Neural Network Practical 2, Discrete Hopfield Neural Networks

Ioannis Tsiombikas

January 12, 2006

1 Workings of the Hopfield Net

The Discrete Hopfield Neural Network (hereafter refered to as DHNN), is an iterative neural network that works as an associative memory.

It's iterative characterization stems from the fact that unlike the typical Feed Forward Neural Network (FFNN), the process of recalling a stored pattern is iterative in nature. The (possibly corrupted) input pattern that we want to recognize is presented as the *previous output of the neurons*, and the network proceeds to iteratively modify its output continuously, until it converges to the original pattern.

1.1 Energy Function

As the pattern recall process goes on, we can calculate the amount of energy currently in the network at each instance, which is eventually minimized when it converges. This energy is calculated by the equation:

$$E = -\frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{N} s_j s_i w_{ij}$$

Where N is the number of neurons in the network, s is the current state of the network, and w is the *weight matrix*, a square NxN matrix with the weights of every neurons' connection to every other neuron [1].

[1] And since neurons cannot be connected to themselves, the main diagonal of this matrix, is 0

1.2 Storing Patterns

Patterns are stored through a very simple process, which is a modification of the Hebb rule. An elegant way to put this whole process mathematically is:

$$W'_{ij} = W_{ij} + (1 - \delta_{ij}) x_i^T x_j$$

Where δ is the *Kronecker delta*.

Or, in a simpler, more "algorithmic" notation, we can say that we calculate the *new* weights, we add the *previous* weights to the matrix produced by multiplying the transpose of our pattern with itself: $W' = W + x^T x$, and then zero-out the main diagonal.

1.3 Recalling Patterns

In order to recall a pattern, as I mentioned earlier, we feed our – possibly corrupted – pattern to the network, as a previous output/state, and we let it converge to the stored pattern over a number of iterations.

1.4 Network Memory

A Hopfield neural network cannot store an infinite ammount of patterns. The number of patterns we can store is a function of the complexity of the network, in terms of neurons.

Specifically this function is known to be:

$$p_{max} = \frac{N}{2 \ln N}$$

Where N is again the number of neurons in the network.

1.5 Examples

Now let's see some examples of storing and recalling patterns with a DHNN, by using the implementation I created for this practical.

1.5.1 DHNN Program Usage

To understand the examples, I will provide a short overview of the usage of my DHNN program, called (unimaginatively enough) *dhnn*. The program, expects the training patterns (i.e. the patterns we wish to store in its memory) in a file, and the (possibly corrupted) pattern we wish to recall, through

the standard input stream. By default it looks for a file named "training_set" in the current directory, but that can be overridden with the "-t" switch.

So in order to recall a pattern similar to (1 -1 -1 1) from the stored patterns which are in a file called "tset" we should use either:

```
$ echo '1 -1 -1 1' | ./dhnn -t tset
```

or,

```
$ ./dhnn -t tset <inp
```

in case we have our input in another file called "inp".

1.5.2 Example 1

For the first example let's consider an example from the slides. We store the pattern (1 -1 1 1) and we try to recall it with (-1 -1 1 1). The output of the program is the following:

```
$ echo '-1 -1 1 1' | ./dhnn -t ex1_tset
trying to defer the nnet size from the training set...
nnet size: 4
calculated neural network memory: 1.442695
training: 1 -1 1 1
input: -1 -1 1 1
Energy: -6.00
output: 1 -1 1 1
```

As we can see, the DHNN recalled the correct pattern from the noisy one, and finished with an energy of -6. [2]

The equivalent *Bidirectional Associative Map* (BAM) would have two layers of 4 neurons each, and the following weight matrix:

$$
\begin{aligned}
W' &= W + s^T t \\
W' &= \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 & 1 \end{pmatrix} \\
W' &= \begin{pmatrix} 1 & -1 & 1 & 1 \\ -1 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 \end{pmatrix}
\end{aligned}
$$

[2] While the program runs, it will update this energy indication continuously, and in case it runs to fast to see it, the "-d" option can be specified to insert a small delay between the iterations.

Which, apart from the main diagonal which is non-zero, it is exactly the same as the DHNN weight matrix.

In the rest of the examples, for brevity, I will refrain from presenting the whole output.

1.5.3 Example 2

For the second example, we store the following two patterns in the network: (-1 -1 1 1 -1 1 1 1 1 -1) and (1 1 1 -1 -1 -1 1 1 -1 1) Then we try to retrieve the second one, by providing the following input vector: (-1 1 1 -1 -1 -1 1 1 -1 -1)

```
$ echo '-1 1 1 -1 -1 -1 1 1 -1 -1' | ./dhnn -t ex2_tset
 ...
Energy: -42.00
output: 1 1 1 -1 -1 -1 1 1 -1 1
```

The equivalent BAM would have two layers of 10 neurons each, and the following weight matrix:

$$\begin{pmatrix} 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \\ 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \\ 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \\ 0 & 0 & -2 & 0 & 2 & 0 & -2 & -2 & 0 & 0 \\ 0 & 0 & -2 & 0 & 2 & 0 & -2 & -2 & 0 & 0 \\ 0 & 0 & -2 & 0 & 2 & 0 & -2 & -2 & 0 & 0 \\ 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \\ 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \\ 0 & 0 & -2 & 0 & 2 & 0 & -2 & -2 & 0 & 0 \\ 2 & 2 & 0 & -2 & 0 & -2 & 0 & 0 & -2 & 2 \end{pmatrix}$$

2 Implementation Details

For this practical, I created a small Discrere Hopfield Neural Network, implemented as a reusable ISO C library.

2.1 DHNN Library

The library presents the following very simple interface to the rest of the program:

```
/* create and destroy a dhnn */
void *dh_create(int nodes);
```

```
void dh_free(void *dh);

/* get count of nodes and memory capacity */
int dh_get_node_count(void *dh);
float dh_get_memory(void *dh);

/* train, or store a pattern into the network memory */
void dh_train(void *dh, int *pattern);

/* recall a pattern from memory (output goes to the
 * caller-allocated array, passed in the second parameter)
 */
void dh_process(void *dh, int *out, int *pattern);

/* this tells the library to delay (or not) in every recall
 * iteration, so that we can see the energy changing during
 * the process. By default it does not delay.
 */
void dh_set_slow_processing(int enable);
```

The *dh_create* and *dh_free* functions are responsible for constructing and deleting a DHNN. The DHNN itself is an opaque data type, manipulated internally by the library, not visible to the rest of the program (data hiding).

The *dh_train* function, accepts a pattern as an array of integers, and stores it to the network, by modifying the *weights* matrix as shown previously.

The *dh_process* function, accepts an input in its *pattern* argument, and runs the recall algorithm, as explained above, until the difference of the output of all the neurons from their previous states becomes 0.

2.2 Main Program

The main program (in `p2.c`) starts by parsing the command line and setting the appropriate variables. Then it goes on to call *defer_size* if we have not specified a neuron count with the "-s" argument, in order to find out the size of the network it needs to create.

Then, the neural network is created by calling the appropriate dhnn library function, and it is trained with each training pattern in the given file, in the *training* function.

Finally, the standard input is parsed with the help of *get_input* (which calls *parse_pattern* internally), and that input pattern is fed to the *dh_process* function, to recall the appropriate pattern from the DHNN memory.

The only other point worth mentioning, is that to maintain robustness with big pattern vectors, the *parse_pattern* function, stores every token as it is encountered in a linked list, the elements of which, are placed in an array allocated at the end, after the size is known.