2 Executive Summary

The goal of this project was to develop a *domain specific language* (DSL) that could be used to build, train, and test *artificial neural networks* (ANNs). The DSL allows someone who is building an ANN to implement it in a language that focuses on the problem domain. This provides several benefits over using a general-purpose language:

- Ease of writing code
- Speed of creating the solution
- Clarity of the solution

There are many existing libraries available to assist with building an ANN, but there currently did not exist a DSL. This project developed the DSL in such a way as to allow any library to be used by the DSL to implement the ANN (as long as the appropriate adapter for the library is supplied). The DSL was developed using the Scrum agile process.
The DSL was evaluated using the DSL to solve real-world problems with data from the UCI repository, by creating a new Tic Tac Toe playing agent in some existing software and by informal feedback from users of existing ANN libraries. The result of the evaluation showed that the DSL was a practical and useful tool capable of real world use. The evaluation also showed where the next stage of development should focus.

The key achievements of the project are:

- The development and release of the DSL for ANNs, called *Neural* (see sections 7 & 8)
- The creation of a design pattern for providing a single interface (an application programming interface or API) over multiple libraries, each with different APIs (see section 9.5)
- Establishing an open source project for *Neural*, that will exist beyond the lifetime of this product (see section 10.2)

3 Contents

1 Acknowledgements 1

2 Executive Summary 2

3 Contents 3

4 Introduction 5

4.1 Document Structure 5

4.2 Conventions & Terminology 5

5 Goals 6

5.1 Aims 6

5.2 Objectives 6

5.3 Deliverables 6

5.3.1 The Domain Specific Language 6

5.3.2 An Available Resource 6

5.4 Added Value 6

6 Background 7

6.1 Artificial Neural Networks 7

6.1.1 Fundamentals 7

6.1.2 Learning in Artificial Neural Networks 11

6.1.3 Developing an Artificial Neural Network 14

6.1.4 Other Types of Artificial Neural Networks 15

6.1.5 Summary 16

6.2 A Survey Of Current Tools 16

6.2.1 Introduction 16

6.2.2 Why Use Existing Libraries & Tools? 16

6.2.3 Libraries for Artificial Neural Networks 16

6.2.4 Tools for Implementing the Domain Specific Language 19

6.2.5 Summary 19

6.3 Domain Specific Languages 19

6.3.1 Introduction 19

6.3.2 What Is A Domain Specific Language? 19

6.3.3 Why Use a Domain Specific Language? 20

6.3.4 Building a DSL 21

6.3.5 What Do We Want To Build With The DSL? 22

6.3.6 Summary 23

6.4 Related Work 24

7 Designing and Implementing the DSL 25

7.1 The Approach 25

7.1.1 Test Driven Development 26

7.1.2 Tools 27

7.2 The Development Story 27

7.3 The Neural Language 30

7.3.1 Language Definition 31

7.3.2 Network Definition 31

7.3.3 Activation Definition 32

7.3.4 Training 32

7.3.5 Testing 33

7.3.6 A Full Example 33

7.4 The Architecture of Neural 34

7.4.1 The Neural Platform *34*
7.4.2 The Neural API *35*
7.4.3 Integrating Underlying Libraries *36*
7.5 Summary 36

8 Results 37
8.1 Using Neural 37
8.1.1 Create a network to use programmatically *37*
8.1.2 Create and train a network to use programmatically *37*
8.1.3 Create, train, and test to use programmatically *38*
8.1.4 Command Line use *38*
8.1.5 Errors *39*
8.1.6 Changing the ANN library *40*
8.2 The Scope of Neural 40
8.3 Extending Neural 41
8.3.1 Extending The ANN Library Coverage *41*
8.3.2 Implementing New Underlying Libraries *43*
8.4 Summary 43

9 Analysis & Evaluation 44
9.1 Examples From Neuroph/UCI 44
9.2 Building An Agent for Tic Tac Toe 46
9.3 Feedback from users 47
9.4 What Should Be Changed or Added 47
9.5 Neural As A Design Pattern 48
9.6 Lessons Learned 49
9.6.1 Do not underestimate working with libraries *49*
9.6.2 No battle plan survives contact with the enemy[29] *50*
9.6.3 The Curse Of Generality *50*
9.6.4 Benefits of TDD *51*
9.7 Summary 51

10 Summary 52
10.1 Achieving The Goals 52
10.2 Future Plans 53

11 References 54

12 Web Resources 56
12.1 ANN Libraries 56
12.2 Language Resources 56
12.3 Other 56

13 Abbreviations 57

14 Appendix 58
14.1 Neural API 58
14.2 Tic Tac Toe Agent 58
14.3 A Very Short Summary of Scrum 61
14.3.1 Basic Concept *61*
14.3.2 User Stories *61*
14.3.3 Backlogs *61*
14.3.4 Estimation & Velocity *61*
14.3.5 Burndown Charts *61*
14.3.6 The Process *62*
14.3.7 Summary *62*

4 Introduction

The aim of this project is to design and implement a domain specific language (DSL) for researching, creating, and using artificial neural networks (ANNs). This language will be used for quick and easy implementation of ANNs by the domain experts (for example, scientists). It will also allow the design and implementation of new artificial neural network algorithms to be described concisely and precisely.

4.1 Document Structure

This document is laid out in the following key parts:

- **Goals**: Defining the aims and objectives for the project
- **Background**: Giving the reader some background on ANNs & DSLs
- **Designing and Implementing The DSL**: Describing the development of the DSL, and the code created
- **Results**: Showing the scope of what was created, together with a description of DSL, and how the DSL should be used
- **Analysis & Evaluation**: Evaluation of the DSL with real world examples, and actual users, together with an analysis of the project.

4.2 Conventions & Terminology

Within this document, code fragments are presented in the following form:

```
public void helloWorld();
```

These are left inline to preserve the narrative.
Where full examples, or larger fragments are needed a figure is used, such as:

```
public class HelloWorld {
          public void hello() {
                    System.out.println("Hello world!");
          }
}
```

Figure 1 Example

5 Goals

5.1 Aims

The aim of this project is to design and implement a domain specific language for researching, creating, and using artificial neural networks. This language will allow non-software engineers to quickly and easily implement artificial neural networks. It will also allow the design and implementation of new artificial neural network algorithms to be described concisely and precisely.

5.2 Objectives

The following objectives will be met during the course of the project:

- Review the current state of software and tools for neural networks specifically and machine learning in general
- Evaluate the needs for a domain specific language
- Design and implement the domain specific language
- Test the language against real world problems by implementing artificial neural networks
- Evaluation of the language by it's potential users
- Make the language available for use by others

5.3 Deliverables

5.3.1 The Domain Specific Language

The exact scope of the language is to be decided during the project, dependent on the outcome of the evaluation phase. However, at minimum, it will be in the form of a declarative language capable of expressing the design of an artificial neural network in a concise and precise manner. Optionally it will also provide the ability to define experiments to both train and use the artificial neural network. A further optional deliverable is to generalise the language such that it can be extended to other areas of machine learning, such as evolutionary algorithms or reinforcement learning.

5.3.2 An Available Resource

Just creating the language is not enough, it needs to be available for others to use. The implementation of the language will be open sourced and hosted online using a service such as SourceForge, or GitHub. Furthermore a plan will be put in place for maintenance and further development of the language.
It is also the intention to have a paper describing the language published so that it can be found, used and referred to in future research.

5.4 Added Value

This project aims to fill two needs. Firstly to provide a platform for implementing and using artificial neural networks, without focussing on the software engineering. Secondly to provide researchers with a tool that allows them to describe networks and algorithms in a concise and precise manner, something that is unavailable at this moment.
This project provides the base for developing a standard base of domain specific languages that fill similar needs across a range of computer science disciplines.
The value of this project will be shown by the interest and uptake for others within the scientific community.

6 Background

6.1 Artificial Neural Networks

This section gives an overview of artificial neural networks from an end user's point of view. It aims for a broad coverage, rather than narrowing down into any one area.
The key goal is to be able to understand artificial neural networks enough so that their use and implementation can be considered and a domain specific language can be designed.
Initially we focus on the fundamental concepts of artificial neural networks, and proceed to look at some more complex networks. Finally we will consider the process of implementing an artificial neural network.
This section is mostly sourced from Haykin[1], and it should be assumed that the material originates from this source, unless otherwise referenced. It is also assumed that the reader has some familiarity with artificial neural networks already.

6.1.1 Fundamentals

Artificial neural networks (or ANNs) are computer programs inspired by biological nervous systems, or specifically, brains. A brain works completely differently to a computer processor, and to quote Haykins[1], brains are a "*highly complex, nonlinear, and parallel computer*". An ANN is effectively a software simulation of biological hardware. Mostly we are not too interested in retaining a true simulation of the biology and are only interested in using the concepts for computational purposes. Section 0 discusses the relationship between ANNs and Neuroscience.
While a computer works by executing a set of instructions for loading, manipulating and storing numbers in a memory, an ANN works as a network of nodes with each taking input and under various conditions, producing output (for example if the input reaches a certain threshold). ANNs are inherently learning machines, that is, they improve their performance in a certain task over time, with experience. They achieve this by modifying the strengths of the connections between the nodes.
An ANN can be proved to be Turing complete[2], that is, it is capable of performing any computable algorithm. This means it can perform any task a normal computer processor can, however the tasks for which it is suited may be different, and some of these are examined further in section 0.

A Biological Neuron

A neuron is a single cell found in the nervous systems and is the fundamental unit from which brains are built. It is also the fundamental unit of computation in the brain. Although there are many types of neurons, a typical neuron can be seen in Figure 2 and its behaviour described as following:

1. The neuron receives an electrical current through the dendrites
2. When the electrical potential in the cell passes a particular threshold the neuron fires
3. The neuron transmits an electrical current through it's axons
4. The axons of one neuron are connected to other neuron's dendrites through synapses, which form a chemical bridge between the two. The synaptic bridge can be regulated by chemicals in the brain know as neurotransmitters.
5. By analogy to software programs, the dendrites can be thought of as input, the axons as output, and the neurotransmitters as functioning as some sort of global control over behaviours.

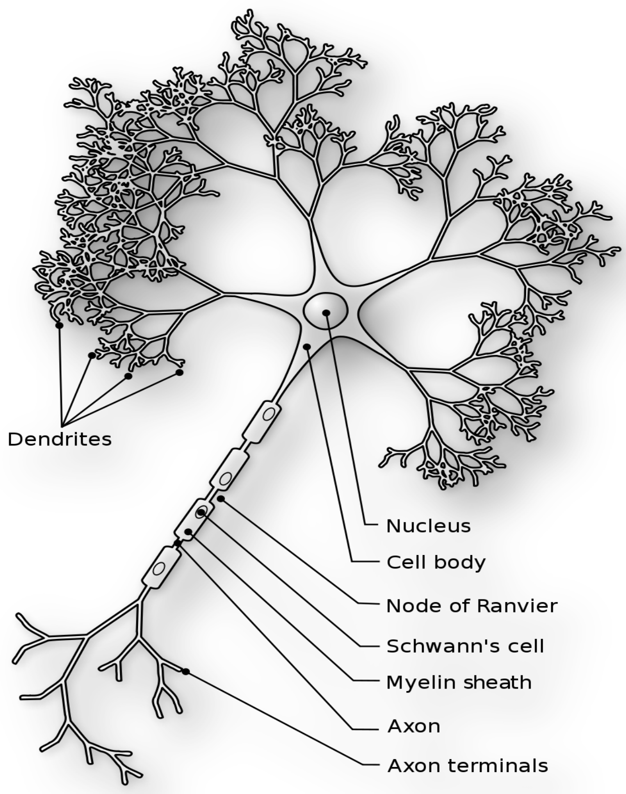

Figure 2 A typical neuron found in a brain (image from WikiMedia Creative Commons)

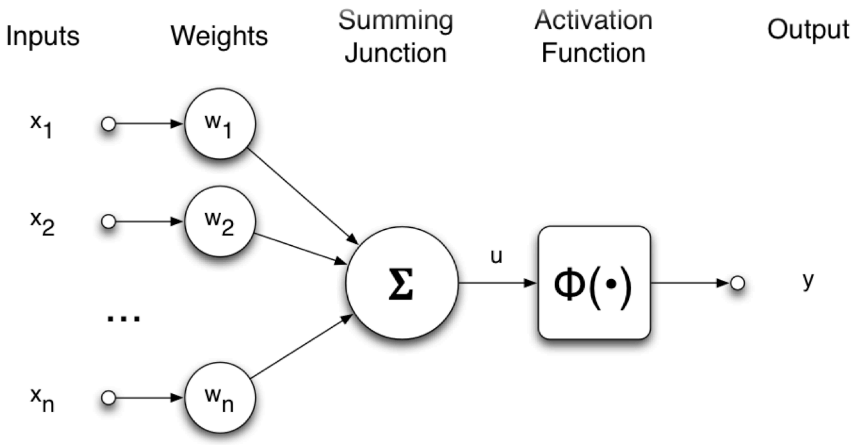

Figure 3 A basic model of an artificial neuron

A Basic Model of an Artificial Neuron And An Artificial Neural Network

The model shown in Figure 3 is used as a basis for most neurons in ANNs. Each input signal x is modulated by a synaptic weight w, and then the inputs summed so that the input to a neuron is:

$$u = \sum_{i=1}^{n} x_i w_i$$

The activation function is then applied to generate the output:

$$y = \Phi(u)$$

A simple example of an activation function is a threshold function:

$$\Phi(u) = \begin{cases} 1 \; if \; u > t \\ 0 \; if \; u \leq t \end{cases}$$

where a neuron would provide an output signal of 1 if the weighted sum of the inputs exceeds some threshold *t.*
With this basic model, each neuron can be fully described by the following parameters:

- The weighting of each synapse (w)
- The activation function (Φ)

An artificial neural network is then created from a set of neurons with connections between them. Connections can be from input to the network, to output from the network, or between the inputs and outputs of neurons within the network. The network can then be fully described by the following parameters:

- Input connections
- Connection between neurons
- Output connections

Activation Functions

The activation function defines the output of a neuron given a particular input. Haykins[1] describes three basic types:

- **Threshold**: where the output is zero until an input threshold is reached
- **Piecewise linear**: where output is linearly dependent on the input until a maximum is reached
- **Sigmoidal**: where there is a non-linear relationship, an example of which might be the $tanh(u)$ function

However from a software engineer's point of view, any function could be used that produces output given an input (obviously, with varying degrees of usefulness). An example of a more complex function might be a stochastic (probabilistic) function designed to simulate the noisiness of a real neural network.

Types of Networks

There are three basic types of ANN architecture:

Single-layer Feed Forward

In a layered architecture, the neurons form layers whose output maps to the input of the next layer, and never vice versa (hence *feed forward*). In the single-layer feed forward network there is only one layer of neurons providing output.

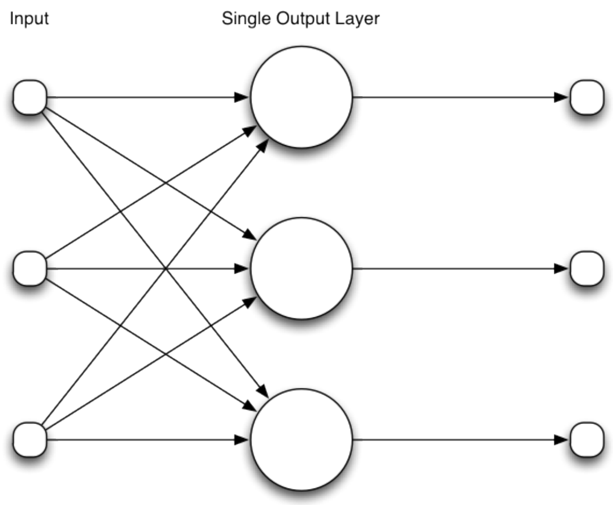

Figure 4 A single-layer feed forward network

Multi-Layered Feed Forward

A multi-layered architecture consists of one or more hidden layers. The hidden layers allow extra dimensions of neuron interactions, and increase the capabilities of the network.

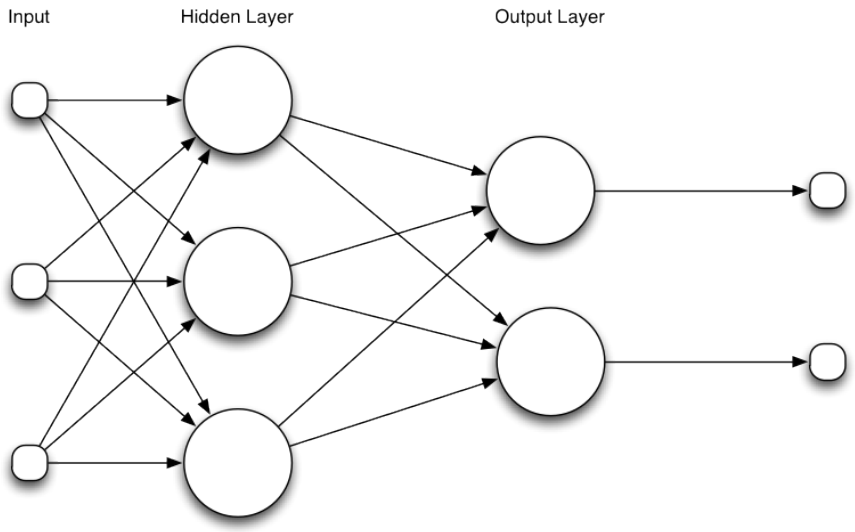

Figure 5 A multi-layer feed forward network

Recurrent Network

A recurrent network is one is which has one or more feedback loops. This allows the network have a dynamic state. Typically these networks involve the use of unit delay elements, which can result in nonlinear behaviour.

The Relationship With Neuroscience

Since an artificial neural network is an abstract model of a biologic network, it also provides benefits to neuroscience. Neuroscience also uses artificial neural network models but in a different way.
I have classified the two differently based on their goals:

1. Neuroscience Model: where the goal is to simulate some aspect of biology in order to understand it
2. Artificial Neural Network: where the goal is to perform some task (with the ANN as just a means to an end)

The first focuses on representing some biological truth, while the second is freer to abstract, or even to have abilities/parameters that are not present in biology.
Given the similarities between both, it is worth keeping both options open when considering software for artificial neural networks.

6.1.2 Learning in Artificial Neural Networks

We have talked about how an ANN operates, based on the weights of the synapses. But how do we find those weights? The network must start with some default (perhaps random) set of weights and changes are made through training to successively improve the weights. We call this learning.
The following sections give an overview of the learning algorithms that may be used for an ANN.

Hebbian

The Hebbian learning rule is one of the oldest learning rules for ANNs. It can be

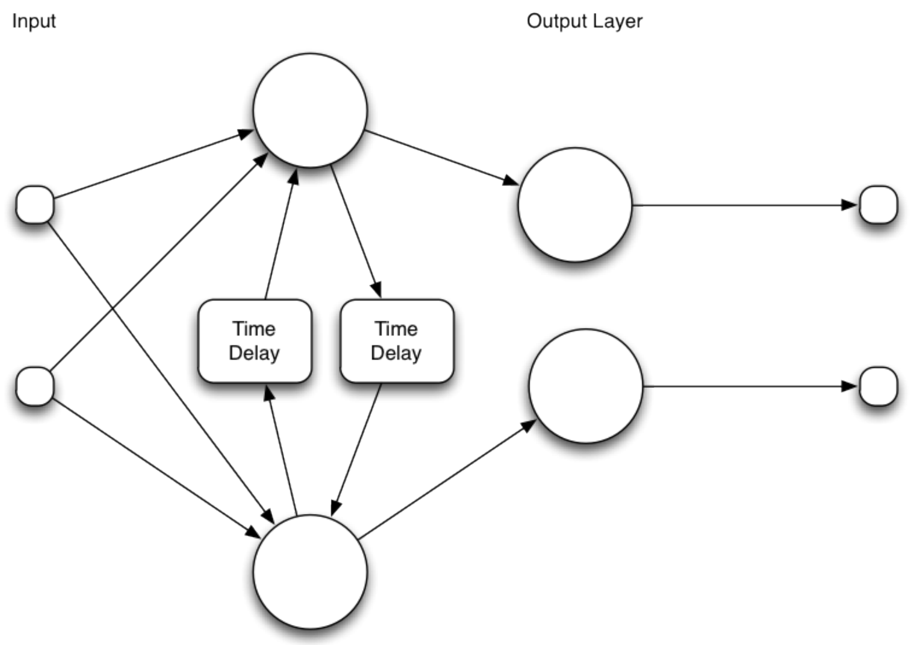

Figure 6 A recurrent network

characterised in the following two stages applied to each input into the neuron:

1. If both input and output to the neuron are active then increase the weighting
2. If just one of the input or output is active then decrease the weight

In it's simplest, this can be visualised mathematically with the following:

$$\Delta w_i = \eta \, y_i x_i$$

where Δw_i is the change to the weight, y_i and x_i the output and input to a neuron, and η a constant representing the learning rate.

Back Propagation

Back propagation is a common method of learning. Given a supervised training set, the network input is calculated and then values fed forward through the network to provide the output values. The deltas between the actual and expected output are *back propagated* through the whole network. Each weight can then be updated according to the learning rate.

Boltzmann

The Boltzmann rule is a stochastic learning rule based on ideas from statistical mechanics. A Boltzmann machine is an ANN that uses the Boltzmann learning rule.

$$E = -\frac{1}{2}\sum_{j}\sum_{k} w_{jk}x_j x_k$$

Where x_j is the state of neuron *j* and w_{jk} is the synaptic weight between *i* and j.

The learning rule operates by picking a neuron at random and flipping it with probability:

$$P(x_k \rightarrow -x_k) = \frac{1}{1 + e^{-\Delta E_k / T}}$$

where ΔE_k is the energy change that would result from the flip, and T represent the temperature, in this this case being a tuneable parameter of the system.

A Boltzmann machine is a specific instance of a *stochastic machine*.

Supervised versus Unsupervised Training

There are two forms of training that can occur with artificial neural networks, or indeed with machine learning in general. These are 'supervised training' and 'unsupervised training'.

Supervised training consists of using training data that corresponds to a labelled output set. An example of this might be classification of pictures into animals, where the training set would consist of pictures of animals together with their correct classification.

In unsupervised training there is no knowledge of correct outputs. The network must derive it's own analysis of the input data. An example of this might be a network that makes time series predictions on an input signal.

Cross Validation

Ideally training should happen with a large training data set, and then the resultant trained network can operate on a statistically similar test data set, before being used "for real". The purpose of the test data set is to measure the accuracy of the trained

network. Since we want to train on as much data as possible, using a separate test set may be undesirable.
Cross validation is a technique that maximises our training data while providing a useful test data set. It works in the following way:

1. The data is partitioned into random subsets
2. For each subset X the ANN is trained on every subset apart from X and then tested using X

Tasks for Artificial Neural Networks
Artificial neural network are capable of many things (indeed, since they are Turing complete they can perform any computable operation). It is possible to reduce the problems for which they are efficient down to a few broad categories. Haykins reduces it down to 5 which are described below plus *Beamforming* which I consider part of *Data Processing*. There are, of course, other applications across a broad range of problem domains.

Pattern Association
Pattern association is the task where after being supplied with one form of input, the network will provide an associated output. In *auto-association* the output pattern is the same as the input pattern, and the input pattern may be a noisy or partial pattern. In *hetero-association* the input and output patterns are different, and this could be thought of as a key-value pair.
Pattern association is typically unsupervised learning with the training being the storage of patterns, and the testing being the recall.

Pattern Recognition
Pattern recognition is the task where after being supplied with some input, the network will classify it into one or more types. This is typically a supervised training where the network is initially trained with examples of each class, and then during testing presented with a new pattern it must classify correctly.
Although Haykins called this *Pattern Recognition,* I prefer the term *Classification* which seems more prevalent through the machine learning literature.

Function Approximation
The output of an artificial neural network can be considered a nonlinear function of the input:

$$\boldsymbol{d} = \boldsymbol{f}(\boldsymbol{x})$$

where **d** is the output, given input **x**.
For a function approximation task, the ideal function **F** must be approximated, but is not known. Instead training data supplying examples of **d** and **x** is used to allow the network to learn an approximation to **F**, such that:

$$\|\boldsymbol{F}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x})\| < \varepsilon$$

where ε is a small positive number.

Control Systems
Controlling a physical system (or it may be an external system of any kind) is another task type for artificial neural networks. In this task a system would take control signals and provide some form of feedback in return. The ANN compares the feedback signal with the reference signal to control the system in order to keep it in line with the required behaviour (as described by the reference signal).

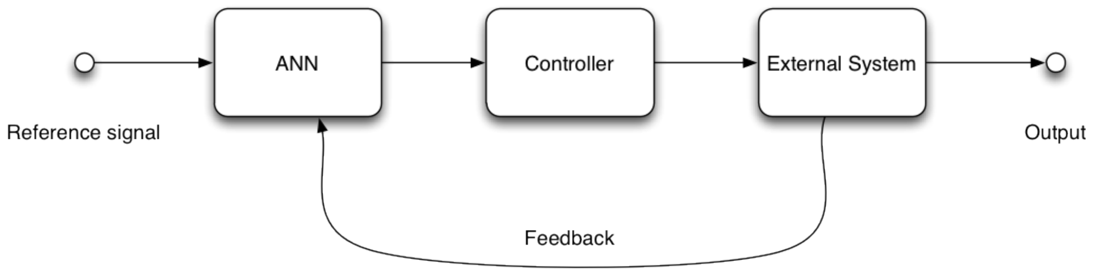

Figure 7 Typical architecture for an artificial neural network controlling an external system

Data Processing
Haykins classes this as *Filtering*, however I prefer to think about filtering as a subset of *Data Processing*. The difference is just the terminology, but I think the original use of the term 'filtering' was misleading. Data Processing is any task in which the network receives some input data and performs some kind of analysis or manipulation of it. This is a wide-ranging class of task, and I think Haykins is using it as a bit of a catch-all for everything left at the end.
Subcategories of this task include:

- Filtering: extracting signals from within noisy data or from other masking signals
- Compression: reducing the size of a data set while maintaining the amount of information
- Prediction: receiving a sequence on input data and predicting future data based on this

6.1.3 Developing an Artificial Neural Network

Design
According to Senyard[3] two key design decisions typically influence the development of an ANN. These decisions are not software design decisions, instead they are problem domain decisions driven purely by the particular problem to be solved:

- Which network architecture to use (e.g. multi-layer feed forward)
- Which learning algorithm to use

Following the key design decisions the next set of decisions are typically[3,4]:

- Parameters controlling the structure of the network (e.g. number of layers)
- Learning algorithm parameters
- Training and testing strategy
- Error levels

The results of the decisions above cover most of the needed parameters to fully implement an ANN.

Software Development of an Artificial Neural Network
Rodvolds[4] suggests an ANN development process that has the following steps:

1. Network requirements, goals, & constraints
2. Data gathering and pre-processing
3. Training and testing loops
4. Network Deployment
5. Independent testing & verification

where stage 3, training and testing loops, is able to try the following variations:

- Variation of ANN topologies (i.e. architecture)
- Variation of ANN paradigms (i.e. learning algorithms, connection etc.)
- Selection and combination of ANN input neurons

This process shows that the focus here is on the ANN and the data (i.e. the problem domain) and not on software engineering.

6.1.4 Other Types of Artificial Neural Networks

Section 6.1.1 describes several fundamental types of artificial neural networks, however other types of networks exist. This section describes a set of more complex networks and some of their properties that need to be considered when designing an ANN of that type. There are obviously more than these, but these have been selected to give a flavour of what might be encountered.

Radial Basis Function Network

A Radial Basis Function, or RBF, network is an artificial neural network that uses radial basis functions as the activation function for the neuron. The network takes the form of a multi-layered feed forward network, with the hidden layer containing linear radial basis functions. It is typically used for:

- Time series prediction[5]
- Function approximation
- Control Systems[6]

When designing and implementing these networks, additional consideration needs to be given to extra parameters used by the radial basis functions (the centre vectors, and RBF width).

Dynamic Recurrent Networks

Recurrent networks can have feedback that is either local or global. Dynamic recurrent networks are recurrent networks with global feedback. Like other recurrent networks they use unit delays, but will involve feedback into the input.

Although a discussion of their properties is beyond the scope of this paper, the interesting properties from a software point of view are:

- Unit delays
- Feedback into the input layer

Although Haykins positions this as a separate topic from the basic recurrent network (and in terms of mathematical analysis, it is), for our needs it has similar requirements to the basic recurrent network described in section 6.1.1

Committee Machines

A committee machine consists of more than one artificial neural network. Each of these processes the same test data and the results are combined to produce a single output.

Examples of committee machines are:

- Ensemble averaging – otherwise identical networks are initialised differently, trained on the same data set and the output averaged. The result is slightly better than the average of the individual networks[7]
- Boosting – identical networks are trained on different data sets. The results of the test data are combined. The overall result is that the output from the committee machine performs as a much better algorithm than any of the individual machines on their own

Evolutionary Artificial Neural Networks

Evolution is another method of learning. It can be combined with artificial neural networks in three ways[8]:

- Weights
- Connections & architecture
- Learning rules

In each of these cases the information is encoded into a genotype and normal evolutionary algorithms are used. When calculating the fitness of the neural network, it is trained on a training set, then tested as normal, with the difference between the actual and expected outputs forming the fitness.
There are two problems to overcome when implementing an evolutionary ANN[9]:

- Noisiness due to random initial weights can mean the fitness measure is not accurate. This can be overcome by training many networks with different initial weights, but this is computationally expensive.
- Encoding of parameters into a suitable genome (although I consider this to be a problem for any evolutionary algorithm)

6.1.5 Summary
In this section of the paper, we have had a flying overview of artificial neural networks. It has not been necessary to understand all of artificial neural networks at an implementation level, as the intention for this project is to reuse existing ANN libraries. As such developing a domain specific language relies more on the functionality of the ANN libraries. The intention has been to show the range of network architectures available and some of the development decisions that are made during their implementation.

6.2 A Survey Of Current Tools

6.2.1 Introduction
This section looks at existing libraries and tools available or both artificial neural networks and for implementing domain specific languages. Initially we discuss the reasons for reusing such libraries & tools, and then goes on to discuss and compare the libraries and tools available for both ANNs and DSLs. Finally there will be a short discussion of PMML, a mark-up language used to specify ANNs, as well as other machine learning algorithms & models.

6.2.2 Why Use Existing Libraries & Tools?
The focus of this project is about building a domain specific language for artificial neural networks, and not about building a reusable library to implement ANNs. As such it is better to reuse libraries if they are available.
The reuse of libraries is good because:

- Reuse of existing code: reducing the time taken to implement a system. This allows time to be spent on the real problems instead of implementing things that have been done before.
- Greater stability and reduced risk: all the code in the library has already been through entire development cycles, and should be stable & tested.
- Choice of implementation: if designed properly, the library can be swapped for another, providing a choice of implementation. Different implementations with different attributes can be used as required.

6.2.3 Libraries for Artificial Neural Networks
This section discusses five existing libraries that can be used for implementing artificial neural networks. It then compares the features of each.

Listing of Artificial Neural Network Libraries
Links to the websites of all these libraries can be found in Section 12.1.

PyBrain
PyBrain is an open source machine-learning library written in the Python language. It focuses on ANNs, reinforcement learning, and optimisation. It aims to be an easy-to-use and modular framework, suitable for both students and researchers.

PyNN
PyNN is a library used to create neuronal network models (biological) written in Python. It provides the ability to run the models on various backend libraries, all of which are focused on simulating biologic neural networks. PyNN doesn't provide the ease of use

Feature	PyBrain	PyNN	Encog	Neuroph	dANN
ANN Basics	Full	Full	Full	Full	Full
Activation Function Library	Good	Some	Full	Good	Good
Feed Forward Networks	Full	Some	Full	Good	Good
Recurrent Networks	Full	Some	Full	Some	-
Boltzmann Machines	Good	-	Good	Good	-
Self Organising Maps	Good	-	Good	Good	-
Neuroscience Models	-	Good	-	-	Some
Other Network Models	Good	-	Good	Some	Some
Other Machine Learning	Good	-	Some	-	Some
Supervised Training	Good	-	Good	Some	Some
Unsupervised Training	Good	-	Good	Some	Some
Reinforcement Learning	Good	-	Good	Some	Some
Other Learning Methods	Good	Some	Good	-	-
Data Models For Input/Output	-	-	Good	-	-
Optimisation	Some	-	Some	-	Some
Genetic Algorithms	Some	-	Good	-	Good
Testing/Training Environments	Good	-	-	-	-
GPU Execution	-	-	Some	-	-
Visualisation	Some	-	-	-	-

Table 1 A comparison of the main features of several ANN libraries

that PyBrain does, but it does provide a gateway into the biological simulation side of neural networks.

Encog

Encog is a library for artificial neural networks and machine learning. It has two versions, one in Java and the other in C#. Encog has a wide variety of network models, activation functions and learning techniques built into it. Encog also provides the ability to run on GPUs (Graphical Processing Units) through the use of OpenCL[10].

Neuroph

Neuroph is a lightweight framework for building artificial neural networks, written in Java. It provides a small number of powerful base classes for implementing ANNs.

dANN

dANN is a library for artificial intelligence and genetic algorithms, as well as machine learning in general. It is written in Java and boasts the widest range of built-in abilities, however within ANNs it's range may be limited.

Table 1 shows a comparison of the features of interest to this project for each library.

Using Multiple Libraries

Different ANN libraries offer different features. Even libraries that offer less features than other libraries, may outperform them in the tasks that they do. So choosing a library on which to base a solution is difficult, without knowing how that solution will be used.

Fortunately, judicious use of design patterns[11] can ensure that the decision of which library to use is left until run time. Figure 8 shows how the Factory and Adapter design patterns can be used to break the coupling between the DSL and the ANN implementation library. The DSL uses a LibraryAdapterFactory to load an object of type LibraryAdapter. It doesn't know what the concrete type is and doesn't care, the actual type would probably be controlled through a configuration parameter. Each ANN library then has it's own LibraryAdapter to adapt from the interface required by the DSL to how the library is implemented. To add another library to this pattern would only take the implementation of a new LibraryAdapter.

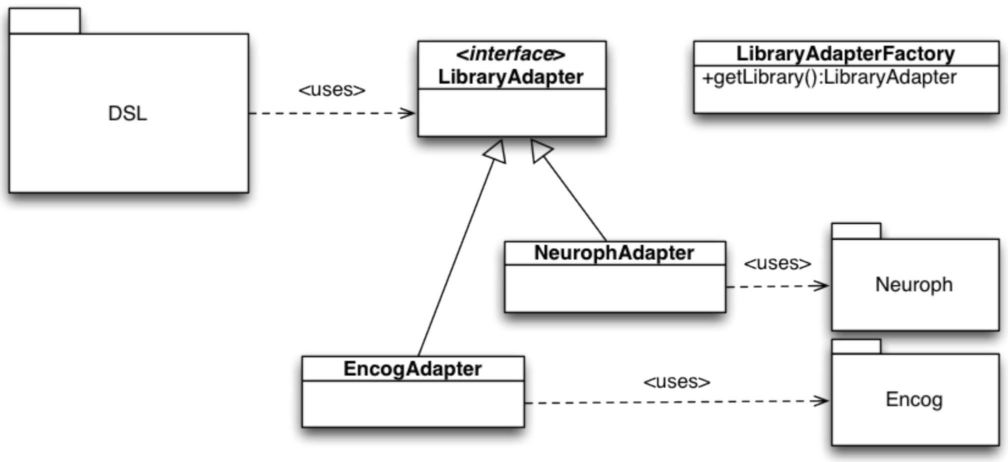

Figure 8 A design for the capability of using multiple ANN libraries

6.2.4 Tools for Implementing the Domain Specific Language

For implementing the DSL there are two main decisions to make:

- What programming language to develop the DSL with
- What support tools to use

Each of the decisions can affect the other.

Language

For the implementation language, the libraries for ANNs constrained the choice to just Java & Python and both of these languages can integrate with libraries written in the other language. Given that I have 13 years experience as a Java software engineer, existing skills weigh heavily in favour of Java.
Equally, looking at the tools available in each language, there is a wider range available in Java than in Python.

Parser

Web URLs for these tools can be found in Section 12.2.
JavaCC (Java Compiler Compiler) is a compiler compiler[12] that generates Java code from a Backus Naur Form[13] (BNF) grammar specification. The code generated forms an interpreter that can read scripts in the target language and execute desired Java code. In the initial generated form there are only stubs for later expansion.
JParsec is a recursive descent parser[14]. It does not use a grammar specification to define the syntax, instead relying on Java objects that are created and combined together.
The end result of using both tools is the same: a Java library that will parse and execute the DSL scripts, so the decision of which to use is a matter of taste.

6.2.5 Summary

We have seen a range of libraries that could be used for building ANNs. They range from simple to extensive, and have various strengths and weaknesses. Since we do not want to commit to a single library, we want to allow the DSL to use a library that can be selected at runtime, and we have seen the design patterns that will allow this.
The implementation language of choice is Java, and the particular tool to be used to implement the DSL has not yet been chosen but the decision will not affect the project significantly.

6.3 Domain Specific Languages

6.3.1 Introduction

This section will describe what a domain specific language (DSL) is, why we should use one, and how it should be implemented. It will go on to discuss the key features of the DSL that will be developed.

6.3.2 What Is A Domain Specific Language?

A domain specific language is a programming language that is designed to be used for programming in a particular problem domain. Normal languages such as Java/C/FORTRAN can be thought of as *general purpose languages* (GPLs), and they provide the abilities to easily work through most computable problems. A DSL instead focuses on making it as easy as possible to solve problems in the specific problem domain for which they are designed. Some examples of DSLs can be seen in Table 2.

Domain Specific Language	Purpose
HTML	Description of the presentation of web pages
SQL	Querying and controlling databases
COBOL	Programming within the business domain
ESDL	Describing evolutionary algorithms

Table 2 Some examples of domain specific languages

DSLs may range from languages such as HTML, which are highly specialised towards their problem domain, and not capable of general computing, to languages such as COBOL, which are arguably general-purpose languages.
Programming languages could be visualised on a horizontal scale[15], with DSLs on the far left and GPL such as C on the right. Although the real properties of languages encompass many more dimensions than this, the analogy is apt. There is no definite border between a GPL and a DSL, even though the extremes are clear.

6.3.3 Why Use a Domain Specific Language?

There are several reasons for using a DSL over writing the code in a GPL[15].
The first is that a DSL provides a higher level of abstraction over the problem that is being solved. This means the code is simpler, and simpler code means less code, something all software engineers strive for.
The second reason is that a DSL provides a large amount of reuse. This comes from the high level of abstraction, which results in less low level code being (or needed to be) written.
Thirdly, because of the high level of abstraction, the DSL is much easier to use for a problem domain practitioner. The practitioner may not even be a software engineer (and this is one reason to develop a DSL), so making it easy for someone to write code is very important.
Finally, the resultant code solution is highly communicable. Because of the high level of abstraction the code is aligned with the problem domain, not general computing, so it is easy for practitioners to communicate about the problem domain using the DSL.
Figure 9 shows two pieces of code, the first in Java, and the second in a hypothetical DSL. Note the difference in readability and clarity of the problem in the second piece of code compared to the GPL version.

```
//In Java
Transaction transaction = transactionManager.begin();
withdraw(transaction, “USD”, 95552000, 1000, true);
deposit(transaction, “USD”, 97893451, 1000, true);
transaction.complete();


//In BankingDSL
transfer 1000 in USD from 95552000 to 97893451
```

Figure 9 Comparison of well factored Java with hypothetical banking DSL

Human & Computer Roles – A Justification For DSLs

What roles do a human and a computer play during the development of a piece of software? At a coarse granular level, a human writes code, and then a computer executes

the code, however that only applies if the human is writing pure machine code. If this were true then our languages would be optimised for the use of the computer.
At a finer granular level:

- Human writes the code
- Computer compiles the code
- Computer executes the code

Now we have a middle step where the computer optimises the code (i.e. compiles) for it's own use. At this point we can start to think about making the human writing code step as easy as possible.
As we progress through the years and higher levels of abstractions in languages, more and more steps in between the human and computer code appear, this gives us greater chance to optimise for the human writing the code.
If we now start to consider solving a problem, rather just writing code we also gain another step at the beginning:

- Human creates problem solution
- Human writes code
- Computer compiles code
- Computer executes code

What is now worth considering is that the human is performing a compilation step. This is very inefficient operation for a human. This is where the DSL enters the field:

- Human creates problem solution in DSL
- Computer compiles DSL to GPL
- Computer compiles GPL
- Computer executes code

The examples above give a clear message that writing code for problem solutions is ultimately a human exercise and compilation, interpretation & execution is an exercise for the computer. The more optimisation & compilation we can get the computer to do, the easier we make it for the end user to create problem solutions, and the better a system we will have.

6.3.4 Building a DSL

Building a DSL is a software engineering exercise. As with all software engineering, there are various methods by which a piece of software can be developed, but most can be categorised into two types of software engineering methods: Waterfall and Agile.
Waterfall is a software engineering method originally proposed[16] in 1970, and although not named *Waterfall* at the time it became the template for later methods. Waterfall is a heavyweight process that takes each step in the development of a software system in order, and focuses on getting every stage correct before proceeding to the next.
Agile software engineering methods were first proposed[17] in 2001[1]. The most common properties of agile software methods are that they are iterative and that they rely heavily on interaction with the end user to ensure what is built is correct.

A Waterfall Method

Mernick[15] talks about several phases:

Decision Phase

Making the decision to use a DSL, and following patterns to help decide whether the use of DSL will add value to system.

[1] The agile manifesto can still be found at http://agilemanifesto.org/ (as of September 2012)

Analysis Phase
Analysis of the problem domain by using domain analysis methods such as FAST[18] to build a hierarchy of features. This hierarchy then becomes a base for the languages.

Design Phase
In this phase the language itself is designed. Decisions such as whether to make it formal or informal, or whether to piggyback off another language or create an entirely new syntax, are made.

Implementation Phase
In this phase the executable language is created. Mernick[15] talks about the various options available for how to implement the language but these are very focused on developing around C (or similar) tool chain.
Another option, not considered by Mernick, and that is repeatedly used, is that of executing the language script as a function/library call within another language. Mernick does consider "interpreters" but only in the sense of stand-alone programs. Examples of such libraries that load and execute scripts are Jython or JRuby, both of which allow other languages to be executed from a Java program.

van Deurson[19] also backs this waterfall approach to development of a DSL with an in-depth analysis phase of the problem domain to construct a feature set that becomes the basis for a the language.

This is a very traditional method and follows a typical Waterfall model of software development. It invests much up-front in analysis and design before implementation begins.

An Agile Method
Freeman describes a different method of developing an embedded DSL. In their case they evolved over a set of iterations, with each iteration using the lessons learnt from the previous one. Their process for each iteration was as follows:

- Examine what works and doesn't work in the current software
- Make appropriate changes (producing a usable DSL)
- Use the DSL (on real problems)

This kind of iterative form of development uses experience instead of analysis to inform design decisions. It approaches a DSL in a more human compatible way than a clinical analysis by making changes and then seeing the impact of those changes upon the users.

Waterfall versus Agile
I disagree with the Waterfall model of development, especially for something as user-focused as a domain specific language. A DSL is typically (and is so in this case) geared towards a practitioner in a problem domain. It should be entirely focused on what works for the end user (see Section 6.3.3), and since end users are human, logical analysis of what works for them is often wrong.

6.3.5 What Do We Want To Build With The DSL?
So far we have covered both artificial neural networks and domain specific languages so it is worth spending some time now discussing what it is we want to build with the domain specific language.

The properties of the DSL can be broken up into two sets of features, the first feature that allows a neural net to be specified, and the second that allows it to be used for training or testing. By breaking the features into these two it can be seen that one could be used without the other.

The key features in the following sections describe broad themes that will need to be considered during development, each key feature will need to be broken down into smaller features to facilitate development.

Definition Key Features

Feature	Description
Activation Function	Define an activation function type (encode an algorithm into a reusable object)
Neuron Definition	Define a neuron type within the network (in terms of it's properties such as it's activation function)
Learning Rule Definition	Define a learning rule type to be used within a network
Network Definition	Defining a single neural network, with properties such as number and type of neurons, learning rules, & architecture such as layers and connections
High Level Architecture	Defining ensembles or committees of networks, different functional groups of networks etc.

Table 3 Key features for the description of an ANN

Train/Test Key Features

Feature	Description
Input Data	Describe how input data is fed into the system (the data source and any manipulation of the data)
Output Data	Describe how output data is fed out of the system (the data destination, and any manipulation of the data)
Defining Test Data	Describe the location and labelling of test data, stopping or convergence criteria
Execution of Training	Execute training runs
Execution of Testing	Execute testing runs

Table 4 Key features for the testing & training of an ANN

6.3.6 Summary

There are good reasons to create and use a domain specific language, and we have seen two approaches to building them. We have also described some of the key features the DSL will have.
Next, we will look at some libraries and tools that will make the job of building the DSL easier.

6.4 Related Work

It is worth looking at a similar DSL to help inform any decisions that needed about our DSL. Evolutionary Systems Description Language[20] (ESDL) is a DSL for specifying an evolutionary algorithm. Figure 8 shows an example of ESDL, the language primitives are shown in uppercase and the variables in lowercase.

ESDL is a simple and straightforward language. The author’s intent was to keep it simple so that it could be used to express the evolutionary algorithm as clearly as possible. In this it succeeds, as can be seen in Figure 10, the evolutionary algorithm is clear.

However, in my opinion, the language is perhaps a little too simple. Although the high level algorithm is clear, there are other important algorithms that are not expressed in this language and merely referenced by name. An example of this is the selection operators, which are defined in Java code and just referred to here (e.g. binary_tournament). To see these algorithms one has to revert to reading Java.

```
FROM random_real SELECT 500 population YIELD population
BEGIN generation_equivalent
    REPEAT 500
          FROM population SELECT 2 parents USING binary_tournament FROM parents SELECT
                    offspring USING crossover(per_pair_rate=0.9),
                    mutate(per_gene_rate=0.01)
          FROM offspring SELECT 1 replacer USING best
          FROM population SELECT 1 replacee , rest USING uniform_shuffle
          YIELD offspring , replacee
          FROM replacer , rest SELECT population
    END REPEAT
    YIELD population
END
```

Figure 10 An example of ESDL, a domain specific language for describing genetic algorithms

7 Designing and Implementing the DSL

This section describes the implementation project that created the DSL. It will describe the approach used, and the rationale behind it, and then the story of the software development. Following this, it will describe the final product (the DSL) and its design and architecture. The project was implemented by 1 person over 2 months, making this a reasonably small piece of software development.[21]

Story No.	Title	Description	Estimate (Story Points)	Priority
1	Load Definition	User can load a script, and be given an object that corresponds to the ANN	2	1
2	Activation Definition	User can define an activation function in a script	8	2
3	Neuron Definition	User can define a neuron in a script	8	3
4	Network Definition	User can define a network in a script (input/output/connections)	16	4
5	Activation Reuse	User can reuse an activation function already defined in the underlying library	2	5
6	Neuron Reuse	User can reuse a neuron from the underlying library	2	6
7	Network reuse	User can reuse a network from the underlying library	2	7
8	Training Definition	User can define a training algorithm in the script	8	8
9	Training Reuse	User can reuse a training algorithm from the underlying library	2	9
10	Simple Input/Output	User can define how input and output data is retrieved	4	10
11	Multiple Backend	User can select which ANN library to use as backend	4	11
12	Define Test Data	User can define test data (for example labels)	4	12
13	Training	User can define training runs in the script and execute	8	13
14	Testing	User can use a trained ANN to test on real data	8	14
15	Input/Output Source	Input & Output can be from/to various sources	8	15
16	Multiple Networks	User can define multiple networks and how they connect	16	16
17	High Level Networks	User can reuse committee machines and other networks found in underlying libs	4	17

Table 5 The set of user stories initially planned for development (i.e. before development commenced)

7.1 The Approach

Section 6.3.4 describes the benefits that an agile process can bring to a software development project, and for those reasons an agile process, Scrum[22] was chosen for this project. While it is not the goal of this thesis to fully describe Scrum, a summary of

Story No.	Title	Description	Estimate (Story Points)	Priority
1	Create XOR	Create a XOR feed forward network using Encog	16	1
2	Basic Training	Allow the network to be trained programmatically	8	2
3	Basic Testing	Allow the network to be tested directly from the DSL	8	3
4	Neuroph	Allow the underlying to be switched between Encog and Neuroph	4	4
5	Hopfield	Allow a Hopfield network to be defined	2	5
6	Test Hopfield	Train and test a Hopfield network	4	6
7	Tanh	Be able to choose the type of activation function	4	7
8	Other Activation	Expand on the range of activation functions for Encog	2	8
9	Training Type	Allow type of training to be selected (add a second type)	4	9
10	Training Parameters	Add the following parameters for training: Epocs, restarts, error.	2	10
11	Training Data	Allow training data to be defined in the script	4	11
12	Testing Data	Allow testing data to be defined in the script	2	12
13	Data Sources	Add the references to data sources to the script (CSV & patterns)	8	13
14	Command line	Run from a command line binary	1	14
15	Add more	Add more activation functions/training/ networks as time permits	-	15

Table 6 The re-planned set of user stories (Note the final story, "Add more" is just a place holder for individual stories that will be added later)

the Scrum process and terminology is included in section 14.3 for the reader who is not familiar with this process.
At the start of the project a product backlog was created and the features created and prioritised. These features were based on the functionality required, that had been discovered during the research review. This product backlog can be seen in Table 5. After a short time of working on the software development it became clear that the initial plan needed revising. Whilst the features that were described were all valid and the priorities reasonable, after working with both the Encog ANN library and JParsec, a new set of features and priorities made more sense.
The initial plan was based on desired language elements. This, I believe, was a mistake and it made more sense to base the features on whole examples. The reason for this is that the individual language elements were not easily testable on their own, whilst working from entire examples were testable. Re-planning in this way allowed Test Driven Development to be used, and resulted in the product backlog shown in Table 6.

7.1.1 Test Driven Development

Test Driven Development (TDD) is a programming process which originates in the practices of the Extreme Programming (XP) process[23]. TDD follows a simple iterative process to write code:

1. Write an automated unit test that fails
2. Write code that makes the test pass

The central philosophy[24] of TDD is that the programmer is not allowed to touch the actual code without an automated test failing. This results in source code being built that has a very high level of automated test coverage. This high level of code coverage then has the knock-on effect of enabling refactoring and redesign at all stages of the project.
By iteratively creating a test that fails, and then making that test pass, source code is built that is well tested and designed and created through experience with use of that code. This is something that more tradition up front design does not address (instead, relying on long design-code-redesign iterations or the experience of the designer).

7.1.2 Tools

In this project, since Java was the implementation, JUnit was used to provide the automated testing framework. Also, with Eclipse being used as the IDE (Integrated Development Environment), a plugin called Infinitest was used. This plugin constantly scans the source code for changes, and when they occur runs any tests that might be affected. It then provides a clear visual clue to the developer as to whether all tests are passing, or if not, which ones fail and where.
Using TDD with Scrum, and with a new backlog based on working examples became the starting point for the actual development.

7.2 The Development Story

In this section the key stages of the software development project are described. The intention is not to show each the implementation of each feature but to show where the important decisions and designs are made. Since the project followed an agile process, there was no big upfront design, instead relying on practices such as the use of design patterns[11], TDD[24], refactoring[25] etc. to provide an environment in which changes could be easily and safely made as needed. The following sections are presented in chronological order.

Project Hosting & Tools

One of the main goals of this project was to make the language and code open source. To this end a project was created at GitHub and this used as the master source code repository. This ensured off-site backup during the project and full versioning using the Git SCM (Source Code Management). At this point it was necessary to give the project a name: *Neural*.
The project site can be found at: http://github.com/mbcx4jrh/Neural

Hello World (or, defining a XOR network)

Following the process for TDD meant that it was necessary to define a test before writing code. The simple example of defining a feed forward network with one hidden layer for simulating a XOR logic gate was chosen as the starting point. A script was written to define such a network (see Figure 12), and then a JUnit test created that loaded the script and created an ANN (see Figure 11). Only at this point was actual source code written, and its goal was to make the test pass.
At this point the decision was made to create an interface for objects representing ANNs and that concrete implementations of this interface would be passed from a parser object, acting as a Factory[11].
After the initial test passed, further additions were made to the automated test, to start testing input & output so that the code could be progressed.

```
network xor is feedforward {
          layer {
                    size 2
                    activation input
          }
          layer {
                    size 3
                    activation sigmoid
                    biased
          }
          layer {
                    size 1
                    activation sigmoid
                    biased
          }
}
```

Figure 12 The first Neural script, used to build an ANN capable of learning the XOR operation

```
@Test
public void testBasicXor() {
          Network network;
          ScriptParser = new ScriptParser();
          network = parser.parse(FileUtils.readFileToString("xor-1.neural");
}
```

Figure 11 The first test case, which merely reads a file and produces a Network object.
No other code has been written so this test fails by virtue of not compiling.

Adding Neuroph

The first ANN features to be added were implemented using the Encog ANN library. It was a goal of the project that the DSL could be implemented across many ANN libraries so the initial XOR test was replicated, and altered to use the Neuroph library. This was done by adding the following code:

```
parser.setUnderlyingLibrary("neural.networks.NeurophNetworkFactory");
```

to the test, which, of course, made it fail. Implementing this method and the resulting functionality enabled the underlying ANN library to be selected.
The Abstract Factory[11]design pattern was adopted for this feature, as the function of an abstract factory is that concrete implementations of the factory are used in different situations. Here, each ANN library would have it's own concrete factory, and set with the method shown above, that way, integrating an new ANN library can be done by implementing a new factory for that library.

Defining A New Network Type

A second example was created, this time defining a Hopfield network and can be seen in Figure 13. Now the network factory needed to be able to produce different concrete networks based on the name provided in the script. To do this a properties file was used with entries such as:

```
network.hopfield neural.networks.encog.EncogHopfieldNetwork
```

This property links the name of a network type "*hopfield*" in the script, to a class to instantiate by the network factory. Supplying such a property file for each underlying ANN library would allow the network factories to tie a keyword in the DSL to an implementing class to instantiate.

```
network hopfield_test is hopfield {
          size 5
}
```

Figure 13 The second *Neural* script created for the purposes of testing.

Adding Activation Functions

The initial feed forward networks all used the same activation function (a sigmoid) so new tests were created that did exactly the same except with different activation functions. By following the same pattern as with different network types, the properties files were used to tie the activation function name in the DSL to an implementing class, for example, the properties file:

```
activation.sigmoid org.engine.encog.network.ActivationSigmoid
```

binds the identifier *"sigmoid"* in the DSL to the implementing class.
Note that in the case of the Encog library, the activation functions could be used directly. This was because they all had a default constructor and so could be passed directly to the network without trouble. In the case of the Neuroph library, it was more troublesome and adapters[11] had to be used instead.

Training

Similarly to activation functions, training in the first example was a fixed type. New test scripts were created with various training types, and the same pattern as for the activation and network types was used to define a training type, for example in the properties file:

```
training.backpropagation neural.networks.encog.BackPropAdapter
```

This binds the identifier "*backpropagation*" in the DSL to the training implementation stated.
At this point additional generic training parameters were also added to the language, allowing training to run for a max number of epochs, to a certain error rate, and finally if both of those fail to converge then to restart the training a certain number of times.

Training and Testing Data

At this stage training and testing of the ANNs produced from the DSL had to happen programmatically, that is, passing the data directly to the network objects. To allow this to happen automatically, the definition of data blocks was added to both training and testing statements within the DSL script, for example:

```
input {
  1 0 1,
  0 0 0,
  1 1 0
}
```

And again, this was done by adding new tests that contained these data blocks, and then adding the code that made these tests pass.
At this point there was now a script that could define an ANN, the training method, and the training and testing data. Training could be simply invoked by calling:

```
network.train();
```

Data Sources
Although defining data directly in the DSL script is useful for simple examples, for more realistic problems it is far more useful to be able to define a source for data. A new version of the XOR test was created, but this time referring to file on the system, for example:

```
input csvfile “scripts/xor-test.csv”
```

The same properties file pattern was used as before, linking the input type of *“csvfile”* to an implementing class through this entry in the properties file:

```
data.csvfile neural.data.CSVFileSource
```

This allows different type of data sources to be used, with the second parameter to the statement being the identifier of the source (which in this case is file name).

Command Line Use
All usage, so far, of the *Neural* DSL was through its API. It would be useful to be able to define full scripts that contained both an ANN definition and training and testing data, and run them from the command line. This would avoid the need to use Java programming at all.
A command line executable was created so that the following would load a script, create an ANN, train the ANN, and run test data through the ANN:

```
./neural scripts/xor-cmdline.neural
```

Documentation
As one of the goals is to create an on-going open source project that is used by others, documentation is a must. This means documentation both on how to use *Neural*, and on how to extend it. Fortunately projects on GitHub get a wiki associated with the project, so this wiki was used to provide documentation to users and developers of the language.

Hopefully the reader now has an idea of the process and journey that produced this first version of *Neural*. The story presented above was not aimed to provide a technical description of *Neural* (for this see sections 7.4 & 7.3), but to provide a sense of how the project progressed. The project took 2 months to get to this stage and a significant amount effort was spent building a code base that had a high level of automated test coverage.

7.3 The Neural Language
The DSL, *Neural*, could be described in many ways, both formal and informal. In this section the DSL is described formally in section 7.3.1, and then more informally in sections 7.3.2 to 7.3.5. Finally a full working example, that can be used from the command line is given in section 7.3.6.

7.3.1 Language Definition

In Extended Backus-Naur Form (EBNF)[26] the language can be described as:

```
neural_script = [ activation ], network, [ training ], [ testing ]

network = “network”, identifier, “is”, identifier, network_block

network_block = “{”, parameters, { layer }, “}”

parameters = “parameters”, “{”, { parameter }, “}”

parameter = identifier, double

layer = “layer”, “{”, “size”, double,
        “activation”, identifier, [ biased ], “}”

activation = “activation”, identifier, “is”, identifier,
             “{”, { parameter }, “}”

training = “training”, “{“, training_params, “}”

training_params = “type”, identifier,
                        [ “error”, double, “%” ],
                        [ “epochs”, integer ],
                        [ “restart”, integer ],
                        [ input_data, output_data ]

input_data = “input”, data_block | data_ref

output_data = “output”, data_block | data_ref

data_block = “{”, { double }, [ “,”, { double } ], “}”

data_ref = identifier, id

id = ““”, identifier, “””

testing = “testing”, “{”, input_data, [ output_ref ]

output_ref = “output”, data_ref

identifier = < defined as Java identifier >
double     = < defined as Java double >
```

The definitions for “identifier”, “integer” and “double” have been omitted, but can be considered the same as the definitions in the Java language.

7.3.2 Network Definition

The network section of a *Neural* script defines how an ANN is constructed. Informally the network section looks like:

```
network [name] is [type] {
          parameters {
                    [parameter-name] [parameter-value]
                    ...
          }
          layer {
                    size [number-of-nodes]
                    activation [activation-name]
                    biased
          }
          ...
}
```

In this definition both the parameters and the layer sections are optional, with a network typically having at minimum one or the other. The biased keyword is also optional. Table 7 shows the meaning of the possible values in the script.

Value	Meaning	Examples
[name]	The name of the ANN (unused by Neural)	Abc123, joe, xor_example
[type]	The type of ANN that is to be created	feedforward, hopfield
[parameter-name]	The name of a network parameter	size, length, low
[parameter-value]	The value of a network parameter (a double)	1, 0, -1.234, 0.023
[number-of-nodes]	The number of nodes in the layer	1, 3, 27
[activation-name]	The name of an activation function (this can be either an actual activation function of the name of an activation section in the script)	sigmoid, threshold

Table 7 Explanation of values in the network section of a *Neural* script

7.3.3 Activation Definition

The activation section of a script allows the user to refer to an activation function with a certain set of parameters for later use. The intention is to save repeating the same parameters every time they are used. This section of the script looks like the following:

```
activation [name] is [type] {
          [parameter-name] [parameter-value]
          ...
}
```

Table 8 shows the meaning of the possible values.

Value	Meaning	Example
[name]	The identifier to be used elsewhere in the script	my_sigmoid, abc123
[type]	The type of activation function	sigmoid, threshold
[parameter-name]	The name of an activation function parameter	high, low
[parameter-value]	The value of an activation function parameter (a double)	1, 0, -5.23, 0.1234

Table 8 Explanation of values in the activation section of a *Neural* script

7.3.4 Training

The training section of a script defines the type of training and the training parameters, with the option of also adding training data. This section looks like the following:

```
training {
          type [training-type]
          error [training-error]
          epochs [max-epochs]
          restart [restarts]
          input [data-source] [source-id]
          output [data-source] [source-id]
}
```

The type parameter is the only mandatory parameter. The values for these parameters can be seen in Table 9. Additionally the input and output data may reference the data directly using the following notation:

```
input {
```

```
           a1 b1 c1,
           a2 b2 c2,
           a2 b3 c2
}
```

Where a1, b1, & c1 are the first set of training data, and a2, b2 and c2 the second and etc.

Value	Meaning	Examples
[training-type]	The training algorithm that will be used to perform the training	backpropagation, resilient_propagaton
[training-error]	The error rate (as a percentage) at which the training will halt	0.1%, 0.23%, 5%
[max-epochs]	The maximum number of training epochs, after which training will halt. Defaults to 100,000	1, 10000, 250001
[restarts]	If training hit the maximum number of epochs without reaching the required error rate, then this controls the number of time training is restarted after reinitialising the network. Defaults to 1	1, 10, 50
[data-source]	The type of data source to be used	csvfile, patternfile
[source-id]	An identifier describing the data source location. For a file data source this might be a filename. This is enclosed in quotes	"/User/joe/test1.csv", "xor-3"

Table 9 Explanation of values in the training section of a *Neural* script

7.3.5 Testing

The testing section of a script defines the data that will be passed to the ANN and the location of that data. It also, optionally, defines where to put the output from the ANN, by default writing to the console. The testing section looks like the following:

```
testing {
           input [data-source] [source-id]
           output [data-source] [source-id]
}
```

The input data source can be replaced by referencing the data directly as in section 7.3.4, and the meaning of [data-source] and [source-id] found in Table 9.

7.3.6 A Full Example

Figure 14 shows a full example for a feed forward ANN that is capable of learning the XOR operation. This example contains it's own training data, and reads the test data from a CSV file. Although this example uses the activation section, it does so only to reference the activation function by a different name (and so to change the type of activation function only one change is needed, rather than for each time it is used in a layer).

```
activation my_activation is sigmoid

network xor-example is feedforward {
          layer {
                    size 2
                    activation input
                    biased
          }
          layer {
                    size 3
                    activation my_activation
                    biased
          }
          layer {
                    size 1
                    activation my_activation
          }
}

training {
          type resilient_propagation
          error 0.01%
          epochs 50000
          restart 5
          input {
                    0 0,
                    1 0,
                    0 1,
                    1 1
          }
          output {
                    0,
                    1,
                    1,
                    0
          }
}

testing {
          input csvfile "scripts/xor-test.csv"
}
```

Figure 14 A full example of a *Neural* script for learning a XOR operation

7.4 The Architecture of Neural

The final result of this project is a Java library that can be used to parse *Neural* scripts and can instantiate and use the ANNs defined in those scripts. This section describes three technical architecture views of this library: the overall platform, the API from a users point of view, and how the underlying ANN libraries (such as Encog or Neuroph) are integrated.

7.4.1 The Neural Platform

The *Neural* platform (see Figure 15) is based on the Java Virtual Machine (JVM). It is a set of libraries, usable by a user in two ways: either through a Java API (see section 7.4.2) or through a command line binary. The platform uses the JParsec library to parse scripts and produce an abstract syntax tree (AST), which essentially is an object representation of the script. The layer shown as the *Neural Core* in Figure 15 uses a series of adapters and bridges[11] to integrate the underlying libraries into the platform. The *Neural Core* is then able to build concrete objects, which implement the *Neural API* and supply these either to the command line utility or the API user.

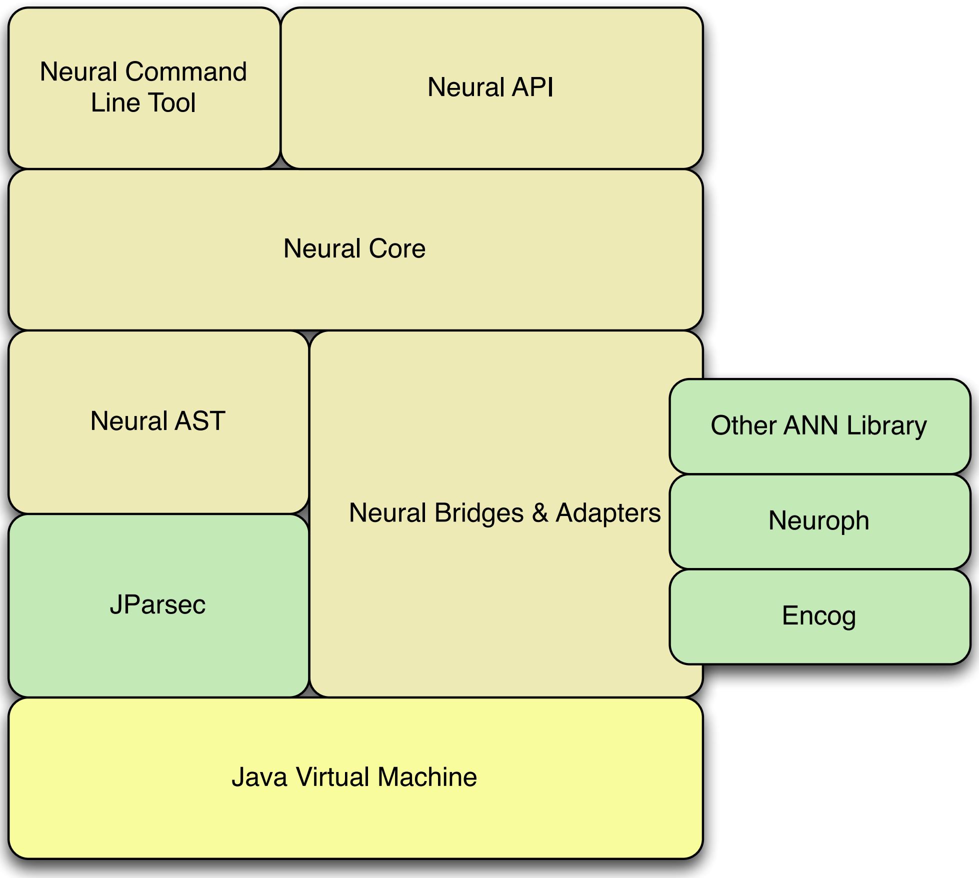

Figure 15 The Neural platform architecture

7.4.2 The Neural API

Neural presents a simple API to the user, consisting of just two classes (see Figure 16). The ScriptParser class performs the role (as the name suggests) of parsing a script, into an object that implements the Network interface. This class is a factory for Network objects.

The Network interface allows the user to train, and test an ANN. The script that defined the ANN may, or may not, have had training and testing data defined, so the it is possible to call both the train and test methods with, and without, test data. All data passed to the ANN is in the form of double arrays as these were the lowest common denominator.

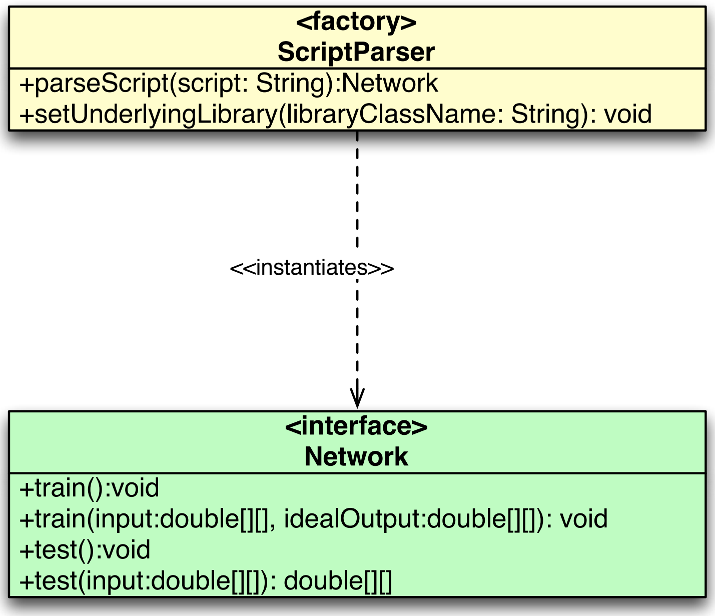
<factory>
ScriptParser
+parseScript(script: String):Network
+setUnderlyingLibrary(libraryClassName: String): void
<<instantiates>>
<interface>
Network
+train():void
+train(input:double[][], idealOutput:double[][]): void
+test():void
+test(input:double[][]): double[][]

Figure 16 The API for Neural (as seen by the user)

7.4.3 Integrating Underlying Libraries

Neural integrates the underlying ANN libraries using the Abstract Factory design pattern[11]. The ScriptParser class uses the abstract factory NetworkFactory to create Network objects, and by default the concrete instance of the abstract factory is for the Encog library. This factory can be changed at runtime using the setUnderlyingLibrary method.
Since different libraries might have different properties for the various functions, the current version of *Neural* contains a basic form of validation for the properties that can be set. Validation occurs after the script has been parsed to an AST and applies to parameter blocks for the network and the activation sections. Validation is controlled by a validation file containing entries such as:

```
activation.threshold.mandatory high low
activation.competative.optional size
```

Breaking down the first entry shows that it is for an activation function parameter block, for the threshold function, and that the high and low parameters are mandatory. The second entry describes an optional parameter for the competitive activation function.

7.5 Summary

This chapter has described the project to develop the *Neural* language. It has described the way in which the project was developed, both the approach and the actions. It has also shown the specification of the end result as a technical architecture and as a specification of the language. The next section will describe further how to use the language itself and how to extend the platform.

8 Results

This section examines the functionality of *Neural*. It firsts looks at how to use the language both programmatically (i.e. using it as a Java library API) and from the command line to execute scripts. It then goes on to describe the scope of the functionality current covered, before discussing how to extend the language further.

8.1 Using Neural

Neural can be used in several ways, from using the script to only define an ANN for use programmatically, to using the script to define and train an ANN, and then to perform calculations with it (testing). With all of the following sections the script shown in Figure 14 could be used (even though some parts of it might not be needed)

8.1.1 Create a network to use programmatically

Given a neural script, it is possible to parse the script and produce an untrained ANN, ready for use programmatically. To do this the following code would be needed:

```
// Load script from file (using Apache Commons)
String script = FileUtils.readFileToString(new File(name));

// Create a Neural parser
ScriptParser parser = new ScriptParser();

// Parse the script and return an artificial neural network
Network network = parser.parseScript(script);
```

The network object can now be used by calling its train or test methods, and passing data to it. For example, to use the train method:

```
//Load training data (readXXXData() methods are omitted here)
double[][] input = readInputData();
double[][] output = readOuputData();

//train the network
network.train(input, output);
```

Similarly, to use the network to compute output data:

```
//Load input data
double[][] input = readInputData();

//compute the output
double[][] output = network.compute(input);
```

When these methods are called, any references to training and testing data are ignored, and the data passed in via the methods is used instead. However, the training type and parameters in the training section are used, and so must be defined.

8.1.2 Create and train a network to use programmatically

If the training section of the *Neural* script contains the training data, either by reference or embedded, then the network can created as in section 8.1.1, and trained using the data in the script by calling the following method:

```
// train the network using the data in the script
network.train();
```

The network will be trained and can now be used programmatically.

8.1.3 Create, train, and test to use programmatically

A *Neural* script that contains all of the network, training, and testing sections that contain the necessary data can be used programmatically, as in the following code:

```
// Load script from file (using Apache Commons)
String script = FileUtils.readFileToString(new File(name));

// Create a Neural parser
ScriptParser parser = new ScriptParser();

// Parse the script and return an artificial neural network
Network network = parser.parseScript(script);

// Train the network using the data in the training section of the script
network.train();

// Test using the data in the testing section of the script
network.compute();
```

The results of passing the testing data to the network will now be delivered to the output device defined in the script. If no output device is defined then the results will be delivered to the console.
An example of an output device is the memory device, which can be defined in the testing section of the script as:

```
testing {
          input csvfile “xor.csv”
          output memory “xor-1”
}
```

The output of each computation is then stored in a Singleton[11] in memory and can be retrieved using the key “xor-1”, as with the following code:

```
MemoryTester tester = MemoryTesterStore.getInstance().retrieve(“xor-1”);
double[][] output = tester.getOutputData();
```

8.1.4 Command Line use

Neural comes with a command line binary. This binary will load a script, create the ANN, train it, and then perform the testing. This has the same effect as the code in section 8.1.3. An example with the script in Figure 14 would give the following output:

```
$ ./bin/neural scripts/xor-cmdline.neural
Creating parser...
Reading file ‘scripts/xor-cmdline.neural’...
Parsing script...
Training network...
Testing network...
Input: [0.0, 0.0]    Output: [0.011439126875298236]
Input: [1.0, 0.0]    Output: [0.9908491756463426]
Input: [0.0, 1.0]    Output: [0.9909028910691131]
Input: [1.0, 1.0]    Output: [0.010116505779657186]
```

8.1.5 Errors

Three types of errors typically occur when using *Neural*, and all will result in a runtime NeuralException being thrown (or displayed, in the case of the command line binary):

Neural Syntax Errors

These types of errors occur when the syntax is used incorrectly, such as the following script (note 'as' instead of 'is'):

```
network joe as hopfield {
}
```

This would result in the following:

```
Exception in thread "main" org.codehaus.jparsec.error.ParserException: line 1, column 13:
is expected, a encountered.
```

Where the exception is clearly indicating the point in the file at which the error occurs.

Invalid Property Error

These types of errors occur when the types of networks, training, activation functions etc. do not bind to an implementing class via the properties file (see section 7.4.3). For example, in the following script the network type 'hopeful' is not implemented:

```
network joe is hopeful {
}
```

This would result in the exception:

```
Exception in thread "main" neural.NeuralException: Unknown property (network.hopeful)
```

Validation Errors

These types of error occur because of invalid parameters in the parameters section of either the activation definition or the network definition. For example:

```
network joe is hopfield {
          parameters {
                    high 5
          }
}
```

This would result in the exception:

```
Exception in thread "main" neural.NeuralException: Invalid network definition 'joe' [Parameter
'high' is present but not allowed as optional or mandantory, Mandantory parameter 'size' is
missing]
```

Which states that the parameter 'high' is not allowed, and that a required parameter is missing.

8.1.6 Changing the ANN library

By default, *Neural*, uses the Encog ANN library to implement the networks. This can changed by calling a method on the ScriptParser with the name of the factory for the alternative library. For example, to get *Neural* to use the Neuroph ANN library the following code would be used:

```
//Create parser
ScriptParser parser = new ScriptParser()

//Set underlying ANN library
parser.setUnderlyingLibrary("neural.netoworks.NeurophNetworkFactory");
```

Any scripts that are parsed would now produce networks implemented by the Neuroph library.

8.2 The Scope of Neural

One of the goals of this project was to start up a project that would have a life beyond this thesis. While being able to cover the full functionality of both the Encog and Neuroph ANN libraries would have been good, realistically, time constraints made this impossible. The *Neural* platform was built as an extendable platform, so expanding the current functionality is matter of tie, rather than design, and will continue into the future.

Encog was chosen as the reference platform for *Neural*, with Neuroph also being implemented across just a few features to act as a proof of concept for the expandability. Tables 8,9, & 10 show the current feature coverage of *Neural* across both libraries.

Network Type	Encog	Neuroph
Feed Forward (Perceptron) Neural Network	✔	✔
Hopfield Network	✔	✔
Adaptive Resonance Theory 1 (ART1)	✔	
Adaline	✔	
Bidirectional Associative Memory (BAM)		
Boltzmann Machine		
Counterpropagation Neural Network		
Elman Recurrent Neural Network		
Jordan Recurrent Neural Network	✔	
Neuroevolution of Augmenting Topologies (NEAT)		
Radial Basis Function Network		
Self Organizing Map		

Table 10 Coverage of ANN types for Encog and Neuroph (feature set is from Encog)

Activation Function	Encog	Neuroph
Bipolar	✔	
Competitive	✔	
Elliot	✔	
Gaussian	✔	
Hyperbolic Tangent	✔	✔
Linear	✔	
Sine Wave	✔	
Sigmoid	✔	✔
SoftMax	✔	
Step (threshold)	✔	
Tangential	✔	

Table 11 Coverage of activation functions for Encog and Neuroph (feature set is from Encog)

Training Method	Encog	Neural
ADALINE Training	✔	
Backpropagation	✔	✔
Competitive Learning		
Hopfield Learning	✔	✔
Instar and Outstar Training		
Levenberg Marquadt (LMT)		
Manhattan Update Rule Propagation		
Nelder Mead Training		
Resilient Propagation	✔	✔
Scaled Conjugate Training	✔	

Table 12 Coverage of training method for Encog and Neuroph (feature set is from Encog)

8.3 Extending Neural

One of the goals is to have an extendable language, both in functionality it provides through network types, activation functions, and training methods, but also with the ability to use different underlying ANN libraries. This section describes how these extensions can implemented, and also shows how the design of the *Neural* platform is conducive to extending the syntax.

8.3.1 Extending The ANN Library Coverage

While the method in which a particular Network Factory implements the underlying ANN library is entirely up to the Network Factory, both Neuroph and Encog Network Factories follow the same pattern. This section will describe how to increase the coverage for the Encog Network Factory.
Note that for adding activation functions and training types, this in highly dependent on how the Network Factory as been implemented.

Adding A New Network Type

To add a new type of ANN, a new implementation of the interface Network is required. The Network implementation adapts the underlying library implementation of the ANN to the *Neural* interface (the Adapter design pattern[11]). This is made easier by extending the abstract class AbstractNetwork, which provides a default implementation for many of the methods. The implementation must take care of two key methods:

```
public void initNetwork(NetworkDef definition);
public double[][] compute(double[][] input);
```

The first constructs and initialises the underlying library's ANN, and then the second will perform computations with the ANN. The training methods are taken care of by default by the AbstractNetwork, but may need overriding depending on the type of network. Figure 17 shows the full interface to the Network object that would need implementing if the AbstractNetwork were not used.

```
public interface Network

          // initialization methods
          public void initNetwork(NetworkDef networkDefinition);
          public void initTraining(TrainingDef trainingDefinition);
          public void initTesting(TestingDef testingDefinition);

          // returns the name of the ANN
          public setName();

          // returns the type of network
          public String getType();

          // training methods
          public void train();
          public void train(double[][] input, double[][] output);

          // compute methods
          public void compute();
          public double[] compute(double[] input);
}
```

Figure 17 The interface for the Network object in *Neural*

After implementing the Network interface, an entry must be placed in the properties file (for Encog, in encog.properties), that binds the network type in the *Neural* script to the new class. For example, if we wanted to create a new ANN implementation for the Elman Recurrent Network, we might want to refer to it in the script as 'elman', and we would create a new class, implementing Network, called 'ElmanNetwork'. The entry in the encog.properties file would therefore be:

```
network.elman neural.networks.encog.ElmanNetwork
```

After this *Neural* would recognize this network type and scripts such as the following could be used:

```
network abc is elman {
          ...
}
```

Adding New Activation Functions

Adding a new activation function for the Encog library merely entails adding a new entry into the properties file. Encog's own activation functions all implement a default constructor and are passed directly to the ANN object, hence a trivial operation to add them.

Parameters are also easy to add for the Encog library. Since all activation functions implement a getParameter/setParameter interface, the Encog Network Factory can use this directly. All that is required is the correct validation properties adding.

For example, to add the Step activation the class needs to be added to the encog.properties file:

```
public interface TrainingAdapter {

          // initialization method
          public void init(TrainingDef trainingDefinition, Network network);

          // training methods
          public void train();
          public void train(double[][] input, double[][] output);
}
```

Figure 18 The interface for adapting from the underlying library's training class

```
activation.step org.encog.engine.network.activation.StepActivation
```

Then an entry for the optional parameters, is added to the validation.properties file:

```
activation.step.optional high low center
```

Now the new activation function can be used in *Neural* scripts:

```
activation my_act is step {
          high 0.9
          center 0.4
          low 0.1
}
```

Adding New Training Methods

Training methods are also added by added with a new entry in the encog.properties, however, in this case there is one slight complication: in the Encog library training method objects are not instantiated with a default constructor. To overcome this the Adapter[11] design pattern is used to adapt the Encog training class to one usable by *Neural* (see Figure 18).

8.3.2 Implementing New Underlying Libraries

A key goal of the project was to be able to use different ANN libraries, and *Neural* currently uses two, Encog and Neuroph. To add an ANN library so that it can be used by *Neural* requires only that the NetworkFactory interface be implemented, and the class name passed to the ScriptParser using the setUnderlyingLibrary method. This results in the parser using the new network factory from that point on. Of course it is also necessary to implement the various features of the new library through the methods described in section 8.3.1.

8.4 Summary

The chapter has described the current scope of *Neural* and how to use and extend it. We have seen how the platform was designed to be simple to use and simple to extend. The next chapter will look at how successful this has been.

9 Analysis & Evaluation

The *Neural* language was evaluated using three different methods:

- Working through several examples that use the data from the UCI repository
- Building an ANN agent for an a Tic Tac Toe game that currently uses Sarsa and Q-Learning reinforcement learning algorithms
- Getting user feedback via a short questionnaire

Each of these evaluation methods provides it's own feedback and recommendations for improvements, and these are collated in section 9.3.
Additionally we will look at the overall design of *Neural* as a pattern to be used more generally when implementing a DSL or API over multiple underlying libraries. And finally, we will look at some of the lessons learnt and mistakes made during the project.

9.1 Examples From Neuroph/UCI

The Neuroph project website has many tutorials that involve experiment with different ANN designs to process real world data from the UCI Machine Learning Repository (see section 12 for the addresses of these sites). These tutorials all follow a similar process in order to try and obtain an optimal ANN design for processing the data (typically classification or function approximation). The stages of this process are:

1. Prepare the data set
2. Create a training set
3. Create an ANN
4. Train the ANN
5. Test the network to ensure it is trained properly

Stages 3-5 are repeated until an acceptable ANN design is found.
This process was followed for 5 of the tutorials, using *Neural* to implement the ANNs, and during each stage notes were made on what went well and what difficulties occurred. These are summarized below.

Preparing The Data Set

The Neuroph examples all perform normalization on the original data from the UCI repository. In the examples this is performed using a spread sheet and CSV files produced, and this was replicated when using Neural.
It was noted in all cases, that it would be an easy addition to the language to have a "normalizing" data source, so that the original data could be used directly. It was surprising that Neuroph does not have the feature, given its prevalence throughout the tutorials.

Creating A Training Set

Neuroph allows the user to create separate training and testing sets. It assumes that the data is already split into these sets, and that the input and outputs are in one file.
Using *Neural* is similar, except we must go the extra step of splitting out the input and output data manually into separate files. It would have been better here to have the input and output data in one file and to be able to reference that.

Creating An ANN

The tutorials all involved feed forward networks, and each time this stage was repeated the process of creating a new ANN was followed using the Neuroph GUI.
Using *Neural* made the process much easier, as when changing an ANNs parameters a quick edit could be made to the file (for example changing the number of hidden nodes) and it simply rerun from the command line. See Figure 19 for an example of a script used in the tutorials.

Training The ANN

For both Neuroph and *Neural* this was a straightforward step, for Neuroph just a button press, and for *Neural* running from the command line. What did differ, however was the quality of the output, as Neuroph could show a graph of the error as the training progressed. It would be good if *Neural* had some sort of output to show how training progressed. Additionally it was noted once or twice that *Neural* did not reach the required error level within the default number of epochs, however it did not tell the user this.

Testing The ANN

When Neuroph tests the ANN it shows not just the computation results but also errors (include total mean square error) as well. For *Neural*, there is just the result of the computation, and the errors had to be calculated in a spread sheet. The actual results of the learning were identical.
It would be good to have Neural produce the same kind of output as Neuroph.

Summary

Neural worked well as a means of defining an ANN and being able to easily change and run. However it's handling of training and testing output let it down when compared against the Neuroph tutorial. The following features would have made running these tutorials extremely trivial:

- Normalisation of raw data
- Combine input and output data in the same file
- Reporting of errors during training and testing (this could be to the command line)
- Cross validation to avoid having to split training/test data

With these features *Neural* would have perform much better than the Neuroph tools.

```
activation my_activation is sigmoid

network concrete_strength is feedforward {
          layer {
                    size 14
                    activation input
                    biased
          }
          layer {
                    size 20
                    activation my_activation
                    biased
          }
          layer {
                    size 1
                    activation my_activation
          }
}

training {
          type backpropagation
          error 0.1%
          input  csvfile “concrete_train_input.csv”
          output csvfile “concrete_train_output.csv”
}

testing {
          input csvfile “concrete_test.csv”
}
```

Figure 19 An example of a script used while following the Neuroph examples (to be run from the command line)

9.2 Building An Agent for Tic Tac Toe

As part of the Learning in Autonomous Systems course at The University Bristol, a software package is used to investigate various machine learning algorithms playing a game of Tic Tac Toe. The software source code was written in Java and designed to be extendible, especially allowing additional machine learning algorithms to be added, and play against one another. It was decided that it would be a good idea to implement an ANN agent to attempt to play against other algorithms.

The implementation for the ANN agent was simple and kept a record of all moves in a game for which that player finally won (up to a maximum training set size). At the end of each game the ANN agent would reinitialize the ANN and train from the current training set. When asked by the software for a move, the agent would use the current state of the board as input and select the output with the highest score. If the move were invalid then a random valid move would be chosen.

The idea was for the ANN agent to learn moves that resulted in a win, given the current state of the board. A by-product of this was that only valid moves would be learned.

The architecture of the ANN was a feed forward network with 9 input and 9 output nodes. The number of hidden nodes could be varied, and the starting point can be seen in Figure 20. The code for the ANN agent can be found in the Appendix (section 14.2).

```
network nn_agent is feedforward {
          layer {
                    activation input
                    size 9
          }
          layer {
                    activation sigmoid
                    size 18
          }
          layer {
                    activation sigmoid
                    size 9
          }
}

training {
          type resilient_propagation
          error 0.1%
}
```

Figure 20 The *Neural* script for the ANN player in Tic Tac Toe

What Worked Well

Creating and training the ANN was very simple with the *Neural* API. Most of the code written was to convert between the Tic Tac Toe software's representation of the board and the double arrays required by *Neural*.

Once the code was written, training parameters and the network structure (for example, the number and size of hidden layers) could be varied without changing any code. This made it very easy to experiment with.

What Was Difficult

It was difficult to view what was actually happening during training. It would have been good to be able to see the errors during training, and also whether or no the required error had been reached.

Result

As a result of the above an ANN agent successfully played Tic Tac Toe. The ANN agent easily learnt valid moves, and it would win consistently against a random player.

However against the Sarsa or Q-Learning algorithms, it lost (this was not unexpected as these algorithms learn perfect games). However, the purpose of this was not build a great playing ANN agent but to investigate how well *Neural* performs in facilitating that build.

9.3 Feedback from users

Two groups of users were approached and asked for informal feedback in the form of a short questionnaire. The first group was a set of existing Encog users (Neuroph users were considered but the coverage of functionality from that library was too low). These users are already using and experimenting with ANNs and range in the experience from student to professional. The second group was a group of peers on the Advanced Computer Science MSc. at The University of Bristol. Most of these had little experience with ANNs (ANNs were not on this years syllabus). The results were collated and are given as a summary across all users.

The Questionnaire

The questionnaire was presented through email and asked the following questions:

1. What is your level of experience with ANNs?
2. How easy was it to use Neural?
3. How steep was the learning curve for Neural features?
4. What do you think is missing?
5. Would you use Neural in future (assuming you are using ANNs)?

Results

Total of 9 replies:

1. Range from inexperienced to very experienced, with most in the middle
2. Easy to download. Easy to include library in own projects. Some issues with location of properties files.
3. Generally a flat learning curve with using the language. People inexperienced in Java thought extending the language would be difficult.
4. Cross validation. Normalisation. Finer control of training parameters. More training methods.
5. Approximately 60% replied positively, but mostly on the condition that it would be further developed and supported.

Analysis

The Neural language was easy to learn and accessible for all. There were some slight problems with the packaging of the binary, but these could be easily fixed. *Neural* is a tool that will be used, but it must be an active project in order for it to be used.
Features that would be desirable to add are:

- Cross validation
- Normalisation
- More training parameters (for example, learning rates)

9.4 What Should Be Changed or Added

The evaluation has suggested several features that should be added:

Cross Validation

Cross validation is a technique for utilizing a single data set for training and testing[27]. One of the most common issues with using *Neural* for real problems is a lack of training and testing analysis. This feature could be added as a decorator type design pattern[11] to the existing data sources, or it could be implemented more fully within the language to allow more control (for example, with the number of folds).

More Detailed Training Information
The ability to see how the error changes as the training progresses is a useful feature. This could be accomplished easy with some command line options that output the information to console, or some file. It would be simple to use this for plotting graphs using whatever tool the user wished (such as gplot).

Normalisation
Normalisation would speed up the process of preparing the data and although it had initially been discarded as outside of the scope of this project, it becomes clear when running through the UCI examples that it would be extremely beneficial, and time saving. It could be implemented in a similar way to the cross validation, either through a decorator design pattern with data sources, or through it's own syntax in the training section of a script.

Increased Functionality Coverage
The key areas to increase coverage would be with the training algorithms. Adding new training algorithms would allow more experimentation, as would allowing greater parameter control on existing algorithms. This would take the form of allowing a parameters block in the training section, and adding new implementation of existing parameters.
Expanding coverage across more of the Neuroph library, and completing the Encog coverage would also improve the usability.

Packaging
Although not a major feature, improving the packaging would increase the accessibility of *Neural.* Currently the deliverable is a whole project including jars, source code, class files, property files, etc. It would be good to supply a small binary only package that wrapped up the jars and property files in a single package.

GUI
When experimenting with different network designs for the UCI examples, it was helpful to have a GUI tool to visualize the training. Visualizing the network while looking good, does not actually help much with the experiments.
One direction this feature could take is to provide an IDE type environment in which *Neural* scripts could be created, edited and run. The Eclipse IDE platform, or the NetBeans IDE platform would be ideal for this.

9.5 Neural As A Design Pattern

In software engineering a design pattern is a reusable solution to a commonly encountered problem[28], and it is possible to examine the *Neural* platform and extract the design as a pattern that could be applied to other similar problems.

The Problem
A common set of algorithms (for example, machine learning) are implemented in a software library, with each algorithm implemented in a single class. Many such software libraries exist for these algorithms, however each provides a different interface. A single API is wanted so that these libraries can be interchanged.

The Solution
By using the Abstract Factory, and Adapter patterns[11] a common API can be provided that underneath is implemented using a particular library.
Figure 21 shows the class diagram for this pattern, and the key elements are:

- The **Gateway** class provides a means to access the **Algorithm** implementations
- The **Gateway** class uses the **AbstractFactory**, and which particular concrete instance is used is controlled by the method **setUnderlyingLibrary()**

- The **ConcreteFactory** supplies instances of **Algorithm** that adapt the part of the **SpecificLibrary** that implements the required algorithm.
- A different library can be implemented by creating a new implementation of **AbstractFactory** and an **Algorithm** implementation for each algorithm in the new library

This design pattern can be applied to situations when multiple libraries need to be substituted for one another. This might be because of differing capabilities, performance, or functionality. A particular example is to apply this to machine learning in general, where multiple libraries exist that implement the various machine learning algorithms.

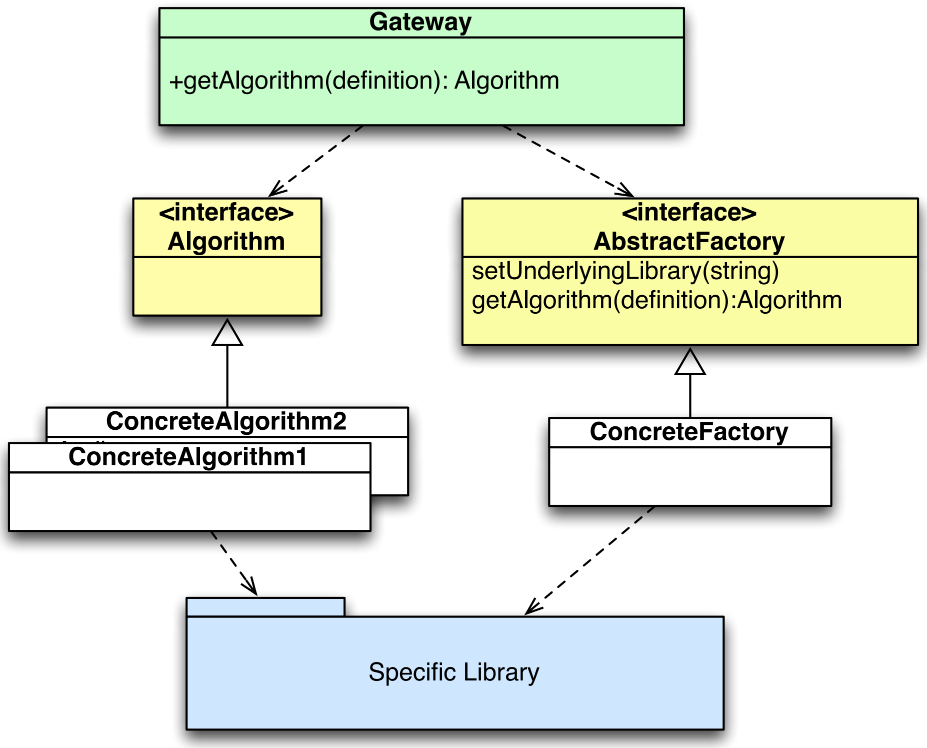

Figure 21 A design pattern for providing a single API for multiple set of libraries

9.6 Lessons Learned

Any project has mistakes. Agile projects especially, embrace mistakes, and takes time to review and learn from these mistakes. What follows is a (non-exhaustive!) sample of the mistakes that were either made or narrowly avoided during this project.

9.6.1 Do not underestimate working with libraries

When planning and estimating time it is easy to underestimate the amount of time it might take to work out how to use a particular library to perform some function properly. Every library has it own idiosyncrasies, some are very well designed and easy to use, while others are unfathomable. Most sit somewhere in between these two extremes. The amount of time it can take to figure out complex functionality can be large (and significant for a small 1 person project). It is very easy to read an API and work through some tutorials, only to find out later the tutorial was too simple and real world use is far more complicated.
The risk of this happening increases as the number of libraries in use increase, and this became apparent when attempting to work with as many underlying ANN libraries at the beginning of this project. Learning each library take time.

9.6.2 No battle plan survives contact with the enemy[29]

The famous quote from Moltke aptly applies to most software projects and there are many reasons why the best of plans can start to unravel once the project starts. In this case the set of features initially planned did not correspond easily to testable end-to-end scenarios with the libraries. Also once more information was known about the ANN libraries it because more obvious to build scenarios that corresponded closely with the supplied examples from the libraries.
At the beginning of a project, not everything is known:

- The feature set can change due to experience using the software
- The feature set can change due to limitations only known from using libraries
- There may be technical limitations or problems that are only discovered once the project is fully engaged

So it becomes important to be able to review the current state of the project and, if necessary, re-plan and change direction accordingly.

9.6.3 The Curse Of Generality

It is very easy to overgeneralize an interface. In fact, it is entirely possible to generalize every interface until the signature looks something like:

```
public Object doSomething(Object input);
```

I call this "The Curse of Generality". It occurs when we start to generalize an interface, but then don't know when to stop.
When initially looking at the Network interface, considerations were given to generalizing it to cover machine learning in general. Simplified, this looked like:

```
public interface Network {
          double[] compute{double[] input);
          void train(double[][] input, double[][] output);
}
```

And a generic machine learning interface might look like:

```
public interface MLAlgorithm {
          double[] compute(double[] input);
          void train(double[][] input , double[][] output);
}
```

There isn't much change here, just the name of the interface. However it was then tempting to generalize again, so that this could apply to any form of data processing with:

```
public interface DataProcessor {
          Data compute(Data data);
          void init(Data data);
}
```

Here we have done away with the doubles and replaced with a generic Data object (and is only one step away from Object). We have also renamed the train method to "init" to reflect that it might not just be training, but initializing some processor.
All of this, although generalizing for multiple uses, is destroying the original interface with each step. The power of a simple interface comes from its directness and clarity, something the DataProcessor interface would not have if it were being used for ANNs.

It is, of course, a trade off, but I would argue that in this case it is better to keep the original interface than to generalise to the data processor interface. The machine learning interface however, would be a useful generalisation.

9.6.4 Benefits of TDD

Many smaller issues occurred during the software development, and some large refactoring had to take place. Some redesign of the key interfaces even took place at some points. Normally if design changes occur late in a project then the cost can be large.

Test Driven Development (TDD) was a great help in reducing this cost down to the same as at the start of the project. Because every addition or alteration to the source code was preceded by adding or altering an automated test there existed a large set of JUnit automated tests. These tests were run every time a change was made and gave instant feedback on the correctness of the code.

With a large set of automated tests, changes could be made with full knowledge of whether the change has broken or preserved the features. This ability was invaluable when making design changes during the project. Without this ability the time spent ensuring that the code still worked manually would have been immense, and the final deliverable would probably have had many bugs.

As it stands there are **no** known bugs in the code.

9.7 Summary

We have seen how *Neural* has been evaluated and that it has been a successful project. It is also clear how the platform needs to evolve to be more useful in future. This chapter has also shown a new design pattern for implementing APIs over multiple underlying libraries – something that it useful across many areas of software development.

10 Summary

This project was a software development for 1 person over 2 months. During that time the DSL, *Neural*, was created, evolved, and initially released. It was has now been used to attempt real world problems outside of its own project (see section 9.1). The evaluation of the use of *Neural* is that it is a good base on which to build a fit for purpose language for the future. The amount of work needed to bring *Neural* to that level is relatively small, and perhaps only constitutes another months worth of work.
One criticism that might be made of this project is "where is the language design?" and hopefully that has been addressed by the agile nature of the project (see sections 6.3.4 & 7.1). One example of the agile process driving the design through real use of the API is with the Network API method:

```
public double[] compute(double[] input);
```

As to begin with, this method had the signature:

```
public void compute(double[]input, double[]output);
```

And although it is only a small change on a small part of the API it represents how agile design occurs. The initial version of this method mirrored the equivalent method in the Encog API, but after using it repeatedly in automated tests, it became clear that the final version made more sense to a programmer. Because of the high test coverage, a key method in the API could be changed with confidence that nothing would break.
Another example of the agile process is with cross validation and normalisation. Early in the project these features were deliberately not included, with the rationale being that this was data manipulation and it would be outside the scope of the project. During evaluation (actually using the language), it became clear that these two features would be highly beneficial. Here we see the benefit again of using something in anger, and one regret is that at least one of the evaluation problems could have been tried during the project in order to bump up the priority of these features.

10.1 Achieving The Goals

The primary goals of the project were:

- Review & Evaluate
- Build & Test
- Evaluate
- Make Available

These have all succeeded. A review of the background of ANNs & DSLs, combined with an analysis of the need for a DSL for ANNs showed what needed to be done. The DSL, *Neural,* was built and tested on real world examples (from the UCI repository). An evaluation of the quality of the DSL was performed and results combined into a feature list for the future. Finally, *Neural* is available here:

http://github.com/mbcx4jrh/neural

One additional goal that has not been completed, yet, is to present the DSL in a journal paper. This will wait until the next release of *Neural*.

10.2 Future Plans

The project is hosted on GitHub and available to use. It's open source and there have already been several volunteers to help with the future development of the language. These volunteers have come from the community of Encog users, and include one researcher and one PhD student. It is my intention to continue to lead this open source project, starting with the list of things to add in section 9.4.

The following features (see section 9.4) are next in line for further development:

- Cross-validation
- Normalisation
- More information output during training & testing
- Improved packaging

Finally, to quote Jeff Heaton, the creator of Encog, and owner of Heaton Research:

"I like it."

11 References

1. Haykin, S. *Neural Networks: A Comprehensive Foundation (2nd Edition)*. (Prentice Hall: 1998).
2. Siegelmann, H. T. & Sontag, E. D. Turing computability with neural nets. *Applied Mathematics Letters* **4**, 77–80 (1991).
3. Senyard, A., Kazmierczak, E. & Sterling, L. Software engineering methods for neural networks. 468–477 (2003).
4. Rodvold, D. M. A software development process model for artificial neural networks in critical applications. **5**, 3317–3322 (1999).
5. Lee, C.-C. & Shih, C.-Y. Time Series Prediction Using Robust Radial Basis Function with Two-Stage Learning Rule. **2**, 382–387 (2007).
6. Shah, M. A. & Meckl, P. H. On-line control of a nonlinear system using radial basis function neural networks. **6**, 4265–4269 (1995).
7. Maclin, R. & Opitz, D. Popular Ensemble Methods: An Empirical Study. *arXiv.org* **cs.AI**, (2011).
8. Ding, S., Li, H., Su, C., Yu, J. & Jin, F. Evolutionary artificial neural networks: a review. *Artificial Intelligence Review* 1–10 (2011).
9. Yao, X. Evolving artificial neural networks. **87**, 1423–1447 (1999).
10. Chu, S.-L. & Hsiao, C.-C. OpenCL: Make Ubiquitous Supercomputing Possible. *HPCC* 556–561 (2010).
11. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison-Wesley Professional: 1994).
12. Brooker, R. & Morris, D. Experience with the Compiler Compiler. *The Computer Journal* (1967).
13. Knuth, D. E. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM* **7**, 735–736 (1964).
14. Johnstone, A. & Scott, E. Generalised recursive descent parsing and follow-determinism. *Compiler Construction* **1383**, 16–30 (1998).
15. Mernik, M., Heering, J. & Sloane, A. When and how to develop domain-specific languages. *Acm Computing Surveys* **37**, 316–344 (2005).
16. Royce, W. Managing the development of large software systems. (1970).
17. Beck, K., Beedle, M. & Van Bennekum, A. Manifesto for agile software development. *The Agile …* (2001).
18. Coplien, J., Hoffman, D. & Weiss, D. Commonality and variability in software engineering. *Ieee Software* **15**, 37–+ (1998).
19. van Deursen, A. & Klint, P. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* **10**, 1–17 (2004).
20. Dower, S. & Woodward, C. J. ESDL: a simple description language for population-based evolutionary computation. *Proceedings of the 13th annual conference on Genetic and evolutionary computation* 1045–1052 (2011).
21. McConnell, S. Software Project Survival Guide - Steve McConnell - Google Books. (2009).
22. Schwaber, K. *Agile Project Management with Scrum*. (O'Reilly Media, Inc.: 2009).
23. Beck, K. & Fowler, M. *Planning Extreme Programming*. (Addison-Wesley Professional: 2001).
24. Janzen, D. S. & Saiedian, H. On the influence of test-driven development on software design. *… Proceedings 19th Conference on* (2006).
25. Fowler, M. & Beck, K. Refactoring: improving the design of existing code. (1999).
26. Scowen, R. S. Extended BNF-a generic base standard. (1998).

27. Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International joint Conference on artificial intelligence* (1995).
28. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns. Elements of Reusable Object-oriented Software*. (Addsion Wesley Longman: 1999).
29. graf von, H. M. *Militärische Werke, Volume 2*. (Nabu Press: 2012).

12 Web Resources

URLs are correct and accessible at the time of writing (September 2012).

12.1 ANN Libraries

dANN	http://wiki.syncleus.com/index.php/dANN
Encog	http://www.heatonresearch.com/encog
Neuroph	http://neuroph.sourceforge.net/
PyBrain	http://pybrain.org
PyNN	http://neuralensemble.org/trac/PyNN

12.2 Language Resources

JavaCC	http://javacc.java.net/
JParsec	http://jparsec.codehaus.org/

12.3 Other

UCI Machine Learning Repository	http://www.ics.uci.edu/~mlearn/

13 Abbreviations

ANN	Artificial Neural Network
API	Application Programmers Interface
AST	Abstract Syntax Tree
CSV	Comma Separated Value
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
GPL	General Purpose Language
GPU	Graphical Processing Unit
GUI	Graphical User Interface
JVM	Java Virtual Machine
RBF	Radial Basis Function

14 Appendix

14.1 Neural API

14.2 Tic Tac Toe Agent

The code using the *Neural* API is highlighted.

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import neural.Network;
import neural.ScriptParser;
import neural.tools.WindowedData;

import org.apache.commons.io.FileUtils;


public class NeuralAgent implements Agent {

          private static final boolean DEBUG = false;

          private Network network;
          private List<Data> episodes;
          private Agent randomAgent;
          private TTTEnv env;
          private int trainStrength;
          private WindowedData dataWindow;

          /**
           * Basic algorithm here is:
           *
           * Input : 9 nodes representing the board (-1 0, 1)
           *
           * hidden layers
           *
           * Output: 9 Nodes representing the choice of next move
           *
           * Online Training: play game storing all inout/outputs.
           *     if win then train using this data
           *     if lose then train with opponents data
           * @param env
           */

          public NeuralAgent(TTTEnv env, int trainStrength, int windowSize) {
                    this.env = env;
                    this.trainStrength = trainStrength;
                    dataWindow = new WindowedData(windowSize);
                    initNN();
                    randomAgent = new RandomAgent(env);
          }

          @Override
          public Action newEpisode(State initialState) {
                    //reset history
                    episodes = new ArrayList<Data>();
                    //return first move
                    return nextMove(initialState);
          }

          @Override
          public void endEpisode(float finalReward) {
```

```
			out("Game over - final reward: "+finalReward);
			//decide if train with history

			if (finalReward > 0)
				trainNetwork();
			else
				reverseTrain();
		}

		@Override
		public Action getAction(State state, float reward) {
			return nextMove(state);
		}

		public void out(String msg) {
			if (DEBUG) System.out.println(msg);
		}

		public void out(boolean b, String m) {
			if (b) System.out.println(m);
		}

		private Action nextMove(State state) {
			double[] input = getData(state);
			double[] output = network.compute(input);

			TTTAction action =  getAction(output, state);
			episodes.add(new Data(input, getData(action)));
			return action;
		}

		private double[] getData(TTTAction action) {
			double[] out = new double[9];
			short x = action.row();
			short y = action.col();
			out[x*3+y] = 1;
			return out;
		}

		private double[] getData(State state) {
			double[] data = new double[9];
			TTTState s = (TTTState)state;
			for (int x= 0; x<3; x++) {
				for (int y=0; y<3; y++) {
					data[x*3+y] = map(s.get(x, y));
				}
			}
			return data;
		}

		private double map(TTTSymbol sym) {
			switch (sym) {
			case BLANK:
				return 0;
			case O:
				return -1;
			case X:
				return 1;
			default:
				return 0;

			}
		}

		private TTTAction getAction(double[] output, State state) {
			int m = 0;
			double max = Double.MIN_VALUE;
			out("Net chose: "+Arrays.toString(output));
			for (int i=0; i<output.length; i++) {
				if (output[i] > max) {
					max = output[i];
					m = i;
				}
			}
			out(".. so move is "+m);

			short x = (short) (m % 3);
			short y = (short) (m / 3);
			out("Next move: "+x+", "+y);
			TTTAction action = new TTTAction(x, y);
			if (!valid(action, state)) {
				out(false,"Net chose invalid action");
				return (TTTAction)randomAgent.getAction(state, 0);
			}
			else {
				out(false, "Net chose good!");
				return action;
			}
	}

	private boolean valid(Action action, State state) {
			Action[] possibleMoves = env.getActionArray(state);
			return Arrays.asList(possibleMoves).contains(action);
	}

	private void trainNetwork() {
			int i=trainStrength;
			while (i-- >0) {
				out("Training network...");
				for (Data data: episodes) {
					dataWindow.addData(data.input, data.output);
				}
			}
			network.train(dataWindow.getInputData(), dataWindow.getOutputData());
	}

	private void reverseTrain() {
			reverseEpisodes();
			trainNetwork();
	}

	private void reverseEpisodes() {
			List<Data> newEps = new ArrayList<Data>();
			double[] in;
			for (Data d: episodes) {
				in = d.input;
				for (int i=0; i<in.length; i++) {
					in[i] = - in[i];
				}
				newEps.add(new Data(in ,d.output));
			}
			episodes = newEps;
	}

	private void initNN() {
			String script = null;
			try {
				script = FileUtils.readFileToString(new File("first.neural"));
			}
			catch (IOException e) {
				e.printStackTrace();
				System.exit(-1);
			}

			ScriptParser parser = new ScriptParser();
			network = parser.parseScript(script);
	}
```

```
          class Data {
                    public Data(double[] input, double[] output) {
                              this.input = input;
                              this.output = output;
                    }

                    double[] input;
                    double[] output;
          }

}
```

14.3 A Very Short Summary of Scrum

This project used the Scrum development methodology. The best place to learn about Scrum is from the online guides maintained by the creators of Scrum:

- http://www.scrum.org/

This appendix will not attempt to justify and detail all of Scrum, but instead, to cover terminology and basic process that will be used on this project. The intent is to give the reader a crash course introduction should they previously had no contact with Scrum.

14.3.1 Basic Concept

Scrum is an agile methodology. Like most agile methodologies, it is iterative. Scrum calls its iterations *Sprints*. A sprint can last any amount of time, normally between a week and 30 days, but is always fixed in length for the duration of the project. In this project, sprints of 1 week was be used.

14.3.2 User Stories

The features of a product are described by *user stories*. A user story is a description of a feature from a user's (end-user, or external system) point of view. A single user story delivers a feature to the end product.

14.3.3 Backlogs

Scrum uses two backlogs, known as the *product backlog* and the *sprint backlog*. The product backlog is a list of all user stories for the product, and is prioritised according to the desirability of the user story in the final product. The sprint backlog is a list of user stories to be implemented in a sprint.

14.3.4 Estimation & Velocity

User stories are estimated when added to the product backlog. They are assigned a story point value. Story points represent the complexity of a story and how much work is expected to complete it.
When a sprint has been completed the number of story points that have been completed can be added up to give a value known as the *sprint velocity*. This velocity is used to predict the velocity in future sprints, and determine which user stories will be completed at what time.

14.3.5 Burndown Charts

A *burndown chart* is a graph showing the number of story points left on the y-axis and the date on the x-axis. There are two types of burndown charts: *product burndown* and *sprint burndown* charts, and they are different only in the scope of the x-axis.

Burndown charts are Scrum's primary way of predicting and visualising development progress. Velocities can be shown and end dates calculated. The power of the burndown chart relies on it being always up to date, and in many teams I have worked with they would draw it by hand on a white board on the wall next to them.
An example of a simple burndown chart can be seen in Figure 22.

14.3.6 The Process

Each sprint follows the following process:

Sprint Planning

- During *sprint planning* users stories are taken from the product backlog in priority order.
- Each story is broken into tasks and the tasks estimated (in hours)
- When enough tasks to fill the developers time have been assigned then the planning ends

Development

During development the developers work through their task. Every day begins with a 15 minute long meeting known as a *Scrum*. The scrum is a simple meeting covering only 3 things:

- What did I do yesterday
- What am I going to do today
- What impediments might prevent me from working

At this time the sprint burndown chart is also updated.

Review & Retrospective

At the end of the sprint all the completed stories are demonstrated, evaluated and confirmed. Any incomplete stories are returned to the backlog. At time the backlog may be reprioritised, or stories added or removed, based on the outcome of the review.
Also, a process called the *retrospective* takes place. This is intended to examine how the process has performed over the last sprint, and decide if any changes need to take place.

14.3.7 Summary

The information in the sections above should have familiarised the reader with the central concepts of Scrum. Further reading is recommended if the reader wishes to fully understand it.

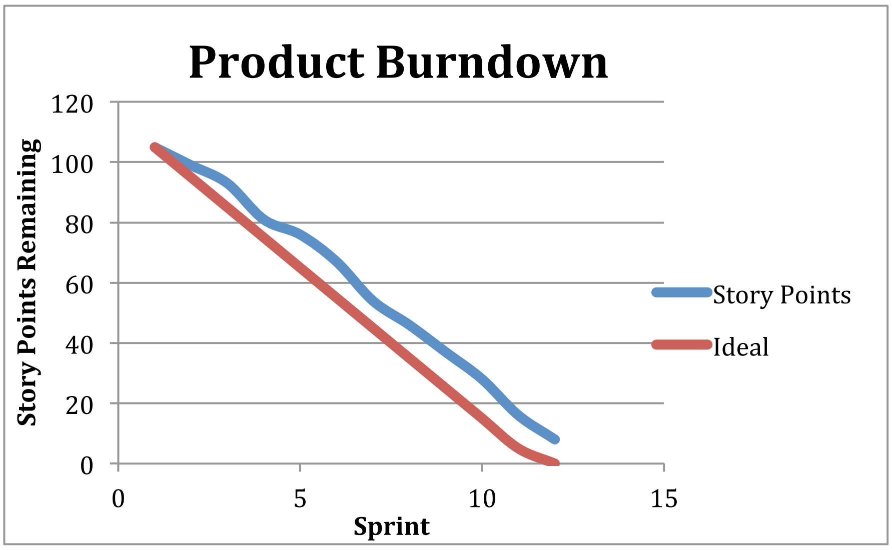

Figure 22 A simple product burndown chart