AN IMPROVED NEURAL NETWORK BUILDER THAT INCLUDES GRAPHICAL NETWORKS AND PI NODES

M.S. AL-RAWI
Computer Science Dept, King Abdullah the Second School for Information Technology. Jordan University
email: rawi@ju.edu.jo
phone: 06-5355000-4518
Mail: POB 13496, Amman 11942, Jordan

A. B. TARAKJI
CEO
Spacetoon Online
bisher@spacetoon.com
(+963) 93-858930

Abstract

Neural Network Builder is a software tool that aids in the design, training, and testing of neural networks using an easy user friendly interface. This tool is an efficient neural network simulator, it can be used for projects that can help in the understanding of neural networks by exhibiting different algorithms and ideas and allowing the tweaking and monitoring of the network's many properties. In addition to ease of creating and simulating, the object oriented code provides a valuable software environment that allows the development of new algorithms and theories. To our knowledge, this is the first simulator that implements high order nodes.

Keywords : High Order Neural Network, GUI, Java, Backpropagation, Sigma-Pi networks.

1. Introduction

In a research aimed to build a neural network that could recognize the English alphabet, most of research led to "hard coded" neural networks, networks that could do one task only. These NN were fast but led to problems due to there extreme specialization in a certain task, any change to the number of neurons, the topology, or just simple learning rate needed a recompile. Such parameters could be exposed with some careful programming in runtime, but still the whole network was bound by its specialized nature. In addition to these shortcomings, the code resembled nothing that we had learned in theory; the whole NN was just a big array of pointers with another array that held the connections, making the bridging between theory and practice a hard task for those still new to the entire concept.

The main aim of this research is to build a general network where all aspects could be changed at runtime, in a simple and clear way, using an object oriented design that is analogous to what we have learned. To take this task to a higher level of similarity between theory and practical application, we allowed the network to be drawn and updated graphically bringing additional similarity between the classroom and the screen. Also thrown into the research are the implementation of different theories, theories that are not used a lot in commercial programs since they are too hard to control or inefficient, but are very important to the understanding of NN and how they work. One such feature is the High order neural network (HONN) with PI nodes. Also this is multiplatform, written in Java, it allows this code to be run anywhere without modification, it also lends itself to being easily exposed to the web, and its computations to be distributed across networks.

There are many neural network simulators available both commercial and free (too many in fact) [4], but most of these simulators are system dependent, and written for few platforms (UNIX or Windows). The majority of these simulators have some sort of GUI but most of them do not depict the network graphically, nor allow the creation of PI nodes. There are some software packages that do all that, such as NeuralBuilder from NeuralSolutions, a commercial application that works solely on Windows. There are also some portable C++ simulators such as YANNS, but most of the simulators available are written in C to maximize performance. Not many simulators are written in Java, but the open source project "OpenAi" is a notable example. At the end of the day, a simple search in Google will give the reader thousands of results.

2. Theoretical material used in the neural network builder

Following is a non comprehensive introduction to the theory and mathematics behind neural networks, most importantly the theories relevant to this research. A neural network is a powerful data modeling tool that is able to capture and represent complex input/output relationships. The motivation for the development of neural network technology stemmed from the desire to develop an artificial system that could perform "intelligent" tasks similar to those performed by the human brain. Neural networks resemble the human brain in the following two ways:

- A neural network acquires knowledge through learning.
- A neural network's knowledge is stored within inter-neuron connection strengths known as synaptic weights.

The most common neural network model is the multilayer perceptron (MLP). This type of neural network is known as a supervised network because it requires a desired output in order to learn. The goal of this type of network is to create a model that correctly maps the input to the output using data so that the model can then be used to produce the output when the desired output is unknown The MLP and many other neural networks learn using an algorithm called *backpropagation*. With backpropagation, the input data is repeatedly presented to the neural network. With each presentation the output of the neural network is compared to the desired output and an error is computed. This error is then fed back (backpropagated) to the neural network and used to adjust the weights such that the error decreases with each iteration and the neural model gets closer and closer to producing the desired output. This process is known as "training".

2.1 Multilayer Network Structure

A neural network with one or more layers of nodes between the input and the output nodes is called *multilayer network*.

The multilayer *network structure*, or *topology*, consists of an input layer, two or more hidden layers, and one output layer. The input nodes pass values to the first hidden layer, its nodes to the second and so on till producing outputs.

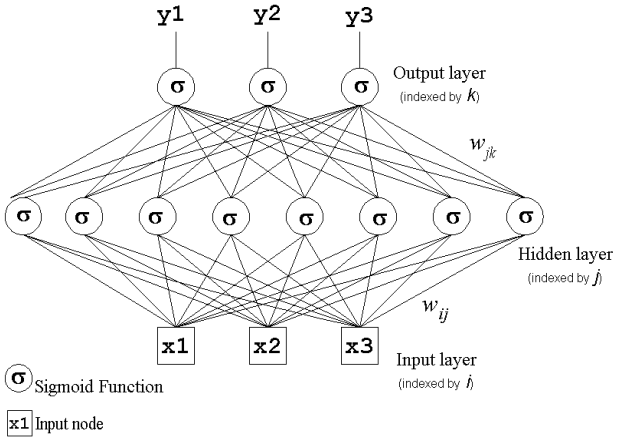

Fig. 1 A Multilayer Perceptron (MLP), A two-layer neural network that implements the function: $f(\mathrm{x}) = \sigma(w_{jk}\sigma(w_{ij}x_i + w_{0j}) + w_{0k})$,

These are the hidden units that enable the multilayer network to learn complex tasks by extracting progressively more meaningful information from the input examples.

2.2 Higher Order Networks

The *higher order neural networks* (HONN) have been developed with intention to enhance the nonlinear descriptive capacity of the feed-forward multilayer perceptron networks. This is achieved by means of increasing the nonlinear descriptive capability of the individual neurons.

A higher order neural network has summation (sigma) as well as product (pi) units. A HONN builds multivariate high-order polynomial models:

$$P(X) = w_0 + \sum_i w_i x_i + \sum_i\sum_j w_{ij} x_i x_j + \sum_i\sum_j\sum_k w_{ijk} x_i x_j x_k + \ldots \quad (1)$$

or written concisely:

$$P(\mathrm{x}) = w_0 + \sum w_i \prod x_j^{\mathrm{r}} \quad (2)$$

The Sigma-Pi neural networks (SPN) are such feedforward networks where each layer contains higher-order terms. Often the layers have summation units feeded via weighted connections by intermediate product unit outcomes.

2.3 Sigma-Pi Network Structure

Sigma-Pi neural networks are sparsely connected HONN. Researchers restrict the polynomial order (that is the network topology) to a configuration sufficient to achieve the desired degree of accuracy using a priori knowledge about the given task.

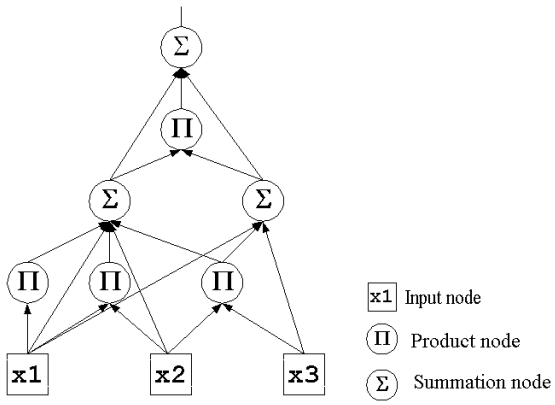

Fig. 2 A Sigma-Pi network

The *sigma units* compute the sum of weighted inputs p_j from the lower *j*-th layer:

$$s = \sum_j w_i p_j \quad (3)$$

The *pi units* compute the product of weighted inputs x_i from the lower *i*-th layer:

$$p_j = \prod_i w_i x_i \quad (4)$$

2.4 BackPropagation:

MLP became applicable on practical tasks after the discovery of a supervised training algorithm for learning their weights; this is the *backpropagation learning algorithm*. The backpropagation algorithm for training is iterative with the weights adjusted after the presentation of each example.

The error backpropagation algorithm includes two passes through the network: *forward pass* and *backward pass*. During the backward pass the weights are adjusted in accordance with the *error correction*

rule. It suggests that the actual network output is subtracted from the given output in the example. The weights are adjusted so as to make the network output closer to the desired one.

2.4.1 Derivation of the Backpropagation Algorithm

The backpropagation algorithm [1] for training the multilayer perceptron implements a *generalized delta rule* according to which with each training example every weight is updated as follows:

$$w = w + \Delta w \qquad (5)$$

where: $\Delta w = - \eta \, \partial Ee / \partial w$ and $Ee = (1/2) \Sigma_k (y_k - o_k)^2$

The implementation of the generalized delta rule requires deriving an expression for the computation of the derivatives $\partial Ee / \partial w$:

$$\partial Ee / \partial w = (\partial Ee / \partial s)(\partial s / \partial w) \qquad (6)$$

The first part $\partial Ee / \partial s$ reflect the change of the error as a function of the change in the network weighted input to the unit. The second part $\partial s / \partial w$ reflects the change of the error as a function of the change of particular weight w to that node. Since:

$$\partial s / \partial w = \partial (\Sigma_l w_l o_l) / \partial w = o \qquad (7)$$

the expression is reduced as follows: $\partial Ee / \partial w = (\partial Ee / \partial s) o$

2.4.1.1 Delta Rule for weights j→k on connections to nodes in the output layer

$$\partial Ee / \partial w_{jk} = (\partial Ee / \partial s_k) o_j \qquad (8)$$

$$\partial Ee / \partial s_k = (\partial Ee / \partial o_k)(\partial o_k / \partial s_k)$$

$$\partial Ee / \partial o_k = (\partial ((1/2) \Sigma_l (y_l - o_l)^2)) / \partial o_k$$

$$= (\partial ((1/2)(y_k - o_k)^2)) / \partial o_k \text{ // comment: from the case } l = k$$

$$= (1/2) 2 (y_k - o_k) [\partial (y_k - o_k) / \partial o_k]$$

$$= - (y_k - o_k)$$

$$\partial o_k / \partial s_k = \partial \sigma (s_k) / \partial s_k$$

$$= \partial \sigma ' (s_k)$$

if σ is sigmoid function $o_k (1 - o_k)$ (this can be obtained from $s (s_k) = 1 / (1 + e^{-sk})$)

Therefore:

$$\partial Ee / \partial s_k = - (y_k - o_k) o_k (1 - o_k) \qquad (9)$$

and when we substitute: $\beta_k = o_k (1 - o_k) [y_k - o_k]$

the *Delta rule for the output units* becomes:

$$\Delta w_{jk} = -\partial Ee / \partial w_{jk} = \eta \, \beta_k \, o_j \qquad (10)$$

2.4.1.2 Delta Rule for weights i→j on connections to nodes in the hidden layer

In this case the error depends on the errors committed by all output units:

$$\partial Ee / \partial w_{ij} = (\partial Ee / \partial s_j) o_i$$

$$\partial Ee / \partial s_j = \Sigma_k (\partial Ee / \partial s_k)(\partial s_k / \partial s_j)$$

$$= \Sigma_k (-\beta_k) \partial s_k / \partial s_j \text{ // comment: from } \partial Ee / \partial s_k = -\beta_k$$

$$= \Sigma_k (-\beta_k) (\partial s_k / \partial o_j)(\partial o_j / \partial s_j)$$

$$= \Sigma_k (-\beta_k) w_{jk} (\partial o_j / \partial s_j)$$

$$= \Sigma_k (-\beta_k) w_{jk} o_j (1 - o_j)$$

Therefore, when we substitute:

$$\beta_j = - \partial Ee / \partial s_j = o_j (1 - o_j) \Sigma_k (-\beta_k) w_{jk} \qquad (11)$$

the *Delta rule for the hidden units* becomes:

$$\Delta w_{ij} = -\partial Ee / \partial w_{ij} = \eta \, \beta_j \, o_i \qquad (12)$$

2.5 Backpropagation Algorithm for Sigma-Pi Networks

The principles of the backpropagation learning algorithm are valid also for Sigma-Pi networks.

2.5.1 Sigma-Pi Delta Rule for weights i→ j on connections to nodes in the hidden layers

$$\partial Ee / \partial s_j = \Sigma_k (\partial Ee / \partial s_k)(\partial s_k / \partial s_j)$$

$$= \Sigma_k (-\beta_k) \partial s_k / \partial s_j \text{ // comment: from } \partial Ee / \partial s_k = -\beta_k$$

$$= \Sigma_k (-\beta_k) (\partial s_k / \partial o_j)(\partial o_j / \partial s_j)$$

$$= \Sigma_k (-\beta_k) (s'_k) (\partial o_j / \partial s_j)$$

$$= \Sigma_k (-\beta_k) (s'_k) o_j (1 - o_j) \quad (13)$$

where for s_k defined: $s_k = w_0 + \Sigma_{j1} w_{j1} o_{j1} + \Sigma_{j1}\Sigma_{j2} w_{j1j2} o_{j1} o_{j2} + \Sigma_{j1}\Sigma_{j2}\Sigma_{j3} w_{j1j2j3} o_{j1} o_{j2} o_{j3} + ...$

its derivative ($\partial s_k / o_{j2}$)=s'_k is: $s'_k = \Sigma_{j1} w_{j1j2} o_{j1} + \Sigma_{j1} \Sigma_{j3} w_{j1j2j3} o_{j1} o_{j3} + ...$

Therefore, the benefit is:

$$\beta_j = - \partial Ee / \partial s_j = o_j (1 - o_j) \sum_k \beta_k s'_k \quad (14)$$

3. The Proposed Algorithm

The main goal of this research is to give the user an intuitive tool to manage High order neural networks, namely Sigma Pi networks. The introduction of Pi nodes in the network complicates the mathematics and algorithms needed to correctly compute the output of the network, especially the computation of backpropagation. We also wanted to allow the user to use arbitrary topology, to connect any node to any other node in the network, allowing the testing of different topologies and configurations. This also creates lots of problems in the network's basic computations. The following algorithms try to find solutions to these problems.

3.1 The network structure:

The first step to finding solutions is to create a structure for the network that allows arbitrary and changing topology. Usually networks are created by using arrays for the nodes and connections; this will not exactly work here.

A better structure would be one less used as it is usually slower; create an object for each node in the neural network, with a list of connection going in and a list of connections going out, so when ever this node is connected to another, we just create a link (which is implemented as another object) and add a reference of it to the out links list of the connect from node, and another reference to the in links list of the connected to node. These nodes can be arranged into layers, each layer having a list of all the nodes in it, and then the high order network can contain many layers, also maintained by a list. Whereas this structure will allow connections between different nodes of the network, it creates a big problem in the computation of backpropagation, since the latter needs to compute the derivative of the error rule. The adding of an unknown number of layers also complicates this task.

To solve these problems, the proposed algorithm breaks down the derivation rules to their basic elements and tries to reconstruct them according to the topology.

3.2 The FeedForward Algorithm

The first algorithm would be the feedforward algorithm, one that presents the network with a sample at its input layer and allows it to be forward propagated till we reach the output layer.

This algorithm is fairly simple with the suggested network structure, get a layer then loop through all its nodes computing the output with the following steps:

compute the output for each node:

a. sum = 0 ; i = 1;
b. get the incoming link(i) from the in links list
c. get the other endpoint's output Oj(the neuron from which the link came from)
d. multiply Oj by the links weight
e. add Oj to sum
f. i = i+1
g. if there are more links go to (b)
h. the output = σ(sum) //pass the sum through the activation function

3.3 The Backpropagation for Sigma Nodes Algorithm

There are two kinds of sigma nodes: output nodes and hidden layer nodes.

3.3.1 Output Layer Nodes

The Delta rule for the output units is:

$$\Delta w_{jk} = -\partial Ee / \partial w_{jk} = \eta \beta_k o_j \quad (15)$$

$$\beta_k = \partial\sigma (s_k) / \partial s_k [y_k - o_k] \quad (16)$$

Where j is the hidden layer and k is the output layer

The steps for computing beta for output units are:

1. compute the output for each node:
2. compute the derivative of the activation function
3. beta = (target – output)* σ′(sum)

After computing beta the computation of the Delta rule is simple.

3.3.2 Hidden Layer Nodes

$$\beta_j = - \partial Ee / \partial s_j = \sigma'(o_j) \Sigma_k (-\beta_k) w_{jk} \quad (17)$$

for the hidden layers the beta is a bit more complicated as we have to accumulate the betas for the layer above the neurons layer, this is easily obtained by reading the out links list and getting the computed beta.

1. compute the sum and output for node

2. errorsum = 0 , i=1

3. get link(i) from out links

4. get the “to” node from link (the node that the link leads to)

5. errorsum = errorsum + to.beta*link.wieght

6. beta = σ' (sum) * errorsum //σ' *is the derivative of the activation function*

3.4 The Backpropagation Algorithm for Pi Nodes

$$\beta_j = - \partial Ee / \partial s_j = \sigma' (s_j) \sum_k \beta_k s'_k \qquad (18)$$

The problem with computing this beta is s'_k where

$$s_k = w_0 + \Sigma_{j1} w_{j1} o_{j1} + \Sigma_{j1}\Sigma_{j2} w_{j1j2} o_{j1} o_{j2} + \Sigma_{j1}\Sigma_{j2}\Sigma_{j3} w_{j1j2j3} o_{j1} o_{j2} o_{j3} + \ldots \qquad (19)$$

is the sum of all the incoming data into this pi node

$$(\partial s_k / o_{j2}) = s'_k = \Sigma_{j1} w_{j1j2} o_{j1} + \Sigma_{j1} \Sigma_{j3} w_{j1j2j3} o_{j1} o_{j3} + \ldots \qquad (20)$$

it is apparent that the derivation $\partial s_k / o_{j2}$ is basically s_k with o_{j2} set to equal 1. With this in mind, the algorithm to compute the s'k for pi nodes is:

1. P is the Pi node, N is the node that P’s output will be derived in respect to
2. i=0;product=1;
3. get link(i) from in links list for P
4. if link(i).from = N then product = product*1
5. else

a- get output o of link(i).from
b- product = product*o

the computation of the beta of such node is the same as Sigma nodes.

4. The Implementation

The following is the implementation of the algorithms presented, these algorithms will be explained more where the code is represented.

4.1 Implementation of the MLP

At the heart of this research is the correct use of object oriented programming to achieve a well structured application, one that can be developed quickly and efficiently, and to be debugged and extended easily. This application must also be user friendly, implementing all graphically interface elements that aid the user in getting the job done. The first choice was to use Java, being a strict object oriented language that allows high productivity through its large library of classes. The second choice was to extend (inherent) from Java’s Graphical User Elements, since most of the classes of the neural network were to be depicted graphically. And the third choice was to allow the changing of any aspect of the network at runtime, but the computational logic had to withstand. With these points in mind, let us look at the classes’ tree:

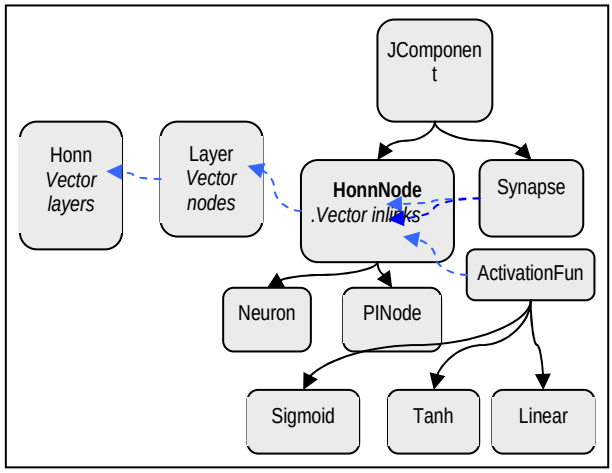

Fig. 3 The main classes used in neural network builder

As shown, the basic element of the Honn (high order neural n etwork) class is the Neuron, but since we have to different kinds of neurons, sigma and pi neurons, we need to build the HONN from a “higher order” class, which is the HonnNode class as shown. This class is abstract since it contains some abstract methods, functions with no bodies that should be implemented in the children of this class. From this class we extend two classes: Neuron, and PINode. This design allows us to use polymorphism on the Honn class, by building from the abstract HonnNode class, and these classes will be defined at runtime to be one the subclasses.

The most important functions of HonnNode are : computeOutput , getOutput, and setActivationFunction. Also in this class we have two Vectors (resizable arrays) that contain pointers to Synapses, the connectors that connect neurons to each other. Each Synapse contains a weight value, this is updated when the backpropagation is computed.

Also within each HonnNode is the activation function, since we wanted to allow it to be changed at runtime, we also allowed the ActivationFunction class to be abstract with two important methods : computeOuput, and computeDerivative, which are overrided by the class that implement ActivationFunction, meaning we can call these functions without knowing which activation function is used by the node.

It also can be noticed that Synapse and HonnNode extend JCompnent from the Java Graphical Components Libraray “Swing”. This allows these two classes to be “drawn” be simply overriding the paintComponent() function. By adding instances of these classes to a container (like a JPanel) these objects will automatically be drawn.

With these building blocks, the Layer class uses HonnNodes to create a "layer" of neurons, the Honn class then uses multiple layers to create a full MLP.

4.2 Implementing the Backpropogation rule:

The following functions are from the Neuron Class:

The first is the computeOutput which calculates the output of this neuron:

```
public void computeOutput()
  {
    Enumeration e = inlinks.elements();
    Synapse s;
    sum=0.0;
    while(e.hasMoreElements())
    {
      s = (Synapse)e.nextElement();
      sum += s.from.getOutput()*s.weight;
    }
    output = aFunction.computeFunction(sum); // activation
function
  }
```

As is apparent the code loops through all the synapses that are going into the neuron and multiplies their weight by the output of the other end (the from neuron), this number is accumulated then to create the sum.

The output is the result of the activation function associated with this neuron with the sum as its parameter.

The next function is the backpropagation function to compute the error delta from the output to the last hidden layer.

```
   public void computeBackpropDelta(double d) // for an
output neuron
  {
    delta = (d - output) *
aFunction.computeDerivative(sum);//sum must be computed
  }
```

you must provide the function with the correct target for this neuron which is simply a floating point number.

4.2.1 Implementing the backpropagation for hidden layers:

This function shows how to compute the backpropagation delta for hidden units, the important piece of code here is the testing if the next node is a SIGMA node or a PI node:

```
if( synapse.to instanceof Neuron)
```

this allows to compute the derivative of pi nodes in a different way, which brings us to the details of implementing backpropagation of PI nodes, as discussed in the next section.

```
  public void computeBackpropDelta() // for a hidden neuron
  {
    double errorSum = 0.0;
    Synapse synapse;
    Enumeration e = outlinks.elements();
    while(e.hasMoreElements())
    {
      synapse = (Synapse)e.nextElement();
      if(synapse.to instanceof Neuron) // the next node is a
sigma node
      {
        errorSum += synapse.to.delta * synapse.weight ;
      }
      else //the next node is a pi node
      {
        PINode p = (PINode)synapse.to;
        double temp = p.computeDerivativeForNeuron(this);
        // now get all the deltas from the nodes connected to
this PI
        Enumeration eS = p.outlinks.elements();
        while(eS.hasMoreElements())
        {
          Synapse s = (Synapse)eS.nextElement();
          errorSum += s.to.delta*temp*s.weight;
        }
      }
    }
```

4.2.2 Implementing backpropagation delta for PI nodes

The hardest part in the previous equation is the computation of s'_k since it totally dependent on the topology of the network. This problem is easily solved with the current architecture of the network which allows the nodes to be connected in an arbitrary way. The previous code presented for the neuron backpropagation delta showed a bit of the computation of s'_k:

```
else //the next node is a pi node
      {
        PINode p = (PINode)synapse.to;
        double temp = p.computeDerivativeForNeuron(this);
        // now get all the deltas from the nodes connected to
this PI
        Enumeration eS = p.outlinks.elements();
        while(eS.hasMoreElements())
        {
          Synapse s = (Synapse)eS.nextElement();
          errorSum += s.to.delta*temp*s.weight;
        }
      }
    }
```

this code uses a function unique to the PINode class: computeDerivativeForNeuron,which computes the sk for this particular Neuron:

5. The Application

This is a typical screenshot of the application written, as seen the application consists of different windows each with its own functionality. The application allows the creation of as many of these windows as necessary to maximize the idea of trying different configurations with different data. The top left window is the main Honn window, which allows creating, editing, connecting and running simulations on high order networks. The top right window is the Input/Output

data window, which allows creating the data for the network and save it if needed. Beneath that is the graph window that shows the error of the network in comparison to the number of iterations (epochs) the network has been trained.

```
public double computeDerivativeForNeuron(Neuron n)
{
  // this function computes the function
  // d(this Pi node output)/d(output of n)
  // by considering the output of n is 1 and computing the
product of the rest
  double prod = 1.0;
  Enumeration e = inlinks.elements();
  Synapse s;
  while(e.hasMoreElements())
  {
    s = (Synapse)e.nextElement();
    if(s.from == n)
    {
      //dont do anything
    }
    else
    {
      prod *= s.from.getOutput();
    }
  }
  return prod;
}
```

Fig. 4 The main GUI window used in the neural network builder.

The console window shows relevant information on the creation and running of the network. Any number of honn windows, graph windows, and data windows can be created using the top menu; this will allow testing a network against multiple data sets without resetting the program. It also will allow the test of different networks (with different topologies for instance) with the same data, making it easy to compare and evaluate.

The main window also contains a graphical depiction of the neural network, showing many important aspects of each node in the HONN. Mouse events are available on the nodes, right clicking on a node will allow the editing of the activation function, the momentum, and the learning rate of the nodes. Left clicking on the nodes will allow connecting of the nodes interactively.

The Data window allows two kinds of inputs, one that takes numbers (floating numbers) as input data (the input vector) and lets the user choose an output from a number of classes. This kind of data is the typical "classified data" where the input represents a "class" from a group of classes.

The other kind of input is what we call "numerical output" data, the input vector represents one floating point number, meaning such data can be recognized by a network with only one output, but this output can take many different values.

The error graph scales automatically when more iterations are run, meaning that one can train the network for a certain number of epochs, monitor results, tweak some parameters and rerun the training process, all while seeing the error graph in the window.

6. A Case Study: The Parity Bit Problem

The parity bit problem is rather interesting as it is used to since it is a very demanding classification task for neural networks to solve, because the target-output changes whenever a single bit in the input vector changes, and usually networks do not converge easily.The main idea is to make the network function as a XOR gate with multiple inputs. With the tool we have created we tried the simplest XOR example: the one with two bits.

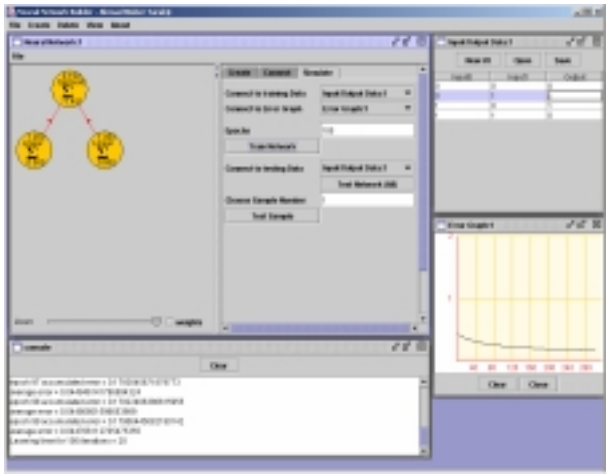

Fig. 5 Without any hidden layers the network will not converge: after 300 iterations the accumulated error is still larger than 0.1 and when testing the samples only 50% are accepted.

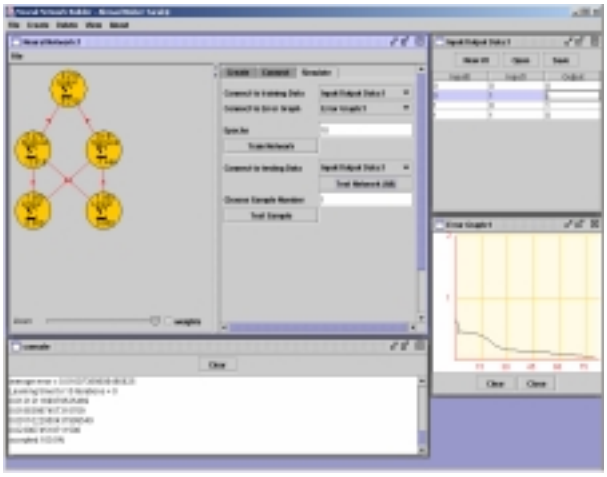

Fig. 6 With a single hidden layer, two neurons: On the first tests, the network didn't converge even after 300 iterations. However, after adjusting the learning rate (0.8) and changing the output function to Linear the network converged easily after about 70 epochs.

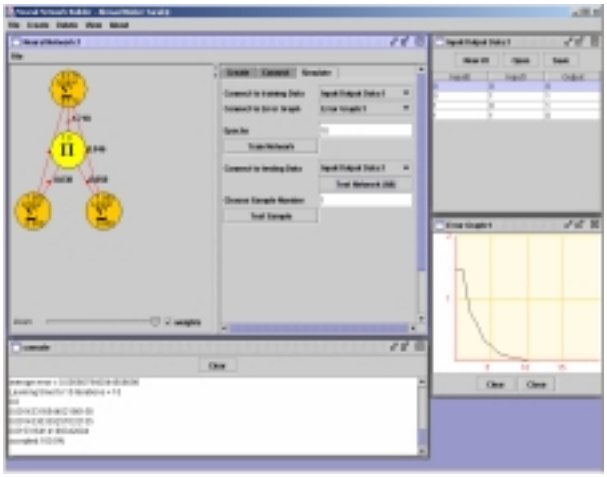

Fig. 7 With a single Pi node and shortcut links: with learning rate 0.6 and the output node with a Linear function this network converges extremely fast: 10 iterations only.

7. Conclusions

All fields of computer science can benefit from a well structured, well designed application with a good user interface, especially one like neural networks, where trail and error is dominant. AI is both an exciting and involving area, and the more tools the researchers are given, the faster they will give us new artificial marvels that we can benefit from. This application will always be a "in work project", since an endless numbers of features and enhancements can be added. Maybe some of the more significant features would be:

- More Neural Networks: the world of pattern classification contains tens of different networks that this application could easily implement with its current structure. Unsupervised networks such as Kohnen networks, and self organizing feature maps could be easily added to the application.
- More modifying algorithms: Algorithms that modify the networks topology or properties in intelligent ways, such as algorithms that change the learning rate at run time such as "CGD" and network pruning algorithms such as "Optimal Brain Damage" algorithm. Even genetic algorithms that allow the "evolving" of the network can be added.
- Adding a compile feature: this feature will allow the user to take the current neural network after he has trained it, then "compile" it into java code, i.e. creating a java class that contains a function that takes an array of data as its parameter. This function will then "classify" this array and return the result. The network would be highly optimized as it has been "frozen" into code, and it would allow use of it in different programs.

8. References

[1] C. Bishop, "*Neural Networks for Pattern Recognition*", Oxford University Press, Oxford, UK, (1995).

[2] L. Franco and S.A. Cannas, "Generalization and selection of examples in feedforward neural networks", *Neural Computation* 12(9), 2000, 2405-2426.

[3] L.Franco and S.A. Cannas, Generalization Properties of Modular Networks: Implementing the Parity Function, *IEEE transactions on neural networks*, 12(6), 2001, 1306-1313.

[4] A.A. Ghorbani and K. Owrangh, "Stacked Generalization in Neural Networks: Generalization on Statically Neutral Problems", *Proc. of the IEEE/INNS International Joint conference on Neural Networks (IJCNN'2001)*, Washington D.C., USA, 2001,1715-1720.

[5] C.L.Giles, and C.W. Omlin, "Pruning Recurrent Neural Networks for Improved Generalization Performance", *IEEE transactions on neural networks*, 5(5), 1994, 848-855.

[6] S. Hochreiter, and J. Schmidhuber, "LSTN Can Solve Hard Long Time Lag Problems", In. Mozer, M.C.,. Jordan, M.I, Petsche, T. eds., *Advances in Neural Information Processing Systems 9 (*NIPS'9), Cambridge MA: MIT Press, 1997, 473-479.

[7] M.E. Hohil, D. Liu, and S.H. Smith, "Solving the N-bit parity problem using neural networks", *Neural Networks*, 12(9), 1999, 1321-1323.

[8] D.E. Rumelhart and J.L, McClelland, *Parallel distributed processing* (Vol.1). Cambridge, MA: MIT press, 1986.

[9] R. Setiono, On the solution of the parity problem by a single hidden layer feedforward neural network, *Neurocomputing*, 16(3), 1997, 225-235.

[10] S. Haykin, *Neural Networks. A Comprehensive Foundation*, Second Edition, Prentice-Hall, Inc., New Jersey, 1999.

[11] N. Nilsson, "Introduction to Machine Learning", Chapter 4: *Neural Networks*, 1996.

[12] M. Heywood and P. Noakes, A Framework for Improved Training of Sigma-Pi Networks, IEEE Transaction on Neural Networks, 6(4), 1995, 893-903.

[13] Y. Shin and J. Ghosh, The Pi-Sigma Network: An Efficient Higher-Order Network for Pattem Classification and Function Approximation, *Proc. Int.Joint Conference on Neural Networks IJCNN, Seattle*, vol. I, 1991, 13-18.

[14] R.Bone, M. Crucianu, J.P. Asselen de Beauville, *Yet Another Neural Network Simulator*, Computer in Use, France, 1998.