SANDIA REPORT

SAND2004-4420
Unlimited Release
Printed December 2004

Identifying generalities in data sets using periodic Hopfield networks—Initial status report

Hamilton Link & Alex Bäcker

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the
United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

This page left intentionally blank.

Identifying generalities in data sets using periodic Hopfield networks—Initial status report

Hamilton Link
Dept. 5632
helink@sandia.gov

Alex Bäcker
Dept. 9212
alex@caltech.edu

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

We present a novel class of dynamic neural networks that is capable of learning, in an unsupervised manner, attractors that correspond to generalities in a data set. Upon presentation of a test stimulus, the networks follow a sequence of attractors that correspond to subsets of increasing size or generality in the original data set. The networks, inspired by those of the insect antennal lobe, build upon a modified Hopfield network in which nodes are periodically suppressed, global inhibition is gradually strengthened, and the weight of input neurons is gradually decreased relative to recurrent connections. This allows the networks to converge on a Hopfield network's equilibrium within each suppression cycle, and to switch between attractors in between cycles. The fast mutually reinforcing excitatory connections that dominate dynamics within cycles ensures the robust error-tolerant behavior that characterizes Hopfield networks. The cyclic inhibition releases the network from what would otherwise be stable equilibriums or attractors. Increasing global inhibition and decreasing dependence on the input leads successive attractors to differ, and to display increasing generality. As the network is faced with stronger inhibition, only neurons connected with stronger mutually excitatory connections will remain on; successive attractors will consist of sets of neurons that are more strongly correlated, and will tend to select increasingly generic characteristics of the data. Using artificial data, we were able to identify configurations of the network that appeared to produce a sequence of increasingly general results. The next logical steps are to apply these networks to suitable real-world data that can be characterized by a hierarchy of increasing generality and observe the network's performance. This report describes the work, data, and results, the current understanding of the results, and how the work could be continued. The code, data, and preliminary results are included and are available as an archive.

Table of Contents

Introduction

The intent of this work was to develop modified Hopfield networks, in which selected nodes' values are periodically suppressed, to retrieve a series of increasingly "general" images related to an input. A good generalization would be one that contains elements present in all or most of a set of training images related to the input image. A possible additional constraint would be generalizations that are self-consistent, i.e. they contain only elements that are found together in some training image.

Hopfield networks are fully-connected associative neural networks, as described in many neurocomputing textbooks (Hecht-Neilson 1990, Hertz 1991). When trained on a set of node values such as an image, they are able to retrieve a matching image from a damaged or noisy version.

Approach

In Lisp, we implemented a Hopfield network data structure with several control variables for the network's computation. Specifically, the configuration of the network included five variables: `suppress-and-hold`, `tanh-of-dotproduct`, `suppress-to-value`, `source-image-weight`, and `floating-threshold`. These variables allowed us to adjust important properties of the neural network and observe their impact on the network's ability to create sequences of reasonable generalizations.

`suppress-and-hold` is a Boolean variable that alters the nature of node suppression in the network. The network periodically has particular nodes' values forced to a suppression value, and when this variable is true this forced value is maintained by those nodes until the network converges to a steady state with the suppression. Then the suppression is relaxed, and the network is allowed to converge again, after which time the next image in the sequence is recorded. When this variable is false, the suppressed nodes' values are set to the suppression value, the network is allowed to converge, and an image is recorded. In the latter case, the network proceeds in each

round from a partially intact representation of the previous element of the growing sequence. In the former the network always proceeds from a state that has been reached effectively using only a portion of the network.

`tanh-of-dotproduct` is a Boolean variable that controls the transfer function of the network. If it is false, the transfer function is simply the dot product of the vector of incoming node values with the trained weight vector. If it is true, the tanh function is applied to the dot product's result. Using tanh is a common approach to bounding the value of network nodes.

`suppress-to-value` determines the value nodes are kept at while being suppressed, or it is the value suppressed nodes are set with in between rounds (depending on the value of `suppress-and-hold`). For our tests it was considered a discrete variable with three possible values (), -1, -0.5, and 0.

`source-image-weight` determines the weight that the sample image provided to the network is given when going from one sequence result to the next. It is included in a weighted sum with the network state after suppression is relaxed and before the network is allowed to run. For our tests it was considered a discrete variable with five possible values, 0, 0.5, 0.75, 0.88, and 0.95. This weight decays geometrically as each sequence element is taken.

`floating-threshold` determines the cut-off point for nodes to produce a positive value given their transfer function result. For our tests it was considered a discrete variable with ten possible values, 0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.30, 0.35, 0.4, and 0.5. This value increases linearly over the course of a sequence.

As an experiment, we prepared a training set of nine images (as described in the next section). These were used to create a Hopfield network using the standard symmetric Hebbian learning rule. In any one trial run of the network, one of the training images has random noise added to it, and it is given as input to the network. The result is recorded, nodes are suppressed based on the controlling variables, and another result is recorded. This continues until a sequence of fifteen output images have been collected. The output images are then analyzed and their logical description (also described in the next section) is written into a file. Taking all possible control variable values gives 600 configurations of the network, and each configuration was run 100 times with new random noise added to the same image. This produced a file of 60,000 output sequences from which statistics were generated.

Data

Although real image data would have provided a very natural and intuitive sense of what good generalizations would be, we decided against using images as a training set. We initially pre-processed a set of images into reasonable images, cropping and scaling a few dozen pictures into 128x128 pixel thumbnail images, converting them into grayscale and using a variable threshold to turn these into 1-bit black and white images that were

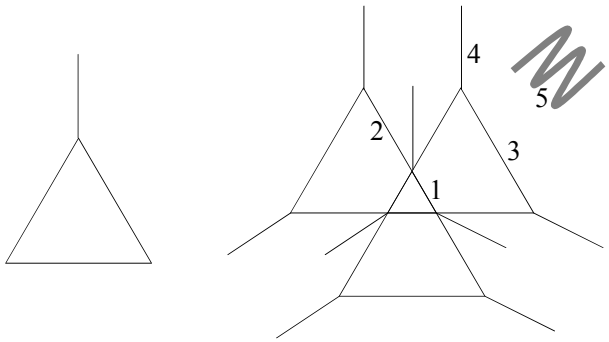

Figure 1. A sample training image, and the logical relationship of the nine training images, with overlap (1), part (2), full (3), tail (4), and background (5) components making the network.

approximately recognizable. However, the images proved so idiosyncratic that it would have been difficult to come up with feature extractors to get node values for neural network input. The training set we resorted to for our initial experiment was ultimately a set of three partially overlapping triangles, each with a "tail" to distinguish one from the other (Figure 1). The lengths of the sides of the triangles (in the number of corresponding nodes in the network), the amount of overlap, and the lengths of the tails, and the quantity of nodes corresponding to empty background were all variables fixed arbitrarily for the duration of this experiment. The total size of the network was 430 nodes: 100 background nodes and 90 for each complete triangle, plus 10 for each tail, with an overlap of 10 nodes at the overlapping triangle corners (i.e. 100 + 3*90 + 9*10 - 3*10 = 430). The artificial nature of the images allowed us to see what a periodically suppressed neural net was capable of without jumping to biased conclusions based on semantically loaded data sets.

The output sequences returned by the network were lists of 16 bipolar-valued vectors of 430 elements (the noisy input image and 15 results), and as such were somewhat inscrutable in their raw form. After some test runs and some initial observations, an analysis routine was written to compare these output vectors with the logical structure of the "triangles" and produce symbolic descriptions instead. This analysis routine looked for partial and complete shapes corresponding to the input triangle, and produced one of the values `triangle-with-tail`, `triangle`, `triangle-with-arc`, `little-triangle`, `shared-bits`, `shared-bit`, `damaged-triangle`, and `damaged-triangle-with-arc` (Figure 2). The result from the 60,000 trials was then sequences of 15 of these symbols, plus the symbol `noisy-image` (generated as a validation measure). The trials' sequences were composed of runs of these symbols, with any symbol represented in only one run in a given trial (for example, a sequence was not found that went from `triangle` to `shared-bits` and then returned to `triangle`) and as the configurations varied the lengths and starting positions of these runs varied as well.

Several statistics were computed for each configuration from the symbolic sequence results:

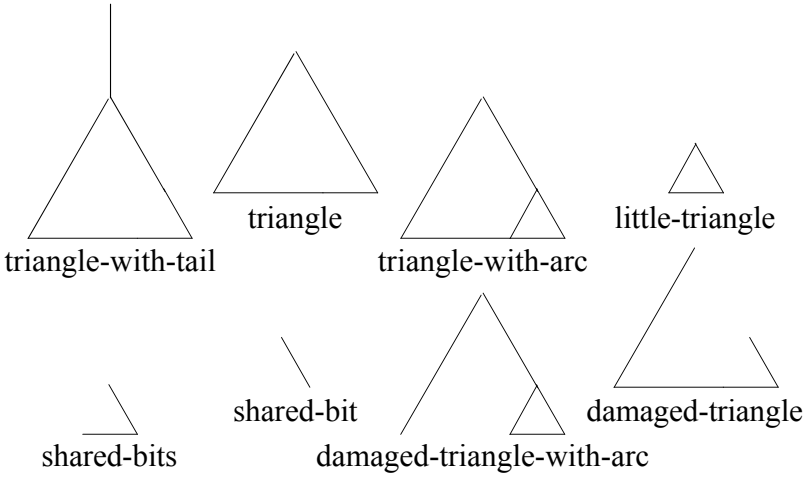

Figure 2. Observed patterns and their labels. Note that the damaged patterns' missing edges varied. These are examples of those patterns.

Average (duplicate-free) sequence length. This was done by removing duplicate elements from the symbolic sequences, and computing the average resulting length over 100 trials for each configuration.

Frequency of recognizing each pattern. This was done by taking the list of possible symbols (pattern names), and computing the percentage of trials for a configuration in which that symbol appeared.

Average initial position of each pattern. For each symbol recognized at any time by a configuration, the position (from 1 to 15) of the start of that symbol's run was averaged across the trials for that configuration in which it appeared at all.

Frequency of each symbol appearing in each position. A more detailed view of the previous statistic, this was computed by keeping counters for each position and each symbol, and incrementing the appropriate counter as the trials for a configuration were scanned.

Frequency of each pattern sequence. When duplicates were removed (but order retained for the remainder), we found only 17 distinct pattern lists (see Appendix A). Not including noisy-image, these varied in length from one pattern to four. For each configuration, the number of trials in which each of these pattern lists appeared was counted.

Results

After 100 trials for each of 600 configurations were run and the unreduced symbolic sequences saved into a file, this file was processed to extract the statistics described above. First, a list of the 17 unique reduced sequences was created from the file, making the list shown in Appendix A. The ordering of these sequences was rearranged a few times as the

Sequence	% of all trials
(triangle △ little-triangle ◬)	37.8%
(triangle △ shared-bits ◺)	25.3%
(triangle △ shared-bits ◺ little-triangle ◬)	22.4%
(triangle △)	9.5%
(triangle △ shared-bits ◺ shared-bit ◹)	2.4%
(triangle △ triangle-with-arc ◮ little-triangle ◬)	1.7%
(triangle △ triangle-with-arc ◮)	0.3%

Table 1. The set of commonly appearing pattern sequences.

analysis proceeded, until the list shown was arrived at based on the total number of appearances and the number of appearances in any particular trial. After the statistics from the previous section had been generated on the results, the two that we focused on were average sequence length and pattern sequence frequencies for each configuration.

Based on pattern sequence frequencies within each trial, the sequences observed can be divided fairly well into two sets. The first seven make up the set of more commonly identified sequences (Table 1). These did not necessarily appear across all 600 configurations, but they did appear with regularity in certain configurations, and appeared in significant frequencies in those configurations. The remaining ten configurations occurred in only one or two trials for any configuration (and never more than four), and each represented less than 0.2% of the results of all trials. The one exceptional sequence in this latter set of rare sequences was `(noisy-image triangle shared-bit)`, which appeared in 37% of the trials for one configuration and almost not at all otherwise, making up a total of 54 (about 0.0008%) of the 60,000 trial results.

Conclusions

Given the data we created for this initial test it would be premature to draw many conclusions from our results. However, some points of interest should be noted, in order to focus further study in applying a periodically suppressed Hopfield network to real-world data.

The different pattern sequences that appear may or may not be appropriate answers to the need for a network that can generalize from a sample image. The original triangles with tails were never recovered, but the triangle common to three triangles with tails was invariably the first thing the network identified. This could be considered a reasonable generalization. The `shared-bits` pattern might be the next logical step, considering that a less specific description for the set of three overlapping triangles with tails is "the images containing the partial edges in `shared-bits`." A comparable description could be created for the six images that share a single `shared-bit`. Considering these descriptions makes the `little-triangle` pattern somewhat suspect, however, as it contains components that are never seen together in any training example.

The pattern sequence currently seen as most desirable from this point of view is `(noisy-image triangle shared-bits shared-bit)`. Although it appears in only 2.4%

`suppress-and-hold`	nil	nil	nil
`tanh-of-dotproduct`	nil	t	t
`suppress-to-value`	0.0	0.0	0.0
`source-image-weight`	0.0-0.5	0.88	0.95
`floating-threshold`	0.35-0.40	0.05-0.50	0.05-0.50
Pattern sequence frequency	100%	60.1%	19.5%

Table 2. Configurations producing (triangle shared-bits shared-bit)

of all trials, these appearances are concentrated in a few sets of configurations, and the most common pattern sequence appearing alongside this one is the slightly shorter pattern `(noisy-image triangle shared-bits)`. Possibly if the raw sequences had all been carried further, this distinction would not have arisen. The specific configurations which exhibited these results are shown in Table 2, along with the local frequencies of these patterns in those configurations. These regions are also highlighted in Appendix A.

Continuing the Work

First and foremost, it would be interesting to apply the technique to real-world data sets. If reasonable generalizations exist in those data sets, it would be interesting to observe which generalizations the network gravitates towards, and what applications would benefit from being able to tease out this information. One possible source of data would be animal feature sets; the ability to create a network that travels from a specific animal up through a recognizable taxonomy would be particularly promising.

From a technical perspective, there are several noteworthy issues that would be the logical place for continuing this work. The pattern sequences were run for fifteen rounds in each trial, whether or not the network stopped returning an image. This null image was included in some of the statistics, giving configurations that terminated early a biased length. Also, sequences that took more rounds to shift from one image to the next may have had their pattern sequence truncated. A more appropriate test for the statistics of interest would be to run the trials for each configuration until they returned the null image, and not include the `noisy-image` or null image in any statistical computations.

Once the statistical processing was improved, particular configurations should be analyzed to determine why they are producing their particular results. As noted, the series `(triangle shared-bits shared-bit)` is particularly interesting from the standpoint of generalization, and configurations that generate this or similar series should be focused upon. It may also be worth looking into the source of the `little-triangle` and `triangle-with-arc` as internally inconsistent results from the point of view of the training set. Finally, some time could be spent confirming or disconfirming the hypothesis that the more rare sequences are outliers caused by sensitivity to noise.

References

Hecht-Nielson, R. *Neurocomputing*. Addison-Wesley, 1990.

Hertz, J., A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*, vol. I of *Santa Fe Institute Studies in the Sciences of Complexity*. Westview Press, Santa Fe, NM, 1991.

Appendix A: Results

This appendix contains the list of sequences generated by the network, what the elements in those sequences mean, and statistics generated from the raw network results, specifically the average sequence length, and the frequencies in each configuration of the most common sequences.

These are all the abbreviated sequences generated during testing. Series of repeated elements have been shortened to a single element, meaning that (triangle triangle shared-bit) and (triangle shared-bit shared-bit) would be considered equivalent. They have been reordered based on their frequency across all 600 configurations (with 100 trials per configuration). The length of these frequencies does not include the input, noisy-image, so for example configurations with an average result length of one are those that always produced the fourth sequence below, (noisy-image triangle). The first seven occur in the most significant quantities of trials in varying numbers of configurations. Those frequencies (number of occurrences out of 100 trial results) are displayed in the last section of the results. The remaining ten sequences are all currently considered anomalies, due to their infrequency and in most cases due to their inclusion of specific "damaged" images.

(noisy-image triangle little-triangle)
(noisy-image triangle shared-bits)
(noisy-image triangle shared-bits little-triangle)
(noisy-image triangle)
(noisy-image triangle shared-bits shared-bit)
(noisy-image triangle triangle-with-arc little-triangle)
(noisy-image triangle triangle-with-arc)

(noisy-image triangle shared-bit)
(noisy-image triangle damaged-triangle little-triangle)
(noisy-image triangle damaged-triangle shared-bits)
(noisy-image triangle damaged-triangle-with-arc little-triangle)
(noisy-image triangle damaged-triangle)
(noisy-image triangle damaged-triangle shared-bits little-triangle)
(noisy-image triangle little-triangle shared-bits)
(noisy-image triangle damaged-triangle shared-bits shared-bit)
(noisy-image triangle damaged-triangle shared-bit)
(noisy-image triangle damaged-triangle-with-arc)

Average number of levels identified vs. 5 control variables

```
a-<Hold inhibition and converge between rounds (nil/t)>
 -<Use tanh on dotproduct (nil/t)>

          [suppress to -1]            [suppress to -0.5]           [suppress to 0.0]

 -> Weight of source image (geometrically decays each round), as (0.0 0.5 0.75 0.88 0.95)
|
v Decision threshold (increases linearly over each round),
  as (nil 0.05 0.1 0.15 0.2 0.25 0.30 0.35 0.4 0.5)

? (setf a-nil-nil (make-array '(10 3 5) :displaced-to (coerce (subseq x 0 150) 'vector)))
#3a(((2.0  2.02 2.22 2.42 1.35)   (2.0  2.46 2.74 1.87 1.0 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.02 2.05 2.28 1.33)   (2.0  2.09 2.53 2.0  1.0 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.02 2.04 2.57 1.89)   (2.0  2.37 2.59 2.32 1.0 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.01 2.2  2.72 2.34)   (2.0  2.58 2.65 2.37 1.0 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.0  2.35 2.67 2.65)   (2.0  2.84 2.69 2.5  1.07)   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.01 2.62 2.68 2.2 )   (2.0  2.93 2.75 2.61 1.2 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.02 2.71 2.7  2.15)   (2.0  2.88 2.72 2.7  1.62)   (2.0  2.0  1.96 1.0  1.0 ))
    ((2.0  2.01 2.8  2.69 2.03)   (2.0  2.88 2.74 2.91 1.85)   (4.0  4.0  3.63 1.32 1.0 ))
    ((2.0  2.02 2.84 2.8  2.04)   (2.0  2.92 2.84 2.95 1.98)   (4.0  4.0  3.16 2.11 1.02))
    ((2.0  2.05 2.96 2.86 2.02)   (2.0  2.82 2.76 2.04 2.0 )   (3.0  3.0  3.07 3.48 1.52)))

? (setf a-nil-t (make-array '(10 3 5) :displaced-to (coerce (subseq x 150 300) 'vector)))
#3a(((2.0  2.04 2.18 2.41 1.3 )   (2.0  2.44 2.7  1.85 1.0 )   (1.0  1.0  1.0  1.0  1.0 ))
    ((2.0  2.13 2.88 2.01 2.07)   (2.0  2.86 2.3  2.0  2.01)   (3.0  3.0  2.94 3.61 2.17))
    ((2.0  2.17 2.89 2.03 2.09)   (2.0  2.84 2.37 2.0  2.01)   (3.0  3.0  2.96 3.6  2.18))
    ((2.0  2.23 2.84 2.01 2.04)   (2.0  2.83 2.31 2.0  2.0 )   (3.0  3.0  2.96 3.58 2.14))
    ((2.0  2.19 2.92 2.0  2.02)   (2.0  2.84 2.38 2.0  2.0 )   (3.0  3.0  2.97 3.54 2.17))
    ((2.0  2.16 2.9  2.03 2.08)   (2.0  2.85 2.34 2.01 2.0 )   (3.0  3.0  2.96 3.69 2.18))
    ((2.0  2.18 2.9  2.0  2.07)   (2.0  2.89 2.36 2.0  2.01)   (3.0  3.0  2.95 3.6  2.21))
    ((2.0  2.24 2.93 2.01 2.05)   (2.0  2.83 2.29 2.0  2.0 )   (3.0  3.0  2.92 3.61 2.27))
    ((2.0  2.2  2.91 2.01 2.04)   (2.0  2.9  2.29 2.01 2.03)   (3.0  3.0  2.96 3.6  2.2 ))
    ((2.0  2.18 2.92 2.03 2.04)   (2.0  2.82 2.28 2.0  2.01)   (3.0  3.0  2.94 3.6  2.18)))

? (setf a-t-nil (make-array '(10 3 5) :displaced-to (coerce (subseq x 300 450) 'vector)))
#3a(((2.0  2.04 2.22 2.37 1.61)   (2.0  2.05 2.2  2.44 1.47)   (2.0  2.04 2.21 2.51 1.61))
    ((2.0  2.01 2.11 2.29 1.71)   (2.0  2.03 2.08 2.39 1.77)   (2.0  2.03 2.07 2.24 1.61))
    ((2.0  2.02 2.17 2.83 2.37)   (2.0  2.01 2.07 2.85 2.27)   (2.0  2.03 2.07 2.84 2.23))
    ((2.0  2.0  2.28 2.89 2.59)   (2.0  2.0  2.17 2.87 2.65)   (2.0  2.02 2.24 2.86 2.65))
    ((2.0  2.01 2.47 2.84 2.71)   (2.0  2.02 2.47 2.89 2.66)   (2.0  2.0  2.31 2.84 2.77))
    ((2.0  2.01 2.6  2.91 2.79)   (2.0  2.01 2.63 2.86 2.8 )   (2.0  2.0  2.49 2.9  2.85))
    ((2.0  2.02 2.74 2.82 2.41)   (2.0  2.02 2.75 2.87 2.41)   (2.0  2.06 2.71 2.91 2.47))
    ((2.0  2.02 2.88 2.84 2.16)   (2.0  2.02 2.9  2.81 2.09)   (2.0  2.04 2.89 2.84 2.17))
    ((2.0  2.06 2.9  2.91 2.03)   (2.0  2.04 2.91 2.88 2.0 )   (2.0  2.04 2.93 2.86 2.01))
    ((2.0  2.07 2.96 2.9  2.0 )   (2.0  2.04 2.97 2.95 2.01)   (2.0  2.05 2.97 2.9  2.03)))

? (setf a-t-t (make-array '(10 3 5) :displaced-to (coerce (subseq x 450 600) 'vector)))
#3a(((2.0  2.04 2.22 2.38 1.61)   (2.0  2.09 2.2  2.38 1.6 )   (2.0  2.05 2.2  2.43 1.52))
    ((2.0  2.18 2.89 2.39 2.02)   (2.0  2.19 2.96 2.42 2.04)   (2.0  2.19 2.93 2.37 2.07))
    ((2.0  2.26 2.97 2.35 2.06)   (2.0  2.21 2.95 2.29 2.06)   (2.0  2.29 2.95 2.22 2.07))
    ((2.0  2.17 2.96 2.35 2.08)   (2.0  2.19 2.95 2.42 2.05)   (2.0  2.12 2.93 2.33 2.04))
    ((2.0  2.16 2.94 2.38 2.1 )   (2.0  2.18 2.92 2.35 2.05)   (2.0  2.23 2.96 2.37 2.06))
    ((2.0  2.11 2.92 2.49 2.04)   (2.0  2.13 2.97 2.39 2.05)   (2.0  2.17 2.95 2.41 2.06))
    ((2.0  2.08 2.94 2.39 2.03)   (2.0  2.24 2.9  2.33 2.04)   (2.0  2.18 2.91 2.37 2.02))
    ((2.0  2.15 2.91 2.36 2.02)   (2.0  2.25 2.92 2.32 2.09)   (2.0  2.14 2.95 2.38 2.07))
    ((2.0  2.16 2.96 2.4  2.02)   (2.0  2.19 2.94 2.35 2.06)   (2.0  2.17 2.96 2.37 2.04))
    ((2.0  2.23 2.89 2.38 2.05)   (2.0  2.17 2.92 2.28 2.04)   (2.0  2.15 2.93 2.34 2.01)))
```

Number of observations each of sequence 1-7, for each configuration.

These three columns correspond to the first three blocks above (a-nil-nil)

(100)	(100)	(100)
(98 2)	(54 45)	(100)
(78 21)	(26 74)	(100)
(58 40)	(40 27 33)	(100)
(75 10 15)	(100)	(100)
(100)	(100)	(100)
(98 2)	(91 9)	(100)
(95 5)	(47 53)	(100)
(72 27)	(93 1 2 3)	(100)
(27 3 1 68)	(100)	(100)
(100)	(100)	(100)
(98 2)	(63 37)	(100)
(96 4)	(41 58)	(100)
(43 54)	(68 31)	(100)
(48 5 14 28)	(100)	(100)
(100)	(100)	(100)
(99 1)	(42 57)	(100)
(80 19)	(35 65)	(100)
(28 70)	(63 37)	(100)
(34 13 42 9)	(100)	(100)
(100)	(100)	(100)
(100)	(16 84)	(100)
(65 34)	(31 69)	(100)
(33 66)	(50 48)	(100)
(29 62 2)	(7 93)	(100)
(100)	(100)	(100)
(99 1)	(7 93)	(100)
(38 61)	(25 74)	(100)
(32 66)	(39 60)	(100)
(80 19)	(18 81)	(100)
(100)	(100)	(100)
(98 1)	(12 88)	(100)
(29 71)	(28 72)	(95 4)
(30 70)	(30 68)	(100)
(85 13)	(59 39)	(100)
(100)	(100)	(100)
(99 1)	(12 88)	(100)
(20 79)	(26 74)	(63)
(31 69)	(10 88)	(31 68)
(97 2)	(79 16)	(100)
(100)	(100)	(100)
(98 2)	(8 92)	(100)
(16 83)	(16 84)	(84 16)
(20 80)	(5 95)	(87 10)
(96 1)	(98 2)	(2 98)
(100)	(100)	(100)
(95 5)	(18 82)	(100)
(4 96)	(24 75)	(93 7)
(14 85)	(96 2)	(49 50)
(98)	(100)	(52 48)

These three columns correspond to the second three blocks above (a-nil-t)

(100)	(100)	(100)
(96 4)	(56 44)	(100)
(82 18)	(30 67)	(100)
(59 38)	(1 40 25 34)	(100)
(79 9 12)	(100)	(100)
(100)	(100)	(100)
(87 13)	(14 86)	(100)
(12 88)	(71 28)	(92 7 1)
(99 1)	(100)	(38 61)
(94 5)	(99)	(79 2 19)
(100)	(100)	(100)
(83 17)	(16 83)	(100)
(11 89)	(63 35)	(90 6 2)
(97 1)	(100)	(39 60)
(91 8)	(99)	(78 2 20)
(100)	(100)	(100)
(77 23)	(17 83)	(100)
(16 84)	(69 31)	(93 5)
(99)	(100)	(42 58)
(96 3)	(100)	(82 2 16)
(100)	(100)	(100)
(81 19)	(16 83)	(100)
(8 92)	(1 61 38)	(92 5 2)
(100)	(100)	(45 54)
(98 1)	(100)	(81 1 18)
(100)	(100)	(100)
(84 16)	(15 83)	(100)
(10 90)	(66 34)	(90 6 2)
(97)	(99)	(29 67)
(92 7)	(100)	(82 17)
(100)	(100)	(100)
(82 17)	(11 89)	(100)
(10 90)	(64 35)	(87 7 2)
(100)	(100)	(38 60)
(93 5)	(99)	(79 21)
(100)	(100)	(100)
(76 24)	(17 83)	(100)
(7 93)	(71 28)	(88 9 1)
(99)	(100)	(39 61)
(95 5)	(100)	(73 27)
(100)	(100)	(100)
(80 20)	(10 90)	(100)
(9 91)	(71 27)	(94 4)
(99)	(99)	(40 60)
(96 3)	(97)	(80 20)
(100)	(100)	(100)
(82 15)	(18 79)	(100)
(8 90)	(72 28)	(87 9 2)
(97)	(100)	(40 60)
(96 2)	(99)	(82 18)

These three columns correspond to the third three blocks above (a-t-nil)

```
(100                 ) (100                 ) (100                 )
( 96              4  ) ( 95              5  ) ( 96              4  )
( 78             22  ) ( 80             20  ) ( 79             20  )
( 63             36  ) ( 56             42  ) ( 49             49  )
(          63    24 12) (          67    14 19) (          63    23 13)

(100                 ) (100                 ) (100                 )
( 99              1  ) ( 97              3  ) ( 97              2  )
( 89             10  ) ( 92              6  ) ( 93              7  )
( 71    28           ) ( 61    37           ) ( 76    24           )
( 17 12 20 50        ) ( 22 13 21 44        ) ( 18 11 14 55        )

(100                 ) (100                 ) (100                 )
( 98              1  ) ( 99              1  ) ( 97              3  )
( 83    16           ) ( 93     7           ) ( 93     7           )
( 17    83           ) ( 15    83           ) ( 16    84           )
( 35 17 40  5        ) ( 35 14 35 12        ) ( 25 10 43 21        )

(100                 ) (100                 ) (100                 )
(100                 ) (100                 ) ( 98              2  )
( 72    25           ) ( 83    16           ) ( 76    24           )
( 11    88           ) ( 13    83           ) ( 14    86           )
( 30  5 62  3        ) ( 26  7 64  1        ) ( 30  2 63  1        )

(100                 ) (100                 ) (100                 )
( 99              1  ) ( 98              2  ) (100                 )
( 53    47           ) ( 53    46           ) ( 69    30           )
( 16    84           ) ( 11    89           ) ( 16    82           )
( 28  1 67           ) ( 32  4 62           ) ( 18  5 75           )

(100                 ) (100                 ) (100                 )
( 99     1           ) ( 99     1           ) (100                 )
( 40    60           ) ( 37    62           ) ( 51    48           )
(  9    90           ) ( 14    84           ) ( 10    90           )
(    22 76           ) (    21 78           ) (    15 85           )

(100                 ) (100                 ) (100                 )
( 98     2           ) ( 98     2           ) ( 94     5           )
( 26    74           ) ( 25    74           ) ( 29    69           )
( 18    82           ) ( 13    84           ) (  9    89           )
(    59 39           ) (    59 40           ) (    53 45           )

(100                 ) (100                 ) (100                 )
( 98     2           ) ( 98     2           ) ( 96     4           )
( 12    88           ) ( 10    90           ) ( 11    89           )
( 16    82           ) ( 19    79           ) ( 16    83           )
(    85 13           ) (    91  8           ) (    83 16           )

(100                 ) (100                 ) (100                 )
( 94     6           ) ( 96     4           ) ( 96     4           )
( 10    89           ) (  9    91           ) (  7    92           )
(  9    90           ) ( 12    87           ) ( 14    85           )
(    97  2           ) (   100              ) (    99  1           )

(100                 ) (100                 ) (100                 )
( 93     7           ) ( 96     4           ) ( 95     5           )
(  4    96           ) (  3    96           ) (  3    97           )
( 10    90           ) (  5    94           ) ( 10    89           )
(   100              ) (    99              ) (    97              )
```

These three columns correspond to the fourth three blocks above (a-t-t)

```
(100                 ) (100                 ) (100                 )
( 96              4  ) ( 91              9  ) ( 95              5  )
( 78             22  ) ( 80             19  ) ( 80             20  )
( 62             38  ) ( 62             37  ) ( 57             41  )
(          63    24 13) (          61    21 18) (          64    16 20)

(100                 ) (100                 ) (100                 )
( 82    18           ) ( 81    19           ) ( 81    19           )
( 11    89           ) (  4    96           ) (  7    93           )
(    61 37           ) (    58 41           ) (    63 37           )
(    98        1     ) (    96        4     ) (    93        4     )

(100                 ) (100                 ) (100                 )
( 74    26           ) ( 79    21           ) ( 71    28           )
(  3    97           ) (  5    94           ) (  5    95           )
(    65 34           ) (    71 28           ) (    78 22           )
(    94        5     ) (    94        5     ) (    93        6     )

(100                 ) (100                 ) (100                 )
( 83    16           ) ( 81    19           ) ( 88    12           )
(  4    96           ) (  5    95           ) (  7    93           )
(    65 35           ) (    58 42           ) (    67 32           )
(    92        6     ) (    95        5     ) (    96        3     )

(100                 ) (100                 ) (100                 )
( 84    16           ) ( 82    18           ) ( 77    22           )
(  6    94           ) (  8    91           ) (  4    96           )
(    62 37           ) (    65 32           ) (    63 35           )
(    91        7     ) (    95        4     ) (    94        5     )

(100                 ) (100                 ) (100                 )
( 89    11           ) ( 87    13           ) ( 83    16           )
(  8    92           ) (  3    97           ) (  5    95           )
(    52 47           ) (    61 39           ) (    59 41           )
(    96        3     ) (    95        5     ) (    94        6     )

(100                 ) (100                 ) (100                 )
( 92     8           ) ( 76    23           ) ( 82    17           )
(  6    94           ) ( 10    90           ) (  9    91           )
(    61 39           ) (    67 32           ) (    63 37           )
(    97        3     ) (    96        3     ) (    98        2     )

(100                 ) (100                 ) (100                 )
( 85    15           ) ( 75    25           ) ( 86    13           )
(  9    90           ) (  8    92           ) (  5    95           )
(    64 36           ) (    68 32           ) (    62 38           )
(    98        2     ) (    92        6     ) (    94        5     )

(100                 ) (100                 ) (100                 )
( 84    15           ) ( 81    19           ) ( 83    16           )
(  4    96           ) (  6    94           ) (  4    96           )
(    60 40           ) (    65 35           ) (    63 37           )
(    98        2     ) (    94        4     ) (    96        4     )

(100                 ) (100                 ) (100                 )
( 77    23           ) ( 83    16           ) ( 85    15           )
( 11    88           ) (  8    92           ) (  7    92           )
(    62 38           ) (    72 28           ) (    66 34           )
(    95        4     ) (    96        4     ) (    99        1     )
```

Appendix B: Software

The following two files are the current state of the code. The code is not incredibly complex, it is provided here for completeness. The files are “simple.lisp” and “straightforward.lisp” and are two independent Hopfield network implementations (their names were chosen in anticipation of the project’s increasing complexity, and may not actually indicate the lucidity of the implementations). The first was used to identify additional requirements and flesh out the project objectives before the second was written. The results described in this paper are based on the latter.

simple.lisp

```
(in-package :cl-user)

;; Implementation of the most basic Hopfield network with a periodic
;; suppression

;; not adjusting the decision threshold of the neurons over time.
;; using a simple set of four arbitrarily chosen patterns as a training set.
;; not providing the original input image into rounds after the first.
;; holding the inhibition in place until convergence every other cycle.
;; using -1 as the output of inhibited neurons.

(defun dot-product-transfer-function (network node)
  (with-slots (size weights value-source) network
    (loop for source from 0 to (1- size) sum
         (* (aref value-source source)
            (aref weights node source)))))
(defun tanh-dot-product-transfer-function (network node)
  (with-slots (size weights value-source) network
    (tanh (loop for source from 0 to (1- size) sum
                (* (aref value-source source)
                   (aref weights node source))))))

(defclass hopfield-network ()
  (;; persistent state (size, weight, etc.)
   (size :initform nil :initarg :size :accessor size)
   (transfer-function :initform #'dot-product-transfer-function
                      :initarg :transfer-function :accessor transfer-function)
   (thresholds :initform nil :accessor thresholds)
   (weights :initform nil :accessor weights)
   ;; state while running (last/next node states)
   (value-source :initform nil :accessor value-source)
   (value-target :initform nil :accessor value-target)
   ))

(defmethod shared-initialize :after ((network hopfield-network) slot-names &key)
  (declare (ignore slot-names))
  (with-slots (size thresholds weights value-source value-target) network
    ;; ensure value vectors (source and target) for matrix mul
    (setf value-source (make-array size :element-type 'single-float
                                        :initial-element 0.0)
          value-target (make-array size :element-type 'single-float
                                        :initial-element 0.0)
          thresholds (make-array size :element-type 'single-float
                                      :initial-element 0.0)
          weights (make-array (list size size) :element-type 'single-float
                                               :initial-element 0.0))))

;; given a number of nodes (= associative binary image size) and a set
;; of images, make a network
(defmethod train ((network hopfield-network) images)
  (with-slots (size weights) network
    (dotimes (i size)
      (dotimes (j size)
        (when (/= i j) ; diagonal is already zero
          (setf (aref weights i j)
                (/ (loop for image in images sum
                         (if (eq (aref image i) (aref image j))
                           1.0
                           -1.0))
```

```
                  size))))))
  network)

;; given an image of size equal to the number of nodes in the network,
;; return the image the network converges to

(defmethod iterate ((network hopfield-network))
  (with-slots (size thresholds transfer-function value-source value-target) network
    ;; compute value-target as weight/value dot products
    (dotimes (node size)
      (setf (aref value-target node)
            (funcall transfer-function network node)))
    ;; scan value-target applying threshold
    (dotimes (node size)
      (cond ((< (aref value-target node)
                (aref thresholds node))
             (setf (aref value-target node) -1.0))
            ((< (aref thresholds node)
                (aref value-target node))
             (setf (aref value-target node) 1.0))
            (t (setf (aref value-target node)
                     (aref value-source node)))))))

(defmethod changed-p ((network hopfield-network))
  (with-slots (size value-source value-target) network
    ;; check for changes between last (value-source) and next (value-target) node outputs
    (dotimes (node size)
      (when (/= (aref value-source node)
                (aref value-target node))
        (return t)))))

(defmethod update ((network hopfield-network))
  (with-slots (size value-source value-target) network
    ;; copy value-target into value-source
    (dotimes (node size)
      (setf (aref value-source node) (aref value-target node)))))

(defmethod retrieve ((network hopfield-network) image)
  (with-slots (size value-source value-target) network
    (flet ((init-value-source () ; from image
             (dotimes (i size)
               (setf (aref value-source i)
                     (if (aref image i) 1.0 -1.0)))))
      ;; run the update rule to convergence
      (init-value-source)
      (loop until (progn (iterate network)
                         (update network)
                         (not (changed-p network))))
      (let ((result (make-array size :initial-element nil)))
        (dotimes (i size)
          (when (< 0 (aref value-source i))
            (setf (aref result i) t)))
        result))))

;; specialized subclass of a hopfield network that can inhibit all but
;; "cyclic" nodes suppressed

(defclass cyclic-hopfield-network (hopfield-network)
  ((inhibiting :initform nil :initarg :inhibiting :accessor inhibiting)
   (cyclic-nodes :initform nil :initarg :cyclic-nodes :accessor cyclic-nodes)))

(defmethod changed-p :around ((network cyclic-hopfield-network))
  (with-slots (value-source value-target) network
    (if (inhibiting network)
      ;; check for changes between last (value-source) and next
      ;; (value-target) node outputs for non-suppressed nodes
      (loop for node in (cyclic-nodes network) do
           (when (/= (aref value-source node)
                     (aref value-target node))
             (return t)))
      (call-next-method))))

(defmethod update :around ((network cyclic-hopfield-network))
  (with-slots (size value-source value-target) network
    (if (inhibiting network)
      (let ((cn (copy-seq (cyclic-nodes network))))
        (dotimes (i size)
          (cond ((and cn (= i (first cn)))
                 (pop cn)
                 (setf (aref value-source i) (aref value-target i)))
                (t (setf (aref value-source i) -1.0)))))
```

```
      (call-next-method))))

(defmacro with-inhibition ((network) &body body)
  (let ((networksym (gensym)))
  `(let ((,networksym ,network))
     (unwind-protect
        (progn (setf (inhibiting ,networksym) t)
               ,@body) ; <- this is the return value produced
       (setf (inhibiting ,networksym) nil)))))

(defmethod retrieve-sequence ((network cyclic-hopfield-network) image0)
  (flet ((array-equal (a1 a2) ; ... assuming same size
           (loop for i from 0 to (1- (size network)) do
                (when (not (eq (aref a1 i) (aref a2 i)))
                  (return nil))
                finally (return t))))
    (let* ((image (retrieve network image0))
           (projection (with-inhibition (network)
                         (retrieve network image)))
           (image-sequence (list image image0))
           (projection-sequence nil))
      (loop do
           (push projection projection-sequence)
           (push (setf image (retrieve network projection))
                 image-sequence)
           (with-inhibition (network)
             (setf projection (retrieve network image)))
           until (array-equal projection (first projection-sequence)))
      (values (reverse projection-sequence)
              (reverse image-sequence)))))

#|
;;;; Testing network construction and image retrieval

(setf first-half (append (loop for i from 1 to 50 collecting t)
                         (loop for i from 1 to 50 collecting nil)))
(setf full-random (loop for i from 1 to 100 collect
                        (if (< 0.5 (random 1.0)) t nil)))
(setf last-third (append (loop for i from 1 to 66 collect nil)
                         (loop for i from 67 to 100 collect t)))
(setf first-quarter (append (loop for i from 1 to 25 collect t)
                            (loop for i from 26 to 100 collect nil)))
(setf net (train (make-instance 'hopfield-network :size 100)
                 (list (coerce first-half 'vector)
                       (coerce full-random 'vector)
                       (coerce last-third 'vector)
                       (coerce first-quarter 'vector))))

(defun noise (L percent)
  (loop for elt in L
        for i from 1 to (length L) collecting
       (if (> percent (random 1.0)) ; flip one third at random
         (not elt)
         elt)))

(defun mask (L start &optional end)
  (loop for elt in L
        for i from 0 to (1- (length L)) collecting
        (if (<= start i (or end (length L)))
          elt
          nil)))

(retrieve net (coerce (noise first-half 0.3) 'vector))
(retrieve net (coerce (noise full-random 0.3) 'vector))
(retrieve net (coerce (noise last-third 0.3) 'vector))
(retrieve net (coerce (noise first-quarter 0.3) 'vector))


;;;; Testing path generation starting with an image

(setf net (train (make-instance 'cyclic-hopfield-network :size 100
                                :cyclic-nodes (loop for i from 0 to 99 by 4 collecting i))
                 (list (coerce first-half 'vector)
                       (coerce full-random 'vector)
                       (coerce last-third 'vector)
                       (coerce first-quarter 'vector))))
(retrieve-sequence net (coerce (noise first-half 0.3) 'vector))
(retrieve-sequence net (coerce (noise full-random 0.3) 'vector))
(retrieve-sequence net (coerce (noise last-third 0.3) 'vector))
(retrieve-sequence net (coerce (noise first-quarter 0.3) 'vector))
```

|#

straightforward.lisp

```
(in-package :cl-user)

;; Implementation of the a Hopfield network with periodic suppression
;; and several variable parameters and metrics

;; use tanh or not after the dot product
(defparameter *tanh-of-dotproduct* nil)
;; adjust Ti from one round to the next (tanh of round# or round# based on *tanh-of-dotproduct*)
(defparameter *floating-threshold* nil)
;; suppress or suppress-and-converge with a flag
(defparameter *suppress-and-hold* nil)
;; suppress to 0 or -1 (i.e. with or without bias)
(defparameter *suppress-to-value* -1.0)
;; include the original input as input every round or even during
;;   convergence (?), with a 0..1.0 weighted sum, decreasing geometrically
(defparameter *source-image-weight* 0.0)
;; generate logical patterns in the vectors, with
;;   sections of vector assigned labels and compositing training
;;   patterns from labels (in effect, the triangle training set does this)
;; run until convergence or for a certain number of rounds
(defparameter *max-rounds* :converge)

;; I still need to...
;; bound the learned weights in the system or not (later)
;; specify a vector of all these parameters and run a
;;   series of tests and diagram or graph the performance
;; randomly or systematically suppress a variable % of nodes


(defclass hopfield-network ()
  (;; persistent state (size, weight, etc.)
   (size :initform nil :initarg :size :accessor size)
   (weights :initform nil :accessor weights)
   ;; state while running (last/next node states)
   (value-source :initform nil :accessor value-source)
   (value-target :initform nil :accessor value-target)))
(defmethod shared-initialize :after ((network hopfield-network) slot-names &key)
  (declare (ignore slot-names))
  (with-slots (size weights value-source value-target) network
    ;; ensure value vectors (source and target) for matrix mul
    (setf weights (make-array (list size size) :element-type 'single-float
                                               :initial-element 0.0)
          value-source (make-array size :element-type 'single-float
                                        :initial-element 0.0)
          value-target (make-array size :element-type 'single-float
                                        :initial-element 0.0)))
  network)
;; given a number of nodes (= associative binary image size) and a set
;; of images, make a network
(defmethod train ((network hopfield-network) images)
  (with-slots (size weights) network
```

```
    (dotimes (i size)
      (dotimes (j size)
        (when (/= i j) ; diagonal is already zero
          (setf (aref weights i j)
                (/ (loop for image in images sum
                         (if (eq (aref image i) (aref image j))
                           1.0
                           -1.0))
                   size))))))
  network)

(defmethod transfer-function ((network hopfield-network) node)
  (declare (special *tanh-of-dotproduct*))
  (with-slots (size weights value-source) network
    (let ((dotproduct (loop for source from 0 to (1- size) sum
                            (* (aref value-source source)
                               (aref weights node source)))))
      (if *tanh-of-dotproduct*
        (tanh dotproduct)
        dotproduct))))

(defmethod thresholds ((network hopfield-network))
  (make-array (size network) :element-type 'single-float :initial-element 0.0))

(defmethod iterate ((network hopfield-network))
  (with-slots (size value-source value-target) network
    ;; compute value-target as weight/value dot products
    (dotimes (node size)
      (setf (aref value-target node)
            (transfer-function network node)))
    ;; scan value-target applying threshold
    (let ((thresholds (thresholds network)))
      (dotimes (node size)
        (cond ((< (aref value-target node)
                  (aref thresholds node))
               (setf (aref value-target node) -1.0))
              ((< (aref thresholds node)
                  (aref value-target node))
               (setf (aref value-target node) 1.0))
              (t (setf (aref value-target node)
                       (aref value-source node)))))))))

(defmethod changed-p ((network hopfield-network))
  (with-slots (size value-source value-target) network
    ;; check for changes between last (value-source) and next (value-target) node outputs
    (dotimes (node size)
      (when (/= (aref value-source node)
                (aref value-target node))
        (return t)))))

(defmethod update ((network hopfield-network))
  (with-slots (size value-source value-target) network
    ;; copy value-target into value-source
    (dotimes (node size)
      (setf (aref value-source node) (aref value-target node)))))
```

```
;; given an image of size equal to the number of nodes in the network,
;; return the image the network converges to
(defmethod retrieve ((network hopfield-network) image)
  (with-slots (size value-source value-target) network
    (flet ((init-value-source () ; from image
             (dotimes (i size)
               (setf (aref value-source i)
                     (cond ((numberp (aref image i)) (aref image i))
                           ((aref image i) 1.0)
                           (t -1.0)))))))
      ;; run the update rule to convergence
      (init-value-source)
      (loop until (progn (iterate network)
                         (update network)
                         (not (changed-p network))))
      (let ((result (make-array size :initial-element nil)))
        (dotimes (i size)
          (when (< 0 (aref value-source i))
            (setf (aref result i) t)))
        result))))


(let* ((tail-size 10)
       (side-size 30)
       (overlap-size 10)
       (background-size 100)
       )
  (defun triangle-training-set ()
    (flet ((tail (onp) (loop for i from 1 to tail-size collecting (if onp t nil)))
           (side (onp) (loop for i from 1 to side-size collecting (if onp t nil)))
           (part-side (onp) (loop for i from 1 to (- side-size overlap-size)
                                  collecting (if onp t nil)))
           (overlap-side (onp) (loop for i from 1 to overlap-size collecting (if onp t nil)))
           (background () (loop for i from 1 to background-size collecting nil)))
      ;; sample image (triangle 1, tail 1)
      #+ignore
      (append (side t) (part-side t) (part-side t) (tail t) (tail nil) (tail nil)
              (side nil) (part-side nil) (part-side nil) (tail nil) (tail nil) (tail nil)
              (side nil) (part-side nil) (part-side nil) (tail nil) (tail nil) (tail nil)
              (overlap-side nil) (overlap-side t) (overlap-side t)
              (background))
      (flet ((sample-image (triangle tail)
               (append (loop for tr from 1 to 3 append
                             (append (side (eq tr triangle))
                                     (part-side (eq tr triangle))
                                     (part-side (eq tr triangle))
                                     (loop for ta from 1 to 3 append
                                           (tail (and (eq tr triangle) (eq ta tail))))))
                       (loop for tr from 1 to 3 append
                             (overlap-side (not (eq tr triangle))))
                       (background))))
        (loop for triangle from 1 to 3 append
              (loop for tail from 1 to 3 collecting
                    (coerce (sample-image triangle tail) 'vector))))))
  (defun triangle-explain (pattern image pattern-list)
    (labels ((array-equal (a1 a2) ; ... assuming same size
```

```
  (loop for i from 0 to (1- (length a1)) do
       (when (not (eq (aref a1 i) (aref a2 i)))
         (return nil))
     finally (return t)))
(on-p (image start length)
  (let ((count 0))
    (loop for i from start to (1- (+ start length)) do
         (when (aref image i) (incf count)))
    (cond ((= count length) t)
          ((zerop count) nil)
          (t (float (/ count length))))))
(%explain (p) ; explain similarities or differences from
  ;; ideal pattern pretty much interested in structure of the
  ;; triangle terms, amount of damage etc.:
  ;; triangle-with-tail
  ;; traingle-no-tail
  ;; '(full-side part-side part-side shared-bit shared-bit tail tail tail) <- for extra/missing bits
  ;; '((full-side part-side) (tail full-side)) <- for more than one represented triangle
  ;; '((full-side . 0.9) (part-side . 0.1)) <- for partial pieces
  ;; 'unknown <- for stuff appearing in the background bits
  ;; 'extra-stuff
  (let ((triangle-size (+ (* 3 side-size) (* 3 tail-size) (* -2 overlap-size)))
        (points-of-interest nil)
        #+ignore (extra nil))
    #+ignore
    (loop for i from 0 to (+ (length pattern) -1 (- background-size)) do
         (when (and (not (aref pattern i)) (aref p i))
           (push i extra)))
    #+ignore
    (when extra
      (push (cons 'extra-parts extra) points-of-interest))
    (when (on-p p (- (length pattern) background-size) background-size)
      (pushnew 'background-artifacts points-of-interest))
    (cond (;; triangle 1 is the "right answer"
           (on-p pattern 0 (+ (* 3 side-size) (* -2 overlap-size)))
           (when (on-p p 0 side-size) (push 'full-side points-of-interest))
           (when (on-p p side-size (- side-size overlap-size))
             (push 'part-side points-of-interest))
           (when (on-p p (- (* 2 side-size) overlap-size) (- side-size overlap-size))
             (push 'part-side points-of-interest))
           (when (on-p p (- (* 3 side-size) (* 2 overlap-size)) tail-size)
             (push 'tail points-of-interest))
           (when (on-p p (+ (* 3 side-size) (* -2 overlap-size) tail-size) tail-size)
             (push 'tail points-of-interest))
           (when (on-p p (+ (* 3 side-size) (* -2 overlap-size) (* 2 tail-size)) tail-size)
             (push 'tail points-of-interest))
           (when (on-p p (* 3 triangle-size) overlap-size) (push 'extra-shared-bit points-of-interest))
           (when (on-p p (+ (* 3 triangle-size) overlap-size) overlap-size)
             (push 'shared-bit points-of-interest))
           (when (on-p p (+ (* 3 triangle-size) (* 2 overlap-size)) overlap-size)
             (push 'shared-bit points-of-interest)))
          (;; triangle 2
           (on-p pattern triangle-size (+ (* 3 side-size) (* -2 overlap-size)))
           (when (on-p p triangle-size side-size) (push 'full-side points-of-interest))
           (when (on-p p (+ triangle-size side-size) (- side-size overlap-size))
             (push 'part-side points-of-interest))
```

```
        (when (on-p p (+ triangle-size (- (* 2 side-size) overlap-size)) (- side-size overlap-size))
          (push 'part-side points-of-interest))
        (when (on-p p (+ triangle-size (- (* 3 side-size) (* 2 overlap-size))) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ triangle-size (* 3 side-size) (* -2 overlap-size) tail-size) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ triangle-size (* 3 side-size) (* -2 overlap-size) (* 2 tail-size)) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ (* 3 triangle-size) overlap-size) overlap-size)
          (push 'extra-shared-bit points-of-interest))
        (when (on-p p (* 3 triangle-size) overlap-size)
          (push 'shared-bit points-of-interest))
        (when (on-p p (+ (* 3 triangle-size) (* 2 overlap-size)) overlap-size)
          (push 'shared-bit points-of-interest)))
       (;; triangle 3
        (on-p pattern (* 2 triangle-size) (+ (* 3 side-size) (* -2 overlap-size)))
        (when (on-p p (* 2 triangle-size) side-size) (push 'full-side points-of-interest))
        (when (on-p p (+ (* 2 triangle-size) side-size) (- side-size overlap-size))
          (push 'part-side points-of-interest))
        (when (on-p p (+ (* 2 triangle-size) (- (* 2 side-size) overlap-size)) (- side-size overlap-size))
          (push 'part-side points-of-interest))
        (when (on-p p (+ (* 2 triangle-size) (- (* 3 side-size) (* 2 overlap-size))) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ (* 2 triangle-size) (* 3 side-size) (* -2 overlap-size) tail-size) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ (* 2 triangle-size) (* 3 side-size) (* -2 overlap-size) (* 2 tail-size)) tail-size)
          (push 'tail points-of-interest))
        (when (on-p p (+ (* 3 triangle-size) (* 2 overlap-size)) overlap-size)
          (push 'extra-shared-bit points-of-interest))
        (when (on-p p (* 3 triangle-size) overlap-size)
          (push 'shared-bit points-of-interest))
        (when (on-p p (+ (* 3 triangle-size) overlap-size) overlap-size)
          (push 'shared-bit points-of-interest))))
(cond ((equal points-of-interest '(shared-bit shared-bit tail part-side part-side full-side))
       'triangle-with-tail)
      ((equal points-of-interest '(shared-bit shared-bit part-side part-side full-side))
       'triangle)
      ((equal points-of-interest '(shared-bit shared-bit part-side full-side))
       'damaged-triangle)
      ((equal points-of-interest '(shared-bit shared-bit part-side part-side))
       'damaged-triangle)
      ((equal points-of-interest '(shared-bit shared-bit full-side))
       'damaged-triangle)
      ((equal points-of-interest '(shared-bit shared-bit part-side))
       'damaged-triangle)
      ((equal points-of-interest '(shared-bit shared-bit extra-shared-bit part-side part-side full-side))
       'triangle-with-arc)
      ((equal points-of-interest '(shared-bit shared-bit extra-shared-bit part-side full-side))
       'damaged-triangle-with-arc)
      ((equal points-of-interest '(shared-bit shared-bit extra-shared-bit part-side part-side))
       'damaged-triangle-with-arc)
      ((equal points-of-interest '(shared-bit shared-bit extra-shared-bit))
       'little-triangle)
      ((equal points-of-interest '(shared-bit shared-bit))
       'shared-bits)
      ((equal points-of-interest '(shared-bit))
```

```
                     'shared-bit)
                    (t points-of-interest))
               )))
      (cons (if (array-equal image (first pattern-list))
            'noisy-image
            'error)
          (loop for p in (rest pattern-list) collecting
               (%explain p)))))
  )

(defun noise (vector percent)
  (coerce (loop for i from 0 to (1- (length vector)) collecting
               (let ((elt (aref vector i)))
                 (if (> percent (random 1.0)) ; flip one third at random
                   (not elt)
                   elt)))
          'vector))

(defun mask (vector start &optional end)
  (coerce (loop for i from 0 to (1- (length vector)) collecting
               (let ((elt (aref vector i)))
                 (if (<= start i (or end (length vector)))
                   elt
                   nil)))
          'vector))

(defun diffs (v1 v2)
  (loop for i from 0 to (1- (length v1))
       if (not (eq (aref v1 i) (aref v2 i)))
       collect i))

;; specialized subclass of a hopfield network that can inhibit all but
;; "cyclic" nodes suppressed

(defclass cyclic-hopfield-network (hopfield-network)
  ((inhibiting :initform nil :initarg :inhibiting :accessor inhibiting)
   (cyclic-nodes :initform nil :initarg :cyclic-nodes :accessor cyclic-nodes)
   (seq-round :initform nil :initarg :seq-round :accessor seq-round)))

(defmethod thresholds ((network cyclic-hopfield-network))
  (declare (special *tanh-of-dotproduct*))
  (make-array (size network)
              :element-type 'single-float
              :initial-element (if *floating-threshold*
                                 (if *tanh-of-dotproduct*
                                   (tanh (float (* *floating-threshold* (seq-round network))))
                                   (float (* *floating-threshold* (seq-round network))))
                                 0.0)))

(defmethod changed-p :around ((network cyclic-hopfield-network))
  (with-slots (value-source value-target) network
    (if (inhibiting network)
      ;; check for changes between last (value-source) and next
      ;; (value-target) node outputs for non-suppressed nodes
      (loop for node in (cyclic-nodes network) do
           (when (/= (aref value-source node)
```

```
                    (aref value-target node))
              (return t)))
      (call-next-method))))

(defmethod suppress ((network cyclic-hopfield-network) image &optional result)
  (declare (special *suppress-to-value*))
  (or result (setf result (make-array (length image))))
  (let ((cn (copy-seq (cyclic-nodes network))))
    (dotimes (i (size network))
      (cond ((and cn (= i (first cn)))
             (pop cn)
             (setf (aref result i) (aref image i)))
            (t (setf (aref result i) *suppress-to-value*))))
    result))

(defmethod update :around ((network cyclic-hopfield-network))
  (with-slots (size value-source value-target) network
    (if (inhibiting network)
      (suppress network value-target value-source)
      (call-next-method))))

(defmacro with-inhibition ((network) &body body)
  (let ((networksym (gensym)))
  `(let ((,networksym ,network))
     (unwind-protect
        (progn (setf (inhibiting ,networksym) t)
               ,@body) ; <- this is the return value produced
       (setf (inhibiting ,networksym) nil)))))

(defun weighted-sum (v1 v2 round-number)
  (flet ((value-of (x)
           (cond ((numberp x) x)
                 (x 1.0)
                 (t -1.0))))
    (let ((result (make-array (length v1) :element-type 'single-float))
          (weight (expt *source-image-weight* round-number)))
      (dotimes (i (length v1))
        (setf (aref result i)
              (+ (* (- 1.0 weight) (value-of (aref v1 i)))
                 (* weight (value-of (aref v2 i))))))
      result)))

(defmethod retrieve-sequence ((network cyclic-hopfield-network) image0)
  (declare (special *suppress-and-hold*))
  (flet ((array-equal (a1 a2) ; ... assuming same size
           (loop for i from 0 to (1- (size network)) do
                (when (not (eq (aref a1 i) (aref a2 i)))
                  (return nil))
              finally (return t))))
    (setf (seq-round network) 0)
    (if *suppress-and-hold*
      (let* ((image (retrieve network image0))
             (image-sequence (list image image0))
             (projection (with-inhibition (network)
                           (retrieve network image)))
             (projection-sequence nil))
```

```
        (loop do
             (incf (seq-round network))
             (push projection projection-sequence)
             (push (setf image (retrieve network (weighted-sum projection image0
                                                               (seq-round network))))
                   image-sequence)
             (with-inhibition (network)
               (setf projection (retrieve network image)))
           until (if (eq *max-rounds* :converge)
                     (array-equal projection (first projection-sequence))
                     (> (seq-round network) *max-rounds*)))
        (reverse image-sequence))
      (let* ((image (retrieve network image0))
             (image-sequence (list image image0)))
        (loop do
             (incf (seq-round network))
             (setf image (retrieve network (weighted-sum (suppress network image) image0
                                                         (seq-round network))))
             (push image image-sequence)
             until (if (eq *max-rounds* :converge)
                       (array-equal (first image-sequence) (second image-sequence))
                       (> (seq-round network) *max-rounds*)))
        (pop image-sequence)
        (reverse image-sequence)))))


#|

;;;; Testing network construction and image retrieval

(setf triangles (loop for i from 1 to 10 append
                     (triangle-training-set)))
(setf net (train (make-instance 'cyclic-hopfield-network :size (length (first triangles))
                                :cyclic-nodes (loop for i from 0 to (1- (length (first triangles))) by 4
                                                    collecting i))
                 triangles))

;; why are none of these being properly recovered? tails are being
;; dropped (totally and in all cases)
(loop for triangle in triangles collect
     (diffs triangle (retrieve net (noise triangle 0.3))))

(let ((*tanh-of-dotproduct* nil)
      (*floating-threshold* 1.0)
      (*suppress-and-hold* nil)
      (*suppress-to-value* 0.0)
      (*source-image-weight* 0.9))
  ;; recovers triangle (no tail), then 0-2 shared bits, then nothing
  (progn (pprint
  (loop for triangle in (triangle-training-set) collect
       (let ((noisy-triangle (noise triangle 0.3)))
         (triangle-explain triangle noisy-triangle (retrieve-sequence net noisy-triangle)))))
         nil))

(let ((*tanh-of-dotproduct* t)
      (*floating-threshold* 0.4)
```

```
      (*suppress-and-hold* nil)
      (*suppress-to-value* 0.0)
      (*source-image-weight* 0.8)
      (*max-rounds* 10))
  (progn (pprint
  (loop for triangle in (triangle-training-set) collect
       (let ((noisy-triangle (noise triangle 0.2)))
         (triangle-explain triangle noisy-triangle (retrieve-sequence net noisy-triangle)))))
         nil))

(process-run-function
 "Neural Net test"
 #'(lambda ()
     (let* ((triangle (first (triangle-training-set))))
       (with-open-file (s #p"/Users/helink/temp/net-results2.txt"
                          :direction :output :if-exists :supersede :if-does-not-exist :create)
         (loop for hold-down in '(nil t) do
              (loop for tanh-p in '(nil t) do
                   (loop for threshold in '(nil 0.05 0.1 0.15 0.2 0.25 0.30 0.35 0.4 0.5) do
                        (loop for baseline in '(-1.0 -0.5 0.0) do
                             (loop for source in '(0.0 0.5 0.75 0.88 0.95) do
                                  (let ((*suppress-and-hold*   hold-down)
                                        (*tanh-of-dotproduct*  tanh-p)
                                        (*floating-threshold*  threshold)
                                        (*suppress-to-value*   baseline)
                                        (*source-image-weight* source)
                                        (*max-rounds* 15))
                                    (let* ((noisy-triangles (loop for i from 1 to 100 collecting (noise triangle 0.2)))
                                           (results (loop for noisy-triangle in noisy-triangles collecting
                                                          (retrieve-sequence net noisy-triangle)))
                                           (explanations (loop for noisy-triangle in noisy-triangles
                                                               for result in results collecting
                                                                 (triangle-explain triangle noisy-triangle result))))
                                      (print hold-down s)
                                      (print tanh-p s)
                                      (print threshold s)
                                      (print baseline s)
                                      (print source s)
                                      (print explanations s)
                                      (princ #\.)))
                                  )))))
         ))))

(let ((levels '(triangle-with-tail triangle damaged-triangle
                triangle-with-arc damaged-triangle-with-arc
                little-triangle shared-bits shared-bit)))
  (with-open-file (s #p"/Users/helink/temp/net-results2.txt"
                     :direction :input :if-does-not-exist :error)
    (with-open-file (s2 #p"/Users/helink/temp/net-statistics.txt"
                        :direction :output :if-exists :supersede :if-does-not-exist :create)
      (with-open-file (s3 #p"/Users/helink/temp/net-statistics3.txt"
                          :direction :output :if-exists :supersede :if-does-not-exist :create)
        (loop for hold-down in '(nil t) do
             (loop for tanh-p in '(nil t) do
                  (loop for threshold in '(nil 0.05 0.1 0.15 0.2 0.25 0.30 0.35 0.4 0.5) do
```

```
(loop for baseline in '(-1.0 -0.5 0.0) do
     (loop for source in '(0.0 0.5 0.75 0.88 0.95) do
          (let ((*suppress-and-hold*   hold-down)
                (*tanh-of-dotproduct*  tanh-p)
                (*floating-threshold*  threshold)
                (*suppress-to-value*   baseline)
                (*source-image-weight* source)
                (*max-rounds* 15))
            (dotimes (i 5) (read s))
            (let* ((explanations (read s)))
              (when ;; constraints on what to print out
                  t ;; (and (not hold-down) tanh-p)
                ;; statistics...
                (print hold-down s2)
                (print tanh-p s2)
                (print threshold s2)
                (print baseline s2)
                (print source s2)
                ;; average number of levels recognized
                (push (float (/ (loop for explanation in explanations sum
                                      (1- (length (remove-duplicates explanation
                                                                     :test 'equal))))
                                100))
                      x)
                (print (float (/ (loop for explanation in explanations sum
                                       (1- (length (remove-duplicates explanation
                                                                      :test 'equal))))
                                 100))
                       s2)
                ;; likelihood of recognizing canonical second, third, etc. levels
                (print (loop for elt in levels
                          collecting (float (/ (count elt explanations :test #'find)
                                               100)))
                       s2)
                ;; average iterations to recognized second, third, etc. level (time to generalize)
                ;; ... make counter for all elts, inc by position of first appearance, sum/100
                ;; compute variance too?
                (let ((counters (make-array (length levels) :initial-element 0)) ; number of appearances
                      (sums (make-array (length levels) :initial-element 0)))
                  (print (loop for i from 0 to (1- (length levels))
                            for elt in levels collecting
                              (progn (loop for exp in explanations do
                                           (when (position elt exp)
                                             (incf (aref counters i))
                                             (incf (aref sums i) (position elt exp))))
                                     (if (/= 0 (aref counters i))
                                       (float (/ (aref sums i) (aref counters i)))
                                       nil)))
                         s2))
                ;; frequencies of each possible item in positions 1..15
                (let ((freq-table (make-array (list (length (first explanations))
                                                    (length levels))
                                              :initial-element 0)))
                  (loop for exp in explanations do
                       (loop for i from 0 to (1- (length (first explanations)))
                          for elt in exp do
```

```
                                        (when (position elt levels)
                                          (incf (aref freq-table i (position elt levels)) 1))))
                                  (pprint freq-table s2))
                                ;; frequencies of the various sequences (after remove-dupes)
                                ;; under the various variable regimes
                                (let ((sequence-counts (make-array (length *sequences*) :initial-element 0)))
                                  (loop for exp in explanations do
                                       (incf (aref sequence-counts
                                                   (position (remove nil (remove-duplicates exp :test 'equal))
                                                             *sequences*
                                                             :test 'equal))))
                                  (print sequence-counts s3))
                                ;; also, what happens when I vary the %age of nodes suppressed
                                )))
                            )))))
        ))))

(defparameter *sequences*
  '(;; these cases are all pretty common, and for any of the 12 blocks the behavior across
    ;; a block (in terms of preferring one of these sequences to another) is very consistent
    (noisy-image triangle little-triangle)
    (noisy-image triangle shared-bits)
    (noisy-image triangle shared-bits little-triangle)
    (noisy-image triangle) ;; the common case for configurations that average close to 1.0

    ;; these are less frequent but happen in statistically significant frequencies (>5%)
    (noisy-image triangle shared-bits shared-bit) ; this is common across one block, rare otherwise
    (noisy-image triangle triangle-with-arc little-triangle)
    (noisy-image triangle triangle-with-arc)

    ;; these are anomalies
    (noisy-image triangle shared-bit) ; this only occurred significantly in one configuration

    ;; these all occur in less than 5% samples of any trial, usually 1-2% or not at all
    (noisy-image triangle damaged-triangle little-triangle) ; this is a common damage case
    (noisy-image triangle damaged-triangle shared-bits) ; this is a common damage case
    (noisy-image triangle damaged-triangle-with-arc little-triangle)
    (noisy-image triangle damaged-triangle)
    (noisy-image triangle damaged-triangle shared-bits little-triangle)
    (noisy-image triangle little-triangle shared-bits)
    (noisy-image triangle damaged-triangle shared-bits shared-bit)
    (noisy-image triangle damaged-triangle shared-bit)
    (noisy-image triangle damaged-triangle-with-arc)
    ))

(progn
  (loop for i from 0 to (1- 150) by 15  do
     (let ((print-this (subseq foo i (+ i 15)))
           (and-print-this (subseq foo (+ i 150) (+ i 15 150))))
       (princ #\newline)
       (loop for j from 0 to 4 do
            (format t "(~{~:[   ~;~:*~3d~]~})  (~{~:[   ~;~:*~3d~]~})   (~{~:[   ~;~:*~3d~]~})           ~
                       (~{~:[   ~;~:*~3d~]~})  (~{~:[   ~;~:*~3d~]~})   (~{~:[   ~;~:*~3d~]~})~%"
                    (loop for elt in (subseq (coerce (elt print-this (+ (* 0 5) j)) 'list) 0 7)
                         collecting (and (<= 1 elt) elt))
                    (loop for elt in (subseq (coerce (elt print-this (+ (* 1 5) j)) 'list) 0 7)
```

```
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt print-this (+ (* 2 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 0 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 1 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 2 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                ))))
  (princ #\newline)
  (princ #\newline)
  (princ #\newline)
  (loop for i from 300 to (1- 450) by 15  do
     (let ((print-this (subseq foo i (+ i 15)))
           (and-print-this (subseq foo (+ i 150) (+ i 15 150))))
       (princ #\newline)
       (loop for j from 0 to 4 do
            (format t "(~{~:[   ~;~:*~3d~]~})  (~{~:[   ~;~:*~3d~]~})   (~{~:[   ~;~:*~3d~]~})           ~
                       (~{~:[   ~;~:*~3d~]~})  (~{~:[   ~;~:*~3d~]~})   (~{~:[   ~;~:*~3d~]~})~%"
                (loop for elt in (subseq (coerce (elt print-this (+ (* 0 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt print-this (+ (* 1 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt print-this (+ (* 2 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 0 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 1 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                (loop for elt in (subseq (coerce (elt and-print-this (+ (* 2 5) j)) 'list) 0 7)
                    collecting (and (<= 1 elt) elt))
                )))))

|#
```

Distribution

MS0310	9212	A. Bäcker
MS0310	9212	M. D. Rintoul
MS0451	5533	D. P. Gallegos
MS0451	5533	A. I. Gonzales
MS0451	5534	C. D. Harrison
MS0451	5533	J. J. Jones
MS0451	5533	K. W. Larson
MS0455	5517	H. E. Link
MS0455	5517	R. S. Tamashiro
MS0974	5502	L. J. Ellis
MS1170	15240	R. D. Skocypec
MS1188	15241	J. S. Wagner
MS1351	5533	B. D. Farkas
MS9018	8945-1	Central Technical Files
MS0899	9616	Technical Library