (and simple mathematics): usually a sum of squared errors. For subsequent runs, we can look at the size of the previous weights to see whether any of the weights are particularly high. If they are, the output from a sigmoid cell will be quite steep and there is a risk that the overall output from the network might be too bumpy. In these circumstances it is more likely that the network will be overfitting the data. The algorithm may also become stuck in a local minimum where the error surface is too rough. To avoid this problem we can use a different cost function:

- Weight decay. This includes a penalty if the weights get too high: it forces the network to make more use of all of its cells rather than produce a complex mapping function
- Weight capping. This sets a range, for example (-3,3) in which all the weights must lie
- Noisy weights. This adds an element of disturbance to force the weights to move out of a minimum (although this will need to be reduced as training progresses)

Minimisation algorithm

What minimisation algorithm should we use?

If we are using backpropagation, the formulae used to minimise the weights depend on the partial derivatives of the various functions involved. The mathematical formula may need to be reworked to suit our chosen cost function.

Levenberg-Marquardt can be unstable if you alter its partial derivatives: to date, gradient descent has proved more robust.

It is possible to use momentum to try to make sure that the minimisation does not get caught in a local minimum.

Values

What else do we need to specify before training starts?

We need to develop a feel for the appropriate values to use in the algorithms. This can come from previous runs or from a training run with a small sample size to devise a rule of thumb. For example (showing the values used in the MFR problem):

- weight decay (from 0.05 to 0.00005 during a training run)
- weight capping (maximum size of 3)
- noise (from 5% of a weight value to 0.000005% during a training run)
- initial learning rate (0.01 for initial run 0.0000000001 for a re-train)
- error gradient (for adaptive learning rates) (1.05 initially to 1 during a training run)
- sample size (1,000 to 5,000 when using fully-specified Levenberg-Marquardt, 32,000 when using the low-memory version)
- number of epochs per training run (100)

Initial weights

How should we initialise the weights?

Normally we would pick the starting points for the weights and biases randomly, but we may have a feel for a useful starting points (either from linear regression or from previous training runs). If we are picking the weights randomly then we need to make sure that the sloped part of the sigmoid function covers the possible range of inputs. Without this procedure there is a risk that the backpropagation routine gets stuck on the flat part at the extremes of the sigmoid curve and doesn't minimise the weights very quickly.

If our initial choice of weights was successful on the previous training run, we should choose different weights to check that we were not caught in a local minimum. If we were comparing strategies for training networks, we should use the same initial conditions. To retrain and reduce time we can use the final weights from the last training run.

Training method

Will we present the whole training set of data before adjusting the weights or adjust the weights after the network has seen each example?

Example by example training will involve lower processing time for each epoch as only one example is presented before the weights are adjusted. But it may take a lot longer to stabilise as it will be a while before the network has enough information to produce a reliable model which won't be baffled by new data. It is preferable to use batch training - indeed some algorithms (eg Levenberg-Marquardt) require batches of data.

Training time

When will we be satisfied that training is complete?

The best approach is to carry out a test run and plot the learning curve (the error) as the training progresses. The training time or number of epochs can then be set at a maximum. Alternatively, the subsequent training runs can be supervised and stopped either when the error seems to have settled down or when the learning rate or step size converges in the same way as for the test run.

Test the network

This crucial stage is where we check how well the network has done and look out for symptoms of possible problems - things we would want to adjust when producing a new network.

Incomplete training

How does the error change as training progresses?

If the error was still decreasing when we stopped the training it is likely that the training was incomplete. The best way to check this is to compare the final error with that for a network working on the same problem but using fewer cells. With a smaller network, training will have finished sooner and the error is likely to have levelled off. With a larger network, you would expect to be able to achieve a lower error, eventually: if this hasn't happened, you may need to increase the training time.

Overfitting

How do the errors on the training and validation sets compare?

If the learning curve has flattened out, be wary that the network might be overfitting. However, overfitting will only be an issue where we have:

- too many cells to solve the problem efficiently
- allowed the network to carry on modelling until it has modelled even what we would accept as "noise" in our data

We can check for this by running the validation set of data through the network. The noise in the validation data should be different to the noise in the training data, so the network will not be as good a model and the error in the validation set will tend to increase as the number of cells increases.

Solutions are:

- Reduce the number of cells
- Use weight decay to artificially reduce the number of parameters in the network
- Increase the number of samples in the bootstrap (if used)

Local minimum

Are we convinced that we have reached the global minimum?

If the learning curve has levelled off we may be happy that the training has finished, but the algorithm may have got stuck in a local minimum. Looking at the error of this network compared to one with different numbers of cells will help - we would expect a smooth curve down as the number of cells increase - with some random variation.

If the final error for our network is short of the expected value we need to produce another neural network by starting with different initial values for the weights. Ideally we should always produce a number of neural networks of the same size to check that we have reached a global minimum. We then go on to select the best performing network by looking at its performance in the tests by using other measures eg pick the neural network with the highest IQ.

If the error is continually increasing as the number of cells increases, we may need to adjust the cost function to control the weights to stop the error surface getting too bumpy (see Overweight, below).

Struggling with the problem

What do the errors look like?

We need to check for any correlation between the inputs and the error or the outputs and the error. If there are patterns emerging, these can be removed by weighting the errors. Look for any areas of data for which the model's error is significantly higher than others. If the model isn't coping particularly well for certain areas of the data we can:

- use a different cost function to force it to concentrate harder on the more difficult areas of the problem
- split the problem up into different areas of input
- add more cases where the errors are high

Overweight

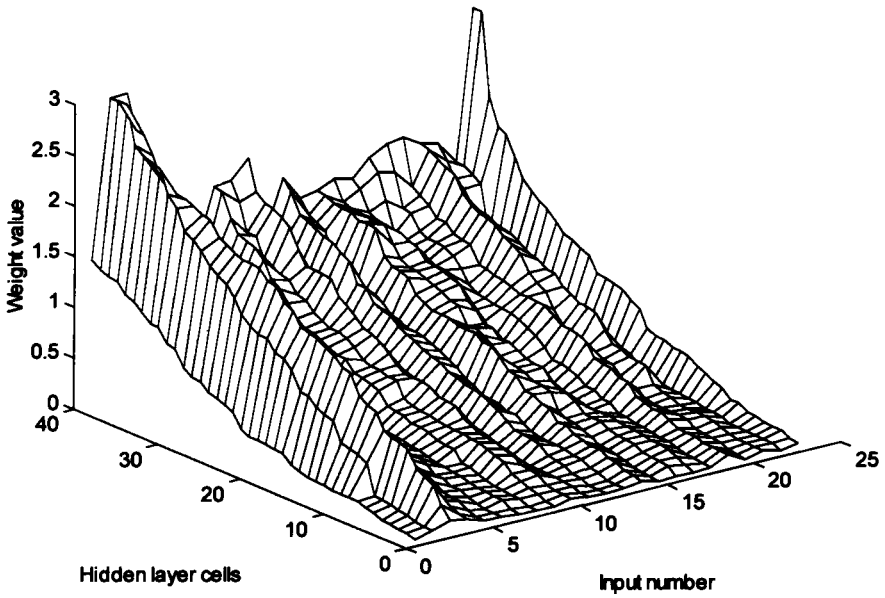
What do the weights look like?

The DTI guidelines recommend that weights should be normally distributed. The graph below shows the weights for a 40-celled network sorted by size for each input. Our best networks did not necessarily have weights which were distributed normally, but rather progressed evenly from one side of the graph to the other. The shape tends to be influenced by the weight decay parameter - being steeper when this parameter is larger.

If any of the weights are too large you would expect ridges somewhere in the output. This can be avoided by adding or strengthening the weight penalty term in the cost function so that the network doesn't get too narrow-minded and stuck in local minima. This will be shown up in the graph of the errors on the training data as the number of cells increase. It will mean that the network will not be as good at generalising, so needs to be managed.

If most of the hidden layer weights are insignificant, you should reduce the weight decay in the hidden layer and increase the weight decay in the response layer. Weights can also be spurred into action by adding some noise as part of the weight update process.

Small weights can also be pruned out using a technique which forces the cells to forget about an input that it doesn't use very much. This is referred to as "optimal brain surgery".



Inefficient use of data

How well is the neural network using the data to create the model?

We have developed a measure of the network's IQ. This is a single index figure describing how well the network is using the data.

We can also view the individual variable contributions to performance made by an input using a graphical representation. Using our prior knowledge of the problem, we can discriminate between two networks which have the same architecture but address the problem in different ways.

Using the chart, we can see if there is an input which is being rejected by network as insignificant. This analysis may point to inputs which the network is handling in the same way - we may be able to remove one of them and simplify (reduce the dimensionally of) the problem.

It has been documented that feedforward networks with sigmoid activation functions require more training sets to establish redundant inputs than networks using other activation functions.

Unsuccessful test - what next?

What are the test results?

If the error goes up dramatically when the net is presented with a new set of data, it may not have had enough data to learn the problem. Alternatively, it may have overfitted the problem, particularly if there are too many cells. Comparing the results with the results for a smaller number of cells will help check whether there is any evidence of overfitting and therefore whether we need to use more data to learn the problem.

Most problems at this stage can be solved by:

- Transforming the inputs or outputs in a different way
- Increasing the amount of data in the training set
- Combining more than one neural network to solve the problem
- Acquiring more processing memory
- Changing the neural network architecture, eg using feedback networks or using activation functions from the radial basis family

---

"A definition is the enclosing of a wilderness of idea within a wall of words."

Samuel Butler

Activation function — The mathematical function computed by each cell, to convert its input value to its output value. Frequently used activation functions in feedforward networks are various forms of sigmoid functions and linear functions.

Also known as: transfer function.

Algorithm — A set of rules for calculation or problem solving.

Architecture — The generic type of neural network that is used.

Back-propagation — A learning algorithm for a neural network. (See Appendix B.)

Bias — The contribution to an activation function which is independent of the inputs.

Bootstrapping — A training technique. This involves repeatedly selecting a different subset of the total data available for training. Sampling from the training set in this way can allow more efficient use of the available time.

Cell — *See neuron.*

Channel Islands — "The Channel Islands are fragments of France that fell into the sea and were gathered up by England. Of the four islands Sark, the smallest, is the most beautiful; Jersey the largest, is the prettiest; Guernsey wild and charming, shares their characteristics. " [20]

Competitive network — A competitive neural network has cells which compete with their neighbours so that only one cell will respond to a particular input pattern.

Cost function — The mathematical function that is minimised by the training algorithm.

Also known as: error function, performance function.

ELT — English Life Table. It is published several years after each census and shows life expectancy and experienced mortality. The latest table, published this year, is ELT15.

Empirical test Optimising the design (eg the number of hidden cells in a neural network) by experimentation and assessing the performance on a test set.

Epoch A pass through the training algorithm after which the weights are altered. During an epoch the network may be presented with the whole training set, one example, or a sample, depending on the method chosen.

Error surface A plot of the errors against possible parameter values.

Feedforward neural network A generic term for neural networks in which the signals are passed in a forward direction only. Once a signal has passed through a cell it does not return to that cells or any other cells that were passed through earlier.

Generalised Linear Modelling (GLM) A mathematical technique, used in particular by actuaries for modelling general insurance claims. It involves transforming inputs and outputs so that a linear relationship occurs.

Generalisation The ability of a neural network to produce a sensible answer when presented with data that was not used to train it.

Global minimum When the weights are set at the point where the value of the cost function is smallest.

Gradient descent A method for calculating weight adjustments, commonly used in conjunction with backpropagation.

Guaranteed Minimum Pension (GMP) A notional pension. Most pension schemes that were contracted out of the State Earnings Related Pension Scheme (SERPS) before 6 April 1997 have to provide a pension for service before that date equal to the Guaranteed Minimum Pension (GMP). The GMP is broadly equal to the SERPS pension that would have arisen if the member had remained in SERPS.

Hard limit function An activation function for a cell in which there is an abrupt transition to the output limits.

Hidden cells Cells that are not associated with inputs or outputs in a neural network but whose activation functions are the internal states of the neural network.

Input cells Cells whose activation functions are set equal to the values of the input data components.

Input variable One of the components of the input data.

IQ Intelligence Quotient. Also our measure of a neural network's performance.

Lateral thinking A technique invented by Edward de Bono. Rather than following an obvious direct line of thought it involves considering seemingly unrelated ideas, many of which will prove fruitless but some of which may provide new insight.

Layer When cells are arranged in groups and one group is fully interconnected to the next group then each group is known as a layer.

Learning The process of acquiring knowledge which a neural network undergoes during training.

Learning curve A plot of the cost function (hopefully improving) over the training time.

Learning rate The rate at which a learning algorithm attempts to proceed towards its goal. Simple algorithms have a constant rate but others vary the rate, using a high rate when far away from the target but slowing down when closer to avoid missing the finer detail.

Linear predictor The answer obtained by multiplying each input by its appropriate weight and adding the bias. The linear predictor is the input for the activation function for a cell.

Local minimum A point which is not the global minimum of the cost function but where a small change in the weights increases the cost function.

Marshmallow Soft, sickly sweet, scrumptious when dipped in molten chocolate. May inhibit weight decay, but otherwise nothing to do with neural networks.

Minimum Funding Requirement (MFR) The MFR is a minimum funding test introduced under the Pensions Act 1995. After a phasing-in period running from 6 April 1997, it requires occupational pension schemes to hold assets at least equal to a certain level. The same method and basis also forms a legal minimum for individual transfer values calculations.

Neural network A mathematical model. A number of individual cells connected together to form a network. The power of a neural network depends on how many cells there are, how they are connected, and how each cell operates. Neural networks are based on the biological process of the human brain. They are not programmed; they learn from example and generalise by experience.

Glossary

Neuron

A single processing element within a neural network. A neuron receives signals from the outside world or inputs from the outputs of other neurons. The neuron uses this information to produce its output using a single mathematical formula. This output is then fed to other neurons or directly to the outside world. A neuron is also a nerve cell within the brain.

Also known as: cell, unit.

Parameters

In the context of neural networks we have used parameters as the collective term for weights and biases.

q_x

A measure of mortality. q_x is the probability that a person aged x will die during the next year.

Recurrent neural network

A neural network architecture where the output is returned as an input to create feedback.

Response cells

The cells of a neural network whose activation function outputs are the outputs of the network.

Also known as: output cells.

Overfitting

The consequence of having too complex a network. The neural network performs well on the training data but poorly on the test data because the complexity of the neural network is such that it starts to fit the measurement noise associated with the training data.

Overtraining

This is where training has been carried out to a point where the neural network starts to learn the noise in the training set. This results in poor generalisation.

Regression

Another word for curve-fitting, but applied in multiple dimensions. A neural network that is required to produce a continuous output, mapping from multiple inputs to multiple outputs is performing a regression task.

Sigmoid

S-shaped.

Stopped training

The practice of stopping a training algorithm when the performance on the test sets starts to deteriorate.

Supervised training — Training is carried out by presenting to the neural network a sequence of examples about the problem to be solved. These examples are in the form of sets of input data which are each associated with a target output. The difference between the target output and the output produced by the neural network is used as the basis to compute changes to the weights. This is repeated until the output is the same as that required within the training data, to within a specified tolerance.

Synapse — A path from one neuron to another to transfer information. Also called connections.

Test set — The data sets used to test the neural network after training has finished.

Training — The process whereby the neural network's weights are determined in order to get optimal performance, usually by the optimisation of the cost function.

Training set — The sets of input data and (where appropriate) the associated target outputs used to train the network.

Training run — One application of the training algorithm: from the start where the weights are initialised, to the point where the training stops.

Transfer function — *See activation function.*

Unsupervised learning — In unsupervised learning the training set has no output response associated with the input stimuli. The network will make associations between the input data.

Validation set — *See test set.*

Variable contribution — An interim step in calculating our IQ measure of a network's performance. It measures the effect on the error of replacing an input with its average value.

Weight — The value associated with a connection between neurons in a neural network. This value determines how much of the output of one neuron is fed to the input of another.

Weight decay — An alteration to a learning algorithm to reduce the size of the weights.

See also marshmallow.

References

"I don't know what I may seem to the world, but as to myself I seem to have been only like a boy playing on the sea-shore and diverting myself in now and then finding a smoother pebble and a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me."

Sir Isaac Newton

1. W S McCulloch and W Pitts [1943]: "A logical calculus of the immanent in nervous activity" Bulletin of Mathematical Biophysics 5, 115-133.
2. Marvin Minksy and Seymour Papert [1969]: "Perceptrons" The MIT Press.
3. David E Rumelhart, Geoffrey E Hinton and Ronald J Williams [1986]: "Learning Representations by Back-propagating Errors" Nature 323:533-536.
4. The Times [11 August 1997].
5. Dirk-Emma Baestaens, Willem Max Van den Bergh and Douglas Wood [1994] "Neural Network Solutions for trading in Financial Markets" Financial Times/Pitman.
6. Edward de Bono [1969]: "The Mechanism of Mind" Pelican.
7. Edward de Bono [1992]: "Serious Creativity" Harper Collins.
8. The Economist [9 October 1993]: "The Frontiers of Finance" survey now published as a chapter in "Going Digital" [1996] The Economist/Profile Books.
9. http://www.uwo.ca/its/news/FAQ/ai-faq/neural-nets.
10. Bart Kosko [1993]: "Fuzzy Thinking" Flamingo.
11. Murray Smith [1996]: "Neural Networks for Financial Modelling" International Thompson Computer Press.
12. B D Ripley [1993]: "Statistical Aspects of Neural Networks" in "Networks and Chaos: Statistical and Probabilistic Aspects" Chapman & Hall.
13. Martyn Dorey [1996]: "Neural Networks Applications in Modelling".
14. ActEd notes - Subject D.
15. Government Actuary's Department [1984]: "English Life Table number 14".
16. G Cybenko [1988]: "Approximations by superpositions of a sigmoidal function" Urbana University of Illinois.
17. A N Kolmogorov [1957] (in Russian) [1963] (translation): "On the representation of continuous functions of several variables by the superposition of continuous functions of one variable and addition" American Mathematical Society 2, 55-59 popularised in R Hecht-Nielsen [1991]: "Neurocomputing" Addison-Wesley.
18. M C Polman [1990]: "The Guernsey 1986 Mortality Table Report".
19. M Savage [1994]: "Guernsey Mortality".
20. V Hugo [1883]: "L'Archipel de la Manche".
21. Martyn Dorey [1995]: "Guernsey Mortality".
22. The Actuary [July 1997].
23. The Institute and Faculty of Actuaries [1997]: "GN11 (v 7.0) Retirement Benefit Schemes - Transfer Values".
24. The Institute and Faculty of Actuaries [1997]: "GN27 Retirement Benefit Schemes - Minimum Funding Requirement".

25. B A Pearlmutter and R Rosenfeld [1991]: "Chaitin-Kolmogorov Complexity and Generalization in Neural Networks".

26. Sontag [1991]: "Remarks on Interpolation and Recognition using Neural Networks".

27. A S Weigend, B A Huberman and D E Rumelhart [1990]: "Predicting the future: A connectionist approach".

28. S Biswas and S Venkatesh [1991]: "The Devil and the Network: What Sparsity Implies to Robustness and Memory".

29. C Bishop [1997]: "Neural Networks for Pattern Recognition".

30. Y Chauvin [1990]: "Generalization Dynamics in LMS Trained Linear Networks".

31. B Hassibi, D G Stork and G J Wolff [1992]: "Optimal brain surgeon and general network pruning".

32. http://www.uwo.ca/its/news/FAQ/ai-faq/neural-nets/part3.

33. Y Le Cun, I Kanter and S Solla [1991]: "Second-Order Properties of Error Surfaces: Learning Time and Generalisation".

34. P Girosi [1990]: "Networks for Approximation and Learning".

35. Krogh and Hertz [1990]: "Dynamics of Generalisation in Linear Perceptrons".

36. Hugh Jarce [1995]: "Introduction to GLIM".

37. DTI [1993]: "Neural computing: learning solutions".

38. The Actuary [October 1997].