Programming a Hopfield Neural Network for pattern recognition from scratch in C

My first foray into the world of ML, 6 years ago, as a young undergrad (2017)

Matthew Bain
December 13, 2017

Introduction

A Hopfield neural network is a recurrent artificial neural network [3], in which each unit, V_i, in the network is connected to every other unit in the network with a weight, T_{ij}. The biological equivalent of these artificial units can be thought of as neurons, and the connections between them as synapses [1]. A Hopfield network is capable of storing memories, which can be represented as patterns, p. Each unit, V_i within these patterns is described by a bipolar value, such as either +1 or -1, the biological equivalents of which can be thought of as either a neuron firing maximally or a neuron that is not firing, respectively.

Hebbian Learning

To store these patterns, a Hopfield model involves the formation of a weight matrix, T, which is calculated using equation (1):

(1) $T_{ij} = \sum_{i \neq j} V_i V_j$,

where $V_i V$ (i.e., the outer product, VV^T, of a given pattern), is summed over p. The formation of a weight matrix is equivalent to the 'learning' of patterns, in that information about the patterns is stored within the matrix. Hopfield describes these learned patterns as content-addressable, or associative, memories [1]. The contents of an associative memory can be retrieved from memory. In the context of a Hopfield network, this means that the system can take in noisy inputs and retrieve a similar or identical item from memory, in effect *recognizing* the unknown input based on incomplete information.

Updating the network

In order to retrieve stored memories based on noisy input patterns, the network 'updates' the state of one unit, or neuron, at a time, according to (2):

(2) $V_i = +1 \; if \; \sum_{i \neq j} T_{ij} V_j > 0, \; V_i = -1 \; if \; \sum_{i \neq j} T_{ij} V_j < 0$,

where V_i represents the state of neuron i after it is updated and 0 is the threshold firing rate of a single neuron [1]. If the sum of the product of the neuron's current state and its j associated synaptic weights exceeds threshold, its state, V_i, is changed to +1. If this new state is different than its previous state, the neuron, and by extension, the network, is said to have 'updated', potentially bringing the state of the network closer to the state of a pattern stored in memory.

Each updated neuron provides the input state for the next iteration of updating. This process of updating continues *asynchronously*, i.e., through the random selection of one neuron at a time, until some condition is met. At this point, whether or not the network has converged upon a stored memory state will depend on the quantity of neurons that have updated as well as several parameters of the stored patterns. According to Hopfield, each time a neuron's state is updated, the 'energy', E of the network is reduced.

Energy Function

Hopfield (1982) describes the energy of the system as in (3):

(3) $E = -\sum_j \sum_i (T_{ij} V_i V_j)/2$

A plot of the energy of the system over state space (i.e., the potential states of the network) gives several local minima corresponding to the stored memories. According to the *proof of decreasing energy* [2], an updating network is guaranteed to converge upon one of these local minima. In other words, an updated neural network is guaranteed to have lower energy than its previous state. This is illustrated by (4):

(4) $E^{new} - E^{old} = -(V_i{}^{new} - V_i{}^{old})(\sum_j T_{ij} V_j)$

There are two potential outcomes of updating a neuron:

a. If $T_{ij}V_j < 0$, then V_i is updated to a lower value (i.e., -1), i.e., $V_i{}^{new} - V_i{}^{old} < 0$
b. If $T_{ij}V_j > 0$, *then* V_i is updated to a higher value (i.e., +1) $V_i{}^{new} - V_i{}^{old} > 0$

When the signs of the terms of either scenario are plugged into (4), the outcome is the same (i.e., ΔE is negative).

As a result of this principle, a plot of network energy as a function of update number produces a monotonically decreasing function, where E is always converging upon some stable local minima. Because local minima correspond to the states of stored patterns, this process will theoretically continue until a pattern is perfectly retrieved from memory.

Spurious States

However, in reality it is not quite so straightforward. Similar memories tend to exist within similar regions of state space, and as a result, reinforce each other. This alters the shape of local minima [3] and can give rise to emergent collective properties [1] such as spontaneous states. Adding too many patterns (similar patterns, in particular), to memory, can therefore cause the system to overload and make it unreliable. Input states can sometimes converge upon local minima that either do not resemble the input states at all, or are some amalgamation of multiple patterns stored in memory, or are entirely spontaneous.

Implementation

1 Summary of Model

In this report I detail my implementation of Hopfield's model in C (see *hopfield.c; Makefile)*. My objective was to create a neural net capable of classifying (or recognizing) randomly generated (or otherwise noisy) input patterns into categories defined by exemplar patterns that they most closely resemble.

1.1 Steps

1. Create network (using struct definition of net)
2. Convert inputted patterns to bipolar (+1; -1) format
3. Calculate weight matrix
4. Recognize the pattern through asynchronous updating of the state of the network (simulate the net until it converges upon local minimum)

1.2 Global Parameters

p = quantity of inputted memory 'exemplars'
u = quantity of inputted 'unknown' patterns
X = width of patterns
Y = height of patterns
N = number of units in patterns / neurons in neural net

threshold = firing threshold of individual neurons (assumed to be 0, as per Hopfield (1982))
minUpdate = minimum number of times *N* randomly selected neurons given chance to update state before output pattern produced (local minimum selected)

Note: input patterns, stored in *pattern1* and *pattern2*, are also defined globally, along with the definition of a struct for creating the network (i.e., the *output* and *weight* matrices).

2 Discussion

2.2 Storing the patterns

To implement the model, I started with a simple 20-neuron (*N* = 20) network. I chose to input the patterns in two separate 3D character arrays: (1) *pattern1[p][Y][X]* stores patterns that are learned by the neural network using the Hebbian learning rule. These patterns serve as content-addressable memories (or *exemplars*). (2) *pattern2* stores random/noisy patterns of the same dimensions. Using the memory exemplars in *pattern1* the net attempts to recognize/classify patterns in *pattern 2.* Individual units (or pixels) in these patterns take on one of two potential states, represented by either a "O" or " " (a blank space).

Initializing the system

To initialize the system, I first defined a struct containing (1) a pointer to an array of integers, *output*, and (2) a pointer to an array of pointers, *weight* (i.e., a the basis for a 2D weight matrix). I then allocated memory for these matrices using *malloc*. To store the inputted patterns in the neural net, I first declared two empty 2D integer arrays, *exemplar[u][N]* and *unknown[u][N]* . I then converted the units of the inputted patterns to bipolar values (following the rule "O" = +1 and " " = -1) and stored them in these 2D arrays. To do this, I reduced the two dimensions, *[i]* and *[j]*, to one dimension, *[N]*, using the expression: *i*X+j*.

Updating the network

I decided to allow updating of the network to continue as long as the following condition holds true:

$$it - itofLastUpdate < N*minUpdate,$$

where *it* counts the number of times the network has called *updateNet*, and *itofLastUpdate* keeps track of the value of *it* corresponding to the most recent function call of *updateNet* which returned a value of 1 (indicating that the state of the neuron did, in fact, change).

Based on this definition, *asynCor* continues to call *updateNet* until *N* neurons are passed *minUpdate* times since the last iteration in which the network was successfully updated. If *updateNet* returns null over *N*minUpdate* iterations, *asynCor* breaks out of the while loop. This indicates that the network energy has converged upon a local minimum and, ideally, that the pattern has been accurately recognized.

Program output; Determining minUpdate

1. Output of model for $p = 1$; $u = 1$; $N = 20$; *minUpdate* = 0.01:

```
INPUT:

→OO    O O
→O  OO  OOO
------------------------------
OUTPUT:

→OO O  O O
→O  OO  OOO
------------------------------
Error[compared to exemplar 1] = 9.000000
------------------------------
```

Error here refers to cumulative error. That is, a value of 1 is added to error for every unit that is different between the output and a given exemplar. In this case, *Error* = 9 because a *minUpdate* of 0.01 does not allow the network to update the states of any neurons even once (0.01*N = 0.2, which is always < it – itofLastUpdate). This would indicate that the input has 9 elements that do not match exemplar 1, but in reality it has 10 elements that do not match exemplar 1. This observation can be explained by the use of a 'do while loop' to control function calls of *updateNet*, as

this ensures that *updateNet* is called at least once. In order to allow the model to find the memorized state, the network needs to update at least 10 times to correct each error, so a *minUpdate* value of at least ~0.5 (20*0.5 = 10) is required.

2. Output of model for $p = 1$; $u = 1$; $N = 20$; *minUpdate* = 1.4:

```
INPUT:

→OO    O O
→O  OO  OOO
------------------------------
OUTPUT:

→OO OO OO O
→O OOOOO OO
------------------------------
Error[compared to exemplar 1] = 1.000000
The output is similar to exemplar 1.
------------------------------
```

Increasing *minUpdate* to 1.4 reduces *Error* from 9 to 1.

3. Output of model for $p = 1$; $u = 1$; $N = 20$; *minUpdate* = 1.5:

```
INPUT:

→OO    O O
→O  OO  OOO
------------------------------
OUTPUT:

→OO OO OO O
→O OO OO OO
------------------------------
Error[compared to exemplar 1] = 0.000000
The output is identical to exemplar 1.
------------------------------
```

Cumulative error = 0. Therefore, the model finds the memorized state (exemplar 1) when *minUpdate* = 1.5.

2.3 Selected output 1

Output of model for p = 2; u = 1 (random); N = 100; *minUpdate* = 10:

```
INPUT:

→O    O O O
→O O   OOO
→ O  O  O O
→  O O O O
→   O   O
→ O O OO  O
→O   O  O
→ OOO  OO O
→OO O O   O
→    O   OO
------------------------------
OUTPUT:

→
→
→    OOO
→  O     O
→ O       O
→ O O O O O
→ O
→  O
→    O O O
→
------------------------------
Error[compared to exemplar 1] = 49.000000
Error[compared to exemplar 2] = 0.000000
The output is identical to exemplar 2.
------------------------------
```

Discussion of observations

Starting from a random configuration of neurons (u), the network does find one of the memorized patterns (it finds the 'e'). This pattern is related to the inputted pattern in that a number of its units match those of the input pattern (more so than exemplar 2). This is shown in the following test, in which a small value (0.01) of *minUpdate* was used so that the state of only two neurons changed. The results of this test show that the random input pattern is more similar to exemplar 2 (the outputted 'e' pattern) than exemplar 1, which suggests that its energy is more likely to converge upon the local minimum corresponding to exemplar 2.

```
INPUT:

→O    O O O
→O O   OOO
→ O  O  O O
→  O O O O
→   O   O
→ O O OO  O
→O   O  O
→ OOO  OO O
→OO O O   O
→    O   OO
------------------------------
OUTPUT:

→O    O O O
→O O   OOO
→ O  O  O O
→  O O O O
→   O   O
→ O O OO  O
→O   O  O
→ OOO  O  O
→OO   OO  O
→    O   OO
------------------------------
Error[compared to exemplar 1] = 48.000000
Error[compared to exemplar 2] = 39.000000
------------------------------
```

Associative memory

Associative memory refers to memory that is content-addressable. A memorized pattern is called an associative memory because its contents can be retrieved from memory to serve as a reference for the classification of unknown input patterns.

Finding local minimum

In accordance with the discussion above (see 2.2) concerning *minUpdate*, I decided to stop having the system evolve when *updateNet* returns null 10**N* consecutive times (i.e., *minUpdate* = 10). This provides plenty of opportunity for the network to correct and converge upon a local minimum energy state.

Error

My measure of "error", *e,* is defined as follows:

$$\sum_i e = | (network->output[i] - exemplar[n][i]) | /2$$

Each unit that is different between *network->output[i]* (set to the current state of the network) and *exemplar[n][i]* leads to an increase in *e* of 1 (this makes *e* a measure of *cumulative* error, to be precise) [4]. The validity of this measure can be illustrated by laying out all possible combinations of *exemplar[n][i]* and *input[i]* states and their corresponding

differences:

- **Case 1**: same state
  - 1 – 1 = 0
  - -1 – (-1) = 0
- **Case 2**: different state
  - 1 – (-1) = 2
  - -1 -1 = -2
    - Because we want these two cases to both produce an increase in *e* of 1, we must divide the differences by 2 and apply the absolute value function (normalize their outputs).

The located pattern, according to the output of the program, is identical to exemplar 2. This is shown by an *e* of 0.

2.4 Selected output 2

1. Output of model for $p = 3$; $u = 1$; $N = 100$; *minUpdate* = 10:

```
INPUT:

→O    O O O
→O O   OOO
→ O  O  O O
→  O O O O
→   O   O
→ O O OO  O
→O   O  O
→ OOO  OO O
→OO O O   O
→    O   OO
------------------------------
OUTPUT:

→
→
→    OOO
→  O     O
→ O       O
→ O O O O O
→ O
→  O
→    O O O
→
------------------------------
Error[compared to exemplar 1] = 49.000000
Error[compared to exemplar 2] = 0.000000
The output is identical to exemplar 2.
Error[compared to exemplar 3] = 45.000000
------------------------------
```

The network successfully retrieves a memory exemplar.

2. Output of model for $p = 4$; $u = 1$; $N = 100$; *minUpdate* = 10:

```
29 ------------------------------
30 Error[compared to exemplar 1] = 49.000000
31 Error[compared to exemplar 2] = 0.000000
32 The output is identical to exemplar 2.
33 Error[compared to exemplar 3] = 45.000000
34 Error[compared to exemplar 4] = 53.000000
35 ------------------------------
```

The network successfully retrieves a memory exemplar (no longer displaying patterns to conserve space).

3. Output of model for $p = 5$; $u = 1$; $N = 100$; *minUpdate* = 10:

```
29 ------------------------------
30 Error[compared to exemplar 1] = 49.000000
31 Error[compared to exemplar 2] = 0.000000
32 The output is identical to exemplar 2.
33 Error[compared to exemplar 3] = 45.000000
34 Error[compared to exemplar 4] = 53.000000
35 Error[compared to exemplar 5] = 39.000000
36 ------------------------------
```

The network successfully retrieves a memory exemplar.

4. Output of model for $p = 6$; $u = 1$; $N = 100$; *minUpdate* = 10:

```
29 ------------------------------
30 Error[compared to exemplar 1] = 48.000000
31 Error[compared to exemplar 2] = 7.000000
32 The output is similar to exemplar 2.
33 Error[compared to exemplar 3] = 38.000000
34 Error[compared to exemplar 4] = 60.000000
35 Error[compared to exemplar 5] = 32.000000
36 Error[compared to exemplar 6] = 34.000000
37 ------------------------------
```

The network no longer successfully retrieves a memory exemplar.

Table 1*:* Average Error as a function of increasing p
(u = 1; N = 100; minUpdate = 10)

p	Average Cumulative Error ($\sum_p e / p$)
2	24.5
3	31.33
4	36.75
5	37.2
6	36.5
7	32.43
8	32.63
9	31.33
10	31.23
11	30.36
12	29.67
13	30.08

14	29.42
15	29.62
16	29.63
17	29.12
18	28.94
19	28.47
20	29.90

5. Output of model for p = 20; u = 1; N = 100; *minUpdate* = 10:

```
16 ----------------------------
17 OUTPUT:
18
19 ⟶
20 ⟶          O
21 ⟶
22 ⟶
23 ⟶      O
24 ⟶     OOOO
25 ⟶
26 ⟶
27 ⟶
28 ⟶
29 ----------------------------
```

Increasing p

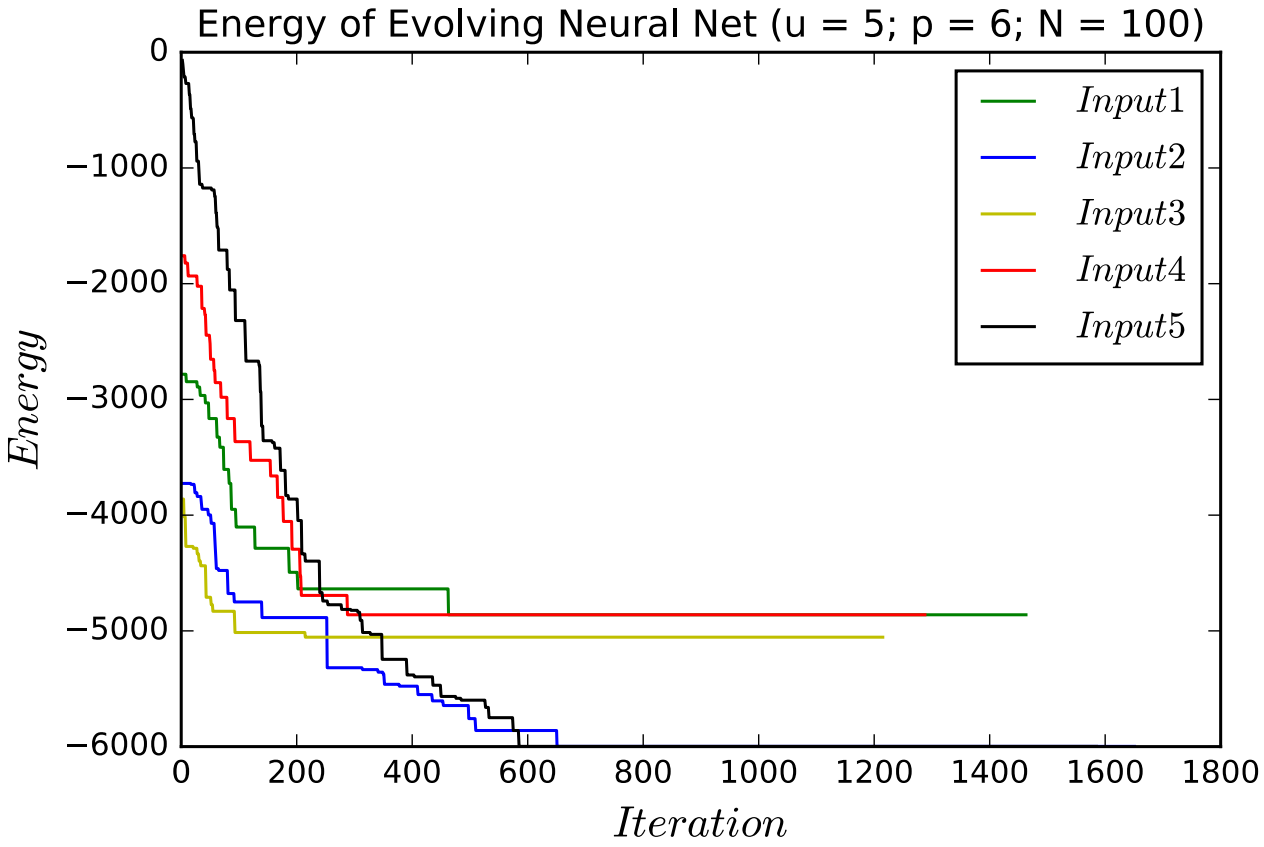
As p is increased, the errors become more widely distributed, but the average tends to remain relatively stable (see Table 1). This can be explained by the central limit theorem, which states that higher sample sizes provide a more accurate representation of the mean by capturing the true variance of the population from which samples are drawn.

Discussion of $p_{max,}$

When p is increased significantly beyond $p_{max,}$ = 13.8 (i.e., 0.138*100), the output all tends to look the same (i.e., very sparse and formless – see output (5) of program, above), which suggests the emergence of spurious states due to excessive reinforcement of certain configurations (overloading the network). As a result, as new exemplars are added, some very small errors arise, in the range of 5-10, likely due to these exemplars happening to look quite similar to these spurious states. In this case, beyond 10 exemplars only one exemplar (the 'house') was successfully retrieved, suggesting that its structure is quite orthogonal to the other exemplars (i.e., it occupies distinct areas of state space).

Based on these observations, this estimate of $p_{max,}$ seems reasonable.

2.5 Plot of model performance



Corresponding Output of Program

Input1:

```
INPUT:

	OO  O O OO
	 O    O
	OO    OOOO
	         O
	O  OO  O
	O  OOOO  O
	
	OOO    OOO
	 O    O
	OO O  O OO
------------------------------
OUTPUT:

	OO O  O OO
	 O    O
	OOOOOOOOOO
	
	O  OOOO  O
	O  OOOO  O
	
	OOOOOOOOOO
	 O    O
	OO O  O OO
------------------------------
The output is identical to exemplar 1.
------------------------------
```

Input2:

```
INPUT:

	      O
	   O    O
	    O
	  O     O
	     O   O
	 O O   O O
	 O
	  O
	      O O
	  OO
------------------------------
OUTPUT:

	
	 OO    OO
	O
	O
	
	 O OOO O
	         O
	         O
	        O
	   OOOO
------------------------------
The output is similar to exemplar 5.
The output is similar to exemplar 6.
------------------------------
```

Input3:

```
INPUT:

	OO O  OOOO
	   OO
	   OO
	    O
	   OO
	    O
	   O
	    O
	   OO   O
	OOOO  OOOO
------------------------------
OUTPUT:

	OO OOOO OO
	   OO
	   OO

	   OO
	   OO

	   OO
	   O
	OOOOOOOOOO
------------------------------
The output is similar to exemplar 3.
------------------------------
```

Input4:

```
INPUT:

	OO  O
	   OOOO O
	OOOO OOOOO
	 OOOOOO
	O         O
	  OO  OO
	O O       O
	O       O O
	    O     O
	OO  OO O O
------------------------------
OUTPUT:

	OO  O
	OO OOOO
	OOOOOOOO
	OOOOOOOOOO
	O         O
	O OO      O
	O OO      O
	O       OOO
	O       O O
	OOOOOO O O
------------------------------
The output is identical to exemplar 4.
------------------------------
```

Input5:

```
INPUT:

→O    O O O
→O O   OOO
→ O  O  O O
→  O O O O
→   O   O
→ O O OO  O
→O   O  O
→ OOO  OO O
→OO O O   O
→    O   OO
------------------------------
OUTPUT:

→
→ OO    OO
→O
→O
→
→ O OOO O
→         O
→         O
→        O
→   OOOO
------------------------------
The output is similar to exemplar 5.
The output is similar to exemplar 6.
------------------------------
```

Discussion of plot

The plot of the energy, *E,* of the evolving network as a function of iteration, *it*, is a monotonically decreasing function. This is true for each input pattern (starting configuration), *u*, represented by the 5 different lines. Each decrement in the trajectory of a line corresponds with a successful change of state for a single neuron (see the discussion about 'proof of decreasing energy' in introduction).

Comparing this plot to the output of the program reveals some interesting patterns. Firstly, Input1 and Input4 converge on the same energy but different patterns. They are also the only patterns whose outputs are identical to a memory exemplar. This suggests that the local minima corresponding to the energies of these exemplars exist at equal 'depths' but different regions of state space. This makes sense for stored patterns that are not reinforced excessively by other patterns.

In contrast, Input2 and Input5 converge on the same energy as well as the same pattern. The energy on which they converge is lower than that of Input1/Input4, and their output patterns are not identical to any memory exemplar. This would suggest that the converged-

upon state is spurious. The corresponding local minimum is likely quite heavily reinforced by separate components of several patterns (e.g., the bottom rows of the patterns resembling an 'S', 'I' and 'house'). This reinforcement would lead to the overlapping local minima merging and summating, producing a wide, deep valley in the energy function.

Finally, Input3 has a final energy that is in between that of Input1/Input4 and Input2/ Input5. This makes sense given that its output pattern is not identical to any memory exemplars, but is quite similar to the 'I'. This would suggest that its final state is not spurious, but has been slightly reinforced by nearby patterns.

These observations can also be explained by the error values of each output pattern. The errors (not shown here) indicate that Input1 is *identical* to exemplar 1, Input4 is *identical* to exemplar 4, Input2/Input 5 are *similar to both* exemplar 5 and 6 (suggesting that these exemplars reinforced each other and merged in state space), and Input 3 is *similar* to exemplar 3.

3 Programs

3.1 hopfield.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// A program for generating and simulating a Hopfield neural network that
recognizes simple patterns.
// To compile: 'make hopfield'

// Define macros for input pattern parameters
#define p sizeof(pattern1)/sizeof(pattern1[0]) // p = number of input
patterns for learning (memory exemplars); defined in terms of 'pattern1'
#define u sizeof(pattern2)/sizeof(pattern2[0]) // u = number of input
patterns for recognition (unknown patterns)

// Case 1 parameters (N = 20)
//#define X 10 // width of patterns
//#define Y 2 // initial height of patterns

// Case 2 parameters (N = 100)
#define X 10 // width of patterns
#define Y 10 // height of patterns

#define N (X*Y) // number of units (neurons) within patterns

// Network parameters
#define threshold 0 // firing threshold of individual neurons (assumed to
= 0)
#define minUpdate 10 // minimum number of times N random neurons given
chance to update

// Define struct for creating new network
typedef struct {
    int* output;
    int** weight;
} net;

// Pointers to output files
FILE *fp1; // stores output of program
FILE *fp2; // stores energies of evolving network
FILE *fp3; // stores instructions for plotting data

// Input patterns (exemplars) for learning; stored in 3D character array
'pattern1'
char pattern1[][Y][X] = {
/*{"OO OO OO O",
 "O OO OO OO"}*/

{"OO O  O OO",
 "  O    O  ",
 "OOOOOOOOOO",
 "          ",
```

```
 "O  OOOO  O",
 "O  OOOO  O",
 "          ",
 "OOOOOOOOOO",
 "  O    O  ",
 "OO O  O OO"},

{"          ",
 "          ",
 "    OOO   ",
 "  O     O ",
 " O       O",
 " O O O O O",
 " O        ",
 "  O       ",
 "    O O O ",
 "          "},

{"OOOOOOOOOO",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "    OO    ",
 "OOOOOOOOOO"},

{"OO  O     ",
 "OO OOOO   ",
 "OOOOOOOO  ",
 "OOOOOOOOOO",
 "O        O",
 "O OO     O",
 "O OO     O",
 "O      OOO",
 "O      O O",
 "OOOOOO O O"},

{"          ",
 " OOOOOOOO ",
 "O         ",
 "O         ",
 "O         ",
 " OOOOOOOO ",
 "         O",
 "         O",
 "         O",
 " OOOOOOOO "},

{"   OOOO   ",
 " OO    OO ",
 "O        O",
```

```
 "O         O",
 "    OO     ",
 "    OO     ",
 "O         O",
 "O         O",
 " OO     OO ",
 "   OOOO   "},


/*{"O O O     ",
 "     O O O",
 "    O     ",
 "OOO O  OOO",
 " O  O   O ",
 "  O O O   ",
 " O     O  ",
 "   O O    ",
 "    O     ",
 " O  O  O  "},


{"  OOOOOO  ",
 " O      O ",
 "O        O",
 "O        O",
 "OOOOOOOOOO",
 "O        O",
 "O        O",
 "O        O",
 "O        O",
 "O        O"},

{"        O  ",
 "        O  ",
 "        O  ",
 "    OO O  ",
 "    OO O  ",
 "   OOOO   ",
 "   OOOO   ",
 "    OO    ",
 "   OOOO   ",
 " OOOOOOOO "},

{"O        O",
 " O      O ",
 "  O    O  ",
 "   O  O   ",
 "   O  O   ",
 "   O  O   ",
 "   O  O   ",
 "  O    O  ",
 " O      O ",
 "O        O"},
```

```
{"          ",
 "          ",
 "          ",
 "    OO    ",
 "  OOOOOO  ",
 "OOOOOOOOOO",
 "  OOOOOO  ",
 "    OO    ",
 "          ",
 "          "},

{"  O    O  ",
 "  O    O  ",
 "  O    O  ",
 "  O    O  ",
 "  OOOOOO  ",
 "  OOOOOO  ",
 "  O    O  ",
 "  O    O  ",
 "  O    O  ",
 "  O    O  "},

{"          ",
 "OOO OO OOO",
 "   O  O   ",
 "    OO    ",
 "OO O  O OO",
 "    OO    ",
 "    OO    ",
 "OO O  O OO",
 "    OO    ",
 "   O  O   "},

{"     O    ",
 "     O    ",
 "OOO O O OO",
 "    OOO   ",
 "    OOO   ",
 "    OOO   ",
 "   O  O   ",
 "  O    O  ",
 " O      O ",
 "O        O"},

{"          ",
 "OOOOOOOOOO",
 "O        O",
 " O      O ",
 "  O    O  ",
 "  O    O  ",
 "   O  O   ",
 "    OO    ",
 "   OOOO   ",
 "          "},
```

```
{"          ",
 "     O    ",
 "    OOO   ",
 "OOOO   OOO",
 "  OO   OO ",
 "   OOOOO  ",
 "  OOO OOO ",
 " OO    OO ",
 "O        O",
 "          "},

{"O O   O O ",
 "    O     ",
 " O O O O  ",
 "          ",
 "  O O O   ",
 "          ",
 "   O O    ",
 "          ",
 "    O     ",
 "    O     "},

{"          ",
 "O OO  OO O",
 " O      O ",
 " O      O ",
 " O      O ",
 " OOOOOOOO ",
 " O      O ",
 "  O    o  ",
 "  O    O  ",
 "  OOOOOO  "},

{"O        O",
 "O      O  ",
 "O    O    ",
 "O  O      ",
 "O O       ",
 "O  O      ",
 "O    O    ",
 "O      O  ",
 "O        O",
 "          "},

{"OOOO  OOOO",
 "OOOO  OOOO",
 "OOO    OOO",
 "OO      OO",
 "          ",
 "          ",
 "OO      OO",
 "OOO    OOO",
```

```
 "OOOO  OOOO",
 "OOOOO OOOO"}*/
};

// Input patterns (unknowns) for recognition; stored in 'pattern2'
char pattern2[][Y][X] = {
/*{"OO    O O ",
 "O  OO  OOO"}*/

// resembles exemplar 1
{"OO  O O OO",
 "  O    O  ",
 "OO    OOOO",
 "         O",
 "O  OO  O  ",
 "O  OOOO  O",
 "          ",
 "OOO    OOO",
 "  O    O  ",
 "OO O  O OO"},

// resembles exemplar 2
{"      O   ",
 "   O    O ",
 "    O     ",
 "  O     O ",
 "     O   O",
 " O O   O O",
 " O        ",
 "  O       ",
 "      O O ",
 "  OO      "},

 // resembles exemplar 3
{"OO O  OOOO",
 "    OO    ",
 "    OO    ",
 "     O    ",
 "    OO    ",
 "     O    ",
 "    O     ",
 "     O    ",
 "    OO   O",
 "OOOO  OOOO"},

// resembles exemplar 4
{"OO  O     ",
 "   OOOO O ",
 "OOOO OOOOO",
 " OOOOOO   ",
 "O        O",
 "  OO  OO  ",
 "O O      O",
 "O      O O",
```

```
 "    O    O",
 "OO  OO O O"},

// resembles exemplar 5
/*{"      O   ",
 "  OOOOO   ",
 "O       OO",
 "O     OO  ",
 "O         ",
 " OO   OOO ",
 "    O    O",
 "   OO    O",
 "         O",
 " OOO OOOO "},

// resembles exemplar 6
{"    OOO   ",
 " OO     O ",
 "O    OO  O",
 "O        O",
 "    O   O ",
 " O  O     ",
 "         O",
 "O         ",
 " OO    OO ",
 "     OO   "},*/

// random configuration
{"O    O O O",
 "O O   OOO ",
 " O  O  O O",
 "  O O O O ",
 "   O   O  ",
 " O O OO  O",
 "O   O  O  ",
 " OOO  OO O",
 "OO O O   O",
 "    O   OO"}
};

// Function for initialization of network.
void initializeNet(net* network)
{
    int i; // for loop counter

    // allocate memory for output matrix
    network->output = (int*)malloc(sizeof(int)*N);
    // allocate array of pointers to pointers (for 2D weight matrix)
    network->weight = (int**)malloc(sizeof(int*)*N);

    // allocate memory for N arrays of pointers to int objects for
'weight'
    for (i=0; i<N; i++) {
        network->weight[i] = (int*)malloc(sizeof(int)*N);
```

```
    }

    // open file for writing energy (NRG) calculations
    fp2 = fopen("hopEnergy.dat", "w");
    if(fp2==NULL) { // if file is not successfully opened
        printf("failed to open file: hopEnergy.dat\n"); // error message
        exit(1);
    }

    // open file for writing instructions for plotting NRG calculations
    fp3 = fopen("hopPlotInstructions.txt", "w");
    if(fp3==NULL) { // if file is not successfully opened
        printf("failed to open file: hopPlotInstructions.txt\n");
        exit(1);
    }
}

int unknown[u][N]; // 2D matrix to store unknown patterns
int exemplar[p][N]; // 2D matrix to store exemplar patterns

// Function for converting inputted patterns to bipolar (-1 or +1) and
converting 3D input matrices to 2D matrices.
void bipolarTransform(net* network)
{
    int n,i,j; // declare for-loop counters

    // 'unknown' matrix conversion
    for(n=0; n<u; n++) {
        for(i=0; i<Y; i++) {
            for(j=0; j<X; j++) {
                if(pattern2[n][i][j] == 'O') {
                    unknown[n][i*X+j] = 1; //
                } else unknown[n][i*X+j] = -1;
            }
        }
    }
    // 'exemplar' matrix conversion
    for(n=0; n<p; n++) {
        for(i=0; i<Y; i++) {
            for(j=0; j<X; j++) {
                if(pattern1[n][i][j] == 'O') {
                    exemplar[n][i*X+j] = 1;
                } else exemplar[n][i*X+j] = -1;
            }
        }
    }
}

// Function for creating weight matrix (learning patterns / storing
memories).
void calculateWeights(net* network)
{
    int i,j,n; // declare for loop counters
    int weight; // value of weight matrix at ij, summed over p exemplars
```

```
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            weight = 0; // set initial state of weight
            if(i!=j) { // neurons are not connected to themselves
                // sum weight matrices of exemplars
                for(n=0; n<p; n++) {
                    // find element ij of weight matrix using outer
product
                    weight += exemplar[n][i] * exemplar[n][j];
                }
            }
            // fill weight matrix
            network->weight[i][j] = weight;
        }
    }
}

// Function for calculating cumulative or average error of output against
each exemplar stored in memory.
float error(net* network, int* output)
{
    float e; // e = error
    int i, n;

    for(n=0; n<p; n++) { // count through each exemplar
        e = 0;
        // add 1 to error for every unit that is different between output
and exemplar
        for(i=0; i<N; i++) {
            e += abs(network->output[i] - exemplar[n][i])/2;
        }
        //e = e/N; // divide e by N for average error

        // print number of neurons in incorrect state (statement accurate
for cumulative or average error)
        fprintf(fp1, "Error[compared to exemplar %d] = %f\n", n+1, e);

        // more compact print to see if output is identical to any
exemplar (only accurate for cumulative error)
        if(e<1) { // float comparison so do not check for equality
            fprintf(fp1, "The output is identical to exemplar %d.\n",
n+1);
        } else if (e<N/5) {
            fprintf(fp1, "The output is similar to exemplar %d.\n", n+1);
        }
    }
    fprintf(fp1, "-----------------------------\n");
}

// Function for calculating energy (as defined by Hopfield (1982)) of the
network.
float Energy(net* network)
{
```

```
    int i, j;
    float E; // E = NRG
    int sum;

    sum = 0; // set initial state of sum
    // calculate sum of (connection weights to neuron ij)*(unit ij of
output outer product) over N neurons
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            if(j!=i) {
                sum += network->weight[i][j] * network->output[i] *
network->output[j];
            }
        }
    }
    E = (-1./2.)*sum; // divide sum by -0.5 to complete NRG calculation
    return E;
}

// Function for printing unknown inputs and recognized outputs.
void printNet(net* network)
{
    int i,j;

    // convert bipolar back to charcters (1 = 'O'; -1 = " ") and print
    for(i=0; i<Y; i++) { // iterate row
        fprintf(fp1, "\t");
        for(j=0; j<X; j++) { // print out entire row
            if(network->output[i*X+j] == 1) {
                fprintf(fp1, "O");
            } else if(network->output[i*X+j] == -1) {
                fprintf(fp1, " ");
            }
        }
        fprintf(fp1, "\n");
    }
    fprintf(fp1, "-----------------------------\n");
}

// Function for setting initial state of network->output to contents of
'unknown' array. Output vector provides input patterns of program.
void setInput(net* network, int* unknown)
{
    int i;

    // fill 'network->output' w/ contents of 'unknown'
    for(i=0; i<N; i++) {
        network->output[i] = unknown[i];
    }
    fprintf(fp1, "INPUT:\n\n");
    printNet(network); // call 'printNet' to print input patterns
}
```

```
// Function for setting 'output' array to final state of network->output.
Output vector provides output patterns of program.
void getOutput(net* network, int* output)
{
    int i;

    for(i=0; i<N; i++) {
        output[i] = network->output[i];
    }
    fprintf(fp1, "OUTPUT:\n\n");
    printNet(network); // call 'printNet' to print output patterns

    // call 'error' to calculate and print errors between corresponding
inputs and outputs
    error(network, output);
}

// Function for updating state of individual neuron (i) of network.
int updateNet(net* network, int i)
{
    int  j;
    int  sum, out;
    int updated;

    sum = 0;
    updated = 0; // set initial state of 'updated' to 0
    // calculate sum of connection weights * current state of network over
N for neuron i
    for(j=0; j<N; j++) {
        sum += network->weight[i][j] * network->output[j];
    }

    // if sum does not exceed firing threshold
    if(sum < threshold) out = -1; // set 'out' to -1
    if(sum > threshold) out = 1; // otherwise set 'out' to +1

    // if value of 'out' is different from current state of neuron i
    if(out != network->output[i]) {
        updated = 1; // change 'updated' and return 1 to asynCor
        network->output[i] = out; // update state of neuron i
    }

    // call 'Energy' to calculate current NRG of network and retrieve
output
    float E;
    E = Energy(network);

    // print calculated energies for each function call to 'hopEnergy.dat'
    fprintf(fp2,"%f  \n", E);

    return updated; // returns 0 (initial state) if no conditions met
}
```

```
int x = 1; // define starting value of first column of 'hopEnergy.dat' -
serves as index of x values (iteration) and corresponding y values
(NRGies)

// Function that selects random neuron (i) and recursively calls
'updateNet' to asynchronously update network.
// Function called 'u' times (once for each unknown pattern).
void asynCor(net* network)
{
    int it; // counts iteration of calls to ‘updateNet’ for ‘u’
    int itofLastUpdate; // last iteration in which unit successfully
updated

    // 'it' and 'itofLastUpdate' reset to 0 every time function called
    it = 0;
    itofLastUpdate = 0;

    do {
        it++; // iterate 'it' at least once

        // 'x' counts each iteration of 'it' (does not reset to 0 once
condition of while loop met and function called again)
        fprintf(fp2,"%d  ", x++);
        fprintf(fp2,"%d  ", it);

        // select random neuron 'i' between 1 and 100 to be updated
        // if 'updateNet' returns 1 (not 0)
        if(updateNet(network, rand() % (N))) {
            // set 'itofLastUpdate' to value of 'it' and increment 'it'
            itofLastUpdate = it;
        }
    // function continues to call 'updateNet' until 'N' neurons passed
'minUpdate' times since nework last updated (network given N*'minUpdate'
chances to update; if not updated, 'updateNet' returns null each time and
function breaks out of while loop)
    // failure to update means network energy state is local minimum and
pattern has been recognized
    } while (it - itofLastUpdate < minUpdate*N);
}

// Function for simulating the network and recognizing the unknown pattern
(finding the local minimum).
void Recognize(net* network, int* unknown)
{
    int output[N]; // declare array to store outputs for printing

    // Print unknown pattern to be recognized
    setInput(network, unknown);

    // Asynchronously update network
    asynCor(network);

    // Print recognized output pattern
    getOutput(network, output);
```

```
}

// Main function for creating and simulating a neural network that
recognizes unknown input patterns given a set of exemplar patterns to
learn from.
void main()
{
    // open file for writing output of program
    fp1 = fopen("hopOutput.txt", "w");
    if(fp1==NULL) { // if file is not successfully opened
        printf("failed to open file: hopOutput.txt\n");
        exit(1);
    }

    // print u and p (calculated by macros)
    printf("\nSee file: hopOutput.txt to view the output of the
program.\n\n");
    fprintf(fp1, "# of unknown patterns: %d\n# of exemplars: %d\n\n", u,
p);

    net network; // declare new network using struct
    int n;

    // Allocate memory and create net
    initializeNet(&network);

    // Convert patterns to bipolar
    bipolarTransform(&network);

    // Calculate weight matrix
    calculateWeights(&network);

    fprintf(fp3,"Copy and paste the following information into
'hopPlot.py' to plot energies for each unknown input pattern (modify line
colour accordingly):\n\n");

    // declare array to store x-ranges over which NRGies for each input
pattern are displayed in fp2
    int xRange[u];

    // Recognize the pattern (find the local minimum)
    for(n=0; n<u; n++) {
        if(n>0) fprintf(fp2,"\n");
        xRange[n] = x;
        Recognize(&network, unknown[n]); // pass unknown patterns as input
    }

    // print instructions for plotting NRG (starting and ending lines,
i.e., 'xRanges' to index for plotting NRGies of each inputted pattern)
    for(n=0; n<u-1; n++) {
        fprintf(fp3,"plt.plot(fn[%d:%d,1], fn[%d:%d,2], 'b',
label=r'$Input%d$')\n", xRange[n], xRange[n+1]-1, xRange[n], xRange[n+1]-
1, n+1);
    }
```

```
    // instructions for plotting NRG of final input pattern
    fprintf(fp3,"plt.plot(fn[%d:,1], fn[%d:,2], 'b',
label=r'$Input%d$')\n\n", xRange[u-1], xRange[u-1], u);

    // instructions for plotting title of plot
    fprintf(fp3, "plt.title('Energy of Evolving Neural Net (u = %d; p =
%d; N = %d)')", u, p, N);

    // close files
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);

}
```

3.2 Makefile

```
hopfield: hopfield.o
      cc -ltrapfpe -lm -g $(CFLAGS) hopfield.o -o hopfield

hopfield.o: hopfield.c
      cc $(CFLAGS) -c hopfield.c
```

3.3 Also see:

hopOutput.txt
hopEnergy.dat
hotPlotInstructions.txt
hopEnergyPlot.ipynb

References

[1] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*(8), 2554–2558. https://doi.org/10.1073/pnas.79.8.2554

[2] [MATH 4IQB]. (2013, Nov 17). Hopfield Networks [Video File]. Retrieved from: https://www.youtube.com/watch?v=gfPUWwBkXZY

[3] Wikipedia contributors. (2014, May). Hopfield network. Wikipedia. [Online] Available: https://en.wikipedia.org/wiki/Hopfield_network#Initialization_and_running

[4] Altes, R. A. (n.d.). 8248 92037, 541–548.