Boone

Basic Object-Oriented Neural Environment

August Mayer, August 2003
(Boone v0.2)
Johannes Mory, August 2015
(added Spiking Neural Network section)
Helmut A. Mayer, March 2016
(started adaptation to v0.9.9)

Abstract

The goal of the Boone project ("Basic Object-Oriented Neural Environment") is the creation of a **general, basic framework** for **neural network modelling** using **object-oriented technologies**. It should be **well-structured**, flexible, manageable and reusable, have a clean external interface, and it should provide the means for evaluating **new biologically motivated concepts** in the computing world. To cooperate well with the existing NetJEN framework, **Java** is used as the programming language.

Currently, means for modeling, generation and training of **feed-forward** networks using error back-propagation and **Hopfield** recurrent networks are implemented. Some test programs demonstrate principles of the general usage of the framework; additionally, an emulation layer enables NetJEN XS to use this new package instead of the native one.

This document contains a description of the project and of the architecture, as well as guidelines for using and extending the framework.

Please report errors and omissions to `August Mayer <amayer@cosy.sbg.ac.at>`.

Contents

Part I
Overview and Analysis

1 Motivation

The Boone project (whose working title was OONET: "Object-oriented Neural Network framework") was started because of a need for a **flexible, object-oriented neural network package** for use in various in-house projects. It should replace the **NetJENXS network framework** in the long term, which is relatively close to the code produced by SNNS (Stuttgart Neural Network Simulator, a free and very powerful simulator and environment for neural networks, which also offers a facility to automatically generate C code from a neural network model); the latter is in C and doesn't use object-oriented paradigms. **Modular extensibility** is another goal of the new package, which is necessary to evaluate new, biologically-motivated network types and topologies.

The Java language was selected for **compatibility** with the existing projects, and because of its general advantages such as **portability** (enabling the user to simply switch to a faster machine if the need arises) and nice language facilities for **object oriented** software. It is generally slow on numeric computations and certain other low-level operations such as list handling, compared to e.g. C or C++, but newer just-in-time compilers, such as Sun's Hotspot JIT, reduce this advantage to a minimum. Java also seems to favor a style of breaking down complex problems into code that results in **better manageability and reusability** than other environments.

2 Requirements

There are the following requirements to this neural network framework:

- **Compatible with NetJEN XS.**
  The framework should be able to replace the native one of NetJEN XS; if the architecture is too different, an emulation layer must be provided temporarily until NetJEN XS moves to this new framework permanently. NetJEN XS is a complex software tool for creation, training and evolution of neural networks in Java.
- **Object oriented.**
  The framework should structure the task into meaningful functional units which is sensible and extensible, while maintaining a manageable level of complexity.
- **Local processing.**
  The functionality should be located in those program parts (i.e. classes) which are logically responsible. There shouldn't be an omniscient super-class which trains the entire network externally; instead, neurons should be able to train themselves, and a mixture of different training algorithms for the neurons should be possible. This is interesting especially for introducing more biological concepts, because it enables the concurrent usage of several different network and neuron types in one model, which is closer to the original nerve system structure in animals and humans.
- **Extended and new network types.**
  The nerve system and the brain should be the guides for the neural network models that are simulated with the framework. Thus, it should enable the users to experiment with various algorithms, network / neuron types and ways of combination.
- **Standard network types.**
  Even if the possibility to create advanced models should exist, it should also easily be possible to create and use standard types of neural networks such as feed-forward nets with error back-propagation, Kohonen and Hopfield networks.
- **Speed isn't king.**
  The framework should be as fast as possible, but flexibility and reusability have higher priority.
- **Analysable, Introspection.**
  It should be possible to analyse the network and, and to trace how and why the results occur.

3 Method, Timeline

Spring/Summer 2002 Planning, Search for material.

September 2002 Project start; Analysis of the preceding frameworks efforts:
NetJENXS Neural Network Framework [NetJENXS 02],
OO-ANN framework project;
Creation of the layout.

November 2002 Basic framework, i.e. neurons, links etc.

December 2002 Registry, training classes; extension of the basic framework.

January 2003 I/O system.

February 2003 Emulation layer for NetJEN XS.

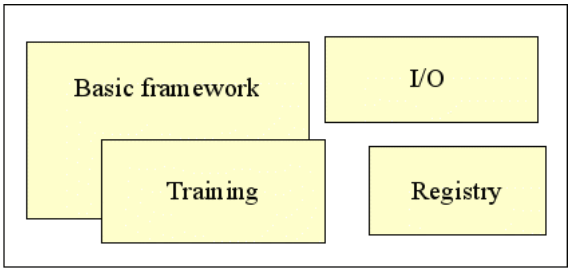
March 2003 Back-propagation, Hopfield networks.

April, May 2003 Interface clean-up, testing and finishing the project (Release 0.1.1)

June 2003 Some more documentation, Release 0.1.2.

August 2003 Extensions and corrections, Release 0.1.3.

Part II
Framework Overview and Design



1 The Basic Framework

The basic framework consists of classes for **neurons, links, neural networks, activation functions, network factories**, and some **helper classes**. It is located in the boone package of the source tree.

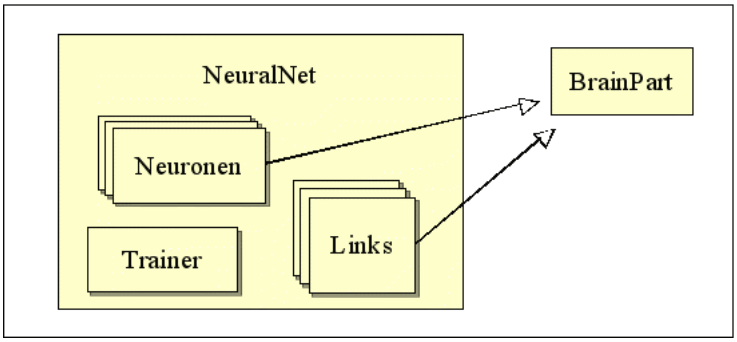
A goal in designing the basic architecture was to create a good object-oriented structure, and specifically to **localize** network operation. There isn't a super-object that performs all the calculations, perhaps using the other classes just as data holders; instead, every part performs its calculations mostly by itself.

Another aspect is that Boone consists of **basic classes** on the one hand, such as neurons, links etc., that are like the building blocks of the architecture. On the other hand, there are also some **basic tools and applications** in the package, such as the feed forward network factory, the backpropagation trainers and other things. It is possible, though probably not very useful except for cases where complete control is needed, to just use the building blocks to construct and evaluate neural networks. Moreover, if Boone doesn't provide a specific ability, such as Kohonen networks, it is possible to implement it outside of the Boone package, but using the framework and its various provisions. This flexibility is needed when experimenting

with unconventional or novel network types, and the feasibility of this is one of the design goals of the project.

This chapter presents an overview and some general directions; for detailed object and method descriptions please see the **generated Javadoc API documentation**.

1.1 `Neurons` and `Links`, and the `NeuralNet`



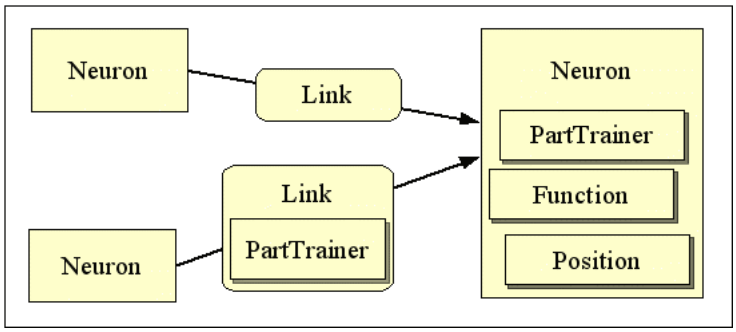
Neurons (class `Neuron`) and **links** (class `Link`) share a common base class called BrainPart, which contains common elements such as ID, name and part trainer, which are described below. There are two flags in the standard `Neuron` class signaling whether it is an input or an output neuron, so neurons can be either, both, or just plain intemediate neurons.

The links between the neurons are represented by full-grown Link objects. This incurs a certain overhead in handling and management compared to approaches using a connection matrix or the like, but it provides for more **flexibility** to replace those classes when modeling more complex network types, such as when using links that connect more than two neurons.

Moreover, **training is separated** from the network evaluation by using extra trainer objects. Thus, the same neuron and link classes can be used in both feed-forward and Hopfield networks, by using an approach that works for both network types. It should only be necessary for more special networks to override those classes, though it is certainly possible and provided for. Furthermore, the trainer classes are divided into network `Trainer`s and so-called `PartTrainers`, whereby the `Trainer` controls the overall method of training, and the `PartTrainer`s train the specific neurons and links. Thus, there is one Trainer instance per network, and one PartTrainer for each `BrainPart` (`Neurons` and `Links`).

There is a **container class** called `NeuralNet` which contains a list of all neurons and links, references to the network trainer and tick list, and methods for setting up and evaluating the network. This class has only minimal functionality beside maintaining the lists, and delegates to the neurons, links and trainers themselves instead. In other words, neurons and links are added to and removed from this container class. The `NeuralNet` also keeps track of the input and output neurons, and provides easy methods to set the network input for the network evaluation, and extract the output after it is done.

1.2 Helper classes



The neuron and link classes use some **helper classes**. The `Position` class contains a 3D position for the neurons. The `TickList` class contains a list of objects implementing the `Tickable` interface (which currently only `Neuron`s do), and beside standard list methods, `TickList`s provide mainly for a `postpone()` method that removes the current element from the list and appends it at the tail, which is used when sorting the neurons into calculation order (i.e. creating the topology cache, in NetJEN diction). The goal of this is that all of the neuron's input values are calculated when a neuron is evaluated.

`Function` is an abstraction of a **mathematical function** and maps an input value to an output value, and it also calculates first- and second-degree derivative function values. There are a number of predefined functions, defined as static inner classes of the Function class, which are `Identity`, `Sigmoid`, `TanHyp`, `Exponential`, `Sinus`, `Signum` (returning 0 if the input is 0), `SignumWithoutZero` (returning -1 in this case), `AtLeast`, `AtMost`, `GreaterThan`, `LessThan`, `Clip` and `AboutEqual`. The `Composite` function encapsulates another function, and multiplies and adds factors to the input value before calling this enclosed function. Not all of these functions have continuous derivatives, and thus, not all of the classes implement the respective derivative functions.

1.3 Network Generation

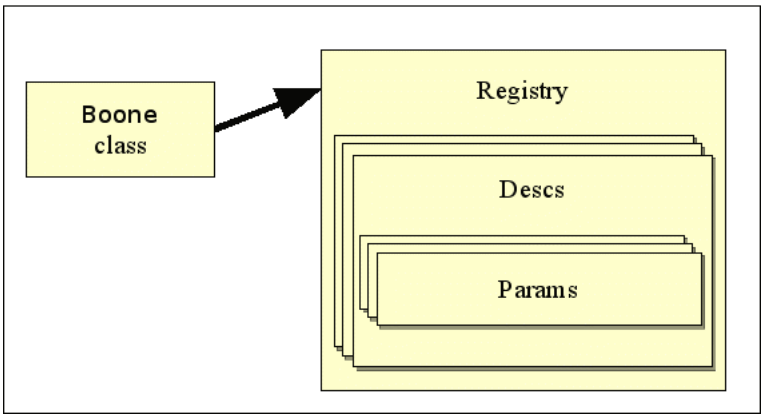
To **generate new neural networks**, the hard but flexible way is to create a `NeuralNet` object and add neurons and links one by one. The easy way is by using the `NetFactory` classes. Those take some properties, such as the number of neurons in the various layers, and produce a ready-made neural network.

For example, the `NetFactory.FeedForward` class produces feed-forward networks consisting of layers with given numbers of neurons, and the `NetFactory.Hopfield` class produces a fully-connected Hopfield network. The `NetFactory` class itself is abstract.

The `SimpleNetFactory` class is even easier to use; it provides just two `create` methods for feed forward and for recurrent networks. The price is some inflexibility; it is however possible to customize the generated network afterwards.

The names for the trainer and the activation function are the internal Java class names (inner classes are separated with "$" instead of "."). For a listing of those names, please see Appendix 1.

2 Introspection – The Registry System



The registry system provides meta-information like name, class and parameter descriptions about the classes in the framework, so that objects can be dynamically created, parameters values queried and set, and methods be called. This is especially needed during serialization (loading and saving) and for GUIs that need to enumerate and describe the available elements.

The rationale behind the design is that there is a central `Registry` instance contained in the `boone.Boone` class, where descriptors for all the different classes that need to **provide dynamic information** are stored. Thus, a program can **evaluate at runtime** which classes of a given base type or a given kind are **available**. It is even possible to add new classes at runtime, creating a plugin infrastructure. Also, new kinds of classes can be added later without a need to alter the registry framework itself; this was a weak point in the design of the NetJEN XS framework.

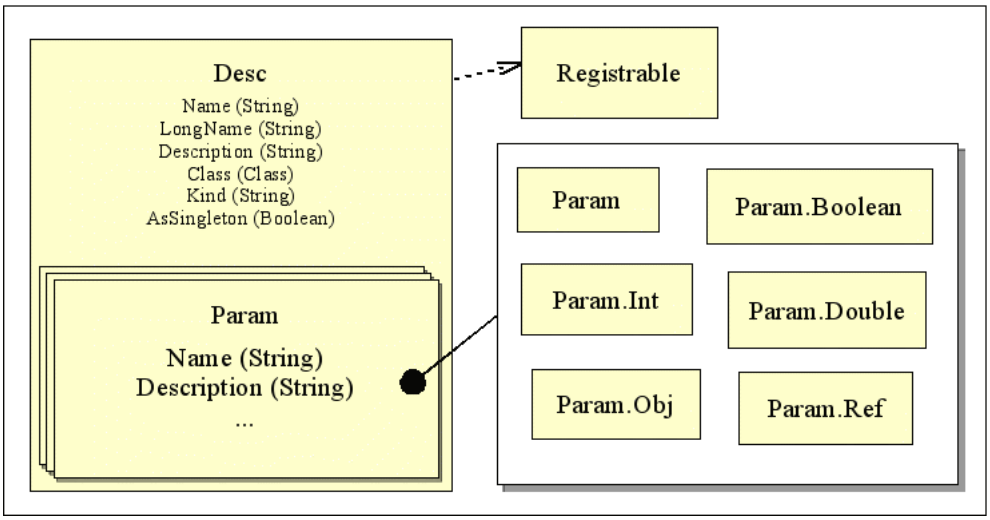
The registry framework is also used extensively in the I/O system and provides special features for (de-)serialization, such as `load()` and `store()` methods in `Param`, and overridden `setField()` and `getField()` methods in `Param.Int`, `Param.Double` and `Param.Boolean`. For a more in-depth description of I/O, please see below in chapter 2.4, "I/O – Loading and Saving Networks and Patterns.".

2.1 The Registry Class, and the Registrable Interface

The classes for the registry system are located in the package `boone.registry`. The `Registry` class basically holds a list of objects implementing the `Registrable` interface, which usually are objects of class `Desc`, in several `Hashtable`s for speed. New items can be registered and deregistered, and items can be designed as default items for their kind, which can be retrieved later. A `Registrable` can be retrieved using either a name or a `Class` object as a key; this is useful for getting the description for an arbitrary object.

A `Registrable` only needs to implement two methods, `getName()` and `getObjClass()`. Usually, objects of the `Desc` class are used, which implements `Registrable` and already has some predefined functionality and attributes that are sufficient for ordinary use.

2.2 The `Desc` and `Param` Classes



The `Desc` class implements the `Registrable` interface, as explained above; thus it can be stored in `Registry` objects. Moreover, it **contains fields for name, description and the `Class` object** for the class to register, and a **list of `Param` objects** containing descriptions of the parameters (i.e. fields) of the described class. It also contains a `newInstance()` method to create new object instances dynamically.

A flag denotes whether the class should be a **singleton** class, i.e. whether it only should be created once. This is used when the class is wholly stateless, creates no side effects, and it doesn't matter which instance is returned. The flag is especially respected by the `newInstance()` method.

The `Param` class contains fields for the properties `name` and `description`, and methods for modifying the field in an object instance, called `getField()` and `setField()`. The implementation tries to use getter and setter methods via reflection, and falls back to modifying public instance variables if necessary.

There are five derived classes called `Param.Double`, `Param.Int`, `Param.Boolean`, `Param.Obj` and `Param.Ref`, which store additional things for some field types, such as upper and lower bounds of valid content or a default value, and which handle objects and references (see below). Thus, for example, GUIs can find out at runtime which values are valid in a given object field. Generally, string parameters will be described with basic `Param` objects, and `double`, `int`, `boolean` and object values using one of those derived classes.

2.3 The `Param.Obj` and `Param.Ref` Classes

There are two `Param` subclasses that deal with objects that are contained in other objects. The `Param.Obj` class describes sub-objects, which in turn need to have meta-information available again to be correctly serialized. The `Param.Ref` class defines a reference to an object that is primarily contained elsewhere, and where only a reference should be stored.

Note that there is no `Param` class that describes a list of objects. This is intentional, because those lists often need to be initialized in a specific order. For example, for Neural networks, all the neurons need to be initialized before the links can be created correctly.

2.4 Registering new Classes

The programmer using the framework can store descriptions for new classes in the central Boone repository (class `boone.Boone`) too, just as it is done for the standard classes. For this, he basically needs to create an instance of the `Desc` class, describing the new class.

If the class is a new kind of class (i.e. isn't derived of an existing kind such as functions, trainers etc.), we should define a String constant describing the kind of the class; this can be used to e.g. get a default class for the kind of classes.

For example, to register a new class called `OutIdentity` (as used in the NetJENXS compatibility class `WrapFunction`), one could use the following code:

```
/**
 *  special identity function for Output Functions.
 *  This is necessary because we need to be able to distinguish between the
 *  function types when converting back.....
 */
public static class OutIdentity extends Function.Identity {}

/** Desc object for the {@link OutIdentity} class. */
public static Desc outIdentityDesc = new Desc(
    "Identity Output function",
    "Identity function for output / compatibility function",
    OutIdentity.class,
    Standard.FUNCTION_KIND,
    true); // singleton

// static initializer: register our special classes
static
{
    Boone.instance();  // initialize the Boone class, if necessary
    Boone.getRegistry().register(outIdentityDesc);
}
```

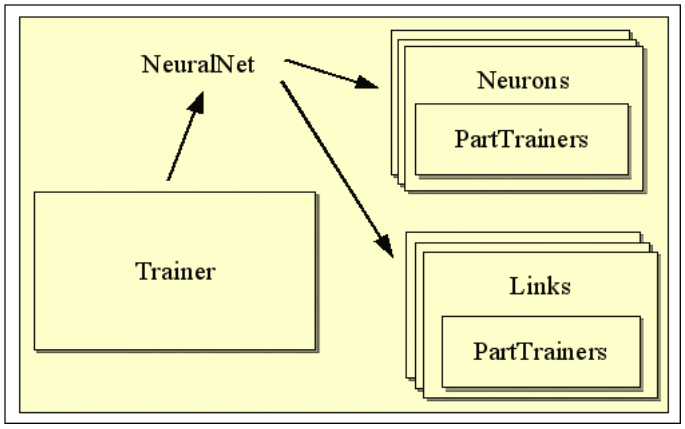
Please note that, when using this kind of static initializers, the class has to have been used somewhen before the I/O framework is used. Otherwise, the new descriptors aren't registered because the static initializer hasn't been called.

3 Training and Evaluating the Net

The base classes for training are located in the central `boone` package; specific training classes, such as error back-propagation, are assembled in the `boone.training` sub-package.

3.1 Base Classes for Training

There are two base classes: the `Trainer` **class is an overall trainer class** for the whole neural network, and implements the general scheme of training. The `PartTrainer` **class contains per-element training** data and methods, and the network `Trainer` class calls the `PartTrainer`'s method for element-specific operations.



The `Trainer` class provides schemes for supervised as well as unsupervised training, and also for testing and evaluating the network. The inner interfaces `EpochBased` and `TurnBased` provide means to have methods called at each training epoch (i.e. after presenting all training patterns once) or turn (this is after each pattern presentation), and such methods can be called for both external listeners, and internally for derived training classes.

There are usually separate `PartTrainer` classes for `Neuron`s and `Link`s. Now, it would be possible to write one big `train()` method in the `Trainer` class, and to use the `PartTrainer` classes to just store extra values (such as errors, momentum etc.) for training the respective element. However, a basic concept of the framework is to localize training as much as possible; potentially, it should be possible to attach `PartTrainer`s of different types to each `Neuron` and `Link`, and then train the neural network with different algorithms at the element level. This is also useful with advanced `Neuron` or `Link` classes, e.g. links connecting more than two neurons, or when other factors such as neuro-modulators influence the result.

Trainers, network as well as part trainers, have meta-descriptions (`Desc` objects) associated with them as described above, so that they can be loaded and stored. Network trainers also implement the `Referentiable` interface, so they can be referenced by other classes during persistence. The `createPartTrainer()`-function creates default part trainers for given `BrainPart`s, mainly for use during the construction of new networks.

3.2 Specific Trainer Implementations

Two network topologies are implemented: **feed-forward networks** and **Hopfield recurrent networks**. For feed-forward networks, the `BackpropTrainer` class in the `boone.training` package provides a standard error-backpropagation implementation, while the `BackpropMomentumTrainer` and the `RpropTrainer` classes provide back-propagation with momentum and resilient back-propagation algorithms. The `HebbTrainer` class is a simple hebbian learning trainer. `PartTrainer`s for each of those algorithms are defined as inner classes; for example, the `RpropTrainer.LinkTrainer` defines a `PartTrainer` for `Link`s for use with the resilient back-propagation algorithm.

For Hopfield networks, the `HopfieldDeltaTrainer` class implements the training of Hopfield networks using the delta (perceptron) learning rule. There only is a `PartTrainer` class for neurons here; none is needed for the links here.

3.3 The Feed-Forward Trainers

As described above, there are four predefined trainers for feed-forward networks, three using flavors of the backpropagation algorithm, and one with a simple Hebbian weight modification rule.

The simple back-propagation trainer class, `BackpropTrainer`, performs learning using a standard back-propagation algorithm, with an optional momentum term (turned off by default). Each training turn, it first evaluates the network, then calculates the network error using the error function (initialized with a default `TrainingSignalGenerator.SquareSumError` object). Then, the neuron's part trainers are called, in the inverse order of the tick list. The error that is returned is the one calculated by the error function. The neuron `PartTrainers` first calculates and adds the bias change, then calls the `train()` method of the preceding links' part trainers. This method first propagates the link's error to the source neuron, and then modifies the weight accordingly. The learning factor that is used for both link and bias modification is stored by the overall `BackpropTrainer` object, as well as the momentum rate.

Another modification of the backpropagation algorithm called Resilient Backpropagation is implemented by the `RpropTrainer` class (see e.g. [Riedmiller 94]), which is a more sophisticated algorithm that usually learns quite fast. Here, only a sum of the back-propagated error gradients is accumulated every training turn and for each neuron bias and link weight. At the end of the epoch (i.e. after training all the patterns in the training set), this error gradient sum is used to decide whether to add or to subtract a weight update value (delta). Also, this delta is modified for the next epoch; if the error gradient has kept the same sign as during the previous epoch, the delta is increased using a fixed factor (the `rateIncFactor` property in the trainer class), because we then assume that we are going in the correct direction of the minimum; if the error gradient changes the sign, the delta is decreased by multiplying with

the `rateDecFactor` value, because we have overstepped the minimum.

Finally, the `HebbTrainer` is a trainer implementation that is simple and probably not very useful. Each turn, it modifies the link weights by adding the product of the output values of the sink and source neuron and a learn rate (by default 0.3).

3.4 The Hopfield Trainer

The `HopfieldDeltaTrainer` class implements a trainer for Hopfield network, using an iterative approach. This trainer first evaluates the network, and then calls the neuron trainers' train() method, which modifies the active links (i.e. the links that did contribute to the neuron value, that is, which had a non-zero weight) with a delta value calculated from the learning factor and the neuron's error divided equally among those active links. Please also note that this is a recurrent network; thus, for network evaluation, it usually needs multiple evaluation cycles per pattern. By default, the network factories set this maximum cycle number to 100.

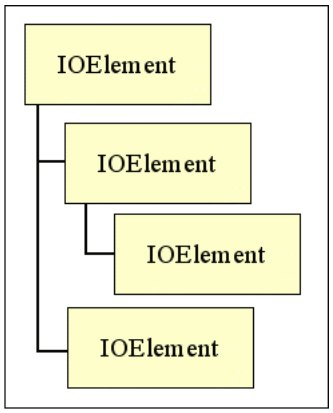
4 I/O - Loading and Saving Networks and Patterns

The I/O framework provides for the infrastructure to load and save networks and pattern data. Its classes are in the `boone.io` package, but provisions for loading and saving are distributed throughout the framework.

The basic idea behind the architecture is a two-step procedure: first, the network is serialized to a tree of `IOElement` objects, and then this tree is stored to a stream, either in "raw" format as a custom XML file, or converted to some other format (this isn't implemented yet; interesting targets are the SNNS or the Joone file format). Conversely, when loading from a stream, a tree of `IOElements` is created from the input first, which is then examined by the framework to construct the network.

Note that there also file handlers for training and test pattern data; those generally don't use the `IOElement` approach. For concrete examples of how to use the I/O classes, please see part 3 below.

4.1 The `IOElement` Class, and the Framework-internal Provisions



The `IOElement` class is comparable to the `org.w3c.dom.Element` (i.e. a DOM Element); it can store a key, a value, some attributes defined by a name and a value, and some sub-elements. Thus, a rich **tree of IOElements** can be created.

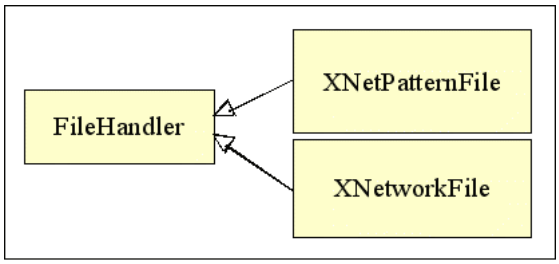
For saving, this is done in the `Boone.storeObject` method by first examining the meta-information for the fields, storing the parameters of the class, and then by calling the `store`

method of the class, if such a method exists. First the `NeuralNet` object is saved, and the algorithm recurses down to storing the `Neurons`, `Links`, `PartTrainers`, `Functions` and everything else.

Conversely, the `Boone.loadObject` method first creates the respective object, then initializes the parameters from the `IOElement` tree and finally calls the `load` method, if there is one. In order for this to work, it is necessary that all objects that are saved and loaded have their meta-description available in the Boone registry, so that the objects can be constructed correctly.

Note that, in the `IOElements`, not the real Java class name is used to identify the objects, but the `name` given in the `Desc` object for the class. This name is shorter, systematic and better to use; for a listing of the existing names, please see Appendix 1.

4.2 `FileHandler`, `XNetPatternFile` and `XNetworkFile`



All file handlers of the framework are derived from the `FileHandler` class, which defines basic methods for reading, writing and finding out whether reading and writing is possible for a given `File`. `FileHandlers` are also registered in the `Boone` registry. FileHandlers don't directly operate on files, but write to and read from streams. The idea is that usually, it is very specific to the application what should happen if the file can't be opened or used.

`XNetPatternFile` can load and store `PatternSet` objects, which hold multiple double arrays with input and output values, to a GZip-compressed XML stream. This is for loading and storing test and learning patterns.

The class `boone.io.snns.SNNSPatternFile` is a file handler for SNNS pattern files. They operate on `SNNSPatternSet` objects that also store the extended attributes of this file format (more concretely, the dimension information for the patterns).

`XNetworkFile` stores a network by converting it to an `IOElement` tree as described above, and then translating it to an XML DOM tree and writing it to a GZip-compressed XML stream, or reads such a stream and creates first an `IOElement` tree and then a `NeuralNet` object out of it.

5 The NetJEN Compatibility Framework

The classes of the NetJEN compatibility framework replace some of the respective classes of the former framework in `edu.cosy.netJEN.framework` and `edu.cosy.netJEN.framework.net`.There are several other, small changes throughout the whole package, because the network classes weren't properly isolated and internals were used in some places. Specifically, the EvoNet class is now the only interface to the network, and access to the `NeuralNet` class is now forbidden.

5.1 Changes in `edu.cosy.netJEN.framework` and other Subpackages

The main changes occurred in the `EvoNet` class which mostly encapsulates network operation, training and evolution; large parts of it were redone for the Boone framework. Some changes were also done in the `Function` class to delegate to the Boone function types where necessary,

and in the other classes of the original framework.

There are also some smaller changes in the subpackages `io` and `evolution`, necessary because of the switch from the `NeuralNet` to the `EvoNet` wrapper class. The `evolution.Individual` class, for example, was a bit critical, because it uses a template `NeuralNet` and evolves `clone()`'d copies of it, storing the evolution result in the original `EvoNet` object. It also adds its own `Delegates` (= Listeners) to both the `EvoNet` and `NeuralNet` objects.

5.2 Emulation Classes in `edu.cosy.netJEN.framework.net`

All the original classes in `edu.cosy.netJEN.framework.net` were thrown away, because the Boone framework fully replaces them. To provide for compatibility for classes that use characteristics of the old package, some emulation classes were created. For example, the `EvoNet` object returns classes of type `Unit` (equivalent to the `boone.Neuron` class, and encapsulating an instance of that class) to its clients. Also, a number of numerical constants are defined here (e.g. in `Unit` for selecting input/output/hidden neurons). There is also some functionality that is realized in some other way; `Result`s and `ResultSet`s as well as `Pattern`s and `PatternSet`s are replaced by `double` arrays in the Boone framework, which has less functionality, but is sufficient for most uses.

The classes that 'encapsulate' the functionality of the Boone framework are `Unit`, `Link`, `ActFunction`, `OutFunction` and `TrainingFunction`. Functionality that is replaced by using `double` arrays in the Boone framework is emulated by `Pattern`, `PatternSet`, `Result` and `ResultSet`.

There aren't `InitFunctions` like `RandomizeWeights` or `SetWeightsToValue` anymore in the Boone framework, those were removed without replacement. To reset the network to a fresh state, this must be done by hand, but it's as simple as enumerating the links and neurons and setting the weights and biases to some random or constant value. This is done by the emulation classes for NetJEN.

The `WrapFunctions` class contains wrappers to convert between the new function types and the NetJEN activation, output and training functions. For each NetJEN function, a wrapper class is defined as an inner class of `WrapFunctions`, and a `registerFunctions()` method is provided to register those wrappers. A `wrap()` method takes a Boone function or trainer and wraps it for use in the compatibility framework. The framework takes care that the result of this wrapping is always of the same class, i.e. that the `Identity` function, for example, is correctly mapped to `OFIdentity` or `AFLinear` (in this case by actually using derivative classes called `OutIdentity` and `ActIdentity`).

Part III
Using the Boone Framework

This chapter provides some examples for using the Boone framework, and (hopefully) some more insight into the general usage. Three examples are discussed, first a network to solve the XOR problem, then examples of how to save and load patterns and networks, and finally an example of using Hopfield networks for pattern recognition.

1 Solving the XOR Problem

The XOR problem is quite famous, because it can't be solved using only a single perceptron. Furthermore, the example nicely shows the capabilities of the framework model (please see e.g. [Rojas 96], p. 62f., or [Bishop 02], p. 86ff.). There are two binary inputs and one output, and the network in between needs to have more than one neuron. The network should learn that the input (1, 0) and (0, 1) both produce 1 as output, and (1, 1) and (0, 0) both 0. The demo program is fully available under the class name `test.XOrTest`.

To create a suitable feed forward network, we use the following call:

```
NeuralNet net = SimpleNetFactory.createFeedForward(
   new int[] {2, 2, 1},    // 2 input, 2 hidden and 1 output neuron
   false,                  // not fully connected
   null,                   // no special activation function
   "Trainer.Rprop");       // Rprop trainer
```

Then, we need to train the network, which can be achieved as follows:

```
double[][] inPatterns  = new double[][] {
                              { 0, 0 }, { 0, 1 }, { 1, 0 }, { 1, 1 } };
double[][] outPatterns = new double[][] {
                              { 0 },    { 1 },    { 1 },    { 0 } };

System.out.println("*** Training...");
Trainer trainer = net.getTrainer();
for(int i=0; i<100; i++)
    trainer.train(inPatterns, outPatterns, 10, null);
```

This first defines the input and output pattern arrays (i.e. `inPatterns[i]` should produce `outPatterns[i]`). Then the `train()` method is called 100 times; it presents each pattern to the network and performs training based on the results. The `10` is the number of times each pattern should be presented to the network (this is a completely arbitrary number here), and in place of the `null` parameter, an object implementing `Trainer.TurnBased` or `Trainer.EpochBased` (or both) could be specified whose methods would be called before and after training turns or epochs.

Finally, the following code performs a test on the trained network:

```
System.out.println("\n*** Testing the network...");
double[] errs = net.getTrainer().test(inPatterns, outPatterns);
System.out.println();
for(int i=0; i<errs.length; i++)
    System.out.println("Error " + i + " = " + errs[i]);
```

This calls the `test()` method of the trainer, which returns a `double` array containing the error values for all the given patterns, i.e. the difference between the network results from the `inPatterns` and the given `outPatterns`, as a sum of squares. Those error values are printed out.

2 Saving and Loading

Saving and loading patterns and networks is quite straightfoward. There are two example programs to demonstrate how this works; they are called `test.NetIOTest` for networks and `test.PatternIOTest` for patterns.

A network can be stored as follows (not showing exception handling):

```
XNetworkFile netFile = new XNetworkFile();
OutputStream fOut = new FileOutputStream("test/test.xnet");
netFile.write(fOut, net);
fOut.close();
```

This simply creates a `XNetworkFile` instance, opens an output stream, writes the network to the stream using the `XNetworkFile` instance, and closes the stream. Loading a network works mostly the same, as demonstrated here (also without exception handling).

```
XNetworkFile netFile = new XNetworkFile();
InputStream ins = new FileInputStream("test/test.xnet");
NeuralNet net = (NeuralNet) netFile.read(ins);
ins.close();
```

Moreover, the `canRead()` and `canWrite()` functions of the XNetworkFile class can be used in Swing file selectors to choose valid files for loading and saving.

Loading and saving patterns can be done exactly the same way, but using `XNetPatternFile.PatternSets` with the `double` arrays to store instead, and using the class `XNetPatternFile` to store to the stream.

3 Hopfield Networks

Hopfield networks are used the same way as other networks; a complete example is supplied with the class `test.HopfieldTest`. To create a complete network, the line

```
NeuralNet net = SimpleNetFactory.createRecurrent(9, null, null);
```

is sufficient. The `9` is the number of neurons that the network should contain. By default, the `HopfieldDeltaTrainer` is used, and a step function (`AtLeast(0)`) as the activation function.

Hopfield networks are by standard fully connected, and all neurons are input and output neurons as well. Neurons can only be either active or inactive, i.e. the output value of each neuron can only be 1 or -1.

To train the network, this can be used:

```
println("Training.");
int i;
int maxCycles = 100;
double err = -1;
for(i=0; i<maxCycles && err != 0; i++)
    err = net.getTrainer().train(
                  trainingPatterns, trainingPatterns, 1, null);

System.out.println("Used " + i + " training cycles (of "
                  + maxCycles + " max)");
```

It is sufficient to train until the error is 0, because then, no more weight changes will occur internally.

Finally, to send a pattern through the network, this code can be used:

```
println("Testing " + name + ".");
net.setInput(pattern);
net.innervate();
double[] result = net.getOutput(null);
```

If learning terminated successfully, the result patterns for the training patterns stay exactly the same after running the net; for patterns that are similar to training patterns, the closest one should be selected if the training went well.

Part IV
Spiking Neural Networks

Version 0.9.5 extends the Boone framework to include support for the simulation of spiking neural networks (SNN). The algorithm for the simulation is event-driven, i.e. the neurons are updated only when they receive or emit a spike.

The *Spike Response Model* (SRM) (Gerstner and Kistler, 2002) is used as SNN model. This general model gives a simple description of spike generation in neurons. The potential of a neuron is determined by the timing of the received and emitted spikes. The SRM incorporates the *Integrate-and-Fire Model*. The potential of a neuron u, at time t, is given by

$$u(t) = \eta(t - \hat{t}) + \sum_{i=0}^{n-1} w_i \sum_{k=0}^{l_i} \epsilon_i(t - t_i^k)$$

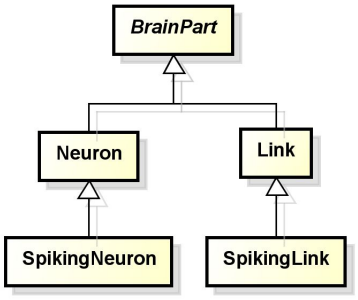
where $\hat{t}$ is the firing time of the most recent spike of neuron u, n is the number of input links of u, w_i the weights of the input links, and t_i^k the firing times of the received spikes from neuron i, i.e. the postsynaptic potentials generated by input link i. A postsynaptic potential is a spike that is emitted by the connecting link whenever the source neuron emits a spike. The number of received spikes that are used for the calculation, the value of l_i, is defined by the `horizon` of the `SpikeEventQueue` in neuron u. This queue holds the most recent received spikes. Function η describes the form of the spike, and function ϵ_i describes the form of the postsynaptic potential.

A neuron emits a new spike when its potential passes a predefined threshold from below. However, it is also possible for the user to explicitly tell a neuron to fire. The second method is intended to enable the user to define when a specific input neuron should emit a new spike, since in the context of spiking neural networks, the network input corresponds to the timing of the emited spikes of the input neurons.

Since the form of the spikes is known, it is sufficient to calculate the firing times of the neuron. There is no need to calculate the entire course of the potential. This significantly simplifies the simulation of spiking neural networks, reduces the computational cost, and as a consequence, allows the simulation of networks with a large number of neurons.

The implementation introduces main classes `SpikingNeuralNet`, `SpikingNeuron` and `SpikingLink`, and functions `Spike` and `PostsynapticPotential`, implementing functions η and ϵ respectively. In the context of the event-driven simulation process, classes `SpikeEvent`, `SpikeEventQueue` and `SpikeEventBuffer`, and a container class `SpikeSet`, for network input and output, are introduced.

1 Basic SNN Classes



Spiking neurons (class `SpikingNeuron`) and **spiking links** (class `SpikingLink`) extend the base classes `Neuron` and `Link`, respectively, and are responsible for generating all events during a simulation run. Since spiking neural networks are simulated, an event represents a spike. Both, spiking neurons and spiking links, add the generated events (spikes) to a referenced event buffer that is contained in the `SpikingNeuralNet` they belong to. The added events are dispatched, from the lowest to the largest timestamp, by the **spiking neural network** (class `SpikingNeualNet`), to the network components that are responsible to handle the events.

To describe the form of the spikes and postsynaptic potentials that are emitted by the spiking neurons and spiking links, respectively, we used their activation function. This means, the activation function of the spiking neuron is equivalent to the function η in the SRM, and the activation function of the spiking link is equivalent to the function ϵ in the SRM.

Since the neurons are only updated when they receive or emit a spike, it is not sufficient to only check if the potential of the neuron reached the threshold at the point in time the event was raised. Therefore, every time the neuron is updated, we have to predict if and when a new spike will be generated in the future. This means that it is determined how the potential

of the updated neuron is altered by the new spike and if, based on the current spike history of this neuron, the potential will reach the threshold of this neuron. If the prediction algorithm, a root finding algorithm, finds an intersection, a new spike, this means a new event, with a timestamp equal to the intersection is emitted. The root finding algorithm is implemented in the class `RootSolver`.

The spiking neural network offers two simulation modes. In default mode, every event is dispatched and handled as soon as it is added to the event buffer. Therefore, the simulation time - the timestamp, with unit milliseconds, of the currently handled spike event - is independent of the real time. In real-time mode, the simulation time equals the real time. This means that an event with timestamp `t` is handled (approximately) `t` milliseconds after the simulation started. In both modes a new thread, executing the entire simulation run, is created and the execution of the programm that initiated the simulation run continues.

There is a new class `SpikingNetFactory`, extending `NetFactory`, that offers methods to conveniently construct feed-forward, as well as recurrent spiking neural networks.

2 Root Finding Algorithm

The currently implemented **root finding algorithm** has two stages. First the algorithm samples the potential of the neuron at discrete points. The step size is adapted, based on the current derivative and distance to the threshold value. As soon as a point above the threshold is found, a bisection search with the last two samples as the left and right bracket, respectively, one above and one below the threshold, is performed.

This algorithm will never return incorrect, non-existing roots. However, due to the discrete sample points, the algorithm cannot guarentee to find every root. This will only occur in extrem cases, where the potential crosses the threshold for an extrem short time period with a very flat angle. Still, changes to the algorithm for the adaption of the step sizes in the sampling stage, may further minimize the probability that intersections are missed.

3 Spike Forms

The class `Spike` provides the **default function** that describes the form of the spikes. This means, it provides the function η in the SRM. The defaults for the parameters are set to values that ensure that the function describes the form of spikes, generated by biological neurons.

$$\eta(t) = \text{abs} \cdot \text{H}(\delta_{\text{abs}} - t) - \text{rel} \cdot \text{e}^{-\frac{(t-\delta_{\text{abs}})}{\delta_{\text{rel}}})} \cdot \text{H}(t - \delta_{\text{abs}})$$

where H is the heaviside function, δ_{abs} the duration of the absolute refractoriness phase in milliseconds, δ_{rel} a factor that defines the duration of the relative refractoriness phase, *abs* the maximum value in millivolt that the neuron potential is above its resting potential during the absolute refractoriness phase, and *rel* the maximum value in millivolt that the neuron potential is below its resting potential during the relative refractoriness phase. The resting potential of a neuron is its potential in the absence of emitted and received spikes. The terms absolute and relative refractoriness are borrowed from descriptions of spikes of biological neurons. During the absolute refractoriness phase, the neuron cannot fire a new spike, during the relative refractoriness phase, the neuron is less likely to fire a new spike.

The following table shows all parameters of the class `Spike` that can be set by the user.

Parameter	**Spike Feature**		**Default Value**
`abs`	Absolute Refractoriness Maximum	`[mV]`	110
`rel`	Relative Refractoriness Minimum	`[mV]`	55
δ_{abs}	Absolute Refractoriness Duration	`[ms]`	2
δ_{rel}	Relative Refractoriness Duration	`[ms]`	0.5

The class `PostsynapticPotential` provides the default function that describes the form of the postsynaptic potentials. This means, it provides the function ϵ in the SRM. The defaults

for the parameters are set to values that ensure that the function describes the form of postsynaptic potentials, generated by biological synapses.

$$\epsilon(t) = \alpha \cdot (e^{-(\lambda \cdot t)} - e^{-(\mu \cdot t)})$$

where α scales the minimum, in case of inhibitory postsynaptic potentials, or maximum, in case of excitatory postsynaptic potentials, as well as the ascending and descending slope of the postsynaptic potential. The effect of the factors λ and μ depends on their values relativ to each other. If λ is smaller than μ, we get an excitatory postsynaptic potential. If λ is larger than μ, we get an inhibitory postsynaptic potential. The factors λ and μ also define the ascending and descending slope of the postsynaptic potential.

The following table shows all parameters of the class `PostsynapticPotential` that can be set by the user.

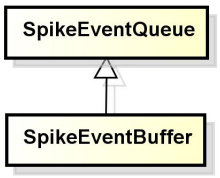
Parameter	Postsynaptic Potential Feature		Default Value
α	Scales the Maximum/Minimum Scales the ascending and descending slope	`[mV]`	1.5
λ	Defines, with μ, if excitatory or inhibitory Defines, with μ, the ascending and descending slope	`[mV]`	0.7
μ	Defines, with λ, if excitatory or inhibitory Defines, with λ, the ascending and descending slope	`[mV]`	0.8

Both classes offer parameterized constructors to allow the user to set these parameters.

4 Events and Event Management

An **event** (class `SpikeEvent`) in the simulation represents either an emitted spike, or a received spike (postsynaptic potential). The type of the event, i.e. the type of the spike, is identified by static constants. An event object has two references of type `BrainPart`, identifying which network component generated the event and which network component will handle the event, respectively. Also, it has a timestamp, a double value representing the time in milliseconds, and a reference to a `Function`.

If the event represents an emitted spike, the function describes the form of the spike. If the event represents a received spike, i.e. a postsynaptic potential, the function describes the form of the postsynaptic potential.



The **global event buffer** (class `SpikeEventBuffer`) extends the `SpikeEventQueue` class and is the heart of the simulation. Every event that is generated by the spiking neurons and spiking links is added to the buffer. The events in the buffer are queued and dispatched in chronological order, the oldest event is the head of the queue and dispatched first.

Every spiking neuron has two **local spike queues** (class `SpikeEventQueue`) to store already emitted and received spikes. The queue extends the standard queue interface to include methods for spike management. Most important, the queue has a property `horizon`, which defines the time span a spike remains in the queue. Every time a spiking neuron handles an event, the current simulation time is compared to the timestamps of the queued spikes. Every spike, with a distance larger than the horizon of its queue, is removed. This is reasonable, since the effect of spikes on the neuron potential diminishes over time.

5 Network Input and Output

A **spike set** (class `SpikeSet`) holds the network input, the recorded network output, as well as target outputs for potential training purposes. The input of a spiking neural network consists of lists that contain timestamps (double values), one list for each input neuron. The spike set offers methods to link a given input neuron to a given list of timestamps. This means each list defines the points in the simulation time, at which the linked input neuron will emit a spike. The actual form of the emited spikes is described by the activation function of the input neuron.

Like the network input, the network output also consists of lists of timestamps, one for each output neuron. A specific list contains the timestamps of the emitted spikes of the output neuron that is linked to this list. Recording of spike timestamps is turned on/off separately for each output neuron.

6 Using Spiking Neural Networks

The complete source code for the following examples can be found in the test programm `test.SpikingNetTest`.

Use the static methods of the class `SpikingNetFactory` to conveniently create spiking neural networks. You can either provide your own templates for the neurons, links and activation functions, or just set the corresponding argument to null, to use the defaults.

To create a standard feed forward spiking neural network with random delays for the links, we use the following call:

```
SpikingNeuralNet net =
  SpikingNetFactory.createFeedForward(
  new int[] {3,3,3},// 3 input, 3 hidden and 3 output neurons
  true,   // fully connected
  null,   // neuron template
  null,   // link template
  null,   // spike form template
  null,   // postsynaptic potential form template
  new double[]{0,5}); // random delays in the interval [0,5] ms
```

Next, we create a spike set and create some random network input. This tells the input neurons when to emit a spike. This can be achieved as follows:

```
SpikeSet spikeSet = new SpikeSet();
// iterate over all input neurons and create for each input neuron
    a list of random timestamps (unit milliseconds) that tells the
    input neuron when to emit a new spike
for (int i=0; i < net.getInputNeuronCount(); i++) {
  SpikingNeuron inputNeuron =
      (SpikingNeuron)spikeNet.getInputNeuron(i);
  // the list telling the input neuron, handled in the current
      iteration, the times at which it should emit a spike
  List<Double> input = new ArrayList<Double>();
  for (int j=0; j < 10; j++) {
    input.add(Math.random() * 200);
  }
  // link the created list of timestamps to the input neuron
      handled in the current iteration
  spikeSet.setInputSpikes(inputNeuron, input);
}
```

Now that we linked each input neuron to a list of timestamps, that tell the input neuron when to emit a new spike, we can set the network input with the following single line:

```
net.setInput(spikeSet);
```

To record the output of a specific neuron, we have to tell the neuron to start recording to our spike set:

```
// iterate over all output neurons and turn for each output neuron
    the recording of spike timestamps on
for (int i=0; i < net.getOutputNeuronCount(); i++) {
  SpikingNeuron outputNeuron =
      (SpikingNeuron)net.getOutputNeuron(i);
  outputNeuron.recordStart(spikeSet);
}
```

Next, we start the simulation and wait until the state of the network is stable. This means we wait until the event buffer is empty, because if there are no more events to handle, no new events can be generated.

```
net.innervate();
while (net.isActive()) {
  try { Thread.sleep(1000); }
  catch (InterruptedException e) { e.printStackTrace(); }
}
```

To get the recorded output of a specific neuron, use following code:

```
// get the recorded output of all output neurons
for (int i=0; i < net.getOutputNeuronCount(); i++) {
  SpikingNeuron outputNeuron =
      (SpikingNeuron)net.getOutputNeuron(i);
  List<Double> output = spikeSet.getOutputSpikes(outputNeuron);
}
```

The current simulation run can be stopped in two ways. The preferred method is to stop the current simulation run explicitly. Also, this way the simulation run can be continued later:

```
// stop current simulation run explicitly
net.stop();
// this way it is possible to resume the current simulation run
net.resume();
```

The second method is to stop the current simulation run by starting a new simulation run:

```
// stop current simulation run implicitly by starting a new one
net.innervate();
```

Saving and loading a spike set can be achieved as follows:

```
spikeSet.save(new File("data"));
SpikeSet myData = new SpikeSet();
myData.load(new File("data"));
```

6.1 Real-time mode

A spiking neural network is switched to real-time mode with following call:

```
net.setRealTimeMode(true);
```

Make sure to stop the current simulation run before switching simulation modes.

In the following short example, random input neurons emit random spikes for 5 seconds. Since we are in real-time mode, the network will be active for at least 5 seconds. We can set the network input in the same way we did in standard mode. However, in cases like this, where new input is given repeatedly and often, and is also very small, it is more convenient and effective to use the following method:

```
net.innervate();
long start = System.currentTimeMillis();
// Run for 5 seconds
while (System.currentTimeMillis() - start < 5000) {
  // wait for some random time period
  try { Thread.sleep((long)(Math.random() * 25)); }
  catch (InterruptedException e) { e.printStackTrace(); }
```

```
  // select random input neuron
  randomNeuron =
      (SpikingNeuron)net.getInputNeuron((int)(Math.random() * 3));
   // explicitly tell the picked input neuron to fire
   // the timestamp is equal to the time elapsed since this
       simulation run was started
  randomNeuron.fire((System.currentTimeMillis() - start));
}
```

Part V
Future Work

The framework is currently quite stable in the implemented form, and no major bugs should surface during use. However, there are a number of features and other things that are desirable, yet couldn't be realized in the project timeframe.

One point of future work is that the framework hasn't been extensively tested. To do that properly, a suite of **test classes** that automate future checking should be created. Also, the **performance** of the framework should be evaluated, optimized as far as possible and desirable in relation to maintainability, and compared to other frameworks. It was never the intent of the project to create something for mainly modeling standard networks, but it was a goal to provide for easy evolving and trying out new kinds of networks, because for standard types, there are already many excellent simulators available. It should be **evaluated** under which circumstances the Boone framework would be of good use.

Also, we initially intended to implement a greater range of neural network types, such as **Elman** and **Kohonen** network types. These fell away due to time constraints, but nonetheless they would come in handy. Also it would be interesting to implement the use of **"layer" objects** with groups of neurons instead of single connected neurons, which should improve speed greatly for simple network types. The operations, such as evaluation and training, could then be realized as matrix functions on arrays of input values.

Currently, the algorithm of network operation is fixed inside of the neuron and link objects, although it is still quite flexible; if this operation should work in a non-standard way, neuron, link and/or neural network classes have to be overridden. An idea to be more flexible is to use a **"net function"**, which would encapsulate the concrete algorithm of updating the network. Another possible goal is to implement **time-coded operation** instead of the current rate-coded operation mode.

The I/O framework can currently only load and save files in the Boone native XML format; it should be able to load and maybe save files in the **SNNS** and **Joone** format for interfacing with other simulation environments.

Finally, the **emulation layer for NetJEN XS** to use the Boone framework is currently realized with minimum effort, and only with a local development version; those changes should be committed to the main version of NetJEN XS. When doing this, the network framework that is actually used should be better encapsulated using e.g. the EvoNet class as an intermediary.

Class Identifying Names

The following class names are registered by default for serialization and the NetFactories:

- **Basic:**
  - boone.NeuralNet
  - boone.Neuron
  - boone.Link
  - boone.map.Position

- **Net factories:**
  - boone.NetFactory$FeedForward (default)
  - boone.NetFactory$Recurrent
- **Trainers:**
  - boone.training.HebbTrainer
  - boone.training.BackpropTrainer (default)
  - boone.training.RpropTrainer
  - boone.training.HopfieldDeltaTrainer
- **PartTrainers:**
  - boone.training.Hebb$NeuronTrainer
  - boone.training.Hebb$LinkTrainer
  - boone.training.Backprop$NeuronTrainer
  - boone.training.Backprop$LinkTrainer
  - boone.training.Rprop$NeuronTrainer
  - boone.training.Rprop$LinkTrainer
  - boone.training.HopfieldDelta$NeuronTrainer
- **Functions:**
  - boone.map.Function$Composite
  - boone.map.Function$Identity
  - boone.map.Function$AtLeast
  - boone.map.Function$AtMost
  - boone.map.Function$LessThan
  - boone.map.Function$GreaterThan
  - boone.map.Function$AboutEqual
  - boone.map.Function$Exponential
  - boone.map.Function$Sigmoid (default)
  - boone.map.Function$Signum
  - boone.map.Function$SignumWithoutZero
  - boone.map.Function$Sinus
  - boone.map.Function$TanHyp
  - boone.map.Function$Clip
- **Error Functions:**
  - boone.TrainingSignalGenerator$SquareSumError (default)
- **File handlers:**
  - boone.io.XNetworkFile
  - boone.io.XnetPatternFile
  - boone.io.snns.SNNSPatternFile

Glossary (Selected)

Cycle

A cycle is presenting a pattern to the network exactly once, during network evaluation. For recurrent networks, uch as Hopfield networks, it is often necessary to evaluate the network repeatedly for a given input pattern, until the output pattern is stable, or a maximum number of evaluations has been reached. In other words, a cycle is exactly one such evaluation step.

For feed-forward networks, this usually isn't necessary, because the algorithm just needs a single cycle to calculate the network output for the given pattern.

Epoch

A training epoch is the duration of training a complete set of training patterns. For example, the methods of epoch listeners are called before or after training a complete set of training patterns, as opposed to turn listeners, the methods of which are called after every single pattern.

Net factory

A net factory can assemble a finished neural network, according to some parameters that are specified by the user. It is possible to do this construction in the application using the framework, but most of the time, this factory provides enough functionality to be a shortcut.

Turn

We call one pattern presentation a training turn. For example, the methods of turn listeners are called before or after training every single pattern, as opposed to the epoch listeners, which are called before or after training a complete set of training patterns.

Bibliography

[Bishop 02]

Bishop, Christopher M. Neural Networks for Pattern Recognition Oxford University Press, 2003; ISBN 0-19-853864-2.

[Huber 95]

Huber, Reinhold; Schweiger, Roland Object-Oriented Analysis of Biological Neural Networks using OMT Technical Report siGis-004-1995; Salzburg Interest Group on Integrated Systems.

[NetJENXS 02]

Lukesch, Stefan; Planitzer, Dietmar NetJEN XS Framework – Technical Manual Published internally, 2002.

[Riedmiller 94]

Riedmiller, Martin Rprop – Description and Implementation Details Technical Report; University of Karlsruhe, Germany.

[Rojas 96] Rojas, Raul Neural Networks – A Systematic Introduction Springer-Verlag, 1996; ISBN 3-540-60505-3.