A Neural Network Approach to Software Project Effort Estimation

C. W. Dawson

School of Mathematics and Computing, University of Derby, Kedleston Road, Derby, DE22 1GB, UK

Abstract

One of the major problems with software development projects is that it is extremely difficult to accurately estimate their likely cost, duration and effort requirements. This invariably leads to problems in project management and control. Part of the problem is that during the early stages of these projects, very little is known about the problem domain, and initial estimates tend to be best guesses by a project manager. Artificial neural networks appear well suited to problems of this nature as they can be trained to understand the explicit and inexplicit factors that drive a software project’s cost. For this reason, artificial neural networks were investigated as a potential tool to improve software project effort estimation using project data supplied by BT. In addition, in order to deal with uncertainties that exist in initial project estimates, the concept of neural network simulation was developed and employed. This paper discusses this concept and comments on the results that were obtained when artificial neural networks were trained and tested on the data supplied by BT.
1. Introduction

Neural network research is extremely diverse, as artificial neural networks prove to be particularly flexible and powerful in their application. For this reason, artificial neural networks were suggested as an alternative tool for software project cost estimation within BT. A project was undertaken, therefore, to investigate the suitability of using artificial neural networks for software project cost estimation, and to develop and train a number of networks on previous project histories - the results of which are presented here. The networks chosen for this work were feed forward, three layered networks, trained using a variation of the backpropagation algorithm.

An approach, which has not previously been fully explored, is the way in which artificial neural networks deal with initial fuzzy estimates (such as those guesses made at the early stages of a software development project). A simple solution to this problem is to enable trained neural networks to be simulated in order to provide uncertainty based estimates from these fuzzy inputs. This paper discusses this concept, and also discusses how the step size was dynamically controlled in the backpropagation algorithm.

1.1 Software Project Cost Estimation

Unfortunately, cost estimation in software development projects is not an exact science, as there are a number of qualitative factors involved that can rarely be explicitly identified. Because of this, several techniques and approaches have been proposed for software project cost estimation, and the quantity of literature on the subject is enormous. Without an ability to measure, or accurately estimate software project characteristics, organisations have little management control over a project’s behaviour, risks are not clearly identified, and potentially disastrous projects may be undertaken. In fact ‘15% of software projects are not completed due to grossly inaccurate estimates of development effort’ [1]. Also, according to surveys by Charette [2] and in Computer Weekly (19 May, 1994):

- only 20% of projects fall within their estimated cost and schedule
- 66% of companies significantly underestimate software projects’ durations and costs

Figure 1 shows how software metrics can be classified into three main categories - product metrics, process metrics, and project metrics [3] which relate to the three main aspects of a software development. Figure 1 shows how these aspects interrelate with each another, and where metrics are generated and used. A software development project will use a particular development process to produce a software product.

Usually, software project estimates are based on initial estimates of both the product that is being developed and the process that is being adopted. For example, product estimates might include quantitative measures such as size of the product (in terms of lines of code measured by an agreed standard), functionality (in terms of function points), and qualitative measures such as complexity and quality. Process estimates might include factors such as team productivity, effectiveness of defect removal procedures and so on.

Techniques that are often used for software project estimates include COCOMO (which uses an algorithmic approach based on product estimates...
[4]), Delphi (a technique used to balance different experts’ opinions [5]), and function points (a technique that measures the complexity of a product [6]).

Figure 1: Relationship between Project, Process and Product

2. The BT Project

A project was undertaken at BT’s Martlesham Heath laboratories to investigate the effectiveness of a neural network approach to software project cost estimation. Several programs were written to implement artificial neural networks which were then trained and tested on previous project histories. The results of this work are presented later.

2.1 The Programs Developed

Four main programs were written for this project. The first program allows a user to create training sets for the neural networks, based on data from previous software project histories. The second program normalises the data within a training set, before it is passed on to the third program that generates and trains the neural networks using backpropogation. The backpropogation algorithm was adapted to include momentum [7] and a dynamically controlled step size algorithm (discussed below). The final program takes a trained neural network and allows a user to ask it various questions. These questions can include simulation of the network to provide estimates of project cost/effort requirements, based on initial best guesses by the user (discussed later). The distribution functions generated by this program, as text files, can then be analysed to determine the uncertainties involved with the estimated project.

3. Step Size in the Backpropogation Algorithm

In order to improve the training cycle and efficiency of the backpropogation algorithm used in the programs developed, and to avoid a network becoming caught in local minima, a variation on the backpropogation algorithm was used. Other authors have suggested similar, automated tools for adjusting the step size, for example, Jacobs [8] and Fahlman [9]. The algorithm used reduces the step size of the backpropogation algorithm during training, based on the number of training cycles (epochs) that have been performed. The
equation chosen for this variation is presented in equation (1) and is based on the sigmoid function used in the backpropogation algorithm. The shape of this function is shown in figure 2. In this case, \( x \) represents the number of epochs performed, \( r \) the number of epochs required, \( q \) the initial step size, and \( p \) the final step size.

\[
f(x) = p + (q - p) \left(1 - \frac{1}{1 + e^{\left(10 - \frac{20x}{r}\right)}}\right)
\]

(1)

This algorithm allows the initial step size to remain relatively high (around 0.1), allowing the weights to move towards a global error minima. As training continues, and approaches the required number of epochs, the step size is reduced to a lower level (around 0.01) to ensure that the weights stabilise (hopefully) at the global minima, and not in oscillation between two minimum values.

![Figure 2: Step Size Versus Epochs Performed](image)

4. Neural Network Simulation

Using artificial neural networks to predict software development effort is not a new idea. For example, Kumar et alia [10] used neural networks to estimate manpower buildup levels in software projects, based on task concurrency, inverse application complexity, and schedule pressures. In addition, Hakkarainen et alia [11] used artificial neural networks to predict software product size.

This paper addresses another related issue by introducing the idea that artificial neural networks do not need to be limited to deterministic inputs, that produce a single estimate for a software project’s effort requirements. This is achieved by simulating trained networks in order to account for uncertainties in initial estimates provided by project managers. The trained networks thus provide probability functions that represent the uncertainty involved in the potential cost of a software development project. Figure 3 provides a visual overview of this concept.

Trained neural networks require single sets of inputs in order to
provide an expected output(s). Quite clearly, if the initial inputs are best
guesses, any variation from true values can have a significant affect on the
value of a trained network’s response. A network should be able to provide
an indication of the uncertainty involved in its outputs, knowing the
variability of the initial inputs.

In order to achieve this aim, the networks developed at BT can be
simulated with selected input distribution functions; such as normal,
rectangular, triangular, and discrete. Thus, if a project manager provides an
initial estimate of the size of a software product as, for example, 50000 LOC
(lines of code) plus or minus 10%, a trained network can simulate this input
as a rectangular distribution with limits 45000 LOC to 55000 LOC. The
network samples from this distribution a number of times (selected by the
user, but usually about 1000) and builds up a picture of the likely cost of this
project based on this fuzzy input. The network thus generates a distribution
of the likely cost of the project that can be analysed. In figure 3, an initial
cost driver passed to the trained neural network is represented by a normal
distribution function. The output generated by the simulation is represented
by the probability effort distribution. In addition, the cumulative distribution
function has also been generated from this output to determine the probability
of completing the project within particular limits.

5. The Data Used

Two sources of data were obtained from BT for this project, and
consequently two separate sets of neural networks were trained. The first
data source provided information on twenty six projects relating to a larger
suite of three software system releases. These data provided early product
code estimates from which overall project effort estimates were made by
managers. In addition, as these projects had already completed, these data
also provided actual overall effort expended on each project, so that a
comparison could be made with the initial effort estimates. It was clear that
managers generally overestimated final project requirements by around 30%
on average in this data set, although their estimates did appear consistent.

As a starting point, these data provided an ideal baseline for
developing and testing neural networks for software project effort estimation.
A network was trained on these project histories on the relationship between initial coding estimates (used by managers to guess overall estimates) and final, actual project effort. As the initial coding estimates were only best guesses at the initial stages of a project, the trained neural networks were programmed to simulate these fuzzy guesses using a Monte Carlo-type approach. Although the networks were rather simplistic, with a single input and a single output, their important feature was their ability to be simulated in order to deal with best guessed inputs.

The second data source at BT provided more complex relationships between initial cost drivers and actual project efforts. In these data, four cost drivers were presented:

- Effective Source Lines of Code - code written, not reused
- Function Points
- MBI - Manpower Buildup Index
- PI - Productivity Index

These led to values for actual effort (in person months), cost accumulated (pounds sterling), and duration (in months) for each project. When these data were analysed statistically, it became clear that there was no explicit correlation between any of the cost drivers and any of a project’s actual cost, effort or duration requirements. Consequently, a neural network approach seemed well suited to a problem of this nature, where the true relationships between cost drivers and effort were implicit and difficult to determine. Networks were trained, therefore, with four inputs and two outputs on these data (as effort was a direct function of cost). Table 1 provides an example of one of the training pairs used, from a previous project history.

<table>
<thead>
<tr>
<th>Project</th>
<th>PI</th>
<th>MBI</th>
<th>ESLOC</th>
<th>FP</th>
<th>Effort</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project 4</td>
<td>15.9</td>
<td>8.6</td>
<td>33000</td>
<td>300</td>
<td>168.4</td>
<td>842000</td>
<td>7.4</td>
</tr>
</tbody>
</table>

Table 1: An Example of a Training Pair from the Second Data Source

6. Evaluation

This section presents some of the results that were obtained when neural networks were trained using the second source of data from BT. The results from the first data source are not discussed as these data represented a rather simplistic, linear relationship between a single cost driver and expected project costs. This did not really test the ability of a neural network to find implicit relationships between several variables, although these data were used to test the concept of neural network simulation - a technique which appeared to be well suited to estimates of this nature.

6.1 Training

In order to test the neural networks developed on the second data source, thirteen of the twenty three available projects were used - these being projects in which a full set of information had been recorded. Thirteen neural networks were initially trained with one training pair (representing the
information of one project) excluded each time. Therefore, each of the thirteen networks was trained on the data from twelve of the thirteen projects. This enabled the excluded training pair to be used as a test case on which the accuracy of a trained network could be evaluated. The alternative would have been to use the networks on future projects, the results of which would take months to obtain.

It was noted that the number of nodes within the hidden layer of the trained networks could have a significant affect on their ability to generalise. In order to evaluate the affects of this, networks were trained with 3, 5, 7, and 10 hidden nodes, and consequently 102 networks were trained in all - each network being trained for one hundred thousand epochs. However, it is now felt that this number of epochs may be too large and consequently networks may lose their ability to generalise. This is particularly important when network simulation is used because generalisation is the key to success here. The number of epochs required to train a neural network for simulation is an area that requires a more detailed investigation.

6.2 Results

Table 2 summarises the results that were obtained when the thirteen sets of trained networks were tested on each missing project’s data. The Min and Max values refer to the worst underestimates and worst overestimates produced by each network set for both cost and duration. The MSRE values are the mean squared relative errors for each network set - calculated as the mean of the square of the errors relative to each actual project duration and cost. These values provide more meaningful measures of overall network performance than relative differences alone. The last two lines of values show the mean and variance of each network set’s estimates for both duration and cost.

<table>
<thead>
<tr>
<th>Number of Hidden Nodes</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>53%</td>
<td>42%</td>
<td>48%</td>
<td>40%</td>
</tr>
<tr>
<td>Max</td>
<td>333%</td>
<td>182%</td>
<td>289%</td>
<td>184%</td>
</tr>
<tr>
<td>MSRE</td>
<td>0.54</td>
<td>0.17</td>
<td>0.39</td>
<td>0.22</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>52%</td>
<td>42%</td>
<td>40%</td>
<td>31%</td>
</tr>
<tr>
<td>Max</td>
<td>184%</td>
<td>196%</td>
<td>205%</td>
<td>203%</td>
</tr>
<tr>
<td>MSRE</td>
<td>0.10</td>
<td>0.15</td>
<td>0.17</td>
<td>0.16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>112%</td>
<td>107%</td>
<td>105%</td>
<td>102%</td>
</tr>
<tr>
<td>Time</td>
<td>57%</td>
<td>18%</td>
<td>18%</td>
<td>24%</td>
</tr>
</tbody>
</table>

Table 2: Results of Neural Network Testing

At first glance it might appear that the best results were those produced from networks that were trained with 10 hidden nodes. In these cases the networks produced results, on average, within 2% of the actual costs of the excluded projects, and within 1% of the actual duration of these projects. It should be noted, however, that although these figures appear quite accurate,
they do not reflect a consistently accurate estimate of each projects’ cost and duration. Sometimes the networks would overestimate the costs/durations and sometimes they would underestimate them - this averaging leads to these apparent accurate results. The MSRE figure provides a more meaningful measurement of network accuracy in these cases. With this measurement it is clear that networks with 5 hidden nodes provide a more accurate estimate of costs (overall) whilst a more accurate estimate of duration, overall, is obtained with 3 hidden nodes. This emphasises the differences that network configurations can have on the accuracy of results that are obtained and implies that network selection and construction is as important a part of neural network development as training itself.

6.3 Summary

It became clear as the networks were tested that some of the estimates were consistently inaccurate for particular projects, irrespective of the hidden node configuration. These inaccuracies highlighted other factors that were affecting these projects’ durations and costs. For example, the sixth project, of the thirteen, was consistently overestimated by each trained network. This was due to this project being a second release of a software system - something that had not been trained into the networks and had not been represented in the input data. It was only during testing that these anomalies were spotted and their affects could be anticipated. For those projects were additional factors were not present, the trained networks provided quite reasonable estimates for costs and durations based on the four input drivers. For example, a network with 7 hidden nodes estimated the cost of the third project to within 1% of its actual cost. Further networks would need to be trained, however, in order for other factors, not initially identified, to be rationalised.

7. Conclusion

In conclusion, artificial neural networks have proved that in certain cases they can accurately estimate software project costs and durations, providing that all factors affecting a project can be determined, and a network’s configuration can be sensibly formed. Future work will involve comparing the results obtained here with other costing techniques. In addition, this paper has introduced the novel concept of neural network simulation. This allows fuzzy estimates of a software product to be considered by a network, which can then identify the affects of this uncertainty on software project effort and cost requirements.

8. Acknowledgments

Thanks must go to Richard Ward and Paul Smith of BT who assisted in this project and provided some valuable comments during the writing of this paper.

9. References


