# Computational Investigation of Neural Networks Using the Hopfield model

Evan Breitsch

Allegheny College

Department of Physics

Spring 2014



# Computational Investigation of Neural Networks Using the Hopfield model

Signature:_____

Evan Breitsch

Allegheny College

Department of Physics

Spring 2014

Abstract

Neural networks draw inspiration from the structure of our brains, and they provide a unique alternative to finding solutions where traditional computational methods fail. The study of neural networks reaches into a variety of disciplines like economics, biology, sociology, physics, and medicine. The interconnected neurons of the brain provide the inspiration for artificial neurons that can be connected in different ways to enable management and recognition of input. Analysts can use neural networks to study the behavior of stocks for predictive assessment, and computational biologists can use them to model cancer growth based on nutrient acquisition (de Oliveira et al. 2013, Beaudin 2007). This senior composition aims to

use computational methods to investigate the Hopfield Model. Advancing the understanding of the capabilities and limitations of the Hopfield neural networks will allow us to realize and properly assess the possible applications of the model.


## Acknowledgments

I would like to thank my first and second readers, Dr. Rahman and Dr. Lombardi, for their kindness, patience, and encouragement throughout my years and Allegheny College and especially during my senior year. I would like to extend thanks to my brothers for helping me learn to program in Python, and to my parents for their support and encouragement.

## Table of Contents


**Introduction** ..... 2

**Why neural networks?** ..... 3

**A Biological Neuron and Its Network** ..... 4

**Artificial Neural Networks** ..... 8

- **A single layer perceptron** ..... 10
- **A Multilayer Perceptron Solution** ..... 13

**The Hopfield Model** ..... 17

**The Ising Model** ..... 27

**Computers with Neural Networks** ..... 29

**Methods** ..... 31

- **The 1D Hopfield network program** ..... 32

**Results** ..... 49

**Conclusion** ..... 52

**Future Work** ..... 53

**Bibliography** ..... 54

## Introduction

Traditional computational methods often require an impractical number of iterations to find the solution to an optimization problem. However, an approximate solution can be achieved by processing inputs from the system in question using a network of perceptron units. These perceptron units may interact in a variety of ways to good method of approaching a solution for an optimization problem. If we think of the problem as recognizing patterns in hand written notes, we can define our system as the letters contained in the notes. To our highly parallel processing brains, interpreting the scribbled text might seem like quite a trivial task, but, for an artificial system, recognizing text can be computationally burdensome especially for the rigid classification structures typically used for computers. Conventional computers do not process information in parallel like we do, but instead perform operations serially (one operation after another). Computers are also limited by the way they store information using location-addressable memory. Serial processing and location-addressable memory help computers excel at computationally simple processes like the precise multiplication of large numbers easy but fall short with complex processes like text recognition.

Neural networks are simplified models of the brain's neurons, which are more adapted for recognition tasks. These simplified models incorporate the brain's parallel processing structure, yet simplify the way individual neurons process information. Neurons connect with weighted influence on one another to form a

network. This weighted influence is analogous to the way neurotransmitters and receptor proteins relate to transmit signals between nerves at the synapse. The coordinated action of nervous signals in the brain allows us the ability to learn, recognize, and process information. The act of learning to read requires the ability to learn and recognize the symbols comprising words of meaning. The English language alone consists of 26 letters that must be distinguished to interpret text--setting up an appropriate network of perceptrons allows us to teach a neural network program to correctly identify each of these characters and even group them into recognizable words (Davis 2010). Neural networks provide a means to break conventional processing and storage methods that limit computers capacity process information most easily handled by our brains.

## Why neural networks?

Computers manage to handle numerically intensive tasks, but our biological brains far outreach the ability of even the most advanced computers when it comes to basic cognitive functions: pattern recognition, learning, and communication. Popular science fiction's portrayal of the future maintains the ubiquitousness of artificial intelligence for customary use in automatons, human interaction with computers, and transportation. The level of performance for automatons featured in popular science fiction must have an adaptable self-learning algorithm for them to exhibit humanoid functionality envisioned by our imaginations. In order to communicate with humans, robots would require advanced communication skills to understand the vast number of dialogues used by humans. Current technology

requires great care in pronunciation on the user's part when delivering commands to voice recognition programs like Siri as part of the iPhone. However, future advances in voice recognition should pick up on every dialect with ease by learning the various nuances of conversation like a child does in the early stages of development. In order for robots to help in everyday tasks, they would require advanced ability in pattern recognition to identify their surroundings and understand the functions we would like them to perform. In fact, most of the attributes of a futuristic robot depend on the computing advantages of neural networks to make them a reality.

Beyond applications in robots, neural networks play a significant role in the modeling of many complex dynamical systems (systems that change with time) with seemingliy unsolvable differential equations: models of the stock exchange, cell culture, cancer growth, foam flow, sociological interactions, (de Oliveira et al. 2013, Beaudin 2007) etc.

## A Biological Neuron and Its Network

Neurons make up the nervous system and serve to relay information about the external world and the body's state to the brain for processing. For example, the pain sensed from stepping on hot sand at the beach on a sweltering summer day travels from the thermoreceptors in the skin and up the axons leading to the brain. Along the way these sensory nervous signals take synaptic stops at interneurons that act as a terminal for the activation of a withdrawal reflex from the hot sand (Fox 232). The interneurons may be the source of the first signals to the muscles for

reflexive movement, but further processing of the pain signal occurs in the brain. Functional magnetic resonance imaging tests will detect parts of the brain that activate when receiving nervous signals, such as pain, from peripheral senses and in cognitive thought processing (Chialvo 2010). Once our brain recognizes the cause of the pain and identifies the safety of the cool wet sand, we decisively hurry ourselves toward the shade to remedy the discomfort.

The biological neurons that transmit nerve signals are comprised of four components: the soma (cell body), dendrites, synapse, and axons (Figure 1).

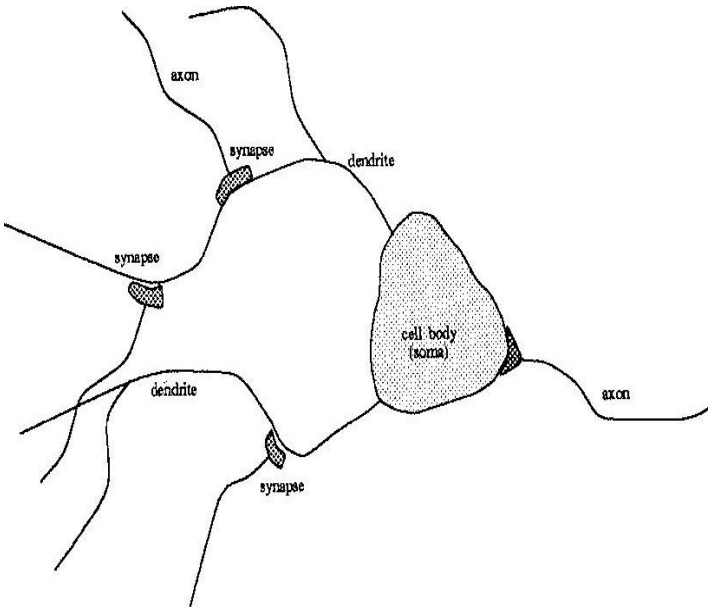


**Figure 1** Parts of a neuron: dendrite, synapse, cell body (soma), axon. (Beale 1990)

In the nervous system, information propagates as an action potential, a chain reaction of ion influx across the nerve cell membrane of the dendrites or axons. The action potential signals traveling to the soma are received by dendrites or by the soma itself. The soma will only activate when the input (total action potential) received by the soma is sufficiently large at a particular instant. The action potential traveling from the dendrites to the soma is controlled by interaction with axons of other somas in a region called the synapse. The synapse is comprised of the terminal end of an axon and the portion of intercepted dendrite (Figure 2).

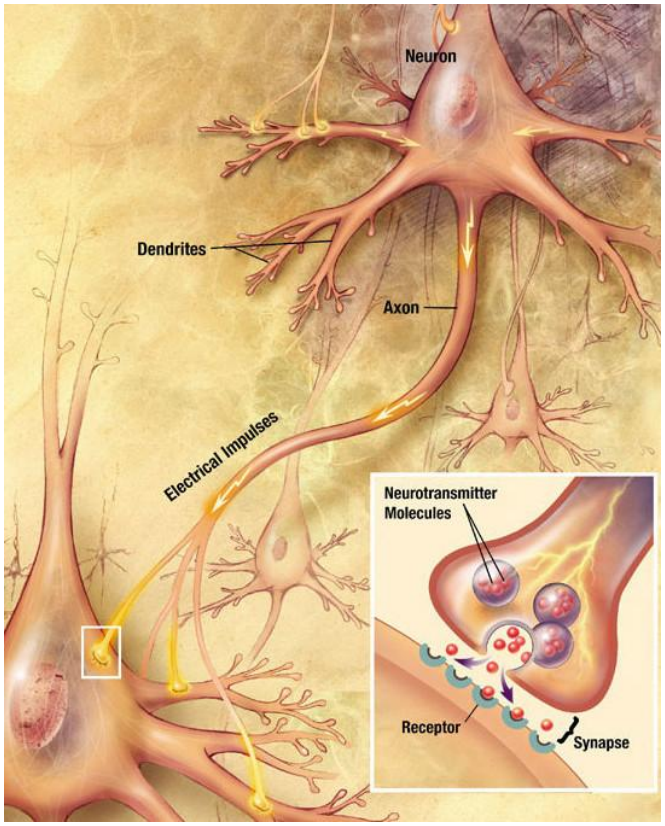


**Figure 2** The terminal end of the axon releases neurotransmitters when an action potential (electrical impulse) is received. Neurotransmitters cross the synapse and act on dendritic receptors of the post-synaptic cell (Wikimedia Commons).

Specific chemicals called neurotransmitters are released when voltage gated calcium channels, proteins that control calcium ion exchange across the cell

membrane, are activated by an action potential terminating at the end of the axon. These neurotransmitters traverse the gap known as the synaptic cleft between the terminal end of the axon and dendrite. The neurotransmitters diffuse through the synaptic cleft to the dendrite where they activate neurotransmitter receptors on its surface. The number of receptors and the amount of neurotransmitters released determine the strength of the interaction between two neurons (Kruger 2011).

Synapses may either function to enhance or limit the electrical potential difference of positively charged ions across the membrane of dendrites emanating from the soma (Beale 1990). The electrical potential of the nerve cell results from the complex and often competing actions of regulating ion-transport proteins in the nerve cell membrane. The soma initiates an action potential at the axon hillock when enough voltage gated channels open to raise the electric potential above the activation threshold. Typical values for thresholds fall around -55 mV with a resting membrane potential near -70 mV (Fox 176). The action potential travels down the axon from the axon hillock to intercept the postsynaptic dendrites across the synaptic cleft.

The high connectivity of neurons and their varied sensitivities to other connected neurons provide the mechanism for memory formation and processing in the brain. The biological neural network that makes up the brain is immensely complex, but we can simplify the system to only its most basic functionality in order to elicit similar functionality in artificial neural networks.

## Artificial Neural Networks

Computer models that use networks of artificial neurons (perceptrons) attempt to recreate the organization and functioning principles of neurons in the brain.

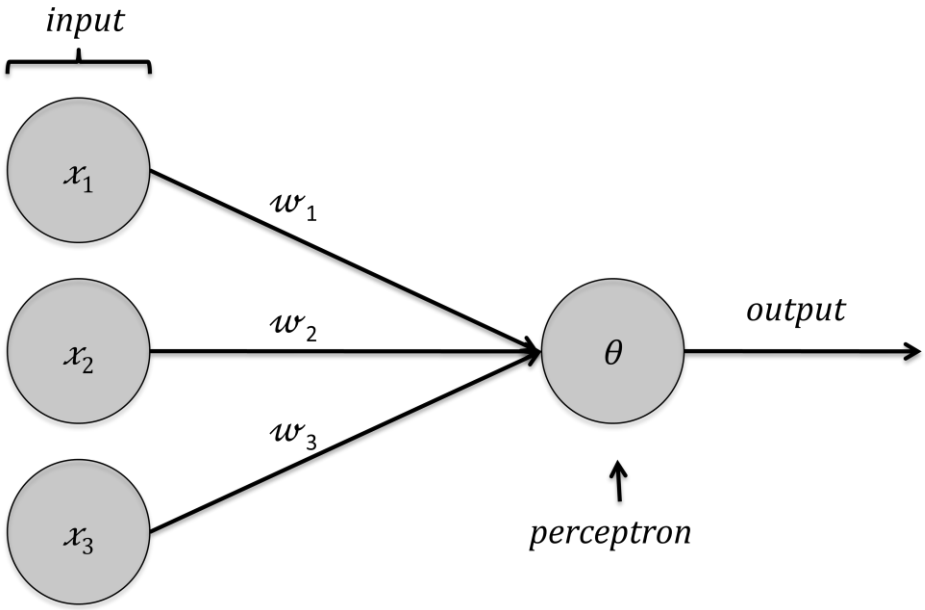


**Figure 3** An example for the single layer perceptron model. The perceptron of the single layer model is also the output unit, meaning there are no hidden layers. The nodes (grey circles) labeled with $x_i$ designations are the inputs to the perceptron of threshold $\theta$. The input notes influence the activation of the perceptron with a weight $w_i$. The unit will turn on and send output if the weighted sum is greater than or equal to the threshold.

Perceptrons function similarly to the soma and activate only when the sum of all its inputs reaches a certain threshold $\theta$, just as the signal received from the branching dendrites activates the soma (Figure 3). McCulloch and Pitts proposed these types of perceptrons in 1943.

Once activated, the perceptron will send out a signal to any other connected perceptron units, or else generate the final output of the network. Biologically, the concentration of neurotransmitters released and number of postsynaptic receptors define the weight between two neurons. For an artificial neural network, the

amplitude with which the signal is sent is proportional to the weight of the connection between the perceptron units. Input units distribute input to the system, while the output units acts as the terminating units system. A biological example of input units might be the photoreceptors (light detecting cells) of eyes that detect visual sensory input. These photoreceptors send a signal for processing in the occipital lobe of the brain that terminates as visual perception. The processing units of a multilayered neural network, excluding the final output perceptron, comprise the *hidden layers* of the network. The activity of a *hidden layer* is not stored in the final output, nor is the state of its perceptrons exactly predicted by the input. Therefore, we can access information regarding only the input and the resultant output. One could relate this to the experience of recognizing the letter "T" without consciously identifying the symbol as two perpendicular lines in alignment with the orientation of the page. We intuitively know the phonetic identification of the letter without noticeable cognitive effort because of our extensive practice in learning to read. The simplified structure of a biological neural network provides an accessible model for use in computation while retaining the defining functionality of the nerve cell, its connectivity, and therefore the particular advantage of associative memory.

Neurons function rapidly ($10^3$ Hz) (Beale 1991), though not as fast as computers (at an order of $10^9$ Hz). Unlike conventional serially functioning computers, neural networks work in a complex interconnected fashion to produce a processing structure similar to the one used in the brain for cognitive functions of consciousness and physiological control. Processing by neural networks might be

though of as a "divide and conquer" approach rather than a single processor of information serially rushing towards a solution at a speed six orders of magnitude, or a million times, faster. The "divide and conquer" approach provides the advantage of affording subtle differences in the management of input while still producing the correct output (Beale 4). Depending on the complexity of the system being modeled, even very small variations in input can produce substantial divergence in eventual output, as was most famously addressed by Poincare in his attempt to solve the "many body" problem (Mitchell 21, 32). Therefore, despite slower processing speeds, neural networks outdo conventional processes by finding stable outputs, (ideally), for similar inputs to a complex system.

**A single layer perceptron**

The single-layer network behaves similarly to how a single biological neuron would process incoming stimulus from its dendrites (Figure 2). The *N* number of inputs ($x_i$) to the each* perceptron unit have an associated weight ($w_i$) such that the weighted sum of the product of inputs and weights causes the unit to turn on if the sum is greater than or equal to the threshold, or bias ($\theta$):

$$\sum_{i=1}^{N} w_i x_i \geq \theta$$

The inputs ($x_i$) can have either values of 1 or 0, as characterized by the all or nothing signaling inspired by action potentials in the biological nervous system. We can simplify this inequality by adding a bias to the perceptron by subtracting θ from both sides

$$\sum_{i=1}^{N} w_i x_i - \theta \geq 0$$

We then incorporate the bias into the summation by extending the lower bound to $i = 0$ and setting $x_0$ to 1 and $w_0$ to $\theta$:

*The single layer model can have more than one output, but this output may not be combined for a secondary assessment. Doing so would form a new layer in the network. The perceptrons of a single layer model may have different inputs but must share at least one input or else the perceptrons would be independent of each other. If it is not connected then it is not part of the network.

$$\sum_{i=0}^{N} w_i x_i \geq 0 \quad .$$

The inequality now defines a condition, or voltage in the analogy of neurons, that must be met to activate a biased perceptron. Setting this condition to $x_s$

$$x_s = \sum_{i=0}^{N} w_i x_i \quad ,$$

we can give an expression for the *Heaviside function* $f_h(x_s)$ for all possible $x_i$ and $w_i$:

$$f_h(x_s) = f_h \left[ \sum_{i=0}^{N} w_i x_i \right].$$

The new expression gives the output evaluated by the Heaviside function $f_h$ such that

$$f_h(x_s) = 1 \qquad x_s > 1$$
$$f_h(x_s) = 0 \qquad x_s \leq 1.$$

The Heaviside function determines whether the unit is either on ($f_h$ = 1) or off ($f_h$ = 0) when the input to a perceptron exceeds the bias or threshold. The *Heaviside function* is computationally advantageous because it yields a bi-state non-linear output essential to logical operations (Hopfield 1982). Computers function on bi-state instructions such as binary code and often perform logical operations to provide non-linear output such as the Boolean values true (*value*=1) and false (*value*=0). Performing computation with non-linear output as binary reduces the memory needed by a computer program and can shorten run times, thus improving the program's efficiency.

The single layer perceptron allows for the discrimination between groups of linearly separable types on input. Different types of input to a neural network are linearly separable if they can be distinguished by a single decision boundary defined by the weights of the inputs (Figure 4).

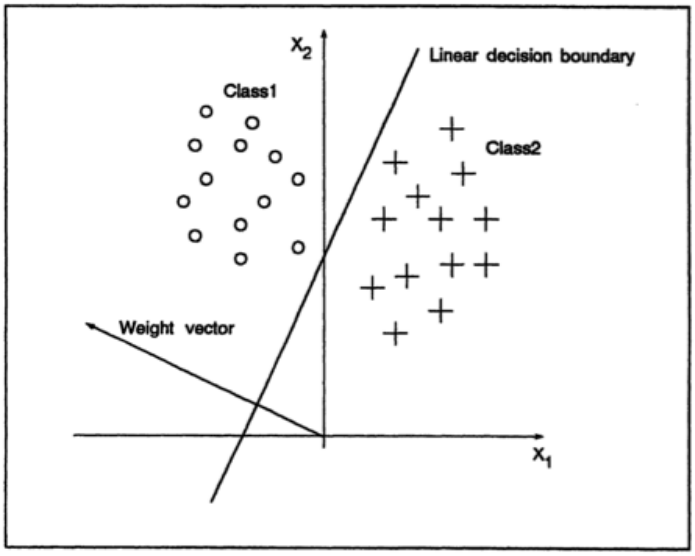


**Figure 4** Mapping of two types of inputs, or classes, that are characterized by 2 parameters, $X_1$ and $X_2$. These classes can be separated by a single linear decision boundary (a line for 2 parameters measured, a plane for 3 parameters measured, etc.) that is determined by the weights used to differentiate the classes. The weights for the input are described by the weight vector's components (Beale).

Linearly separable classes may be distinguished using a single layer model. However, only perceptrons of 2 or 3 layers may discriminate between linearly inseparable classes, those that may not be separated by a single decision boundary. (Figure 4). An example of linearly inseparable groups is the exclusive or problem (XOR). The exclusive or problem asks that the final output for the network should occur when one of the inputs is on while the other is off. No matter how we set the weights for a single layer network, we cannot solve the XOR problem (Beale 58).

## A Multilayer Perceptron Solution

Multilayer perceptrons, as the name suggests, have more than one perceptron unit processing the input to the network. Neural networks can be combined in up to 3 processing layers of perceptrons, which include all perceptron units except those distributing input to the network. Multilayer networks may be connected in a variety of ways (Figure 5) to solve a particular problem whose solution is optimized by that particular arrangement. It follows from proofs not shown here that no more than 3 layers are needed; this is because any number of layers greater than 3 would

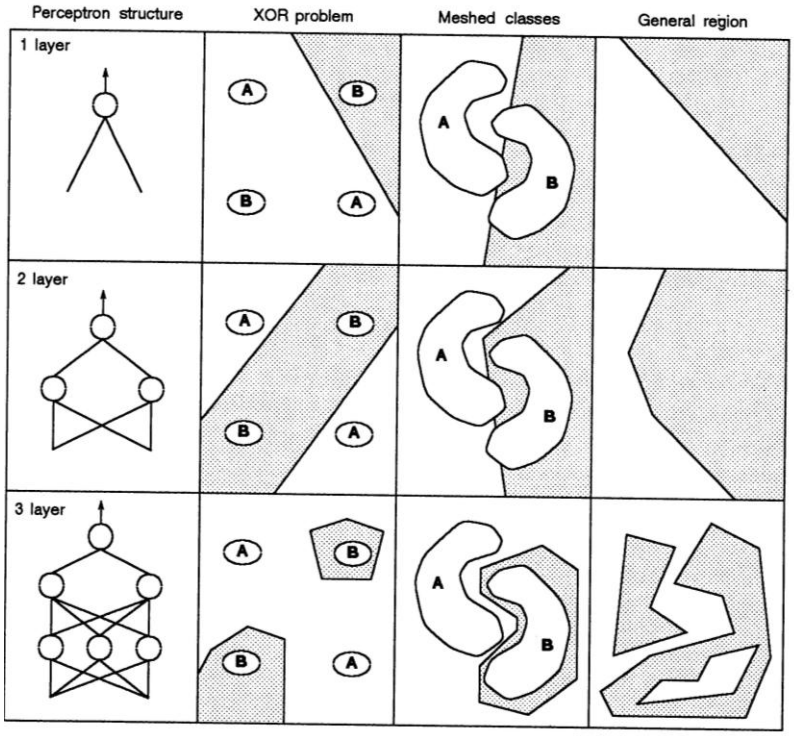


**Figure 5.** Using additional layers in a neural network solves the exclusive or problem. This chart shows how varying the number of layers changes the decision boundaries formed to classify the two classes **A** and **B**. The second column shows accessible solutions to the exclusive or problem, while the third column shows the performance of different layers for more difficult to resolve class boundaries (non-shaded region of bean shaped boundary enclosing class sets **A** and **B**) (Beale 84).

be degenerate with another arrangement of perceptrons of 3 layers or less (Beale 84).

To solve the XOR problem, the simplest solution is to arrange a 2-layer perceptron in which a hidden layer's output is effectively zero when receiving signal from both inputs (Figure 6). This allows for the segregation pictured for the 2-layer case in the XOR problem column of Figure 4. The network shown in figure 6 uses

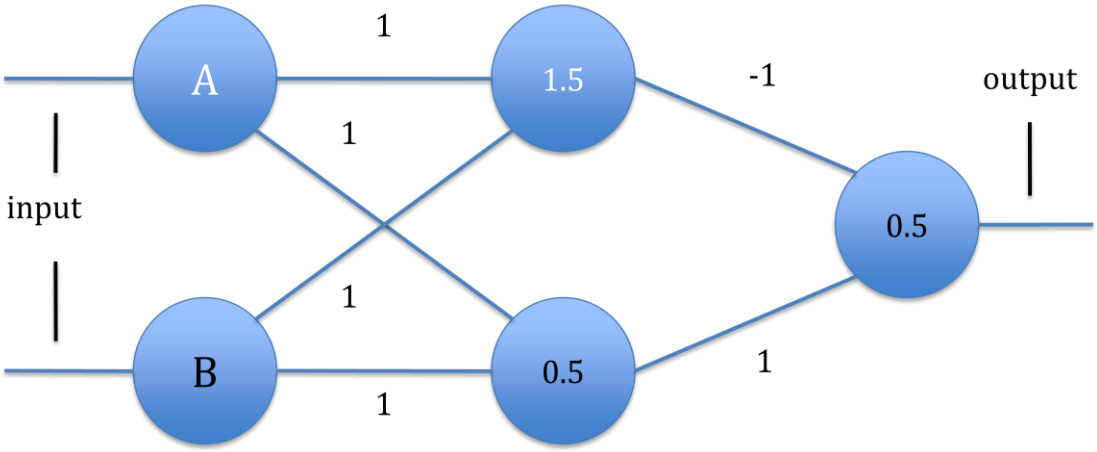


**Figure 6** Solution of the exclusive or problem using a 2-layer perceptron with perceptrons pictured as blue circles and lines representing the weights between them. The hidden unit shows a threshold of 1.5 that is reaches only if both inputs are active (Beale 78).

two layers of perceptrons with predetermined weights to process input from nodes **A** and **B**. If neither **A** nor **B** are active, then there is no input to the system and no output. Since both nodes are off and there is no output, this is a correct response to the XOR problem. If only node **A** is active, the perceptron with a threshold of 0.5 will become active one after the other because their threshold has been reached by net input to those perceptrons. The same is true if only node **B** is active. If both nodes **A** and **B** are active, the first-layer perceptron with a 0.5 threshold will still activate. This time however, the perceptron with a threshold of 1.5 will also activate since the

input to both perceptrons of the first layer is 2. The output from these first layer perceptrons cancel because the top weight contributes a -1 and the bottom weight adds +1 to the input of the final output perceptron. With zero net input, the final output perceptron remains inactive and we have a complete solution for the XOR problem. We can achieve solutions to many other nonlinear problems by using different combinations of weights and connectivity between perceptrons of 2 or more layers. The three-layer perceptron models offer the highest resolution between two classes with boundaries defined by convex hulls (arbitrary n-sided shapes) as shown in the comparison of meshed classes of Figure 4.

## The Hopfield Model

In his 1982 paper, John Hopfield presented a revised theory on neural networks and made an important connection between his model and that of Ising and Lenz. Hopfield's 1982 paper gained popularity because it presented a viable means of storing content-addressable memory, a particularly important feature for artificial intelligence. This type of memory could allow for mutations or small variation in the input while still producing the desired output. Hopfield draws an analogy for content addressable memory by the way DNA, despite random mutations, encodes information for the necessary proteins in a living organism (Hopfield 1982). We might also describe content addressable memory by the way we recognize a familiar person by only seeing half of their face. Although we only see an incomplete image of the person, we are still able to process just enough information to correctly recall the identity of someone we know. This robustness of the Hopfield model to noisy or

even incomplete input makes it a valuable asset for artificial intelligence. Although grossly simplified in structure and function, the Hopfield model is comparable to a network of neurons recalling information in the brain, and it is no surprise that they both feature the same advantages for learning and recognizing patterns. The popularity of the Hopfield model should be attributed to its feature of content addressable memory and similarity to networks of neurons in the brain. We shall see in a later section how the Hopfield model also draws the attention of physicists by its close association with the well-studied Ising model.

Before the introduction of the Hopfield model in 1982, there had been significant research on mainly feed-forward networks for processing information in unidirectional flow, e.g. $A \rightarrow B \rightarrow C \rightarrow D$ (Hopfield 1982). Contrastingly, Hopfield's model makes use of recurrent neural networks and their feature of strong backward coupling. The neural network of the Hopfield model is fully recurrent, such that every neuron is directly connected to all other neurons (Figure 7).

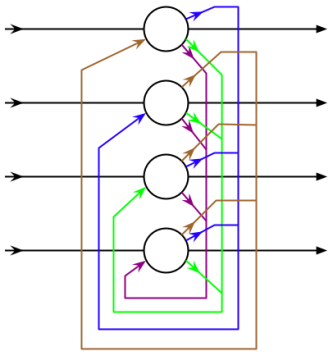

**Figure 7** The Hopfield network is typified as a recurrent network. The colored lines show the possible paths of the directed cycle for interactions between the neurons of a recurrent network.

The neurons of recurrent neural networks generate output based on the internal "energy" of the network due to the interactions of neurons amongst directed cycles. These directed cycles are some path that a neuron's output would take to return as input to that same neuron, thereby establishing backward coupling within the network. For example, the blue and brown paths of figure 7 correspond to the upper most neuron's output and input respectively, and together comprise the directed cycles for that neuron. Without the brown paths, the upper most neuron in figure 7 would not exhibit backward coupling and would not be affected by the states of the other neurons. However, removing the brown paths would make the network only partially recurrent, which is inappropriate for the Hopfield model. Each interaction between neurons has a particular weight that determines the strength of the interaction, and as we shall see, the interactions of two neurons with each other ought to have the same weight.

Unlike the feed-forward networks of multilayer perceptrons, input to a Hopfield network does not actually propagate through the network to generate output. The closed loop structure of directed cycles in the Hopfield network will not allow for feed-forward processing. Instead, the Hopfield network alters the states of the neurons so as to lower the "energy" of the network. The learning algorithm, which is derived from the energy function for the Hopfield model, determines these weights before the network is used to recall learned patterns. The applied input to the Hopfield model defines the initial states of the neurons and may be a string of binary values made of 0's and 1's, e.g. (0,0,1,0,0,1,0), or bipolar values made of -1's and +1's, e.g. (-1,1,1,-1,1,-1,-1). The activity of a neuron depends on the value of its state. For

binary input, the state values of +1 are active and 0 are inactive; for bipolar input, the state values of +1 are active and -1 are inactive. Regardless, the pattern storing functionality of the Hopfield network remains pretty much the same. When recalling a pattern, the states of the neurons are assessed asynchronously, and only change when it is energy favorable to do so. The final output of the Hopfield network is the set of neurons' state values for a local energy minimum and is achieved when the neurons no longer change states with each update of the network, e.g. (0100001)→ *update neurons' states*→(0100001).

Updating all neurons at once, synchronously, can cause the system to resonate between two equivalent energy minima in state space, e.g. (0100001)→ *update neurons' states*→(1011110), where both sets of states have the same value. Synchronous updating may be used cautiously in the Hopfield model, but asynchronous updating agrees more with processes in nature, and addresses the problem of the network resonating between two equivalent energy minima (Hopfield 1982). It is also important to make the assertion that the neurons in a Hopfield network may not connect to themselves. Such connections would always act to increase the energy of the network for any pattern stored; therefore, we will prohibit any neuron directly connecting with itself.

The associated "energy" of a Hopfield network is not physical energy and is sometimes referred to as a constraint satisfaction. Hopfield networks reach the lowest 'energy' when there is maximum agreement between units. Maximum agreement depends on the weights between neurons as well as their states. Agreement within a Hopfield network occurs when a neuron is in a state that agrees

with its input; e.g. if a neuron is active and its input exceeds its threshold, the neuron is in agreement with its input. Greater agreement between neurons and their input lowers the energy of the network as determined by the constraint satisfaction. The constraint satisfaction is given by

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i \; s_i \quad ,$$

where $s_i$ and $s_j$ are the states (bipolar: ± 1 ; binary: 0 and +1) and $\theta_i$ is the threshold of the $i$th neuron. In this energy function, we should maintain that no neuron connects with itself $i \neq j$ and the network should be symmetric $w_{ij} = w_{ji}$. Non-symmetric networks can still recall patterns to an extent, however, more symmetric networks allow for better performance in pattern recall (Hopfield 1982).

To discuss the energy of the network, we will choose to handle bipolar, as opposed to binary, values of input and output for simplicity. We will also choose to set the threshold for each neuron to zero. Applying a threshold to a neuron may be energetically advantageous to one pattern, yet it may increase the energy of a subsequent pattern (Beale 139). The application of thresholds to neurons would make the network biased toward learning only certain patterns, and to avoid this we will set the threshold of each neuron to zero thereby removing the threshold term from the constraint satisfaction,

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j \quad .$$

The new equation for energy shows that disagreement between neurons $i$ and $j$ increases the system's energy when the weight $w_{ij}$ between them is positive. However, negative weights $w_{ij}$ would cause the energy of the system to increase

when $s_i = s_j$. As we will see in the next section, the interaction of neurons' states based on determined weights is strikingly similar to the way magnetic spins interact in the Ising model.

Storing patterns into a Hopfield network involves applying modifications to the network's weights $w_{ij}$ in the connection matrix. In order to store many patterns of bipolar values into the Hopfield network, the energy associated with each pattern should be as low as possible (Beale 139). Recall that the energy of the network is determined by the neurons' states and interaction weights. During pattern recall, the Hopfield network is given input and the neurons' states are changed one at a time to lower the energy of the network. The lowest achieved energy corresponds to a set of the neurons' states that had been previously stored. Since the network recalls patterns with the lowest energies, we ought to store patterns using appropriate weights for the lowest possible energy. The weights for each newly learned pattern are added to the working weights of the connection matrix for the network. In this way, the connection matrix represents the sum of all respective connection weights for each pattern learned (Figure 8).

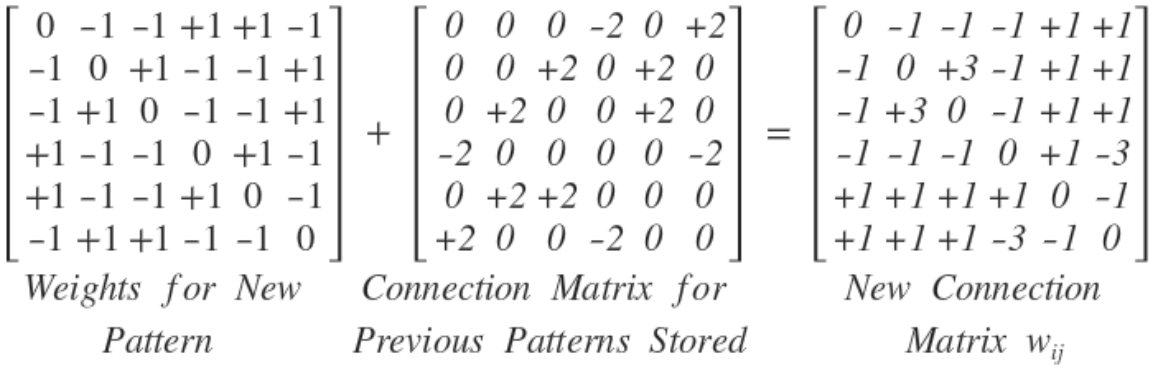


**Figure 8** Example of adding weights for a new pattern to the connection matrix weights $w_{ij}$.

In order to show how to store the weights for a new pattern in the network we will split the energy function into 2 parts

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}^{s-1} x_i x_j - \frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}^{s} x_i x_j .$$

The term on the right can be thought of as a signal term representing the energy contributed to the network from learning a new pattern *s*. The term on the left is the energy of all previously learned patterns *s*-1, which can be thought of as a noise term that could interfere with the eventual recall of pattern *s.* Changing the left term would undo the previous storage of patterns, but the right term may be altered by choosing effective weight contribution $w^s_{ij}$ to minimize the energy contribution from the storage of pattern *s*. Although we could choose any real number for the weights, using a +/-1 increment for the weights provides a good balance to both store and recall patterns from the network (Hinton). We will want the signal term on the right to be negative in order to reduce the energy of the network for each state value of pattern *s*. Therefore, we ought to set the weights of the right term to the product $x_i x_j$:

$$w_{ij}^{s} = x_i x_j .$$

This rule has the effect of making the summation of the right term in the constraint satisfaction always positive, which reduces the energy contribution of learning pattern *s*. We will use the equation above as the learning rule for all patterns learned by the network.

Recalling patterns from the Hopfield network requires applying input to set the initial states of the neurons in the network, and then allowing the network to update these weights to minimize the networks energy. We can rearrange the previous equation to show the change in energy of the network by altering the state of a particular neuron $x_k$,

$$\Delta E = -\Delta x_k \left[ \sum_j x_j w_{kj} \right] .$$

If the neuron changes state from +1 to -1, $\Delta x_k$ will be negative. Conversely, the state change, $\Delta x_k$, is positive if the neuron changes from -1 to +1. An energetically favorable change of state depends on the k-th neuron's connections with other neurons in the network. The other neurons in the network influence the state of neuron *k* with an effect proportional to their connection weight with neuron k, $w_{kj}$. The sign of neuron *k* should therefore agree with the weighted average of its connections with the rest of the network (Figure 8). This agreement would correspond to a negative change in energy, which is energetically favorable to the Hopfield network.

| | |
|---|---|
| $\sum_{j \neq k} x_j w_{kj} > 0$ | $x_k = +1$ |
| $\sum_{j \neq k} x_j w_{kj} < 0$ | $x_k = -1$ |
| $\sum_{j \neq k} x_j w_{kj} = 0$ | $leave\ x_k\ in\ current\ state$ |

**Figure 9** The rules above dictate the energetically favorable state changes for the

As we mentioned earlier, the Hopfield network typically updates the neurons' states asynchronously until the energy can no longer be reduced. Algorithmically speaking, the network should first choose a random neuron in the network to update and change the state of that neuron if it is energetically favorable. The network should continue randomly choosing and updating the remaining neurons until all neurons have been updated. The full update of all neurons in the network represents an iteration of updating for the network. Iterations of updating continue until any subsequent iteration will not reduce the energy of the network any further. The set of neuron states after the final iteration represents a previously learned pattern that is most similar to the initial input to the Hopfield network.

The Hopfield network provides a robust method of storing and accessing content addressable memory. However, error in recall could occur when too many patterns are stored. The length of the patterns stored, i.e. number of neurons in the network,

determines the storage capacity for a Hopfield network. Errors in recall typically occur when the number of patterns stored is greater than 10-15% of the *N* number of neurons in the network (Beale 144). The energy minima in the network's state space, which represent patterns stored in the network, tend to drift away from their locations in state space. More specifically, errors in recall begin when *M* number of patterns exceeds the ratio of *N* neurons over the log of *N* neurons (Anderson 411),

$$M \propto \frac{N}{\log N}.$$

The high dimensionality of the Hopfield network's state space prevents direct visualization of the energy minima, but we will visualize a projection of the state space to help understand drift (Figure 10).

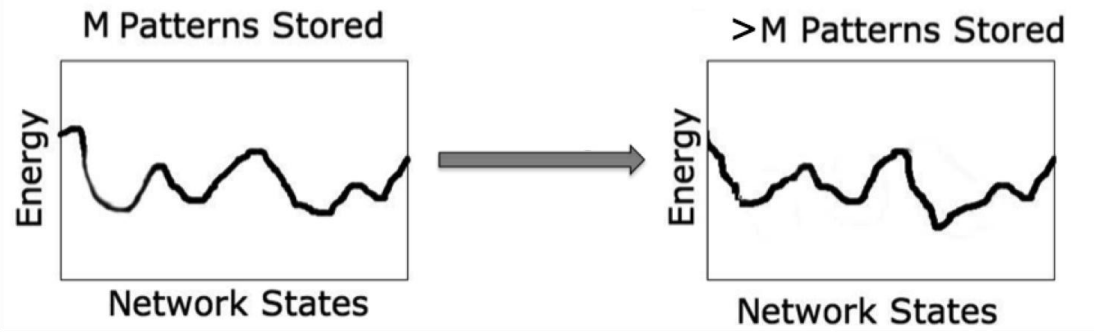


**Figure 10** As a Hopfield network learns more patterns, the local minima for network energy begin to drift away from their original locations in state space. These drifts account for error in pattern recall, which typically begins when *M* is nearly 10-15% of the number *N* neurons in the network.

The two energy landscapes of figure 10 are different in appearance but appear related so as to indicate the initial drift of the energy minima. Energies associated with previously learned patterns may change due to the additional learning of patterns. Previously learned patterns may no longer represent minima in the new energy landscape once the number of patterns exceeds the network's capacity.

## The Ising Model

A general study similar to Hopfield neural networks is the Ising model. This model can be any system of two possible states that is mapped into an array. The array can have numerous dimensions and interacting parameters to model some process involving the transition of microstates. By using an approach to neural networks modeled after the Ising model, we can investigate phase transitions of the neural system (the system energy minima or the maximum agreement between the units' assessment of the input). John Hopfield's 1982 paper noted the similarity between his proposed neural network the Ising model (Hopfield 1982). Thus, Hopfield ushered the idea that neural networks could be modeled using a perceived energy of agreement among networks of neural units. Hopfield's findings enticed the interest of many physicists to join the pursuit of a general solution to neural networks since the Ising model had been studied for more than 60 years by that time. However, neural networks provided a more difficult challenge than anticipated, and it was not long before many of the physicists moved on to some other low hanging fruit.

The connection drawn by Hopfield and the advantages of using the well-studied Ising model for phase transitions make such a model the ideal perspective when studying the nonlinear system of a neural network. Nonlinearity refers to the way in which the output is not directly proportional to the input of the neural network. The Ising model may be applied to any system comprised of interacting units with two possible states that can be mapped to an array. Wilhelm Lenz originally proposed the model to his graduate student Ernst Ising for the study of paramagnetic phase

transitions, but the model has since been used to explain numerous phenomena including the infamous traveling salesman problem.

A useful way to understand the Ising model is its application to spin states resulting in bulk magnetization which functions similarly to the modeled neurons of a neural network. In a ferromagnetic material, such as iron or nickel, the spins of unpaired electrons act as current loops to produce small magnetic dipoles. Each magnetic spin can be modeled as a bi-state system with an array of spins pointing either parallel or anti-parallel relative to the external magnetic field *B* (Figure 9). The summation of *N* magnetic spins ($s_m = \pm \frac{1}{2}$) leads to the bulk magnetization *M* of the paramagnet given by

$$M = \sum_{i=1}^{N} s_i .$$

The magnetic spins will experience a force due to the magnetic field *B* that gives rise to a net torque

$$\vec{\tau} = \vec{\mu} \times \vec{B}.$$

There is a local magnetic field strength *H* in the presence of an external magnetic

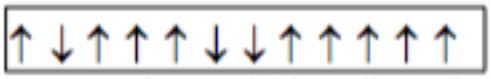

**Figure 9.** The arrows shown represent a set of spin-states for a 1 dimensional

field. The local field strength *H* influences the orientations of the magnetic spins such that they tend to point parallel due to an external field in order to lower the

energy of the system. When evaluating the energy in the Ising model, the magnetic spins of electrons for a ferromagnetic material have two states, spin up ($s$ = +1) and spin down ($s$ = -1). We can model the energy of the system by considering the spin states for $N$ electrons and their nearest neighbors,

$$E = -J \sum_{i,j=\mathrm{nn}(i)}^{N} s_i s_j - H \sum_{i=1}^{N} s_i ,$$

where $J$ is the interaction exchange energy and nn($i$) are the nearest neighbors of $i$th electron. Notice that the right hand summation minimizes energy when $s_i$ points parallel ($s$ = +1) with $H$, and the left hand summation tends to lower energy when both $s_i$ and $s_j$ are oriented in the same direction. The lowest energy state of the system would then be one in which all spins point parallel with external field $H$.

Here it is important to consider the boundary conditions of the system where spins at the edge of the array have fewer neighbors. One solution is to use a periodic boundary condition with a finite number of spins to simulate an infinite lattice. The periodic boundary condition functions such that the $n$th spin interacts with the 1st spin for that row or column. The periodic boundary condition is implicit in the organization of the Hopfield model since each perceptron unit interacts with all other units. The Ising model may be generalized for more than 2 states. This is known as the Potts model.

## Computers with Neural Networks

The artificial neural network models are applied with the intent to perform mostly associative memory based tasks. Unlike the typical memory of a computer,

associative memory evaluates an input of strings made of 0's and 1's to determine the similarity of the input to other previously stored memories. Traditional computer storage has addressable memory that is accessed for precise cases of 0's and 1's. In contrast, the associative memories of a neural network are accessed in a content addressable way by assessing recognizable patterns in the string and accessing the memory closest to the input pattern (Gould, 1996). Table 1 shows the accessed memory recalled from an input string of bits; the errors from attempting to recall string 1 are highlighted in yellow.

| String | Stored Strings of Bits |
|---|---|
| 1 | 1110001110 |
| 2 | 0001110001 |
| 3 | 1100110011 |
| 4 | 0011001100 |
| 5 | 1111100000 |
| 6 | 1111000011 |
| 7 | 0000111100 |
| 8 | 0000000111 |
| 9 | 0000111000 |
| 10 | 1111111000 |

| Input: | String Retrieved: |
|---|---|
| 1110001110 | 1111000110 |

**Figure 10.** Stored memory and an example of retrieved memory based on an input.

The neural network algorithm was adapted from *An Introduction to Computer Simulation Methods* (Gould and Tobochnik, pgs. 527-530).

In this way, information input undergoes a multitude of small judgments and processes before emerging as a recognized memory, or one close to a stored memory. This is not to say that the emergent memories driven by an input's content depends on exact sequences of 0's and 1's. The content accessed depends on the probability that the input shares attributes of that content. It is also not surprising here that the accessed memory does not match the input to the network since the

ratio of memories stored to string length is 1 (100%), much higher than the cut-off determined by Hopfield (13.6%).

## Methods

Harvey Gould introduces the algorithm for a 1-dimensional Hopfield network in his book, *An Introduction to Computational Simulation Methods*. In this study, the algorithm was modified and written in the Python programming language (Appendix 1). The code stored binary strings of equal length as patterns in the connection matrix $w_{ij}$ with dimensions of the string length squared. The Hopfield model's dependencies on pattern size (string length) and number of patterns stored were assessed by determining the Hamming distance, i.e. the number of bits that differ between the input and output of the network. For example, the string retrieved in Figure 10 has a Hamming distance of two in relation to the input string. It should be noted that the program adapted from Gould's algorithm does not include an optimization procedure. The program used in this work conveys the initial response of the network to a previously stored input. However, this work tests only previously stored input, and it is reasonably expected that the program will correctly recall patterns that were previously learned. Any deviations between input and output to the network would be the result of "noise" in the network related to the number of patterns stored and the length of those patterns. Excessive "noise" beyond the storage capacity of the Hopfield network may cause network energy minima to drift in state space. These drifts contribute to the observed deviations between input and output that are measured as the Hamming distance.

Measurements for the Hamming distance take place after only a single iteration of updating the network's neurons. For the purpose of this study, one of the initial patterns stored in $w_{ij}$ was randomly chosen as the input to the network in order to calculate the Hamming distance from previously learned patterns. We might not expect the pattern retrieved to match the input memory of the system if the memory had not been processed previously, so a previously stored memory was used.

### The 1D Hopfield network program

The Hopfield modeling program, Hopfield_1D.py, will allow us the ability to observe hamming distance behavior for many combinations of *pattern lengths* and *number of stored patterns*, e.g. all conditions from 1-50 *number of patterns stored* for each *pattern length* from 1-100 binary states. The version presented here uses built-in array handling methods of the NumPy library. The code has been specifically run on python interpreter 2.7, and may not be syntactically correct for all python interpreters. NumPy is designed specifically for handling multi-dimensional ndarrys (*n*-dimensional arrays), and allows for direct linear algebra operations.

The Python interpreter reads down the script and handles the code listed from the top first.

```
 8 import numpy as np
 9 import os
10 import time
11
12 maxNumpat = 40
13 maxPatlen = 20
14 numberOfTrials = 1
```

The interpreter will first import the library for NumPy and we will call it *np* in the code that follows. Importing *os* allows us to handle files on the computer directly

when we save the hamming distance data, while *time* provides access to peripheral methods we will use to tract processing times. In lines 12-14 the global variable (constants used in many methods of the code) *maxNumpat, maxPatlen* and *numberOfTrials* correspond to the variable for the maximum number of patterns we wish to test, maximum length of a pattern, and the number of trials to take respectively. Each trial will use different randomly generated sets of patterns to test. Taking multiple trials for each set of *maxNumpat* and *maxPatlen* will allow us to find an average over the number of trials taken (5 shown above).

Now that we've imported the necessary libraries and set the conditions for running the program our attention must move to the function definitions (def functions).

```
def main():
def memorize(patList, numpat, patlen):
def recall(memory, targetPat):
def saveHamDistances(avgHamDis):
def showPlot(data, title):
```

The def statements listed above are all the functions used in the program. The def main() statement contains the bulk of the code and is processed first because of the typical conditional statement written at the end the program shown below:

```
if (__name__ == "__main__"):
    main()
```

This statement tells the interpreter to run the def main() function first. The other functions are appropriately called from within the main() function and will be covered after first looking at main().

The main() function will serve as the main script for our program to follow. Important arrays and constants are declared and assigned in first few lines of the main() function:

```
16 def main():
17     t0 = time.time()
18     avgHamDistances = np.zeros((maxNumpat,maxPatlen))
19     avgErrors = np.zeros((maxNumpat,maxPatlen))
20     seedForRand = 997 #1234
```

In line 17, *t0* is the value of CPU-time upon entering the main() function and will be used towards the end of main() to calculate the processing time for the program runtime. Time constants are created throughout the program to help measure improvements in efficiency when originally writing and revising the code. These time constants have been retained in the code explained here to give an idea of how improvements in efficiency could be measured during the coding process. In line 18, *avgHamDistances* is declared as a 2D array of size *maxNumpat* rows and *maxPatlen* collumns. The same declaration is given to *avgErrors*, but *avgHamDistances* and *avgErrors* exist as two different arrays for the two different types of data they will hold. Array *avgHamDistances* will contain the average hamming distance of the number of trials set by numberOfTrials in line 14 for each set of number of patterns and pattern length. The array *avgErrors* will contain the percent error in recall for each set of pattern number and length and is averaged over the number of trials for each set. The constant *seedForRand* will serve as a seed number used to generate random numbers and patterns for the pattern creation.

The for loops in the main() function allow the program to iterate through all sets of pattern sizes and numbers and the number of trails for each.

```
21      for numpat in range(1,maxNumpat +1):
22          print numpat, time.time()
23          for patlen in range(1, maxPatlen +1):
24              hams = []
25              for trial in range(numberOfTrials):
```

The for loop structure in Python syntax and for line 21 follows a statement equivalent to: for each value *numpat* in the *range* of values from 1 to (*maxNumpat* + 1), iterate the following code after the colon until *numpat* is equal to (*maxNumpat* + 1) and then leave the loop. Python for loops iterate over a set of objects, but in this case those objects happen to be the values 1 to (*maxNumpat* + 1). The equivalent statements for line 21 also applies to the for loops of lines 23 and 25, but the range of values are different for each as well as the variable name used to iterate over each range. The print statement on line 22 serves as a bench mark for us to know the progress of the code and provides which iteration of the number of patterns we are storing, *numpat*, as well as the time the iteration began using the *time.time()*method. The object *hams* on line 24 is a one-dimensional array that will be used to store the hamming distances for each trial. It is important to re-define *hams* with each iteration of *patlen* because a new array is needed for each new set of trials. These for loops set up the iteration we need to handle all sets of pattern size and number of patterns stored. The for loops also allow us to iterate the appropriate number trials in order to find an average over the sets of conditions. We only need to change the bounds of the *range()* method to look at particular sets of conditions.

The next step in the Hopfield_1D.py program is to generate random patterns to store. These patterns are 1D arrays of 1's and 0's. These patterns are generated in

lines 27-29. On lines 30 and 31 a target pattern is selected from these patterns and then stored as targetPat for input to the *recall*() function.

```
27                  randomObj = np.random.RandomState(seedForRand)
28                  seedForRand += 1
29                  patList = randomObj.randint(0,2, size=(numpat, patlen))
30                  randomInt = randomObj.randint(0, numpat, 1)
31                  targetPat = patList[randomInt][:]
```

A random type object, *randomObj*, must first be created to use the efficient and extensive library of NumPy's random number generator. The creation of this random object occurs on line 27 and is also the first instance of the super class NumPy, which we have dubbed *np*. Python is an object-oriented program in which we create instances of a class of objects we wish to exploit. To reach a subclass of a particular class, we must append the name of the class with the "." similar to the way directories are accessed in a computer through the use of a forward slash, "/". The random type object that we declare and define, *randomObj*, is the *RandomState* object of the *random* class that is a subclass of NumPy. Thought of another way, we access a particular subclass in the order NumPy→*random*→*RandomState* and define *randomObj* as an object of the *RandomState* sub-class. This random type object has methods associated with it that we wish to exploit in line 29 and 30. The creation of this object necessitates a seed number to be generated, *seedForRand*, that it will use to generate random numbers. The seed number is incremented with each trial loop in order to generate different numbers each time. In line 29, the method *randit*() of *randomObj* creates an array of a *numpat* number of randomly generated patterns each with a length of *patlen*. The first two parameters of the *randint*() method set the bounds of the random integers generated, such that the lowest and highest

values possible are 0 and 1 respectively. The next line also utilizes the method *randint*() and generates a random integer used to select a target pattern for later use. In line 31, *targetMem* is defined as the pattern located on the *randomInt*-th row of *memList*. Notice that this definition makes use of a special notation ([:]) in NumPy to select the entirety of the *randomInt*-th row of the 2D array *patList*. In Python, the row index corresponds to the first index of an array while the column index occupies the second.

Now that the patterns have been created, it is time to process them with the Hopfield 1D algorithm. The connection matrix of the 1D Hopfield model is a 2D array of size *patlen* by *patlen*, or length of the patterns squared. This 2D array makes

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$

**Figure 11** The Krocnecker product applies the necessary multiplicative operations to generate the connection matrix of a particular pattern of +/- 1's (Wikimedia Commons).

use of the Kronecker product to provide a connection matrix for each pattern (Figure 11). Each of these connection matrices are added together to yield the net connection matrix for all patterns stored. The Hopfield_1D.py program performs these operations in the memorize function called in line 34.

```
33                tm1 = time.time()
34                memory = memorize(patList, numpat, patlen)
35                tm2 = time.time()
36                memorizeTime = tm2 - tm1
```

The program knows that memorize is a callable function and that it requires 3 parameters: *patList*, the list of random patterns; *numpat*, the number of patterns to store; and *patlen*, the length of the patterns. Here *tm1* and *tm2* are used to store the cpu time before and after calling *memorize*() allowing us to measure the time taken, stored as *memorizeTime*, to process our. A new variable, *memory*, is created on line 34 but its assignment is not to the function *memorize* itself but instead the array returned from calling that function. Let's look at the memorize function and the array it returns.

```
67 def memorize(patList, numpat, patlen):
68     memory = np.zeros((patlen, patlen), dtype=int)
69     for pat in patList:
70 
71         zeros = pat == 0
72         pat = pat - zeros
73         tpat = np.array([pat]).T
74         relation = np.kron(pat, tpat)
75         np.fill_diagonal(relation,0)
76         memory = memory + relation
77 
78     return memory
```

The first thing needed is a place to store the net connection matrix, which we will call *memory* and define it as a 2D array (i.e. a matrix) filled with zeroes. Line 68 defines the matrix *memory* using NumPy's built in *zeros*() method, which requires parameters of size, *(patlen, patlen)*, and data type to be stored, *dtype=int*. The values that will be stored in the connection matrix are integers so the *dtype* should be set to *int*, or integer.

The for loop of line 69 iterates through each pattern, *pat*, in *patList* to process the patterns one at a time. Within the loop, a boolean array of the same length as *pat* is generated to identify the locations of 0's. Given that the value of *True* is +1 and *False* is 0, we can subtract the array of trues *zeros* from *pat* to change each '0' to a '-1'. This allows for much simpler methods of broadcasting the memory arrays, saving time wasted by inefficient iterations. We will examine this difference following the discussion of the *memorize* function above.

A transpose of the new *pat* array is set to *tpat*, a column rather than row vector, for multiplication in the line that follows. The Numpy function *kron()* takes the Kronecker product of *pat* and *tpat* and returns a 2D matrix which we will call *relation*. Next, the diagonal of *relation* is filled with zeros to finalize the creation of a connection matrix for *pat*. This newly formed connection matrix, *relation,* is finally added to the net connection matrix *memory*. This process continues for the remainder of patterns left in *patList*, after which, *memory* is returned to def main().

The Hopfield learning algorithm used for Hopfield_1D.py an exclusive-or-type processing for the generation of a connection matrix. Remember that the exclusive-or case allows for output in a neural network when either one node or the other is active, but not when both are active. However, in this case, the output of the exclusive-or process serves acts to inhibit association between two compared elements of a pattern. To generate the connection matrix we set its *i*- *j*-th element to +/- 1 depending: set to +1 if both values of the *i*-th and *j*-th elements in the pattern are equal; set to -1 if the values of the two elements are not equal. The process of changing 0's to -1's in the patterns as described earlier takes care of this exclusive-

or consideration. The correct value for the *i*- *j*-th element of the memory's connection matrix is the product of the +/- 1 values of the *memory* elements of index *i* and *j*.

| Connection Matrix Value | Product of Memory Elements |
|---|---|
| *relation*[i][j] = +1 | *mem*[i] * *mem*[j] = +1 |
| *relation*[i][j] = -1 | *mem*[i] * *mem*[j] = -1 |

**Figure 12** The exclusive-or condition applies implicitly by using the Kronecker product to find the connection matrix. The value of the *i*- *j*-th element in the connection matrix is set to +1 if the values of the *i*-th and *j*-th elements in the pattern are equal and set to -1 otherwise.

Using this exclusive-or inhibition effect we can simply take the Kronecker product of a memory, *pat* and its transpose, *tpat*, to obtain its corresponding connection matrix. Leaving the 0's in the pattern demands element-by-element iteration because there is no discretion in the products of 0 * 1 and 0 * 0. This type of iteration is shown below.

```
77 def memorize(memList, numpat, patlen):
78     memory = np.zeros((patlen, patlen), dtype=int)
79     for mem in memList:
80         i = 0
81         while (i < patlen):
82             j = 0
83             while (j < patlen):
84                 if (j != i):
85                     memory[i][j] = memory[i][j] + (2*mem[i] - 1)*(2*mem[j] - 1)
86                 j += 1
87             i += 1
88 
89     return memory
```

The version of *memorize*() shown above differs in its use of element-by-element iteration to form a connection matrix *memory* for the each pattern to be stored. In line 85, *memory*[i][j] is essentially defined to +1 if *i=j* and -1 otherwise. This operation only occurs if, by the condition in line 72, *j* does not equal, "!=", *i*. The

definition of the Hopfield case does not allow a node of a neural network to have a connection with itself, such that *i=j*. The iterative approach shown above is not necessary in high-level languages like Python. In fact, avoiding such iterations by using the built-in methods of libraries like NumPy allow for much faster processing. For comparison, creating an array of sequential integers from 0 - 10 million is ~25 times faster using the built in method, *arrange*(), for *ndarrays* of NumPy (Bressert 6). The program Hopfield_1D.py makes use of NumPy's *ndarray* and built-in methods as much as possible because of their efficiency when compared to the general iterative approach.

Once the net connection matrix has been set, we can perform recall to test for the associative memory performance.

```
39              tr1 = time.time()
40              hammingDistance = recall(memory, targetPat)
41              tr2 = time.time()
42              recallTime = tr2 - tr1
```

Again, lines 39, 41, and 42 provide the means of timing the recall process of line 40. The call in line 40 to the function *recall*() takes the parameters *memory*, the net connection matrix, and *targetPat*, the randomly selected memory from *patList*. The function *recall*() uses *targetPat* as input to the Hopfield network defined by *memory*. The function returns both the generated memory, *output*, and the *hammingDistance*, the number of mismatches between *targetPat* and *output*.

The *recall*() function makes use of the built in *ndarray* methods on every line to process *targetPat* and *memory*.

```
81 def recall(memory, targetPat):
82
83     output = np.multiply(memory, targetPat)
84     output = np.sum(output, axis=1)
85     negative_indices = output < 0
86     output[negative_indices] = 0
87     positive_indices = output > 0
88     output[positive_indices] = 1
89     misMatch = np.not_equal(output, targetPat)
90     hammingDistance = np.sum(misMatch)
91     return hammingDistance
```

In line 83, the NumPy method *multiply*() multiplies the 2D connection matrix, *memory*, and the 1D input array, *targetMem* element-wise in a row-by-row priority as illustrated below. This is equivalent to the following matrix product:

$$A = \begin{bmatrix} a, b, c \\ d, e, f \\ g, h, i \end{bmatrix} \qquad B = [x, y, z]$$

$$np.multiply(A,B) = \begin{bmatrix} ax, by, cz \\ dx, ey, fz \\ gx, hy, iz \end{bmatrix}$$

**Figure 13** Numpy's *multiply*() method for *nadarray*s applies the necessary multiplication operation between two arrays of size N by N and N. This type of matrix vector multiplication is the first step in the *recall*() function.

Remember here that x, y, and z elements of *targetPat* are either 0's or 1's. The input to the connection matrix, **A**, can be thought of as **B**, *targetPat.* By multiplying the values of *targetPat* element-wise through the connection matrix, certain columns in the memory matrix become activated while other are not. This is precisely the idea of the Hopfield network! An input, *targetPat*, is applied to the network defined by

our weights, *memory*, and the addition of all weights within a row sum to the input to the particular node associated with that node. After all, the rows and columns of the connection matrix correspond to the connections between particular nodes in the Hopfield network. In line 83, the element-wise product of *memory* and *targetPat* is called *output*. The rows of *output* are summed in the next line as a 1D array that will be called *output.* It is ok that we refer to this newly created 1D object using the same name as before since it we no longer need the 2D matrix. Calling the output of a variable once processed by the same variable name is a subtlety in programming used often to represent a transformation and allows greater memory allocation by simultaneously disregarding the no longer needed processed 2D matrix (not to be confused with *memory,* which is retained). The next few lines compress the output values, $\boldsymbol{x_s}$, by setting negative values to 0 and positive values to 1 by applying the learning rule below,

$$f_h(x_s) = 0 \text{ if } x_s \leq 0$$
$$f_h(x_s) = 1 \text{ if } x_s > 0$$

**Figure 14** The learning rule applied makes use of the Heaviside function, $\boldsymbol{f_h}$, and effectively compresses the integers of the output into bit values.

After compression, we can compare this final *output* to the memory we wanted to recall, *targetPat,* in line 89 using he built in NumPy method not_equal(). This method returns an array the shared size of *targetPat* and *output* with elements

containing the Boolean values of true and false, or +1 and 0 respectively. The summation of these elements using the method sum() provides an integer value for our hamming distance, *hammingDistance*, which is the number elements that the output did not match with *targetMem*. Lastly, the *recall*() function returns the *hammingDistance* for use in the *main*() function.

In Line 44, the hamming distance is added to a Python list containing the returned hamming distances for the particular pattern size and length for each trial.

```
39              tr1 = time.time()
40              hammingDistance = recall(memory, targetPat)
41              tr2 = time.time()
42              recallTime = tr2 - tr1
43
44              hams.append(hammingDistance)
```

The average hamming distance is computed in line 46 following the prescribed number of trials.

```
46          avgDis = float(sum(hams))/numberOfTrials
47          avgError = avgDis/patlen
48          avgHamDistances[numpat-1][patlen-1] = avgDis
49          avgErrors[numpat-1][patlen-1] = avgError
```

This number, *avgDis*, is used in the next line to determine the normalized error for the length of the particular pattern and is defined as *avgError*. Both the *avgDis* and *avgError* are stored in the 2D arrays *avgHamDistances* and *avgErrors* respectively containing the data for all sets of pattern size and length. The set of operations described above repeat for all sets of pattern size and length before continuing on to save and display the data obtained.

```
51      saveHamDistances(avgHamDistances)
52      showPlot(avgHamDistances)
53      t1 = time.time()
54      timeTaken = t1 - t0
```

The *saveHamDistances*() function is one defined in the Hopfield_1D.py program. This function saves the 2D array of *avgHamDistances* into a comma-separated value, CSV, file. Programs often use CSV files for data storage because of how easy they are to use and for compatibility with many spreadsheet programs. CSV files provide a simple way to store the data in a permanent location for later analysis.

```
93 def saveHamDistances(avgHamDis):
94
95     hamFile = ('hammingDistance_Len-'+ str(maxPatlen) + "_Num-"
96                + str(maxNumpat) + '_Trials-'+ str(numberOfTrials) + '.csv')
97     try:
98         (open(hamFile))
99     except IOError:
100        f = open(hamFile, 'a')
101    else:
102        os.remove(hamFile)
103        f = open(hamFile, 'a')
104    for row in avgHamDis:
105        for col in row:
106            f.write(str(col) +',')
107        f.write('\n')
108    f.close()
```

The file name is defined on line 95-96 and concatenates, or pieces together, the variables used for the particular run of the program. An example of the saved *hamFile* name is “hammingDistance_Len-40_Num-20_Trials-4.csv. These unique filenames provide for later identification of the parameters used to generate the data. In essence, the variable *hamFile* stores the string of the file name to be stored.

The next few lines use the file name in *hamFile* to create the file if one does not already exist and to overwrite the previous file if it does exist. Lines 97-103 use a method called “exceptions handling” to check the existence of a file name. Note

that files will be both stored and identified in the same directory as the program Hopfield_1D.py. Lines 97 and 98 will try to open *hamFile*.csv and will "throw" the exception *IOError* if the file does not exist. In this case, the program will perform the code on line 100 and then continue on at 104. If the file does exist, lines 102 and 103 will remove *hamFile*.csv from the directory and create and open a new *hamFile*.csv.

The now empty and accessed *hamFile*.csv uses row-major order to write each datum followed by a comma. Python is a row-major order language type, meaning, a stored array will be accessed entire rows at a time. Row-major order languages iterates arrays best when the columns vary the fastest. A new line character, "\n", is written at the end of each row in *hamFile*.csv. Both the "," and "\n" characters provide key file input/output information for any spreadsheet program reading the file. Once all datum have been written, the file closes and we continue back in the *main*() function.

The next operation is to display the data using the *showPlot*() function defined in the program.

```
110 def showPlot(data, title):
111     import matplotlib.pyplot as plt
112     from mpl_toolkits.mplot3d import Axes3D # this is actually needed
113     fig = plt.figure()
114     ax = fig.gca(projection='3d')
115     X = np.arange(1, maxPatlen + 1, 1)
116     Y = np.arange(1, maxNumpat + 1, 1)
117     X, Y = np.meshgrid(X, Y)
118     Z = data
```

The first operation is to import the Matplotlib class *pyplot* as *plt*. Matplotlib is an excellent python-specific and open source alternative to the comparative commercial plotting utilities offered by Matlab. We will define *fig* as the plot of the

*data* we wish to show and *ax* as the axes of the plot. In this case, the axes will be 3-dimensional in representation and we define these axes to *ax* in order to modify the axes titles later on. The *X-values* of the plot are set in line 115 to range from 1 to (*maxPatlen* +1) and a similar operation is performed to set the *Y-values* in 116. The *X-* and *Y-values* are processed using the NumPy function *meshgrid*() to set up the 2D coordinate matrices needed to plot the *data* set to the *Z-values*.

The next operation is to set the plot type applying the *plot_surface*() function to *ax* on line 120. This function takes the following parameters: *X-*, *Y-*, and *Z-values*; *rstride*, the x-axis step; *cstride*, the y-axis step; *alpha*, the opacity of the surface; *cmap*, the color scheme for the plot.

```
120     ax.plot_surface(X, Y, Z, rstride=1, cstride=1, alpha =0.3, cmap="hot")
121
122     ax.set_xlabel('Pattern Length')
123     ax.set_xlim(0, maxPatlen)
124     ax.set_ylabel('Number of Patterns')
125     ax.set_ylim(0, maxNumpat + 2)
126     ax.set_zlabel(title)
127     ax.set_zlim(0, np.amax(Z))
128
129     plt.show()
```

In line 122-123 we set the axis titles and the min- and max-values represented for each axis of the plot. Finally, we show the plot in a new pop-up window using *plt.show*() in line 129. A particular NumPy function we have not yet seen, *amax*(), finds and returns the maximum value in an array of any size. Setting the *z-limit* to the maximum average hamming distance is an easy way to automatically size the plot to fit the *data*.

The last lines of code in the *main*() function provide run-time information regarding parameters chosen.

```
56    print "_____________________________________"
57    print "Max # patterns:          ", maxNumpat
58    print "Max length of pattern:   ", maxPatlen
59    print "Number of Trials:        ", numberOfTrials
60    print "Time Taken:              ", timeTaken
61    print "Memorize time            ", memorizeTime
62    print "Recall time              ", recallTime
63    print "end of run"
```

We use the Python command print followed by the strings and value we wish to print separated by a comma. The above execution prints to the bash-shell as shown in the example below:

```
Max # patterns:          20
Max length of pattern:   40
Time Taken:              49.8099310398
Memorize time            0.356177091599
Recall time              0.00434303283691
end of run
>>>
```

**Figure 15** Hopfield_1D.py prints information regarding the parameters and performance times for the most task intensive set of pattern length and number.

These final print statements conclude the complete run of Hopfield_1D.py.

A great reason to use Python and the associated libraries of NumPy and Matplotlib are the extensive documentation and growing use of Python. The Python community consists of users with a range of expertise making it easy to find solutions from the most basic to very detailed problems. Documentation and access to forums can be found at ***www.python.org*** and ***www.scipy.org***.

## Results

The normalized Hamming distance for 10 patterns stored in a Hopfield network was found to fall off rapidly as the length of patterns learned was increased (Figure 16). This agrees with the trends proposed by Hopfield, who showed significant errors begin to occur when the number of patterns learned exceeds 10% of the pattern length, where the lengths is the number of binary state values comprising the pattern (Figure 16)(Anderson 1995).

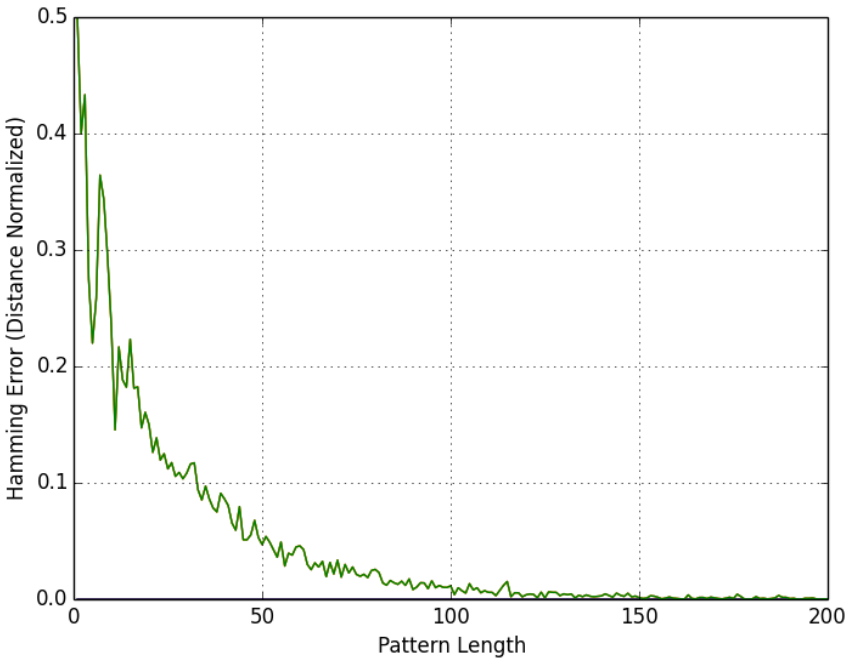


**Figure 16** Normalized Hamming distance in response to pattern lengths (number of binary state values) for 10 patterns stored.

Recall that the energy minima in state space for a Hopfield network will tend to drift when the number of patterns stored exceeds $0.10N$ patterns stored for $N$ neurons in the network or for pattern lengths of $N$ binary state values. The same storage capacity of the network is also seen when varying the number of patterns stored while holding the length of patterns constant (Figures 17).

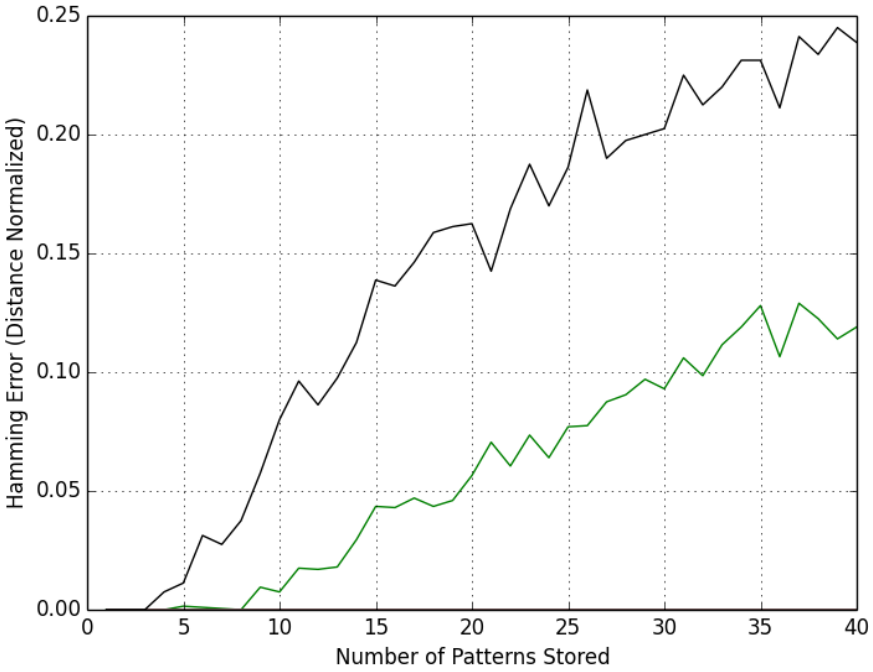


**Figure 17** Normalized Hamming distance in response to the number of patterns stored. Top line indicates error in response to patterns with a length of 40 binary state values. Bottom line, (green), indicates response to patterns with a length of 100 binary states.

Figure 17 also shows the improvement in recall of patterns, i.e. lower Hamming distances, for increased pattern length. The normalized Hamming distances measured were significantly reduced when increasing the pattern length from 40 to 100 binary state values.

The normalized hamming distance was found for many different networks with pattern lengths of up to 100 binary state values and up to 20 patterns stored. The

normalized hamming distance was reduced for networks containing higher pattern lengths and lower numbers of patterns stored (Figure 18).

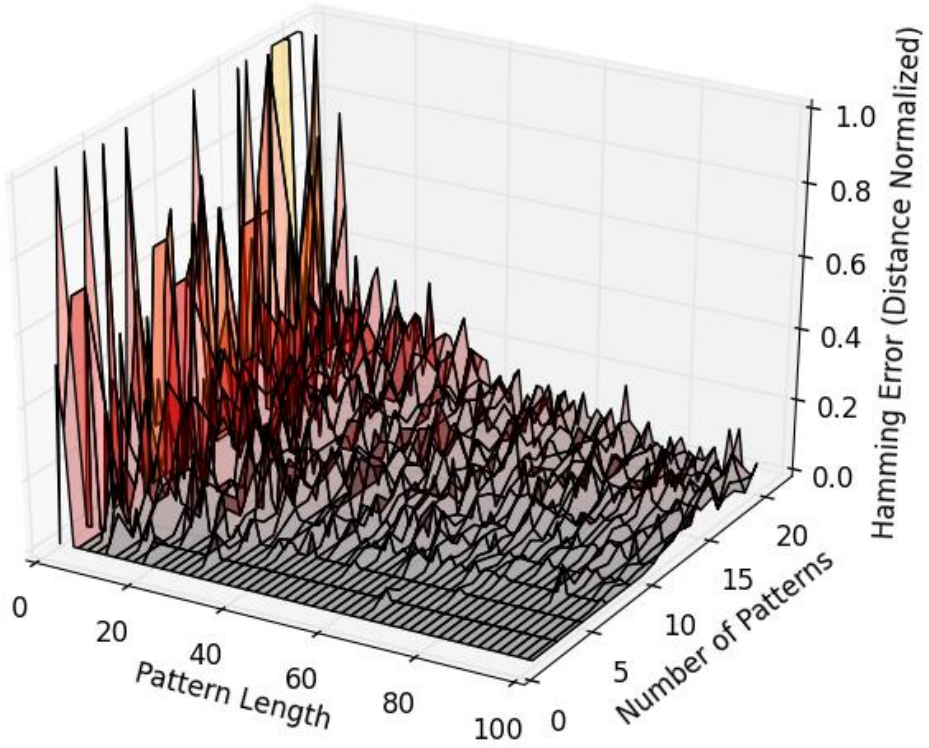


**Figure 18** Surface-plot for the normalized hamming distance for 2000 different networks with pattern lengths of up to 100 binary state values and up to 20 patterns

The normalized Hamming distance was measured for 2-dimensional patterns of size $N^2$. The measurement of normalized Hamming distances was once again measured for many different networks with pattern sizes of up to $40^2$ binary state values and up to 20 patterns stored (Figure 19).

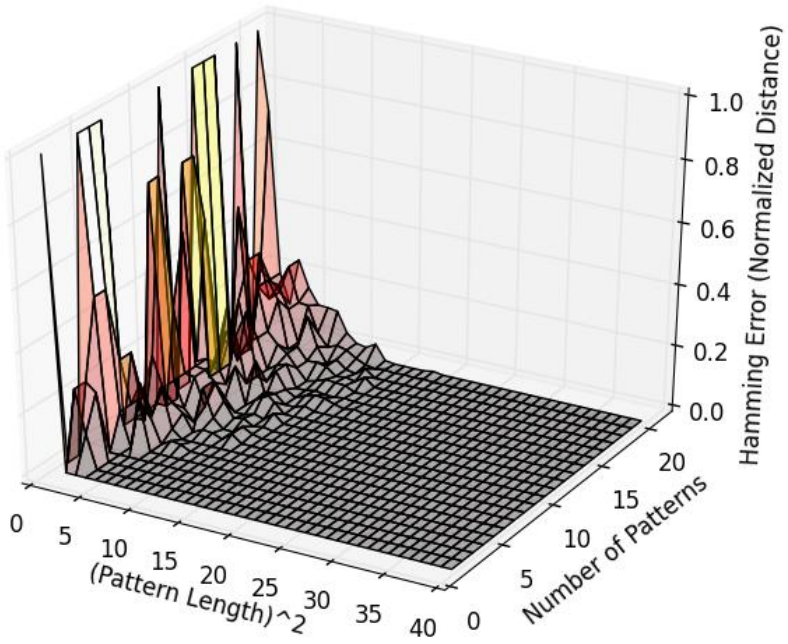


**Figure 19** Surface-plot for the normalized hamming distance for 2000 different networks with pattern sizes up to $40^2$ binary state values and up to 20 patterns stored

The normalized Hamming distance for the storage of 2-dimensional patterns falls off substantially large increase in the number of neurons within the network, which is equivalent to the square of the pattern length $N^2$.

## Conclusion

All of the results support the assertion that the normalized Hamming distance, or error in pattern recall, depends jointly on the number of patterns stored and the pattern length. Taking into account the capacity of a Hopfield network to limit the amount of patterns stored within the network can minimize error in recall. The programs used for this study, Hopfield_1D.py and Hopfield_2D.py, only updated the Hopfield network once. Unlike the full evolution of an input pattern to one previously stored, the single update performed in this study is more susceptible to

excessive “noise” in the network. Excessive “noise” in the network due to greater number of patterns stored causes the drift of energy minima in the state space of the network. The energy minima correspond to patterns stored in the network. The Hamming distance measured between the input and output to the Hopfield network are due in-part to the excessive “noise” of the network. The results of this work also supported the typical range for excessive error in recall beginning when number of patterns stored is 10-15% of the number of state values of the patterns stored, i.e. the number of neurons in the Hopfield network.

## Future Work

After simulating the 1-dimensional case of the Hopfield, it is the interest of this study to simulate the 2-dimensional case. The 2-dimensional case takes input in the form of a 2-dimensional matrix instead of a 1-dimensional array. The Hopfield model for the 2-dimensional case is used primarily in image processing where it is studied extensively for its applicability in medical imaging, lexical identification, and the study of the visual cortex (Peterson 2002, Davis 2010, Chialvo 2010). Hopefully interest in neural networks and particularly the Hopfield model will continue to grow.

## Bibliography

Anderson, James A., and Joel Davis. An introduction to neural networks. Vol. 1. Cambridge, MA: MIT press, 1995.

Beale, Russell, and Tom Jackson. Neural Computing-an introduction. CRC Press, 1990.

Beaudin, Laura. "A Review of the Potts Model." 2007.

Bressert, Eli. "Introduction." Introduction. SciPy and NumPy:. Beijing: O'Reilly, 2012. 1-4. Print.

Chialvo, Dante R. "Emergent complex neural dynamics." Nature Physics 6.10 (2010): 744-750.

Davis, Colin J. "The spatial coding model of visual word identification." Psychological Review 117.3 (2010): 713.

de Oliveira, Fagner A., Cristiane N. Nobre, and Luis E. Zárate. "Applying Artificial Neural Networks to Prediction of Stock Price and Improvement of the Directional Prediction Index-Case Study of PETR4, Petrobras, Brazil." Expert Systems with Applications (2013).

Fox, Stuart Ira. Human Physiology. New York: McGraw-Hill, 2011. 232. Print.

Hinton, Geoffrey. "Hopfield Nets." Coursera. Web. 10 Mar. 2014. <class.coursera.org/neuralnet-2012-001/>.

Hopfield, John J. "Neural networks and physical systems with emergent collective computational abilities." Proceedings of the national academy of sciences 79.8 (1982): 2554-2558.

Krugers, Harm J., et al. "Regulation of excitatory synapses and fearful memories by stress hormones." Frontiers in behavioral neuroscience 5 (2011).

Mcculloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." The Bulletin of Mathematical Biophysics 5.4 (1943): 115-33. Print.

Mitchell, Melanie. "Dynamics, Chaos, and Prediction." Complexity: A Guided Tour. Oxford: Oxford UP, 2011. 21+. Print.

Petersen, Michael, Dick de Ridder, and Heinz Handels. "Image processing with neural networks—a review." Pattern recognition 35.10 (2002): 2279-2301.

Sergiu, Clumac. "Financial Predictor via Neural Network." - CodeProject. N.p., n.d. Web. 10 Apr. 2014. <http://www.codeproject.com/Articles/175777/Financial- predictor-via-neural-network>.