ANNE: Artificial Neural Network Editor

A Neural Network Modelling Tool

Peter Coetzee
Department of Computing,
Imperial College London
plc06@doc.ic.ac.uk

Fred van den Driessche
Department of Computing,
Imperial College London
jv06@doc.ic.ac.uk

Ismail Gunsaya
Department of Computing,
Imperial College London
ig106@doc.ic.ac.uk

Chris Matthews
Department of Computing,
Imperial College London
ctm06@doc.ic.ac.uk

Stephen Wray
Department of Computing,
Imperial College London
sjw06@doc.ic.ac.uk

ABSTRACT

In this report we will present a framework for building and operating large scale neural network models of aspects of brain activity and function. Further, we present a graphical tool to permit the user to specify the function of an individual neuron, the connectivity of the neurons at a global and local scale, and allow scalable simulations to be run on it. We will also present a framework for training these neural networks using a variety of methods.

1. INTRODUCTION

The original description for the project was as follows:

> *The aim of this project is to build a flexible tool for building large scale neural network models of aspects of brain functioning. The tool will allow the user to specify what the function of an individual neuron is, what the overall connectivity of the neurons is, and will then build the corresponding network and allow simulations to be run on it.*

There are three obvious key requirements from the initial description:

- Flexibility – It is important the tool is able to work with a number of different neural network paradigms; to facilitate this we designed our solution to be highly modular and pluggable with further extensions which would require no changes to be made to the core framework.
- High Detail Modelling – Users should be able to manipulate neurons on an individual level, as well as their connectivity. However, with the scale of networks desired performance and usability issues arise. The framework upon which ANNE is built was designed to handle large networks from the outset, enabling the best possible performance.
- Build and Simulate Networks – The ability to run and train networks, as well as save them to an intermediate format for interoperability with external tools. Spike Time Dependent Plasticity (STDP) was implemented for network training as well as a number of network execution features and data output features. For intermediate export, the standard XML-based NeuroML format was selected.

This work was conducted under the guidance of Murray Shanahan, as part of an Imperial College London, Department of Computing software engineering project. It was presented as part of the proceedings of the 3rd year group project seminar.

Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2AZ, United Kingdom

Neural Networks are involved in a number of areas of research, especially within areas of Artificial Intelligence in Computer Science, and in computational Neuroscience to attain a better understanding of how the brain functions. Neural Networks have the ability to learn relatively complex functions, and have a particular strength in dealing with noisy input data. Their applications can range from the simplest logical operators, such as *AND, OR* and *XOR* to complex facial and emotion recognition from photographs or facial markers.

One area where our solution is aiming to be used is with the iCub robot that the Department of Computing, Imperial College London has recently acquired. The iCub is a sophisticated robot which is designed to have the proportions and movement of a 3.5 year old child. The main goal of the iCub project, which is run by the RobotCub Consortium[1], is to study human cognition through the implementation of biologically motivated algorithms.

This is why there is an emphasis within ANNE towards biological networks which includes neurones such as Excitatory and Inhibitory Spiking Neurones. ANNE supports more traditional artificially inspired networks as well, but because of the pluggable nature of ANNE, adding support for new Neurone types is very simple, hopefully extending the useful lifetime of the application without the need for waiting for new release cycles to complete, as well as allowing other developers to extend and focus the tool to their own particular requirements.

Overall, ANNE provides a simple and intuitive user interface which gives the user a high degree of control over designing, training and simulating large-scale networks. It gives developers a solid framework which is highly pluggable, making application extensions trivial to integrate and distribute; thus providing ANNE with a longer lifetime and the ability to model whole new (potentially yet unconceived) network paradigms.

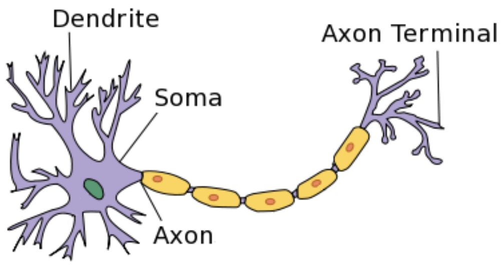


Figure 1: Biological Neurone

2. BACKGROUND

2.1 Biological Model of Neural Networks

Biological Neurones are cells in the nervous system and brain that process and transmit information through electro-chemical signals. They are the primary cell found in the brain and spinal cord of animals. There are many types of neurone and many of them interface with different aspects of a biological system, such as muscles or sensory receptors.

The basic neurone consists of a cell body called the *soma* and a long thin *axon*. The cell body has a dendritic tree which receives electro-chemical signals from other neurones. The axon has tree terminals which propagate the signal from the neurone to the next neurones. The signal is transmitted by the release of a neurotransmitter chemical into the *synaptic cleft* (or just synapse) which is a gap between these terminals and the dendrites of the next neurone.

Communication between neurones is called synaptic transmission. This is triggered by action potential, the propagating electrical signal which is produced by the electrically excitable membrane in the neurone. Synapses connect the terminals to the dendrites of neurones; they are capable of either increasing or decreasing the membrance potential of neurones they are attached to.

The human brain has about 10^{11} neurones with each on average having 7,000 synaptic connections per neurone[2].

2.2 Artificial Model of Neural Networks

An Artificial Neural Network (ANN) is a computer-based model representing a biological neural network. An ANN consists of a collection of neurons, interconnected by synapses. A neuron is in essence a mathematical function to model the output of a biological neuron in the brain, given a set of inputs. Synapses have weights (represented by ω) that are used as inputs for the neuron's mathematical function.

2.3 Types of Neurons

2.3.1 Perceptrons

The perceptron takes a vector of real input values (from the charge (x) from its input synapses; let these be represented as $c = x_1 * \omega_1 + x_2 * \omega_2 + x_3 * \omega_3 + \ldots + x_i * \omega_i$) and outputs a value to each of its output synapses depending on a threshold function. There are a variety of threshold functions[3] that can be used to influence the perceptron's behaviour, for example;

- The step function outputs 1 if c exceeds the threshold, 0 otherwise.
- The sign function outputs 1 if c exceeds the threshold, -1 otherwise.
- The linear function simply outputs c.

2.3.2 The Sigmoid Unit

An extension to the perceptron model, the logistic sigmoid unit, instead operates over a differentiable continuous output function. The logistic sigmoid function calculates the output as $output = \frac{1}{1+e^{-c}}$

This tends towards 1 as c increases, and towards 0 as c decreases; it thus is not entirely dissimilar to the step function, except in that it is instead a continuous *squash* function. Other squash functions are sometimes used, including those with some constant before the c term in the logistic sigmoid function, or using the hyperbolic tangent function *tanh*.

2.4 Network Topology

With the concept of the perceptron in place, the logical next step is to connect them in an Artificial Neural Network. The most common way of doing this with perceptrons is in a feed-forward layered graph. In this topology, the network is laid out as a series of layers of any number of perceptrons. Each layer is fully connected to the next (i.e. each perceptron in layer *i* is connected to each perceptron in layer *i+1*). There is thus a forward flow of charge, from the inputs to the network through each layer until the outputs are reached. The choice of number of perceptrons in each layer is important in deciding the potential accuracy of the network as a functional system.

2.5 Training

Training a network of perceptrons (be they sigmoid units or not) requires the notion of a set of *inputs* and *targets* that the network must learn; as such, it is a supervised learning paradigm. Within this, any number of algorithms may be used to perform the actual training. Typically these algorithms run for a given number of maximum iterations, or until some "stop" condition is met; e.g. a target accuracy.

2.5.1 Random Training

Perhaps the simplest (and least efficient!) training methodology is the random trainer; it simply modifies the synaptic weights at each level of the network randomly and re-runs the inputs through the network. If the accuracy of the network has improved as a result of the weight changes, the changes are deemed a success and kept; otherwise they are rolled back and the process repeated.

2.5.2 Back Propagation

The first step in back-propagation training is the "feed forward" step. In this, the inputs are run through the network and its outputs are compared to the targets. The trainer then calculates an error value for each output. The synaptic weight of each synapse between the output layer and the one before it are used to determine how much each synapse "contributed" to this error, and thus by how much it should be altered to compensate for the error. At this point the trainer requires a notion of the "learning rate"; the proportion of the error by which to alter the synaptic weights. These factors are all combined to decide the amount by which to alter the synaptic weights at this layer. Next comes the back-propagation step;

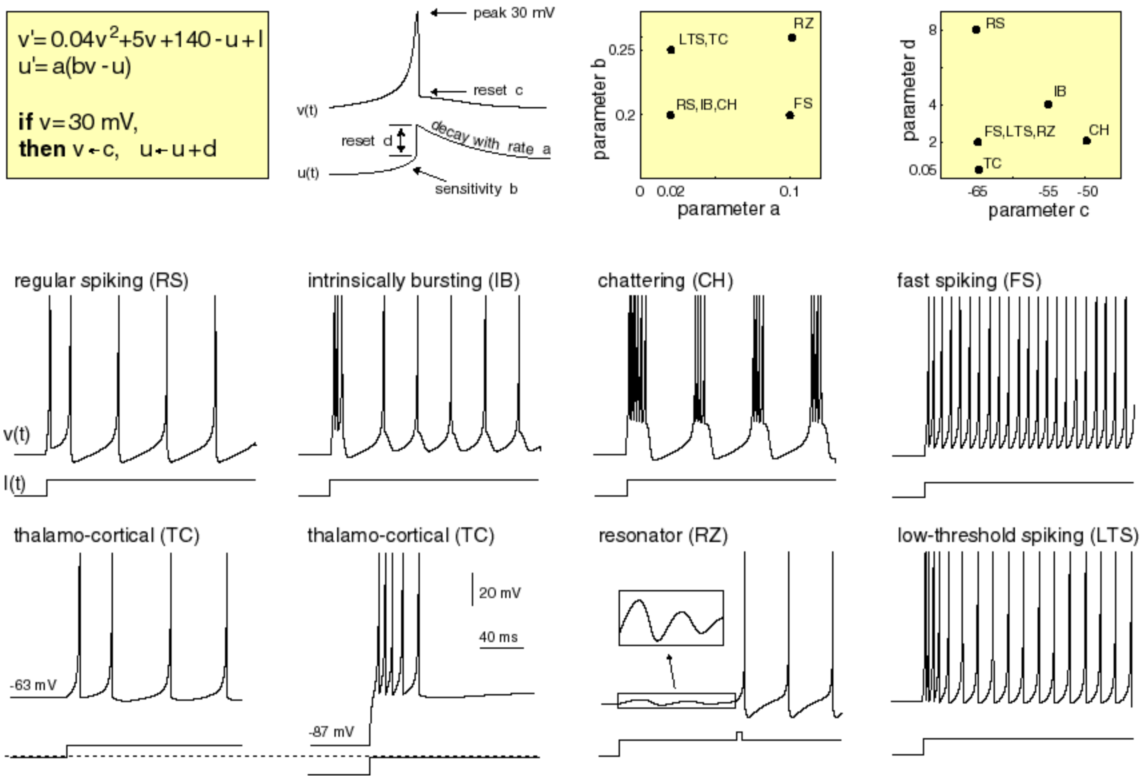


Figure 2: Spiking Neurone Behaviour

these weighted errors are propagated back to the previous layer of the network as their "target error", and the process repeats. This continues until the input layer is reached.

2.6 Advanced ANN Models

2.6.1 Spiking Neurons (Excitatory and Inhibitory)

In reality, biological neurons are not nearly as simple as the perceptron's model of them. Perceptrons fail to model the temporal aspects of firing in the brain; neurones take time to charge and then fire in spikes across their synapse. This also takes time. Furthermore, they then have a period of reduced susceptibility to charge – a so-called "recovery period" in which it requires a very great amount of charge indeed to cause them to fire. Finally, there are multiple types of neurones – those that excite other neurons, and those that inhibit their firing.

Hodgkin and Huxley[4] first modelled these neurones mathematically, but their model was too complex to be able to scale and compute with. Eugene Izhikevich pioneered a simple model (Figure reffig:anns:spiking[1].) of spiking neurones[5] that was efficient enough to run large networks (˜1000 neurones) in real-time with millisecond precision. It almost exactly models the spike-timing dynamics of the observed firing patterns in a rat's cortex.

2.6.2 Polychronization and STDP

A later Izhikevich investigation[6] showed that it was possible to compute using these spiking neurones, and to train the network in a very similar manner to how a biological brain learns. The fundamental principle of STDP, or Spike-Timing-Dependent Plasticity[7], is the Hebbian-based learning rule; to increase the strength of synapses between neruons that fire at approximately the same time. By the same token, synapses between neurons that fire at very different times are decreased in strength. In this way, the network forms a sort of spatial and temporal associative memory.

Suppose one were to present a spike of input at a cluster of neurons **A** on a randomly initialised homogenous network. It may cause a few other neurons it is connected to fire semi-randomly. If one were then to present input spikes at two clusters simultaneously, **A** *and* **B**, and train the network with STDP then it would learn that association. If a spike were presented at either **A** or at **B** then the network would spike automatically at the other.

3. ARCHITECTURAL DESIGN

3.1 Design Rationale

As highlighted in our introduction, it is integrally important that the ANNE system is architected in such a manner as to be modular, pluggable, and highly configurable. To achieve this, the architecture of the system needed to be carefully selected at the start of the design process, and strict principles maintained throughout.

Two early decisions were taken to facilitate this. The first of these was to implement the primary components of the system as singleton *services*, thus making it easy to reduce coupling and implementation specifics. Furthermore, we elected to split ANNE into three distinct components. The first of these would be the **Framework**; this provides basic programming constructs that would be useful for many applications, and is in no way specific to ANNE or to

[1]Electronic version of the figure and reproduction permissions are freely available at www.izhikevich.com

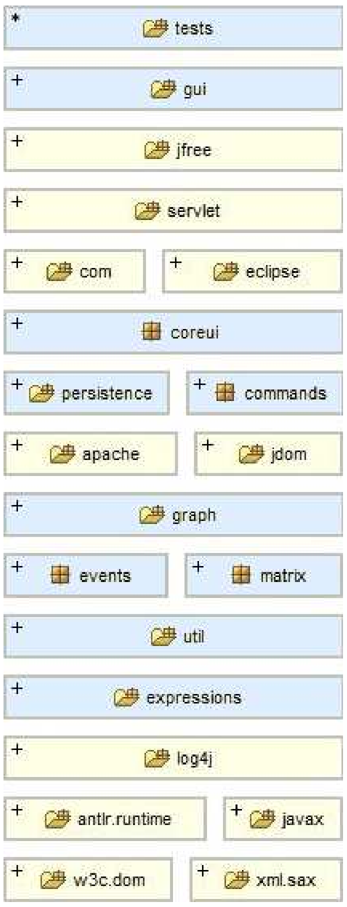


Figure 3: Structure chart of internal (blue) and external (yellow) dependencies

Neural Networks. On top of this was constructed a set of Neural network APIs, and a graphical user interface built atop the Eclipse SWT widget library. This forms the **ANNE Environment**. The final section of the project are a collection of **Plugins**, designed to extend the behaviour of the basic environment and facilitate complex and user-friendly interaction with the Neural Network.

The package structure of the project was designed to permit separation of these three concerns across their relevant sections. Plugins pertaining to the GUI, for example, belonged in a sub-package of *gui*. Basic utility classes belonged in a *util* package. Over-all, the namespace for the entire project was (according to standard Java naming convention), *uk.ac.ic.doc.neuralnets*. With a correctly designed structure, it should be feasible to remove any package of interest, along with its dependencies, and to utilise it outside of the rest of the ANNE environment. Furthermore, it is integrally important that dependency leakage is avoided; no GUI-specific libraries or components may appear in the Neural Network packages, for example, so that the neural network may be run in a headless server environment, or as part of some alternative UI.

The final architecture's separation (and its full dependency layering) can be seen in Figure 3. As is clear from the lack of back-references, it is simple to (for example) replace the entire GUI package, built on SWT, with a CLI interface, or even one based on a client-server technology. This is made clearer with the package

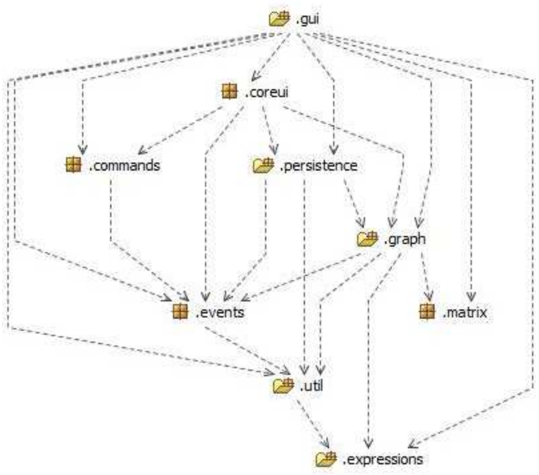


Figure 4: Internal package dependencies

dependency hierarchy in Figure 4.

3.2 Optimisation

This final architecture is by no mans a "perfect first draft". A large number of design iterations went into removing all back-references from the architecture and ensure packages were completely self-contained. One invaluable tool in performing this optimisation was the commercial Structure101[8] (also responsible for the above diagrams). This excellent tool highlighted any back-references and tangles in our code, and facilitated the process of repairing our structure. Unfortunately, it is an expensive piece of software and could only be employed for a 15-day trial period; for this reason it was not used until the end of the project's life-cycle, in its final refactoring stage. At this point the architecture was approximately 50% tangled, but had strong cohesion. By the time refactoring was complete, it exhibited no tangles at any package level. The only remaining tangles were of two or three classes, such as those which required (for efficiency of implementation and development) pointers to each other, such as in our directed graph implementation.

The most significant piece of this refactoring was in the GUI package. Most activity in the GUI was processed by a single facade class, the *GUIManager*. To remove this absolute dependency from all UI classes that were not SWT specific, two interfaces were created in a new *coreui* package, which abstracted out the behaviour of an "Interface Manager", and one that was capable of Zooming into a Neural Network. By doing this, the Main class of the application could perform dependency injection of the *GUIManager* into these classes.

4. FRAMEWORK

4.1 Introduction

As discussed previously, the framework of the system (Figure 5) provides a basis of eight key services and interfaces upon which the rest of the system is built. These are completely generic and capable of providing their functionality to any type of application.

4.2 Expressions

The first of these services was for Expressions. These are provided as a set of tools for evaluating simple mathematical expressions,

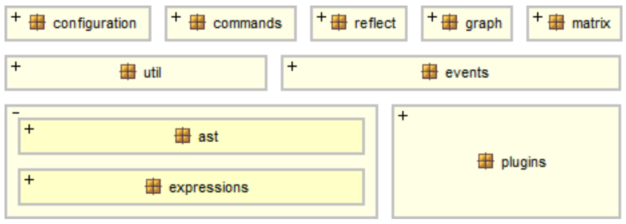


Figure 5: Framework Architecture

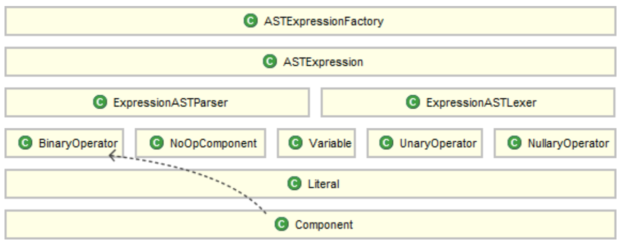


Figure 6: Framework: AST Expressions

including support for dynamic variables and a variety of built-in functions (hyperbolic, trigonometric, exponential etc.)

There are two versions of this package available; the first supports a simple “parse-and-evaluate" model, while the second version builds an abstract syntax tree and has better support for variables in expressions, as well as being more efficient. We will here focus on the AST Expression package.

AST Expressions are built on top of the ANTLR parser generator. This is used to convert a grammar into the parser and lexer, seen in the middle of Figure 6. Below this the abstract syntax tree can be seen; it is comprised of a tree of Component objects. These are either Literal values, or functions (be they nullary, unary, or binary). They are responsible for performing the operation assigned to them, as instructed in the parser. They also store pointers to their child nodes, and are capable of identifying any variables within themselves (recursively).

The abstract Component type has some knowledge of the types of expressions, and uses this to do simplification in bracketing according to the standard mathematical order of operations. While parsing occurs, the constructors of the operators all handle simplifaction of constant term expressions as far as possible.

The ASTExpression class wraps the behaviour of ANTLR and the syntax tree in a simple to use object. When using it, a developer simply passes in an expression to parse and it will instruct ANTLR to parse it. It can then bind variables based on the contents of an *@BindVariable* annotation. This annotation permits a developer to annotate their code with the ‘getter’ methods to bind variables, the name of the variable to bind to, and (optionally) whether the variable is dynamically bound (i.e. whether to re-bind its value in the tree every call, or just once).

Extensive profiling was conducted over the AST Expression classes to optimise their operation, as they are fundamental to the operation of the neurone classes, as we will see later. One of the primary causes of inefficiency was the Java reflection system for annotations; caching this data increased the speed of these by approximately 20%. The efficiency was further improved by altering ASTExpressions to be created as Flyweight objects, cached in a *HashMap* by the *ASTExpressionFactory* service.

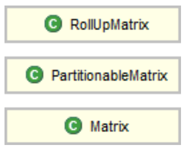


Figure 7: Framework: Matrix Package

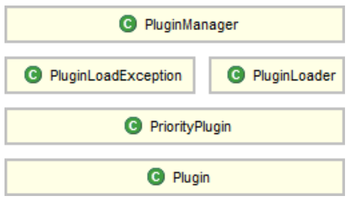


Figure 8: Framework: Plugin Management

4.3 Matrices

The Matrix package (Figure 7) offers three utility classes for dealing with vector or matrix data. The *PartitionableMatrix* class extends this with the concept of *partitioning* a matrix such that it ‘appears’ to be only a sub-matrix of itself. Finally, the *RollUpMatrix* is capable of ‘rolling up’ to a particular size; that is, it transforms itself from a Matrix<T> to a Matrix<Matrix<T>> of smaller dimension, where the original matrix is split into a series of equivalent size sub-matrices in the new output.

4.4 Plugin Management

The plugin system (depicted in Figure 8) is the bedrock upon which the rest of ANNE is built. It forms the basis for all of its flexibility, modularity, and extensibility. As such it was integrally important that it was designed to be as effective as possible, whilst remaining easy to use – both in code and for the end-user of the system.

The guiding principles in the design of the plugin system were speed and ease of extension. It needed to be able to load plugins without a perceivable lag in responsiveness, and it needed to be easy for users to add more plugins once they have installed the system. In order to achieve this, it was decided that the file-system would be used to organise the plugins, according to Type and Plugin Name. Thus, we have a directory hierarchy of:

```
plugins/InterfaceType/PluginName.class
```

For example, a user might have:

```
plugins/SaveService/XML.class
plugins/SaveService/Serialisation.class
plugins/LoadService/XML.class
plugins/LoadService/Serialisation.class
plugins/Trainer/BackPropagation.class
...  et cetera.
```

In this way, adding a new plugin to be loaded is simply a matter of dropping it into the correct plugin directory. The plugin *.class* file is simply a compiled Java class, as output by the *javac* compiler. The

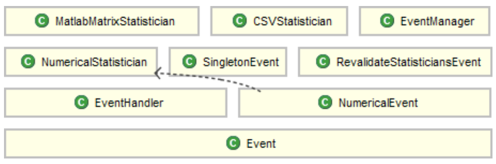


Figure 9: Framework: Event Handling

plugin name need not be the same as its class name; this should be a more user-friendly name to describe its purpose.

From a developer's perspective, the *PluginManager* offers two primary utilites. First, it is capable of listing all plugin names of a given type. This permits a developer to dynamically expand their code as the plugins are added. Secondly, it will instantiate an instance of a plugin for the developer based on Plugin Name, and its type. Using Java Generics it is feasible to do this in a type-safe manner, without the developer having to cast, increasing the safety and integrity of their code.

When a plugin is requested, the manager forwards the request to a custom ClassLoader, the *PluginLoader*. This first checks its cache for that class. If it is not found, then the loader attempts to read it from disk, and use the standard Java ClassLoader methods for defining a class from a stream of bytes. It is possible that this may throw a LinkageError if that class has been previously defined; for example, if an objcet in memory has a reference to a plugin class directly, not through the plugin manager, then it is possible that the class is defined twice. In this case, the loader can simply fetch a Class of the source type from the JVM, cache it in its internal cache, and return it. The Plugin Manager then generates a new instance of this class, and uses the *Class.cast()* method to cast from the returned Object to the requested type.

In general, profiling shows that the PluginManager is very efficient, and generally either hits its cache, or spends most of its time waiting for a Java native method to define the class. One area in which it could be improved is that of dependency management. It currently has no concept of the dependencies of a given plugin; if a plugin depends on other classes (or even contains inner classes), these *must* be available on the class path. One could envisage an improvement to this that were capable of loading plugins with a manifest from a JAR file. Initially we avoided such an implementation in an effort to speed up the code (decompressing the JAR may slow down the implementation significantly), instead opting for a run-script to assemble dependencies from a `plugins/lib` directory.

One extension on the original design that was implemented through the project's life cycle is the notion of an abstract *PriorityPlugin*. This includes a numerical "Priority", and falls back to ordering based on the lexical form of the plugin name. The plugin implements the Java *Comparable* interface, permitting the plugins to be placed directly into a *SortedSet* and retrieved in their specified order. Nothing prevents a plugin developer from overriding this *compareTo* method, should they require some alternative behaviour.

4.5 Event Handling

The Event Handling subsystem (represented in Figure 9) provides a generic means for events to be fired, and to be handled by zero or more pluggable event handlers. Its design permits for a handler to by synchronous ("Do not return control to the firing method until this handler has finished executing") or asynchronous ("Execute this handler as soon as possible after the event is fired, but it is not totally time critical"). There need be no lmitation on how much (or, indeed, how little) data is stored in an event; the entire object should be simply passed into its registered handlers in turn to handle it.

The *EventManager* is implemented as a singleton, containing two sets of mappings from Event Class to a List of *EventHandler* objects; one representing the asynchronous handlers, and one for the synchronous. When an event is fired into the system, it is first added to an event Queue (in fact a *LinkedBlockingQueue* for concurrency control) for processing by the asynchronous handlers. It is then passed into a *handle* method, responsible for firing an event at each of its handlers from a given mapping. This same method is used by the asynchronous dispatcher thread, which *take()*s an Event from the end of the queue before handing it off to the same *handle* method.

When an event is handled, first its class is determined, and it is fired at all handlers registered for that Event class. If the super-class of that event class also happens to be an Event class, the handle method is recursed upon to fire the event at all handlers registered for the super-class, and so on until the given class is no longer a sub-class of Event. In this way, a more general Event type may be used in order to require a handler to be registered for multiple event types at once.

One example of this in use is a special Event type; the *RevalidateStatisticiansEvent*. This is an abstract event class which has a handler registered by the *EventManager* itself. This handler iterates over the collection of all handlers registered to the *EventManager*, checking to see if they are valid. An *EventHandler* may inform the *EventManager* that it is no longer valid and thus cannot accept events. If this is the case, when a *RevalidateStatisticiansEvent* is fired, the *EventManager* may re-create these *EventHandlers* using the *PluginManager*.

A pair of convenience classes are provided; the *NumericalEvent* interface, and the *NumericalStatistician* abstract class. These permit a statistician to access so-called "numerical" events as rows of data. This is the principle the CSV and MatlabMatrix statisticians operate over; the implementing event simply must provide one 'row' of data for the statistician to process. It does this by having the *Event* push its data through into the *NumericalStatistician* as a var-args method call, thus making it easy for the event author to provide their data, and for the handler developer to process it. The default implementation simply stores a list of lists of Double values, for ease in later processing.

When profiled, it was determined that a large portion of the time spent in the *EventManager* was spent awaiting locks over global resources; it tended to use *synchronized* methods, or *synchronized()* over the *EventManager* object for concurrency control. After some reasoning, it was determined that these locks could be tightened to last for less time, and to be more specific. A significant performance boost was gained by moving these to be *synchronized()* locks over the specific resources required (e.g. the *asyncHandlers* map)

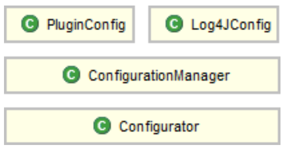


Figure 10: Framework: Configuration System

CommandControl
Command
CommandEvent

Figure 11: Framework: Commands Package

4.6 Configuration

The next significant consumer of *Plugins* is the Configuration system (Figure 10). This is designed to be a general-purpose means by which features of the software (including other plugins) can be configured at run-time. The primary *Configurators* in the system are to configure Log4J, the logging system, and the types of Neurone available. The format of configuration files are currently entirely up to the *Configurator* developer; a future iteration of this design could perhaps provide more structure, and alleviate some of the strain of developing configuators by handling the parsing of the configuration file before the configurator is invoked.

4.7 Commands

The ability to undo mistakes is useful for all editor software, so early on in the project the decision was taken to include full undo and redo functionality. This feature is usually implemented in one of two ways: by recording internal state after each action then rolling back to a previous state when an action is undone (the Memento pattern), or by giving each action an undo method (Command pattern) that reverts all changes when called.

Saving and loading of network state already needed to be implemented in the persistence package, so it would be trivial to utilise this to record the current neural network after each action has been executed. However, large-scale neural networks can result in extremely large data structures and the time spent waiting for one to save after every action would make the program so slow as to be almost unusable. Creating an inverse method for each action requires more effort to develop, but the resulting performance is vastly improved.

Because of this, we implemented our undo and redo functionality using the Command design pattern (see Figure 11). Each action that the user can execute and undo is encapsulated as a *Command*, which is a *Runnable* object containing *undo()* and *execute()* methods. These *Commands* are managed by a central *CommandControl* instance, which maintains stacks of *Commands* that can be undone and redone as well as handling their execution to avoid deadlocks and concurrent edits.

This concurrency safety is handled by a dispatcher within the *CommandControl*, which runs in its own thread and contains a multi-threading safe *BlockingQueue* of commands to be executed. *Commands* are added to this queue when they are created, undone or redone, and they are dispatched sequentially so no problems arise from simultaneous edits to a single data structure. An event is fired when execution of a command finishes so any interested components in the system can be informed of the changes. For example, the undo and redo buttons on a GUI's toolbar can check whether or not the undo and redo stacks are empty and enable or disable themselves accordingly.

Although each action is a separate command, many of them share functionality. For example, the *execute()* method of a command to delete items, and the *undo()* methods of a command to add items are functionally equivalent to each other. This behaviour can (and in general has in our implementation) been factored out into a single operation class, and is available to be used in any future actions that may require it.

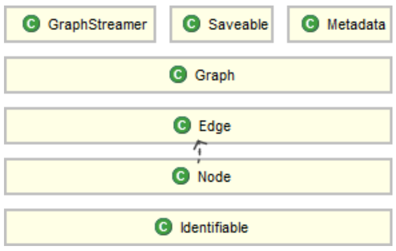


Figure 12: Framework: Graph Package

4.8 Graphs

As a modelling tool for Artificial Neural Networks, it was deemed important that ANNE was built upon a solid, but not overly complex, Graph implementation (Figure 12). This implementation is essentially a directed *Graph*, consisting of *Nodes* and *Edges*. The interface for *Node* is generically parameterised with its *Edge* type, and vice-versa; the *Edge* is parameterised with the types of the *Nodes* at either end.

It was decided, for ease of use, that *Nodes* and their *Edges* should maintain back-pointers to each other, so that it was possible to navigate the graph from *Node* to *Edge* to *Node*.

All elements in the *Graph* implement the *Identifiable* interface, which provides convenience methods for accessing and processing their universally unique identifiers. These permit any implementing code to refer to graph components by a persistent ID, rather than simply by memory pointer; of particular utility in, for example, statistical systems.

The *GraphStreamer* class provides a convenient way for a developer to provide a *Graph* and two transformer classes from *Node* to a type of their choosing, and *Edge* to another type of their choosing. These can then be used to stream the contents of the *Graph* through to another type, in order to support simple type transformations (in a similar vein to a functional-programming '*map*' function).

The final portion of the *Graph* implementation of note is the Metadata system. This provides a simple but effective way of storing any information that is not directly a component of the *Graph*, or its *Nodes*, but that is still relevant. It stores its data as a collection of simple String <key, value> pairs.

4.9 Reflection

The final framework package we will cover is a set of Reflection helper methods. It was deemed important that the software solution's persistence system (to be discussed in detail later) was capable of retrieving and setting fields in an object in a uniform manner. One side effect of this work was to make it possible for a developer to retrieve and set values on a private Field, using Sun's *ReflectionFactory*. Once this was done, it was possible to retrieve a Field object by reflection for the given class, and to mutate its internal private *FieldAccessor* (responsible for reflecting into the field and getting / setting values) to instead operate over a method. This essentially provides JavaBeans-esque functionality to the Java programmer, permitting them to consider Methods and Fields as one collection of "data".

To accomplish this, a few assumptions need be made. The *MethodPseudoAccessor*, responsible for permitting a *Field* object to back-end its logic to a *Method*, attempts to seek setters and getters following the standard pattern of field "someField" having mutators "setSomeField(value)" and getter "getSomeField()". The type of "value" in the previous may be any of *Double*, *Integer*, *String*, or *ASTExpression* in our implementation. It is, however, feasible to extend this to support more types.

The absolute requirement for this system to function over *Field* was, in fact, rescinded during development as an XML persistence library the project depended on ceased to be available. This library would only operate over arrays of Java *Field*s. However, the package's utility was deemed to be sufficiently great that it was kept. In a future iteration of the software, a cleaner and more portable solution to this problem would be to define a "Datum" type, which can back-end its getting and setting of values to either a *Field*, or a *Method*.

5. NEURAL NETWORKS

5.1 Overview

With the solid foundations of the framework in place it was possible to create a fully-featured neural network model for our system. The model was required to emulate Izhikevich's spiking neurone networks as well as classic feed-forward artificial neural networks, both in their contstruction and execution. As the model would have to scale to contain millions of total components an efficient design was required to minimise resource usage and maintain responsiveness for the user. It was also required that neurone parameters be fully configurable during general use of the application.

A correspondence between Izhikevich's network model, specified in Matlab, and our own model needed to be created. Izhikevich's model relies on a number of matrices to hold data describing the neurone charges, synaptic weights and synaptic delays, as well as vectors to describe other neurone parameters. Since Java was being used it seemed apparent to model network components as discrete objects rather than using two-dimensional arrays in a more direct 'port'. This design also enhanced the customisation and specialisation options for neurone parameters on a per neurone basis rather than requiring constant values for all neurones in a network.

Neural networks are executed, or run, by performing discrete **tick** operations on them repeatedly. The ticks are propagated down from the root network to all components. For example, in a feed-forward network a single tick involves propagating charge from the input neurones, through the hidden layers, to the output neurones. In the spiking model a tick involves adding a random thalamic input to all neurones, calculating which neurones have fired and moving charge along the synapse from one neurone to another. This thalamic input is intended to model the activity of the thalamus in the brain; the portion of the brain devoted to processing external stimuli. Its random nature in Izhikevich's model is simply to cause network activity. When attempting to compute using these networks, large-scale random input is obviously not desirable.

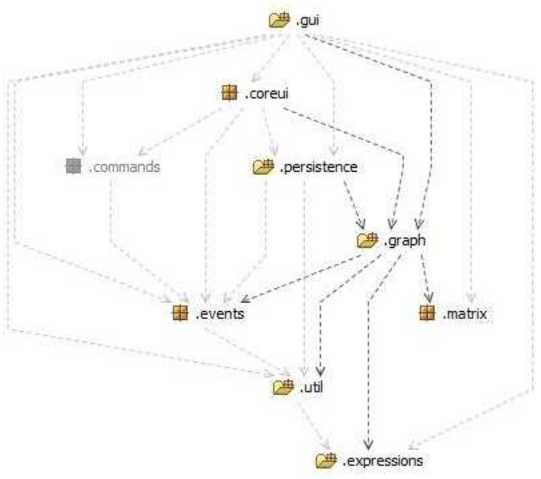


Figure 13: Dependencies on the Graph / Neural Network package

5.2 Design

There are four classes central to the model: *Neurone* and *Synapse*, *NeuralNetwork*, and *NetworkBridge*. Their implementations are specialisations of the general Graph service in the Framework.

The lowest-level building block of a network is the *Neurone*. *Neurone*s are implementations of the *Node* interface from the Graph framework, through a basic implementation of the abstract *NodeBase* class. While *NodeBase* controls aspects concerning abstract node connectivity in a network, *Neurone* specifies the parameterisation such as squash function, trigger values and tick behaviour.

The *Neurone*s are connected together with *Synapse* objects to form the basic network structure. *Synapse*s extend *EdgeBase*, which provides a fundamental abstract implementation of *Edge*, simply adding synaptic weights.

NeuralNetwork is a specialisation of the *Graph* framework class. It polymorphically implements the *Node* interface so that a *NeuralNetwork* can contain both *Neurone*s and other *NeuralNetwork* nodes, allowing for self-containment similar to the directory structure on a file-system. *NeuralNetwork*s also have the ability to tick, which is propagated to all *Neurone*s and sub-networks contained within it.

*NetworkBridge*s are *Edge*s that connect together *NeuralNetwork*s. They contain a bundle of *Edge*s that connect *Node*s inside the networks linked by the bridge. *NetworkBridge*s are created implicitly when a node from one network is connected to a node from another network. For example, if *Neurone* **A** in *NeuralNetwork* **X** is connected to *Neurone* **B** in *NeuralNetwork* **Y** with a *Synapse S* then **X** and **Y** are first connected with a *NetworkBridge* **R**, which contains *S*.

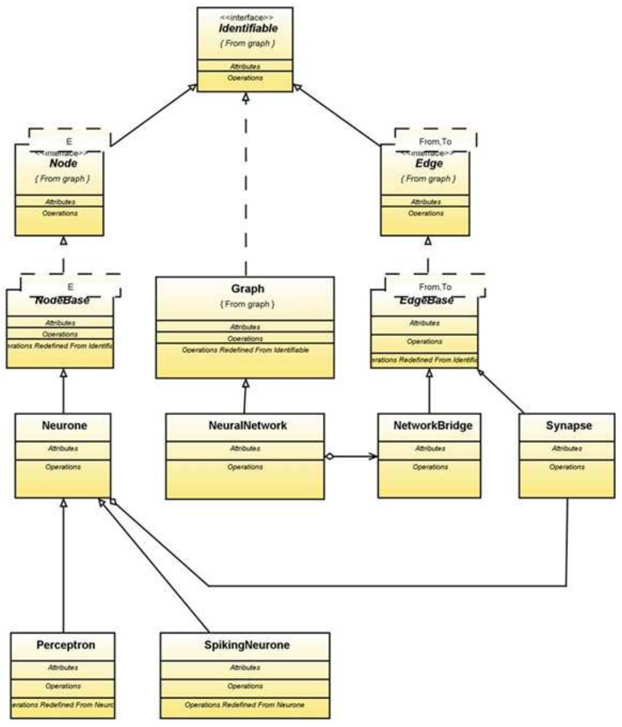


Figure 14: UML Class Diagram of Neural Network Implementation

There are two further specialisations of *Neurone*: *Perceptron*, which is for use in feed-forward networks, and *SpikingNeurone* which is the base class for use in spiking networks, and adds the parameters expected by Izhikevich's model.

Neurone parameters can be configured both while the application is running and offline. While the application is offline modifications can be made in the plain text file *nodetypes.cfg* in the *conf* directory. During program use the *Neurone Designer* can be used; changes made in the designer can be propagated automatically to *nodetypes.cfg*, using the Event System. This also makes it feasible to ensure UI components are informed of changes in a neurone type's parameters.

The creation of neural network components is controlled by specifications and factories in the *manipulation* package. *Neurone*s are created by the *NodeFactory* using *NodeSpecification*s. The *nodetypes.cfg* file is parsed at application launch time by the *NeuroneTypeConfig* configurator which loads the neurone parameter sets into the *NeuroneTypes* registry. A *NodeSpecification* can be requested from the *NeuroneTypes* registry for a named neurone configuration. This *NodeSpecification* can subsequently be passed to the *NodeFactory* to create a concrete node instance. *NodeSpecification* is extended by *SpikingNodeSpecification*, *InhibitorySpecification* and *PerceptronSpecification* which provide default values for these implementations.

*NodeSpecification*s make extensive use of the ASTExpressions service of the Framework. This allows arbitrarily complicated mathematical expressions to be used for neurone parameters where necessary.

*Edge*s are created by passing an *EdgeSpecification* to the *EdgeFactory*.

*NeuralNetwork*s are created in a similar fashion, using *GraphSpecification* types and the *GraphFactory*. A *HomogeneousNetworkSpecification*, a concrete *GraphSpecification*, is created with one or more *NodeSpecification*s and associated node counts as well as an edge probability. The edge probability defines the probability that a given node will be connected to another during network creation. This specification is passed to the *GraphFactory* which is responsible for calling the *NodeFactory* to create the various types of neurone objects, interconnecting the neurones with edges, and returning the resultant objects encapsulated in a *NeuralNetwork* object. It is also possible to invoke the *GraphFactory* with a custom transformer, which will select how to connect neurones together.

Also in the *manipulation* package are the *InteractionUtils*. These are a collection of miscellaneous convenience tools for interacting with *NeuralNetwork*s in various important ways. There are utilities for finding the network that directly contains a given node, finding if a given network **A** contains a given network **B** and finding the lowest common ancestor of two nodes, i.e. the first network that contains both nodes. There are also tools for network birfurcation, connecting two nodes or two sets of nodes in a network, automatically creating the any required *NetworkBridge*s, either fully or in a one to one manner. A control thread for concurrent running, stopping and resetting of the network is also contained in the *InteractionUtils*.

5.3 Input and Output Nodes

The I/O Node system provides a simple and extensible way for running data external to the application through a neural network. For example, an *InputNode* could be created to hash images into data arrays which could then be fed through a network. In a similar way, an *OutputNode* could be created to convert network output to database identifiers and retrieve a database record pertaining to the recognised image; e.g. for application with a facial recognition network.

*InputNode*s provide a matrix of data that is run through the network row by row. They can also provide a matrix of targets that can be used during training, each row of the target matrix corresponding to a row of the data matrix. *InputNode*s are *Foldable* (another interface) denoting that they support the notion of N-Fold testing. N-Fold testing divides the training data into two subsets: one that will be used for training and one for testing. After one fold of training and testing new subsets are created. This process is repeated until each item is used for testing. For example, for a data set of 100 rows, 1-90 are used to train and 91-100 to test, then 1-10 and 21-100 to train and 11-20 to test, and so on until all 100 rows have been used. This folding is implemented using the *PartitionableMatrix* from the Framework.

*OutputNode*s create a specified number of nodes which call an abstract *fire* method when they fire, passing their index and charge. This gives the concrete node implementation full control over the data flowing out of the output nodes. A call to the abstract *setNodes* method allows an *OutputNode* to configure its internal systems when created.

Both the *InputNode*s and *OutputNode*s provide *recreate* and *destroy* housekeeping methods. The *recreate* method may be invoked when configuration data is already in memory and the user need not be

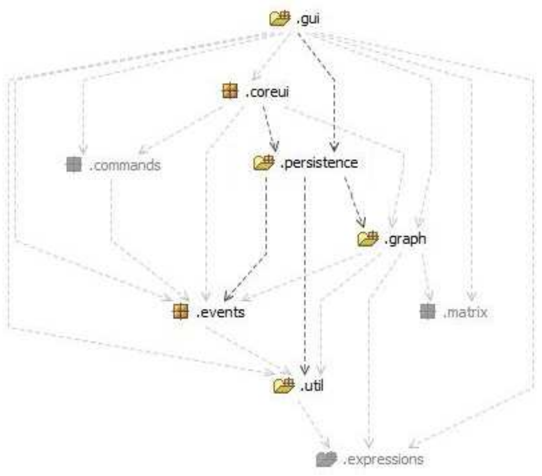
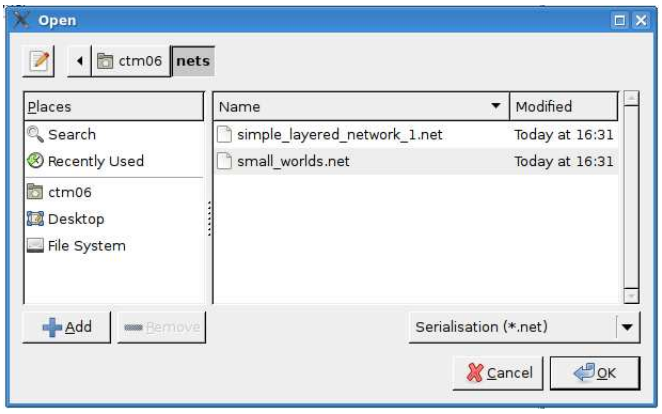


Figure 15: Persistence's location within the framework.



Name	Modified
simple_layered_network_1.net	Today at 16:31
small_worlds.net	Today at 16:31

Serialisation (*.net)

Figure 17: View of Open Dialog

prompted for it again. *Destroy* is a tear-down method, called when the node is removed from the display.

When I/O Nodes are added to a network they must be connected up using the standard User Interface tools to do so; *InputNode*s and *OutputNode*s are both extensions of *NeuralNetwork*, so can be manipulated within the network in exactly the same way.

5.4 Training

The training of neural networks is handled by pluggable *Trainers*. The basic interface defines methods for setting the *InputNodes* for the training data, the number of network ticks for a single test, a *trainOnce* method that performs a single test run, and a *trainFully* which trains to a specified accuracy / maximum iteration count.

One basic abstract implementation of the *Trainer* is the *Stepwise-Trainer* which allows for housekeeping, such as synaptic weight adjustment, to be performed between each training iteration. The back-propagation trainer, random trainer and the granular random trainer all extend the StepwiseTrainer. The STDP (Spike-Timing-Dependent Plasticity) trainer implements the *Trainer* interface itself.

5.5 Events

The neural network package uses the *EventHandler* extensively so that other system modules can be notified of network events. During network creation, *NodeCreatedEvent*s and *EdgeCreatedEvent*s are fired when nodes and edges are created respectively, to provide progress information to the UI. When a network starts or stops running a *NeuralNetworkSimulationEvent* is fired and *NeuralNetworkTickEvent*s are fired each tick. Every time a node fires, a *NodeFired* event is triggered and *NodeChargeUpdateEvent*s are fired when nodes are charged. *NewNeuroneTypeEvent*s are fired when the Neurone Designer creates a new neurone type.

6. PERSISTENCE

6.1 Overview

Figure 15 provides a view of how persistence rests within the application framework in terms of its dependencies and the modules that depend on it. For obvious reasons, the user interface is dependent on the persistence module being present for the ability to save or load neural networks which have been designed within the application. persistence depends on the model of these networks, a few core utilities and events for serialization and de-serialization of networks. This is coherent with the low coupled design of the framework.

Figure 16 is an overview of all the classes and interfaces within the persistence module and their package separation. XML and TNS have their own packages as they require specific classes for their implementation, whereas the Java Serialization services make use of the Java's in-built ability to serialize objects to a persistable storage location. X3D is dependent on the XML system but required no other external classes apart from the base service for its persistence.

6.2 Design

The persistence layer has been designed, like many aspects of the application to be completely modular and pluggable with new services. Furthermore its architecture is such that it is not, in its abstract form, directly related to saving and loading of neural networks. Instead it simply relies on the concepts of a "Saveable" object, "Persistable" data, and "Specifications" of saving, leaving the implementation details entirely up to the particular service. Adding a plug-in requires no existing code to be changed and the new plug-in to extend the *S*aveService for saving networks, or *L*oadService for loading networks. These abstract classes provide the method headers for creating persistence services. A new plug-in doesn't have to implement both loading and saving, as is the case with the X3D service which is only available for exporting networks. A number of persistence plug-ins are provided with the standard distribution, they are:

- *Java Serialization[9]* – The network is serialized using the Java Serializable and ObjectOutputStream interfaces; this is used as the default persistence option and preferred over the other modules, especially when not transporting the neural network to another application. This is because it is extremely fast and guarantees that all information is persisted and will be loaded again. As the serialized network consists of the actual in-memory objects of our implementation, the only applications that would be able to make use of this format are those which use our framework for network representation. Files are given the *.net* extension.

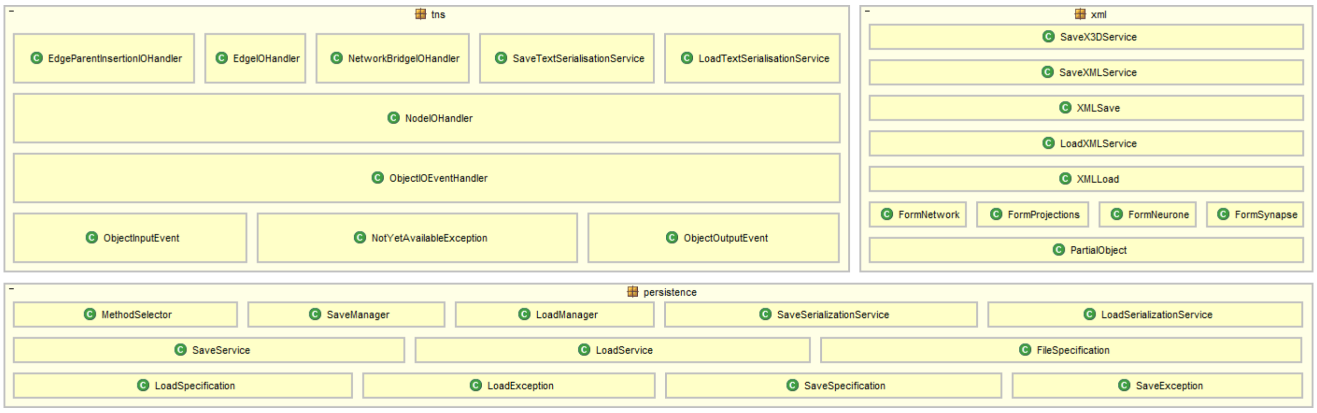


Figure 16: Persistence Modules Implementation.

- *NeuroML[10]* – This is the main persistence module for exporting to a number of other applications that also support the standard NeuroML Layer 3 Network Schema for representing in neural networks. Import is also supported for taking NeuroML from other applications that also use this schema, allowing for some cross-application portability. The XML that is generated has been verified to be well-formed and NeuroML validated, meaning that other applications should accept this format without errors. Files are given the *.xml* extension.

- *TextNetworkSerializer* – A simple fact-based export and import specification which outputs records of all the objects into a plain-text format for storage. Like Java Serialization, however, it is a non-standard format which is only implemented by our framework so cannot be used for exporting to or importing from other tools. It does however have a low storage overhead compared to XML and performs faster than the Java Serializer. Files are given the *.tns* extension.

- *X3D* – A royalty-free open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML[11]. We implemented only an export feature to X3D as applications that implement this standard are designed for viewing as opposed to modelling items. Furthermore the feature is implemented by an XSL transformation from our NeuroML that is generated and we chose not to implement an XSLT for transforming from X3D to NeuroML. Files are given the *.x3d* extension.

Further plug-ins for persistence to expand the project's portability could include PyNN[12] and the Neural Network Tool within Matlab[13].

6.3 Implementation

As previously mentioned, loading and saving functionality is abstracted out as far as possible into plug-ins; all loading and saving is handled by the Save and Load Managers, which is the last point in the core system before the request is passed to the plug-in. It is the Manager's job to take a load or save specification which includes information about where to write or read the data and also which plug-in to use to perform the persistence operation. The manager then loads the required plug-in before making the appropriate method calls. The details of the specification are read by the plug-in and acted on accordingly.

Networks must be Saveable (and Serializable for Java Serialization) to be used with the *SaveManager* and save plug-ins, further still any object that has parameters or variables that require persistance to storage must have them annotated with the annotation *@Persistable*. This is so that when an object is being persisted reflection can be used to retrieve all information from the object that needs to be exported. It is also used when loading persisted information and populating information in objects upon reading a persisted network back into memory.

6.3.1 Serialization

Serialization was the simplest implementation as it makes use of Java's ability to serialize objects to file and allow them to be read back in, as the loading plug-ins are used for when a user wishes to insert a network from file into one which they are editing currently. After the network is loaded, objects must have their Ids regenerated to prevent Id conflictions.

6.3.2 NeuroML

NeuroML is implemented to conform to the NeuroML DTD, which specifies that networks and their neurons must be output followed by the synapses. This requires the NeuroML save service to buffer some information so that it can be output at the correct time. NeuroML does a breadth-first search of the network to discover all the networks and network bridges that are in the network being persisted.

Importing NeuroML is more interesting. Information is not only stored in a single tag, but multiple tags, including meta-data tags. This requires more information to be stored and lead to the *PartialObject* concept, which stores information temporarily, until the load is completed, and then will produce the concrete object with the stored parameters. More information needed to be stored because the XML parser that was used to read the XML was SAX (Simple API for XML)[14]. SAX is event driven and its output is dependent on what the parser's state; i.e. what it has read at a given time influences what you receive. The parser then acts depending on the type of tag seen; if it is an opening tag for a neurone, network or synapse then a partial object is placed on a stack. As further meta-data that is relevant to the last seen partial object is encountered, it is added to the partial object on top of the stack. On closing tags for neurone, network or synapse, the partial object creates the complete object based on the parameters that it has stored and returns it. At the end of the document the network is connected up with sub-networks and the final network can be retrieved. Objects that require methods to be called other than the standard constructor when being loaded from persistant storage may do this by creating methods that are persistable but have no actual value, so that they are called as a kind of 'pseudo-setter' when loaded; any further method calls and operations can be placed within that single public method.

6.3.3 X3D

X3D is currently dependent on the NeuroML XML persistence plug-in, as the network must first be exported to XML. The XML file is then read and an extensible style sheet (XSL file) is applied to the XML to transform it to the X3D schema. X3D persistence was implemented in this manner because it was a quick and easily maintained addition; the XSL was already available from the NeuroML project[15], and the project's existing valid NeuroML made it simple to perform. This implementation is not entirely ideal as it makes use of the Java XML libraries[16], which have problems when applying an XSL transformation to a large XML document. This is why in the Performance Analysis no data is available for save execution time and file size for X3D documents with a node count over 100. Dependencies on other plug-ins are resolved by the Plug-in Manager and are thus not an issue for the persistence Module.

6.3.4 TNS

The Text Network Serialiser is a custom record- and fact-based format, focussing on extensibility and modularity. It streams objects into a standard "header" record, followed by a collection of "facts", as decided by the particular handler or handlers for the object. It builds atop the event and plugin systems to provide a generic means for saving "Identifiable" objects, with "Persistable" fields. TNS was born out of a flaw in Java's standard object serialization, and is thus meant as a replacement for it. On some platforms, empirical evidence shows that the serializer in Java is recursive and will stack-trace on moderately large networks (approx. 1000 neurons in size). This is unacceptable behaviour for a tool designed to model large neural networks, and thus TNS was created as a replacement.

The abstract ObjectIOEventHandler performs serialisation of the header (containing an ID number and class name) and, if this is the first record for the given object, the contents of its @Persistable annotated methods and fields. After this, it dispatches the object to a processing method of the child class to write out any special details not described by @Persistable.

An object can be processed by any number of these handlers, simply outputting an extra record per handler if it has any information to record. The processing method simply fires an Event into the EventManager for each child object that needs serialising (i.e. each neuron, sub-network, and synapse).

Upon Load, these records are read in one-at-a-time, and dispatched through the read methods in the same event handlers, using ID numbers to resolve object references. If insufficient data is available about an object to perform reconstruction a handler can throw an exception which will cause that line of data to be appended to the end of the processing event queue to be dealt with later. This is used in, for example, resolving the "from" and "to" objects within a synapse.

This abstract handler system permits arbitrarily complex objects to be written and read with ease; simply extending ObjectIOEventHandler with your own processing methods for reading and writing is just a case of parsing and serialising the "special body" of your object within its own record.

Size	Serialization	NeuroML (XML)	TNS	X3D
10	0.01	0.00	0.00	0.06
50	0.19	0.06	0.00	0.48
100	0.74	0.65	0.00	4.25
500	16.97	5.43	4.27	?
1000	74.33	19.95	24.36	?

Table 1: Performance Analysis - Save Execution Time

During profiling of the initial implementation of the TNS save and load services it became apparent that the threaded nature of event processing was causing a significant slowdown in execution; about half of the execution time was spent blocking awaiting a lock on the event queue. As a result of this a second implementation, similar in nature to the first, was written. It uses exactly the same handlers, but extends them to be capable of firing events into their own internal event queue. This single-threaded implementation vastly outperforms the original version, highlighting some interesting scalability issues with switching threads in Java along the way.

6.4 Examples

For reference, in Appendix C are examples of an exported network in NeuroML, TNS and X3D; Java serialization was not included as it is not a human readable form. The example network used is one of two Excitatory Spiking Neurones with synapses linking them to each other and back on themselves (otherwise known as 100% connectivity). A view of the network in the ANNE tool can be seen in Figure 18.

It is easy to see that the TNS output is very light-weight but fairly hard to interpret by a casual eye. By contrast, both the XML formats are very long but quite intuitive to read. It is also clear that the X3D output has lost a large amount of information on the neural network, which is the main reason why no importer for X3D could be written.

Finally, there is an example of the X3D output from ANNE being loaded into an X3D viewer called Octaga[17]. This allows users to visualise their networks in a 3D space if they find that feature useful and demonstrates some of the inter-application support within ANNE.

6.5 Performance

Networks were generated with the number of excitatory spiking neurones as specified in the table and a 100% connectivity probability, networks were completely regenerated when changing between sizes.

6.5.1 Benchmarking System

- CPU – 2.33GHz Core 2 Duo
- Memory – 2GB, 667MHz DDR2
- Hard Disk – 160GB, 5,400rpm, 16MB cache
- File System – Journaled HFS+
- OS – Mac OS X 10.5.6

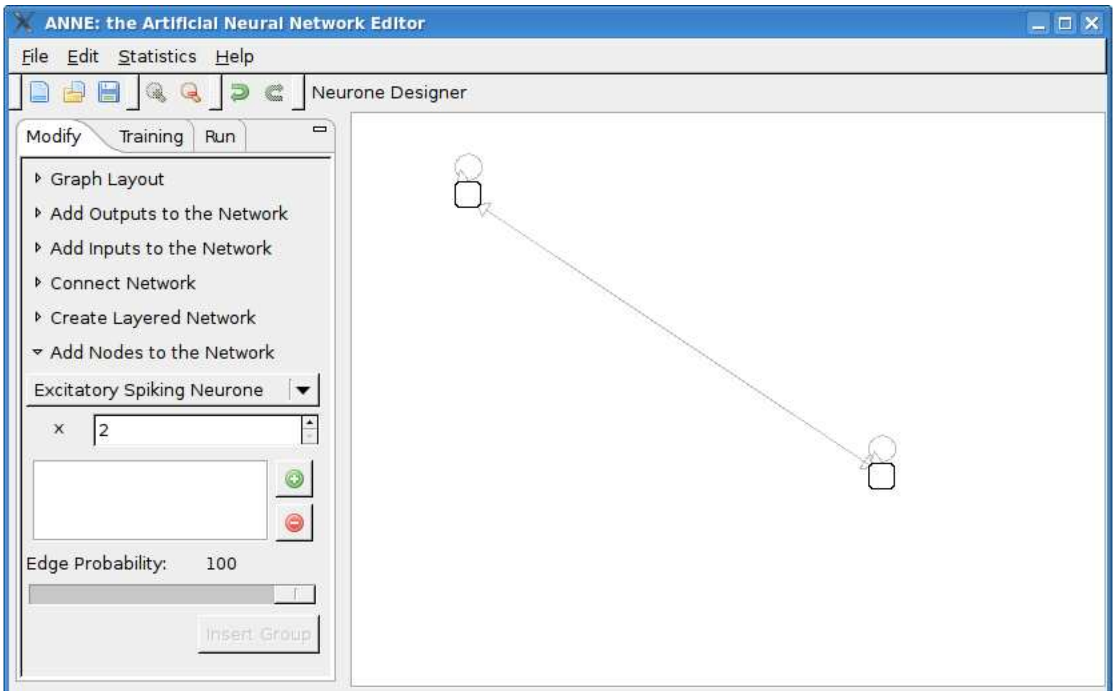


Figure 18: View of the example network in ANNE.

Size	Serialization	NeuroML (XML)	TNS
10	0.06	0.15	0.05
50	0.35	0.41	0.23
100	1.01	0.71	0.69
500	23.37	11.21	20.29
1000	94.67	57.14	2091.74

Table 2: Performance Analysis - Load Execution Time

Size	Serialization	NeuroML	TNS	X3D
10	10.93	37	13.86	21.02
50	185.93	766.38	304.57	464.97
100	716.7	2993.31	1202.03	1841.67
500	17443.26	72470.61	32823.98	?
1000	69552.63	289717.38	122396.96	?

Table 3: Performance Analysis - File Size (KB)

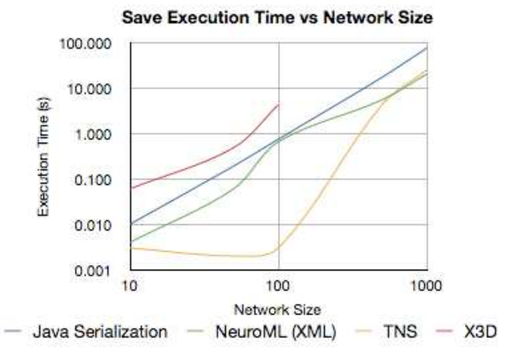


Figure 20: Performance Analysis Graphs: Saving

One clear observation from Tables 1 and 3 is that (as discussed previously) X3D failed to successfully produce any output on networks of size 500 and 1000. Also because of the implementation, nearly twice as much temporary storage is required during export, as the NeuroML XML document needs to first be produced so that it can then be read and can only be deleted after the transformation is complete. It is clear that a future release of the X3D plug-in should be implemented in a more efficient way.

NeuroML and Java Serialization grow linearly on a logarithmic graph for Save and Loading time, suggesting that their execution time would predictably grow with larger networks and would be easily estimated with network sizes within the range of network sizes sampled. NeuroML generally performs better than Java Serialization, especially with large networks, however this is offset by NeuroML having a considerably larger resulting file size than Serialization. With ever increasing network sizes this could start to cause problems – an option of compressing the output of the text based persistence methods could provide further flexibility to the user.

TNS demonstrates some interesting behaviour. Firstly, for networks below 100 neurones in size it requires significantly less time than all other persistence methods and requires no significant increase in time for sizes up to 100. After this it rises sharply but looks to begin levelling out to the same rate of increase as NeuroML and Serialization. This is potentially a result of the time taken in object allocation for the event stream. File sizes generated are significantly less than NeuroML (generally below half the size) but not as small as Serialization and the load time is competitive with that of serialization or NeuroML.

A performance metric was calculated that took into account the loading and save times, as well as the file size generated, compared

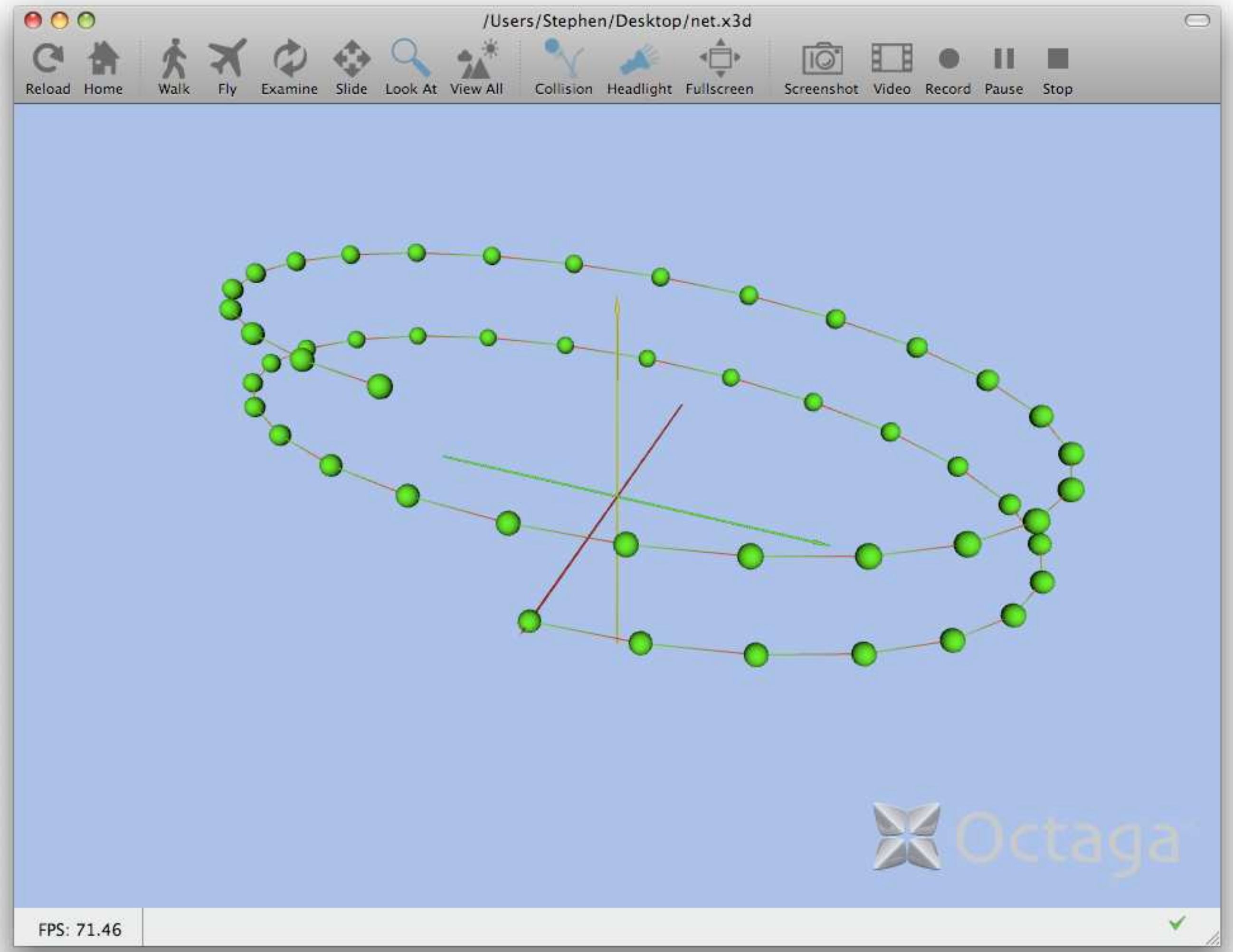


Figure 19: View of a Network produced in ANNE displayed in Octaga

to the network being persisted. The performance metric was as follows:

Performance Metric $= \frac{\text{Load Time}+\text{Save Time}}{\text{File Size}}$

A lower value of this metric implies better peformance, as we can see from the graph which plots the metric against the network size that Java Serialization is the obvious performance winner. This was to be expected as even though it has a poor saving time it produces significantly smaller files than the other persistence methods and requires no extra time for loading. It is, however, important to temper this with the un-portability and readability / editability of its format. TNS and NeuroML look to be improving after reaching an initial storage overhead on network size, TNS beating NeuroML though because of the smaller file sizes produced. X3D was included for completeness, but no real information can be taken from the graph as X3D has no load time and could not complete all the tests.

It is worth noting that this performance analysis was only a fleeting look at persistence's real world observable performance and tests were run on production systems a single time, instead of multiple times to verify results. This means little more statistical analysis can be done on these results as there isn't the required information.

6.6 Evaluation and Further Development

The persistence module provides an easy and extensible way of implementing persistence plug-ins which can export the neural network model to storage or possibly another application. The initial persistence plug-ins provide the average user with a number of options for saving and exporting, with Java Serialization the recommend method for saving a small neural network for use with other instances of the ANNE application or incremental backup. NeuroML export is available for transferring your network to other applications and importing back from it.

Due to the pluggable nature of the system, further versions of a persistence option are not required to be released at the same time as the core framework, allowing for them to be worked on and released separately on a more rolling basis. New persistence options

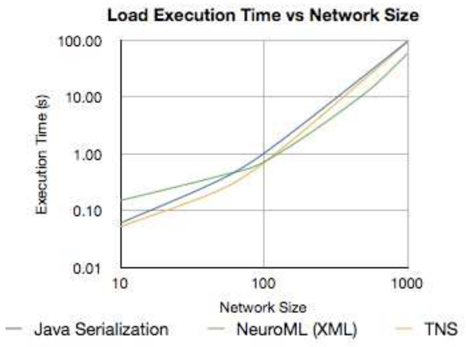


Figure 21: Performance Analysis Graphs: Loading

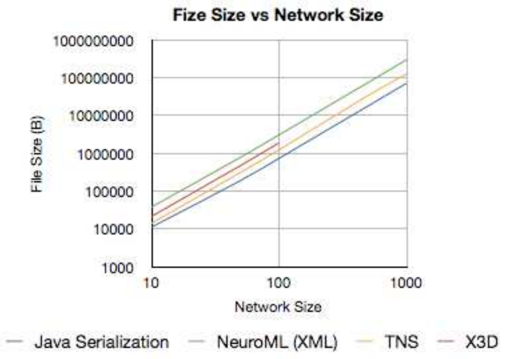


Figure 22: Performance Analysis Graphs: File Size

could be added later on if required for exporting to more specific or proprietary formats, thus allowing developers of other, potentially closed-source, applications to write their own plug-in to read their application output which could be distributed without having to reveal a specification of their file format. This ease of implementation would hopefully help ANNE become a more popular solution to model neural networks with.

Further development upon the persistence API could include:

- Further performance analysis of all the persistence plug-ins and optimizations of their code to improve execution time and memory usage.
- The option for file compression after export would reduce the file sizes produced but obviously at the cost of execution time. There may be a point where a network reaches a certain size that it may be necessary to compress the output file; already with 1000 neurones NeuroML files reach 280MB, which even with current hard drive capacities is a large file. With the option to save to a compressed format, automatic decompression of files when loading would be required so that the new compressed files where supported. Perhaps a compression option that can back-end the serialisation to another service first before compressing its output could be a persistence plug-in to enable this without requiring modifications to the existing code-base.
- ANNE's current X3D implementation is incapable of exporting large networks because of the problems using XSL transformations on large XML documents. A bespoke plug-in

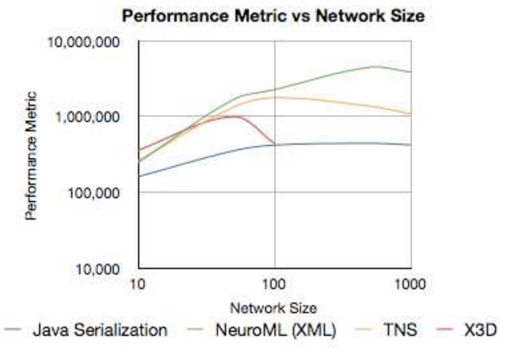


Figure 23: Performance Analysis Graphs: Performance Metric

which is not dependent on the NeuroML plug-in would probably be a better alternative implementation. Furthermore, this would reduce the amount of storage required for export as currently an XML must be produced before being read again and then deleted.

- Further persistence options such as exporting to other neural network modelling standards and file formats, for example the previously mentioned PyNN as well as Matlab's Neural Network tool.
- An auto-save feature would be a useful addition, permitting users not to have to remember to periodically save their work, in case of mistake or system failure. It could be implemented by creating another thread which sleeps for the auto-save time period and then fires the save manager off to save to a temporary location before sleeping again. ANNE could then poll this location for auto-save files in the event of a crash and load a network's state from the previous session.

7. USER INTERFACE

7.1 Overview

ANNE's user interface aims to be as intuitive as possible, yet still allow efficient execution of complex tasks. It is highly modular, and many different aspects of functionality can be added as plugins.

7.2 Design

7.2.1 GUI Framework

When looking for a GUI framework to use for our application, there were certain requirements that needed to be met. The framework had to allow us to easily draw graph objects, as well as move them around the graph by dragging and dropping. This graph also had to be expandable and easy to navigate via scrolling. Several frameworks were tested by implementing simple neural networks in code, and checking whether or not it would be simple to fulfil our requirements.

When starting to look at ways to implement the GUI in Java, we first looked at SWT[22]. SWT has several advantages; it is well documented, cross platform and has plenty of on-line support. After experimenting with SWT we found that while it was suitable for our main GUI layout and menus, it did not meet our requirements for displaying graphs. Drawing each neuron and synapse as a separate SWT canvas gave unusable result as each canvas must be rectangular, and there is no support for transparent backgrounds. This meant that if synapses overlapped, the background of one would

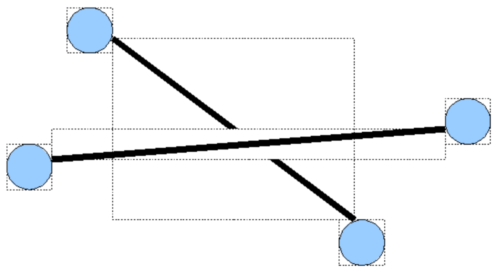


Figure 24: The Problem With Draw2d

obscure the other. The only other option using pure SWT would be to write an entire graph visualisation package that draws its results onto a single canvas, which we deemed to be too time-consuming.

Our next candidate was the Draw2D framework[23], which runs inside an SWT canvas and displays graph nodes and edges. This is a subset of the Graph Editing Framework [24], which is a heavy-weight framework for creating graphical editor software. Although GEF included graph editing functionality, it was too inflexible for our plugin-oriented architecture and multi-layered networks. Draw2D allowed us to have transparent backgrounds on graph objects, but there was still no native support for scrolling, dragging and dropping, or other common graph functions. The lack of clear documentation for the framework caused us to run into problems implementing these features, such as irremovable artefacts appearing whenever the main canvas was scrolled.

We then moved onto Zest[25]. Zest is a larger subset GEF that builds on top of Draw2D and includes the common graph functions described above. This was ideal for our program since it did everything we required, so we decided to use it despite its poor documentation.

7.2.2 GUI Layout

While designing the layout of the application, we took many things into consideration. Our main focus was making the application as easy to use as possible without limiting functionality. The design of the high-level GUI layout was discussed at length towards the start of our project, before we agreed on a single design. We have all had experience programming in Integrated Development Environments (IDEs) such as Eclipse, and felt that many features common to these programs would cross over to creating and testing large-scale artificial neural networks.

The final layout we decided on is split into panels, in a similar manner to existing IDEs. The top panel contains a menu and tool bar, the left panel contains a sidebar for all network editing operations, the bottom panel is a ticker containing important logging information, and the main panel displays a view of the neural network. Each of these panels can be grabbed and resized, as is standard for applications of this type, and their contents can easily be extended using plugins.

We decided to split the sidebar into three sections corresponding to the three ways in which people interact with neural networks: modify, train and run. The modify tab contains plugins related to editing the structure of the network such as adding groups of neurones, adding special inputs and outputs, and connecting existing

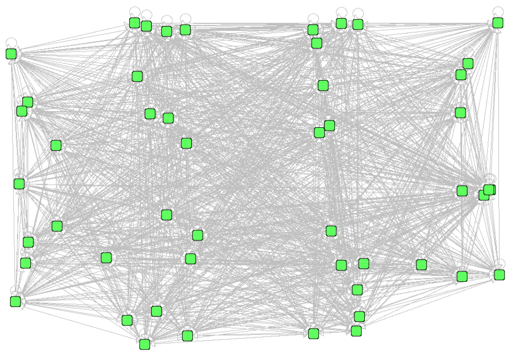


Figure 25: A group of 50 densely-connected neurones

subnetworks. The training tab lets people select a training algorithm, set parameters and train a network. Finally, the run tab lets users run simulations on a network.

7.2.3 Neural Network Visualisation

One of the main issues inherent in building a system of this type is the difficulty involved in viewing and navigating huge networks. A simple graphical solution is to display each neurone and synapse in the neural network as a single graph, which can be scrolled vertically and horizontally as needed. This is fine for networks containing a few dozen nodes and synapses, but finding specific locations in larger graphs can become time-consuming.

A proposed fix for this problem was to let users quickly 'bookmark' frequently-accessed sets of neurones, which could then be viewed using keyboard hotkeys or a menu. This technique was inspired by the interfaces of various Real-Time Strategy computer games, in which users can place units into groups and automatically snap the view to any of their locations, even when they are spread throughout a large battlefield that would be slow to navigate otherwise.

This was more useful than the first implementation, but still had its problems. Users must remember which hotkey relates to which set of neurones, and make sure to add and remove neurones from sets as necessary. Navigating to neurones that are not bookmarked is no easier than before, so a large number of groups must be created if several locations in a network must be accessed often.

Allowing the user to zoom in and out of the whole network is a far more efficient solution from a user perspective, as it enables quick navigation to any section of the graph without having to mentally maintain a list of bookmarks. However, large networks often consist of dense groups of neurones that are highly connected by a huge number of synapses. Even with zooming, the amount of visual clutter present when viewing each separate neurone and synapse makes the task of navigating densely-connected networks far from easy.

This clutter (Figure 25) was reduced by grouping sets of neurones and synapses within a network into single graph entities, then consolidating all synapses between a pair of groups into a single bridge. Our original plan was for these groups to be created and destroyed intelligently by the system as a user zoomed in and out, but the algorithm to do so would be difficult to design. The program must support many existing types of network and be extensible so that many other types can be used in the future, but these types have different notions of how groups should be organised. Implementing

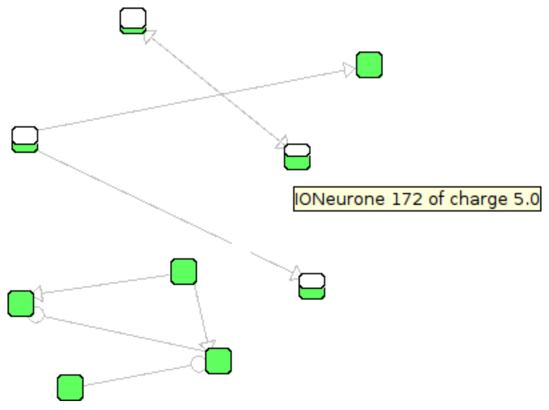


Figure 27: Circle and triangle arrowheads, charge overlays and a visible tooltip

several such algorithms for different network configurations would be time-consuming in itself, and a programmer extending the system to support a new network type would have to write another of these neurone grouping methods.

To aid ease of navigation and customisability as much as possible without compromising ease of extension, we instead let users manually create and edit these internal groups of neurones and synapses in whichever way makes the most sense for their network. These groups can be nested within each other, and navigation consists of zooming into and out of the tree-like hierarchy of groups as needed. Each of these groups in our system is itself defined as a neural network, which makes it trivial to import subnetworks into other networks and fulfil our requirement that modular networks are supported.

A few additional features were added to minimise the time it takes to obtain information from a neural network. Hovering the mouse over a subnetwork or a network bridge displays a tooltip containing a summary of its contents, and clicking the arrow by a subnetwork's name expands it to show the number of components it contains. We found this minimised the time it took to check these values, without the unacceptable visual clutter that would have resulted from displaying all of this data at all times.

Figure 26 shows a layered network containing 3 subnetworks connected by 2 network bridges. One subnetwork is expanded, and the selected network bridge's tooltip is visible.

In addition to neural network's layout, the user must also be able to efficiently view its state. Individual neurones and synapses, much like subnetworks and network bridges, contain tooltips showing the values of their parameters. Also, as is standard for diagrams of neural networks, synapses from inhibitory neurones use circle arrowheads instead of the usual triangles. We felt this gives the user as much information as possible without cluttering large networks.

7.3 Implementation

7.3.1 User Interface Core

Our Graphical User Interface builds on top of ANNE's framework and graph packages without any back-references, in such a way that the entire GUI can be replaced or removed altogether without having to change any code in those packages.

The core features that must be implemented by any user interface are encapsulated in the coreui package, and the InterfaceManager class contains the most basic of these features. Any interface manager must be able to import, edit and export neural networks, handle saving to different file locations and hold a CommandControl object. It also contains an instance of InteractionUtils, which is a class containing common neural network functionality which will be useful to most user interfaces. This includes running and pausing networks, creating sets of nodes, finding the parent networks of nodes, and bifurcating networks.

For user interfaces that support zooming, the ZoomingInterfaceManager interface is available. Different zoom levels are represented as a stack of NeuralNetwork references, with the root network at the base and the current view as the head. A stack containing the IDs of each zoom level is also available, which can be efficiently accessed by anything that needs to know the path from the root network to the current view.

7.3.2 GUI Manager

The InterfaceManager used by our GUI is the GUIManager, which draws the current view of the neural network onto an SWT canvas using the Zest framework. A graph model is imported into the view using two Transformers, which convert Node and Edge types from the model into the appropriate GUI graph elements. If a synapse to or from an external subnetwork is passed into the GUIManager, an appropriate source or sink node is created and the edge is attached.

These entities are arranged on-screen using any chosen Zest layout, with our CachingLayout working underneath to add common functionality. Source and sink nodes are automatically arranged down the left and right sides of the screen respectively, and all node locations are persisted after zooming in and out. This lets users freely zoom in and out of a multi-layered neural network without its on-screen layout constantly changing.

7.3.3 Graph Visualisation

Each of the entities that comprises a neural network must be represented in the main view panel. These components are neurones, synapses, neural networks, and the network bridges that connect these networks. Each of these objects can be dragged and dropped, and they highlight and de-highlight appropriately when selected and deselected.

Neurones are represented in the user interface as GUINode objects. These extend Zest GraphNodes and are drawn as small rounded rectangles, with a white overlay showing the neurone's current level of charge. This overlay begins at the top of the node, and grows to cover more of it as the charge decresases. If a neurone has no charge at all, this overlay covers the entire node. This gives the appearance of the node background colour being the colour of the charge, and the overlay colour being the 'background' colour.

Different types of network inputs and outputs are implemented as subclasses of InputNode and OutputNode, and are kept in the gui.graph.ionodes package. Each type of node implements the Runnable interface, and contains the code needed to initialise any graphical elements used by itself such as SWT shells. The following types of IO node are currently implemented in ANNE:

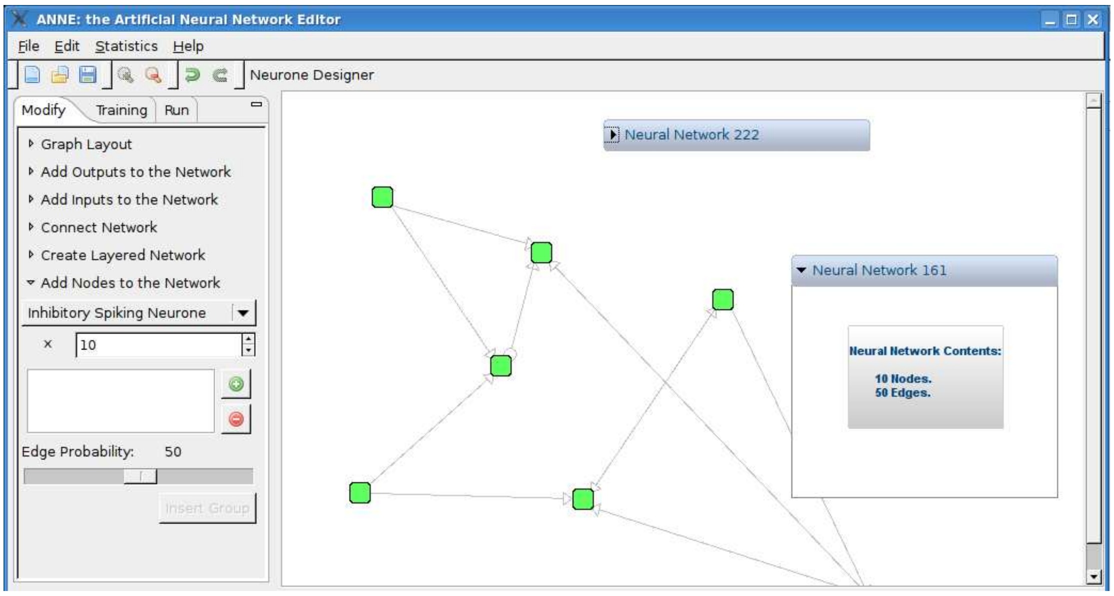


Figure 26: Layered Networks

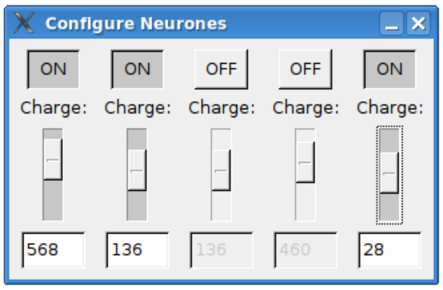


Figure 28: Punching Input Nodes

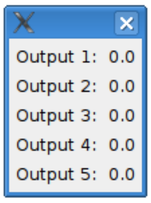


Figure 29: Value Listing Output

- DATInputNode, which takes a DAT input file and creates a subnetwork containing the appropriate neurones.
- PunchingInputNode, which creates the given number of neurones and lets the user control their charge via sliders in an SWT shell. Each neurone's charge can also be disabled and enabled by a button above its slider.
- ValueListingOutputNode, which creates a subnetwork containing the given number of neurones and displays their values in an SWT shell.

Synapses and network bridges are depicted as GUIEdge and GUIBridge objects respectively. These are both implemented as zest connectors, but GUIBridges are much thicker than GUIEdges to represent the fact that they contain several edges going from one neural network to another.

GUINetwork objects represent the neural networks layered inside other networks. These are implemented as Zest GraphContainer objects, which can be expanded to show their contents. The 'contents' shown by a GUINetwork is a single Zest graph node, labelled with a text summary of the number of nodes and edges it contains. This is more readable than showing the entire GUINetwork's contents as a small graph, and also uses far less memory.

Our GUI contains an additional type of GraphNode, which is not included in the internal model of the neural network. This type of purely decorational node is the GUIAnchor. These nodes are shown as small black squares, and they represent connections to and from external subnetworks. These nodes are divided into sources and sinks, which define incoming and outgoing connections respectively. As GUIAnchors are not stored in the internal model, they are created and destroyed after each zoom action and are laid out automatically instead of having their positions persisted.

7.3.4 Top Panel

The top panel of our user interface consists of the tool bar and the menu, both of which are implemented using SWT.

The tool bar is constructed using a CoolBar from the SWT library (Reference SWT). A CoolBar contains several ToolBars, each of which represents a group of buttons. These buttons are simply SWT ToolItems. Users can drag and resize each ToolBar, as is common for applications of this type.

Like the rest of our user interface, the tool bar is pluggable. This lets users add new ToolBars or ToolItems without having to edit the existing GUI code. The ToolItems currently included in the CoolBar are (sorted by ToolBar):

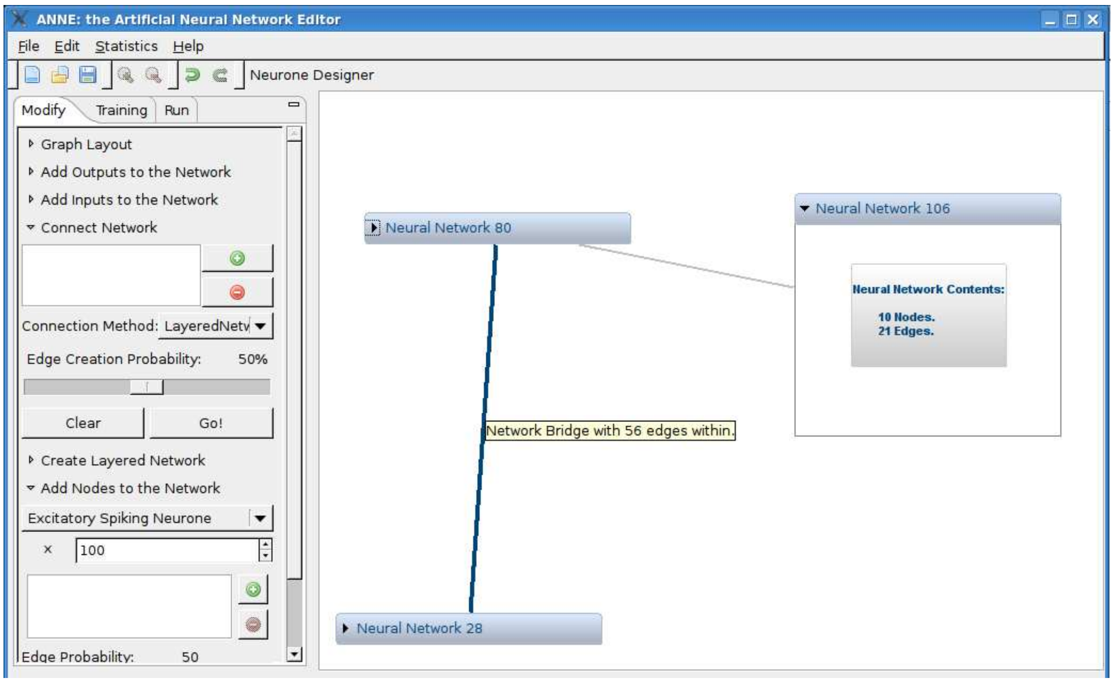

ANNE: the Artificial Neural Network Editor
File Edit Statistics Help
Neurone Designer
Modify Training Run
Graph Layout
Add Outputs to the Network
Add Inputs to the Network
Connect Network
Connection Method: LayeredNetv
Edge Creation Probability: 50%
Clear
Go!
Create Layered Network
Add Nodes to the Network
Excitatory Spiking Neurone
x 100
Edge Probability: 50
Neural Network 80
Neural Network 106
Neural Network Contents:
10 Nodes.
21 Edges.
Network Bridge with 56 edges within.
Neural Network 28

Figure 30: GUINetworks connected via GUIBridges, including a Dropdown Box

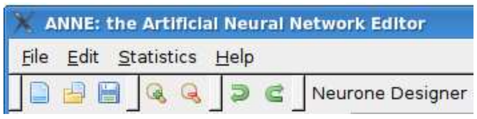

ANNE: the Artificial Neural Network Editor
File Edit Statistics Help
Neurone Designer

Figure 32: Tool bar with all buttons highlighted

ANNE: the Artificial Neural Network Editor
File Edit Statistics Help
Neurone Designer

Figure 33: Tool bar with some buttons highlighted

- New, Open and Save
- Zoom In and Zoom Out
- Undo and Redo

The individual buttons in the ToolBar can also handle incoming events, and react accordingly. The main use for this feature is to tell buttons to enable or disable themselves after appropriate events, such as the undo button checking the size of the undo stack after each CommandEvent and disabling itself if it is empty.

The icon set used for the tool bar is the Silk Icon set[26]. These were picked because they are widely used icons that clearly convey their meanings the user.

The menu uses an SWT Menu object, and is similarly pluggable. Existing plugins contain functionality for opening, loading and saving files, undoing and redoing actions, starting and stopping output plots, and more.

The sidebar is an SWT CTabFolder, and automatically loads plugins in a similar manner to the other panels. The 'train' and 'run' tabs are implemented as TrainingPanel and RunPanel, and the modify tab populates itself with any given NetworkModifier plugins.

The bottom panel is implemented as an SWT container, containing a scrolling text appender. This appender receives incoming log messages from Log4j, formats them, and scrolls the pane so the most recent messages stay onscreen.

7.3.5 Commands and Plugins

Undoable actions within the user interface are implemented to be Commands. These actions currently comprise adding and removing different types of graph items from the network, whilst ensuring that no inconsistencies occur and that the view never tries to display a network that has been removed. To ensure extensibility these commands are invoked by plugins that are implemented either as items to be added to existing menus or toolbars, or as Listeners.

Listeners are in the gui.graph.listener package, and work by listening for events such as mouse clicks or keypresses by extending MousePlugin and KeyboardPlugin respectively. Actions that are implemented as GUI plugins include:

- Selecting several graph elements by dragging a box around them with the mouse, as ElasticBandSelectionListener
- Creating synapses by selecting a node and Ctrl-clicking another node (or command-click on Mac), as EdgeBuildingListener
- Deleting all selected graph items with the 'delete' key, as MassDeletionListener
- Sending an event to inform the zoom buttons on the toolbar of changes in selection, as ZoomListener

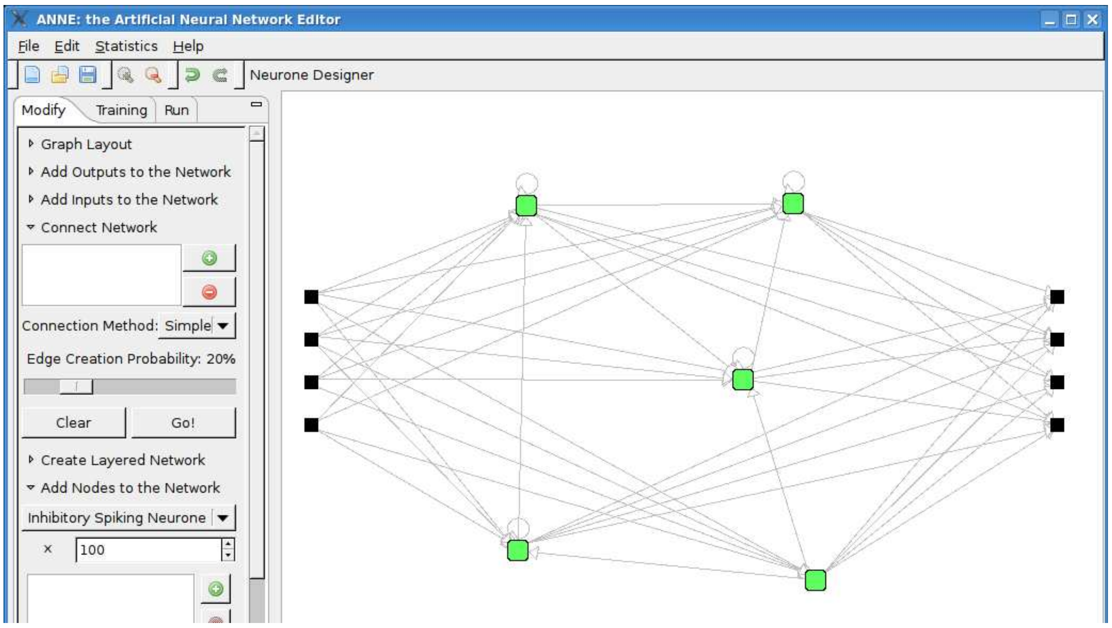

ANNE: the Artificial Neural Network Editor
File Edit Statistics Help
Neurone Designer
Modify
Training
Run
Graph Layout
Add Outputs to the Network
Add Inputs to the Network
Connect Network
Connection Method: Simple
Edge Creation Probability: 20%
Clear
Go!
Create Layered Network
Add Nodes to the Network
Inhibitory Spiking Neurone
x
100

Figure 31: GUIAnchors in a Neural Network

Different statistical methods for connecting networks are also implemented as plugins, based on the base NetworkConnector class. A list of Nodes is passed to these plugins' connect() methods, and a set of edges to be added to the neural network are returned. Our currently implemented network connectors are:

- Simple random connector, which creates edges at random between the given nodes
- Small worlds connector, which connects groups of nodes by re-routing existing edges, on Murray Shanahan's research (at the time of writing, un-published)
- Layered network connector, which creates unidirectional network bridges on a path through the selected subnetworks.

These plugins are automatically loaded into the Modify tab of the sidebar, and can be selected via a drop-down box.

7.3.6 Plots and Statisticians

The Event management system discussed in the Framework section makes it feasible to implement a generic means for exporting runtime data from the system. The primary data identified to be of interest to users is that of neurone firing patterns. As a result, three standard statisticians were implemented to output this data; one for real-time navigable raster plots, and two to output files (comma separated values, and a Matlab matrix with associated plot). All of these statisticians are built atop the *NumericalStatisticin* interface previously discussed.

To achieve this in the GUI, a statistician such as these may have a GUI configurator loaded. This presents itself as an option in the "Statistics" menu (Figure 34), to enable and disable the selected statistician. When enabled, the configurator is requested to configure its *EventHandler*, and report the classes it is to be registered for. In this way, the statistician developer can write a minimum of code.

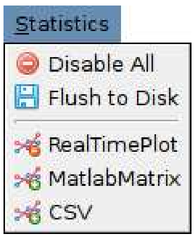


Figure 34: The Statistics Menu – RealTimePlot Enabled

For example, the CSV and Matlab Matrix statistician configurators simply prompt the user for a location to save their output file. They then write this data to disk either when the "Flush Data to Disk" option is selected, or when the statistician is disabled.

The RealTimePlot (Figure 35) displays node firing patterns as a raster plot, updating in real-time as the network runs. It also supports navigation of the plot, including zooming and x-axis panning for browsing longer network runs.

7.4 Evaluation and Further Development

The GUI provides a clean interface for creating, modifying, training and running large-scale neural networks. The layout is intuitive, and working on large networks is kept as simple and manageable as possible. Almost every aspect of the UI is pluggable, so additional requirements that come with new research can be fulfilled without editing existing code Our pluggable architecture makes it trivial to add new functionality as plugins. Listeners, which can add new keyboard or mouse functionality,are especially simple to add. Implementing the common ctrl-z (or command-z on Mac) hotkey combination for 'undo' is as simple as creating the following class, and placing it in the *plugins/KeyboardPlugin* directory:

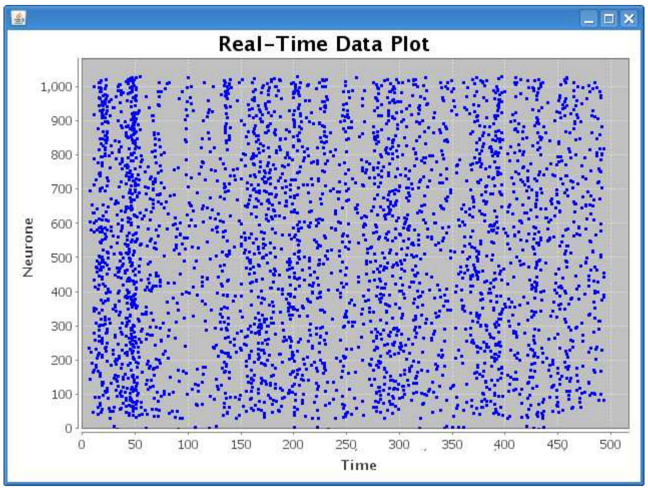


Figure 35: RealTimePlot on a spiking neurone network exhibiting gamma wave patterns

```
public class UndoHotkeyListener extends
        KeyboardPlugin {
    public void keyReleased ( KeyEvent e ) {
        if ( e.keyCode == 'Z' &&
             (e.stateMask & SWT.MOD1) != 0 )
            gm.getCommandControl().undo();
    }

    public String getName() {
        return "UndoHotkey";
    }
}
```

Core elements of the user interface could also be extended, to further increase the ease at which neural networks can be navigated. If a neural network contains extremely small subnetworks, the ability to zoom into the contents of multiple networks at once may be useful. Similarly, if a network contains large subnetworks then either being able to zoom into only part of that subnetwork or automatically dividing the subnetwork into smaller sections could bypass any performance hits resulting from viewing a large number of Zest graph items simultaneously. Support for multiple tabbed or tiled views could also be added, to let users view and edit several layers of a network at once. For other uses of the program, the current user interface could be replaced or removed entirely. A command line based interface containing only the training and running functionality could be used to efficiently test large networks, or the underlying framework and graph modelling code could be used as a subcomponent of a completely different program altogether. As mentioned earlier, the lack of back-references to the GUI package means that no existing code would have to be modified to do any of this.

8. EVALUATION

8.1 Key Requirements

The project started with an initial set of key requirements which must be met for the project to be a success and be able to complete the desired tasks for which it was being commissioned for. Here we will look at those requirements and what their current status is in the initial release of ANNE.

1. Graphical User Interface to allow easy viewing and editing of large neural networks.
2. Describe Neurone and Network connectivity at global and individual levels, include a default set of connectivity algorithms.
3. Neural Network Training and Simulation.
4. Persisting networks to storage, including nodes, edges and state, loading persisted files into the application.
5. Exporting networks to an intermediate neural network description standard, for exporting into other applications, such as Matlab or XML.
6. Manager for training data collections for specified networks.

All key requirements were fully met and except for requirement 6, a manager for training data collections. It was decided that this requirement was already available to the user in the form of their workstation's file system, which grants the user freedoms such as the choice of how to organize and store their data collections. This also allows users to use data collections from different file storage methods such as centralized storage servers and even databases.

Requirements 1 through 5 were all completely implemented and available in the initial release of ANNE. The first and second requirements are handled by the graphical user interface, the third is contained in the neural network section, and the fourth and fifth are both part of the persistence module of the application.

8.2 Further Extensions

In addition to the key requirements for the project, a number of further extensions were also proposed which would have increased the value of the application in function and usability. As with the key requirements we will recap these extensions and evaluate their current status in the initial release of ANNE.

1. Modular training algorithms.
2. Modular squash and output spiking functions.
3. Modular neural networks, the ability to insert networks into other existing networks as sub-networks.
4. Neural Network execution visualization. Firing pattern visualization in a hierarchical context. Raster plots of neurones firing.
5. Custom input and output API for loading data input and visualizing output from a neural network, the ability to feed data into training to test correctness of a given input.
6. Automatic N-Fold permutation for error analysis.

All extensions except for extension 6 are fully implemented in the initial release of ANNE. Extension 6 is partially implemented, in that automatic n-fold permutation code is in the application but has not been connected to the graphical user interface so it is not yet available to users.

Extensions 1 through 3 are all available because of the highly extensible and pluggable framework that ANNE has been designed upon. Extensions 4 and 5 are available from the GUI and neurones show to the user how much charge that they contain as well as when they fire.

8.3 Development Methodology

As a development group we practice an eXtreme Programming (XP) based methodology, XP is a form of agile software development. One of XP's main aims is to reduce the cost of change to the initial requirements of the project and allows the development to change easily and quickly with changes to the environment or specification from the client. This is a highly desirable feature for us, as we were working in a small development group directly with the clien. Changes requested by the client or resulting from implementation problems can be quickly integrated into the development cycle.

As part of XP, we practised some pair programming as well as peer code reviews. At the start of the project this slowed down progress, as some team members were new to working in pairs and initially had trouble clearly communicating ideas for complex code. However it had its advantages in the long term, such as increasing the quality of the code that is produced thanks to code being checked as it is being written and slightly afterwards. The second advantage is that it removes the dependency of the group on a single programmer understanding sections of the code base. With intricate knowledge of code know by more than one person, assistance with that section can be directed to multiple individuals, speeding up response time and also allowing development to continue with if that individual is absent.

Test driven development schemes also helped maintain code correctness and increase the speed of development, and combined with User Acceptance Testing (UAT) it provided a solid foundation to the code and gave confidence to the developers and the client about the robustness of the software.It also ensured that the final solution complied to all of the clients' specification and desired feature set.

Communication, documentation and project tracking are highly important and valued aspects of project management. To increase their effectiveness and quality as a group we made use of a number of software packagse. A Subversion (SVN)[27] repository was used for all source code and documentation control, allowing all members of the group to work concurrently on the project. This extended to multiple individuals editing the same file because of the how the repository elegantly handles multiple revisions, branches of a project, merges and conflicts. The repository also allowed developers to ensure that they had the very latest code base available to them and allowed them to separate incomplete or broken pieces of code from others.

The repository also held documentation for the project but this was also distributed across a Wiki, allowing members of the group to quickly add, contribute to and edit all available material. The repository allowed storage of formatted files which weren't suited to being placed in a wiki, and also allowed multiple editors at the same time with conflict management. An example of a file best suited to the repository is the TeX file that was used to generate this report. The implementation of wiki that we chose to make use of was Trac[28], Trac provides a number of other features other than a wiki, such as bug tracking and tickets which further increased the efficiency of inter-group communication.

Finally we made use of an online web application called Co-op[29], which provides a centralised group discussion area that members can post short messages to. This was extremely useful and heavily used by the group. It enables members to notify the group of progress, ask questions or arrange meetings, in addition to keeping a history by day of all messages posted that can be referred back to and was used as a group log book for the project.

As ANNE was designed as a framework and a set of plug-ins to add functionality, the project follows a Service Oriented Architecture (SOA). Doing so kept the project scalable and segmented into a series of small modules which each acted as a deliverable. This allowed developers to pick from the pool of incomplete modules to work on and divided the work, allowing simple metrics for time and difficulty to be used. Code maintainability was increased significantly and as has already been mentioned in this report, it is almost trivial for other developers external to the project group to extend and add functionality to the application. This keeps the working life time between revisions as long as possible and new features are not dependent on the framework's release cycle. This creates a intrinsically low coupling architecture which is simple to understand and work with.

9. CONCLUSION

9.1 Knowledge Gained

For most of us, this project was our first experience of designing and implementing such a large piece of software from scratch. This ran all the way from obtaining user requirements and designing the high-level architecture to coding and testing each individual feature. As many of these project stages are not usually taught in smaller programming exercises, we gained a lot of valuable experience in them.

Client involvement was new to us, and made a huge difference in how the project turned out. Our original idea of how ANNE was going to work was far different to how we ended up implementing it, and these improvements gradually happened over the course of several client meetings. If we didn't have so many meetings so early in the project a lot of our initial coding effort would have been wasted, so the importance of checking exact requirements as early as possible is something we all learnt.

As ANNE has to deal with very large-scale artificial neural networks, our project ran into several issues with performance. Obtaining acceptable speed and memory usage for viewing, saving and loading, and undoing and redoing changes to these large-scale networks took many careful decisions in the original design stages and plenty of optimisation later on. The techniques we learnt here, such as the performance hits when copying large arrays and the benefits of Java serialisation compared to other formats, will be of use in other future projects.

In addition to the software development techniques learnt during the project, a large amount of research was involved. To create a useful tool we had to understand the methods used in current artificial neural network research, and implementing complicated network training algorithms required understanding they worked. As much of the research we were given was coded using MATLAB, we also had to gain familiarity with the language.

9.2 Development Process Improvements

We spent a lot of time trying out different GUI frameworks. If we knew that the Zest framework was best suited to us, we could have avoided all of the redundant code we wrote to test out different framework candidates and the time spent researching them. This would have let us start the final user interface code sooner, and given us longer to fix bugs and add features.

The program's overall architecture and package organisation could also have been designed more thoroughly before starting to code. Our original architecture wasn't bad, but it was improved greatly after refactoring the entire codebase during the project. Heavier use of UML diagrams or other visual aids could have given us a better idea on how to initially design the project in a way that minimised dependencies between packages.

9.3 Possible Program Improvements

Although ANNE provides the functionality needed to run and train neural networks, the final end product isn't as polished as it could be. Zooming could be taken further to provide the user with more power when using the application. Partial zooming could be integrated to allow the user to zoom in to just a part of a very large network. Multiple zooming could allow the user to select a variety of neural networks and zoom into them, effectively draw all the neurones and synapses from all neural networks on one graph.

The way that Zest handles the layout of the nodes works well, but it can look very untidy and strange. Putting in time to create a more sophisticated layout algorithm could make the laying out of the nodes a little more professional and slick. It could arrange the nodes in a more random fashion, being more spaced out between nodes. The more nodes there are, the less space there should be between them (instead of them being all clumped together in one corner).

Several other common user interface features could be added to ANNE, to improve general efficiency. Examples of these features are copy and paste functionality, and keyboard shortcuts for all actions.

Bug-testing is something you always wish you had more time for. Spending more time on finding and fixing bugs in the program will make using the application a cleaner and more pleasant experience for the user.

9.4 Future Projects Using ANNE

We feel that ANNE can be a basis for others to build on. Due to the pluggable nature of this program it is easy to extend, so many of these additions can be implemented as plugins. More training algorithms could be added to train neural networks using methods invented in new research, and updated neurone and synapse models can similarly be added.

One feature that went through several iterations in the early stages was the issue of navigating large-scale networks. Although we ended up with a 'zoom' function that relies on the user manually creating each internal group that can be zoomed in on, many other options were considered. One such option was that the entire network was stored internally as a single group of neurones and synapses, and the internal 'groups' needed to improve navigability were created on-the-fly by the system as the user zoomed in and out. Ideally, the algorithm would try and intelligently group neurones according to the nature of the neural network, and adapt to new types of network used in future research. This feature was rejected because of time constraints, but would be an incredibly useful addition to the user interface.

10. ACKNOWLEDGEMENTS

Special thanks go to Murray Shanahan, Professor of Cognitive Robotics in the Department of Computing, Imperial College London
(m.shanahan@imperial.ac.uk) for his supervision and guidance on this project.

Also to Andreas Fidjeland (akf@doc.ic.ac.uk), Research Associate with the Cognitive Robotics Group in the Department of Computing, Imperial College London for assistance on this project.

Finally to the Computer Support Group[30] at the Department of Computing, Imperial College London for providing the repository and web hosting as well as support.

11. REFERENCES

[1] The RobotCub Consortium – http://www.robotcub.org/

[2] Neurones, An overview of – http://en.wikipedia.org/wiki/Neurone

[3] Tom Mitchell: "Machine Learning". McGraw-Hill Press, 1997

[4] Hodgkin, A., and Huxley, A: "A quantitative description of membrane current and its application to conduction and excitation in nerve". In The Journal of Physiology, 1952

[5] Izhikevich, E.M.: "Simple Model of Spiking Neurons". In IEEE Transactions on Neural Networks, 2003

[6] Izhikevich, E.M.: "Polychronization: Computation With Spikes". In Neural Computation, 2006

[7] Izhikevich, E.M.: "Spike-Timing Dynamics of Neuronal Groups". In Cerebral Cortex, 2004

[8] Structure101 by HeadwaySoftware - http://www.headwaysoftware.com/

[9] Java Object Serialization – http://java.sun.com/j2se/1.4.2/docs/api/java/io/Serializable.html

[10] NeuroML, Open Standard for modeling neural networks – http://www.neuroml.org/

[11] X3D, Open Standard for 3D modeling – http://www.web3d.org/about/overview/

[12] PyNN, simulator-independent specification of neuronal network models – http://neuralensemble.org/trac/PyNN/

[13] Neural Network Toolbox for Matlab – http://www.mathworks.com/products/neuralnet/

[14] Simple API for XML – http://www.saxproject.org/

[15] XSL transformation for NeuroML to X3D – http://www.neuroml.org/NeuroMLValidator/NeuroMLFiles/Schemata/v1.7.2/Level3/NeuroML_Level3_v1.7.2_X3D.xsl

[16] Java XML Transformer – http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/transform/Transformer.html

[17] Octaga, 3D viewing tool – http://www.octaga.com/

[18] antlr (http://www.antlr.org/) v3.1.1 – A highly flexible and efficient parser and lexer generator for the Expression system.

[19] jfreechart (http://www.jfree.org/jfreechart/) v1.0.11 – A free, open-source, Java charting library, used to render Raster Plots of Neural Network firing.

[20] log4j (http://logging.apache.org/log4j/) v1.2.15 – Log4J provides a fast, highly configurable, logging system for Java.

[21] sax (http://www.saxproject.org/) v2.0.2 – Used for parsing XML documents for loading persistence modules that use XML for persisting neural networks.

[22] swt (http://www.eclipse.org/swt/) v3.4 – The Standard Widget Toolkit provides efficient, portable access to operating user interface facilities.

[23] draw2d (http://www.eclipse.org/gef/overview.html) v3.4.1 – Draw2D provides support for drawing the primitive shapes and lines used in the Zest framework.

[24] gef (http://www.eclipse.org/gef/) v3.4.1 – The Graphical Editing Framework provides a way to create an elaborate graphical editor.

[25] zest (http://www.eclipse.org/gef/zest/) v1.0.1 – The Zest framework supports the creation and editing of graph elements, including scrolling and drag-and-drop functionality.

[26] FamFamFam Silk Icons – http://www.famfamfam.com/lab/icons/silk/

[27] Subversion Repository – http://subversion.tigris.org/

[28] Trac, Project management and bug tracking system – http://trac.edgewall.org/

[29] Co-op, Group Collaboration Software – http://coopapp.com/

[30] Computer Support Group, Department of Computing, Imperial College London – http://www.doc.ic.ac.uk/csg/

[31] StatSVNstatistical package – http://www.statsvn.org/

APPENDIX

The appendices on the following pages are outlined in brief below. Each is a self-contained section.

A. USER GUIDE

A brief guide to using ANNE to build, train, and simulate networks.

B. JAVADOC

The Java Documentation for the API packages; a useful document to distribute for future plugin developers.

C. PERSISTENCE EXAMPLES

Example output from the persistence service.

D. UML DIAGRAMS

Class diagrams for a selection of packages not already described.

E. DEVELOPMENT STATISTICS

Statistical charts of our Subversion activity produced by the StatSVN package [31].