NeuroLib user's and programmer's Manual

J. J. Merelo,
GeNeura and Swarm Teams,
Departamento de Electrónica y Tecnología de Computadores
Facultad de Ciencias
Campus Fuentenueva, s/n, 18071 Granada (Spain)
e-mail: `jmerelo@kal-el.ugr.es`

September 22, 1995

1 Introduction

This `neurolib` is intended to be a basic neural network library to use with other Swarm objects and agents. It is not intended to be complete, or comprehensive, it just provides a framework for designing other neural networks that can be easily mixed in a program with the current ones, and some ready-to-use examples to have a first go at neural-equipped agents. It's quite complete, though, and for most purposes, you can use one of the networks provided with it, or subclass them. There is also a function that links this library to genetic algorith, wo that networks can be evolved.

As Swarm is a system mainly intended for complex systems and artificial life simulations, the neural networks included in the package are mainly those that have been used or could be used in artificial life, although they could also be useful in simple pattern recognition tasks.

2 Neural Networks

A neural network is an algorithm that resembles the way in which the brain works [[2]. It is composed of *neurons* or nodes, each one of them a simple model of a physiological neuron, and *weights*, which are connections with different strength between two neurons. Some of the neurons receive their input from the environment, and are usually designated *input neurons*, and some others give back their output to the environment, and are called *output neurons* (see Figure 1).

Usually neurons are arranged in *layers*, which has a physiological basis too. All layers not connected directly to the environment are called *hidden layers*.

Neural networks are useful in artificial life because of its capability to *learn*. That usually means that they can adapt by changing the value of connections between neurons. There are several kinds of neural networks, attending to the way in which they change weights depending on inputs:

- *Supervised*: the neural network is given the desired output, and weights are changed according to the difference between desired output and actual output.
- *Reinforced*: The network is not presented with the correct output, but receives some signal indicating whether the output is correct or not.
- *Unsupervised*: the network self-organizes according to its own reaction to the inputs, until it reaches a stable state after many input sample presentations.

There is another way of classifying neural networks, according to the propagation of signals inside the net:

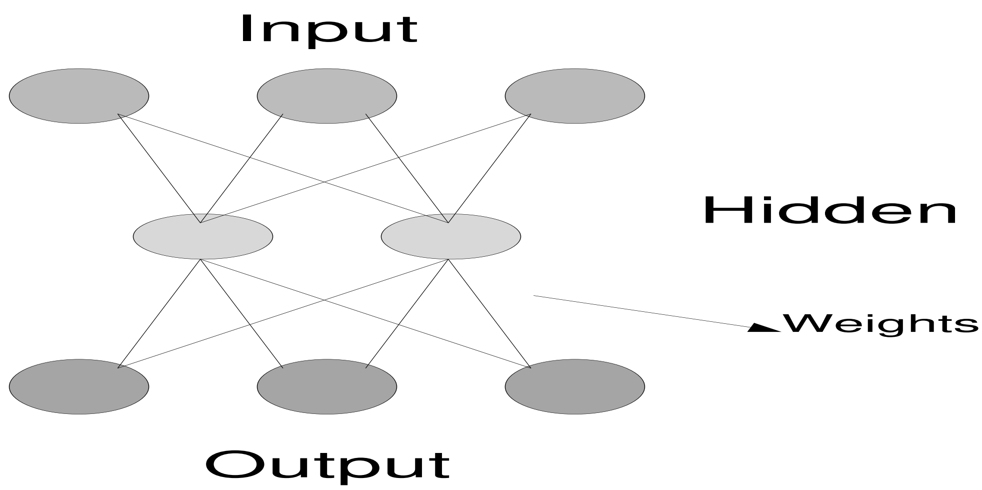

Figure 1: *Usual representation of a feedforward and multilayer neural network, with input neurons on top, a hidden layer, and output neurons in the bottom.*

- *Feedforward*: signal propagates layer by layer, each layer receiving only inputs from its previous layer and giving outputs to its next layer.
- *Feedback or Recurrent*: there are recurrent connections from each layer and neuron to itself, in such a way that the network runs for a while before reaching a stable state. That stable state is considered the output of the network. In these networks all neurons act as input and output neurons, which means that they might all receive inputs from the environment, and its output used to perform some action.

Although all kinds of neural networks are potentially useful in artificial life, not all of them are usually seen in artificial worlds. Supervised neural networks are usually meaningless, since agents do not have a professor that tells them continuosly what they should do to each possible input. Reinforcement learning neural networks are usually the networks of choice, since it is usually possible to know in advance what should be considered a good choice and a bad choice, either by hardwiring it or by genetically coding it inside the agent. Sometimes, purely unsupervised neural networks are used, usually in combination with some selection procedure. And finally, neural networks with fixed weights can also be used, usually combined with some generational method of weight change, like genetic algorithms.

On the other hand, feedback neural networks are usually too heavyweight to be included into an agent, although they will probably be included in the future when parallel execution capabilities are in place. A specific kind of recurrent network is included, which does not need to reach a quiescent state to read its output, and thus is quite fast.

In the present version, `neurolib` includes 5 different neural networks, as well as objects that can be used to implement new neural networks by inheritance. These neural networks and learning algorithms are: Kohonen's Self-Organizing Map, single and multilayer hebbian learning feedforward neural network and competitive learning single layer neural network.

3 `neurolib` structure

`neurolib` has got several different neural network object hierarchies. Within the Feedforward-type networks, there are two hierarchies: the *multilayer* hierarchy and the *single-layer* hierarchy. The *multilayer* hierarchy has for the time being only one member, `Hebb2Layer`, while the *single-layer* hierarchy has got 3. That hierarchy is represented in the figure 2. There is a single member in the recurrent hierarchy, for the time being.

All neural networks added to `neurolib` should support the protocols included in the `neuron.h` header, in order to be partially interchangeable with the others. That protocol includes functions for feedforward and

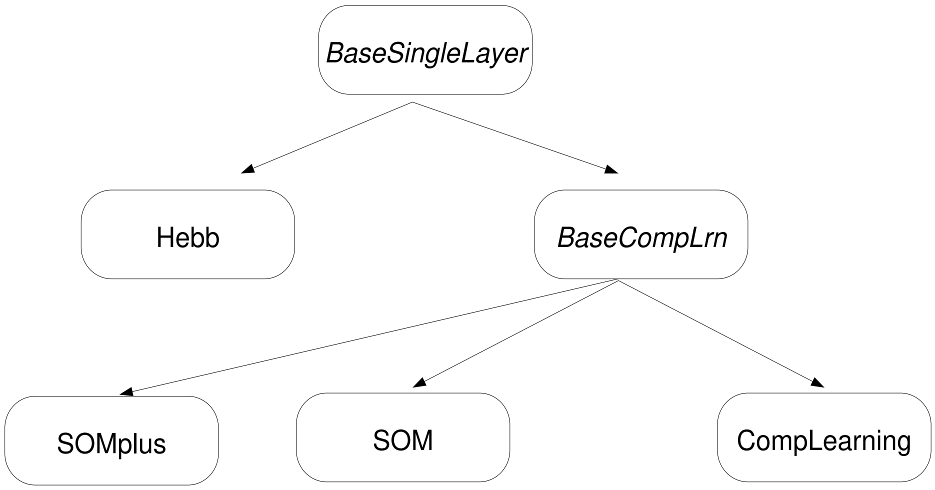

Figure 2: *Class hierarchy in* `neurolib` *. Classes in* italics *represent* abstract *objects, that should not be instantiated.*

training, as well as supervised or unsupervised neural networks. For a complete description, see Appendix A.

In this hierarchy, the *base* objects represent abstract base classes that should not be instantiated, since they have some essential functionality missing, usually training and/or feedforward functions. An error will be issued if you try to instantiate them.

Some of the functions have not been implemented, like the supervised training method. However, they are there like placeholders so that any supervised implementation or a neural network could be interchangeable with any other.

3.1 Single Layer Hebbian Learning Neural Network: `Hebb`

Hebbian learning was probably one of the first methods described to change the weights of a neural network, and it has got a physiological basis (although this claim is probably polemic). It basically states that the weight that connects two neurons should be reinforced if both neurons fire at the same time. It was lately generalized by adding the rule that the weight should be weakened if both neurons fire differentially. That can be expressed in the following way, being A_j and B_i the activations of neurons i and j, and w_{ij} the weight that links both:

$$\Delta w_{ij} = \delta sgn(A_j B_i) \tag{1}$$

where

$$B_i = \sum w_{ij} A_j \tag{2}$$

. sgn is the signum function, that gives 1 for positive numbers and -1 for negative. Output is the numeral of the neuron that has the maximun activation, starting to count from 0.

This network is quite similar to the single-layer perceptron, and very much as useful. It can discern inputs that are linearly separable, but training is fast. Besides, it is a subclass of `BaseSingleLayer`, reimplementing the train and feedForward methods.

3.2 Multilayer Layer Hebbian Learning Neural Network: `Hebb2Layer`

This is basically a hebbian learning neural network with two layers. The main differences with the single layer neural network is that the step function is applied to the outputs of the hidden layer before passing them on to the second layer. Besides, only possitive reinforcement is applied to the first layer, while the second layer receives both possitive and negative reinforcement.

`Hebb2Layer` adds a message to the standard protocol, **`normalize`**, which normalizes weights so that

$$\sum_i {w_{ij}}^2 = 1 \tag{3}$$

for both layers.

3.3 Competitive Learning Neural Network `CompLearning`

Competitive learning neural networks are somewhat different to the hebbian neural networks we saw before. While these corresponded to the *cooperative* learning paradigm, in which all weights change responding to every input, in *competitive learning* neural network only the weight corresponding to one neuron, called *winner*, or a group of neurons, changes responding to inputs. Competitive learning neural networks are quite similar to what are called vector quantization algorithms or template-based methods in the pattern recognition literature.

In this case, the basic idea is that for each input vector, an "output" or winning neuron is selected; this neuron is changed in such a way that it becomes closer to the input vector, i.e., for a given input A_j, being i the winning neuron index

$$\Delta w_{ij} = \alpha(A_j - w_{ij}) \tag{4}$$

Each output neuron has a label, corresponding to a class. The class label of the winning neuron is output, for each input. This neural network is probably more efficient than the others, since it does not need to make many floating point operations for weight updating each time a new input is submitted, but on the other hand is quite sensitive to initial conditions and labelling, in such a way that some initializations might make the network not converge altoghether. Probably used best in combination with genetic algorithms or a wise initialization method.

The **`BaseCompLrn`** abstract superclass implements all the basic functionality of the competitive learning paradigm, like initializing the net, returning the value of a weight, or feeding an input vector to the network. It is actually a subclass of **`BaseSingleLayer`**. This class as well as the other inherit from tt BaseCompLrn.

There are two methods added to the original protocols:

- `-setClasses: (unsigned ) c` sets the number of classes the neural networkwill return, which can be, and usually, is different from the number of neurons. This message should be issued before `createEnd` .
- `-setOrderedLabels` assigns class labels to neurons sequentially, so that label(i) = i % (number of classes). Should be issued before `createEnd` , too.

3.4 Kohonen's Self-Organizing Map `SOM`

This network is also a direct sibling of the basic competitive learning method, once again, training and feedforward methods are overriden. It was first described by Kohonen in 1982. A good description can also be found in [3] It is similar to the former, except that not only the winning neuron changes weight, but also the two neurons that are beside it. These two neurons are called a *neighborhood*, and several definitions other than the basic 2-neuron neighborhood are possible. There are not class labels in each neuron, and the order of the winning neuron is output, although this does not usually matter very much. The parameter α decreases with time. The main result concerning the self-organizing map is that it tries to cover the input space. In this sense, it could be useful for robotics simulation applications.

There are several versions of the Self-Organizing Map, the one implemented has been the one-dimensional SOM. It should be a straightforward exercise, left to the student, to program a 2-dimensional version.

The other class mentioned, SOMplus, is a more sophisticated implementation of this basic algorithm, with more parameters to twiddle with.

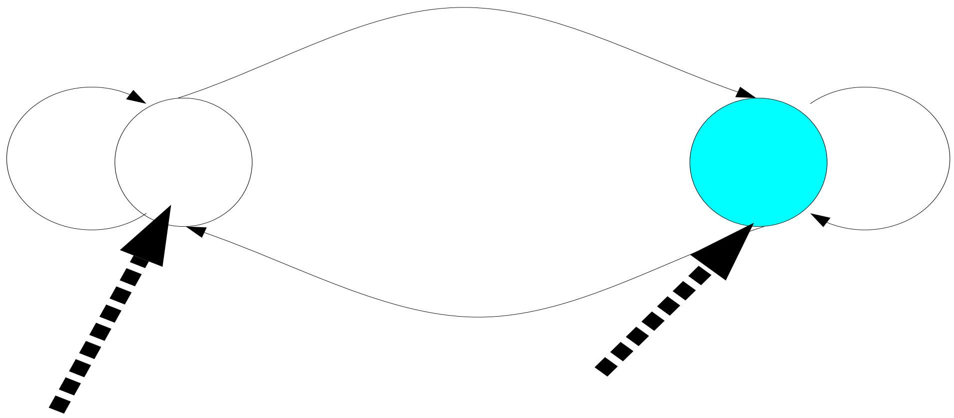

Figure 3: *Representation of a 2-neuron continuous time recurrent neural network. Each neuron has connections to itself, from outside (thick dashed arrows), and to the other neuron.*

3.5 Continuous Time Recurrent Neural Networks: `CRNN`

This type of NN is a development of Hopfield's neural network [2], in which some constraints have been suppresed. I is used by Randall Beer in his robotic simulations and real robots [1].

The usual way this neural networkworks is by receiving input from the environment (which is denominated *clamp*), and then it trundles for a while until it arrives to an attractor, which usually happens in a few steps.

The general formula is

$$\tau_i \dot{y_i} = -y_i + \sum_{j=1}^{N} w_{ji}\sigma(y_j - \theta_j) + I_i(t) \qquad i = 1, 2, \ldots, N \tag{5}$$

The loop used in the others neural networks is not applicable here: it is usually a matter of clamping neurons to an input, and then wait until the outputs have become stable. In an artificial life environment, some neurons can receive input from the environment and the activation of other neurons used to drive actions of the agent; besides, there can be interneurons, which only receive input from other neurons. There is not, however, a clear layer structure.

In this case, weights do not vary with inputs, but the fact that the network arrives to different attractors makes it quite rich in behaviors. There are, of course, methods to make weights change, but the most usual way is to use a evolutive procedure.

4 How to work with `neurolib`

As can be seen in the protocols included in `neuron.h`, many of the methods are interchangeable, which means that a neural network object can respond tomany of the messages in the same way. You can imagine a neural network as a black box that receives inputs, gives outputs, and whose behavior can be altered by sending it some signals.

Neural networks are created using the usual Swarm creation protocol, starting with `createBegin` and ending it with `createEnd`, like this.

Parameters that must be set in each case are usually preceded by the "set" prefix. Anyways, `createEnd` checks for the correctness of the creation, and returns `nil` or issues an error message if something is wrong.

Once the neural networkhas been created, it is usually inserted into a loop in which it is fed some input vector, its output checked against something, and then weights changed; this happens over and over again until some minimun error is reached or until Armaggedon.

```
id neura ;

neura = [Hebb2Layer createBegin: aZone];
[neura setLayerSize: INPUTSIZE: numNeurons : hidSize ];
[neura setRandomWeights];
if ( ! ( neura = [neura createEnd] ) )
    exit(1);
```

Figure 4: Protocol for creation of the neural net, in this case a multilayer Hebbian network

```
// Random input in the [-1,1] range
for ( j = 0 ; j < INPUTSIZE; j ++ ) {
    inp[j] = [uniformRandom rFloat] * 2 -1 ;
}

// x > 0 == class 0; x <= 0 class 1
output = [neura feedForward: inp ];
if ( (( output == 0 ) && ( inp[0] > 0 )) // Two classes >0 and <= 0
      || (( output == 1 ) && ( inp[0] ≤0 ) ) ){
    [neura train: (float) 1 ];
} else {
    error++;
    [neura train: (float) -1 ];
}
```

Figure 5: Training and feeding protocol valid for all kinds of feedforward neural networks

The `train` method receives only reinforcement signals, learning constants are usually defined inside the object.

4.1 Graphical representation

At this point, no representation other than standard Swarm graphics is included. These graphics are two-dimensional, and so can only be used to represent two-dimensional weights, and two-dimensional inputs. Higher-dimensional weights can be represented in several windows as 2-dimensional slices. Probably a future version of `neurolib` will include some kind of graphic representation for weights, like the Rumelhart representation.

A example of how to represent graphically weights is shown in figure 6, and the output in figure 8

The piece of code shown in 6 is valid both for multilayer and single layer networks; only first-layer weights are reprsented in the former case. Input vector can be represented in the same way, including this code

This bits of code have been extracted from the `neurodemo` example, that shows how competitive learning and Hebb 2-layer networks adapt to inputs.

4.2 Evolving neural networks

The `neuron` protocol provides a way to make neural nets evolve, using Swarm's own GA library or any other method you can come up with. There is a `setFromString` function, which takes a string and converts it to the weights of the net. Every two bytes are scales to a floating point number in the range -1, 1, and assigned to each weight in turn. The order in which they are assigned is the same in which they are stored, so that building blocks are not broken. Ideally,

```
    BLTGraph *wgtGraph;
    GraphElement **wgtData;

    // Create Graph
    wgtGraph = [BLTGraph create: aZone];
    [wgtGraph title: "Weight Wandering" ];
    [wgtGraph axisLabelsX: "WeightX" Y: "WeightY"];
    [wgtGraph setScaleModeX: 1 Y: 1]; // "Loose" mode
    [wgtGraph packWith: "-fill both"];
    wgtData = ( GraphElement ** ) malloc( numNeurons * sizeof(GraphElement *));
    for ( j = 0; j < numNeurons; j ++ ) {
        sprintf( Labels, "Weights %d", j );
        wgtData[j] = [wgtGraph createElement];
        [wgtData[j] setLabel: Labels ];
        [wgtData[j] setColor: Colors[j]];
        [wgtData[j] setSymbol: "diamond"];
    }

//------- Inside the main loop

    for ( j = 0; j < numNeurons; j ++ ) {
            float w1, w2;
            if ( nnType ) {
                w1 = [neura getWeights: 0: j: 0 ];
                w2 = [neura getWeights: 0: j: 1 ];
            } else {
                w1 = [neura getWeights: j: 0 ];
                w2 = [neura getWeights: j: 1 ];
            }
//          printf( "Weights %f %f \n", w1, w2 );
            [wgtData[j] addX: w1 Y: w2 ];
    }
```

Figure 6: Nibbles of code for representing the weights using Swarm graphics as moving diamonds in a 2-dimensional space.

an neural network-based agent would include both a neural networkand and a genome, and the neural networkwould be created from the genome at birth.

5 Running the demos

There are three demos inside the directory itself, one that uses `CompLearning` and `Hebb2Layer` networks; other a more or less standard demonstration of Kohonen's SOM, and a demonstration of `CRNN` . Get inside the directory and type `make demo` to compile them.

5.1 `neurodemo`

`neurodemo` shows how neural networks in the package can be used in an uniform way. Different code is needed only to set them up. What you see on the screen when you run it is a standard Swarm entry. The only parameters to change is the learning rate, and the number of neurons, which will be the number of neurons in the hidden layer for `Hebb2Layer` and the number of output neurons for `CompLearning` . Default type is `CompLearning` , you can click into the checkbutton to used multilayer hebbian learning.

Click on Go to run the simulation, and a window with two graphics pop up. Both neural networks attempt to classify two-dimensional dots divided in two classes: one for $x > 0$ and the other for $x <= 0$. The upper one is the

```
GraphElement *inpData;

inpData = [wgtGraph createElement];
[[[[inpData setLabel: "Input"] setColor: "Purple"] setSymbol: "circle"] setOnlyPoint: 1];

// ------ and then -----

[wgtGraph addTo: inpData X: inp[0] Y: inp[1] ];
```

Figure 7: Nibbles of code for representing the an input vector using Swarm graphics. as a moving dot.

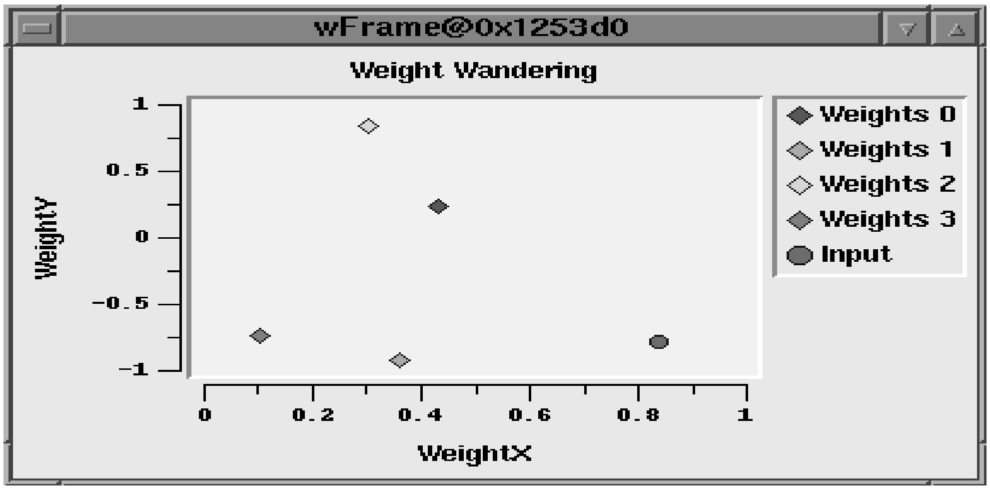

Figure 8: *Graphical representation of the weights of a neural network, as produced by the previous code. Input vector is represented as a circle, and weights as diamonds.*

error during training, ie, the number of times the neural networkhas misclassified an input vector, and the lower one represents the first layer neuronweights in the `Hebb2Layer` case, and neuron weights for the single layer in the other case.

5.2 SOM demo

This demo shows at the same time how the library can be extended and how to use typical neural networks. A new class, `SOMplus`, has been created, making it a subclass of `BaseCompLrn`, which is a more complete implementation of a 1-dimension SOM, and it has been used in the program, SOMdemo.

In this program, neuron weight coordinates are represented as points. Each point is joined to those whose index difference is one with a line. Ideally, if everything works well, the curve that joins the neurons shold change so as to cover maximally the square.

`SOMplus` class just puts all the parameters that govern the SOM back into the object control, cleaning up a bit the main program. Many parameters can be changed from the command line.

5.3 `CRNNdemo`

In this demo, dots represent activations of neurons, instead of the usual weights, which are fixed. Besides, for euclidean problems, only the first two neurons are represented. What happens is the following: an input is generated randomly, and the network is clamped to it. Then inputs are clamped to 0, and the network is let evolve for a while.

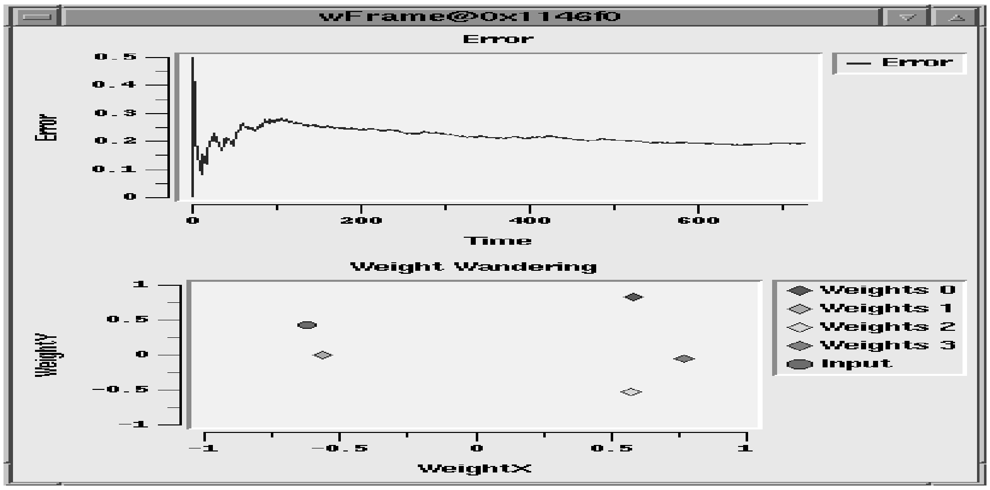

Figure 9: `neurodemo` *demo program.*

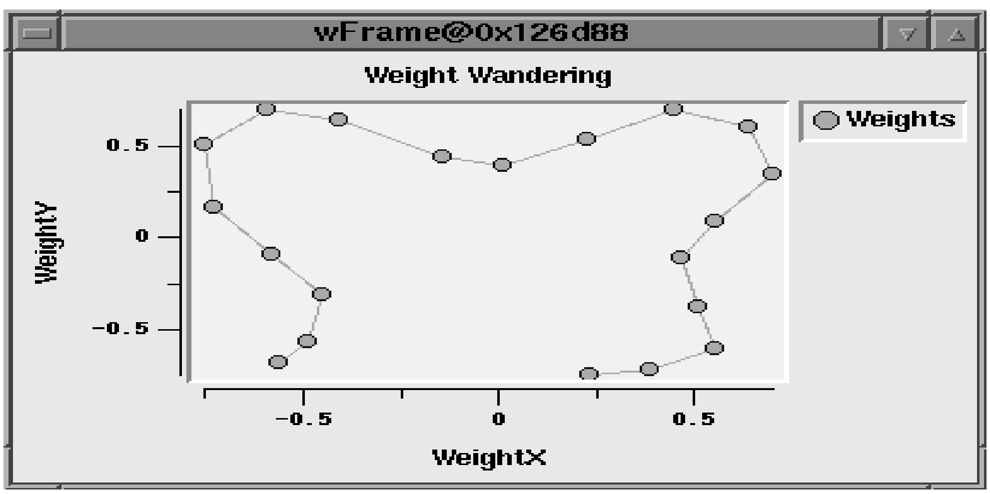

Figure 10: `SOMdemo` *demo program. The curve attemps to cover the square. This was done with 20 neurons default values for the other parameters.*

When it reaches an attractor, which means it no longer means, the activations are plotted in the lower panel, and a new input is generated. This goes on an on for a while.

Acknowledgments

This work has been partially supported by grant PB92-0456 from the DGICYT (Spain).

6 Appendix A: neuron protocol

References

[1] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, To appear.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. McMillan, 1994.

[3] T. Kohonen. The self-organizing map. *Procs. IEEE*, 78:1464 ff., 1990.

```
#ifndef __NEURON_P
#define __NEURON_P

//Standard includes
#include <stdio.h>
#include <malloc.h>
#include <math.h>

// Several things used throughout the library
#import <defobj.h> // For UniformRandom
#import <simtools/global.h>
#import <swarmobject/SwarmObject.h> // for SwarmObject
#import <tkobjc/ZoomRaster.h>

#define BYTESWGT 2

@protocol NN
// I/O related functions
-(unsigned) getInpSize; // Number of input neurons
-(BOOL) getLayers; // True if multilayer <only>

// Weight-related functions
-setRandomWeights; // Randomize weights, result [-1,1]
-setFromString: (unsigned char *) _str; // 2 bytes for each weight

// Destructor
-free; // Dealloc everything
@end

@protocol FeedForward
// Input-related functions
-(unsigned ) feedForward: (float *) inpVec;
@end

@protocol Recurrent
// size and so:
-setNumberNeurons: (unsigned) _n;

// Input/Output related functions
-clamp: (float *) inpVec;
-getActivation: (float * ) _act;
@end

@protocol MultiLayer <FeedForward>

- setLayerSize: (unsigned) inpLyUnits
              : (unsigned) opLyUnits
              : (unsigned) hdLyUnits;

- (float) getWeights: (unsigned) Layer
                    : (unsigned) i
                    : (unsigned) j;// i index always runs over the bottom, or output layer

- setWeightsTo: (float * ) wgtsInputLayer: (float* )wgtsOutputLayer;
- setTrainingConstant: (float) _alpha : (float) _beta;

// Info functions
-(unsigned) getHiddenSize;

@end

@protocol SingleLayer <FeedForward>
- setLayerSize: (unsigned) inpLyUnits: (unsigned) opLyUnits;
- (float) getWeights: (unsigned) i: (unsigned) j;
                                  // i index always runs over the lowest, or bottom layer
- setWeightsTo: (float * ) wgts;
- setTrainingConstant: (float) _alpha;
@end

@protocol Unsupervised // Includes reinforced
- train: (float) delta; // Reinforcement signal
@end

@protocol Supervised
- train: (float) delta: (float*) desiredOutputVec;
@end


// Non-object functions
float sigmoid( float _val);
#endif
```

Figure 11: `neuron.h` protocol