Johns Hopkins University – Neural Networks, Fall 2017

Capacity Comparison Storkey and Hebbian Learning Rules

Chris Jeschke
Kris Cate
12/7/2017

Chris Jeschke
Kris Cate

Executive Summary

The Hopfield network is a recurrent binary neural network focused primarily on improving pattern recognition via machine learning. Once created, the network is trained on a set of training data, and is then able to recognize other similar data when it is presented with it. While it is effective, it has been found that a Hopfield network is unable to store more than a certain amount of data relative to its size. In an attempt to overcome this problem, Amos Storkey came up with an improvement to the Hopfield Network known as the Storkey Learning Rule. With this rule, the a network should be able to retain more data during the training process, allowing a wider range of patterns to be stored in the same sized network when compared to one trained under the traditional Hebbian Learning Rule. In this paper we compare the Storkey Learning Rule against the Hebbian Learning Rule. Experimental results show the Storkey rule to be slightly better at differentiating noisy inputs when compared to the Hebbian. Additionally, the Storkey rule experimentally shows that it affords more capacity than the Hebbian, and its experimental capacity is almost identical to the theoretical capacity put forth by Storkey.

Motivation for the Study

A Hopfield network is a recurrent binary neural network that can be represented with the following equation [1]:

$$\sigma_i(t+1) = sign\left(\sum_{j=1, j\neq i}^{n} w_{ij}\sigma_j(t)\right) for\ i = 1,2,\dots,n \quad (1)$$

where $\sigma_i(t)$ defines the state of node i at time t, which - since it is bipolar - will take on one of ± 1. w_{ij} is the weight between nodes i and j and the matrix of all weights is defined as W. Since this paper is a study on capacity, we are going to define ξ to be the set of υ exemplars $\{\xi^1, \xi^2, \dots, \xi^\upsilon\}$ with each exemplar having length n and $\xi_j^\upsilon \in \{\pm 1\}$. W^υ will be used to define the status of the weight matrix after upsilon exemplars have been introduced.

To evaluate the capacity of the Hopfield network using various learning rules, it is first necessary to understand exactly what is meant by a learning rule. A learning rule defines how the network sets its weights based on the training data so that it can determine the closest match when presented with an exemplar. In his paper defining the rule, Storkey described the key characteristics of a learning rule as follows:

- Local

The Hopfield Network works in parallel, in order to maintain this parallelism it is important that the individual weights should only refer to the status of the nodes on either side of it.

- Incremental

The requirement means that the rule should be able to modify old network configurations rather than referring to the previously learned patterns. This requirement exists to ensure that as new patterns are presented, the network is able to adjust itself to them rather than having to go through the entirety of the previously learned data to adapt to a new exemplar. This means that $W^\upsilon$is only a function of $W^{\upsilon-1}$ and ξ^υ.

- Immediacy

There are two ways that a rule can update the network, it can update immediately or as a limit process. If the learning process uses finite steps to arrive at W^υ then it is immediate, whereas anything else is a limit process.

Capacity

The capacity of the learning rule will be the focus of this paper. The capacity of a network is defined as the ratio of stored exemplars to the number of neurons. The capacity of a network ultimately determines its utility. Hopfield Networks trained using Hebbian Learning have capacity limit of

$$\frac{n}{2\ln(n)} \tag{2}$$

exemplars, where *n* is both the number of the neurons in the network and the exemplar length. Using the formula (2), a network of 10 nodes (n=10) could store approximately 2.17 exemplars without risking error in attempting to recall them. Constructing networks capable of storing greater numbers of exemplars requires exponential increases to the number of nodes, imposing increasing performance penalties as the computational effort to use the network increases. Pursuing approaches to improve the capacity ratio of a network is highly desirable so we can store & recall more exemplars with less computational effort.

Hebbian Learning Rule

The Hebbian learning rule defines the weight matrix as follows [2]:

$$w_{ij} = \frac{1}{n}\sum_{\mu}^{v} \xi_i^{\mu} * \xi_j^{\mu} \tag{3}$$

What this equation tells us is that the weight between nodes i and j is the average of all the product of all exemplars status at nodes i and j. There is a capacity limit of the Hebbian Learning rule of $\frac{n}{2\ln(n)}$. We also confirm that the learning rule meets the requirements set forth; it is local since the weights only depend on adjacent nodes *i*, *j*, it is incremental since it is a sum, and it has the ability to be updated immediately.

Storkey Learning Rule

The Storkey learning rule defines the weight matrix in the following manner:

$$w_{ij}^0 = 0\ \forall\ i,j\ and\ w_{ij}^v = w_{ij}^{v-1} + \frac{1}{n}\xi_i^v\xi_j^v - \frac{1}{n}\xi_i^v h_{ji}^v - \frac{1}{n}\xi_i^v h_{ij}^v \tag{4}$$

$$h_{ij}^v = \sum_{k=1,k\neq i,j}^{n} w_{ij}^v \xi_i^v \tag{5}$$

When presented with a new exemplar the Storkey learning method begins by acting in the same way as the Hebbian. The rule then refines the weight at *i, j* by negating the influence of the activity from the other nodes in the network *k*, where *k* != *i,j*. . This results in more evenly distributed fixed points in the network and an improved capacity (5).

$$\frac{n}{\sqrt{2\ln(n)}} \tag{6}$$

Experimental Setup: Experiment 1, Successful digit exemplar retrieval from Hopfield Networks

The usefulness of a trained Hopfield Network hinges on the ability of the network to recall one of the exemplars *p* on which it was trained, when the network is provided with a noisy variant *p'*. This recollection of an exemplar happens by first imposing the noisy variant *p'* on the network to obtain an initial output, then continuously re-imposing that output and each subsequent output on the network until there are no changes between the outputs, having converged to a fixed point in the system - the expected exemplar *p*.

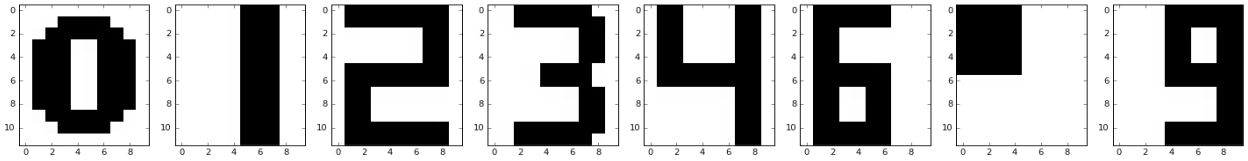
Imposing the noisy input patterns on the network to recall an exemplar can occur synchronously or asynchronously. The former requires that for a time-step t in the iteration, each neuron in the network is activated simultaneously using the output produced from iteration t-1, typically accomplished through a simple matrix multiplication

operation, then updating all weights simultaneously. Unfortunately, the synchronous approach is not without limitations. It provides no strong guarantee of converging to a single output and may instead cycle through multiple output patterns repeatedly. It would therefore be preferable to use a modality that does create convergence so that we can readily identify a final output of the network, from which we can evaluate whether the network successfully recovered the desired exemplar.

Asynchronous updating has been proven to result in this convergence [2]. Asynchronous updating functions by calculating the activity of a single selected node - randomly or purposefully - using the input exemplar imposed on the network. The result of the activation is used to update the input exemplar, after which another node is selected randomly or purposefully for activation. An iteration of asynchronous updating is complete once all nodes have had their activity functions evaluated. The update process is considered to have converged once an iteration of node activations exhibits no changes. That is no node switched from -1 to 1 or vice versa. The resulting output is the recalled exemplar. Storkey applied this method specifically per his description of the network in his paper: "These neurons are fully connected, and are updated asynchronously and in parallel."[1]. Due to this requirement, all experiments in this study will be conducted with asynchronous updating.

With our method for recalling exemplars established, let's move on to our experiment. Here we will evaluate the ability to recover exemplars from a Hopfield network trained under the Hebbian and Storkey learning rules. The exemplars we have chosen are from the Lippmann paper [2], consisting of 8 patterns covering the digits 0,1,2,3,4,6, and 9, with an additional *block* pattern. As seen in the figure below, there is some unavoidable overlap between some of the exemplars, particularly 2,3, and 9, but generally an attempt was made to create distinct figures.

Training Exemplar:

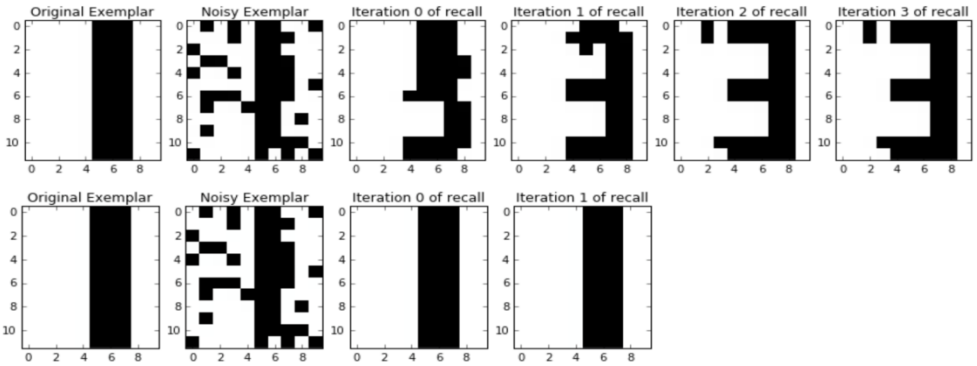


Now that we have chosen our exemplars, we can begin the set of experiments. In the first we will look at what happens when we train a Hopfield Network under the Hebbian Learning Rule, then attempt to recall each exemplar when imposing a noisy variant on the network. We'll continue to replicate Lippmann's example by using the same methodology to generate the noisy exemplar - randomly switching the elements of the exemplar with a probability of $P(switch) = 0.25$ [2].

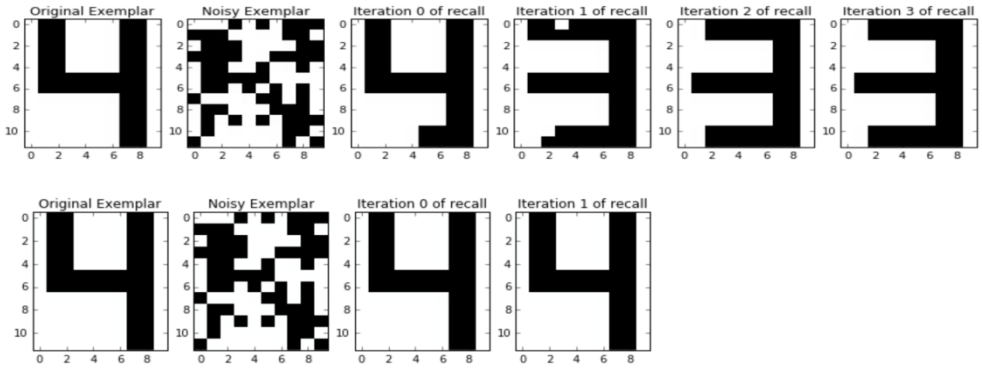
Experimental Results: Experiment 1

The results of the first experiment confirmed the improved utility of a Hopfield network trained under the Storkey learning rule. For a single initial run, there was a clear improvement in the recovery of exemplars 1, 4 and 6; this improvement is best observed by viewing them adjacent one another:

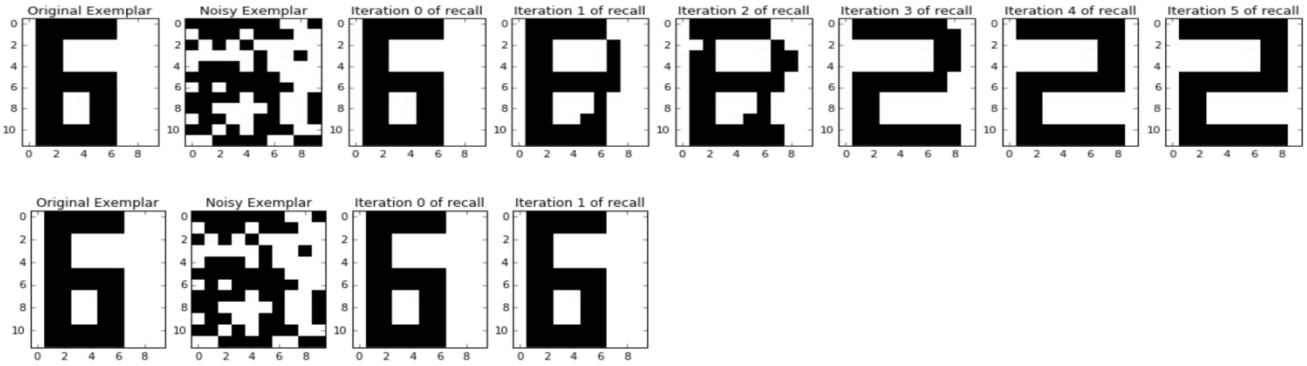
Exemplar 1: Hebbian (top) vs. Storkey (bottom)



Exemplar 4: Hebbian (top) vs. Storkey (bottom)

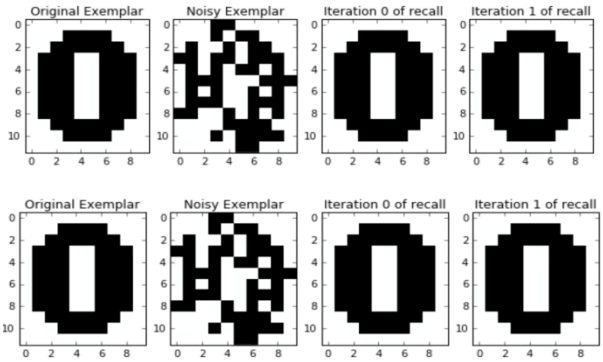


Exemplar 6: Hebbian (top) vs. Storkey (bottom)

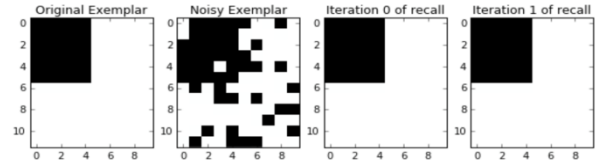


The network trained under the Storkey rule converged with less iteration and retrieved the correct training exemplar in each case. The results when considering the exemplars 0 and *block* showed both rules performing equally well. Each network recovered these exemplars successfully within the same number of iterations:

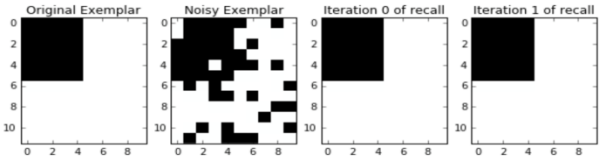
Exemplar 0: Hebbian (top) vs. Storkey (bottom)



Exemplar "block": Hebbian (top) vs. Storkey (bottom)

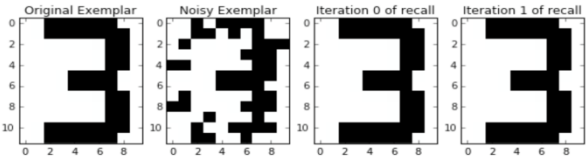


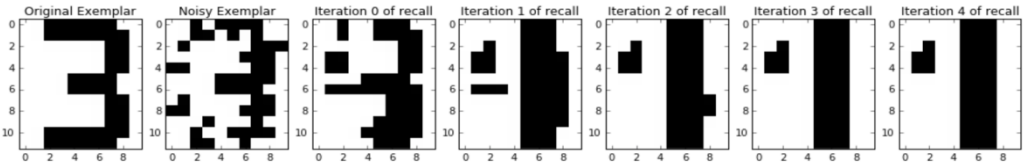
Chris Jeschke
Kris Cate



Exemplar 3 was not able to be recovered by the Storkey trained network, while it was recalled successfully using the Hebbian version:

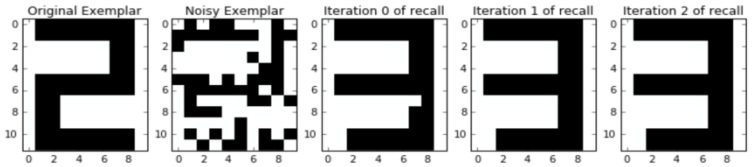
Exemplar 3: Hebbian (top) vs. Storkey (bottom)

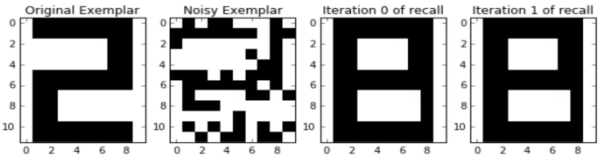




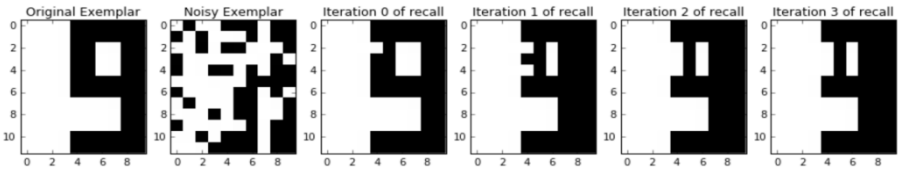
Finally we have exemplars 2 and 9 as outliers for this examination, in that neither was recovered successfully from either network.

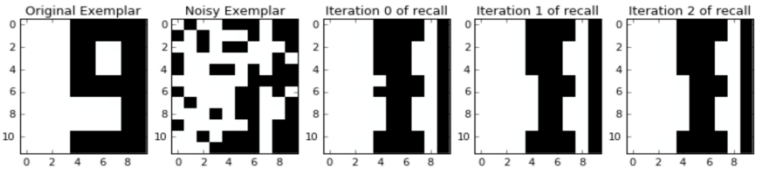
Exemplar 2: Hebbian (top) vs. Storkey (bottom)





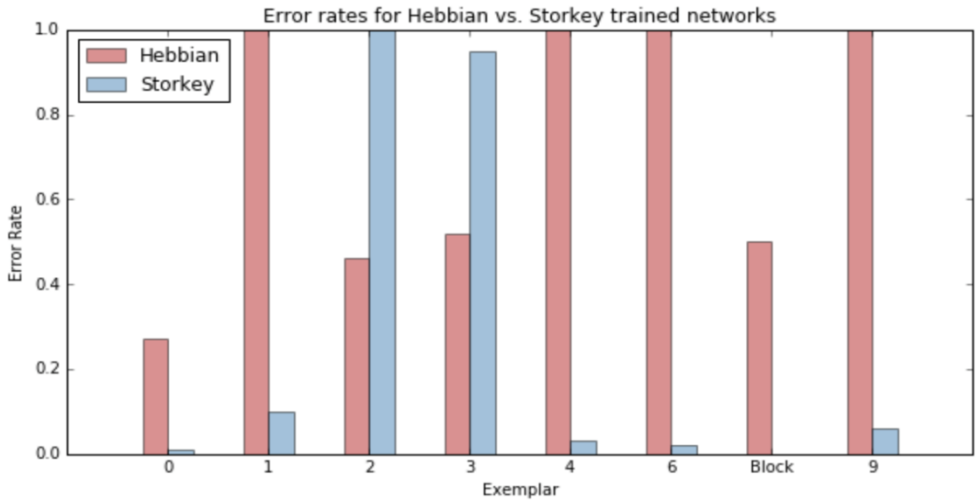
Exemplar 9: Hopfield (top) vs. Storkey (bottom)





Interestingly - and perhaps expectedly - the Hebbian and Storkey trained networks converged to different attractor patterns when attempting to recall exemplars 2 and 9. Conceptually this makes sense as the underlying weight matrix in each network is different, though the recall mechanics are identical.

This experiment concluded with a side-by-side comparison of the rate of error when attempting to recall 100 noisy variations of each exemplar from the Hebbian and Storkey trained networks. The chart shows a clear improvement on the recovery of exemplars 0, 1, 4, 6, *block* and 9 when using the Storkey network:



A final interesting observation is that exemplars 2 and 3 were successfully returned +50% of the time from the Hebbian network, while the Storkey method was only able to return the exemplars successfully less than 10% of the time.

Experimental Setup: Experiment 2:

Our second experiment will demonstrate the improvement yielded from the Storkey Learning rule by constructing increasingly large Hopfield Networks, training them up to and beyond their maximum capacities, while measuring the rate of error when attempting to retrieve exemplars. The experiment's methodology is as follows:

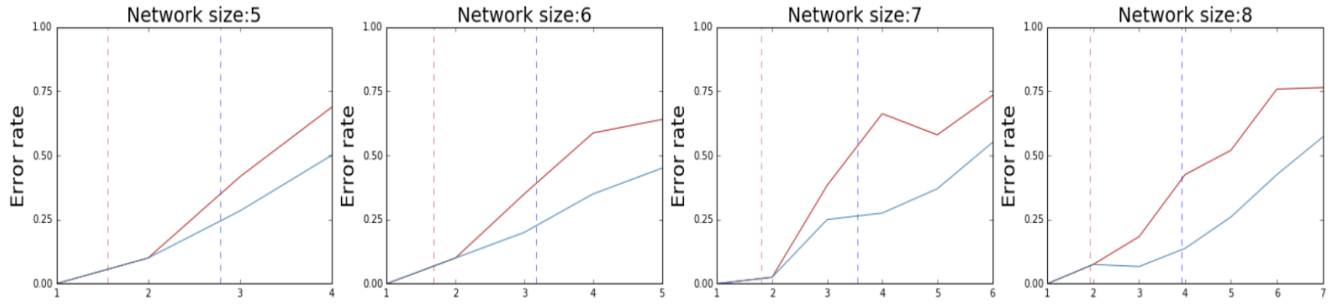
1. Iterate through networks of size *n,* such that $4 < n < 21$.
2. For each network of size *n* iterate through *k* exemplars, such that $1 < k < n$.
3. For each network of size *n* with a population of *k* exemplars, run 20 trials:
    a. For each trial, generate *k* random - though unique - exemplars of length *n*. Each random exemplar is a bipolar vector and no two exemplars in a trial are the same.
    b. Train two networks, one using the Hebbian Learning Rule, and one using the Storkey Learning Rule.
    c. Iterate through the exemplars and impose each upon the networks, recording an error when the recalled exemplar does not match the imposed.
    d. Calculate the error rate for each network type.
    e. After 20 trials, return the mean error rate for that *n, k.*

Experimental Results: Experiment 2

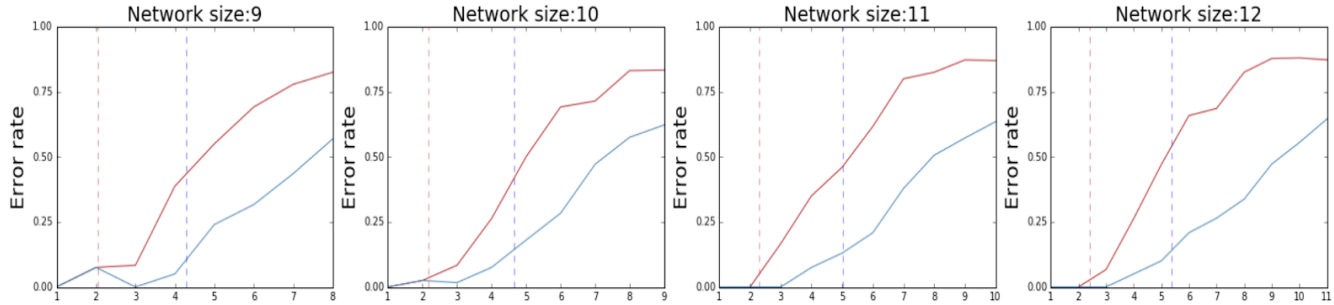
The results of the second experiment create a more generalized, objective comparison between the Hebbian and Storkey trained networks and their ability to recall exemplars as they approach and exceed their capacity bounds. Error rates for the Hebbian (red) and Storkey (blue) networks have been plotted in conjunction with their capacity bounds (dotted vertical lines) to visualize how error rates increase as more and more exemplars are added to the

network. The capacity limit line helps visualize any points of inflection in the error rate as the network's ability to recover exemplars should - theoretically - degrade significantly once the number of exemplars exceeds it.

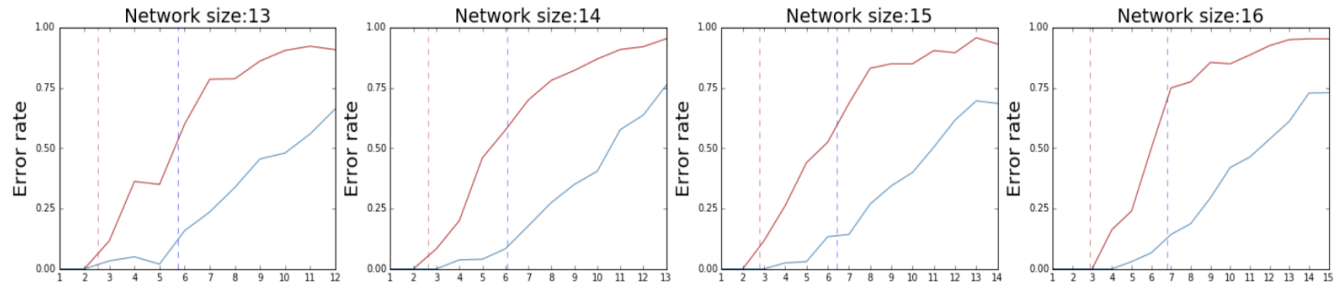
Networks of 5 to 8 nodes



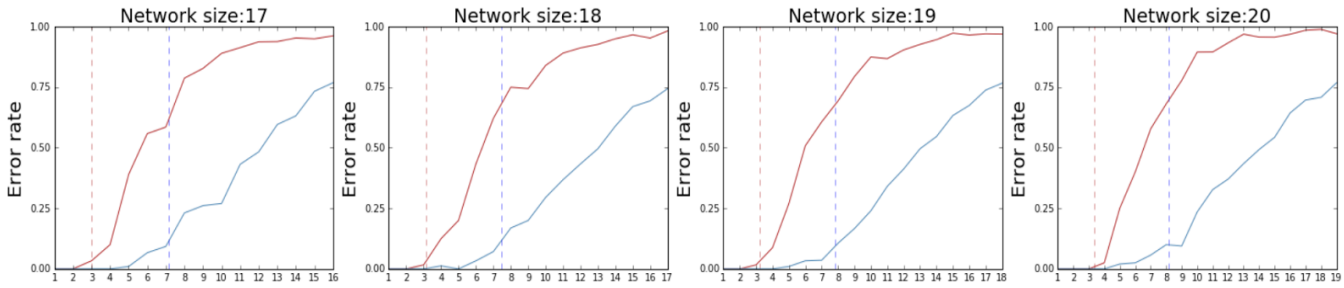
Networks of 9 to 12 nodes



Networks of 13 to 16 nodes



Networks of 17 to 20 nodes



The superiority of the Storkey learning rule becomes increasing evident as the network size increases, with the error rate increasing at a substantially slower rate. Our most extreme example is with a network of 20 neurons where the Hebbian trained network begins degrading promptly at 4, which is close to its capacity of $\frac{20}{2\ln(20)} = 3.33$. The Storkey network continues to perform quite well until 8 exemplars are loaded - close to its capacity bound of $\frac{20}{\sqrt{(2\ln(20))}} = 8.19$.

Further examining our results, one can see that in the majority of the tests, the Hebbian network begins to degrade promptly after exceeding the capacity of the network. In several instances there appears to be a point of inflection near the capacity limit of exemplars. The Storkey trained networks show a much smoother degradation.

Conclusion

The Storkey learning rule was able to perform better in almost every experiment put forth. In the recall testing the Storkey method performed as good as or better than the Hebbian, with the exception of 2 training exemplars. This poor performance may be able to be overcome by adjusting the exemplars so that there is less overlap between active nodes in the training exemplars. This result also came when the Hebbian network was at only 60% of its max capacity, an interesting result would be to examine how well each network is able to recall - in both speed and accuracy - when it's near max capacity. With the exception of those few exemplars that the Storkey networks struggled with during the recall experiment, those networks met or exceeded our expectations in all cases. Its ability to perform well up to 97% of its theoretical capacity is especially interesting. Another interesting result for future study would be to examine the various basins of attractions of each learning rule, and the difference between where the nodes converge to per the learning rule used.

Chris Jeschke
Kris Cate

Works Cited

[1] A. Storkey, "Increasing the capacity of a Hopfield network without sacrificing functionality", Imperial College, Edinburgh, 1997.

[2] Lippmann, Richard P., "An Introduction to Computing with Neural Nets", IEEE ASSP Magazine, April 1987.

Appendices

- Appendix A: Hopfield Network Code
- Appendix B: Lippmann Exemplar Code
- Appendix C: Random Exemplar code

Appendix A: Hopfield Network code

The Hopfield Network was implemented using Python 2.7.12 using a combination of the core Python libraries and Numpy for vector & matrix operations. The API exposed through the public methods of the class focuses on simplicity, allowing consuming code to specify the learning rule ("Hebb" for Hebbian, or Storkey) to use when initializing the network. Exemplars can be recalled by invoking the synchronous and asynchronous recall methods.

```
import copy
import math
import numpy as np

class HopfieldNetwork(object):
    """
    Implements a Hopfield network for exemplar storage and retrieval.  The dimensions of the weight matrix
    for the network are infered from the exemplar vectors supplied during the initialization.
    """

    def __init__(self, v_exemplars, learning_rule='Hebb', debug=False):
        """
        Initialize a Hopfield Network given a list of exemplars and a specified learning rule
        :param v_exemplars: list of exemplars (list of vectors)
        :param learning_rule: "Hebb" = Hebbian Learning Rule,  'Storkey' = Storkey Learning Rule
        :param debug:  Should debug output be printed?  Default is False.
        """
        # Initialize a logger for debug purposes
        def logger(msg):
            if debug:
                print msg

        self.__logger = logger

        # Need at least 1 exemplar
        assert(len(v_exemplars[0]) >= 1)

        # All exemplars should be same length
        n = len(v_exemplars[0])
        for exemplar in v_exemplars:
            assert(len(exemplar) == n)
        self.__exemplars = v_exemplars

        # The number of neurons is the number of elements in each exemplar
        self.__num_neurons = len(self.__exemplars[0])

        # Hebbian learning
        if learning_rule is "Hebb":
            self.__hebbian_learning_rule(v_exemplars)
        elif learning_rule == "Storkey":
            self.__storkey_learning(v_exemplars)
        else:
            print "Unrecognized rule"
            # throw exception...

    def __hebbian_learning_rule(self, v_exemplars):
        """
        Implement the Hebb rule for learning the Hopfield network
```

```
        :param v_exemplars: exemplars
        :return: initialized weight matrix
        """
        # Start with an empty weights matrix
        self.__weight_matrix = np.zeros(shape=(self.__num_neurons, self.__num_neurons))

        # Initialize the weight matrix
        for exemplar in v_exemplars:
            n = len(exemplar)
            # Hebbian Learning Rule:  w_ij_new = w_ij_current + (1/num neurons) * e_i * e_j
            # where i & j are neuron indices & indices into the exemplar
            weights_delta = np.outer(exemplar, exemplar)
            weights_delta = (1.0 /n) * weights_delta

            np.fill_diagonal(weights_delta, 0)
            self.__weight_matrix = np.add(self.__weight_matrix, weights_delta)

        # Capacity for a Hopfield Network trained via Hebbian learning
        if self.__num_neurons > 1:
            self.__capacity = (1.0 * self.__num_neurons) / (2 * math.log(self.__num_neurons))
        else:
            self.__capacity = 1

    def __storkey_learning(self, v_exemplars):
        """
        Implement the Storkey Learning Rule
        :param v_exemplars:
        """

        # Start with empty matrix  (w_ij^0)
        self.__weight_matrix = np.zeros(shape=(self.__num_neurons, self.__num_neurons))

        # Incrementally include each exemplar
        for exemplar in v_exemplars:

            self.__logger("Adding Exemplar {0}".format(exemplar))
            weight_matrix_v = np.zeros(shape=(self.__num_neurons, self.__num_neurons))

            def h(i, j, w, e, n):
                h_ij = sum([w[i][k] * e[k] for k in range(1, n) if k != i and k != j])
                return h_ij

            for i in range(len(exemplar)):
                for j in range(len(exemplar)):

                    # skip i == j
                    if i == j:
                        continue

                    # 1st term = 1/n EV_i EV_j
                    t1 = (1.0 / self.__num_neurons) * exemplar[i] * exemplar[j]

                    # 2nd term  = 1/n E^v_{i}H^v_{j,i}
                    t2 = (1.0 / self.__num_neurons) * exemplar[i] * \
                         h(j, i, self.__weight_matrix, exemplar, self.__num_neurons)
```

```
                # 3rd term = 1/n h^v_{i,j} E^v_{j}
                t3 = (1.0 / self.__num_neurons) * h(i, j, self.__weight_matrix, exemplar, self.__num_neurons) * \
                    exemplar[j]

                weight_matrix_v[i][j] = self.__weight_matrix[i][j] + t1 - t2 - t3
                self.__logger("w_{0},{1} = {2} + {3} - {4} - {5}".format(i, j, self.__weight_matrix[i][j],
                                                        t1, t2, t3))

        # Update the weight matrix
        self.__weight_matrix = weight_matrix_v

        # Capacity for a Hopfield Network trained using Storkey
        if self.__num_neurons > 1:
            self.__capacity = (1.0 * self.__num_neurons) / math.sqrt(2 * math.log(self.__num_neurons))
        else:
            self.__capacity = 1

    @property
    def num_neurons(self):
        return self.__num_neurons

    @property
    def num_exemplars(self):
        return len(self.__v_exemplars)

    @property
    def capacity(self):
        return self.__capacity

    @property
    def weight_matrix(self):
        return self.__weight_matrix

    def synchronous_recall(self, v_p, max_iterations=10):
        """
        Recall an exemplar from the Hopfield network via F(Wv_p) where F is the hard limiting function and W is the
        weight matrix of the network
        :param v_p: a noisy representation of the exemplar we want to recover
        :return: the retrieved exemplar
        """

        def hard_limiter(x):
            """
            Apply the hard limiting function. F(x) = 1 if x>=0 else, -1
            :param x: x
            :return: 1 or -1
            """
            return 1 if x >= 0 else -1

        self.__logger("Using synchronous recall.")
        self.__logger("Input vector is: {0}".format(v_p))

        # results to return
        results = list()

        x_s = v_p
```

```
        converged = False
        i = 0
        while not converged and i < max_iterations:
            x_s_prev = copy.deepcopy(x_s)

            x_s = np.dot(self.__weight_matrix, np.array([x_s]).transpose())
            x_s = map(hard_limiter, x_s)

            results.append(x_s)

            self.__logger("x_s: {0}, x_s_prev: {1}".format(x_s, x_s_prev))
            # Convergence when the state of the neurons (x_s) is unchanged
            if x_s == x_s_prev:
                converged = True

        return results

    def asynchronous_recall(self, v_p):
        """
        Recall an exemplar using asynchronous updating.
        :param v_p: noisy vector we want to recall from
        :return: [(i, result)]
        """

        def hard_limiter(x):
            """
            Apply the hard limiting function. F(x) = 1 if x>=0 else, -1
            :param x: x
            :return: 1 or -1
            """
            return 1 if x >= 0 else -1

        self.__logger("Using asynchronous recall.")
        self.__logger("Input vector is: {0}".format(v_p))
        # results to return
        results = list()
        x_s_prev = v_p
        converged = False
        while not converged:
            x_s = copy.deepcopy(x_s_prev)
            for i, row in enumerate(self.__weight_matrix):
                self.__logger("Evaluating at neuron [{0}]".format(i))
                x_i = np.dot(row, np.array([x_s]).transpose())
                x_i = hard_limiter(x_i)
                self.__logger("x_s[{0}] updated to {1}".format(i, x_i))
                x_s[i] = x_i
            results.append(x_s)
            self.__logger("x_s: {0}, x_s_prev: {1}".format(x_s, x_s_prev))
            # Convergence when the state of the neurons (x_s) is unchanged
            if x_s == x_s_prev:
                converged = True
            x_s_prev = x_s

        return results
```

Appendix B: Lippmann Exemplar code

Generation of the exemplars crafted in Lippmann's paper was accomplished through this class. It is a simple container for the bipolar matrix representations of each exemplar. It also exposes static methods for converting exemplars back and forth between the vector and matrix forms, as the former is used in training and exercising the network while the later is for display.

```
import copy
import random

class LippmanExemplars(object):
    """
    Generates the eight exemplar patterns used in Lippmann's paper.
    Each exemplar is a 12 x 10 matrix with:
        1 = a black pixel
        -1 = a white pixel
    """
    def __init__(self):
        # 12 x 10
        self.__exemplar_1 = [
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
            [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, 1, 1, 1, -1, -1, 1, 1, 1, -1],
            [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
            [-1, -1, -1, 1, 1, 1, 1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        ]

        # 12 x 10
        self.__exemplar_2 = [
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
            [-1, -1, -1, -1, -1, 1, 1, 1, -1, -1],
        ]

        # 12 x 10
        self.__exemplar_3 = [
            [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
            [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
            [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
```

```
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
    ]

    # 12 x 10
    self.__exemplar_4 = [
        [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
        [-1, -1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],
        [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1],
    ]

    # 12 x 10
    self.__exemplar_5 = [
        [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, -1, -1, -1, -1, 1, 1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, 1, 1, 1, 1, 1, 1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
        [-1, -1, -1, -1, -1, -1, -1, 1, 1, -1],
    ]

    # 12 x 10
    self.__exemplar_6 = [
        [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
        [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, -1, -1, -1, -1, -1, -1, -1],
        [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
        [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
        [-1, 1, 1, -1, -1, 1, 1, -1, -1, -1],
        [-1, 1, 1, -1, -1, 1, 1, -1, -1, -1],
        [-1, 1, 1, -1, -1, 1, 1, -1, -1, -1],
        [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
```

```
            [-1, 1, 1, 1, 1, 1, 1, -1, -1, -1],
        ]

        # 12 x 10
        self.__exemplar_7 = [
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
            [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
        ]

        # 12 x 10
        self.__exemplar_8 = [
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
            [-1, -1, -1, -1, 1, 1, -1, -1, 1, 1],
            [-1, -1, -1, -1, 1, 1, -1, -1, 1, 1],
            [-1, -1, -1, -1, 1, 1, -1, -1, 1, 1],
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
            [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1],
            [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1],
            [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1],
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
            [-1, -1, -1, -1, 1, 1, 1, 1, 1, 1],
        ]

    @property
    def exemplars(self):
        return list([
            self.__exemplar_1,
            self.__exemplar_2,
            self.__exemplar_3,
            self.__exemplar_4,
            self.__exemplar_5,
            self.__exemplar_6,
            self.__exemplar_7,
            self.__exemplar_8
        ])

    @staticmethod
    def get_exemplars(as_matrices=False):
        x = LippmanExemplars()

        if as_matrices:
            return x.exemplars
        else:
            exemplars = []
```

```
        for exemplar in x.exemplars:
            v_exemplar = [col for row in exemplar for col in row]
            exemplars.append(v_exemplar)

    return exemplars
  @staticmethod
 def to_matrix(v_exemplar):
    """
    Convert the vector representation of an exemplar to its 12 x 10 matrix form
    :param v_exemplar: the vector representation of the exemplar
    :return: the exemplar as a 12 x 10 matrix
    """
    assert(len(v_exemplar) == 120)
    return list([
        v_exemplar[0:10],
        v_exemplar[10:20],
        v_exemplar[20:30],
        v_exemplar[30:40],
        v_exemplar[40:50],
        v_exemplar[50:60],
        v_exemplar[60:70],
        v_exemplar[70:80],
        v_exemplar[80:90],
        v_exemplar[90:100],
        v_exemplar[100:110],
        v_exemplar[110:120],
    ])

 @staticmethod
 def to_vector(exemplar):
    """
    Convert the matrix representation of the exemplar to its vector form (100 elements)
    :param exemplar: 12x10 matrix form the exemplar
    :return: a 120 element list representing the vector for that exemplar
    """
    assert(len(exemplar) == 12)
    return list([col for row in exemplar for col in row])
  @staticmethod
 def add_noise(exemplar, p=.25):
    """
    Add random noise to an exemplar by flipping the value at each index (-1 => 1, 1 => -1) with probability p.
    :param exemplar: exemplar to add noise to
    :param p: probability with which to add noise
    :return: a copy of the exemplar with noise added
    """
    assert(len(exemplar) == 120)

    def flip_bit(x):
        i = random.uniform(0, 1)
        if i <= p:
            return x * -1
        else:
            return x

    noisy_exemplar = copy.deepcopy(exemplar)
    return [flip_bit(x) for x in noisy_exemplar]
```

Appendix C: Random Exemplars code

Generation of random exemplars for experiment 2 was accomplished using the following code. The API exposed allows consuming code to request a random exemplar population of exemplars having a requested *length*. The generation of a population occurs as follows:

- Generate all numbers on the range 0 to 2^(n-1), where n is the length of the exemplar and therefore the number of nodes in the Hopfield network it would be applied to.
- Select *num_exemplars* exemplars without replacement from the population.
- Convert each selected exemplar to a bit vector.
- Convert each bit vector to bipolar values, switching any 0s to -1.
- Left pad each vector until it is of length *length*.
- Return the population of bipolar encoded vectors as the exemplars.

There is a risk that some randomly generated exemplars share great similarity between one another, thus the use of multiple trials in experiment 2.

```
import random
 class RandomExemplars(object):
  """
  Generates the exemplars for the numbers 0 through 8, returning them as either 10x10 matrices, or
  vectors of length 100.
  """
   @staticmethod
  def get_exemplars(length, num_exemplars, randomize=False):
     """
     Generates num_exemplars bipolar exemplars as a list of vectors of length length.
     :param length: length of the vector
     :param num_exemplars: number of exemplars vectors to generate
     :param randomize: randomly select the exemplars from the interval [0, 2 ** num_exemplars).  Otherwise,
     exemplars are chosen from the range [0, 2 ** num_exemplars) with step = (2 ** length / num_exemplars)
     :return: a list of exemplars
     """
     # Cannot generate more exemplars than nodes
     assert(num_exemplars <= length)
      if randomize:
        population = range(0, 2 ** length)
        samples = random.sample(population, num_exemplars)
     else:
        step = (2 ** length) / num_exemplars
        samples = range(0, 2 ** length, step)
      def bipolar_encoding(sample):
        # convert the binary representation of the sample to bipolar values
        encoded = [1 if i == '1' else -1 for i in bin(sample)[2:]]
         # pad the array up to the required length
        while len(encoded) < length:
           encoded.insert(0, -1)
         return encoded

 return map(bipolar_encoding, samples)
```