Experimenting with Hopfield Nets

David B. Suits
Department of Philosophy
Rochester Institute of Technology
92 Lomb Memorial Drive
Rochester NY 14623

david.suits@rit.edu

Originally written in 1988.

In 1976, Allen Newell and Herbert Simon proposed their Physical Symbol System Hypothesis, a proposal generally regarded as one of the clearest statements of the presumption held by most artificial intelligence researchers. In their hypothesis, they stated that "A physical symbol system has the necessary and sufficient means for general intelligent action." They defined "general intelligence" as intelligence of the same scope as human intelligence. Their suggestion was that certain kinds of manipulation of explicit information can be construed as intelligent behavior.

Long before 1976 the search had concentrated on the proper kinds of explicit representations and the proper kinds of manipulation of that information. (You can trace this back at least as far as Leibniz in the 17th century.) What has emerged from this premise has been a flurry of successful, exciting starts in various areas of research. What has followed has been sad realizations that what is needed to extend these successes is computational power of such magnitude that it is mind boggling. This, the symbol manipulation approach to simulations of intelligence, now seems best suited for severely restricted domains (that is, expert systems).

Forceful explanations of the poverty of explicit symbol manipulation have been advanced. Apparently, it is the explicitness of both the symbols and their manipulations that is a central problem. Is it possible to devise artificial systems that harbor, in some fashion, *implicit* knowledge representations and manipulation?

One such approach involves the modeling of small biological units. This model assumes that simple units linked together in massively parallel architectures might be able to accomplish at least some of the tasks with which the symbol manipulation approach had the most difficulty, especially all those ambiguous tasks falling under the heading of pattern recognition.

The biological unit under investigation here is the neuron. What kinds of things can associations of simple neurons perform that the symbol manipulation model has difficulty with? For my experiments I have simulated a particularly straightforward "neural network" (described by J. J. Hopfield) and have given it the task of recognizing simple patterns. The constituents of the network are not biological neurons, but rather artificial neurons whose properties are much simplified from what is known and conjectured about biological neurons.

Artificial neurons

An artificial neuron is a simple device that computes an output as some function of its inputs (figure 1). The following are some considerations, following the biological metaphor, that might be useful for specifying the nature of such neurons.

We will want to know whether upper or lower bounds should be set on the number of inputs to a neuron. We will also need to specify whether these inputs are to be inhibitory or excitatory. Similarly, although a neuron will have only one output, it is allowed to branch off to become the input to an arbitrary number of other neurons. How will a neuron be designated as excitatory or

inhibitory? Neurophysiological evidence suggests that some neurons are excitatory to other neurons a short distance away, but inhibitory for neurons much farther away.

In an artificial neuron its inputs might be modified by some value (constant or variable) called an input's *weight*. A neurophysiological parallel to this occurs in the occasionally changing synapse strengths of neural connections; some synapses can even be permanently changed.

An artificial neuron, like its biological counterpart, might not be allowed to "fire" until the input signal strength (for example, the sum of its excitations and inhibitions) reaches or passes some threshold. We will want to know what this threshold is, whether it should be the same for all neurons, and whether it is ever allowed to change (and, if so, under what conditions).

After "firing", a neuron might be required to undergo a "recharging" — refractory — period, during which its output would be inactive. When designing an artificial neuron, we must decide what this refractory period is, whether it is to be the same for all neurons, and whether it is allowed to change.

If a neuron receives input during its refractory period, or if the input has not yet passed the neuron's threshold value, what happens to the input? Perhaps the input will be retained until the neuron is ready to deal with it, or perhaps the input will "decay" or grow weaker until used.

There might be an upper bound on the strength of the input, beyond which the neuron will be said to be saturated. Supersaturating a neuron might be one way to cause it to become inhibited or to shut down temporarily with "fatigue".

What will be the outputs of neurons? Will they be limited to binary values (+1 and 0, or +1 and –1, perhaps)? Biological neurons generally distinguish their inputs by the frequency of the input firings. In an artificial neuron, such frequencies might be represented by means of a range of values.

Not only does a wide selection exist for possible neuron types, a practically infinite variety of ways exists for connecting neurons together — from fully connected nets (such as Hopfield nets, as we will see) to randomly connected nets, and from single-layered to multilayered. Nets may be designed to eventually reach a stable state, or to oscillate among certain states. The output of a new might be sent to one or a number of motor devices, or the output might simply be represented by a stable state of the net. The particular architecture chosen will reflect the application for which the net is to be used. To avoid immediate confusion and chaos when toying with the possibilities, it is sometimes helpful to investigate the properties of small collections of neurons and then to build larger structures from the smaller units.

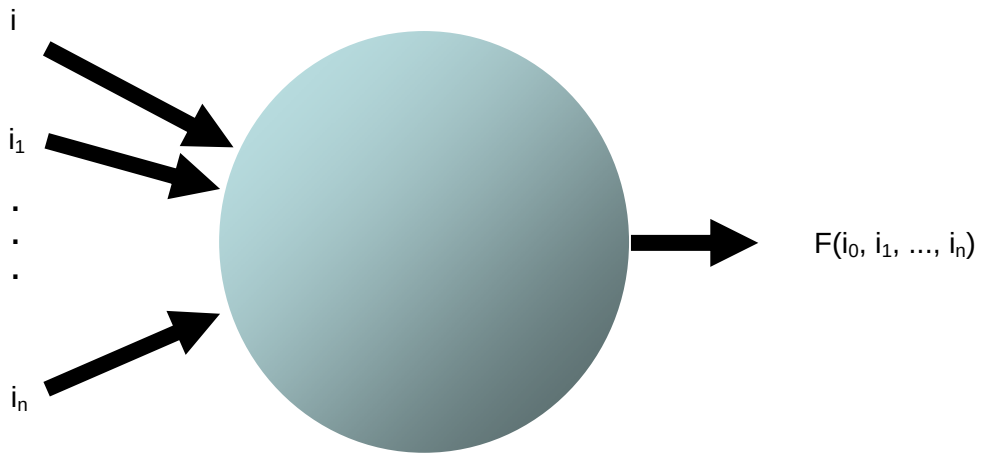

Figure 1. An artificial neuron is a device which computes an output as some function of its inputs.

The Hopfield net

A Hopfield net is a collection of artificial neurons (or *nodes*), each connected to all others by means of links (*arcs*). Each node also has an arc to itself. Each node computes a relatively simple function of all its inputs (each of which has a value of +1 or –1), then sends the result (either +1 or –1) out to all nodes.

Associated with each arc is a "weight" — a modifier that acts on values input to the node. Weights are symmetric, which means that the arcs are nondirectional: WEIGHT[i,j] = WEIGHT[j,i] for all node i, j. Figure 2 is a schematic representation of a possible four-node Hopfield net, with arbitrary values and weights. (The weight of an arc from a node to itself will be 0.) Nodes have no refractory periods and no inhibitory inputs and outputs. Figure 3 presents pseudo-code for the algorithm for a Hopfield net, and Listing One provides a complete program in C. The assignment of connection weights (pseudo-code Step 1, and function **`train()`** in the listing) is a version of a simple learning procedure proposed by D. O. Hebb. (For variations on Hebb's rules, see Rumelhart, Hinton, and McClelland, 1986.)

A Hopfield net begins life *tabula rasa* — that is, all weights are 0. Weights are established during the initial "training" of the net, which is accomplished by feeding the net any number of "exemplars" (training patterns). Each time the net is trained on a pattern, the two values from the pattern nodes at the end of the arc are multiplied together. Since a node's value is either +1 or –1, the product of the two nodes' values will be either +1 (if the values are the same) or –1 (if the values are different). The product is then added to the arc's weight. (If the same pattern is presented to the net multiple times, the arcs will be weighted more and more heavily in favor of that pattern.) It is just as easy to have the net "forget" a training pattern: just subtract the node products from the arcs' weights. (See function **`forget()`** in the listing.)

Once the weights are established, they do not change (unless and until there is another training period), and the net is ready to receive unknown input patterns consisting of just as many elements as the net has nodes. (See Step 2 in the pseudo-code and function **`get_unknown()`** in the listing.) Each node in the net is initialized to the corresponding value (+1 or –1) of the input pattern. The values are then multiplied by the arcs' weights. A non-negative result gives a node a value of +1, and a negative result gives the node a value of –1. (See pseudo-code Step 3 and function **`relax()`** in the

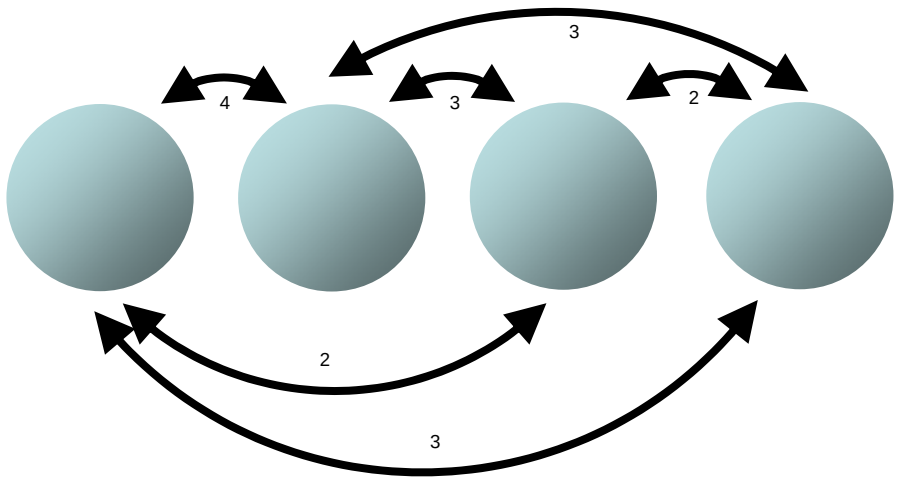

Figure 2. A possible 4 node Hopfield net, with arbitrary weights.

listing.) All nodes continue to be updated in this manner until no changes occur, at which time the net is said to have converged (or "relaxed") to a stable state. Note that convergence makes use of a net's weights but does not change them. Training and convergence may be interleaved in any order. (You might say that training is a net's way of being programmed, and presenting an input to the net is a way of running that program.)

It may be possible for a Hopfield net to enter oscillating states if the updating of outputs is carried on synchronously (that is, by calculating the output for all nodes and only then copying the result *en masse* to the inputs of all nodes). Asynchronous updating might cause the net to act in different ways (although perhaps not noticeably different for many cases), depending on the order of updating. Since the net is supposed to be a parallel machine (each node acting asynchronously in isolation from the others), the simulation of a Hopfield net on a serial machine would seem to favor a random, asynchronous updating scheme (function **`set_sequence()`** in the listing).

A Hopfield net has several properties to notice. It may be used as a content-addressable associative memory. After training, a given input will cause the net to converge toward the exemplar "closest" to the input pattern. "Closest" here describes the action of the net's weights, which (by means of their pattern of values) constitute a kind of memory of the training exemplars. Metaphorically, the exemplars ennumbered in the weights represent "energy minima", and the process of convergence takes the elements of the input pattern to the closest energy minimum. An input pattern that does not match any of the original exemplars will have a tendency to find the best match.

A Hopfield net is limited in the number of exemplars it can be expected to maintain as separate minima. Beyond some point, the exemplars will interfere with each other and cause a distortion in one or more of them, as discussed further below. Moreover, the number of arcs in a Hopfield net increases as the square of the number of nodes; consequently, large nets will quickly prove to be impractical both to construct and to simulate.

Experiments with a Hopfield net

For the experiments here, we will use a 48-node Hopfield net, conceptualized as a rectangular arrangement of eight rows of six nodes each. The purpose of this arrangement is that input and

```
Given N nodes, fully connected, and M exemplars (training patterns), each
with N elements, each of which is either on (+1) or off (−1):

Step 1: Assign the connection weights.
    For each node i ← 0 .. N − 1,
      for each node j ← 0 .. N − 1,
            if i = j then weight [i,j] ← 0
            else weight [i,j] ← sum of:
                  for each exemplar e ← 0 .. N − 1,
                        exemplar[e,i] * exemplar[e,j]
Step 2: Initialize the net with some input pattern (with N elements, each of
which may be +1 or -1).
    For each node i ← 0 .. N − 1,
      output[i] ← pattern[i].
Step 3: "Relax" the net: iterate until there are no more changes (or until
some maximum iterations have been done).
    For each node i ← 0 .. N − 1,
      temp ← sum of:
            for each node j ← 0 .. N − 1,
                  output[i] * weight[i,j].
      if temp ≥ 0 then output[i] ← 1.
      else output[i] ← -1.
Step 4: If there are more unknown input patterns to examine, go to step 2.
```

Figure 3. A Hopfield net algorithm.

output patterns may more easily be recognized by the human observer. Updating is asynchronous, using a random ordering fixed only once at the beginning of the program. Ten exemplar patterns are available. For future reference, the names of these patterns are the names of the ten digits 0…9, which the patterns are meant to depict (see figure 4 and function **`get_patterns()`** in the listing).

Figure 5 shows the results after training the net on the patterns 0, 1, and 2, and then giving the net the unknown input 0. Notice that the net converges at once to the pattern 0. The comment "Weight changes: 0" indicates that all "pixels" in the input pattern already represent an "energy minimum", and consequently the net was immediately in a stable state. Similar results occur for the input patterns 1 and 2. That is, the net "remembered" the three training patterns perfectly. Or, rather, the net was able to immediately reproduce the training patterns perfectly.

Let us now train the net on an additional pattern, 3, and give it 0, 1, 2, and 3 as four unknown inputs. Figure 6 shows that the addition of the fourth exemplar caused some confusion in trying to

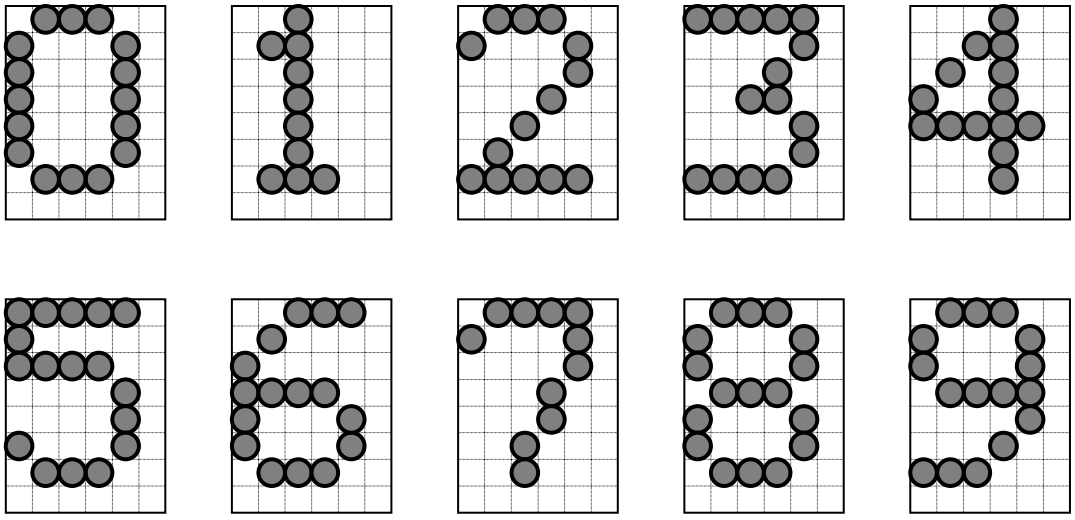

Figure 4. 6 x 8 "thin" patterns.

Figure 5. When trained on exemplars '0', '1', and '2', the net clearly "remembers" them, because the unknown inputs '0', '1', and '2' cause the net to converge immediately to the corresponding exemplars.

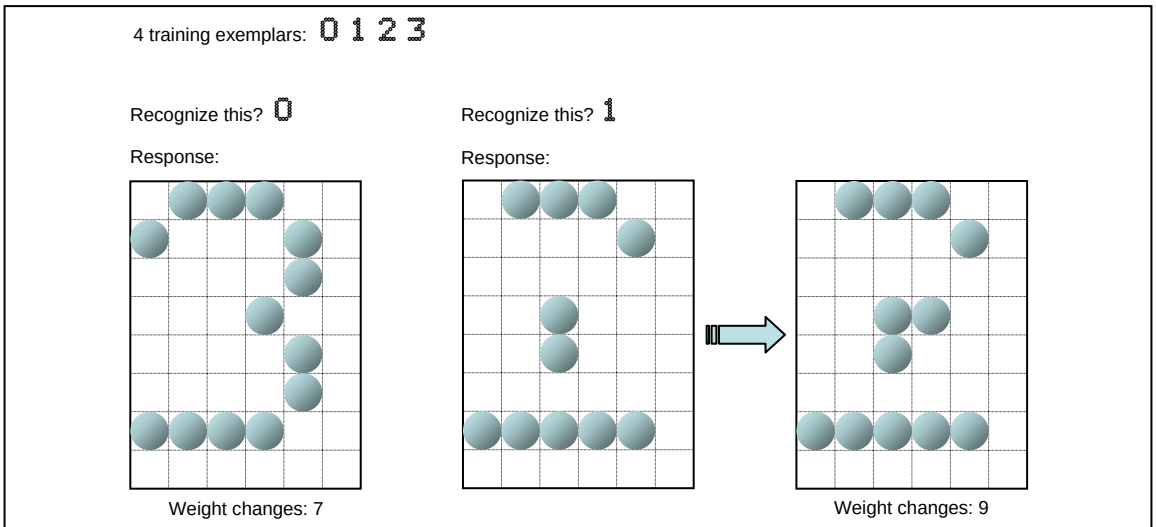

Figure 6. When trained on exemplars '0', '1', '2', and '3', the net seems to get a bit confused.

respond to the inputs. This is somewhat unexpected, because a Hopfield net with *n* nodes id usually expected to be able to differentiate at least .15*n* exemplars, which would be 7.2 in the case of the 48-node net.

An explanation is at hand. A pattern is not simply a relative ordering of dots, because non-dots also account for weight changes during training. Consequently, the exemplars given to the net so far, consisting mostly of white space, or "off pixels", quickly interfere with each other. The solution is to design exemplars that are as close as possible to having half their pixels on and half off. The exemplars in figure 7 are identical to the former set, except that the pixels have been "smeared" to the right. The ratio of pixels on to pixels off now comes much closer to 1:1, and, as the output in figure 8 shows, the net is now capable of handling four of the five exemplars (0, 1, 2, 3, and 4) without confusion. The inaccuracy in reproducing the '4' is due to the similarity of some of the exemplars, especially 0, 3, and 5. Patterns that are carefully prepared to be distinct would improve the net's performance. For example, the eight exemplars in figure 9 were recognized perfectly.

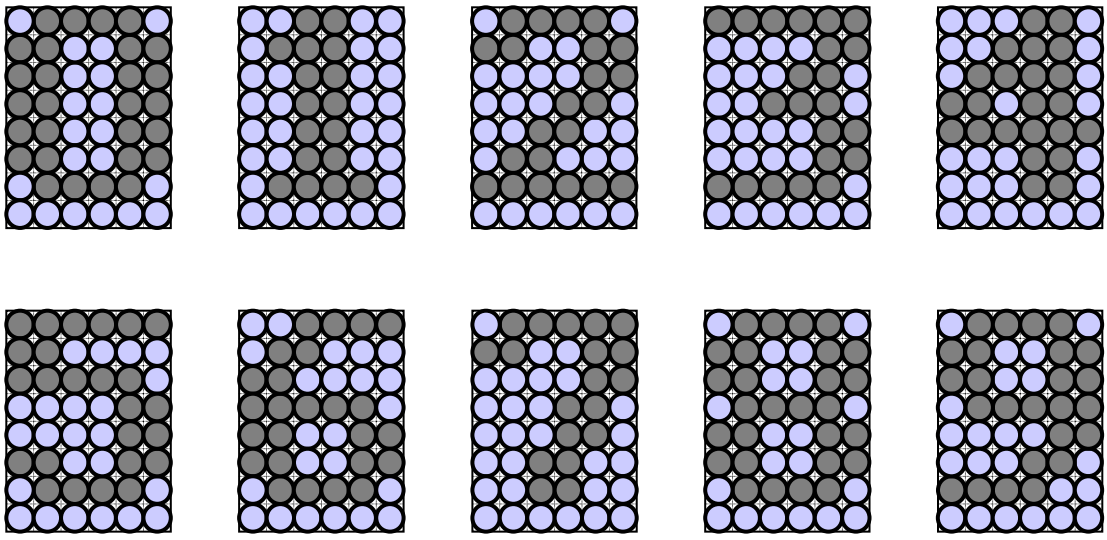

Figure 7. Ten "fat" training patterns whose ratio of dots to spaces approaches 1:1. These patterns will be used from now on.

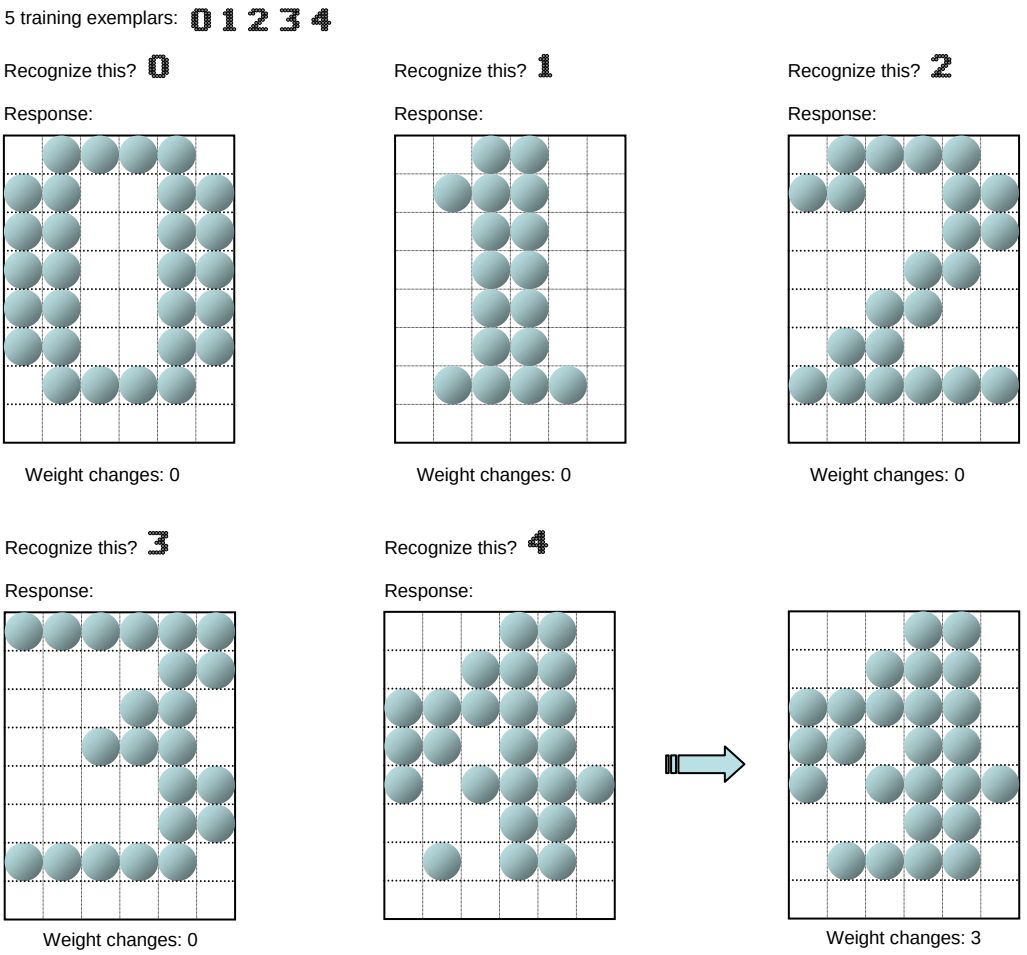

Figure 8. When trained on the five "fat" patterns 0 through 4, the net had trouble on only one of them.

Hopfield nets are capable of classifying an input pattern as being more or less like a known exemplar. That is, an input pattern that does not exactly match an exemplar will call up the exemplar which is the nearest match. To see this in action, train the net on 0 through 4, and then give as unknown input patterns "noisy" variations on each of the exemplars. (A *noisy variation* for this experiment is one in which each pixel is switched from one to off, or vice-versa, with a probability of 0.25.) Figure 10 shows that although not all patterns are recognized perfectly, the net performs fairly well.

A property that any net ought to have is the ability to degrade gracefully in the presence of damage, instead of winking out all at once as though the plug had been pulled. Figure 11 confirms the gradual degradation in performance of the simple Hopfield net when more and more arcs are cut (that is, their weights maintained at 0).

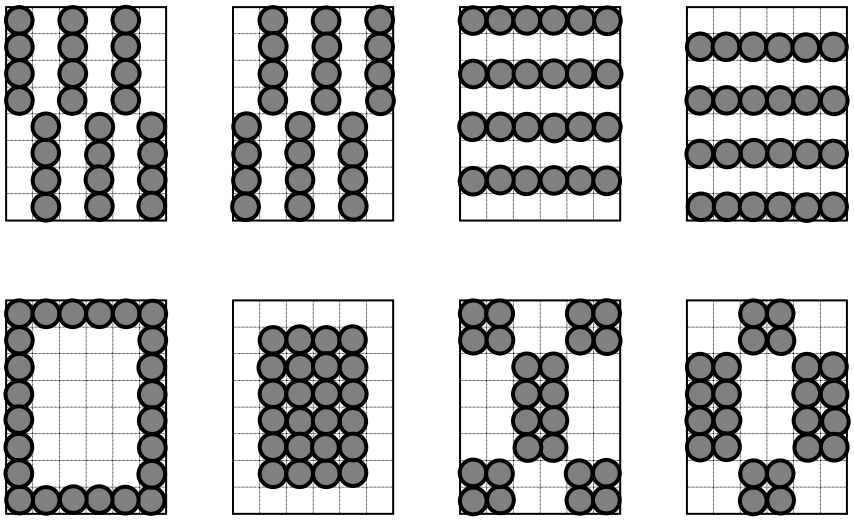

Figure 9. Eight hand-crafted patterns. The 48-node Hopfield net is able to distinguish all of them without interference.

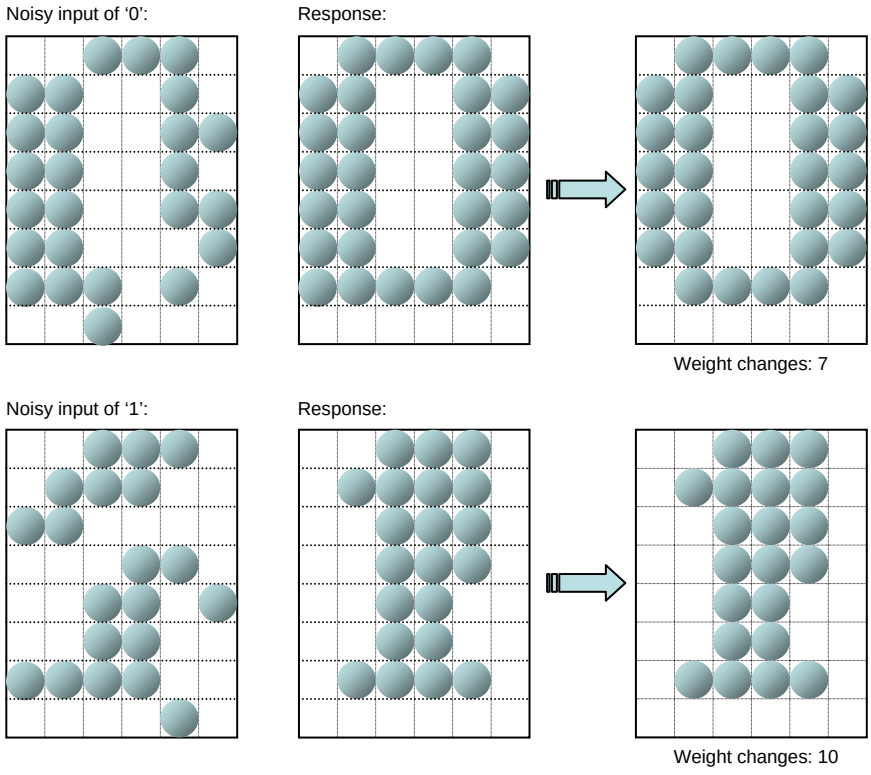

Figure 10. Two examples of net output (trained on '0' through '4') when given noisy input patterns.

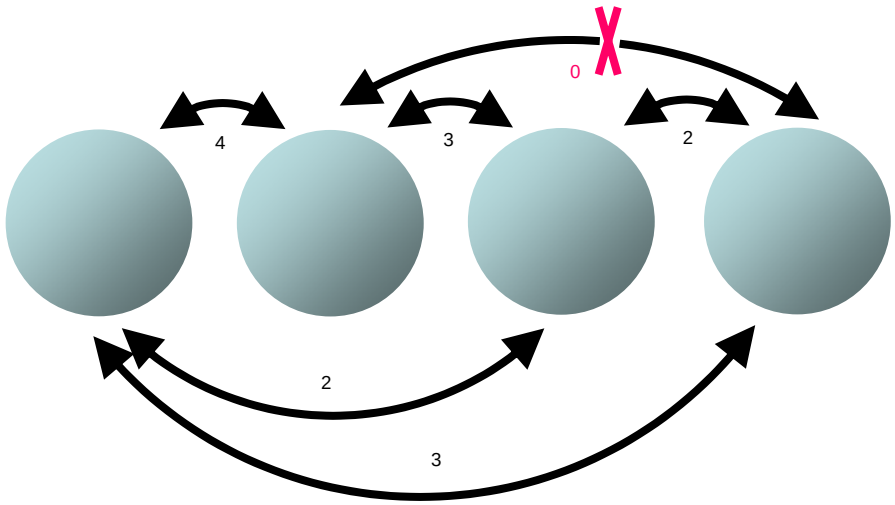

Figure 11. If a single arc is cut (i.e., its weight maintained at 0), then the net can still perform well. But as more and more arcs are cut, the net performs less and less reliably.

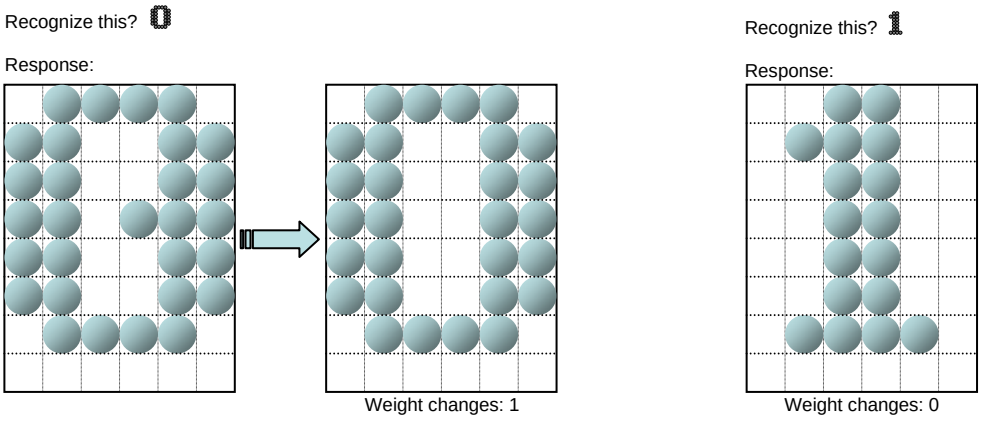

Figure 12. When 20 arcs are broken (1.7% of the total), the net still performs well.

Variations on the Hopfield net

It may be possible to introduce some variations to the Hopfield net without destroying its "Hopfieldness". The following variations are given as suggestions easy to implement for further experiments.

Suppose each node requires a refractory period — a period of time (as measured by some global ticker) during which its output does not change, even though its input does. The result (where each node begins with an arbitrary value between 0 and the refractory period) seems to be a kind of slowing down of the convergence of the net, and the net converges somewhat more "carefully". I have been able to give such a net the exemplars 0 through 5 (which before caused confusion) with somewhat *less* conflict among the exemplars; on occasion it was able to pick out correctly all six of the patterns.

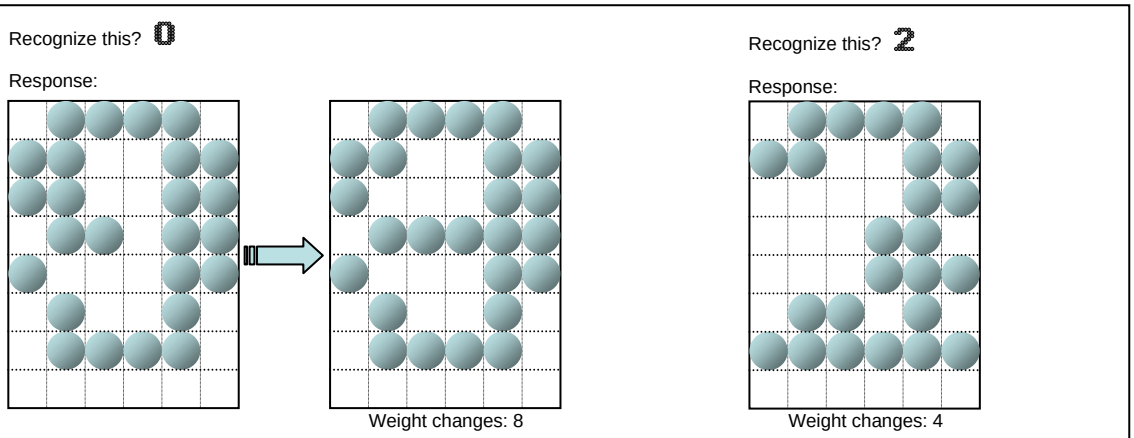

Figure 13. When 70 arcs are broken (6.0% of the total), the net gets confused.

Still another version of refraction might consist of raising the threshold from 0 (see the function **`relax()`** in the listing) whenever a node "fires" (output is +1). The threshold might be raised to an arbitrarily large number and then gradually allowed to drop back to 0.

Perhaps conflict among exemplars might be reduced if the input patterns were changed so as to "fuzzify" them a little, under the assumption that whatever conflict might exist among the exemplars might be spread out and therefore alleviated. Imagine, then, the normal 6×8 net along with an auxiliary net one-fourth the size (3×4). The regular net (a relatively fine-grain net) will be trained as usual, but each node in the auxiliary (coarse-grain) net will represent the logical ORing of a 2×2 area of pixels from each given exemplar. That is, the coarse-grain net will be trained on twelve-pixel patterns. Now relax the coarse-grain net, and for each of its nodes whose pixels are off, make sure that all four corresponding pixels in the fine-grain net are turned off. (All other pixels remain unaffected.) Now relax the fine-grain net. What is the result? My experiments, alas, showed no change whatsoever in the behavior from the original. (It would be interesting to explore the mathematical reasons for that.)

When the net is trained on exemplars, the nodes' weights change. (The changing of the weights just *is* the learning of new patterns.) Another approach to changing weights is to introduce some kind of modifier (a weighting of weights, so to speak) based, say, on the "distances" of nodes from each other. A node far from a second will have less influence on the second than will a third, which is closer to the second. Perhaps the influence will approach 0 after a certain distance (and perhaps beyond that point a weight modifier will become inhibitory).

Finally, try training the net on one or more exemplars and then presenting as unknown input only a part of one of the exemplars. The net ought to be able to recreate the entire original exemplar.

References

Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences of the USA*, 79 (April, 1982): 2554–2558.

Hopfield, John J., and David W. Tank, "Collective Computation in Neuron-Like Circuits", *Scientific American* 257 (December, 1987): 104–114.

Newell, Allen, and Herbert A. Simon, "Computer Science as Empirical Inquiry; Symbols and Search", *Comm. ACM* 19 (March, 1976): 113–126.

Rumelhart, D. E., G. E. Hinton, and J. L. McClelland, "A General Framework for Parallel Distributed Processing", in Rumelhart, D. E., J. L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1 (Cambridge, MA: MIT Press, 1986): 45–76.

```
/*
============================================================
   HOPNET -- An experimental Hopfield net.

   David B. Suits

   V1.0   25 Sept. 20 a.l.

============================================================
*/

#include "stdio.h"

#define NUM_ROWS        8       /* Conceptual rows in the net.          */
#define NUM_COLS        6
#define NUM_NODES       (NUM_ROWS * NUM_COLS)
#define MAX_ITERATIONS  8       /* Convergence (relaxation) of the net  */
                                /*   will stop if not stable by this    */
                                /*   many iterations.                   */
#define PIXEL           '*'     /* An "on" pixel printed to screen. An  */
                                /*   "off" pixel will be a space.       */
#define MAX_PATTERNS    10      /* Num patterns in file "PATTERNS".     */
#define MAX_EXEMPLARS   50      /* Max training patterns. Note that     */
                                /*   MAX_EXAMPLARS does not mean the    */
                                /*   same as MAX_PATTERNS, since a pat- */
                                /*   tern may be presented as an exemp- */
                                /*   lar multiple times. This limit is  */
                                /*   necessary only for defining the    */
                                /*   length of exname[], below.         */

int weight[NUM_NODES][NUM_NODES],
    input[NUM_NODES],           /* Each input pattern goes here.        */
    output[NUM_NODES],          /* Output of the net.                   */
    seq[NUM_NODES],             /* Will hold random updating sequence.  */
    noisy,                      /* TRUE if exemplar is to be "noisy".   */
    num_exemplars,              /* Num training patterns actually given.*/
    patterns[MAX_PATTERNS][NUM_NODES};  /* Patterns read from file are  */
                                        /*   held in this array, just   */
                                        /*   to simplify access.        */

char exnames[MAX_EXEMPLARS + 1], /* String of names (1 char each) of ex-  */
                                 /*    emplars on which the net has been  */
                                 /*    trained.                           */
     unknown;                    /* Unknown input pattern's name.         */

/*
------------------------------------------------------------
MAIN
------------------------------------------------------------
*/

main()
{
  void set_sequence(), train(), initialize(), noisify(), relax(),
       get_patterns(), forget();

  int i, j, c;

    puts("\nSuits's Hopfield net experimenter.\n\n");

    get_patterns();     /* Read patterns in from file "PATTERNS".       */
```

```
    set_sequences();    /* Construct a random updating sequence.        */

    for (i = 0; i < NUM_NODES; ++i)     /* All weights start at 0.      */
        for (j = 0; j < NUM_NODES; ++j)
            weight[i][j] = 0;

    num_exemplars = 0;  /* Net has been trained on nothing so far.      */
    noisy = FALSE;      /* TRUE only for noisifying an input pattern.   */

    while (1) {         /* Do forever ... */
        c = menu();     /* Get user's choice of actions.        */

        switch(c) {
            case 'F' : forget();                /* Forget an exemplar.    */
                       break;
            case 'N' : if (num_exemplars == 0)  /* Input a noisy pattern. */
                            puts("There are no exemplars!\n");
                       else if (get_unknown()) {
                           noisify();
                           noisy = TRUE;
                           initialize();
                           relax();
                           noisy = FALSE;
                        }
                        break;
            case 'T' : train();                 /* Train on new exemplar.    */
                       break;
            case 'U' : if (num_exemplars == 0)  /* Input an unknown pattern. */
                           puts("There are no exemplars!\n");
                       else if (get_unknown()) {
                           initialize();
                           relax();
                       }
                       break;
            case 'Q' : exit();                  /* Quit.         */

            /* If none of the above is chosen, ignore the input. */
        }   /* Switch on main choice. */
    }

} /* main() */


/*
------------------------------------------------------------
FORGET -- "Untrain" the net on a chosen exemplar.
------------------------------------------------------------
*/

void forget()
{
  int c, n, p, i, j, got_it;

    puts("Enter exemplar number to forget: ");
    if ((c = getchar()) < '0' || c > '9') {
        puts("\nInvalid!\n");
        return;
    }

    putchar('\n');

    /* If the named exemplar is not one of the exemplars so far
       used, say so. Otherwise, delete each instance of its name
```

```
       from the list of exemplars and, for each deletion, adjust
       the net's weights.
    */

    p = c - '0';

    got_it = FALSE;

    n = 0;

    while (exname[n]) {
        if (exname[n] == c) {
            got_it = TRUE;
            strcpy(exnames + n, exnames + n + 1);
            --num_exemplars;

            for (i = 0; i < NUM_NODES; ++i)
                for (j = 0; j < NUM_NODES; ++j)
                    if (i != j)
                        weight[i][j] -= patterns[p][i] * patterns[p][j];
        }
        else
             ++n;
    }

    if (!got_it)
        puts("NOT A PRESENT EXEMPLAR!\n");

} /* forget() */


/*
-------------------------------------------------------------
GET_PATTERNS  -- Read all the patterns from a disk file. The file
                 is a straight ASCII text file, with '*' (or any other
                 non-space printable character) for an "on pixel",
                 and a space for an "off pixel". Each pattern must have
                 NUM_ROWS number of lines (even if the lines are blank),
                 but may have less than NUM_COLS columns in each line.
                 (The end of a line will indicate that the remainder
                 of a line has "off pixels".)
-------------------------------------------------------------
*/

void get_patterns()
{
  int p, i, j, c;
  FILE *fp;

    if ((fp = fopen("patterns", "r")) < 1) {
       puts("\nFile PATTERNS not found.\n");
       exit(1);
    }

    for (p = 0; p < MAX_PATTERNS; ++p)          /* Make all patterns blank.  */
         for (i = 0; i < NUM_NODES; ++i)
             patterns[p][i] = -1;

    for (p = 0; p < MAX_PATTERNS; ++p) {
        for (i = 0; i < NUM_ROWS; ++i) {
            for (j = 0; j < NUM_COLS; ++j) {
                if ((c = fgetc(fp)) > ' ')
                   patterns[p][i * NUM_COLS + j] = 1;
```

```
                else {
                    if (c == 13)       /* If a CR ...                    */
                        c = fgetc(fp); /* ... throw away the line feed. */
                    if (c != ' ')      /* EOL or EOF.                    */
                        break;
                }
            }
            if (c == EOF)
                break;
            while (c != 10)           /* Throw away any extra chars on line. */
                c = fgetc(fp);
        }
    }

    fclose(fp);

} /* get_patterns() */


/*
------------------------------------------------------------
GET_UNKNOWN -- Get an unknown pattern for input to the net.
------------------------------------------------------------
*/

int get_unknown()
{
  int i, c;

    puts("Enter number for unknown pattern: ");
    if ((unknown = getchar()) < '0' || unknown > '9') {
       puts("\nINVALID!\n");
       return (FALSE);
    }

    c = unknown - '0';

    for (i = 0; i < NUM_NODES; ++i)
        input[i] = patterns[c][i];
    printf("\n\nT: %s  U: %c", exnames, unknown);

    return (TRUE);

} /* get_unknwon() */


/*
------------------------------------------------------------
INITIALIZE -- Initialize the net with an unknown pattern.
------------------------------------------------------------
*/

void initialize()
{
  void show_output();

  int i;

    for (i = 0; i < NUM_NODES; ++i)
        output[i] = input[i];

    if (noisy) {
       printf("\nTime: 0 Noisy Pattern: %c\n", unknown);
```

```
       show_output();
    }

} /* initialize() */


/*
------------------------------------------------------------
MENU -- Print a menu and return user's selection as uppercase char.
------------------------------------------------------------
*/

int menu()
{
  int c;

    puts("\nExemplars: ");
    if (num_exemplars == 0)
       puts("none\n");
    else
       puts("%s\n", exnames);

    puts("T  Train on an exemplar.\n");
    puts("F  Forget an exemplar.\n");
    puts("U  Get an unknown input pattern and relax the net.\n");
    puts("N  Same as 'U' but "noisify' the input pattern first.\n");
    puts("Q  Quit.\n");

    puts("Choice? ");
    c = toupper(getchar());
    putchar('\n');
    return (c);

} /* menu() */


/*
------------------------------------------------------------
NOISIFY -- If an input pattern is to be a noisy one, then flip
           each of its "pixels" with probability of .25
------------------------------------------------------------
*/

void noisify()
{
  int i;

    for (i = 0; i < NUM_NODES; ++i)
        if ((rand() %4) == 0)
           input[i] = -[input[i];    /* -1 <-> +1  */

} /* noisify() */
```

```
/*
------------------------------------------------------------
RELAX -- Iterate until convergence, or until a maximum number of
            iterations have been done.
------------------------------------------------------------
*/

void relax()
{
  void show_output();

  int time, k, i, k, sum, changes;

    time = -1;                  /* Keep track of iterations done.        */

    while (time < MAX_ITERATIONS) {
        changes = 0;            /* Convergence reached when there are    */
                                /*   no changes in the net's output.     */

        for (k = 0; k < NUM_NODES; ++k) {
            i = seq[k];         /* Pick a node. */
            sum = 0;            /* Sum all weighted inputs to this node. */

            for (j = 0; j < NUM_NODES; ++j)
                sum += weight[i][j] * output[j];

            if (sum >= 0)       /* Threshold is 0, and node output is    */
                sum = 1;        /*    +1 (on) or -1 (off).               */
            else
                sum = -1;

            if (sum != output[i])  /* Different from previous output? */
                ++changes;
            output[i] = sum;       /* New output. */
        }

        printf("\nTime: %d  Changes: %d\n", time, changes);
        show_output();

        if (changes == 0)
           break;

        ++time;
    }

} /* relax() */


/*
------------------------------------------------------------
SET_SEQUENCE -- Construct a random sequences for updating the nodes.
------------------------------------------------------------
*/

void set_sequence()
{
  int i, j, temp;

    for (i = 0; i < NUM_NODES; ++i) {
        j = rand() % NUM_NODES;
        temp = seq[j];
        seq[j] = seq[i];
        seq[i] = temp;
```

```
    }

} /* set_sequence() */


/*
------------------------------------------------------------
SHOW_OUTPUT -- Draw the matrix to show the net's latest output.
------------------------------------------------------------
*/

void show_output()
{
  int i, j;

    for (i = 0; i < NUM_COLS + 2; ++i)  /* Top border.         */
        puts("+ ");
    putchar('\n');

    for (i = 0; i < NUM_ROWS; ++i) {
        puts("+ ");                     /* Left border.        */

        for (j = 0; j < NUM_COLS; ++j) {
            if (output[i * NUM_COLS + j] == 1)
               putchar(PIXEL);
            else
               putchar(' ');
            putchar(' ');
        }

        puts("+\n");                    /* Right border.       */
    }

    for (i = 0; i < NUM_COLS + 2; ++i)      /* Bottom border.      */
        puts("+ ");

    putchar('\n');

} /* show_output() */


/*
------------------------------------------------------------
TRAIN -- Modify the net's weights according to a new training
         pattern (exemplar).
------------------------------------------------------------
*/

void train()
{
  int c, i, j;

    if (num_exemplars >= MAX_EXEMPLARS) {
        puts("NO ROOM FOR MORE EXEMPLARS!\n");
        return;
    }

    puts("Enter pattern number for new exemplar: ");
    if ((c = getchar()) < '0' || c> '9') {
        puts("\nINVALID!\n");
        return;
    }
```

```
    putchar('\n');

    exnames[num_exemplars++] = c;
    exnames[num_exemplars] = '\0';

    c = c - '0';

    for (i = 0; i < NUM_NODES; ++i)
        for (j = 0; j < NUM_NODES; ++j)

            if (i != j)
               weight[i][j] += patterns[c][i] * patterns[c][j];

} /* train() */

/*
============================================================
End of HOPNET
============================================================
*/
```