smallBrain version 1.1
A Neural Network Simulator written in Smalltalk.

Murali Krishnan
krishnan@cs.uiuc.edu

smallBrain is a neural network simulator. The purpose of **smallBrain** is to demonstrate principles of artificial neural networks. It was written in (ParcPlace) Objectworks\Smalltalk Release 4.1.

1 Introduction

Several different network topologies are built in to **smallBrain**–Back Propagation, Hopfield and SOFM networks–and these are what smallBrain simulates. Due to the object oriented design of **smallBrain**, it is (relatively) easy to expand **smallBrain** to include other topologies.

1.1 installing smallBrain

The source code for **smallBrain** is available by anonymous ftp from `st.cs.uiuc.edu`, in the directory `/pub/st80_r41/smallBrain`. The file `README` gives a description of the files available. Besides the source code, sample data and this document are also available.

The source code is a file called `smallBrain.st`. It can be loaded by opening a file list (from the Utilities menu in the Smalltalk Launcher), reading the file, and then doing a "file in". When this is done, an interface window (see below) will be created, along with another Launcher. The new launcher is the same as the default Launcher except that there is a selection called "smallBrain" added. The old Launcher should be killed, and the new one used instead.

1.2 general use notes

Most of the tools (views, buttons, controllers) used by **smallBrain** are tools supplied by Objectworks\Smalltalk Release 4.1.

In all cases, the right mouse button gives a pop-up menu concerned with window management. In some cases the middle mouse button gives a pop-up menu (called an "operate" menu) concerned with operations specific to that display. In most cases, the left mouse button performs an activity (like panning a scrollbar) specific to that display.

In most cases where text must be typed into a display or existing text modified, it is necessary for the text to be "accepted" into the program. Sometimes this is achieved by pressing the "enter" key on the keyboard. In some displays, the "enter" key is used to insert a carriage return, so it will not perform the accept function. In this case, the operate menu will have an accept option.

If there is any text in a display which has been modified but not yet accepted, the text will displayed in blue (as opposed to the usual black). It is important to accept any text displayed in blue before attempting to perform any other functions.

1.3 Neural Network Overview

The descriptions of the networks along with the algorithms simulated in **smallBrain** were taken from James A. Freeman, *Neural Networks: Algorithms, Applications, and Programming Techniques*, Addison Wesley, 1991.

The basic unit of a neural network is a **node**. This is a simple processor which takes in several inputs and computes an output function. The inputs to a node are often the outputs of other nodes. All the inputs are weighted, so as to enhance the contributions of some nodes and ignore the contributions of others.

A group of nodes can be collected into a **layer**. All the nodes in a layer will often perform their computations in parallel, using the same inputs, although each node will weight the inputs differently (see Figure 1).

One or more layers can be incorporated into a **network**. The structure of nodes, layers, and networks varies among the different network topologies. Networks perform *training* and *classification* functions. In training, the network processes input data and "learns" the information provided by it. The actual learning takes place by altering the weights on the connections between nodes.

In classification, the network looks at input data, and provides the information that it has learned about it. A network must be trained before an attempt at classification is made. Often, the training must be monitored so that learning takes place correctly and the classification is accurate.

The learning performed by the network is actually a categorization of the vectors which make up the input data into several *classes*. Sometimes the class designation is annotated to the input vector, but this is not necessary for all types of networks. The classification phase is where the network is provided with a vector and the network responds with the class that the vector belongs to.

1.3.1 Back Propagation Network

This is a three layer feed-forward network (an input layer, one hidden layer, and an output layer). The input layer is just a buffer which holds the values of the input vector (it is not a true network layer) so it must be of the same size as the input vectors. The output layer uses a maximally sparse encoding,[1] so there must be at least as many output layer nodes as there are input vector classes.

The sample data which is supplied has input vectors which are 13 elements in length, and belong to 5 different classes. This requires that a network using this data must have 13 input layer nodes, and at least 5 output layer nodes. There can be any number of hidden layer nodes.

1.3.2 Hopfield Network

This network contains a single layer, which is fully connected. The sample data are 25 element vectors, which represent visual patterns (a 5×5 square grid). Therefore, any network using this data must specify this many nodes in the layer.

The input vectors in the sample data have class information which is ignored by the Hopfield Network, since it perform unsupervised learning (see below). This is provided because it is possible to use these data with a Back Propagation Network.

1.3.3 Self-Organizing Feature Map Network

This network contains a single one-dimensional layer. The sample data contains input vectors which are 51 elements long. Therefore, any network using these data must specify that the input vectors will be of this length. Any number of nodes can be in the layer, but it is recommended a large number relative to the number of classes of the input data.[2]

1.4 Hinton diagram

To demonstrate how learning occurs in a network, **smallBrain** makes use of a Hinton diagram to display the weights of a network layer. The Hinton diagram is a grid in which each row corresponds to the weight vector of a particular node (see Figure 1).

Instead of displaying the actual numbers (which would be too cluttered), the values are represented by colors. Positive values are represented by intensities of red, while negative values are represented by intensities of blue. If it is desired to see the numeric value of a particular grid element, then it can be displayed by depressing the left mouse button when the cursor is on that element.

[1] This means that the output will be such that one node returns the value 1 while the rest return 0.

[2] It is recommended to use 40 nodes for the sample data which is in 3 classes.

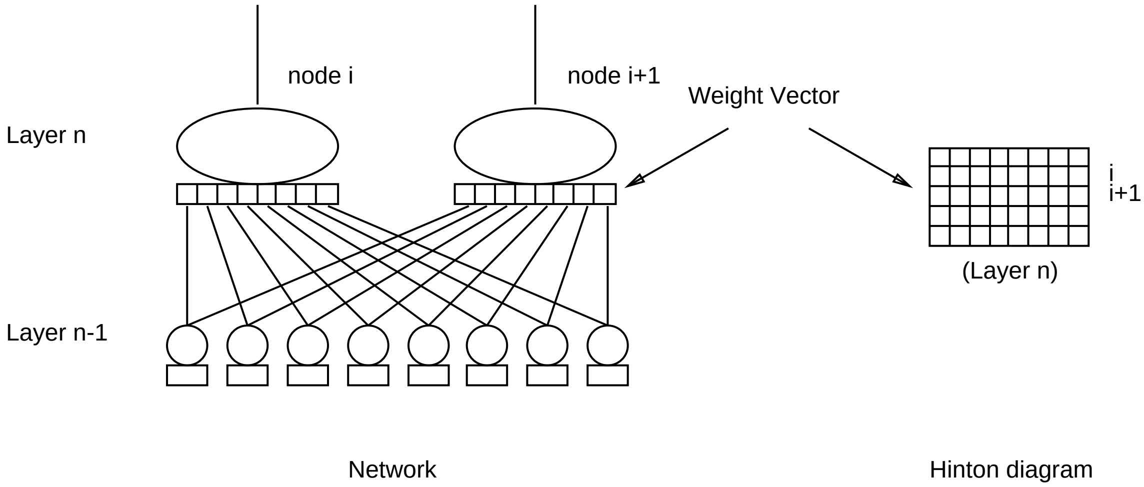

Figure 1: Correspondence of a Hinton Diagram

Figure 2: Hinton Diagram Grid Element Information

Figure 2 shows an example of how information about a grid element is displayed. The term "8@12" means this grid element represents node 8 connection 12. The other term is the actual numeric value of this element (since it is negative, it would have been blue in color).

There is a **threshold** value which represents the maximum absolute value that can be represented by red or blue. If the value of a grid element is greater than the positive value of the threshold, then it is displayed as magenta. It is displayed as cyan if it less than the negative value of the threshold. The threshold value can be displayed and changed by the operate menu.

With each iteration during the training phase, the Hinton diagram is updated so that observation of the learning characteristics is possible by following color changes as weight values change. It it not possible to display the the values of individual elements when training is taking place because user input is not polled when computation is occurring.

1.5 Input Vectors

The input to the networks is a collection of **input vectors**. They are used for both training and classification information. An example vector using the standard format is:

```
[2]<0,5,5,10,8,0,5,4,4,7,3,10,3>
```

where the `[2]` indicates that this vector belongs to the class 2. When specifying an input vector in an input file, it is necessary to include the class (which is always an integer) in square brackets, but the components of the vector need only be separated by whitespace (or commas or angle brackets). A carriage return denotes the end of a vector.

The class information must be a numeric value. The class information is necessary for networks which perform *supervised learning*. This means that the network is presented with the input vector and then told that this vector belongs to a particular class. The network then "learns" this information.

For networks which perform *unsupervised learning*, the class information is optional. (If class information exists, it is ignored.)

For those networks which are doing visual pattern recognition, the input vectors will be binary vectors with a square length.

2 Using smallBrain

There are two main components to **smallBrain**–the interface and the networks. The interface is the mechanism by which the networks get input and send output. Both the interface and the networks are manifested as windows. There is one **interface window** for the entire system and a **control window** for each network that is being simulated. More than one network of any type may created at one time (several control windows may be active).

The process of executing a simulation is the following:

1. open an interface window (if one is not already open).
2. open a network window.
3. load training data into the Input Buffer.
4. set the training parameters in the control window.
5. initiate the training phase by activating the "Train" button.
6. load classification data into the Input Buffer.
7. initiate the classification phase by activating the "Classify" button.

2.1 the Interface Window

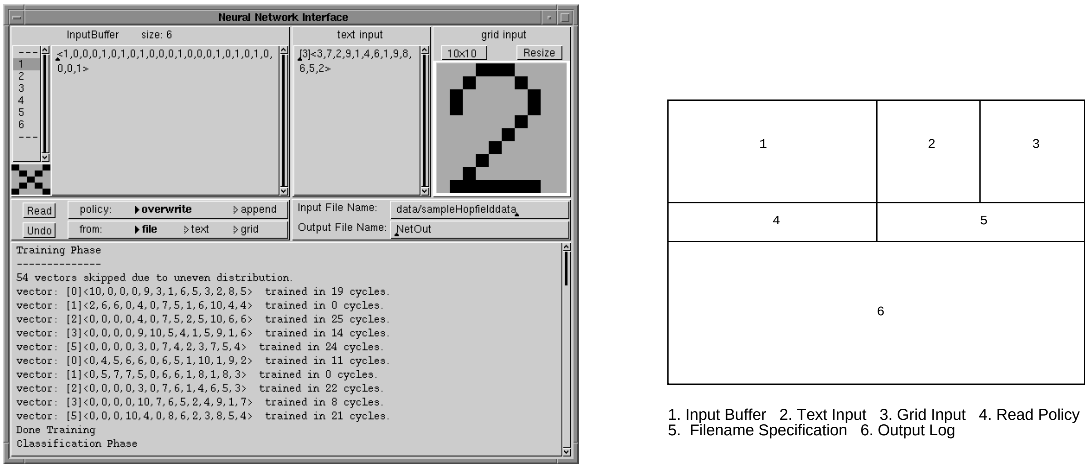

Figure 3: Interface Window and Layout Description

All input to and output from the networks are handled through the **interface** which is controlled via the **interface widow** (see Figure 3). If several interface windows have been opened, only the most recently opened window is valid. For this reason, it is suggested that all but the most recently opened interface window be closed.

The purpose of separating the input and output functions into a separate window is to allow multiple networks to use the same data without the need to reload the data multiple times.

2.1.1 Input Buffer

The Input Buffer is the main component of the interface. It is the area which actually holds the collection of input vectors used by the networks. It can be loaded with data contained in a file or with user-entered data, or a combination of the two.

A text display at the top of the Input Buffer display shows the size (number of input vectors) of the Input Buffer. The main part of the Input Buffer display makes use of the Smalltalk tool called an **Inspector**.[3] The Input Buffer Inspector has been modified to include a third display–a grid view. When the selected object is a binary vector of square length, then the vector will be translated to a picture and displayed here also.

The field display contains an indices of the input vectors. Selection a particular index will cause that input vector to appear in the display in the text display. The vector which is displayed can be editted by the user by typing over values or typing new values. The operate menu in the text display gives two choices:

- **accept**: incorporate any changes made to the vector into the Input Buffer
- **file out**: save the contents of the Input Buffer to a file. If this selection is made, then the user will be prompted for a file name to save the data under.

The operate menu in the field display give three options:

- **inspect**: open a new inspector on this particular vector. The field menu will correspond to the individual fields in a vector, and the text display will correspond to the actual value of that field. It is possible to edit the value in the text display (the change will need to be accepted).
- **remove**: remove this particular vector from the Input Buffer.
- **remove to ...**: remove a *range* of vectors from the Input Buffer. The current vector will be used as one extreme of the range, and the user will be prompted for the other extreme.

2.1.2 Text Input

This window allows the user to specify a vector to load into the Input Buffer (as opposed to loading them from a file). The vector is specified in the same format as if it were in an input file. After it has been typed, it needs to be accepted from the operate menu. Once it has been accepted, it will be redisplayed in the standard vector format. The accepted vector can be inserted into the Input Buffer by means of the "Read" button in the Read Policy display.

2.1.3 Grid Input

This window also allows the user to enter vectors into the Input Buffer, but these vectors represent visual patterns. Internally, the vector is one dimensional with binary values, but it is displayed as a two dimensional grid. Values of `1` are displayed as black squares, and values of `0` are displayed as light gray squares.

The dimensions of the grid (always square) is displayed at the top of the display. There is also a button which allows the grid to be resized.

When the cursor enters this display, it takes the shape of a bordered square. This is to indicate that it is possible to modify the picture via the mouse buttons. The left mouse button performs the "paint" function (any grid element that the cursor is on will become black if this button is pressed), and the middle mouse button performs the "erase" function (any grid element that the cursor is on will become light gray if this button is pressed).

The vector can be inserted into the Input Buffer by means of the "Read" button in the Read Policy display.

2.1.4 Read Policy

This display controls how values enter the Input Buffer. The user must select one of the following two policies:

[3] An inspector consists of two displays–a field display and a text display. The field display shows the indices of the components of the object being inspected. When the index of a particular object is selected (by clicking on that index with the left mouse button), the object is displayed in the text display. In both displays, there is an operate menu whose functions will depend on what operations are allowed on the object being inspected.

- **overwrite**: the new values will replace the current contents of the Input Buffer.
- **append**: the new values will be added to the end of the current Input Buffer contents.

The user must also select the location where the new values will come from (one of the following three):

- **file**: the new values will come from an input file.
- **text**: a new vector will come from the Text Input display.
- **grid**: a new vector will come from the Grid Input display.

When the "Read" button is selected, the actual insertion of values takes place. There is also an "Undo" button which will restore the Input Buffer to the state it was in before the last read was performed.

2.1.5 Filename Specification

This display is where input and output file names are specified. The input file is where input data are taken from when a read is performed in the Read Policy display. The output file is where the contents of the Output Log are archived.

2.1.6 Output Log

When a network produces output messages, this display is where they are written. All active networks will write to this same location. If text already exists here, any new text will be appended.

The operate menu gives three options:

- **clear**: the current contents of the output log are erased.
- **accept**: any modifications made to the text are incorporated.
- **file out**: save the contents of the output log to a file whose name was specified in the Filename Specification display.

2.2 the Control Window

Each network which has been created will have a **control window** which displays and manipulates aspects of that network. Each different type of network will have a different style of control window defined because each network behaves differently, but there are some common functions.

All networks will have buttons to "Train" and "Classify". The train button initiates the phase where the network uses the Input Buffer to train. The classify button initiates the phase where the network will try to classify the information based on the knowledge it acquired in the training phase.

All networks will also have one or more Hinton diagrams[4].

2.2.1 BPN control window

The back propagation control window has two Hinton diagrams–one for the output layer and one for the hidden layer (the input layer is not a true layer so it has no weights to display).

It has three numerical parameters which can be manipulated:

- **alpha**: the momentum parameter.
- **eta**: the learning rate parameter.
- **error limit**: the limit that determines when the network knows it has learned a particular vector. The error is measured by finding the sum of the squares of the difference between the actual output and the expected output. If this value is less than the error limit, then the networks stops training on this vector.

[4] There is one Hinton diagram per layer, so if the network has multiple layers, it will also have multiple Hinton diagrams.

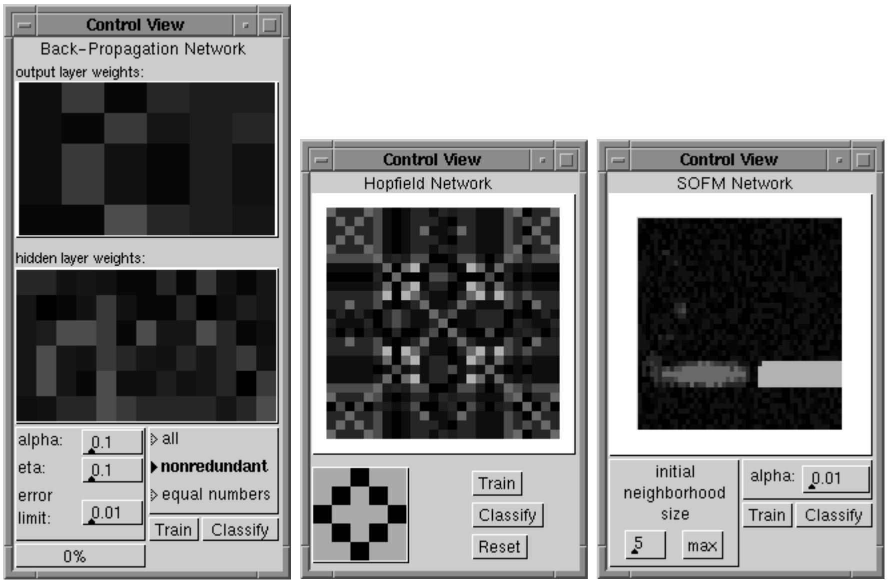

Figure 4: Control Windows for Back Propagation, Hopfield, and SOFM Networks.

There is also a selection of three different ways in which the network can train itself on the collection of input vectors:

- **all**: The network uses all the vectors in the collection.
- **nonredundant**: The network looks at the sequence of vectors given in the collection, and ignores those vectors which are in the same class as the one previous to them. This reduces biassing by not allowing the network to learn the same class with successive vectors.
- **even numbers**: The network looks for the class that has the least number of vectors representing it, and uses only that many vectors from the other classes (selecting them at random). The sequence presented becomes one vector from each class until the reduced collection is exhausted. This reduces biassing because not only is a class repeated by successive vectors, but each class is represented in equal strength.

Since the collection of vectors used for training could be very large, there is a display which shows what percentage of the collection has been processed.

During the training phase, the network produces verbose messages on the Output Log. It shows how many vectors from the Input Buffer were skipped. For each vector in the collection of training vectors, the vector itself is displayed along with the number cycles needed to train the network with this vector.

A rough measure of how well the network is trained can be taken by observing the number of cycles needed to train vectors. If the number of cycles is small (less than 10) when a vector of a different class from the previous vector is trained, then it is likely that the network is sufficiently trained. If it is noticed that the number of cycles remains large throughout the training set, then it is probably necessary to re-train the network.

When classifying vectors, it is assumed that the vectors have class information. Using this information, the network can measure if it has classified the vector correctly. Each vector in the classification collection is printed in the Output Log, along with the classification which the network produced (along with a flag which indicated if the classification was incorrect). When the entire collection is exhausted, the percentage of vectors which were correctly classified is printed.

2.2.2 Hopfield control window

Besides the "Train" and "Classify" buttons, there is also a "Reset" button. Unlike the BPN and SOFM networks where initial weight are small random values, the initial weights in a Hopfield network are all 0.

The reset button resets all the weights back to 0, or erases all the knowledge from the network.

No messages are printed to the Output Log during the training phase, but the number of cycles needed for stabilization is printed during the classification phase. The output of the network is displayed as a binary picture in the control window.

During classification, only the first vector in the Input Buffer is used. This is because only one output can conveniently be displayed in the control window at a time.

2.2.3 SOFM control window

There are two training parameters which can be modified:

- **alpha**: the learning rate parameter.
- **initial neighborhood size**: the inititial size of the neighborhood. This gets progressively decreased as training progresses. There is also a button which forces this value to the maximum possible value for the network topology.

The “output” of this network is the specification of a “winning” node for a particular vector. The network organizes it self into different clusters of nodes based on the classes of the training data. Vectors of the same class are expected to have winning nodes in the same cluster.

During the training phase, all the vectors are read, and the winning node for each vector is printed in the Output Log. An update of the weights is performed based on the neighborhood size, and then another training cycle is started with a reduced training size. The process continues until the neighborhood size reaches 1.

The winning node for each vector is produced for each cycle in the training phase (the neighborhood size for that cycle is also printed). During the classification phase, the winning nodes for all the vectors are printed.

3 Expanding smallBrain

This is a Smalltalk specific description of how to expand **smallBrain** *to include new networks.*

The advantage of the object oriented design is that most of the code can be reused. Unfortunately, the construction of a new Control Window takes a considerable amount of effort. When designing new networks, it is advisable to look at the `openControlViews:` class method of other network classes to see how a Control Window is constructed.

A vector class was defined[5] to be used in the numeric computations in the network. Most of the standard mathematical operations on vectors (dot product, normalization, etc.) have been defined. It might be necessary to add new operation if new networks are added.

3.1 Node Class

```
Object ()
        Model ('dependents')

                Node ('weights' 'net')
                        BPNNode ('output' 'lastChange')
                        HopfieldNode ()
                        SOFMNode ()
```

Figure 5: Node Class Heirarchy.

All nodes contain a weight vector and all nodes calculate an output function. These variable are provided by the superclass `Node`. The operations specific to a particular type of node should be defined in the subclass.

[5] subclass of `OrderedCollection`.

The class heirarchy of the nodes defined in **smallBrain** is shown in Figure 5. The BPN node does not use the `net` directly, but rather adds an new variable `output`.

3.2 Layer Class

```
Object ()
        Model ('dependents')

                Layer ('nodes' 'weightMatrix')
                        BPNLayer ('deltas' 'outputs')
                                BPNOutputLayer ()
                        HopfieldLayer ('outputMatrix')
                        SOFMLayer ('winner')
```

Figure 6: Layer Class Heirarchy.

All layers have a collection of nodes. The weights of these nodes can be displayed in a Hinton diagram, so the superclass `Layer` provides a variable to hold the underlying matrix.

The class heirarchy of the layers defined in **smallBrain** is shown in Figure 6. An SOFM layer is a competitive layer, so it defines a winning node. A Hopfield layer produces an output which can be displayed as a grid, so it defines `outputMatrix` to hold the values. A BPN layer (as it is defined in **smallBrain**) is actually a hidden layer. It defines values used for updating (`deltas`) and holds the output values so that they can be propagated to another layer. The subclass `BPNOutputLayer` only redefines how updating is done.

3.3 NeuralNetwork Class

```
Object ()
        Model ('dependents')

                NeuralNetwork ('interface')
                        BPNNetwork ('inputLayer' 'hiddenLayer' 'outputLayer' 'alpha' 'eta'
                                    'errorLimit' 'dictionary' 'trainingMode' 'percent')
                        HopfieldNetwork ('layer')
                        SOFMNetwork ('layer' 'initNeighborhood' 'alpha')
```

Figure 7: Network Class Heirarchy.

The network is the collection of all subcomponents. It is here where overall network behavior is defined. This will likely be the most complicated class because it's behavior is very broad. The class heirarchy of the networks defined in **smallBrain** is shown in Figure 7.

The superclass `NeuralNetwork` has as single instance variable called `interface` to hold the reference to the interface. The reference to the most recently created interface window is held in the *class* variable `interface`, and this reference is copied into the instance variable just before training or classification is initiated.[6]

Both the Hopfield network and the SOFM network have a single layer, but the BPN has multiple layers. The network class must also hold the values of various training parameters (`alpha`, `eta`, `initNeighborhood`, etc).

All the networks currently defined in **smallBrain** are composed of layers of the same type (an SOFM network consists of an SOFM layer), but this need not be so. It is possible to create **hybrid** networks which are composed of different types of layers.

[6] The reason for using a class variable is to have *all* instances of the networks use the *same* interface.

4 Acknowledgements

The original version **smallBrain** was written in collaboration with Karl Hess and Selvan Kulandaiswamy. Technical advice regarding neural networks was given by Sylvian Ray. Technical advice regarding Smalltalk was given by Ralph Johnson.