Software Engineering Methods for Neural Networks

Anthony Senyard, Ed Kazmierczak, Leon Sterling
Department of Computer Science and Software Engineering
The University of Melbourne
ICT Building, 111 Barry St, Parkville
Melbourne, Victoria, Australia
{anthls, ed, leon}@cs.mu.oz.au

Keywords: neural networks, verification and validation, specification, development process.

Abstract

Neural networks have been used to solve a wide range of problems. Unfortunately, many of the applications of neural networks reported in the literature have been built in an ad-hoc *manner, without being informed by the techniques and tools of software engineering. The problem with developing neural networks in an ad-hoc manner, using a "trial and error" or "build and fix" approach, is that successes are difficult to* repeat*. Building neural networks to solve specific problems using ad-hoc processes is repeatable only if there is a sufficient culture of disciplined practice and experienced people in the organisation to facilitate the process. In this paper we propose a set of methods for developing neural networks that can be used to systematically and repeatably "engineer" neural networks to solve specific problems. We explore the "design problem" for neural networks, and the problem of validating and verifying the operation and learning algorithms for neural network software. A feature of our approach is to separate the generic components of a neural network from the application specific components.*

1 The Problem of Creating Neural Networks for Specific Applications

This paper describes a development process for neural network software, informed by software engineering principles and practices. While there have been numerous successful neural network applications, the development processes reported in the majority of the literature could not be characterised as predictable nor repeatable, both properties central to an engineering approach. It is the lack of a design process and the absence of verification and validation methods for the design prior to implementation which constitute the most significant contribution to the unpredictable and unrepeatable nature of neural network development.

1.1 The Design Problem

In traditional software engineering development, design is a human guided process of developing a solution by choosing, creating and assembling elements to satisfy the users requirements. The elements are sometimes generic (such as design patterns) and sometimes specific to the given requirements. A program is then a collection of data structures and algorithms chosen during design. Making appropriate choices during design is made easier by layering choices at different levels of abstraction, for example by employing architectural and intermediate design levels.

Confidence in the design can be attained through inspections, walkthroughs, prototyping and other similar forms of design evaluation. This approach to design is not applicable for neural network software development because there are no known methods to completely design a neural network *a priori*. Instead neural networks modify their own structure through the application of learning algorithms.

What most often occurs in practice is that a generic *neural network model* (such as a multi layer feed-forward network) is combined with a *learning algorithm* (such as back-propagation) and problem data to create a *specific neural network*. The choice of a learning algorithm effectively replaces the more traditional design step, but the process of applying the learning algorithm is not completely automatic and does require human intervention.

Further, the selection of parameters for the learning algorithm is closer to design in the traditional sense. For neural networks, however, the verification and validation methods, and guidelines for the selection of parameters are not well developed. There is no oracle or theory which explains how to make the myriad of choices required to apply a generic neural network model to a specific problem.

The wide range of choices available to neural network developers has already been identified as an issue in [6]. They consider the problem of training a machine learning model (of which neural networks are an instance) from data and investigate the issues involved in solving real-world prediction problems. They use the term *application development* to describe the overall process of applying a particular model (from a family of candidate models, such as radial basis function networks) to a domain-specific real-world problem. We extend the concept of application development to *problem specification* and *neural network creation*.

Problem specification is the process of selecting the parameters associated with the neural network structure and the learning algorithm. Multiple combinations of selections can be made each combination is referred to as "parameter collection". *Neural network creation* is the process of systematically applying the learning algorithm with each associated parameter collection as defined in the problem specification.

The problem specification and neural network creation steps help us to achieve process predictability and repeatability. We can measure the resources required for a single iteration of the neural network creation step using a single parameter collection. In turn the data from these measurements provide us with a basis for predicting the overall resources required to apply the learning algorithm with all of the parameter collections. If we have historical data from previous projects, we can make this prediction before *neural network creation* begins. By recording all of the parameters associated with neural network creation we (and others) can repeat the development by following and applying the learning algorithm with the parameter collections recorded in the problem specification document.

The *problem specification document* is the key mechanism for identifying all of the choices facing developers and structuring these choices according to various levels of abstraction. It contains information on relevant problem-domain factors: for example, noise in the collected data; data factors, for example data normalisation and problem encoding; and human factors, for example performance requirements. The problem specification also contains information on the selection of particular neural network structures, learning algorithms, and associated parameters. The structure of the problem specification corresponds to the levels of abstraction, such that progressively more detail can be given in each section.

1.2 The Quality Assurance Problem

The major issue of quality assurance surrounding neural network software is the verification and validation of the generic model - how can we be confident that the generic components (eg. neural network models and learning algorithms) are *correct*[1].

This raises an issue concerning the appropriate separation of the specific problem requirements and the generic components for a specific problem. The models of neural networks are common across problems but the specifics of the problem are unique to each particular development. Our approach to addressing this issue is in providing methods for adequately capturing both the specific and generic requirements.

However, in contrast to the problem specification which is unique to each development effort, the generic neural network model can be reused. Indeed, there are a number of advantages in the neural network development community sharing a generic neural network model. Firstly, a common model allows for problems with the model to be resolved by the community rather than by an individual. Secondly, the validation of the model can be conducted via a process of wide spread community review resulting in a standardised neural network model. Finally, a shared model implemented soundly in a prototype could be used to verify the neural network implementation of any neural network developer. For example, achieving the same output from the prototype and the given implementation increases the confidence that the given implementation is correct.

These reasons have motivated our construction of a prototype implementation of a Z specification of neural networks. Space does not permit all of the specification to be presented but a sample is given in Section 4 along with a description of the associated verification and validation activities. Others have attempted to specify neural networks in [33, 34, 13, 9, 10, 1] but without the benefit of a comprehensive development process, and without addressing the matter of systematic analysis of the problem data or verification and validation issues.

Some neural network specific verification and validation work has been addressed in [12] but is limited to benchmarking and does not treat generic and specification concerns as we do in Section 2. Related work on approaches to using formal specifications to verify and validate knowledge-based systems are presented in [19].

1.3 Engineering Neural Networks

The application of software engineering principles to the development of neural networks has received little attention, except in the work of [26] and indirectly in [6]. The aim of this paper is to provide some of the methods required to address the problems in neural network development pre-

[1]By correct we mean that the numerical calculations performed during operation and learning are the same as those specified by accepted references and our software specification.

sented above, and to allow the development of neural networks for specific problems to be incorporated into traditional, planned, software engineering projects.

In the literature the process of developing neural networks has typically started with requirements and then moved to implementation, often leaving design choices implicit. However, making these design choices explicit has a number of advantages. In section 2, we present an approach which aims to enumerate the set of design choices and can be used to observe correlations between particular choices and resulting development outcomes. By recording design choices, repeatability is possible because all of the information required to repeat the development is documented.

2. Factors in the Design of a Neural Network

Our view is that design of neural networks is a process of making choices[2] (possibly creative choices, regarding the models, learning algorithms and associated parameters. The first set of choices that we typically face in designing a neural network to solve a specific problem is to decide on:

(i) the *Neural Network Model*, for example, multiple layer feed forward networks, feedback networks, radial basis function networks, associative memories, support vector machines and the type of transfer function the nodes in the network implement; and

(ii) the *Learning Algorithm*, which often depends on the choice of neural network model, but can be divided into:

- gradient descent methods, such as back propagation and associated derivative algorithms like quickprop and robustprop, and
- stochastic methods, such as simulated annealing, genetic algorithms, evolutionary programming and monte carlo methods.

A complicating factor is that unless there is specific guidance available then we will typically need to choose a number of different combinations of neural network models and learning algorithms in order to determine which combination will satisfy the performance requirements. By documenting these choices and the results for specific problems we can begin to collect enough empirical knowledge to provide guidance in making these choices.

The next step is to choose the parameters associated with our first set of choices. Current models and learning algorithms utilise the following parameters:

(i) parameters governing the structure of the neural network including:

- the number of layers,
- the number of nodes in each layer,
- the transfer function in each node,
- the weights and their initial settings which determine the connectivity between nodes.

(ii) learning algorithm parameters — each learning algorithm has a number of parameters associated with it, such as the learning constant and the momentum factor for back propagation.

(iii) training and testing strategy — what proportion of patterns should be used for training and what proportion for testing?

(iv) training and testing patterns — which patterns, out of all of those available, should be used for testing and training?

(v) error levels — what error levels, both in terms of precision and accuracy, can be expected?

The selection of appropriate parameters is the difference between solving the given problem and not solving it. A major dilemma facing the neural network software developer is what set of parameters should be used for a particular problem? At this stage, no oracle or body of empirical data is available to answer this question. Several attempts to automate this process have been proposed, particularly those which employ evolutionary computing methods [44, 45, 8, 21, 24].

3. The Process of Developing Neural Networks

3.1 Predicting and Repeating Development

There is an excellent reason for recording relevant problem information. By defining the datasets and testing and training strategy in the problem specification phase we can prevent the neural network creation phase from consuming an unbounded and unplanned amount of resources. Another major problem facing neural network development is the unpredictable nature of current practice with regard to the creation of a specific neural network. The current approach can be characterised as an unguided iterative approach along the lines of:

1. Choose a problem representation, create an associated data set and set performance requirements.

2. Train a neural network while varying the associated parameters to achieve the given performance requirements.

3. If the performance requirements are met, stop. Otherwise try again.

[2]The idea of design as a series of choices comes from the Extended ML method [29].

The problem with this approach is that there is no plan for meeting the requirements and the number of iterations is often unbounded. It is particularly hard to isolate the reason for not achieving the performance requirements, was it the parameters associated with the testing and training strategy? Or the problem representation?

Our solution is to break the activities into distinct phases which cannot overlap. This does not stop feedback between the phases but prevents the unbounded consumption of resources in the neural network creation phase. An example of this process is:

1. Create N valid (according to the developers judgement) problem representations and create N associated data sets.

2. Associate each data set with a testing and training strategy and all the parameters required for neural network development (referred to as a "collection").

3. Rank each collection of data sets, strategy and associated parameters for the neural network creation phase. This ranking places an order on which collection will be used to develop a specific neural network first.

4. Beginning with the highest ranking collection, develop a neural network in accordance with the data set, strategy, and parameters supplied.

5. If the performance requirements are met, stop, otherwise proceed to the next collection. If all collections are tried and still none of the performance constraints are met then the neural network development should be reevaluated.

This rest of this section is devoted to a more detailed presentation of the process model for neural network development. This development process can be extended by practitioners to address issues not explicitly covered in this paper.

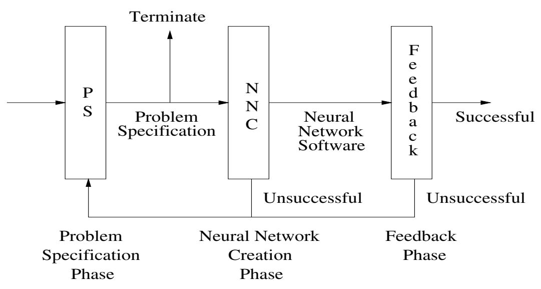

Figure 1. Neural Network Development

At the highest conceptual level there are three distinct phases as shown in figure 1, (1) Problem Specification, (2) Neural Network Creation, and (3) Feedback. However, before we can provide details of the development process we need to identify the roles associated with development.

In a generic description of software development, there are at least three roles: a user, a client and a developer [30]. These correspond to roles in neural network development. Firstly, the **neural network user**, is the person who uses neural network software to solve problems as part of an overall system. Secondly, the nature of neural networks requires an understanding of the problem that is being solved. The role with this understanding is referred to as the **problem domain expert** and roughly corresponds to a client. Thirdly, there is the **neural network developer**. This is an individual or group who creates specific neural network software for a **neural network user** using generic neural network software. In addition to these three roles, there is an additional role involved in neural network software development, the **generic neural network developer**. It is their role to create the generic neural network software based on neural network models. The roles and their relationships are illustrated in figure 2.

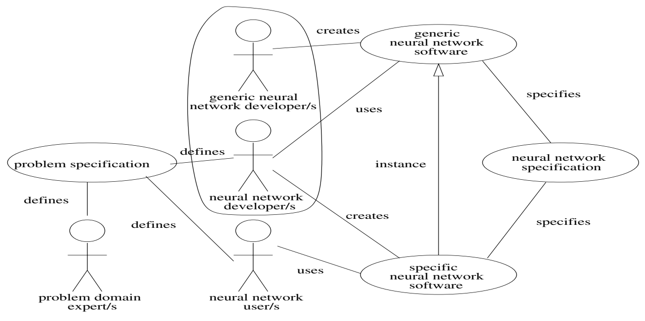
creates
generic neural network software
generic neural network developer/s
specifies
uses
defines
problem specification
instance
neural network specification
neural network developer/s
defines
creates
specifies
defines
uses
specific neural network software
problem domain expert/s
neural network user/s

Figure 2. Development Roles

3.2 The Problem Specification Phase

The first phase is *problem specification* where the problem (as determined by the associated data) is thoroughly defined, as are analysis techniques and precision and accuracy requirements. Problem specification is performed by the user, developer and problem domain expert/client. A majority of this information is used in the next phase of development but some is used to index and characterise the problem for later use as part of a historical record for future development and planning, to allow for repeatability.

There are three main scenarios concerning the specification of neural network development requirements. Firstly, where the client requires that, for some aspect of the system, a neural network is used. In this case, the neural network is a component of a larger system and the accuracy and precision constraints of the neural network are explicitly given by the client.

Secondly, where the client requires some functionality which has associated performance requirements which can be satisfied by using a neural network. This is similar to the first case except that the client has no explicit requirement for using a neural network but the developer determines that a neural network could be an appropriate technology to satisfy the clients requirements and users needs.

Thirdly, where the client requires some functionality which could be satisfied by using a neural network but doesn't have any performance requirements. In this case, the developer must set appropriate performance constraints and can identify alternative technologies (for example, Bayesian belief networks or genetic algorithms) for potential use if neural network development is unsuccessful. The performance requirements are implicit in the design of the system determined by the developer. The dual nature of neural network requirements means a specification of both the problem data and neural network models is needed. In this section we address the problem specific content. The key property of the mechanism for recording the problem specific content is repeatability. However, because of the lack of theory associated with neural network creation, the exact information to record depends on the specific problem. To accommodate this variation, a problem specification document template which can be adapted on a case by case basis is used. The problem specification document captures the problem specific requirements in sufficient detail to allow for repeatability of development.

The problem specification document is not complete once the problem specification phase has finished. The results of neural network creation (an outcome of the next phase) are included in the document for predictability and repeatability purposes. Even unsuccessful neural network training is documented to ensure that unsuccessful development is avoided the next time a similar problem is faced.

The key content in the problem specification document are the *collections* which contain the different problem representations, learning parameters, precision and accuracy requirements and testing and training strategies. These collections give explicit guidelines for creating the specific neural network. Each of the collections is ranked by the likelihood that the collection will achieve the precision and accuracy requirements. The method for predicting the likelihood of achieving given precision and accuracy requirements is based on individual experience. As more experience is gathered within the neural network development community a more suitable method can be developed.

3.3 The Neural Network Creation Phase

The second phase is that of *neural network creation*, where the neural network software is created by combining the collections specified in the problem specification document and the generic neural network technology.

As part of this phase, a specification of the generic neural network technology is needed. Such a specification addresses the second part of the previously identified requirements problem. Three desirable properties of a neural network specification are formal verification. validation methods, and easy implementation and prototype creation.

3.4 The Feedback Phase

The third and final phase, *feedback*, is where the client and user evaluate the performance of the neural network. Depending on the capabilities of the development organisation, alternative methods can be implemented for evaluation purposes. Alternative methods provide a sanity check for the neural network solution that has been developed. These activities establish whether the performance of the neural network is better than available alternatives. The resulting problem specification, project plan, neural network formal model used for development, and the feedback from the user and evaluation of the neural network are placed in the historical record for repeatability and planning (an aspect of predictability) purposes.

4. Phases of Development

4.1 Problem Specification

The aim of the problem specification phase is to specify the problem, analyse the quality of the problem data and specify performance requirements. These activities are supported by a problem specification document template. The template includes sections for data analysis, data representation, and methods for data preparation as well as specifying the performance requirements. The performance requirements stipulate the desired precision and accuracy of the specific neural network software for it to be considered successful. Without these requirements, determining when the neural network creation phase has concluded is problematic and can lead to excessive expenditure of development resources.

The main reason for performing data analysis early in the development process is to provide an assessment of the viability of solving the problem before neural network creation begins. Neural network learning is a data driven technique and for this reason the data used to create the neural network is of critical importance. Unfortunately, neural networks learning algorithms are not so sophisticated as to identify poor data in themselves. Neural networks are robust in the face of poor quality data but it is advisable to assess the data before neural network creation. Creating a neural network with higher quality data increases the likelihood of learning the problem. The difficulty is in deciding if data is actually poor quality or if it only appears to be but this is indicative of the problem-domain. There is a trade-off here: making the data easier to learn can be at the expense of eliminating valuable, possibly even critical, data. This trade-off must be made on a case by case basis but it is clear that recording these decisions for repeatability purposes and future trade-offs is a good idea. Unfortunately, there is insufficient space to give details on analysis methods nor who to interpret the associated results and we refer the reader to the following sources on specific data analysis techniques:

1. Standard data analysis methods, such as, re-scaled range analysis, randomness tests [27], sampling error estimates, principal component analysis [37, 38, 2, 20].
2. Standard visualisation measures, for example, the phase and frequency of the data [14, 11].
3. Statistical summaries, such as the mean, median, mode, mean deviation and standard deviation. These measures of data sets capture key features of central tendency and spread and have nice properties under suitable conditions [43].
4. Graphical summaries in the form of histograms, stem-and-leaf plots and box-plots provide useful information about the centre, spread and other features of a dataset [43].

In the case of problems for which there are few or no similar examples the historical record may be inadequate. The suggestion is to replace the historical record with research into the problem domain.

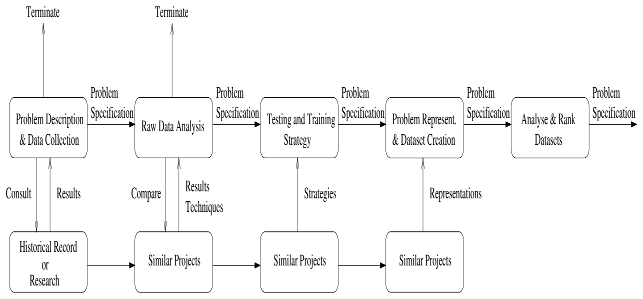

Figure 3. Problem specification phase

Note that some issues are hidden within this process. For example, in one application in mathematical formula recognition, no recognition error levels were supplied by the user and error levels were determined through data analysis [32].

4.2 Neural Network Specification

The hardest single part of building a system is deciding what to build. [7]

Sallis et. al. estimate that approximately 53% of all problems in software development result from poor requirements [28]. Schach et. al. report results from a variety of projects, conducted over the last 25 years, which describe the rapid increase in cost for correcting faults originating from poor requirements as software development progresses [30]. Indeed, there is universal agreement that poor requirements are a significant cause of problems later in the software development process. Given the pivotal role of the generic neural network software we require that the associated requirements are less problematic. One form that requirements can be stated in is a formal model.

The advantages of formal models for requirements specification have been well documented [35, 22, 42, 30, 23]. Firstly, they provide an environment suitable to analyse requirements and design in order to explore the consequences of choices and resolve problems. Typically, a formal model can be used to find ambiguous or incomplete requirements, determine over-specification or predict some aspects of behaviour. This can be accomplished by proving properties of the model or creating a prototype from the model and testing its properties. In particular, rapidly creating a prototype is an inexpensive method for gaining confidence that a specification model is complete, consistent and captures the key properties required of the proposed system (an example of domain directed verification [17]). Prototyping and its relation to the verification and validation of the neural network requirements is the subject of a later section.

Recall in figure 2 that there are two key parties concerned with the neural network requirements, at different levels of abstraction, are the neural network user and the neural network developer. The user is typically concerned with the requirements at a higher level of abstraction (such as the dynamic operation) while the developer is concerned with requirements at all levels of abstraction. Both sets of requirements are presented separately but we show how the two sets of requirements can be kept consistent within a single model.

The Z specification language has been chosen to describe our formal model because of a number of advantages. The mathematical basis of Z allows for powerful abstraction which can be easily communicated and verified. Z is converging to a standard notation that combines many mathematical constructs into an extendible framework. Z has already been used to specify critical systems in other application areas [3, 5, 4, 15, 16, 18, 25]. It is a strongly typed language which is extremely useful for both specification and subsequent development. Z supports high level abstraction which allows us to avoid over specification. A primary *black_box_network*

$input\_size, output\_size : \mathbb{N}_1$

Figure 4. Black box network

benefit of Z is the ability to specify requirements completely independent of any idea of computation. Finally, Z provides a framework for the rigorous proof of the properties of specified neural networks.

In particular, Z helps to eliminate the following two deficiencies from requirements. Firstly, ambiguity — this can be introduced without a consistent and systematic approach to formal specification [39]. For example the use of conflicting natural language descriptions of aspects of the system. While a Z specification can be augmented with an natural language description the Z specification is definitive. Secondly, inconsistency — some specification notations are not sufficiently rigorous which can introduce inconsistencies [22]. Using a mathematical notation allows for static type checking of the specification. This ensures that the Z types used throughout the specification are consistent. There are a number of tools available for type checking and formally verifying Z specifications. For example, *MUZ* (Melbourne University Z type checker) and FUZZ. These tools are used to automatically check the type consistency and 'well formedness' of the Z specification. The specification presented here was type checked using 'muz'. Muz recognises input that conforms to the Spivey Z syntax [36] but with extensions from the draft Z standard [3].

Unfortunately, there is insufficient space in this paper to discuss all of the issues associated with specification or present all of our formal specification (which can be found in [31]). The following snapshot should give the intent and flavour of our specification. In the given schema, the first half (above the line) specifies the properties represented as Z constructs. The second half (below the line) specifies the constraints on the properties. The specifications given here are abstract and do not contain computational information so as to allow the specifications to accommodate a wider range of design and implementation models.

We begin by presenting the user requirements for a neural network as a black box which takes a certain number of input and produces a certain number of output as illustrated in figure 4.

In order to demonstrate the usefulness of our method of specification we will now specify a basic neural network and use this as the basis for defining a layered network through the addition of constraints. The addition of constraints to previously defined models makes the specification framework extendible. We also use this approach to extend the definition of black_box_network to the basic network. By this method, the user and developer requirements are consistent even though they are described at different levels of abstraction.

[3]For further information on the Z standard, please refer to 'http://www.comlab.ox.ac.uk/archive/z.html'

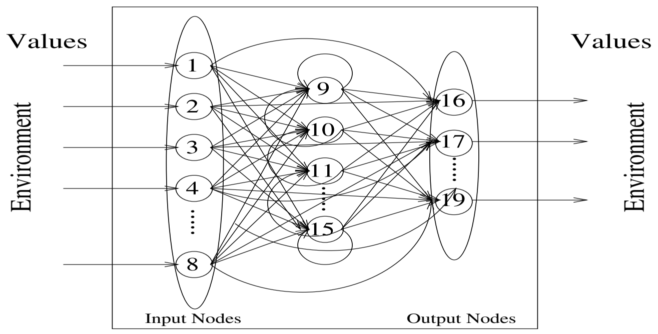

Figure 5. Model of basic network

If we look inside the black box (as illustrated in figure 5) then we can see that the common properties and structure of a basic neural network is:

1. Every network has a sequence of input nodes which can receive stimulus (in the form of values) from the environment, ie. values from outside the network. In the schema we denote input nodes by *input_nodes* (an injective sequence of nodeids).
2. Every network has a sequence of output nodes which transmit values from the network to the environment. In the schema we denote output nodes by *output_nodes* (an injective sequence of nodeids).
3. Each node in a network has an identifier called a nodeid. This allows the specification to define neural networks without knowledge of the node. It also means that we can create networks with a variety of nodes so long as they have a common interface. In this way we can create a heterogeneous network of nodes. That is, a network must be able to have a mixture of nodes with different transfer functions.
4. Every network contains a function, connections, which defines the connectivity of nodes in the network. The connections store the weights which connect nodes together.
5. In all networks, each node has a sequence (ordered set) of source nodes which supply excitory or inhibitory values. In order to be able to apply an inverse mapping of the sources (ie. a sinks function) we use an injective sequence.

We can also identify several building block components: nodes, values, nodeids, and weights. Nodes are a special

$$
\begin{array}{|l}
\hline
\textit{network}[\textit{NODE}] \\
\textit{black\_box\_network} \\
\textit{input\_nodes} : \mathrm{iseq}\,\textit{NODEID} \\
\textit{output\_nodes} : \mathrm{iseq}\,\textit{NODEID} \\
\textit{nodes} : \textit{NODEID} \mathrel{+\!\!\!\to} \textit{NODE} \\
\textit{connections} : (\textit{NODEID} \times \textit{NODEID}) \mathrel{+\!\!\!\to} \textit{WEIGHT} \\
\textit{sources} : \textit{NODEID} \mathrel{+\!\!\!\to} \mathrm{iseq}\,\textit{NODEID} \\
\textit{sinks} : \textit{NODEID} \mathrel{+\!\!\!\to} \mathbb{P}\,\textit{NODEID} \\
\hline
\#\textit{input\_nodes} = \textit{input\_size} \\
\#\textit{output\_nodes} = \textit{output\_size} \\
\mathrm{dom}\,\textit{connections} \subseteq (\mathrm{dom}\,\textit{nodes} \times \mathrm{dom}\,\textit{nodes}) \\
\mathrm{dom}\,\textit{sinks} = (\mathrm{dom}\,\textit{nodes}) \setminus (\mathrm{ran}\,\textit{input\_nodes}) \\
\mathrm{ran}\,\textit{input\_nodes} \cap \mathrm{ran}\,\textit{output\_nodes} = \emptyset \\
\mathrm{dom}\,\textit{sources} = (\mathrm{dom}\,\textit{nodes} \setminus \mathrm{ran}\,\textit{output\_nodes}) \\
\textit{sinks} = (\lambda\, n : \mathrm{dom}\,\textit{sinks} \bullet \\
\qquad \{n' : \mathrm{dom}\,\textit{sources} \mid n \in \mathrm{ran}(\textit{sources}(n'))\}) \\
\hline
\end{array}
$$

Figure 6. Basic neural network

$$
\begin{array}{|l}
\hline
\textit{layered\_network}[\textit{NODE}] \\
\textit{network}[\textit{NODE}] \\
\textit{layers} : \mathrm{seq}_1(\mathbb{P}_1\,\textit{NODEID}) \\
\hline
\#\textit{layers} \geq 2 \\
\textit{layers}\ \text{partition}\ \mathrm{dom}(\textit{nodes}) \\
\mathrm{ran}\,\textit{input\_nodes} = \textit{layers}(1) \\
\mathrm{ran}\,\textit{output\_nodes} = \textit{layers}(\#\textit{layers}) \\
\forall\, i : 2 \mathinner{.\,.} \#\textit{layers} \bullet \bigcup \textit{sinks}(\!|\ \textit{layers}(i-1)\ |\!) = \textit{layers}(i) \\
\hline
\end{array}
$$

Figure 7. Layered neural network

type of component which is a parameter of a given network. The basic neural network specification factors out all the common properties neural networks share. A definition of the sets of possible nodeids, weights and values that a basic network can use is needed before we can specify our basic network requirements for the developer. This is done using the given sets

$$[\textit{NODE}, \textit{VALUE}, \textit{NODEID}, \textit{WEIGHT}]$$

With these definitions made we can specify a basic network as in figure 6. A major advantage of our specification approach is the ability to add constraints to previously defined schemas to create new schemas. Indeed this is how the layered network of figure 7 is defined, by augmenting the network[NODE] schema with additional information.

The properties that the specification in figure 7 implies are:

- There exists only one way to partition a network into layers. In other words the layer function produces a unique function mapping NODEIDs to LAYERs. This mapping must be defined for every layered network. This is implicit in figure 7, where given $x \in \mathrm{dom}\,\textit{nodes}$, layer is determined by $\mu\, i \bullet x \in \textit{layers}(i)$.
- Every node belongs to one and only one layer.
- The nodes in layer i may only have sources from layer $i-1$.
- The set of nodes in the input layer is the set of nodes in layer 1.
- The set of nodes in the output layer is the set of nodes in the last layer, given by #outnodes.
- There are at least two layers in a layered network.

This concludes the specifications and now we move onto the subject of the verification and validation of these specifications.

5. Verification & Validation Methods

The production of prototype implementations is potentially a very convincing way to explore the behaviour of a specification with its eventual user, and it is in that combination that we suggest formal specification and prototyping offer the most synergy. [22]

Our approach to verifying neural network models is through constructing prototype implementations. Prototyping of the formal model allows for the specified properties and the resulting behaviour to be observed. Confidence in the implementation of the prototype and the model is gained by running appropriate test cases and comparing the results with what is expected. There are two issues that must be addressed when validating formal models using a prototype. Firstly, the prototype must be a faithful representation of the model to allow the relevant parties to test the requirements through the prototype. Secondly, the final delivered system must be a correct implementation of the model and behave in the same way as the prototype. Without this it is possible to fail to meet the expectations of the user for the system.

The use of prototypes to gain confidence in the correctness of formal models has been previously proposed for software and hardware development. Meseguer and Preece advocate the use of formal specifications to verify and validate knowledge-based systems [19]. The relevance here, is that, neural networks and knowledge-based systems are similar techniques, both used to solve complex, poorly understood problems which are derived from data. Meseguer and Preece use successively more detailed models, starting with a conceptual model then a formal specification and finally an implemented knowledge-based system. The formal specification is internally verified and validated against the conceptual model and then transformed into the KBS. Decomposing the verification and validation tasks makes them easier to perform and provides a path from conceptual model to implementation.

The automatic transformation of a Z specification into a prototype, using a sound translation scheme, results in a high degree of confidence in the prototype. However, creating a method for automatically transforming a Z specification into a program which results in a sound prototype is a hard problem. The approach of Kazmierczak et. al. in [17, 41] to animating Z specifications gives an analysis method based on modes for determining executability of Z specifications but can only be applied to a subset of the Z language. The problem with their translation scheme is that it was designed to translate an arbitrary Z specification and so cannot effectively generate prototypes for certain specifications. In our scheme we only need to translate a single specification and do not require a completely automatic process. Kazmierczak et. al. use a logic programming approach, and Mercury in particular, to implement their prototypes. Advantages in using a logic programming language for prototyping purposes, have been identified in [40]. These advantages in combination with the similarity of the declarative semantics of logic programming and Z (which make a translation scheme simple and intuitive), the development of Mercury within the department and our familiarity with the language made it an obvious choice.

The model is verified by generating a prototype and exploring its behaviour. The prototype can be applied to solve known problems as test cases and the responses observed. One of the test cases used to verify the prototype was the XOR problem. For example, to verify the recall operation, the results of recall for a defined XOR network were compared with the results published by Gibb and Hamey in [12]. The results obtained from the prototype were identical and instilled a higher degree of confidence that both the model and prototype were correct. Solving more problems will increase the confidence in both the prototype and the formal model and has been carried out in [32].

The most significant advantage of having a prototype derived from the formal model is that it provides an additional executable model against which developers can benchmark their own implementations. The separation of the generic developer requirements and the specific user requirements is reflected in the prototype. The developer can down-load the prototype and instantiate it with any specific network, execute the resulting prototype and compare the results with those of their own implementation. However, we can only be confident of the results generated by the prototype if it is sound with respect to the formal model. Another advantage is that the process of translating the formal model into a prototype is valuable for gaining insight into design and implementation issues.

6 Conclusions

In this paper we have identified the major engineering problems associated with neural network development, the lack of repeatability and predictability of the development process.

To counteract these problems we have proposed a development process which incorporates specialised methods to address the issues specific to neural network development. The development process and the specific methods have been applied to a number of examples.

We present an extendible method for specifying neural network software and methods for the verification and validation of these specifications. The verification and validation method employs prototypes and benchmarking.

We are still experimenting with the method but it has been trialed in a number of examples. We have trialed this process internally in [31] and with a group of independent developers in [32]. Ongoing research into appropriate data analysis methods for specific problems, predicting development outcomes based on empirical evidence and creating a sound prototype based on the formal specification is underway.

References

[1] 9th International Conference of Z Users. *On the Use of Formal Specifications in the Design and Simulation of Artificial Neural Networks*. Springer, Sept. 1995.

[2] H. Attias. Independent factor analysis. *Neural Computation*, 11(4):803–851, 1999.

[3] L. M. Barroca and J. A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, Dec. 1992.

[4] J. P. Bowen and M. G. Hinchey. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, Aug. 1994.

[5] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[6] C. Brodley and P. Smyth. The process of applying machine learning algorithms. *Proceedings of Workshop on Applying Machine Learning in Practice at IMLC-95*, 1995.

[7] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 4(20), 1987.

[8] M. D. and M. R. Hierarchical evolution of neural networks. *Proceedings of the 1998 IEEE Conference on Evolutionary Computation*, 1998.

[9] G. Dorffner. Formal neural network specification and its implications on standardization. *Computer Standards & Interfaces*, 16:205, 1994.

[10] E. Fiesler. Neural network classification and formalization. *Computer Standards & Interfaces*, 16:231, 1994.

[11] B. Fortner. An overview of data dimensions and visualization. *Neurovest Journal*, 4(2):14, Mar. 1996.

[12] J. Gibb and L. Hamey. A comparison of back propagation implementations. Technical report, Department of Computing, Macquarie University, Sept 1995.

[13] R. Golden. A unified framework for connectionist systems. *Biological Cybernetics*, 59(120):109, 1988.

[14] J. Hampton and R. Caldwell. Visualization tools for complexity and finance (or looking before we leap. *Neurovest Journal*, 4(2):7, Mar. 1996.

[15] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, Feb. 1995.

[16] C. W. Johnson. Using Z to support the design of interactive safety-critical systems. *IEE/BCS Software Engineering Journal*, 10(2):49–60, Mar. 1995.

[17] E. Kazmierczak, P. Dart, M. Winikoff, and L. Sterling. Verifying requirements through mathematical modeling and animation. *International Journal of Knowledge Engineering and Software Engineering*, 2000.

[18] J. A. McDermid. Formal methods: Use and relevance for the development of safety critical systems. In P. A. Bennett, editor, *Safety Aspects of Computer Control*. Butterworth-Heinemann, Oxford, UK, 1993.

[19] P. Meseguer and A. Preece. Assessing the role of formal specifications in verification and validation of knowledge-based systems. In *Proceedings of the Third International Conference on Achieving Quality in Software, Edited by S. Bologna and G. Bucci, London, 1996. Chapman & Hall.*, pages 317–328, 1996.

[20] T. P. Minka. Automatic choice of dimensionality for PCA. In *NIPS*, pages 598–604, 2000.

[21] M. P. and M. R. Culling and teaching in neuro-evolution. *Proceedings of the 7th International Conference on Genetic Algorithms*, 1997.

[22] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specifications and Z.* Prentice Hall, 1991.

[23] R. Pressman. *Software Engineering - A Practitioner's Approach.* McGraw Hill, 1992.

[24] J. W. Prior. Eugenic evolution for combinatorial optimization. Technical Report AI98-268, The University of Texas at Austin, 1, 1998.

[25] A. P. Ravn, H. Rischel, and V. Stavridou. Provably correct safety critical software. In *Proc. IFAC Safety of Computer Controlled Systems 1990 (SAFECOMP'90).* Pergamon Press, 1990.

[26] D. Rodvold. A software development process model for artificial neural networks in critical applications. *Proceedings of the 1999 Joint Conference on Neural Networks (IJCNN'99)*, July 1999.

[27] Rukhin. Testing randomness: A suite of statistical procedures. *TPRBAU: SIAM Journal on Theory of Probability and Its Applications*, 45, 2000.

[28] P. J. Sallis, G. Tate, and S. G. MacDonell. *Software Engineering*. Addison-Wesley, 1995.

[29] D. T. Sannella and A. Tarlecki. Extended ml: An institution-independent framework for formal program development. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389. Springer-Verlag, September 1985.

[30] S. Schach. *Classical and Object-Oriented Software Engineering*. McGraw Hill, 1999.

[31] A. Senyard. *Toward the Software Engineering of Neural Networks*. PhD thesis, Computer Science and Software Engineering, FEB 2003.

[32] A. Senyard and E. Kazmierczak. Using an ensemble of neural networks for handwritten mathematical formula recognition. In *To Appear in Proceedings of Engineering Applications of Neural Networks)*, 2003.

[33] L. Smith. A framework for neural net specification. *IEEE Transactions on Software Engineering*, 18(7):601, 1992.

[34] L. Smith. Using a framework to specify a network of temporal neurons. Technical report, University of Stirling, 1996.

[35] I. Sommerville. *Software Engineering*. Addison-Wesley, third edition, 1989.

[36] J. Spivey. *The Z Notation: A Reference Manual. 2nd edition*. Prentice Hall, New York, 1992.

[37] M. Thomason. An introduction to nonstationary analysis and financial time series preprocessing. *Neurovest Journal*, 4(4):31, July 1996.

[38] M. Thomason. Principal components analysis for neural network input variable reduction and financial forecasting (part 2). *Neurovest Journal*, 4(2):30, Mar. 1996.

[39] H. van Vliet. *Software Engineering*. Wiley, 2000.

[40] B. A. Wichmann, editor. *Software in Safety-Related Systems (IEE/BCS Joint Study Report).* John Wiley and Sons, 1992.

[41] M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the Australasian Computer Science Conference*, Feb. 1998.

[42] J. B. Wordsworth, editor. *Software Development with Z - A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1992.

[43] B. S. Yandell. *Practical Data Analysis for Designed Experiments*. Chapman & Hall, 1997.

[44] Yao, Williams, and Riessen. Pepnet: Parallel evolutionary programming for constructing artificial neural networks. *Evolutionary Programming VI*, 1213, 1997.

[45] X. Yao and Y. Liu. A population-based learning algorithm which learns both architectures and weights of neural networks. *Proceedings of ICYCS'95 Workshop on Soft Computing*, July 1995.