Getting Started with Cascade-correlation Learning:

Examples from XOR and Continuous XOR

This material provides an introduction to cascade-correlation learning through seven simple examples. The cascade-correlation algorithm was invented by Scott Fahlman at Carnegie Mellon University. The enclosed software includes compiled versions of both Fahlman's original Lisp code (here referred to as CC) and our user shell (referred to as SDCC) designed to make it easier to use Fahlman's code in designing, running, and analyzing simulations.

The SDCC code facilitates running either standard CC, in which each recruited hidden unit is installed on its own separate layer or sibling-descendant CC (SDCC). In SDCC, the learning algorithm dynamically decides whether it is better to install a new hidden unit on its own separate layer (as a descendant) or on the current highest layer (as a sibling).

Seven accompanying examples show how to use the SDCC program and its accompanying manual. As you will see, a little bit of Lisp programming is required to use SDCC effectively. The introductory chapters of any textbook on Common Lisp will cover what you need in that respect. A particularly good contemporary textbook is Graham, P. (1996). *ANSI Common Lisp*. Englewood Cliffs, NJ: Prentice-Hall. This book is now provided online at http://www.paulgraham.com/onlisp.html .

A good online Lisp tutor is available at http://art2.ph-freiburg.de/Lisp-Course . Lisp basics, except for the important topic of *do* loops, are taught in the first three lessons. You can learn about *do* loops in any good Lisp book, including the Graham book.

A good, free-trial version of Common Lisp can be obtained from http://www.franz.com/ . Installation of this program will be necessary to run the CC and SDCC code.

If you are using this software for the first time, follow this order: open Common Lisp, load the compiled version of CC, and load the compiled version of SDCC. Note that CC must be loaded before SDCC is loaded.

Lisp program code for each of the seven examples is contained in the seven files ex1.cl to ex7.cl. Before evaluating any of the code in these seven files, load the compiled versions of CC and then SDCC, in that order.

Two different example problems are used: the simple exclusive-or problem (XOR), and the slightly more complex continuous-XOR problem. The seven examples include:

1. Running a single XOR network and examining the final weights.
2. Continuous recording of network error in the XOR problem.
3. Running several XOR networks while recording training error.
4. A continuous version of XOR using standard CC.
5. A continuous version of XOR using SDCC.
6. Perceptual effects in continuous XOR.

7. Analyzing knowledge representations in a continuous-XOR network.

Example 1. Running a Single XOR Network and Examining the Final Weights

The simple exclusive-or (XOR) problem has two binary inputs and a single binary output. If either one, but not both, of the two inputs are on, the output should be on. If both inputs are on, or both inputs are off, the output should be off. The input space for XOR is shown in Figure 1. Inputs that are on are portrayed with a 1, and inputs that are off are portrayed with a 0. (Positive vs. negative input, e.g., 1 and -1, or .5 and -.5, respectively, would work just as well.) It turns out that no linear combination of connection weights from the inputs to the output can solve the XOR problem. Graphically, this means that there is no way to draw a line in Figure 1 that separates the input combinations that turn on the output from the input combinations that turn off the output. As shown in Figure 1, it takes two lines to separate the positive input combinations from the negative input combinations.

In more general terms, for problems with more inputs, there is no hyperplane that can separate positive from negative combinations of inputs. The XOR problem is said to be not linearly separable. This means that hidden units are required to learn the problem. Networks without direct input-to-output connections, sometimes called cross connections, require two hidden units to solve the XOR problem. Because CC allows cross connections, it recruits only a single hidden unit in its XOR solution.

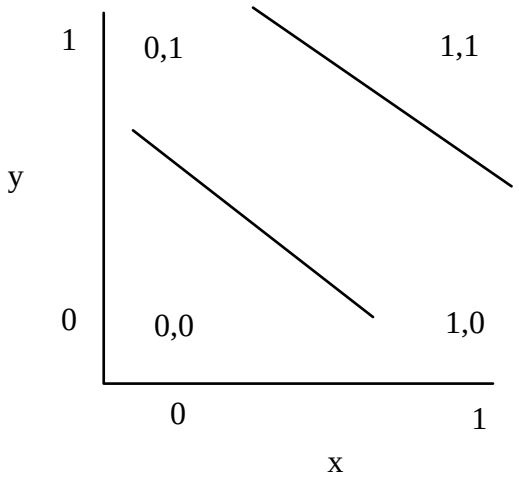

Figure 1. Input space for the XOR problem.

We begin by defining a new variable, called *xor-patterns*. It contains the training patterns for the XOR problem, but we initialize it to *nil* to start with. You should type or copy this *defvar* statement into your Lisp interpreter and evaluate the statement by selecting it and pressing Ctrl-E. Lisp code can be selected by inserting the curser immediately after the last closing parenthesis and double clicking the left mouse button; or by dragging the cursor over the code while holding down the left mouse button. Instead of pressing Ctrl-E, you can evaluate selected code by choosing Incremental Evaluation in the Tools menu. All the Lisp code for this example is contained in the file ex1.lsp. Note that it is

conventional in Common Lisp to indicate global variables by surrounding their names by asterisks. Global variables are those that hold their current values no matter which procedure is currently active.

```
(defvar *xor-patterns* nil)
```

Next, we define a new procedure called *run-xor* that will run a single XOR network. The part in double quotations is documentation that can be read by a human. It is good Lisp practice to list the arguments and verbal description for each defined procedure in this documentation part, so inquisitive users can examine it whenever they need to.

```
(defun run-xor ()
  "()
Run 1 xor net & look at the final weights."
  (setq *xor-patterns* '(((0.0 0.0) (-.5))
                         ((0.0 1.0) (.5))
                         ((1.0 0.0) (.5))
                         ((1.0 1.0) (-.5))))
  (set-patterns *xor-patterns* 'train)
  (train 100 100 25)
  (pprint *weights*)
  (pprint *output-weights*))
```

The first statement in the procedure definition sets the *xor-patterns* variable to a list of training patterns. This list contains sub-lists of one pattern each, where each pattern is composed of a list of input values, followed by a list of output values. In this case, there is only one output per pattern, but in general, there can be any number of inputs or outputs. We use 0.5 to specify that the output should be on, and -0.5 to specify that the output should be off. This is because the default activation function for output units is the sigmoid function, with limits at -0.5 and 0.5, as shown in Figure 2.

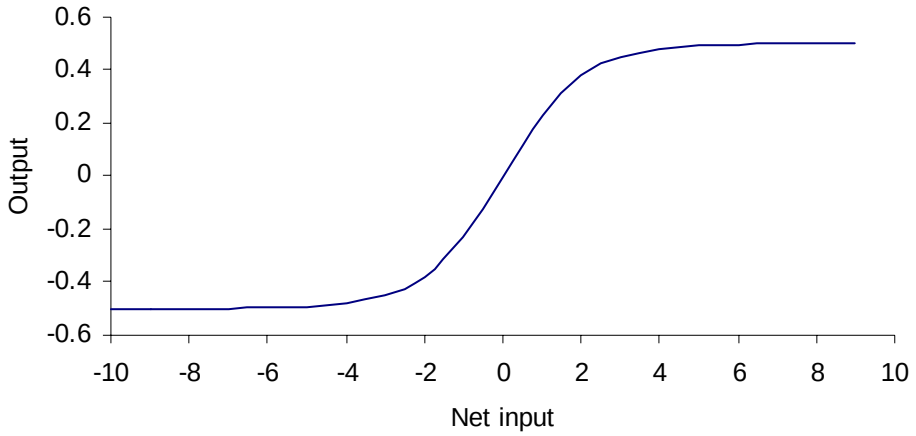

Figure 2. Sigmoid activation function.

The second statement in the procedure definition is a call to the *set-patterns* procedure. As shown in the manual, *set-patterns* takes two arguments – a list of patterns and the type of these patterns ('train, 'test, or 'change-train). The patterns must be formatted as we just did for the XOR patterns, i.e., as a list of patterns, where each pattern is a list composed of a list of inputs followed by a list of outputs. Because we already set *xor-patterns* to the correct value, we simply use the variable name *xor-patterns* in this procedure call. We further specify that the type of these patterns is 'train. This procedure call to *set-patterns*

sets up the network with two inputs and one output and with the XOR patterns as the training environment.

The third statement in the procedure definition calls the *train* procedure, which starts the training. As specified in the manual, *train* has three obligatory and two optional arguments. The three obligatory arguments are *outlimit*, *inlimit*, and *rounds*. *Outlimit* and *inlimit* are upper bounds on the number of cycles in each output and input phase, respectively. The output phase trains weights going into output units, and the input phase trains weights going into candidate hidden units. CC starts in the output phase and then shifts to input phase whenever it needs to recruit a new hidden unit. *Rounds* is an upper limit on the number of unit-installation cycles. Recommended values for *outlimit*, *inlimit*, and *rounds*, respectively, are 100, 100, and 25.

One optional parameter for train is *last-epoch*, with a default value of 999. Training stops after the last epoch. Another optional parameter is *restart*, with a default value of *nil* (the Lisp symbol for *false*). If restart is set to *t* (the Lisp symbol for *true*), then training restarts from the current point without re-initializing the network. This would be useful if you wanted to start with a pre-training period and then shift to a period with different training patterns. In such cases, the type of pattern would be 'change-train rather than 'train.

The training started by the call to *train* continues until victory is reached (i.e., the network's outputs are within *score-threshold* of their targets on all training patterns) or the network fails to learn (signaled by either a stagnation of error reduction or the passing of the maximum number of epochs).

After training is finished, we may well wish to inspect the connection weights that the network has learned. We can do this by printing the weights and the output weights. Here, in the last two lines of *run-xor*, we use the Lisp primitive *pprint*, which prints these weight vectors in a pretty fashion. The weights are those going into hidden units, specified in the *weights* vector. The output weights are those going into output units, as specified in the *output-weights* vector.

You should evaluate the *run- xor* procedure. Then call *run- xor*, by surrounding it with parentheses and evaluating the procedure call. You will see results in the Debug window, looking something like this. Your exact results will vary depending on the initial random weight values selected before training begins.

```
(run-xor)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Epoch 21: Out Stagnant 4 bits wrong, error 1.0000027722408333.

Epoch 45: In Stagnant.  Cor: 1.432170307716521
  Add unit 4: #(-10.61271393182322 22.847886835738677 21.87010546855839)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Victory at 53 epochs, 4 units, 1 hidden, Error 0.278689097398806.

#(NIL NIL NIL #(-10.61271393182322 22.847886835738677 21.87010546855839)
  NIL NIL NIL NIL NIL NIL NIL NIL)
#(#(-0.041129520455222945 -1.6179272883076545 -1.5031975558612576
    5.369657212646787 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0))
```

The first information printed out specifies the settings of a number of key parameters: sigmoid-prime-offset, weight-range, weight-multiplier, output-mu, output-epsilon, output-decay, output-patience, output-change-threshold, the input versions of those parameters, unit-type, output-unit-type, raw-error, the number of candidate hidden units in the pool, and the score-threshold. Definitions of each of these parameters can be found in the SDCC manual. For present purposes, it is not essential that you know those definitions.

```
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400
```

As training continues, every shift of phase is indicated by printing out the epoch number and the reason for the shift. For example,

```
Epoch 21: Out Stagnant 4 bits wrong, error 1.0000027722408333.
```

indicates that the initial output phase ended at epoch 21 because error reduction stagnated. At that point, all four outputs were incorrect and the network's total error was about 1.0. An epoch is a sweep through all of the training patterns. Error is the sum of the squared differences between training targets and output activations, across all training patterns.

The next piece of printed information

```
Epoch 45: In Stagnant.  Cor: 1.432170307716521
  Add unit 4: #(-10.61271393182322 22.847886835738677 21.87010546855839)
```

indicates that at epoch 45, the network shifted from input phase to output phase, because the correlations between candidate unit activations and network error stopped increasing. At that point the modified correlation between the best candidate and network error was about 1.43. This input phase ends with the installation of that best candidate hidden unit, called unit 4 after the bias unit and the two input units have been numbered. The input-side weights to that new hidden unit, from the bias unit, input 1, and input 2 are approximately -10.6, 22.8, and 21.9, respectively, as shown.

When the network reaches victory, the key parameter values are reprinted, followed by the victory declaration

```
Victory at 53 epochs, 4 units, 1 hidden, Error 0.278689097398806.
```

Reaching victory means that the network's actual output activations are within the score-threshold of 0.4 on all of the training patterns. The value of 0.4 is the default score-threshold for sigmoid output units, such as used here. Any output greater than 0.1 is considered correct for a target of 0.5, and any output less than -0.1 is considered correct for a target of -0.5. Any output between -0.1 and 0.1 is considered ambiguous if targets are exclusively -0.5 and 0.5, and training would continue. In this case, victory was reached at 53 epochs. At that point, there were four units in the network, not counting the outputs – bias, input 1, input 2, and 1 hidden unit. The network's error is now about 0.28.

Finally, the last two pretty-printed lines of code in the definition of *run- xor* display the weights the network has learned. The weights vector shows the weights going into hidden units. In this case, only the fourth unit is a hidden unit, so only the fourth sub-vector has actual numbers in it. They are, not surprisingly, the same values as were printed earlier, at the shift out of the first input phase, because input weights to hidden units are frozen once the unit is recruited.

```
#(NIL NIL NIL #(-10.61271393182322 22.847886835738677 21.87010546855839) NIL NIL NIL NIL
NIL NIL NIL NIL)
```

The next pretty-printed vector shows the weights going into the output units. Because there is only one output unit in this network, only the first sub-vector has actual numbers in it. The listing of output weights is in order, from the bias unit, the first input, the second input, and finally the first and only hidden unit.

```
#(#(-0.041129520455222945 -1.6179272883076545 -1.5031975558612576
    5.369657212646787 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0))
```

Close examination of these weight vectors reveals this network's solution to the XOR problem. Direct cross connections appear negligible, with most of the action contributed by the hidden unit. The hidden unit is negatively biased and excited by both inputs, and excites the output. With inputs of 0, 0 the hidden unit does not become active and neither does the output unit. When only one input is on, the hidden unit becomes active with a value near 0.5, and this is sufficient to activate the output unit. Input from the hidden unit in these cases is about 0.5 x 5.4 = 2.7, which is sufficient to overpower the negative input (about -1.6) coming from the bias unit and active input unit.

The more complex case is with both inputs on. This also results in an excitatory input from the hidden unit of about 2.7, but this turns out to be overpowered by the inhibitory net input coming from the two input units of about -3.1. Hence, the output unit is prevented from coming on. Thus, the cross connections are not really negligible in this solution; the cross connections coming from the two input units are critical in inhibiting the output when both inputs are on.

This first example reveals something of the subtlety of neural-network solutions. With relatively complex networks with more weights and training patterns, it becomes rather difficult to characterize solutions, particularly in view of the wide variability of weight patterns between different networks. In the seventh example, we consider another, more generally useful way to analyze network solutions, using Principle Component Analysis of network contributions.

Example 2. Continuous Recording of Network Error in the XOR Problem

In our second example, we redefine the *run-xor* procedure so that we can record the error every epoch, and the points at which any hidden units are installed. Such error recording may provide insight into how new hidden units influence the network's success. We first set four global variables, discussed in the manual, to enable the error recording. Setting *test* to t allows recording to take place every *test-interval*. Setting *test-interval* to 1 ensures that this recording is done every single epoch. Setting *record-train-errors* to *t* specifies that error on training patterns is recorded. Finally, setting *mark-hiddens-errors* to *t* ensures that the epochs at which hidden units are recruited are marked in this error file with an *H*. Note that errors are recorded only at output phase epochs. We don't bother to record error during input phase epochs because output weights remain frozen during input phases; this means that performance cannot change across input-phase epochs. So the epochs across which we are recording are output epochs, not total epochs. As noted in the manual's description of *record-train-errors*, the training errors are recorded in the global variable *train-errors*.

```
(defun run-xor ()
   "()
```

```
Run 1 xor net, look at the final weights, & record training errors."
   (setq *test* t
     *test-interval* 1
     *record-train-errors* t
     *mark-hiddens-errors* t)
   (setq *xor-patterns* '(((0.0 0.0) (-.5))
                          ((0.0 1.0) (.5))
                          ((1.0 0.0) (.5))
                          ((1.0 1.0) (-.5))))
   (set-patterns *xor-patterns* 'train)
   (train 100 100 25)
   (pprint *weights*)
   (pprint *output-weights*)
  (lists->file
   *train-errors*
   "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\examples\\train-errors")))
```

After the training is completed and the weights are printed, we call the procedure *lists->file* to save the error records in a file. Such saving is not critical here, where we are only running one network, but if we decide to run several networks, as in the next example, saving would be critical because as each new network is created, data from the previous network are destroyed. As described in the manual, *lists->file* takes two obligatory arguments and one optional argument. The obligatory arguments are the list to be saved and the file to save it in. In this case, the list to be saved is *train-errors* which, as specified in the manual, is a list of error for the training patterns, in reverse order by epoch. The specification of the file is essentially a hierarchically organized path name in the convention of your operating system. The path used here employs Windows path-naming conventions. The path name you use in your own work will, of course, reflect the naming conventions of your operating system and the way you organize your files.

```
(lists->file
 *train-errors*
 "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\examples\\train-errors")
```

As noted in the manual, *lists->file* reverses the list (in this case, *train-errors*) and prints it in the file (in this case, *train-errors*). This ensures that the error data in that file is in order from the first epoch to the last epoch. Note that all the folders involved in this path (e.g., *Documents and Settings* through *examples*) need to be established before anything can be stored there.

Redefine (evaluate) the *run-xor* procedure. Then run the newly defined procedure as before.

```
(run-xor)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Epoch 18: Out Stagnant 4 bits wrong, error 1.000544023058093.

Epoch 43: In Stagnant.  Cor: 1.4194484298631322
  Add unit 4: #(-7.76061728813323 -20.15966647448332 17.05134543898649)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Victory at 56 epochs, 4 units, 1 hidden, Error 0.30460007869471306.

#(NIL NIL NIL #(-7.76061728813323 -20.15966647448332 17.05134543898649) NIL NIL NIL NIL
NIL NIL NIL NIL)
#(#(0.8585127267145762 1.8349618374086814 -1.3618807499571515 4.224866067332551 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0))
DONE
```

This time, the results reveal a shift to input phase at epoch 18, a shift to output phase at epoch 43, and victory at epoch 56. The weights reveal a different solution than our first

example. In general, one network differs from another, even if trained in the same environment, because they start from different initial random weights (different heredity).

If we open the *train-errors* file, this is what we see:

```
1.082
1.079
1.068
1.05
1.03
1.026
1.023
1.018
1.012
1.008
1.006
1.005
1.005
1.004
1.003
1.001
1.001
1.001
H
0.896
0.893
0.885
0.867
0.831
0.768
0.666
0.555
0.425
0.414
0.379
0.361
0.333
0.305
```

These data are plotted, using a graphics program, in Figure 3. The plot shows a sharp drop in error after the hidden unit is recruited after epoch 18, indicating the critical importance of the new hidden unit in the network's solution. Normally, the *H* rows are removed before such plots can be drawn. In a later example, we'll see how to plot the recruitment epochs as an overlay.

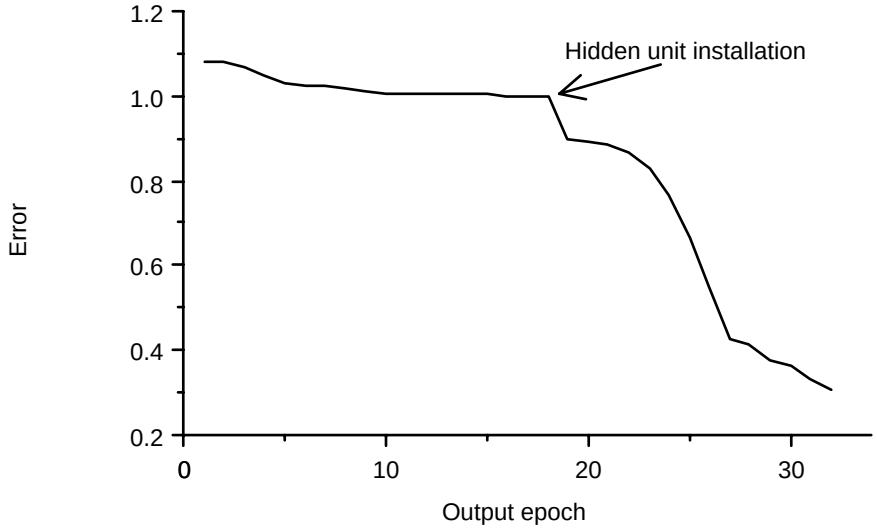

Figure 3. Error reduction on training patterns in an XOR network. A hidden unit was recruited after epoch 18.

Example 3. Running Several XOR Networks While Recording Training Error

In this example, we run several XOR networks, recording the training error every epoch, but not bothering to print the weights. We set the recording parameters and training patterns as in Example 2. Then we put the calls to *set-patterns*, *train*, and *lists-file* inside of a *do* loop. The new procedure, called *run-n-XOR* has a new parameter *n*, which specifies the number of networks to run.

```
(defun run-n-xor (n)
  "(n)
Run n xor nets, recording training errors."
  (setq *test* t
      *test-interval* 1
      *record-train-errors* t
      *mark-hiddens-errors* t)
  (setq *xor-patterns* '(((0.0 0.0) (-.5))
                         ((0.0 1.0) (.5))
                         ((1.0 0.0) (.5))
                         ((1.0 1.0) (-.5))))
  (do ((i 0 (+ i 1)))
      ((= i n) 'done)
    (seed-random)
    (terpri)
    (set-patterns *xor-patterns* 'train)
    (train 100 100 25)
    (lists->file
     *train-errors*
     (concatenate
         'string
       "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\examples\\train-errors"
       (princ-to-string i)))))
```

The *do* loop uses a counter *i*, which is initialized at 0 and incremented by 1 each time through the loop. When *i* reaches the value of *n*, the iteration stops and the word *'done* is printed on the screen.

At the start of each iteration through the *do* loop, we call the procedure *seed-random*. As noted in the manual, *seed-random* seeds the random-number generator using the internal clock of the computer. This ensures that your "random" number sequences are not identical every time you boot up Common Lisp. As such, you should probably use *seed-random* in realistic simulations. You can call it once, before the *do* loop, or at the start of each iteration as we do here.

The tricky part is the file naming in the call to the *lists->file* procedure. There is a separate file for each network, and each such file must have a unique name. This is accomplished by appending the value of *i* to the prefix *train-errors*. We do this with the Lisp primitive *concatenate*, specifying that the items to be concatenated are of the type *string*, that one item is the path name plus the prefix, and that the variable suffix is the value of the counter *i*. The little-known Lisp primitive *princ-to-string* converts the integer *i* to a string as required by concatenate. This creates files with the names *train-errors0*, *train-errors1*, etc., up to *n*-1.

Define and run the *run-n-xor* procedure in the usual way. Don't forget to call this procedure with the *n* argument specified, e.g., (run-n-xor 5) to run five networks. After the run, five new error files should be residing at the end of your file path.

Example 4. A Continuous Version of XOR

The first three examples concerned a very simple problem, XOR. We now move to a more complex problem to illustrate more realistic simulation techniques. The new problem, called continuous XOR, is a continuous version of the classical binary XOR problem. The input space for continuous XOR is divided into four quadrants, as shown in Figure 4. Training input values are incremented in steps of 0.1 starting from 0.1 up to 1.0, yielding 100 *x*, *y* input pairs. Values of *x* up to 0.55 combined with values of *y* above 0.55 produce a positive output target (0.5), as do values of *x* above 0.55 combined with values of *y* below 0.55. Input pairs in the other two quadrants yield a negative output target (represented by -0.5).

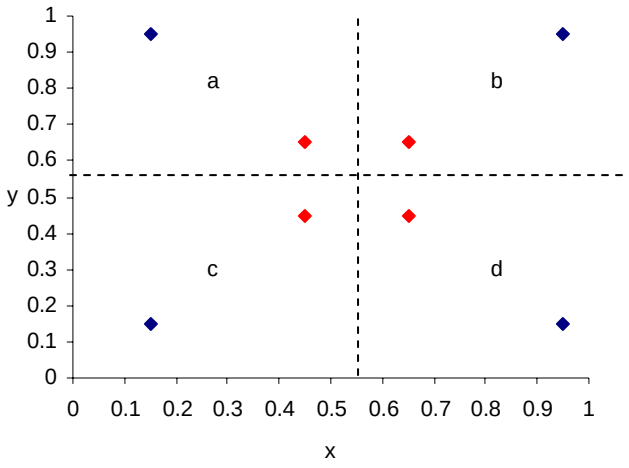

Figure 4. The continuous-XOR problem. The output unit should be on (target of 0.5) only in the *a* and *d* quadrants. The four points plotted (in blue) far from the quadrant boundaries should be easy to assign, whereas the four points plotted near the quadrant boundaries (in red) should be difficult to assign. Distance from quadrant boundaries is dealt with in Example 6.

It is typical in CC simulations for most of the new programming to involve generation of training and/or test patterns. The procedure *continuous-xor* creates training and test patterns. This procedure has two arguments: *start* and *step*. *Start* is the lowest input value and *step* is the increment used in generating the next input value.

```
(defun continuous-xor (start step)
  "(start step)
Make training or test patterns for continuous-xor."
  (do ((x start (float->decimals (+ x step) 2))
       (patterns nil))
      ((> x 1.0) (reverse patterns))
    (do ((y start (float->decimals (+ y step) 2)))
        ((> y 1.0))
      (let ((output (cond ((and (< x .55)
                                (< y .55))
                           -.5)
                          ((and (> x .55)
                                (< y .55))
                           .5)
                          ((and (< x .55)
                                (> y .55))
                           .5)
                          ((and (> x .55)
                                (> y .55))
```

```
                        -.5)))
          (inputs (list x y)))
      (setf patterns (cons (list inputs (list output)) patterns))))))
```

The procedure is structured as two *do* loops with one of them nested inside of the other. Both *do* loops have a natural ending point of 1.0 as the start value is incremented by the step value at each iteration. The outer *do* loop creates values for the first input *x*, and the inner *do* loop creates values for the second input *y*. Both *x* and *y* are initialized to the value of *start*, and are incremented by the value of *step*. Target output values for *x*, *y* pairs are created in the body of the inner *do* loop.

If we call *continuous-xor* with a start value of 0.1 and a step value of 0.1, this creates 100 training patterns. Evaluate *continuous-xor* and pretty-print a run of it with these arguments.

```
(pprint (continuous-xor .1 .1))

(((0.1 0.1) (-0.5)) ((0.1 0.2) (-0.5)) ((0.1 0.3) (-0.5)) ((0.1 0.4) (-0.5))
 ((0.1 0.5) (-0.5)) ((0.1 0.6) (0.5)) ((0.1 0.7) (0.5)) ((0.1 0.8) (0.5))
 ((0.1 0.9) (0.5)) ((0.1 1.0) (0.5)) ((0.2 0.1) (-0.5)) ((0.2 0.2) (-0.5))
 ((0.2 0.3) (-0.5)) ((0.2 0.4) (-0.5)) ((0.2 0.5) (-0.5)) ((0.2 0.6) (0.5))
 ((0.2 0.7) (0.5)) ((0.2 0.8) (0.5)) ((0.2 0.9) (0.5)) ((0.2 1.0) (0.5))
 ((0.3 0.1) (-0.5)) ((0.3 0.2) (-0.5)) ((0.3 0.3) (-0.5)) ((0.3 0.4) (-0.5))
 ((0.3 0.5) (-0.5)) ((0.3 0.6) (0.5)) ((0.3 0.7) (0.5)) ((0.3 0.8) (0.5))
 ((0.3 0.9) (0.5)) ((0.3 1.0) (0.5)) ((0.4 0.1) (-0.5)) ((0.4 0.2) (-0.5))
 ((0.4 0.3) (-0.5)) ((0.4 0.4) (-0.5)) ((0.4 0.5) (-0.5)) ((0.4 0.6) (0.5))
 ((0.4 0.7) (0.5)) ((0.4 0.8) (0.5)) ((0.4 0.9) (0.5)) ((0.4 1.0) (0.5))
 ((0.5 0.1) (-0.5)) ((0.5 0.2) (-0.5)) ((0.5 0.3) (-0.5)) ((0.5 0.4) (-0.5))
 ((0.5 0.5) (-0.5)) ((0.5 0.6) (0.5)) ((0.5 0.7) (0.5)) ((0.5 0.8) (0.5))
 ((0.5 0.9) (0.5)) ((0.5 1.0) (0.5)) ((0.6 0.1) (0.5)) ((0.6 0.2) (0.5))
 ((0.6 0.3) (0.5)) ((0.6 0.4) (0.5)) ((0.6 0.5) (0.5)) ((0.6 0.6) (-0.5))
 ((0.6 0.7) (-0.5)) ((0.6 0.8) (-0.5)) ((0.6 0.9) (-0.5)) ((0.6 1.0) (-0.5))
 ((0.7 0.1) (0.5)) ((0.7 0.2) (0.5)) ((0.7 0.3) (0.5)) ((0.7 0.4) (0.5))
 ((0.7 0.5) (0.5)) ((0.7 0.6) (-0.5)) ((0.7 0.7) (-0.5)) ((0.7 0.8) (-0.5))
 ((0.7 0.9) (-0.5)) ((0.7 1.0) (-0.5)) ((0.8 0.1) (0.5)) ((0.8 0.2) (0.5))
 ((0.8 0.3) (0.5)) ((0.8 0.4) (0.5)) ((0.8 0.5) (0.5)) ((0.8 0.6) (-0.5))
 ((0.8 0.7) (-0.5)) ((0.8 0.8) (-0.5)) ((0.8 0.9) (-0.5)) ((0.8 1.0) (-0.5))
 ((0.9 0.1) (0.5)) ((0.9 0.2) (0.5)) ((0.9 0.3) (0.5)) ((0.9 0.4) (0.5))
 ((0.9 0.5) (0.5)) ((0.9 0.6) (-0.5)) ((0.9 0.7) (-0.5)) ((0.9 0.8) (-0.5))
 ((0.9 0.9) (-0.5)) ((0.9 1.0) (-0.5)) ((1.0 0.1) (0.5)) ((1.0 0.2) (0.5))
 ((1.0 0.3) (0.5)) ((1.0 0.4) (0.5)) ((1.0 0.5) (0.5)) ((1.0 0.6) (-0.5))
 ((1.0 0.7) (-0.5)) ((1.0 0.8) (-0.5)) ((1.0 0.9) (-0.5)) ((1.0 1.0) (-0.5)))
```

We can make 81 fresh test patterns with a start of 0.14 and a step of 0.1. Again, we can pretty-print these patterns for visual examination.

```
(pprint (continuous-xor .14 .1))

(((0.14 0.14) (-0.5)) ((0.14 0.24) (-0.5)) ((0.14 0.34) (-0.5))
 ((0.14 0.44) (-0.5)) ((0.14 0.54) (-0.5)) ((0.14 0.64) (0.5))
 ((0.14 0.74) (0.5)) ((0.14 0.84) (0.5)) ((0.14 0.94) (0.5))
 ((0.24 0.14) (-0.5)) ((0.24 0.24) (-0.5)) ((0.24 0.34) (-0.5))
 ((0.24 0.44) (-0.5)) ((0.24 0.54) (-0.5)) ((0.24 0.64) (0.5))
 ((0.24 0.74) (0.5)) ((0.24 0.84) (0.5)) ((0.24 0.94) (0.5))
 ((0.34 0.14) (-0.5)) ((0.34 0.24) (-0.5)) ((0.34 0.34) (-0.5))
 ((0.34 0.44) (-0.5)) ((0.34 0.54) (-0.5)) ((0.34 0.64) (0.5))
 ((0.34 0.74) (0.5)) ((0.34 0.84) (0.5)) ((0.34 0.94) (0.5))
 ((0.44 0.14) (-0.5)) ((0.44 0.24) (-0.5)) ((0.44 0.34) (-0.5))
 ((0.44 0.44) (-0.5)) ((0.44 0.54) (-0.5)) ((0.44 0.64) (0.5))
 ((0.44 0.74) (0.5)) ((0.44 0.84) (0.5)) ((0.44 0.94) (0.5))
 ((0.54 0.14) (-0.5)) ((0.54 0.24) (-0.5)) ((0.54 0.34) (-0.5))
 ((0.54 0.44) (-0.5)) ((0.54 0.54) (-0.5)) ((0.54 0.64) (0.5))
 ((0.54 0.74) (0.5)) ((0.54 0.84) (0.5)) ((0.54 0.94) (0.5))
 ((0.64 0.14) (0.5)) ((0.64 0.24) (0.5)) ((0.64 0.34) (0.5))
 ((0.64 0.44) (0.5)) ((0.64 0.54) (0.5)) ((0.64 0.64) (-0.5))
 ((0.64 0.74) (-0.5)) ((0.64 0.84) (-0.5)) ((0.64 0.94) (-0.5))
 ((0.74 0.14) (0.5)) ((0.74 0.24) (0.5)) ((0.74 0.34) (0.5))
 ((0.74 0.44) (0.5)) ((0.74 0.54) (0.5)) ((0.74 0.64) (-0.5))
 ((0.74 0.74) (-0.5)) ((0.74 0.84) (-0.5)) ((0.74 0.94) (-0.5))
 ((0.84 0.14) (0.5)) ((0.84 0.24) (0.5)) ((0.84 0.34) (0.5))
 ((0.84 0.44) (0.5)) ((0.84 0.54) (0.5)) ((0.84 0.64) (-0.5))
 ((0.84 0.74) (-0.5)) ((0.84 0.84) (-0.5)) ((0.84 0.94) (-0.5))
 ((0.94 0.14) (0.5)) ((0.94 0.24) (0.5)) ((0.94 0.34) (0.5))
 ((0.94 0.44) (0.5)) ((0.94 0.54) (0.5)) ((0.94 0.64) (-0.5))
```

```
 ((0.94 0.74) (-0.5)) ((0.94 0.84) (-0.5)) ((0.94 0.94) (-0.5)))
```

Now, we define a procedure called *run-continuous-xor*, which runs one network on this problem, recording training and test error every output epoch.

```
(defun run-continuous-xor ()
  "Run 1 continuous-xor net with training & test patterns.
Record training & test error every output epoch."
  (seed-random)
  (setq *test* t
      *test-interval* 1
      *record-train-errors* t
      *record-test-errors* t
      *mark-hiddens-errors* t
      *path* "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\ex4\\")
  (set-patterns (continuous-xor .1 .1) 'train)
  (set-patterns (continuous-xor .14 .1) 'test)
  (train 100 100 25)
  (lists->file
   *train-errors*
   (concatenate 'string
     *path*
     "train errors"))
  (lists->file
   *test-errors*
   (concatenate 'string
     *path*
     "test errors"))
  (pprint *weights*)
  (pprint *output-weights*))
```

After seeding the random number generator and setting the stage for error recording, we set up the training and test patterns using the *set-patterns* procedure. Then we train the network, store the training errors and test errors in separate files, and pretty-print the weights. Notice how we set the global variable *path* to a string of hierachically-arranged folders, and then concatenate the file names *train errors* and *test errors* to this path when storing the errors.

Evaluate this procedure and run it. Your output should look something like this, although not exactly because of different random initial weight values.

```
(run-continuous-xor)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Epoch 8: Out Stagnant 100 bits wrong, error 25.097742735387595.

Epoch 58: In Stagnant.  Cor: 0.9854064915632649
  Add unit 4: #(-6.9769581060672925 20.795267889900714 -20.157520028836117)
Epoch 131: Out Stagnant 22 bits wrong, error 12.46732225204742.

Epoch 195: In Stagnant.  Cor: 1.3542458668722617
  Add unit 5: #(10.759446906752558 -10.226126985656636 -32.966996462205856
-18.637516903438243)
Epoch 286: Out Stagnant 10 bits wrong, error 5.418693963224867.

Epoch 333: In Stagnant.  Cor: 1.411229436045613
  Add unit 6: #(-17.656485190823904 -0.21706868210631017 36.07013146887336
0.030856067460773476 3.5661884813585902)
Epoch 405: Out Stagnant 5 bits wrong, error 3.730326182834624.

Epoch 477: In Stagnant.  Cor: 2.098741020964659
  Add unit 7: #(-1.9314917548277719 -35.99862961109121 6.827510624590333
-0.10984117477012573 -22.89323939178627 10.317764239851444)
Epoch 577: Out Timeout  1 bits wrong, error 1.4787860398336765.

Epoch 624: In Stagnant.  Cor: 2.822709121297379
  Add unit 8: #(0.7193214544433657 0.09288276507985095 -16.1037912071861
-26.89706522923551 2.983804137885525 -1.0906990089940536 -11.353040805199257)
Epoch 652: Out Stagnant 1 bits wrong, error 1.3238166296164118.

Epoch 726: In Stagnant.  Cor: 5.28084541860263
  Add unit 9: #(-5.7447081195418725 18.18376869199651 2.684494661973217
```

```
11.804690002938854 0.4544239551219428 -4.442023825127082 13.802416710309801
11.974653628220283)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Victory at 740 epochs, 9 units, 6 hidden, Error 0.41838290457485217.

#(NIL NIL NIL #(-6.9769581060672925 20.795267889900714 -20.157520028836117)
  #(10.759446906752558 -10.226126985656636 -32.966996462205856
    -18.637516903438243)
  #(-17.656485190823904 -0.21706868210631017 36.07013146887336
    0.030856067460773476 3.5661884813585902)
  #(-1.931491754827719 -35.99862961109121 6.827510624590333
    -0.10984117477012573 -22.89323939178627 10.317764239851444)
  #(0.7193214544433657 0.09288276507985095 -16.1037912071861 -26.89706522923551
    2.983804137885525 -1.0906990089940536 -11.353040805199257)
  #(-5.7447081195418725 18.18376869199651 2.684494661973217 11.804690002938854
    0.4544239551219428 -4.442023825127082 13.802416710309801
    11.974653628220283)
  NIL NIL NIL)
#(#(4.3324000828661235 1.4192397029198422 -5.041302310490373 2.233004553772843
    -11.02426938470583 -10.678808934227446 22.323593555396894 4.491040362234271
    5.820827652476648 0.0 0.0 0.0))
```

This network recruits six hidden units, reaching victory at 740 epochs, confirming that continuous XOR is considerably more complex than binary XOR. Basically, continuous XOR is more complex because of its continuous range of input values and the consequent large number of patterns it must master.

Examination of the weight vectors reveals a complex pattern that is resistant to simple stories. In a later example, we use contribution analysis on such networks to understand their solutions.

The recorded error data on training and test patterns can be found in the files *train errors* and *test errors*, respectively. These data may be copied and pasted into a graphics program and plotted, as we have done in Figure 5. For every epoch with an *H* indicating installation of a hidden unit, the error value for the training patterns is copied and pasted into a third data column and the corresponding row containing the *H* is deleted. This allows an overlay plot, in which symbols such as the triangle shown in Figure 5 may be scatter plotted, fitting just over the line representing error on training patterns. The scalloped error curves seen in Figure 5 are typical of CC networks – after error stagnation, recruitment of a new hidden unit allows a steeper reduction of error. The fact that error on test patterns starts to exceed training error indicates that the network may be over-learning the training patterns in a way that does not generalize well to test patterns. Some neural network practitioners believe that it is time to stop training when that happens. The problem is that such stopping means that test problems are, in fact, contributing to training even if not used directly in error reduction. In other respects, though, the curve for test error closely matches that for training error, indicating good generalization.

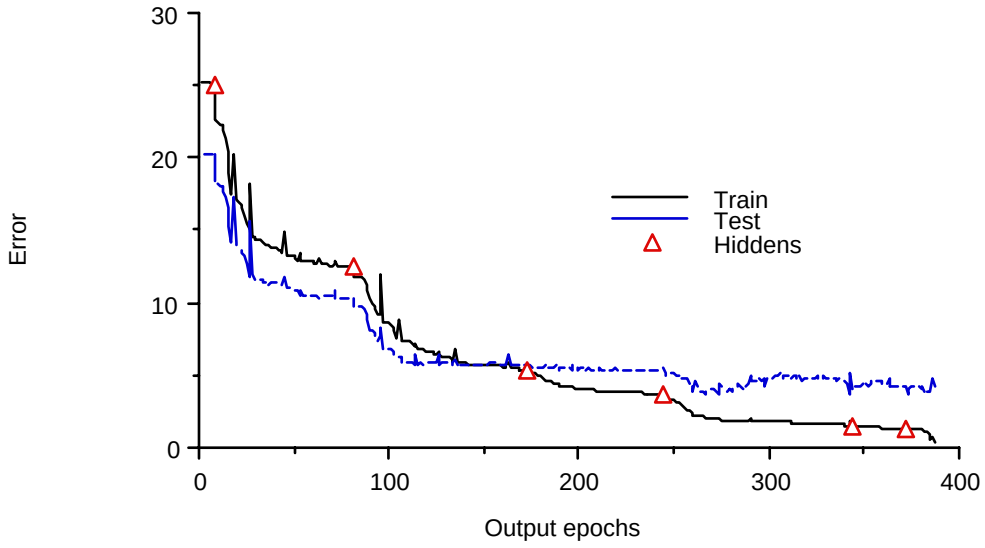

Figure 5. Training and test error for a network learning the continuous XOR problem.

Example 5. Continuous XOR with SDCC

In this fifth example, we use the SDCC option to create a variety of network topologies on the continuous XOR problem. Although the code for the *continuous-xor* procedure remains unchanged from the previous example, a few changes are required to create a procedure called *run-continuous-xor-sdcc*.

```
(defun run-continuous-xor-sdcc (n)
  "Run n continuous-xor nets using sdcc with training & test patterns.
Record training & test error every output epoch."
  (setq *test* t
      *test-interval* 1
      *record-train-errors* t
      *record-test-errors* t
      *mark-hiddens-errors* t
      *sdcc* t
      *path* "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\ex5\\")
  (set-patterns (continuous-xor .1 .1) 'train)
  (set-patterns (continuous-xor .14 .1) 'test)
  (do ((i 0 (1+ i))
       (structures nil (cons (reverse *structure*)
                             structures)))
      ((= i n)
       (progn
         (pprint *weights*)
         (pprint *output-weights*)
         (pprint (reverse structures))))
    (seed-random)
    (terpri)
    (train 100 100 25)
    (lists->file
     *train-errors*
     (concatenate 'string
       *path*
       "train-errors"
       (princ-to-string i)))
    (lists->file
     *test-errors*
     (concatenate
         'string
       *path*
       "test-errors"
       (princ-to-string i)))))
```

Changes include the following. First, the global variable *sdcc* is set to *t* to activate the SDCC option. Second, a slightly different path is used to segregate the stored error files. Third, it can be interesting to store the different network structures created by SDCC. This is done by introducing into the *do* loop a new variable called *structures*, initialized to *nil* and updated each iteration by *consing* the reverse of the global variable *structure* onto *structures*.

```
(cons (reverse *structure*)
      structures)
```

Then, as the *do* loop finishes, we pretty-print the reverse of the *structures* list.

```
(pprint (reverse structures))
```

In this example, we call the new procedure to run three SDCC networks. In these runs, we do not change the global variable *descendant-factor* from its default setting of 0.8. Without this penalty on correlation size, typically not many sibling units would be recruited because of the extra degrees of freedom that descendant units provide. The default value of 0.8 limits network depth without compromising generalization ability.

```
(run-continuous-xor-sdcc 3)
```

The output from this run includes all the usual information as seen in Example 4, supplemented by the hidden-unit structures. For brevity, I reproduce here only the victory declarations and the list of hidden-unit structures.

```
Victory at 988 epochs, 11 units, 8 hidden, Error 0.3536952.
Victory at 671 epochs, 8 units, 5 hidden, Error 0.79904115.
Victory at 948 epochs, 10 units, 7 hidden, Error 0.23127589.
 ((2 2 2 1 1) (4 1) (5 2))
```

The first network has 8 hidden units, with 2, 2, 2, 1, and 1 units in each of 5 successive layers. The second and third networks each have 2 layers of hidden units, layered as 4 and 1 units, and 5 and 2 units, respectively. Careful modelers can note from the connection-weight printouts that the usual cascaded connection weights are absent for hidden units residing in the same layer. Thus, compared to standard CC, SDCC produces leaner networks with fewer layers. Otherwise, we have not noticed substantial functional differences between the two algorithm options.

For description of the SDCC algorithm, see:

Baluja, S., & Fahlman, S. E. (1994). *Reducing network depth in the cascade-correlation learning architecture*. Technical Report CMU-CS-94-209, School of Computer Science, Carnegie Mellon University.

For an example of a psychological SDCC simulation, see:

Shultz, T. R. (2006). Constructive learning in the modeling of psychological development. In Y. Munakata & M. H. Johnson (Eds.), *Processes of change in brain and cognitive development: Attention and performance XXI* (pp. 61-86). Oxford: Oxford University Press.

Example 6. Perceptual Effects in Continuous XOR

One of the nice features of CC simulations has been the fact that perceptual effects, which are known to be quite pervasive in cognitive-developmental phenomena, emerge naturally. Although there is no known psychological literature on the continuous XOR

problem, it is easy to imagine such perceptual effects. It might be expected that *x*, *y* inputs near the decision boundaries would be more difficult to classify than those that lie far from decision boundaries. Here we define a new procedure *run-continuous-xor-perceptual* that uses two sets of hand-designed test patterns that can be used to examine this hypothesis. A list of these two pattern sets, plotted in Figure 4, is set to be the value of the global variable *multi-test-patterns*. The first of these test sets is supposed to be easier than the second. The global variables *record-train-errors*, *record-test-errors*, *mark-hiddens-errors*, and *sdcc* are all set to *nil* (which represents *false* in Common Lisp). This would not be necessary except that these variables would retain their values of *true* from our previous examples. Here we also set *test-interval* to 10, indicating that we are recording only every 10th epoch. There is no single set of test patterns as earlier. Errors are stored in the global variable *multi-errors*, so that is the list that gets stored in the call to *lists->file*. We drop the printing of connection weights for this simulation.

```
(defun run-continuous-xor-perceptual ()
  "Run 1 continuous-xor net with multiple test patterns.
Record error every 10th output epoch."
  (seed-random)
  (setq *test* t
      *test-interval* 10
      *record-train-errors* nil
      *record-test-errors* nil
      *mark-hiddens-errors* nil
      *sdcc* nil
      *path* "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\ex6\\"
      *record-multi-errors* t
      *multi-test-patterns* '((((.15 .95) (.5))
                              ((.95 .95) (-.5))
                              ((.15 .15) (-.5))
                              ((.95 .15) (.5)))
                             (((.45 .65) (.5))
                              ((.65 .65) (-.5))
                              ((.45 .45) (-.5))
                              ((.65 .45) (.5)))))
  (set-patterns (continuous-xor .1 .1) 'train)
  (train 100 100 25)
  (lists->file
   *multi-errors*
   (concatenate 'string
     *path*
     "multi-errors")))
```

Define this new procedure by evaluating it. Then run it. Results might look something like this:

```
(run-continuous-xor-perceptual)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Epoch 21: Out Stagnant 100 bits wrong, error 25.00929012687813.

Epoch 77: In Stagnant.  Cor: 0.89393458177116
  Add unit 4: #(-5.6820622151902365 18.06368779103951 -18.77950900304182)
Epoch 137: Out Stagnant 23 bits wrong, error 12.632020309773509.

Epoch 192: In Stagnant.  Cor: 1.2614247063841277
  Add unit 5: #(10.61265348985958 -10.457117020827615 -21.082733665407698
-5.886684178189322)
Epoch 292: Out Timeout  10 bits wrong, error 4.153006725385906.

Epoch 363: In Stagnant.  Cor: 1.3875251308979086
  Add unit 6: #(-3.9479888564925645 0.5247370082970195 33.72350440564817
0.8941623930657228 30.248010571677696)
Epoch 415: Out Stagnant 2 bits wrong, error 1.75838373517906.

Epoch 479: In Stagnant.  Cor: 2.783107389319063
  Add unit 7: #(2.1124239115133063 -52.33882873981923 13.221419933857296
1.405505641947961 -25.033243371480157 7.911792880117402)
Epoch 579: Out Timeout  2 bits wrong, error 0.48477507935110087.
```

```
Epoch 608: In Stagnant.  Cor: 0.8900629130808858
  Add unit 8: #(-1.545111727618694 -0.0283699447633665 2.785722809503757
-0.28939780468149273 5.724728177951501 67.03469487203213 -8.611037911758903)
SigOff 0.10, WtRng 1.00, WtMul 1.00
OMu 2.00, OEps 0.3500, ODcy 0.0001, OPat 8, OChange 0.010
IMu 2.00, IEps 1.0000, IDcy 0.0000, IPat 8, IChange 0.030
Utype :SIGMOID, Otype :SIGMOID, RawErr NIL, Pool 8, ScTh 0.400

Victory at 674 epochs, 8 units, 5 hidden, Error 0.35442711040502967.
DONE
```

The error data in the *multi-errors* file are plotted in Figure 6. As predicted, they show earlier generalization to the easier patterns far from the decision boundaries than to the more difficult patterns close to the decision boundaries. Eventually, the network shows perfect generalization to both sets of test patterns.

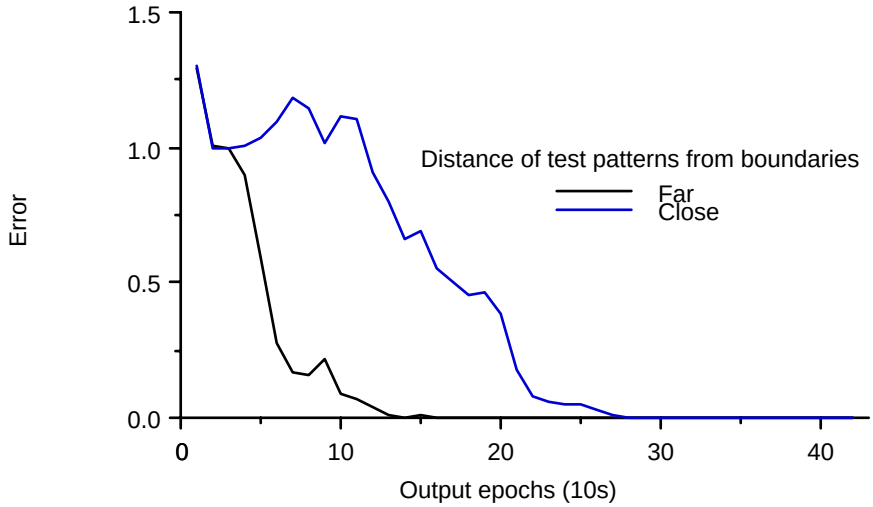

Figure 6. Perceptual effects for a network learning the continuous-XOR problem.

Example 7. Analyzing Knowledge Representations in a Continuous-XOR Network

It is typically a challenge to determine how a network has learned to represent its knowledge of complex tasks such as continuous XOR. One technique that has worked well for us is doing a Principal Component Analysis (PCA) of network contributions. Contributions are products of sending activations and connection weights going into output units. They are particularly suitable for analyzing the knowledge representations in networks, such as those created by CC, that utilize direct cross connections as well as cascaded pathways.

In this example we perform a PCA of the contributions in a single continuous-XOR network at the end of training, using the following procedure, called *knowledge-continuous-xor*.

```
(defun knowledge-continuous-xor ()
  "()
Run 1 continuous-xor net for a final contribution analysis."
  (seed-random)
  (setq *test* nil
      *record-train-errors* nil
      *record-test-errors* nil
      *mark-hiddens-errors* nil
```

```
    *record-multi-errors* nil
    *record-train-contributions* nil
    *sdcc* nil)
(set-patterns (continuous-xor .1 .1) 'train)
(train 100 100 25)
(setq *record-train-contributions* t)
(test-epoch)
(save-contributions
 *train-contributions*
 "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\ex7\\"
 'train
 1))
```

After seeding the random number generator, this procedure sets all previously-used global variables to their default value of *nil*. This would, of course, be unnecessary for runs that occurred after a fresh booting of Lisp. Because we want to record contributions only after training, it is wise to set *record-train-contributions* to *nil* also. This prevents any recording prior to the end of training, just in case *test* and *test-interval* had settings that enabled such testing during training. Then we establish the training patterns and train as usual.

```
(set-patterns (continuous-xor .1 .1) 'train)
(train 100 100 25)
```

Following training, we set *record-train-contributions* to *true*, and call the procedures *test-epoch* and *save-contributions*. If we had used test patterns, we could have recorded contributions on those test patterns in addition or instead.

```
(setq *record-train-contributions* t)
(test-epoch)
(save-contributions
 *train-contributions*
 "c:\\Documents and Settings\\tom\\models\\sdcc\\problems\\ex7\\"
 'train
 1)
```

As explained in the manual, *test-epoch* runs any training and test patterns and records whatever data are needed. The procedure *test-epoch* is called whenever network testing is performed. When using *test* and *test-interval*, *test-epoch* is called automatically. Here, we call it directly.

Also as noted in the manual, the procedure *save-contributions* has four arguments (the file holding the contributions, a file pathway, the type of contribution, and the index of the network in case of multiple networks). The type is either 'train or 'test. Contributions are saved in separate files for each epoch. When using multiple networks, the network argument to *save-contributions* should be updated with a counter in whatever procedure calls *save-contributions*.

We won't bother looking at all of the output, except to note that this particular network learned in 674 epochs and recruited 5 hidden units.

```
Victory at 674 epochs, 8 units, 5 hidden, Error 0.4962046297507869.
```

You may want to evaluate *knowledge-continuous-xor* and run it yourself.

The contributions file, called *net1 train contribs1* looks like this. There are eight columns, corresponding to the bias unit, the two inputs, and the five hidden units in that order. The 100 rows refer to the 100 training patterns, also in the order created by the *continuous-xor* procedure.

```
19.241 -5.94 -16.07 -5.893 5.149 -5.134 -5.037 -2.063
19.241 -5.94 -14.463 -5.902 5.149 -5.134 -5.036 -2.063
19.241 -5.94 -12.856 -5.903 5.149 -5.134 -5.019 -2.063
19.241 -5.94 -11.249 -5.904 5.149 -5.134 -3.999 -2.063
19.241 -5.94 -9.642 -5.904 5.149 -5.133 3.836 -2.063
```

```
19.241 -5.94 -8.035 -5.904 5.149 -5.133 5.015 -2.063
19.241 -5.94 -6.428 -5.904 5.149 -5.133 5.036 -2.063
19.241 -5.94 -4.821 -5.904 5.149 -5.132 5.037 -2.063
19.241 -5.94 -3.214 -5.904 5.149 -5.132 5.037 -2.063
19.241 -5.94 -1.607 -5.904 5.149 -5.132 5.037 -2.063
19.241 -5.346 -16.07 -5.831 5.149 -5.125 -5.037 -2.063
19.241 -5.346 -14.463 -5.891 5.149 -5.124 -5.037 -2.063
19.241 -5.346 -12.856 -5.902 5.149 -5.123 -5.034 -2.063
19.241 -5.346 -11.249 -5.903 5.149 -5.122 -4.845 -2.063
19.241 -5.346 -9.642 -5.904 5.149 -5.12 0.554 -2.063
19.241 -5.346 -8.035 -5.904 5.149 -5.119 4.913 -2.063
19.241 -5.346 -6.428 -5.904 5.149 -5.117 5.035 -2.063
19.241 -5.346 -4.821 -5.904 5.149 -5.115 5.037 -2.063
19.241 -5.346 -3.214 -5.904 5.149 -5.113 5.037 -2.063
19.241 -5.346 -1.607 -5.904 5.147 -5.11 5.037 -2.063
19.241 -4.752 -16.07 -5.402 5.149 -5.071 -5.037 -2.063
19.241 -4.752 -14.463 -5.816 5.149 -5.065 -5.037 -2.063
19.241 -4.752 -12.856 -5.889 5.149 -5.058 -5.036 -2.063
19.241 -4.752 -11.249 -5.901 5.149 -5.05 -5.004 -2.063
19.241 -4.752 -9.642 -5.903 5.149 -5.042 -3.275 -2.063
19.241 -4.752 -8.035 -5.904 5.149 -5.032 4.349 -2.063
19.241 -4.752 -6.428 -5.904 5.148 -5.022 5.025 -2.063
19.241 -4.752 -4.821 -5.904 5.147 -5.011 5.036 -2.063
19.241 -4.752 -3.214 -5.904 5.142 -4.999 5.037 -2.063
19.241 -4.752 -1.607 -5.904 5.124 -4.99 5.037 -2.063
19.241 -4.158 -16.07 -3.066 5.149 -4.731 -5.037 -2.063
19.241 -4.158 -14.463 -5.302 5.149 -4.717 -5.037 -2.063
19.241 -4.158 -12.856 -5.797 5.149 -4.683 -5.036 -2.063
19.241 -4.158 -11.249 -5.886 5.149 -4.64 -5.031 -2.063
19.241 -4.158 -9.642 -5.901 5.148 -4.593 -4.672 -2.063
19.241 -4.158 -8.035 -5.903 5.146 -4.543 2.088 -2.063
19.241 -4.158 -6.428 -5.904 5.139 -4.494 4.972 -2.063
19.241 -4.158 -4.821 -5.904 5.115 -4.458 5.036 -2.063
19.241 -4.158 -3.214 -5.904 5.035 -4.477 5.037 -2.063
19.241 -4.158 -1.607 -5.904 4.778 -4.647 5.037 -2.063
19.241 -3.564 -16.07 2.28 5.149 -2.849 -5.037 -2.063
19.241 -3.564 -14.463 -2.634 5.149 -2.918 -5.037 -2.063
19.241 -3.564 -12.856 -5.184 5.148 -2.874 -5.037 -2.063
19.241 -3.564 -11.249 -5.775 5.145 -2.738 -5.035 -2.063
19.241 -3.564 -9.642 -5.882 5.135 -2.592 -4.959 -2.063
19.241 -3.564 -8.035 -5.9 5.102 -2.508 -1.827 -2.063
19.241 -3.564 -6.428 -5.903 4.994 -2.656 4.624 -2.063
19.241 -3.564 -4.821 -5.904 4.648 -3.416 5.021 -2.063
19.241 -3.564 -3.214 -5.904 3.648 -4.7 5.034 -2.063
19.241 -3.564 -1.607 -5.904 1.412 -5.122 5.032 -2.063
19.241 -2.97 -16.07 5.214 5.149 1.557 -5.037 2.063
19.241 -2.97 -14.463 2.741 5.149 1.623 -5.037 2.063
19.241 -2.97 -12.856 -2.164 5.147 1.525 -5.037 2.063
19.241 -2.97 -11.249 -5.043 5.11 1.424 -5.036 2.063
19.241 -2.97 -9.642 -5.748 4.955 1.01 -5.023 2.063
19.241 -2.97 -8.035 -5.877 4.485 -0.675 -4.782 -2.063
19.241 -2.97 -6.428 -5.899 3.203 -4.139 -4.364 -2.063
19.241 -2.97 -4.821 -5.903 0.639 -5.116 -4.87 -2.063
19.241 -2.97 -3.214 -5.904 -2.29 -5.136 -5.016 -2.063
19.241 -2.97 -1.607 -5.904 -4.086 -5.136 -5.009 -2.063
19.241 -2.376 -16.07 5.802 5.149 4.311 -5.037 2.063
19.241 -2.376 -14.463 5.328 5.149 4.372 -5.037 2.063
19.241 -2.376 -12.856 3.163 5.147 4.396 -5.037 2.063
19.241 -2.376 -11.249 -1.66 5.06 4.265 -5.036 2.063
19.241 -2.376 -9.642 -4.878 3.586 0.039 -5.036 2.063
19.241 -2.376 -8.035 -5.716 0.106 -5.095 -5.037 -2.063
19.241 -2.376 -6.428 -5.871 -2.859 -5.136 -5.037 -2.063
19.241 -2.376 -4.821 -5.898 -4.348 -5.136 -5.037 -2.063
19.241 -2.376 -3.214 -5.903 -4.896 -5.136 -5.037 -2.063
19.241 -2.376 -1.607 -5.904 -5.072 -5.136 -5.036 -2.063
19.241 -1.782 -16.07 5.889 5.149 4.989 -5.037 2.063
19.241 -1.782 -14.463 5.82 5.147 5.002 -5.037 2.063
19.241 -1.782 -12.856 5.424 5.142 5.012 -5.037 2.063
19.241 -1.782 -11.249 3.544 5.078 5.005 -5.036 2.063
19.241 -1.782 -9.642 -1.128 2.608 0.869 -5.037 2.063
19.241 -1.782 -8.035 -4.684 -4.026 -5.136 -5.037 -2.063
19.241 -1.782 -6.428 -5.678 -4.938 -5.136 -5.037 -2.063
19.241 -1.782 -4.821 -5.865 -5.091 -5.136 -5.037 -2.063
19.241 -1.782 -3.214 -5.897 -5.132 -5.136 -5.037 -2.063
19.241 -1.782 -1.607 -5.903 -5.144 -5.136 -5.037 -2.063
19.241 -1.188 -16.07 5.902 5.141 5.111 -5.037 2.063
19.241 -1.188 -14.463 5.892 5.123 5.113 -5.037 2.063
19.241 -1.188 -12.856 5.834 5.06 5.113 -5.037 2.063
19.241 -1.188 -11.249 5.505 4.799 5.103 -5.037 2.063
19.241 -1.188 -9.642 3.884 2.812 4.428 -5.037 2.063
19.241 -1.188 -8.035 -0.577 -4.369 -5.135 -5.037 -2.063
```

```
19.241 -1.188 -6.428 -4.459 -5.121 -5.136 -5.037 -2.063
19.241 -1.188 -4.821 -5.631 -5.145 -5.136 -5.037 -2.063
19.241 -1.188 -3.214 -5.857 -5.148 -5.136 -5.037 -2.063
19.241 -1.188 -1.607 -5.896 -5.149 -5.136 -5.037 -2.063
19.241 -0.594 -16.07 5.903 5.032 5.131 -5.037 2.063
19.241 -0.594 -14.463 5.902 4.768 5.129 -5.037 2.063
19.241 -0.594 -12.856 5.894 3.973 5.114 -5.037 2.063
19.241 -0.594 -11.249 5.846 1.991 4.665 -5.037 2.063
19.241 -0.594 -9.642 5.572 -1.368 -4.183 -5.037 -2.063
19.241 -0.594 -8.035 4.184 -4.392 -5.127 -5.037 -2.063
19.241 -0.594 -6.428 -0.015 -5.126 -5.133 -5.037 -2.063
19.241 -0.594 -4.821 -4.199 -5.148 -5.134 -5.037 -2.063
19.241 -0.594 -3.214 -5.575 -5.149 -5.133 -5.037 -2.063
19.241 -0.594 -1.607 -5.847 -5.149 -5.133 -5.037 -2.063
```

Obviously, this matrix of contributions is too complex to interpret directly. Thus, we subject it to PCA in order to reduce its dimensionality by taking advantage of any correlations among contributions across training patterns. This would be done outside of Lisp in a statistics program. We typically use SPSS for this purpose, analyzing the covariance matrix, sometimes using a *scree* test to decide how many components to retain, or in this case retaining only those eigenvalues greater than 1 x the mean eigenvalue, and applying a *varimax* rotation. We leave out the bias (first) column because it has no variation.

Here is a scree plot showing the eigenvalue for each component. Only the first three components are required and they account for 94.9% of the variance in network contributions.

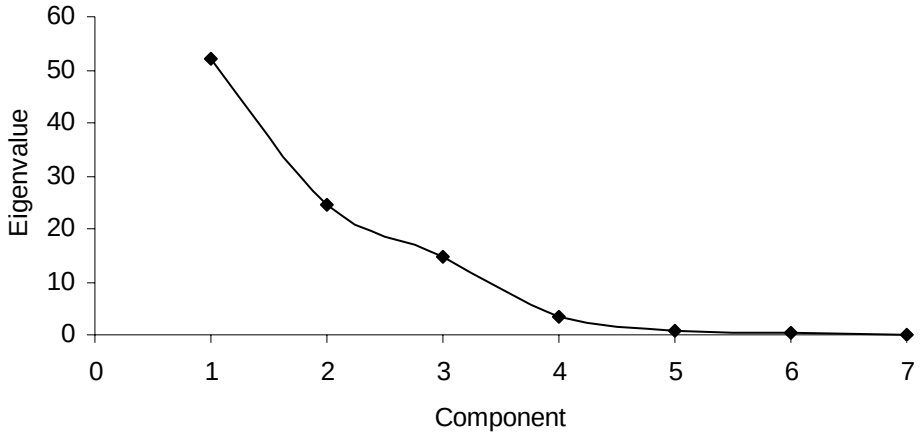

Figure 7. Scree plot of eigenvalues.

Here are the rotated and rescaled (i.e., standardized) loadings produced by the analysis.

	Component		
Contribution	1	2	3
Input 1	0.549	**-0.750**	-0.190
Input 2	-0.414	-0.499	**0.742**
Hidden 1	**0.941**	-0.070	-0.218
Hidden 2	0.232	**0.956**	0.031
Hidden 3	**0.937**	0.087	-0.185
Hidden 4	-0.189	0.397	**0.880**
Hidden 5	**0.889**	0.056	-0.187

The PCA reveals that the percent of total variance in the contributions explained by each of the three components is 44.5, 27.1, and 21.1, respectively. The loadings show that (a) hidden units 1, 3, and 5 load heavily on component 1, (b) input 1 and hidden unit 2 load heavily on component 2, and (c) input 2 and hidden unit 4 load heavily on component 3. Often it is necessary to plot rotated component scores to interpret the particular job that the different components are performing. Generally, rotated component scores can be saved from the PCA. Such scores are plotted in Figure 8 for a different network. As in that network, the present network's first component separates the patterns with positive outputs from those with negative outputs; the second component classifies patterns according to their *x* value; and the third component does so with respect to their *y* value.

In other problem domains, it may be possible to identify the main job of each component by examining the inputs that load on each component.

As explained more fully in the manual, there are four different strategies available for doing contribution analysis of CC and SDCC networks. They can be done at the end of the run as here, at every test interval, at the end of every output phase, and at epochs selected after the completion of the run. The latter three techniques may provide a developmental trace of the network's knowledge representations.

Further discussion of our contribution analysis techniques can be found in:

Shultz, T. R., Oshima-Takane, Y., & Takane, Y. (1995). Analysis of unstandardized contributions in cross connected networks. In D. Touretzky, G. Tesauro, & T. K. Leen, (Eds). *Advances in Neural Information Processing Systems 7* (pp. 601-608). Cambridge, MA: MIT Press.

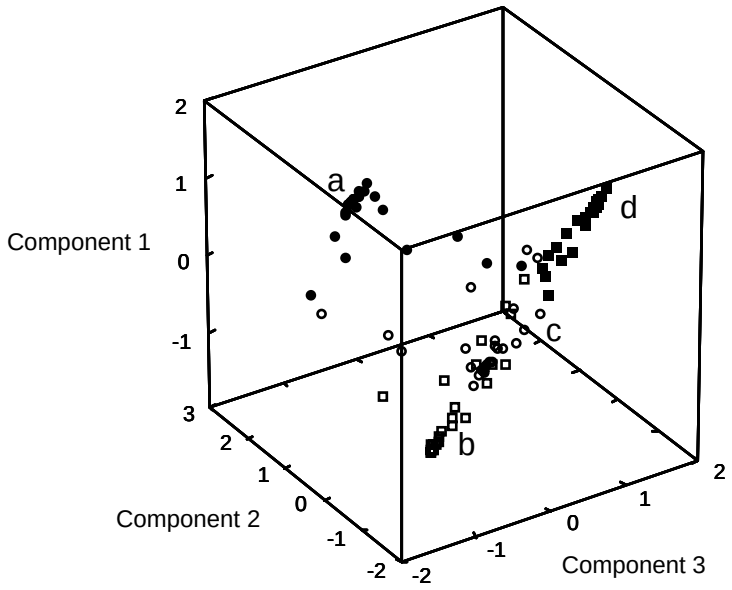

Figure 8. Rotated component scores for a continuous-XOR network. Component scores for the *x*,*y* input pairs in quadrant *a* are labeled with black circles, those from quadrant *b* with white squares, those from quadrant *c* with white circles, and those from quadrant *d* with black squares. The network's task is to distinguish pairs from quadrants *a* and *d* (the black shapes) from pairs from quadrants *b* and *c* (the white shapes). Some of the white shapes appear black because they are so densely packed, but actually all of the truly black shapes are relatively high in the cube.

Thomas R. Shultz

14 June 2006