Title

An Exploration of Varying Conditions in a Hopfield Neural Network and Applications to a DNA Implementation

Permalink

https://escholarship.org/uc/item/1ss7c7k1

Author

Hughes, Bradley Steven

Publication Date

2010

UNIVERSITY OF CALIFORNIA
RIVERSIDE

An Exploration of Varying Conditions in a Hopfield Neural Network and Applications to a DNA Implementation

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Physics

by

Bradley Steven Hughes

August 2010

Dissertation Committee:
Dr. Allen P. Mills, Chairperson
Dr. Roya Zandi
Dr. Ward Beyermann

The Dissertation of Bradley Steven Hughes is approved:

____

____

____
Committee Chairperson

University of California, Riverside

Acknowledgments

This work was funded by in part by the U.S. National Science Foundation under grant no. CCF-0524203.

Dedication

Dedicated to my mom and dad, Ruth E. Stahl and Clayton N. Hughes, for giving me:

The wisdom: “Stay in school and learn all you can.”

The direction: “Aim for the stars, but keep your feet on the ground.”

and

The ability: “You can do anything in the world you want to do.”

All that I am and all that I will be began with you.

ABSTRACT OF THE DISSERTATION

An Exploration of Varying Conditions in a Hopfield Neural Network and Applications to a DNA Implementation

by

Bradley Steven Hughes

Doctor of Philosophy, Graduate Program in Physics
University of California, Riverside, August 2010
Dr. Allen P. Mills Jr., Chairperson

A Hopfield Neural Network is a content addressable memory with elements consisting of the correlations between elements of memory vectors. Recall of a complete memory vector is possible via the introduction of a "corrupted" vector, which is a memory vector with some components altered. It may also be possible to correctly recall memories with the use of a partial vector. It may be possible to create such an information storage and retrieval system using DNA as a working substance. Herein I present some computational results for properties of Hopfield Neural Networks, as well as a theoretical framework for the operation of such a system, including possible limitations in the working substance.

Table of Contents Page Number

List of Figures

Chapter 1 – The Basis of Computation

What is Computation?

Since the days of old, computation has been performed, both by sentient organisms (humans) as well as, possibly, by natural systems. From the Ishango Bone of Zaire (used to count) to the abacus, these systems gave us the requisite components for performing a computation. Naturally, we have advanced significantly since those primitive devices. Today, research in computation is focused on the pursuit of both practical considerations (making computational devices faster and smaller) as well as more theoretical concerns (what are the limits of computation, how does artificial intelligence compare to that found in creatures such as ourselves).

Even before the advent of the first electro-mechanical computer in 1941, researchers began to explore more fundamental questions related to computation. These questions included such topics as “What is, strictly speaking, a computation?”, “What different ways are there to represent a computation?” etc.

In this dissertation, I will address issues dealing with the limitation, optimization and implementation of a Hopfield Neural Network (HNN) architecture. However, to understand the neural network at its basic level, it should first be recognized that a NN is a subset of computation. That is, while it may appear to be some sort of mystical phenomenon to the uninitiated, it is certainly within the domain of physics and computer science to explore questions related to these devices. To that end, I begin with a brief discussion of the basis of computation.

1a. Logic Gates

The implementation of computation, then, requires the ability to manipulate inputs and produce outputs in such a way as to demonstrate meaningful mathematical results. To put it another way, if a computer is to produce output which is "correct" then its physical process should give out an output which corresponds to mathematical equations. For example, if I made a computer from a six sided die tumbling inside a drum (similar to a dryer), and set the input to be whatever two numbers came up in order, and then the output as the third number to be face up when the dryer stopped, this would produce arithmetic nonsense. The first number could be a "1", the second a "3", and the third a "4". Someone might unknowingly conclude initially that this is an "adder"; that is to say that it takes the first two numbers and produces an output which is the arithmetic sum of those two inputs. But once the passerby repeated the process, they would find that there is a completely different set of inputs and outputs, without an arithmetic connection. Eventually (hopefully) the researcher would conclude that there is indeed no relationship between the first two rolls of the die and the third. Therefore, this physical system does not perform a meaningful computation.

However, if I could physically arrange a system so that it produced a meaningful output each time I put in a set of inputs, then that would be a form of computation. With computers today, it is possible to enter a set of inputs and receive a dazzling variety of output from pictures, sounds, text, equations, etc. It is truly incredible that such a stunning array of output possibilities can be derived from very simple initial computational elements. Building up computation from these simple "primitives" is the objective of the next section.

1b. Building Computers

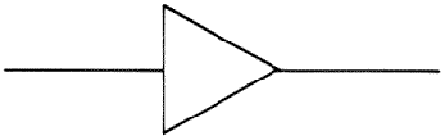
To make sense of the architecture of computers, it is imperative to first understand the models that will be used to observe the behavior of these components. The first of these is the "truth table". This table is an exhaustive list of the possible inputs and outputs for a computing element. For example, if my element was an "identity", it would take whatever was input to it, and produce the same output, in effect "passing on" whatever it was given. If, however, the element was a "NOT" gate, it would take whatever input was given to it and give the opposite value in binary. In this case, an input of "0" would give an output of "1" and vice versa. The truth tables for these two gates, along with their symbols, can be seen in Figure 1.1 below:

Identity Gate

Input	Output
0	0
1	1

Not Gate

Input	Output
0	1
1	0



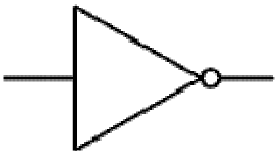


Fig. 1.1 – Identity and NOT Truth Table and Gate

The next logical question is then to ask what "elementary" elements are there in computation? If I add one more gate – the AND gate – shown below (Fig. 1.2):

AND Gate

Input A	Input B	Output
0	0	0
1	0	0
0	1	0
1	1	1

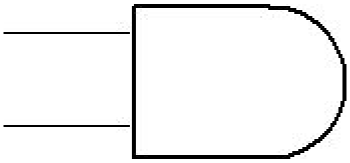


Fig. 1.2 – AND Truth Table and Gate

Using these two symbols, more complex gates can be obtained. For example, by connecting AND and NOT gates in the following manner:

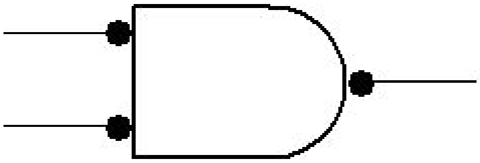


Fig. 1.3 – Construction of OR Gate

I obtain a gate which produces an output of “1” if either A or B is “1”, or if both A and B are set to “1”. This new type of gate can be called an OR gate, meaning that it produces an output of “1” if either of the inputs (or both) are set to “1”. A truth table and new symbol representing this gate are shown below in Fig. 1.4.

Input A	Input B	Output
0	0	0
1	0	1
0	1	1
1	1	1

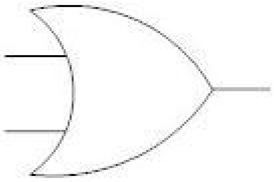


Fig. 1.4 – OR Truth Table and Gate

In terms of building gates from other gates, one would frequently use a set of operators such as AND, NOT, and OR. While it is possible to construct any logical system from some combination of these three, it is interesting to note that some particular binary gates are in and of themselves complete. Both the NAND and NOR gates were proven by Sheffer[1] to be what is referred to as “functionally complete”. This means any other set of logic gates can be constructed from just one of these two gates, used repeatedly and arranged in the appropriate sequence. One example of gate construction using AND, NOT and OR gates is the XOR gate, shown below:

Input A	Input B	Output
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 1.5 – XOR Truth Table

This gate can be constructed in the following manner:

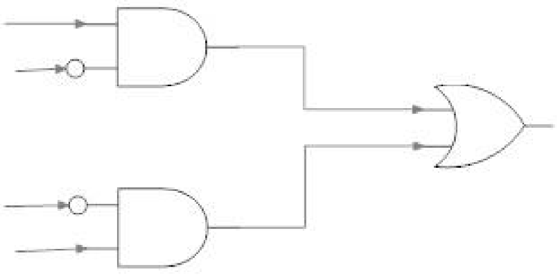


Fig. 1.6 – XOR Construction Diagram

I chose the XOR gate for two purposes: first, to demonstrate an easy example of building gates out of the basic set of AND, OR and NOT; and second to point out the difference between the OR gate, and the XOR gate. The XOR gate is identical to the OR gate, with the exception of the case where both inputs, A and B, are both set to “1”. In that case, the OR gate will have a value of “1”, but the XOR gate will have a value of “0”. The OR gate will return a value of “1” when either A or B is set to “1”, or when both are set to “1”. However, the XOR gate returns a value of “1” if either A or B is on, but “0” if they are both on – consequently this gate is referred to as an “Exclusive OR” gate.

To begin building real computers, which take a given input and produce mathematically correct output, it is convenient to start with a mathematical function which is simple, such as addition. In binary, to add two numbers one needs to sum the farthest column to the right, and see what its value is modulo two. Also, if the values of the inputs are both “1”, the sum modulo two would still be zero, but the “adder” would need to carry over a “1” into the next column. For example, to add the numbers below:

01 +10
11

Fig. 1.7 – Addition Modulo Two

The rightmost column is the addition of 1 + 0 = 1. If the values had both been “0” in this column, the solution would have been 0 + 0 = 0. Likewise, if both inputs had been “1”, the output modulo two would have been 1 + 1 = 0. But this is identical to the truth table for XOR, which was just shown above.

The output of the XOR gate is then the value given for the rightmost column of the output. Naturally, a way is needed to “carry” over a “1” when both of the inputs are “1”, since the XOR does not give that information. Notice that when both inputs are equal to one, the out of the “Carry”, which is an AND gate, will be one. Otherwise it will be zero. This is what is required if something is to be “carried over” in an addition from one column to the next. This configuration of gates is known as a “half adder”.

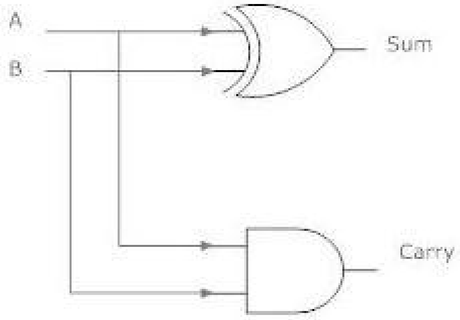


Fig. 1.8 – Half Adder

In order to generalize the set of gates so that they are capable of adding an arbitrary set of numbers, it is necessary to use two of these “half adders” connected with an “OR” gate, as shown below:

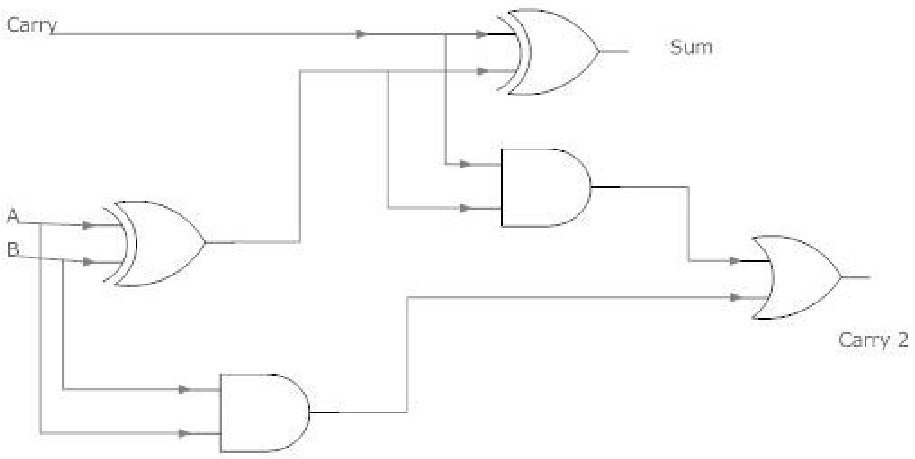


Fig. 1.9 – Full Adder

The truth table for this begins to become cumbersome, but it is given below:

A	B	Carry	S	Carry 2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1

1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 1.10 – Full Adder Truth Table

The reader should note that if this is an arbitrary column in a set of two binary numbers, where "Carry" is the possible value of a carryover from the previous column (which was already computed) and "A" and "B" are the values in that column being added, the output "S" and "Carry 2" are the correct values one should expect to obtain from this process. In this manner, the output of "Carry 2" can be fed into the next adder, to generate the next column of numbers.

Once this is completed, I have shown how to perform one arithmetic operation – that is, addition. To perform other arithmetic operations, such as subtraction, multiplication and division, different gate configurations are required. The construction of each of these gate configurations is shown below.

Subtraction

When subtracting binary numbers, the outputs will now involve the value (S) as well as the amount "borrowed" from the next column to the left. Previously I had a "carry" bit, which was added onto the next column to the left. The truth table, then, for this operation (called a "half subtractor") is given below:

A	B	Difference	Borrowed
0	0	0	0
1	0	1	0
0	1	1	1
1	1	0	0

Fig. 1.11 – Subtraction Truth Table

Interestingly, the Difference column is the output for an XOR gate, just like for addition. However, in contrast to the AND gate used in the full adder, the BORROWED column is the same gate (AND), except the values input are NOT A and B. Similar to the structure of an adder, a half subtractor is shown below:

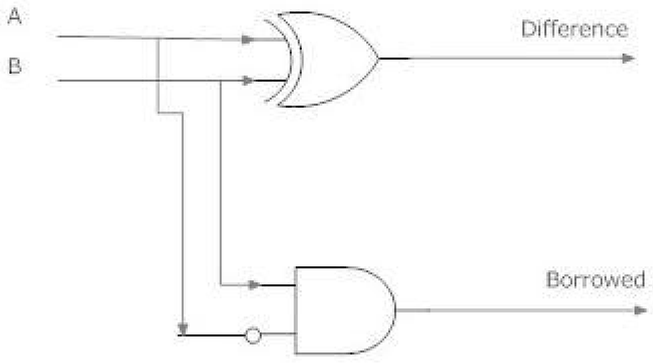


Fig. 1.12 – Subtraction Logic Gates

Multiplication

Multiplication becomes possible using a device which is, in effect, a combination of binary adders. In order to perform multiplication, first imagine there is an operation in the following form: $A_0A_1A_2\ldots A_N$ x $B_0B_1B_2\ldots B_N$, where $\{A_N\}$ and $\{B_N\}$ are the binary digits in an "N" digit number. To perform the operation of multiplication:

1) Operate with an AND gate, pairwise, with each of the bits $A_0..A_N$ on B_0. This produces the first row shown under the input numbers in the figure below.
2) Operate with AND pairwise on all bits $A_0..A_N$ on B_1. This will give a set of inputs, which is the second row. Make sure to have this output "shifted" as shown in the diagram below.
3) Repeat this procedure, shifting one entry with each b_n.

4) These products of the AND gate are then fed into a binary adder. For example, $p_0= a_0b_0$, while $p_1=a_1b_0 + a_0b_1$. A figure representing this procedure is shown here:

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Fig 1.13 – Binary Multiplication of Two Digits

With the ability to perform arithmetic operations using a set of logic gates comes the ability to solve an enormous set of mathematical problems. A question of intense interest in computer science is the issue of which problems can be solved by a computational system and which problems can't. This is codified in the notion of an effective procedure, which as Feynman[2] states is "a set of rules telling you, moment by moment, what to do to achieve a particular end; it is an algorithm". Some problems lend themselves to algorithmic approaches. One example of this is differentiation. Given a function to differentiate, a particular set of rules for completing the problem will generate the correct derivative. Other types of problems, however, are more elusive and may appear to lack an effective procedure – such as integration. However, even elementary integration can be reduced to an effective procedure, as was demonstrated by Risch[3]. While many problems have now been reduced to effective procedures, it will be demonstrated shortly that there are some problems which no computational system can solve. To elaborate on this, it is first necessary to explore different forms of computational systems.

1c. Finite State Machines

With a framework in mind on how the "nuts and bolts" of this computer is to be established, it becomes incumbent to describe what some of the real possibilities and problems are with this framework. To accomplish this, I will first demonstrate the existence of what is referred to as a "finite state machine", followed by a generalization to the much more powerful "Universal Turing Machine".

My discussion of a finite state machine closely follows Minsky[4]. I can think of a finite state machine (FSM) as being a device which takes inputs and produces outputs in a finite number of possible ways. The number of inputs and outputs need not be equal, and the output produced will be dependent on both the stimulus (S) and the state (G) of the machine at the time S is received. Therefore, all of the mathematical operations discussed so far, once implemented, would be an FSM; however they would have only one state but multiple inputs and/or outputs. An FSM would be a machine which could change its state as well as produce an output when an input is received.

A discussion of the properties of this machine is in order. In reality, time occurs along a continuum. However, for my purposes, it will suffice to take the operation of an FSM as occurring in discrete temporal units. Also, the particular working medium (along with the inputs and outputs) are irrelevant. The inputs and outputs can be water in a pipe, or voltage on a wire, or light signals in a fiber optic line. In this discussion, however, I will restrict the inputs to binary values; 0 or 1. It is possible to effect any particular type of computation in this basis without a loss of generality, and with only a minimal slowdown under certain circumstances. Likewise, the working medium for the computation is irrelevant. I can imagine the computational medium as being a valve which reacts in a certain way for the water input, or a system of mirrors and lenses producing a desired output with light, or any other electrical/mechanical system of which I can conceive. With this definition, an FSM could be any number of things which takes inputs and produces outputs. For example, a plant could be thought of as a machine which takes in sunlight, carbon dioxide, and water, and produces oxygen and a different state (a bigger one, presumably) for the plant. The key, then, is to harness the power of physical systems to produce meaningful computational output.

This FSM can be thought of as a device with a limited number of states {Q}. When fed S (a stimulus), the machine will generate a response R. This R will be a function of both S and Q. However, S does not only produce R. It can also act to change Q to a new state, Q'. This relationship between Q(t) and Q(t+1) is a function of both the stimulus and the current state of the machine. These two rules can be codified as shown below:

$$R(t+1) = F(Q(t), S(t))$$

$$Q(t+1) = G(Q(t), S(t))$$

This means that both the response and the state of the machine at the next instant are functions of the current state of the machine and the stimulus received.

Since both functions, F and G, are discrete, I can exhaustively list their possible values. As a trivial example, consider a device that can have a binary stimulus (S=0,1). Upon receipt of S, the machine is in a single state (Q_0 or Q_1). In either of these two states, the machine's R is the opposite value from what is input. So, for example, if someone were to input S=0, the machine would output R=1, and vice versa. This is codified in the first table below, where the S_0 and S_1 represent the possible stimulus input, and the response is listed under the appropriate column for which state the machine is in when the stimulus is received. Also, if the machine receives a stimulus with a value equal to its state number, it switches to the opposite state – otherwise it stays the same. This is codified in the second table below, again where S_0 and S_1 are possible stimuli, and the state of the machine at time=t+1 is given, depending upon which of the two possible states (Q_0 or Q_1) the machine was in at time=t. A set of two tables for this machine can then be used to list out both the state function and the response function for a given machine. These tables are shown below:

R	Q_0	Q_1
S_0	1	1
S_1	0	0

G	Q_0	Q_1
S_0	Q_1	Q_1
S_1	Q_0	Q_0

Fig. 1.14 – State Transition Table

The reader might notice immediately that the first table is the truth table for a NOT gate, since when a “0” is the stimulus, a “1” is the output in all cases, and vice versa. Note that in this case, the state of the machine (Q) is irrelevant, since the output is the same regardless of which state the machine is in when the stimulus is received. In this way, then, this simple finite state machine is exactly a NOT gate. While this seems trivial, it is important to note that more complex designs can be built up depending upon the functions involved. Once the number of states and responses of these FSM’s become large, it will be easier to represent the machine by a “state diagram”, showing an exhaustive pattern of what occurs when the machine is in a certain state and fed a certain input. The state diagram for the toy machine I just created is shown below:

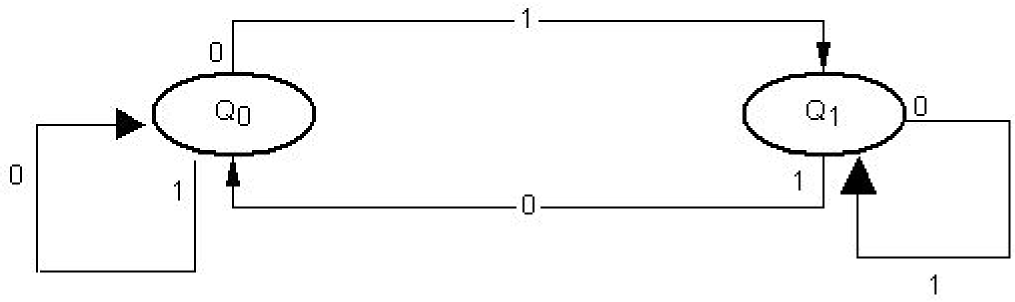


Fig. 1.15 – State Diagram for a NOT FSM

As can be seen, if the machine is in a state (say Q_0), and receives an input of “0”, it will then transition to state Q_1, and produce an output of “1”. Now I have three possible representations for these machines: A series of logic gates, a truth table, and a state diagram. Which one is chosen to use in any particular case will depend on what is being demonstrated. For the moment, I will use state diagrams to illustrate FSM’s.

Some other examples of FSMs are demonstrated here, again closely following Minsky[4].

The Memory Machine

The Memory Machine is a device which has the ability to "remember", by outputting at time t+1, what the input was at time t. In order to accomplish this, it is necessary to have a two state machine. The function tables and state diagrams for this device are shown below:

G	Q_0	Q_1
S_0	Q_0	Q_0
S_1	Q_1	Q_1

F	Q_0	Q_1
S_0	0	1
S_1	0	1

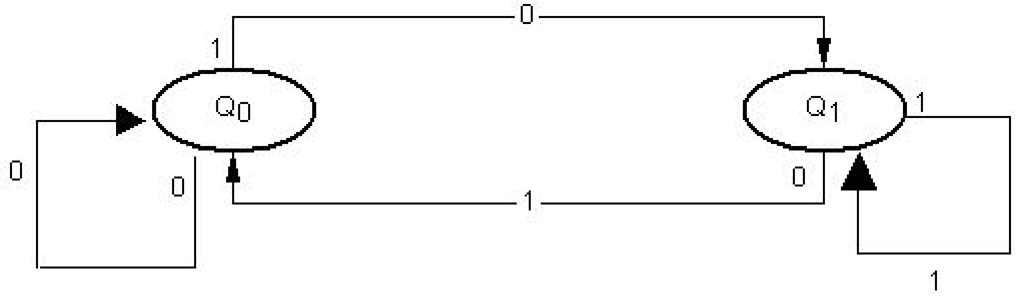


Fig. 1.16 – Truth Table and State Diagram for a Memory Machine

Note that no matter what the input to this machine is (0,1), the output is the same as the input, allowing it to function as a memory of the time the input was entered.

The Parity Machine

Another interesting machine is one which outputs the parity of the number of “1”s it has received. The key feature of such a machine is that the state and output remain the same when a “0” is entered, but change when a “1” is entered. As a consequence, an even number of “1”s will cause the state not to change, but an odd number of “1”s will. The function tables and state diagrams for this device are shown here:

G	Q_0	Q_1
S_0	Q_0	Q_1
S_1	Q_1	Q_0

F	Q_0	Q_1
S_0	0	1
S_1	1	0

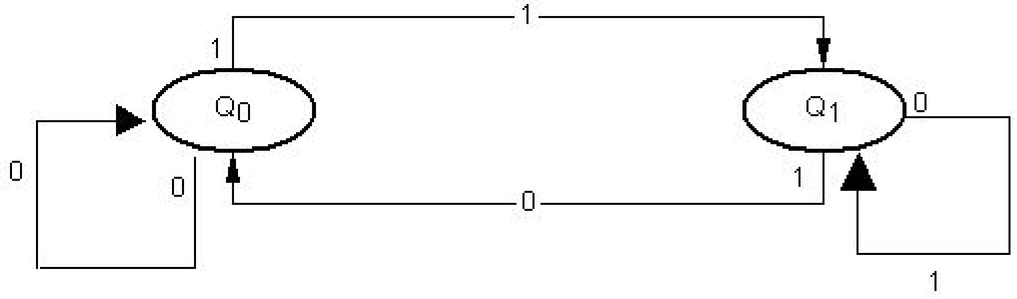


Fig. 1.17 – Truth Table and State Diagram for a Parity Machine

The Binary Adder

Finally, as a last example, I show the Binary Adder. This machine takes two binary numbers of an arbitrary length and adds them, producing their sum as an output. In reality, since the digits are fed in two at a time, only two states are needed. One of these is for when there isn't a "carry" to be put into the next column of the addition and the other is for when there is such a carry. The state diagram is again shown below:

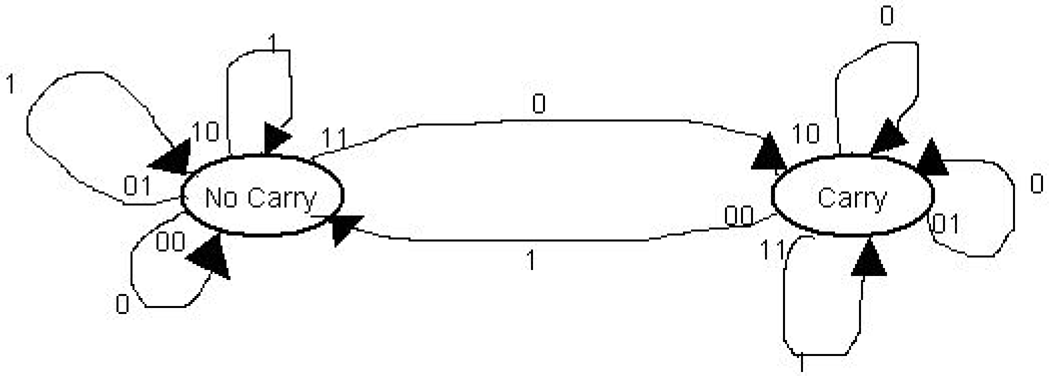


Fig. 1.18 – State Diagram for a Binary Adder

As can be seen from this last machine, devices which can be represented with circuit diagrams and logic tables can also be formulated in terms of state diagrams. And hence this discussion has come full circle, from building machines out of individual components to taking the whole machine as a functional unit, which is represented in its computational path as a state diagram.

However, as I will now show, there are some problems which an FSM can never hope to solve. In order to address these problems, I will need to invoke a more advanced machine – the Universal Turing Machine.

1d. Turing Machines

In 1936 Alan Turing encapsulated the above ideas into a "Turing Machine" (TM)[5]. A TM is a machine which takes an input stimulus (S) and produces an output (R). This is just like the FSM shown above. However, a TM also has the benefit of being able to write an answer out and access it again later. Turing envisioned these machines as being like an FSM, with the benefit of having a theoretically unlimited supply of tape upon which to read and write. As is shown, this unlimited capacity allows a TM to perform computations which are beyond the reach of an FSM.

I demonstrated in the last section that some problems, such as addition and subtraction, could be solved using an FSM. Consider what happens when I try to extend this idea to the multiplication of two binary numbers of arbitrary length. (Note that this is not the same as multiplication of two numbers of a known length.)

Assume for the sake of discussion that there existed some machine which could multiply arbitrarily long strings of binary numbers. Like the situation with the adder (shown previously), numbers are input one digit at a time. Recall that any binary number of the form 2^n is a one, followed by all zeros. For example, $2^2 = 4 = 100$, $2^3 = 8 = 1000$, etc. If I ask the machine to multiply a number of the form 2^n by itself, I am talking about two numbers of the form 1000…. . This is a “1”, followed by n zeros. This will give a result containing 2n + 1 digits. Since the inputs have only n+1 digits, the machine will have to print “n” more zeros after the input has been entered, and then print the “1”. However, once the input has stopped, the machine must cycle through its states with a constant input of zero. The only way, then, to make the machine print a “1” is to let it have more states than there are zeros to be printed remaining (that is, $n<Q$). However, since it is possible to create an arbitrarily large multiplication, there is never a guarantee that the machine will have enough states to cycle through to output the “1”. This leads to the theorem:

“No fixed-state machine can multiply arbitrarily large pairs of binary (or decimal) numbers.”[4]

This then proves that there are some problems which are not solvable within the framework given thus far. However, I have dealt with only FSM’s. In contrast to the FSM, the ability to read and write from an external memory gives the Turing machine an advantage. The quintessential prototype of a TM is a device with a read-write head. This device is capable of reading from a tape, and using this input (S) in conjunction with its state (Q), produces an output (R), which is then written in place of the material in the original input. From there the TM is capable of shifting one cell to the left or right, where it starts the process over again, this time using the contents of the new cell as S. While this may appear to be incredibly simple, it is deceptively so, for all modern day computers are simply bigger versions of this idea, with memory cells and drives replacing paper tape.

The transition functions for a TM involve not only the response function and state functions shown for an FSM, but also a function to determine which way the head needs to move after completing an operation. These functions are:

$Q(t+1) = G(Q(t),S(t))$

$R(t+1) =F(Q(t),S(t))$

D(t+1)=D(Q(t),S(t))

Types for Problems for a TM

As demonstrated above, there are some problems which cannot be solved by an FSM. These problems are in a class which would require a potentially unbounded amount of memory. That being the case, a TM is a likely candidate for these problems, since it has an unlimited amount of tape on which to read and write. Here I show an example of problems which can be solved by a TM, but not by an FSM.

Check the Number of Parentheses

A "parenthesis checker" is a machine that checks whether or not a string of parentheses has the correct number of left and right parentheses and in the correct order to be balanced. For example, the string of parentheses consisting of:

(((( ))))

is a valid string, since each right facing parenthesis lines up in an appropriate sequence with a left facing counterpart. However, the string,

( ))

is not valid, since both the number and alignment of parentheses is incorrect.

It sounds like a trivial matter to imagine a machine which could analyze such a string. I can imagine that a particular machine could be designed which would start at the left side of the string and keep track of the number of parentheses that were "open" in nature. When it got to a "close" parenthesis, it would cancel one of the open parenthesis and keep moving to the right. If there are leftover parentheses at the end of the analysis, then the string is not well formed. Otherwise, it is!

The difficulty with this problem for an FSM, however, comes in the number of parentheses. Since the number of parentheses can be any number, there is no guarantee that the FSM being used to analyze the string will have enough states in it to keep track of the number of open parentheses. Of course, I could design an FSM which could keep track of some finite number of open parentheses, but not one which could solve an arbitrary long problem of this sort.

Solving such an arbitrary length problem is certainly within the abilities of a TM. The state diagram for a TM that performs as a parenthesis checker is[4]:

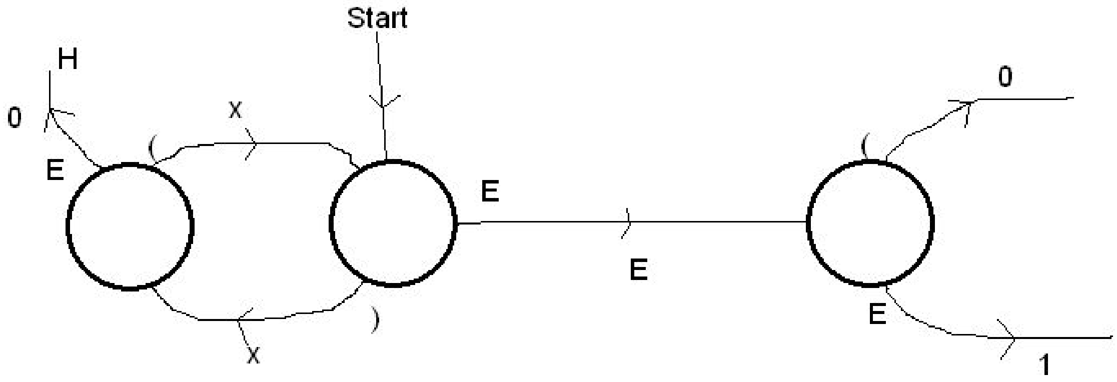


Fig. 1.19 – State Diagram for a Parenthesis Checker

This TM can cycle through states, counting parentheses until it gets to the end of an arbitrarily long string.

It may at this point be thought that the most powerful example of a classical computer has been found through this simple prototype. However, there is a specific type of TM which is more powerful than any other. This is a Universal Turing Machine (UTM). The UTM is a machine which can emulate the operation of any other Turing Machine. The UTM accomplishes this by having the program for the machine which it is emulating written on the tape. This tape is then accessed to tell the UTM how to behave under any specific circumstances, thereby giving it the ability to emulate other TMs. A very interesting consequence of this will be addressed when I consider whether a UTM can emulate itself, leading to what is referred to as the Halting Problem. This will be dealt with in short order.

Chapter 2 – More Advanced Issues in Computation

2a. Proof of the Halting Problem

In dealing with issues in computability, few issues have received more attention than the Halting Problem. Alan Turing was able to formulate the Halting Problem along with Turing Machines in his 1936 paper[5]. In its basic formulation, the halting problem states that it is impossible, in principle, to determine whether or not a machine will halt for a given computation (that is, for a given Turing Machine with an input).

To prove this statement, first assume that there exists some machine (M), which will tell us whether or not a specific other machine (O) will halt, once it has been given O's description and program to analyze (O_T). Since M is capable of determining if O stops, where O is an arbitrary machine, then it should be capable of performing the same operation for machine (O_T, O_T), where now instead of using O as the machine description and O_T as the program, O_T is serving as both the description of the machine AND its operating system. Now add another machine to the set (N), which requires the description O_T for its operation, but otherwise performs like M. N should be able to do the same things M can, but also be capable of copying a block of symbols. Machine N should have two ending states, one printing a "Yes" if (O_T, O_T) stops, and one printing "No" if (O_T, O_T) never stops. Now, by making a small change, create a machine (N'), which is identical to N, with the modification that the new machine doesn't halt if it takes the "Yes" path toward termination. Therefore, N' now has a property that says that it halts if O applied to O_T doesn't halt, and it doesn't halt if O applied to O_T does. So to finish the argument, consider what happens if N' is applied to N_T. This machine would halt if N' applied to N_T did not, and would not halt if N' applied to N_T did. Since this is a contradiction, N' cannot exist. Since N' can't exist, neither can N, and therefore neither can M. Therefore, there is NO machine which can conclude if another arbitrary machine will ever halt.

The Halting Problem has a connection with another limitation in mathematics known as Godel's Incompleteness Theorem. One weak form of the Incompleteness Theorem is that it is impossible have a complete, consistent and sound axiomatization of all statements about the natural numbers. The strong form of Godel's Incompleteness Theorem does not require soundness as a property, merely provability.

2b. Godel's Incompleteness Theorem

Another demonstration of the limitations of mathematics is given by Godel's[6] Incompleteness Theorem. In this theorem, Godel states that it is impossible, in principle, to prove every true statement which can be formulated in an axomatic system. Put simply, there are statements which may be recognized as being true, but cannot be proven to be true by a step by step process. Some have hypothesized that statements like the Goldbach Conjecture (any even number is the sum of two prime numbers) may, in fact, be a statement of this sort.

Godel's proof involved the creation of a "Godel number" for each possible statement. This creation process consists three main steps:

1) Set up axioms for predicate calculus, along with rules of inference to get new formulas from old ones.
2) Set up axioms for arithmetic in the designed predicate calculus
3) Define a unique numbering system for each formula or sequence of formulas in the system.

The axioms Godel listed for predicate calculus provided for:

1) A true formula is implied by any formula

2) Implication distributes itself over formulas. If a formula A implies that if B is true then C is true, then if A is true, A implies B and A implies C.

3) If a formula A implies both something and it's contradiction, then A cannot be true, therefore NOT A must be true.

4) If a formula A implies B and x has no free occurrence in A, then F implies there exists some 'x' in G.

5) If A is true, and A implies B, then B is true.

Note that the last formula above is not a statement about formulas themselves, per se. It is actually a metastatement, referring to how to connect formulas in reasoning. The second requirement above is to construct "standard arithmetic". This is accomplished via the Peano[7] postulates:

$$\begin{aligned}
&1.\forall x\neg(0 = sx)\\
&2.\forall x, y((sx = sy) \rightarrow (x = y))\\
&3.\forall x(x + 0 = x)\\
&4.\forall x, y(x + sy = s(x + y))\\
&5.\forall x, y(x\mathrm{X}sy = x\mathrm{X}y + x)\\
&6.\forall x(x\mathrm{X}0 = 0)\\
&7.\forall x(x = x)\\
&8.\forall x, y, z((x = y) \rightarrow ((x = z) \rightarrow (y = z)))\\
&9.\forall x, y((x = y) \rightarrow (A(x, x) \rightarrow A(x, y)))
\end{aligned}$$

In the above formulas, "s" denotes the "successor function", which means that given a natural number (x), "s" acting on x produces (x+1). So in more comprehensible language, the rules above mean:

1) There doesn’t exist a natural number (x) for which zero is the successor. So zero is the smallest natural number.

2) For all possible values of natural numbers, if two numbers have the same successor, the two numbers are the same. So if I have two numbers, “x” and “y”, and they both have 4 as the next number up from them, the two numbers are equal (and in this case, equal to three). There is no way to have two different numbers have the same successor – thereby ensuring the number line is indeed that, a line.

3) For all values of “x”, x+0 is equal to itself. This defines what zero is.

4) For all values of “x” and “y”, a number “x” plus the successor of “y” is equal to the successor of (x+y). An elementary example of this would be if “x” equals three and “y” equals four. Then (x+sy)=3+5=8. And s(x+y)=s(7)=8.

5) For all values of “x” and “y”, “x” times the successor of “y” is equal to “x” times “y” plus “x”. For example, if x=3 and y=4, then (x*sy)=(3*5)=15, and (x*y +x)=(3*4 + 3)=15.

6) For all values of “x”, x times zero equals zero. This helps define the properties of zero.

7) This rule helps define equality. It shows that any two defined elements are equal if they are identical.

8) For all x, y, and z, if x is equal to y, then if x is equal to z then y is equal to z. This property, known as transitivity, shows that the property of equality carries over between different elements.

9) For all x and y, if x is equal to y, then A(x,x) is equal to A(x,y), where A is any formula having two free variables.

As a final rule, it is necessary to create a "rule of induction", which states that if a predicate is true with zero substituted in it, and if when that predicate is true for a given number it is also true for it's successor, then that predicate is true for all possible numbers x.

Finally, as mentioned above, it is necessary to construct a way to assign a unique number to any possible formula one might construct using the above set of formulas. This is accomplished by first assigning a natural number to all of the basic symbols in an arithmetic alphabet. Then, scan any given formula from left to right, and replace each symbol by a prime number raised to the natural number assigned to that symbol. The prime numbers are connected by multiplication.

For example, if I wished to encode the formula ~x into a Godel number, and "~" had a code number of "2", while "x" had a code number of "3", then this number would be:

$$\sim x = 2^2 * 3^3 = 81$$

Since each composite number has only one way to be factored into primes (by the Unique Factorization Theorem), each formula then is in correspondence with only one (albeit possibly quite large) number. This means that every integer greater than zero now corresponds to a unique set and order of symbols. Many of these integers will represent strings of symbols that are nonsensical. For example, as before, "~" has a Godel number of “2”, then $2^2*3^2*4^2=29$, which is “~ ~ ~”. Therefore the Godel number 29 doesn’t correspond to a meaningful formula – however, many other Godel numbers do.

To finally put all of this into a final proof of Godel’s Incompleteness Theorem, consider the following idea. Imagine that there is a proof (X) of a formula (Y). Since both the proof and the formula are written in symbolic language, they both have their own Godel numbers, (x) and (y). Each of these strings is over the domain of x, y, and z, as listed in the Peano postulates above. Suppose, then, that the formula (Y) is fed it’s own Godel number (y) and that the existence of a proof of the resulting formula is denied. If x is the Godel number of a proof of the formula obtained by substituting y into Y, then it is being said that such a proof does not exist, so this Godel number, x, does not exist.

So to boil this down into a few easier steps:

1) The statement that there is no proof which uses variables (x,y) is provable, and let p be the Godel number of that proof, P.
2) The proof is then true, since P is a proof of the statement with g substituted as one of the free variables.
3) But the existence of the proof contradicts the statement that there does not exist a proof, therefore we are left with the consequence that no proof exists.
4) Therefore, the formula stating there is no proof is true – since it has been established that the statement it makes is true – that there is no proof.

This proof then shows a fundamental limitation in the ability to “bootstrap” mathematics. If steps are taken algorithmically along a path, prior to Godel it was assumed that all true statements would be found along that path. Godel showed conclusively that there could exists statements seen “off the path”, which there was no way to directly access by proof, and yet are true.

With the construction of a computational system, and some understanding of its limits, the time has come to delve headfirst into one particular such environment – the neural network. Understanding what a neural network is and how it operates is the first step toward the eventual implementation of such a system using DNA.

Chapter 3 – Neural Networks

3a. What is a Neural Network?

First and foremost, it is important to recognize that a Neural Network is one example of a computational system. The next few chapters may appear to radically diverge from the first two, however I wish to emphasize that these are merely new clothes for the same creature. Since neural networks can be algorithmically described, they can be modeled by a computational system – and are therefore subject to the rules and limitations imposed by the theory of computation elaborated previously.

A neural network is an interconnection of fundamental processing elements (PE), which are connected to some or all PEs in a set through weights. These processing element are frequently referred to as 'neurons', due to their design similarity to brain structure. Connections which allow for a time delay or for no such delay are allowed. This definition, then, allows for a variety of schemes for system of PEs. As I will show shortly, the models of PEs as well as connection architectures have evolved over the history of neural networks. Along with this evolution, rich applications have developed. I will begin by explaining the history of neural networks, followed by the two major types of neural network systems: Biological Neural Networks (BNN) and Artificial Neural Networks (ANN).

3b. History of Neural Networks

The field of neural networks had its genesis in 1943 with the publication of "A logical calculus of the ideas immanent in nervous activity" by McCulloch and Pitts[8]. In this paper, the authors for the first time asserted that "*because of the 'all-or-none' character of nervous activity, neural events and the relations among them can be treated by means of propositional logic."* (italics mine). Further, McCulloch and Pitts stated that "*...for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes."*(italics mine). This means that any logical operation that can be found (with certain conditions) can be found using a NN. Also, the authors asserted that nervous activity and neural events could be modeled using these networks. This opened a new field for research into how thinking, memory, and perception could be modeled. McCulloch and Pitts had opened up an entirely new field – artificial intelligence.

The next major contribution to the idea of neural networks would not come from computer science, but rather from neuroscience. In 1949, The Organization of Behavior was published by Donald Hebb[9], which stated for the first time an explicit rule for synaptic modification during learning. Hebb proposed that those neurons which "fire together, wire together", meaning that behavior that caused causes a neuron to fire and repeatedly excite another neuron adjacent to it will cause a stronger connection to be formed between those two neurons.

These factors came together as researchers began serious attempts to model brain behavior on computers, with the first attempts at neural simulations[10] in 1956 showing that inhibition must be added to the simulation for a neural network to be effective. Uttley[11] was then able to use modifiable synapses to teach a network to classify sets of binary patterns. In the same year work on the associative memory was begun by Taylor[12]. The associative memory is a system where one memory can be connected to another, so that when one memory is activated, the other is recalled.

The field matured until, in 1958, Frank Rosenblatt[13] introduced a new type of neural network architecture, the Perceptron. The Perceptron differed from the McCulloch and Pitts model in a number of ways, including variable weights and node thresholds, a different value range for nodes, the lack of an inhibitory synapse, and perhaps most important – the ability of "train" the network using a training rule. Mathematically, this is allowing the network to alter the weights of the connections between nodes. Globally, this allows the network to improve its performance at a task. How the network decides to update the aforementioned weights is dependent on the program. That is to say, the designer of the network can tell the network how to update itself. Training rules and their implementation will be covered later in this section.

Rosenblatt's model, while being a considerable advancement over McCulloch and Pitts' model, eventually became "oversold". In response to this perceived overreach, Minsky and Papert[14] published Perceptrons in 1969, a book condemning the Perceptron and its claims. In this book, the authors showed that there was an entire class of problems which could not be successfully represented by the Perceptron. One of the simplest of these was the XOR gate, shown in Chapter 1. This caused research into NNs to wane, as scientists moved into attempting to solve problems in Symbolic AI. For more than ten years, the field of neural networks languished.

However, during this period, progress on NN continued, albeit at a much slower pace. It was during this time that the first work on lateral inhibition was introduced. Lateral inhibition is the idea that while a PE in a given population is excited, other PEs in the vincinity with automatically receive inhibition signals.[15] Grossberg termed this an "on-center off-surround" gain control for a group of neurons. Grossberg[16] also introduced several incarnations of neural networks referred to as the adaptive resonance theory (ART) models, which relied on a principle of self-organization which had not been seen previously. In the ART model, a layer of bottom up recognition as well as a layer of top down data generation compare results. If the patterns match, a state occurs where amplification of neural activity takes place. Grossberg named this "adaptive resonance".

Grossberg pointed out that short-term memories are related to neuron activation values, while long-term memories are related to weights in the connection matrix. This realization regarding storage of information for long time spans would become integral to Hopfield models of neural networks.

During this period, a mathematical problem known as the "credit assignment problem" was being researched. This problem may be defined as the problem of assigning credit or blame to the individual decisions that led to some overall result. In the context of neural networks, this is the question of how to assign credit or blame to hidden neurons in a multilayer neural network. Minsky[17] was the first to use this terminology in referencing the multilayer perceptron, but the question is germane to any neural network with a hidden layer. This problem is ubiquitous in the study of neural networks, since the issue of the successful performance for the network has to be assigned in some way to the individual PEs which comprise the network. Shun-Ichi Amari[18] was able to demonstrate how to solve this problem for adaptive neural networks.

In addition, Amari[19] developed concept forming networks (another forerunner to the Hopfield network). In this paper, Amari also develops neuron pools. In this model, fundamental PEs are small groups of connected neurons, rather than individual neurons.

Also during the 1970's, Teuvo Kohonen advanced the field with his work on adaptive networks. In Kohonen's original paper (1972), the PE is linear and continuous-valued, rather than an all-or-none binary model. Likewise, input values are continuous in this model. In addition, these networks were constructed to have many input and output PEs active at the same time, which is necessary for analysis of complicated input information, such as vision.

After the field of NN languished for over a decade, a solution was found to Minsky's challenge. It is the case that adding a third layer (also called a "hidden layer") of neurons to the NN architecture with the appropriate backpropagation learning rules[20] makes it possible to solve the problems Minsky et al. had pointed out. It is quite possible that Minsky himself knew of this possibility, but could not find a way to use a learning rule to update the network as needed in this new configuration. A diagram of this type of network is shown below:

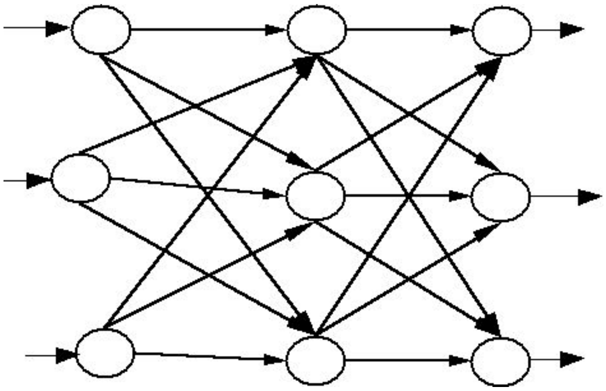


Fig. 3.1 – Basic Neural Network Structure

Once interest in the field of NN was revived, significant new advances were possible. Shortly before the advent of backpropagation solved the challenges put forth by Minsky, J.J. Hopfield[21] showed it was possible to use a neural network as a content addressable memory (CAM). This allowed neural networks to serve not only as classifiers, but as memories.

Linsker[22] was able to bring information theory to the fore in neural network research in 1988, formulating what came to be known as the "Infomax Principle". This principle is designed to preserve the maximum amount of possible information about input patterns, subject to the computational constraints of synapses being used.

Further, in 1988 the concept of "radial basis functions" (RBF) was developed as an alternative to multi-layer perceptrons by Broomhead and Lowe[23]. While RBF networks still employ three layers in their design, their learning rules are novel, in that the functions used for network updates have a diminishing effect on input values farther and farther from the neuron being updated.

With these designs implemented in modern hardware, the field of neural networks is still burgeoning today. Ubiquitous computers combined with new hardware implementations (brain on a chip, DNA computation, quantum computation) promise to continue advancements in the field of neural networks for decades to come.

3c. Biological Neural Networks (BNNs)

Biological neural networks consist of webs of interconnected neurons. A diagram of one of these neurons is shown below[24]:

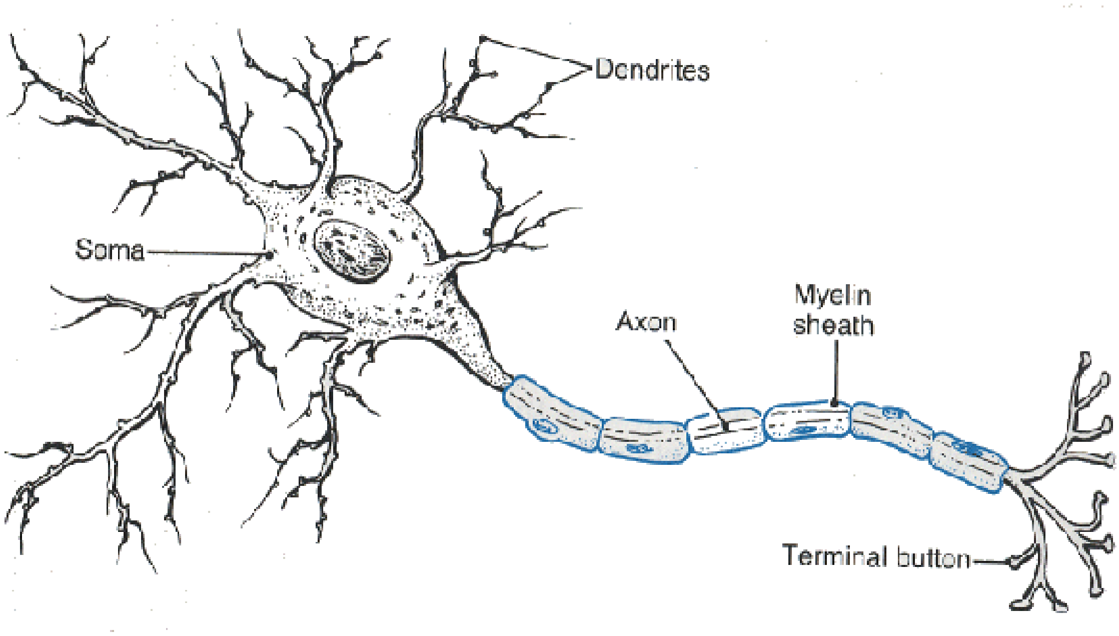


Fig. 3.2 – Diagram of a Biological Neuron [From Hale (REF 24)]

The primary components most relevant for my purposes include the dendrites, cell body, and axons (each of which is labeled above). Information is transmitted chemically through the terminal buttons of a set of neurons into the dendrites of connected neurons via neurotransmitters. These neurotransmitters then provide a signal to the cell body. If the collective contribution of the dendrites exceeds a certain threshold value, the neuron will fire, sending a signal along the axon and into the set of dendrites at the other end of the cell. Once this occurs, the signal is translated into chemical compounds and sent as neurotransmitters again to the next set of neurons.

3d. Artificial Neural Networks

Neural Networks arose out of attempts in artificial intelligence to model brain-like behavior. While the properties and abilities of a particular model of neural network (NN) might differ, an overall definition of a neural network is possible. A neural network is a set of computing elements (each of which can be thought of as an FSM) which collectively give rise to nonlinear behavior. These computing elements can be connected in such a way as to store information, perform specific mathematical operations, or find patterns. The particular ability of a NN will depend on how that network's individual elements are configured.

The two components of any NN are nodes and connections. A node takes inputs and, via some predetermined operation, calculates whether or not to produce a signal out. This outbound signal can have a varying strength, depending upon the function defined at the node producing it. This outbound signal is then sent to other nodes (and/or possibly the node itself). These signals are the connections mentioned above. In this way then, the NN is a crude approximation to behavior of actual neurons. A diagram depicting one possible NN configuration is shown below:

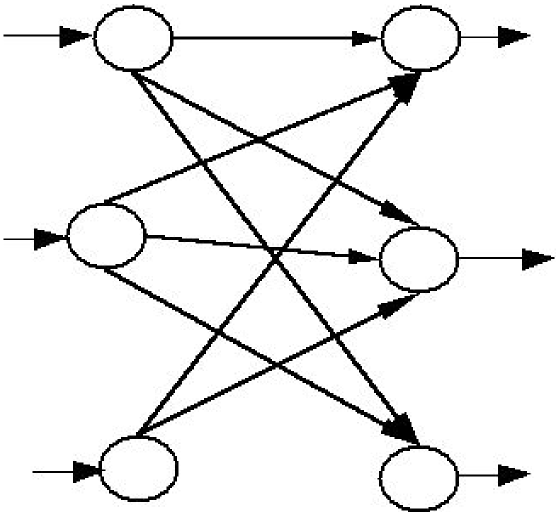


Fig. 3.3 – Feedforward Network Architecture

In the above diagram, the circles represent nodes, and the lines are connections between these nodes. While the above diagram depicts connections between a given node and each of the nodes in the subsequent layer, this is certainly not a prerequisite condition. The connectivity strength (if any) between two nodes is a function of the network architecture, and can be changed during the training of a network to solve a particular problem.

This system is represented mathematically by a set of vectors and matrices. Each layer of nodes is represented by a vector, with each node corresponding to an entry in that vector. The connections between nodes are entries in an m x n matrix, where ‘n’ is the dimensionality of the first layer of neurons, and ‘m’ is the dimensionality of the second layer of neurons. So, for example, if the values being output from the first layer of neurons are o$_1$, o$_2$, and o$_3$ (being output by node 1, 2, and 3 respectively), and the matrix of weights connecting the first layer to the second layer is given by

$$\widetilde{T} = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{pmatrix}$$

In this representation, matrix entry represents the connection strength between a node in the first layer and a node in the second layer. So, for example, the T$_{11}$ entry in the matrix above is the connection strength between the first node in the first layer and the first node in the second layer. The T$_{12}$ matrix entry is the connection strength between the first node in the first layer and the second node in the second layer. As can be seen from the diagram above, then, it is possible to have some entries equal to zero, if a node in the first layer is not connected to a particular node in the second layer. To calculate the total input to a particular node, one needs to multiply the weight matrix by the layer of nodes immediately preceeding the layer of interest. To find the inputs to the second layer of nodes, assuming the values exiting the first layer of nodes are given as above (o$_1$, o$_2$, o$_3$):

$$\begin{pmatrix} o_1' \\ o_2' \\ o_3' \end{pmatrix} = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{pmatrix} \begin{pmatrix} o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} T_{11}o_1 + T_{12}o_2 + T_{13}o_3 \\ T_{21}o_1 + T_{22}o_2 + T_{23}o_3 \\ T_{31}o_1 + T_{32}o_2 + T_{33}o_3 \end{pmatrix}$$

This output from a layer of neurons is fed in as the input for the subsequent layer of neurons. What an individual processing element (or node) does at this point is matter of the characteristics of the nodes being used.

Processing Elements

Processing elements (PE) are designed to simulate some of the salient features of actual neurons. In general, a PE (also sometimes referred to as a "neuron") is an element which takes a set of input connections and produces a single output. In this regard, the PE can be thought of as some form of many-to-one mapping. A diagrammatic representation of one of these nodes is shown here

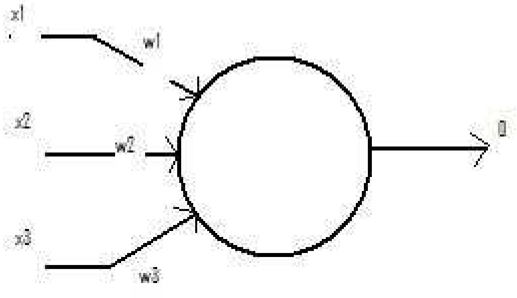


Fig. 3.4 – Diagram of a Neural Network Node

In the figure, each of the input lines goes to the node, which then acts on the collective inputs and produces a single output. The functional relationship for summing all of the inputs for a given neuron is given by:

$$net_i = \left(\sum_{j=1}^{d} w_{ij} x_j + I_i x_j + \theta_i \right)$$

where 'd' is the length of the layer of neurons being used as inputs (in the diagram above d=3), w_{ij} is a connection strength between a neuron being used as an input and the neuron currently under consideration, 'I' is any external driving term (which can vary from neuron to neuron) and Θ is a "threshold function" shifting the value required for a neuron to fire by a constant. Both "I" and "Θ" can be set to zero without a loss of generality. Here I have relabeled 'x' as the input vector:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

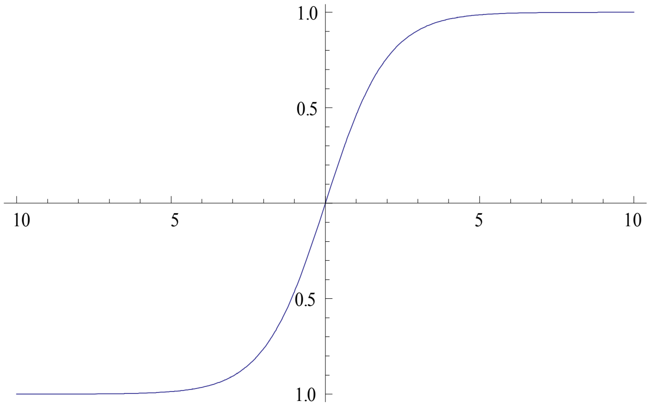
'f' is a nonlinear function (called the activation function). Two frequently used examples of this type of function are the bipolar continuous and bipolar binary functions:

$$f(net) = \frac{2}{1+e^{-\lambda net}} - 1 \qquad = \tanh\left(\frac{\lambda net}{2}\right)$$

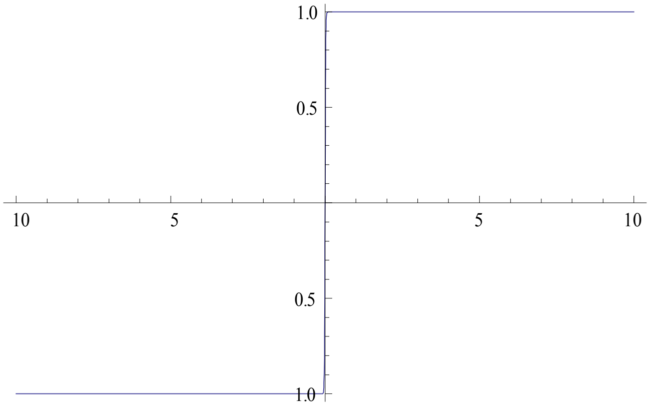
$$f(net) = \operatorname{sgn}(net) = \begin{cases} +1 & net \geq 0 \\ -1 & net < 0 \end{cases}$$

In the above binary bipolar activation function, when 'net' = 0, the output of the function has been selected to be +1. However, as Haykin[25] points out, when 'net' is exactly zero, "the action taken here can be quite arbitrary." Some architectures at this point force the output to +1 (as I have done), some force it to -1, and some leave the output of the neuron in the same state it was in prior to the activation function being applied.

Graphs of these functions are:



Binary Continuous Activation Function



Binary Polar Activation Function

Fig. 3.5 Artificial Neuron Activation Functions

The sigmoidal characteristic of the bipolar continuous activation function results in the designation of a "soft-limiting" activation function, whereas the discrete output of -1 or 1 from the bipolar binary function causes the terminology "hard-limiting" activation function to be used.

The "bipolar" designation is to note that either a positive or negative response of a neuron can be generated. Shifting the graph of each of these functions upward, so that only a "one" or a "zero" can be produced results in a "unipolar" activation function:

$$f(net) = \frac{1}{1 + e^{-\lambda net}}$$

$$f(net) = \operatorname{sgn}(net) = \begin{cases} 1 & net \geq 0 \\ 0 & net < 0 \end{cases}$$

Finally, there is another model of neuron firing which is based on probability instead of deterministic firing. Little[26] proposed the following form for a stochastic activation function

$$P(net) = \frac{1}{1 + e^{(-net/T)}}$$

where 'T' is a "temperature parameter" which allows the introduction of noise to the system. Any of these functions operating on an input vector $\{x_1, x_2, \ldots x_n\}$, which is a set of ‘d’ inputs to each neuron, will produce an output vector (assuming there are ‘d’ neurons in the layer being considered):

$$\vec{o} = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_d \end{pmatrix}$$

This output vector can then be fed, depending on the network’s architecture, to more neurons in another layer, some form of output device, or even back to themselves. The possible configurations for connection are shown below.

3e. Feedforward Networks

A feedforward network is one in which input signals are given to a layer of neurons. The neurons are allowed to act, and the outputs from the acting layer can either be read out or used as inputs for another layer. A graphical representation of this is shown below:

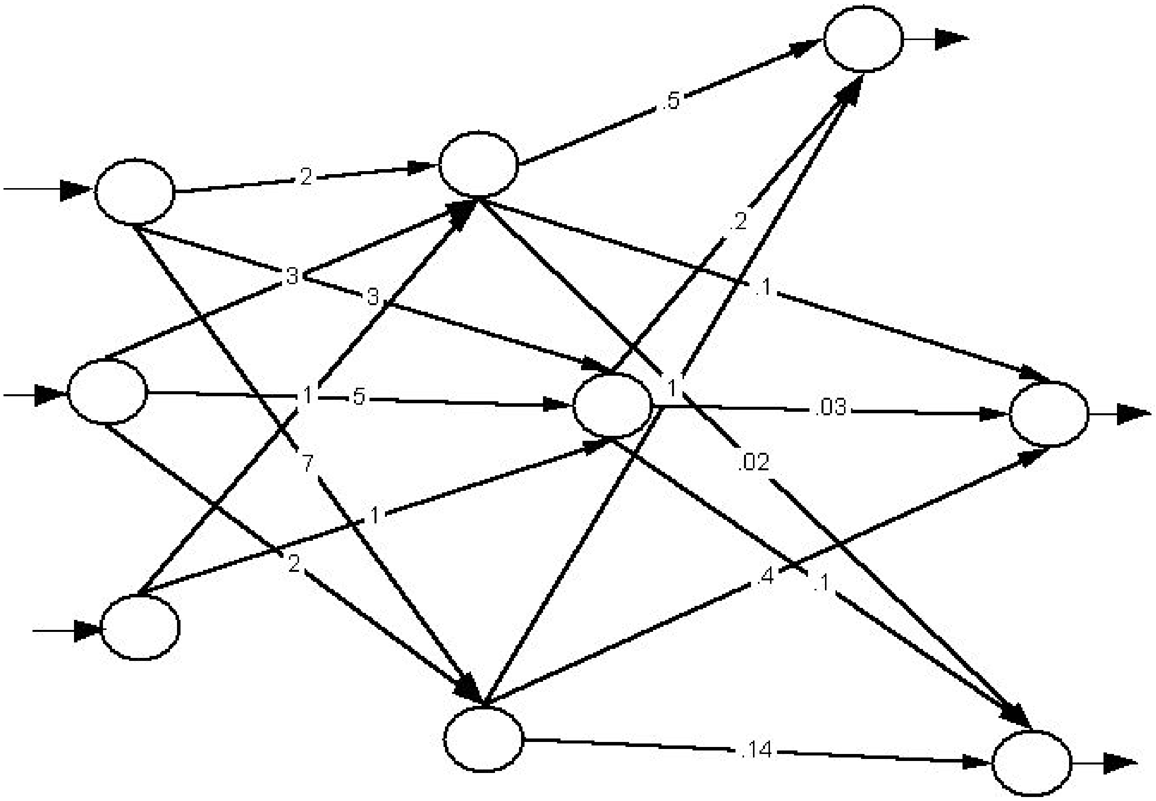


Fig. 3.6 – Weighted Feedforward Neural Network

As an example, assume that the initial input vector is:

$$\vec{x} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

From the graph above, the weight matrix connecting the first layer to the second layer is given by:

$$\widetilde{T} = \begin{pmatrix} 2 & 3 & 7 \\ 3 & 5 & 2 \\ 1 & 1 & 0 \end{pmatrix}$$

This gives the value for Tx shown earlier, which is the input at each neuron. Each neuron then acts on this input with its activation function, also shown earlier. The product of the above input vector and weight matrix yields the “net” input to the activation function:

$$net = \begin{pmatrix} 25 \\ 22 \\ 4 \end{pmatrix}$$

If the activation function was the binary bipolar function shown earlier, the output vector after the first layer would be:

$$o = f(net) = \begin{pmatrix} \dfrac{2}{1+e^{-25\lambda}} - 1 \\ \dfrac{2}{1+e^{-22\lambda}} - 1 \\ \dfrac{2}{1+e^{-4\lambda}} - 1 \end{pmatrix} \approx \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

The values of the output vector in this case are, for all intents and purposes, one. This is because the “net” value is so high. From the graph shown previously, the value of the output vector is much more sensitive to input “net” values for values of “net” close to zero. Reading from the graph, the weight matrix connecting the second and third layers is given by:

$$\widetilde{T} = \begin{pmatrix} .5 & .1 & .02 \\ .2 & .03 & .1 \\ 1 & .4 & .14 \end{pmatrix}$$

Allowing the outputs from the second layer to serve as inputs for the third layer, in this case the final output for this network is (if lambda equals 2):

$$o = \begin{pmatrix} .5528 \\ .3185 \\ .9121 \end{pmatrix}$$

3f. Equivalence of Linear Multilayer and Single layer Feedforward Networks

It should be noted that multilayer and singlelayer linear feedforward networks are equivalent. No gain in computing power is found by adding extra neuron layers. The reason for this is as follows:

Assume that there exists a three layer network with neuron layers f, g, and h, with connectivity matrices A (connecting layers f and g) and B (connecting layers g and h). If an input vector is presented to layer f, then g=Af. Allowing B to act on layer g, it is evident that h=Bg. But since this system is linear, h=B(Af)=BAf. This shows that the multilayer system can be replaced by an equivalent single layer system, with weights comprised of the products of the original connection matrices – no gain in computation power is seen. However, linear networks have no computing power and I do not discuss them further.

3g. Feedback Networks

A feedback network is similar to a feedforward network, with the exception that the outputs from the final layer of neurons are looped around and serve as inputs for the initial layer of neurons at the next time step. A simple case of this is shown below:

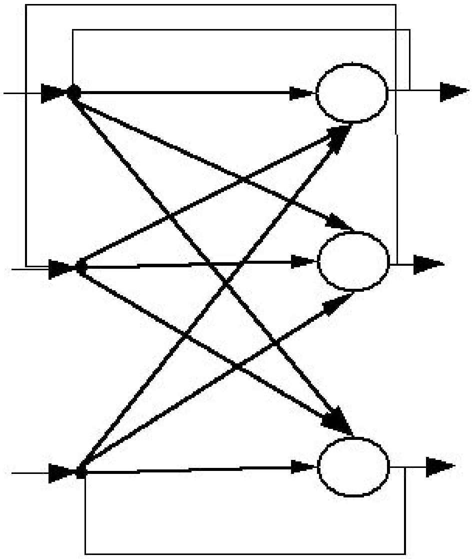


Fig. 3.7 – Feedback Neural Network

In this case then, the output at the next time step is a function of the output at the current time step. This can be written as:

$$o(t+\Delta) = f[\widetilde{T}\vec{o}(t)]$$

In this example, the original input is set at t=0 to initialize the network. It is then removed, and the input at the next time step is the output from the previous time step. If the state of the network is completely dependent on the history of the network starting at t=0, the network is referred to as recurrent.

It should be noted, in this case, that time has been quantized. The delay between states of the network makes it a straightforward task to generate state diagrams (which will be demonstrated in the section on Properties of Neural Networks shortly). Networks of this type are called discrete time networks. However, it is not necessary for a time delay to be present. Assuming an infinitesimal delay between input and output, the state of the network is a continuous time function, which is created by an architecture called a continuous-time network.

3h. Differences between Biological Neural Networks (BNNs) and Artificial Neural Networks (ANNs)[27]

There are several differences between the rules of operation for a BNN vs. an ANN:

1) Eccles' Law – This is a law in neuroscience which states that a neuron either excites or inhibits all neurons to which it is connected. In an ANN, an excitation could correspond to a positive weight, while an inhibition could correspond to a negative weight. Where BNNs can therefore have only a positive or negative weight coming from a particular neuron (but not both), ANNs have connections which can have either a positive or negative weight on a given connection coming from a particular neuron.

2) AC versus DC – In a BNN, a series of pulses across a synapse carries information. Also, a higher value of excitation or inhibition results in higher pulse rates. This is at least partially analogous to an alternating current (but only the forward portion of the current cycle). In ANNs, the opposite is the case. The signal going from one neuron to another is DC only.

3) Types of Processing Elements (PEs) – Where a brain has many types of neuron, an ANN typically only has one. It may be possible to have more types of PEs in the future, but that may not be necessary due to evidence indicating that any type of implementation can be carried out with only two types of PE.[20]

4) Speed – A BNN operates on a cycle time of approximately 10 to 100 milliseconds. In contrast, a desktop computer operates with a cycle time on the order of nanoseconds. With a number of operations required to calculate a new value for a PE (10 – 100), this makes the cycle time for a PE in an ANN approximately 100 nanoseconds. However, due to the parallel nature of a BNN, it is capable of performing some tasks much more quickly than today's ANNs.

5) Quantity of PEs – A BNN such as the human brain contains on the order of 1,000 main modules, each with approximately 500 million neurons.[28] This, in contrast with the relatively few numbers of PEs involved in the operation of an ANN makes the brain much more complex than present models are capable of simulating. It is noteworthy that perhaps if the number of PEs in an ANN should increase by orders of magnitude, a more complex programming scheme than is known today for sensibly adjusting the connectivity types and numbers of nodes in the ANN may be required.

3i. Neural Network Learning Rules

The strength of a neural network is in its ability to "learn", which implies some sort of improvement at a given task. However, even the definition of what constitutes learning can vary from researcher to researcher. A lucid definition of learning in the context of neural networks is given by Mendel and McClaren[29]:

"Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded.

The type of learning is determined by the manner in which the parameter changes take place."

When dealing with a neural network, the question naturally arises as to what is the best way to readjust the connection weights in the network to allow for settlement into a state which allows the network to successfully perform the desired task for which it is being trained. The particular procedure used to determine the adjustment of connection weights during the training phase of a network's operation is called a training rule. Several training rules have been developed since research into NN began, and which rule is applicable for a particular network will depend on both network architecture and the task to be performed. Below is a table summarizing learning rules and their application, with details on each rule following. (This discussion closely follows Zurada[30].)

Learning Rule	Weight Adjustment	Initial Weights	Learn Mode (S/U)	Neuron Characteristics	Neuron or Layer or Neurons
Hebbian	Co_ix_j (j=1,2,..n)	0	U	Any	Neuron
Perceptron	$c[d_i\text{-sgn}(\mathbf{w_i x})]x_j$	Any	S	Binary Bipolar, Binary Unipolar	Neuron
Delta	$c(d_i\text{-}o_i)f'(net_i)x_j$ (j=1,2,..n)	Any	S	Continuous	Neuron
Widrow-Hoff	$c(d_i\text{-}\mathbf{w_i x})x_j$ j=1,2,..n	Any	S	Any	Neuron

Correlation	cd_ix_j j=1,2,..n	0	S	Any	Neuron
Winner Takes All	$\Delta w_{mj}=\alpha(x_j-w_{mj})$ m-winning neuron	Random Normalize	U	Continuous	Layer of p neurons
Outstar	$\beta(d_i-w_{ij})$ i=1,2,..p	0	S	Continuous	Layer of p neurons

Fig. 3.8 – Neural Network Learning Rules

The general rule for training first involves establishing a syntax which denotes the contribution of the input, output, a learning signal, and a teacher's signal. Following the convention of Amari 1990[30], the "general learning rule" is stated as: The weight vector w_i = $[w_{i1}, w_{i2}, w_{i3}, \ldots w_{in}]^t$ increases in proportion to the product of input x and learning signal r. The learning signal is a function of w_i, x, and sometimes d_i (the teacher's signal).

This being the case, then, the equation governing the new weight vector at time t+1, in terms of the training which takes place at time t is:

$$\Delta \vec{w_i}(t) = cr[\vec{w_i}(t), \vec{x}(t), d_i(t)]\vec{x}(t)$$

where “c” is a positive number called the “learning constant”. This number determines the rate of learning. The function r changes from learning rule to learning rule.

Hebbian Learning

Hebbian learning involves the following rule[9].

$$r \underline{\underline{\Delta}} f(\vec{w}_i^t \vec{x})$$

This means that the learning signal is equal to the neuron’s output. Then, according to my general format for weight adjustment,

$$\Delta \vec{w}_i = cf(\vec{w}_i^t \vec{x})\vec{x}$$

This adjustment is an implementation of the classic rule “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”

Perceptron Learning Rule

This rule was developed for training the perceptron[13]. The rule is designed to make the learning signal equal to the difference between the desired output and the actual output.

This makes:

$$r \underline{\underline{\Delta}} d_i - o_i$$

and the corresponding weight adjustment:

$$\Delta \vec{w}_i = c[d_i - \mathrm{sgn}(\vec{w}_i^t \vec{x})]\vec{x}$$

Delta Learning Rule

The Delta learning rule was introduced by McClelland and Rumelhart[31] as one means of supervised training. The learning signal is defined as:

$$r \underline{\underline{\Delta}} [d_i - f(\vec{w}_i^t \vec{x})] f'(\vec{w}_i^t \vec{x})$$

where f'(wx) is the derivative of the activation function. The potential for success using this rule is predicated upon the realization that the squared error is defined to be:

$$E = \frac{1}{2}[d_i - f(\vec{w}_i^t \vec{x})]^2$$

then the error gradient vector value is:

$$\vec{\nabla} E = -(d_i - o_i) f'(\vec{w}_i^t \vec{x})\vec{x}$$

setting the weight changes equal to:

$$\Delta \vec{w_i} = -\eta \vec{\nabla} E$$

and combining these two quantities:

$$\Delta \vec{w_i} = \eta (d_i - o_i) f'(net_i)\vec{x}$$

This means that the adjustment of the weights is defined to minimize the squared error. Combining:

$$\Delta \vec{w_i}(t) = cr[\vec{w_i}(t), \vec{x}(t), d_i(t)]\vec{x}(t)$$

and

$$r \underline{\underline{\Delta}} [d_i - f(\vec{w_i}^t \vec{x})] f'(\vec{w_i}^t \vec{x})$$

I obtain

$$\Delta \vec{w}_i = c(d_i - o_i) f'(net_i)\vec{x}$$

This is the same as the above rule listed for weight adjustment. Therefore, this scheme is designed to minimize the squared error between the desired output and the output obtained for a particular scheme and time step.

Widrow-Hoff Learning Rule

The Widrow-Hoff[32] rule is another method of supervised training of a network. The learning signal and weight adjustment for this rule are given by:

$$r \underline{\underline{\Delta}} d_i - \vec{w}_i^{\,t} \vec{x}$$

$$\Delta \vec{w}_i = c(d_i - \vec{w}_i^{\,t} \vec{x})\vec{x}$$

This is a special case of the Delta learning rule, if the activation function is equal to one.

Correlation Learning Rule

By using the general learning rule:

$$\Delta \vec{w}_i(t) = cr[\vec{w}_i(t), \vec{x}_i(t), d_i(t)]\vec{x}(t)$$

and setting r = d_i. This causes the weight adjustments to be:

$$\Delta\vec{w}_i = cd_i\vec{x}$$

The rule sets the weight increase to be equal to the product of the desired response (d_i) due to an input (x_j). This is a special case of Hebbian learning, except that the learning is supervised, since the desired output is given.

Winner-Take-All Learning Rule

This rule is one which rewards a particular neuron in a layer by adjusting only the weights going to that particular neuron in that time step. The selection of which neuron receives the update is determined by which of the neurons in question receives the maximum activation:

$$\vec{w}_m^{\,t}\vec{x} = \max_{i=1,2,..p}(\vec{w}_i^{\,t}\vec{x})$$

Once the "winner" neuron is found, it is updated according to the formula:

$$\Delta\vec{w}_m = \alpha(\vec{x} - \vec{w}_m)$$

This learning rule corresponds to finding the weight vector that is closest to the input vector x.

Outstar Learning Rule

The Outstar rule is similar to the Winner-Take-All rule, except that instead of dealing with all weights going to a particular node, the weight adjustments are to all connections going out of a given node. The weight adjustments are:

$$\Delta\vec{w}_j = \beta(\vec{d} - \vec{w}_j)$$

Also, in contrast to the Winner-Take-All rule, the Outstar rule updates all neurons in a given layer, rather than just one particular “winner”.

These rules are useful in a single layer network. But as hinted previously, there is a challenge when assigning updates to neurons which are not directly visible to the outside world. This “credit assignment problem”, as it is known, deals with how adjust the weights of neurons which are not directly accessible in the network, since they are in one or more hidden layers.

The solution to this problem is to engage is learning via "back propagation". A NN engaged in learning through back propagation has two passes through the network instead of one. The first is a computation of results through the network with the initialized values of threshold elements. Once the results have reached the output layer they are compared with the desired results. The difference between the outcomes and desired results at each node can be used to compute an "error signal". This error signal can be used to directly compute the modifications of weights attached to that node by a learning rule. For example, with the "delta rule" mentioned previously, the modifications to the output layer of neurons would be

$$\Delta T_{ij} = -\eta \delta_j y_i$$

where "δ" is the gradient of the error function in weight space. This error function can be defined as

$$\varepsilon = \frac{1}{2}\sum_{j=1}^{d} e_j^2$$

where e_j is the error calculated earlier for the 'j'th neuron. The gradient of this function then, in the case of output neurons, is

$$\delta_j = e_j f_j^{'}$$

where f_j is the activation function for the jth neuron. In the case of hidden nodes, this rule becomes modified to

$$\delta_j = f_j^{'} \sum_{k=1}^{d} \delta_k T_{kj}$$

This alteration takes into account the weights of neurons attached to the node of interest.

The culmination of work on neural networks as classification and approximation systems has a rich history, beginning with simple elements to simulate neural signals and concluding with specialized hardware and software implementations still under development today. One particular type of network, a Hopfield Neural Network, will be of special interest.

Chapter 4. Hopfield Neural Networks

4a. Structure of Hopfield Neural Networks

The networks I have discussed so far focus on the architecture of the connections between nodes. As I have demonstrated shortly, these networks are useful for information processing tasks. However, as J.J. Hopfield[21] demonstrated, there is another task for which a neural network is suited – memory recall.

In a Hopfield Neural Network (HNN), information is stored in the connections between nodes. The connections in the network are initially established by presenting data patterns to be stored one by one to the network. Once these "memories" are stored, it is possible to recall them by the presentation of corrupted piece of information. While this memory model may not in all cases be the most efficient, it is helpful because it is easy to program as a "sum of outer products" system, which will be discussed shortly.

There are two main ways of storing data in a HNN; autoassociative memories or heteroassociative memories. An autoassociative memory is one in which when presented with the corrupted pattern, the network outputs the original pattern, uncorrupted and in its entirety. However, in a heteroassociative HNN, memories are stored in pairs. The presentation of the corrupted pattern produces the output of the associated object stored in memory.

For example, suppose I wanted to remember the name of someone, and couldn't remember it exactly – but could remember something similar to it (a common occurrence in humans). If I were to input the "corrupted vector" into my neural network, I should be able to recall the complete name of my friend, of which the uncorrupted components of the input vector is a portion:

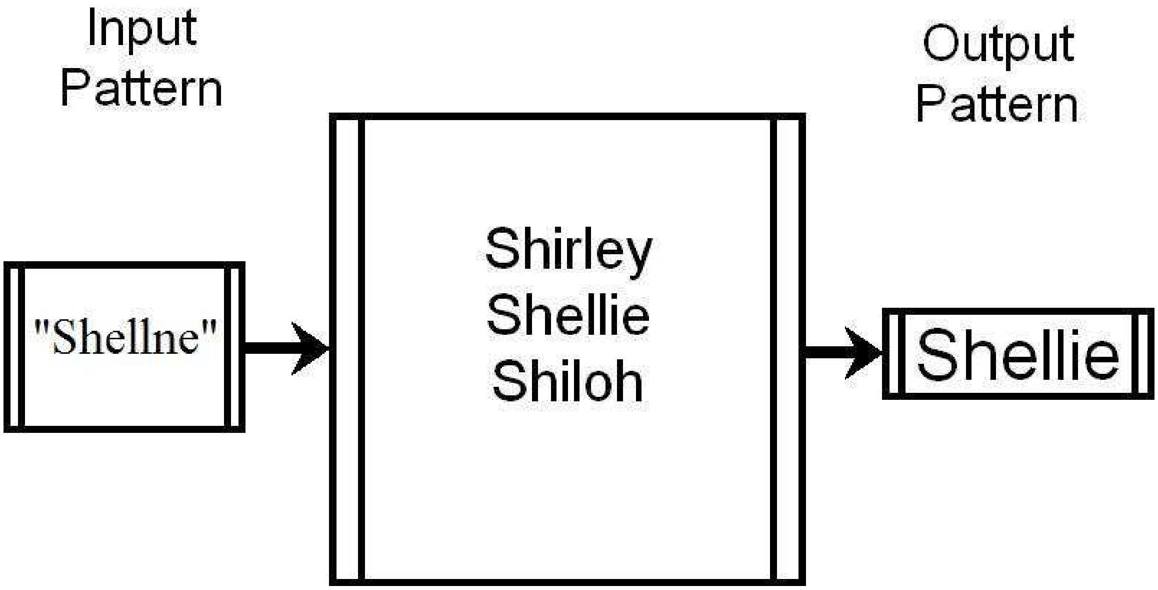


Fig. 4.1 – Memory Illustration for an Autoassociative Neural Network

In some situations, however, something will remind a person (or computer) of something else. This heteroassociative neural network will recall information that is "paired up", but not equal to, the clue for the original memory. As an example of this, suppose I see a person with blonde hair, and my friend has blonde hair. It is possible that the input stimulus (blonde hair) will produce a heteroassociated memory (an image of my friend):

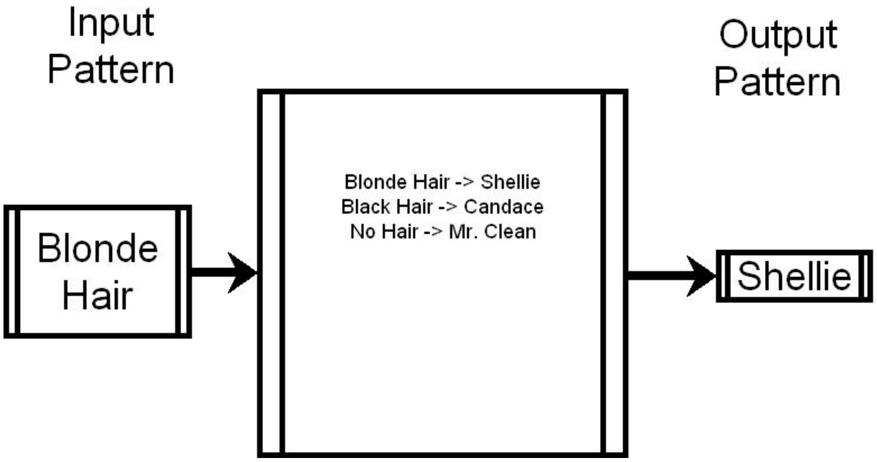


Fig. 4.2 – Memory Illustration for a Heteroassociative Neural Network

The HNN can simulate this behavioral feature. The construction of the working memory of an HNN involves the sum of the outer products of information vectors. To obtain an information vector, it is necessary to convert the data from whatever format it has naturally (such as a picture, sound, selection of text). For example, assume there was a set of three pictures, each containing three pixels. Each of these pixels could be black or white. Presuming I allow for black to be equal to a "1", and white to be equal to a "-1", then each picture can become an information vector with three entries – each entry being a "1" or "-1". This information (in this case binary bipolar in nature) is then stored as a matrix. Consider the following set of information vectors:

$$v^1 = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

$$v^2 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$v^3 = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$$

The memory matrix for these vectors would be (assuming I wish to create an autoassociative network):

$$\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}\begin{pmatrix} 1 & 1 & -1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}\begin{pmatrix} -1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 3 & -1 \\ 3 & 3 & -1 \\ -1 & -1 & 3 \end{pmatrix}$$

Obtaining the original memory is accomplished by an input of the original memory with some components' signs altered (called a "corrupted vector"). The output of the matrix times the corrupted vector is put through a hard limiting function, and the original vector is obtained. If, for example, the corrupt vector:

$$u^1 = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$$

is input into the memory matrix above:

$$\begin{pmatrix} 3 & 3 & -1 \\ 3 & 3 & -1 \\ -1 & -1 & 3 \end{pmatrix}\begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix} = Sgn\begin{pmatrix} 1 \\ 1 \\ -3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

the correct vector is recalled. If a heteroassociative memory is desired, the memory matrix is constructed using the outer products of pairs of vectors that are to correspond to one another, instead of using the outer product of each memory vector with itself.

Encoding Information

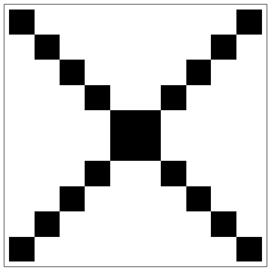
As mentioned in previous sections on the formation of HNNs, memories in an HNN are represented by 'd' dimensional vectors. In Hopfield's original paper memory components were unipolar. However, his subsequent publication[33] allowed for bipolar data representation, which was later extended to a continuous activation function.

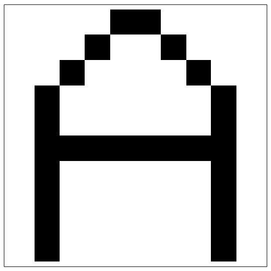
In Hopfield's model, the memory matrix is a neural network constructed of the sum of the outer products of the information vectors. His original work was designed to assess under what conditions a "corrupted vector" is restored to one of the original vectors encoded in memory.

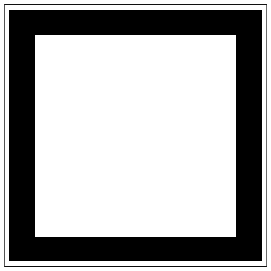
In contrast, it has been suggested by Mills, Yurke and Platzmann[91] that a duobinary representation of information could be possible. In this representation, information is still encoded in a binary bipolar format. However, a value of zero is used to represent a "missing" element of information. It is an interesting problem to compare these two representations of information loss. Some properties of HNNs with these vectors as inputs will be explored in Chapter 6.

4a.1. Encoding Input Vectors

Encoding information as an input vector first involves the conversion of information to be stored. Consider the encoding of a set of pictures, each represented by black or white pixels on a 10x10 grid. The following pictures







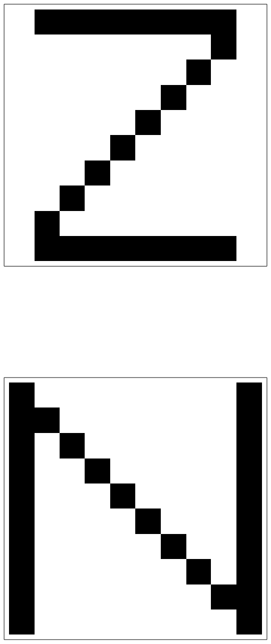


Fig. 4.3 – Binary Valued Pictures for Encoding in Neural Network

can be encoded as information vectors, counting each black pixel as a +1 and each white pixel as a -1. In the process of translating each two dimensional array into a vector, the position of each pixel in the picture is lost, and the process of recall will have to involve a translation from the output vector to a two dimensional array.

For example, the first picture (“X”) can be represented in an array as

{{1,-1,-1,-1,-1,-1,-1,-1,-1,1},{-1,1,-1,-1,-1,-1,-1,-1,1,-1},{-1,-1,1,-1,-1,-1,-1,1,-1,-1},{-1,-1,-1,1,-1,-1,1,-1,-1,-1},{-1,-1,-1,-1,1,1,-1,-1,-1,-1},{-1,-1,-1,-1,1,1,-1,-1,-1,-1},{-1,-1,-1,1,-1,-1,1,-1,-1,-1},{-1,-1,1,-1,-1,-1,-1,1,-1,-1},{-1,1,-1,-1,-1,-1,-1,-1,1,-1},{1,-1,-1,-1,-1,-1,-1,-1,-1,1}}

which when flattened into a vector becomes

{1,-1,-1,-1,-1,-1,-1,-1,-1,1,-1,1,-1,-1,-1,-1,-1,-1,1,-1,-1,-1,1,-1,-1,-1,-1,1,-1,-1,-1,-1,-1,1,-1,-1,1,-1,-1,-1,-1,-1,-1,-1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,-1,-1,-1,-1,-1,-1,-1,1,-1,-1,1,-1,-1,-1,-1,-1,1,-1,-1,-1,-1,1,-1,-1,-1,1,-1,-1,-1,-1,-1,-1,1,-1,1,-1,-1,-1,-1,-1,-1,-1,-1,1}
This vector, combined with vector representations of each of the other letters shown above, can be encoded into a memory matrix by taking the sum of the outer products of these memory vectors.

4a.2 Corrupted Vectors

Once the memory matrix has been constructed, it is possible to create corrupted vectors by selecting a number of pixels to be altered. When samples become large, the corruption scheme for the pixels isn't relevant, as long as the number of pixels corrupted is known. For example, below is a picture of a corrupted version of 'X' created by setting the independent probability that any individual pixel being corrupted to twenty percent

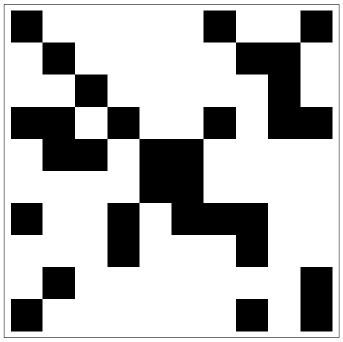


Fig. 4.4 – Corrupted Example Picture

Once the corrupted array is generated, it is flattened into a vector and the memory matrix acts on it, followed by the saturating function. The output vector is then reconstructed into an array by splitting every ten entries into a new row.

Attempting this gives the following output

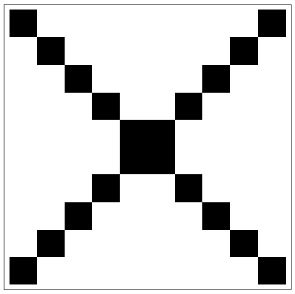


Fig. 4.5 – Restored Example Picture

showing that the corrupted picture has been restored to the correct original memory. A more extreme case can be shown by increasing the probability of each pixel being corrupted. An example of this can be shown below, where each pixel has a thirty five percent chance of being corrupted

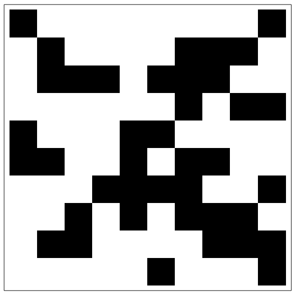


Fig. 4.6 – Second Corrupted Example Picture

This image is obviously considerably more corrupted than the first example. Once the array is flattened and allowed to be acted upon by the memory matrix and saturating function, the output array is

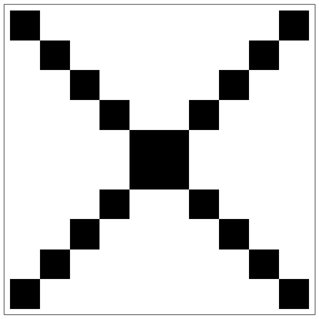


Fig. 4.7 – Second Restored Example Picture

once again giving the desired output. However, this may not be convergent on the first iteration in all cases. For example, the same simulation run again gives a corrupted picture

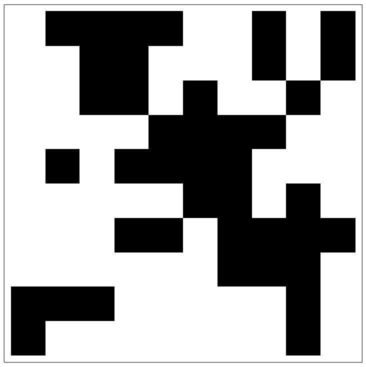


Fig. 4.8 – Third Corrupted Example Picture

Once put through the memory matrix, the output is

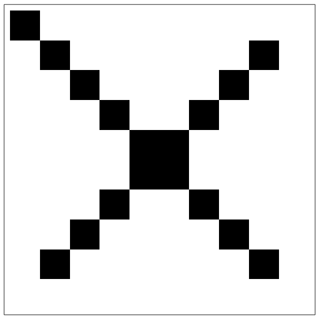


Fig. 4.9 – First Iteration of Third Corrupted Picture

If this output image is flattened into a vector again and acted on for a second time by the memory matrix and saturating function, the output is

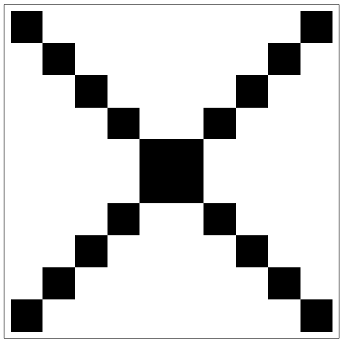


Fig. 4.10 – Second Iteration of Third Corrupted Picture

This has demonstrated a situation where the corruption in the memory was so great that a convergence to the correct memory required multiple iterations with the memory matrix.

Running this trial again with the probability of each pixel being flipped as fifty percent gives

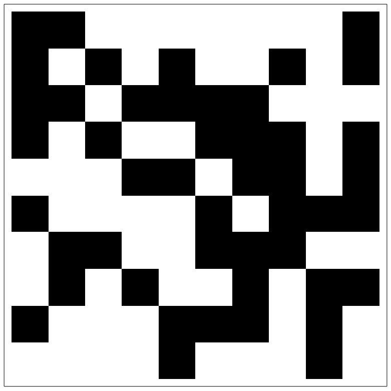


Fig. 4.11 – Highly Corrupted Picture

which, when put through the memory matrix and saturated gives

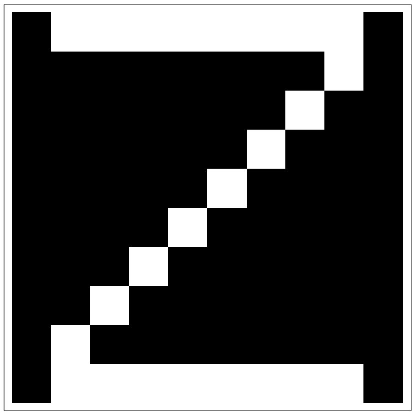


Fig. 4.12 – Output of Highly Corrupted Vector

which is obviously not the desired output vector. Further iterations of the vector acted upon by the memory matrix and saturating function produce the same output, implying that this is a stable memory, but one that was not originally encoded into the HNN. When the number of stored patterns exceeds two, it has been shown not only that new stable states may appear but that original memories may not remain indefinitely as stable points when new memories are added[34]. Further, it has also been demonstrated that it is possible to recall not only an original memory, but its complement instead[35].

4a.3. Incomplete Vectors

In contrast to Hopfield’s original idea of implementing a memory which could act upon a corrupted vector, Mills, Platzman and Yurke[91] have envisioned a memory system in which the HNN acts upon “incomplete” memories. In this situation, the implementation of information is duobinary, having with entries having values of ± 1 or 0, where zero means an absence of information for that entry.

For example, running the simulation in this new format (with a probability for each pixel being lost set to twenty percent) gives the following incomplete picture

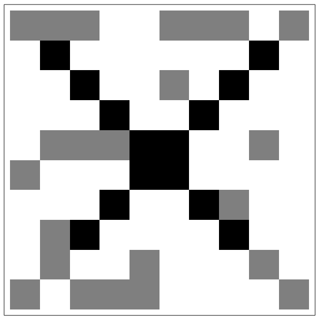


Fig. 4.13 – First Incomplete Picture

Once acted upon by the memory matrix and saturating function, the output vector (once reassembled) becomes

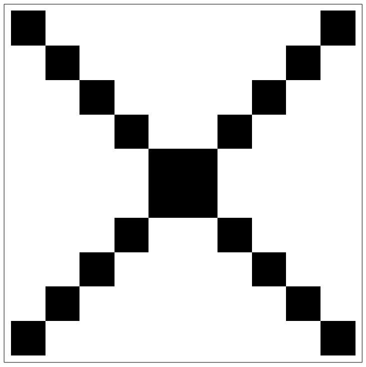


Fig. 4.14 – First Incomplete Picture Restored

which is the desired result. Notice the differences between this type of "incomplete image" and the previously explored "corrupted image". The grey pixels in the picture correspond to "missing" information, meaning that the entry in question could have a desired value of ± 1. Further, the pixels that are clearly defined retain the values ascribed to them in the original entry. This could mean the possibility of recall fidelity with more missing entries than would be possible with corrupted entries. As a preliminary example of this, consider the previous situation, where the probability of a pixel being corrupted was fifty percent. The network was unable to correctly recall one of the memories, presumably due to the existence of "spurious memories". If, however, I attempt this procedure using an incomplete vector with each pixel having a fifty percent chance of being incomplete as opposed to corrupted, I obtain the following for the incomplete picture

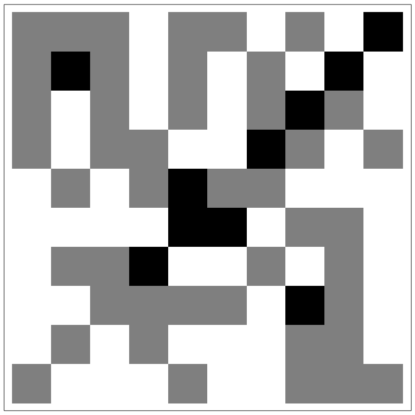


Fig. 4.15 – Second Incomplete Picture

Once acted on by the memory matrix and saturating function, the output is

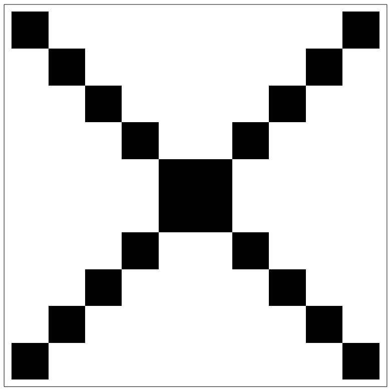


Fig. 4.16 – Second Incomplete Picture Restored

which is confirmed to be the desired output. A comparison of the properties of network recall as functions of these two types of memories is the subject of Chapter 6.

4b. Capacity and Fidelity of Hopfield Networks

Information Content of Vectors and Networks

The information content of a vector in a HNN can be thought of in terms of the vertices of the 'd' dimensional hypercube which makes up its state. Since the determination of the state of the output of a network (in binary bipolar representation) is a vertex on the hypercube, the decision about which vertex to settle upon can be thought of as a series of "yes or no" questions. This type of content generation is amenable to analysis in terms of Shannon Entropy. Solomon Golomb puts it eloquently when he says[36] a Shannon bit "is the amount of information gained (or entropy removed) upon learning the answer to a question whose two possible answers were equally likely, a priori." A more rigorous definition of this can be codified as follows, for a set of 'n' outcomes with probabilities p_1, p_2,.. p_n:

$$H(p_1,...p_n) = \sum_{i=1}^{n} p_i \log\left(\frac{1}{p_i}\right)$$

where all logarithms are base two. The minimum amount of information generated could be calculated by assuming that, overall, each entry has an equal probability of being either plus one or minus one. In this case, the number of Shannon bits generated with each information vector is

$$H(\frac{1}{2}(1),\frac{1}{2}(2)...\frac{1}{2}(d)) = \sum_{i=1}^{d}\frac{1}{2}\log(2) = \frac{d}{2}$$

Memory Capacity of the Hopfield Neural Network

Much interest has been taken in the capacity of a HNN. Hopfield's original paper[21] showed that Hopfield was able to recall n ≈ .15d (where 'd' is the dimensionality of the vectors) memories "before error in recall is severe". Further, he found that as the number of memories increased (with a fixed length of information vectors), vectors could still converge – but could frequently converge to stable states with errors in them. For example, in Hopfield's original publication, he found that when n=5 and d=100, convergence to the correct state occurred with a probability of 1. However, an increase in the number of memories to n=10 (d=100) lead to a convergence to the correct memory with a probability of ≈.7. The other 30 percent of entries converged to other stable states which were incorrect. Finally, an increase in number of memories to n=15 produced correct convergence with a probability of only .2. Further, there was a probability of .1 of converging to a state with between 10 – 19 errors in the stable state, a .15 probability of converging to a solution with 30 - 39 errors in the stable state, and a .2 probability of converging to a stable state with 20 – 29 errors.

Forshaw[37] confirmed this result and extended it to longer memory patterns (O(d/2)), and found that it was consistent with Hopfield's results. However, Peretto[38] contended that Hopfield modified this idea to show that the memory storage capacity is given by

$$M = N/K$$

where N is the number of neurons and K is a constant. A generalization of this result was provide by Keeler[39], who found that this relationship held for Hopfield models with higher order interactions among neurons, as well as for a three layer network. Storage capacity of binary patterns has been estimated to be $O((N/\log N)^2)$, each with size $O(\log N)$[40]. Finally, McEliece[41] showed that for a perfect recall from a vector with up to half of the entries corrupted, there should be no more than $d/(4\log(d))$ memories in the network, while if a small error in recall is tolerable, there can be up to $d/(2\log(d))$ memories in the network.

Errors in Recall

Forshaw[37] ran simulations on recall ability of HNNs as a function of corruption. He also termed this "incompleteness", but it differs in design considerably from the idea of Mills, Platzman and Yurke[91] which involves a duobinary representation of data. In Forshaw's simulation, data representation was binary bipolar. However, in order to delete a portion of the image, he set the entry to -1. In effect, this visually would produce a completely white page, instead of one with data represented by values of black and white pixels. Forshaw's initial simulations, run on a vector of length d=100, showed that for situations where fractions of a pattern were presented to the network, convergence was nearly complete for n=10 if the fraction of the pattern (compared to a particular desired pattern) was greater than .5. As n was increased (to n=30), convergence fell considerably to .5 when the fraction of the pattern shown to the network was also .5. These results assumed synchronous updates to the network. Convergence results were considerably worse for asynchronous updates to the network.

While the above references help determine the number of patterns which can be stored reliably in an HNN, further progress was made by Bruce[42], who demonstrated that for $\alpha \approx .069$, where

$$\alpha = n/d$$

there was a discontinuous change in the fraction of bits recalled correctly. Bruce was able to determine this by observing that in the phase diagram of the Hopfield network the thermodynamic states having overlap with the stored memories disappeared, implying a discontinuity in recall at this point.

Gardner[43] generalized this result to show that for $\alpha < 0.113$, there is a gap between the set of states close to the input vector and another set of states centered around the normalized Hamming Distance = .5 from the input vector (i.e. random vectors or other orthogonal memories). With this, Gardner attempted not only to show that the input vector would fall into a stable vector close to itself, but that there is a definite distance between the input vector and another set of encoded vectors which is far enough way that it will with certainty avoid this set of distant vectors as a result.

Improving Pattern Storage and Recall

Attempts have been made to improve upon the capacity of the HNN since inception. Originally, Hopfield proposed[44] a model of a HNN which would employ "unlearning" memories. This was accomplished by relaxing random states to stable states (which may or may not have originally been encoded in the HNN). This stable state was then used to form a matrix by constructing its outer product. A matrix proportional to this outer product was then subtracted from the original weight matrix to, in effect, "forget" that spurious memory. It was found that this method improved the number of memories that could be recalled correctly and that error correction was improved, but that recall fell to zero as ($n \rightarrow d$).[45] Kleinfeld[46] was able to further verify an increase in memory capacity for an HNN using an "unlearning algorithm".

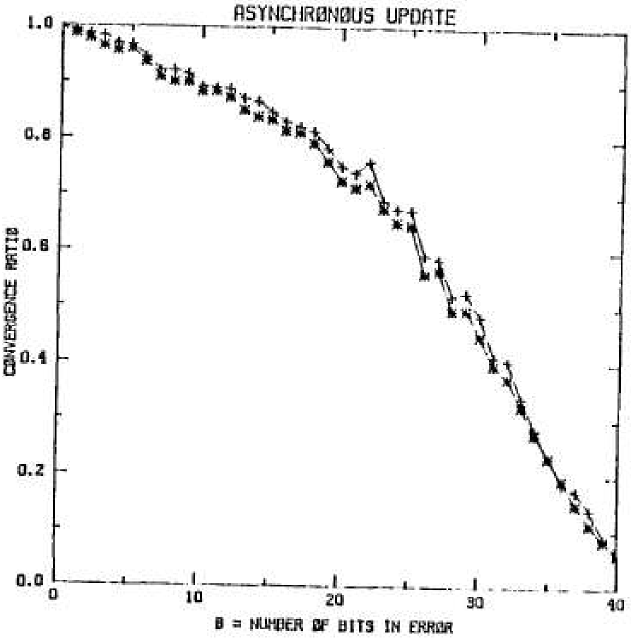
So far, the references mentioned have been measuring the capacity of recall in terms of the number of input vectors and the number of recalls that were correctly performed. However, Abu-Mostafa[47] demonstrated that the asymptotic information capacity of a HNN of 'd' neurons is on the order of d^3 bits. He also demonstrated that the number of state vectors which can be encoded with stability in the same network is bounded by 'd'.

Horn[48] observed that while orthogonal vectors are desired for faithful memory recall, they do not guarantee that a "faithful set" will be formed. That is, it is possible to still generate output vectors that were not members of the original set. Further, Horn was able to construct “faithful sets” by requiring that “certain subsets be forbidden to have a total binary product proportional to the unit vector”.

4c. Altered Constraints and Memory Models for the Hopfield Neural Network

Relaxation of the T_{ii}=0 Condition

One of the conditions mentioned by Hopfield for unconditional convergence of the HNN is the removal of the diagonal elements (T_{ii}) of the memory matrix[21]. Gindi[49] modeled discrete time HNNs (d=100, n=10) which had a small number of corrupted bits. The small number of bits corrupted was set such that the number of corrupted bits presented a corrupted memory whose likelihood of having a greater Hamming Distance from the correct memory than another of the stored memories was very small. Below is shown the convergence from Gindi’s data for both an asynchronous and synchronous update scheme



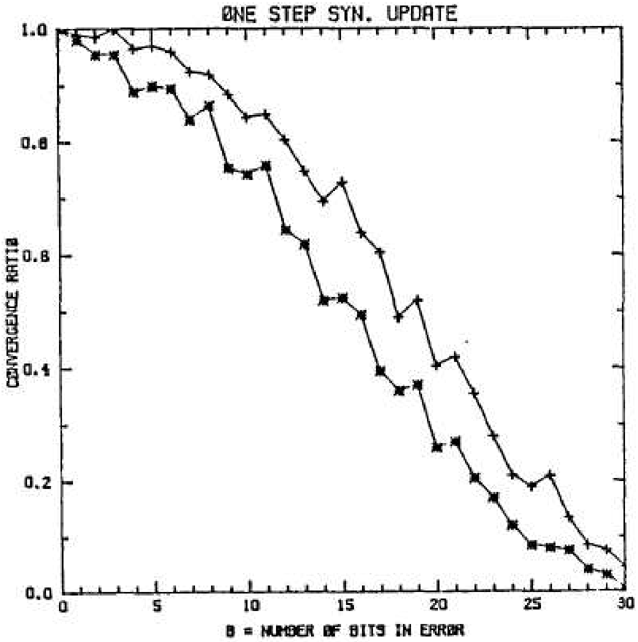


Fig. 4.17 – Gindi's Data for an Asynchronous Update Scheme

(The curve marked with "*" is for cases where T_{ii} is equal to zero, and the curve marked with "+" is for the cases where T_{ii} is not equal to zero.)[49]

In the above plots, the probability of convergence is plotted on the y-axis by calculating the percentage of initial states that converged to the reference memory. As can be seen from the above plots, the case where T_{ii} is not equal to zero actually performed better in Gindi's simulations.

These results were confirmed and extended to networks which were binary unipolar by Gmitro[50], as shown below

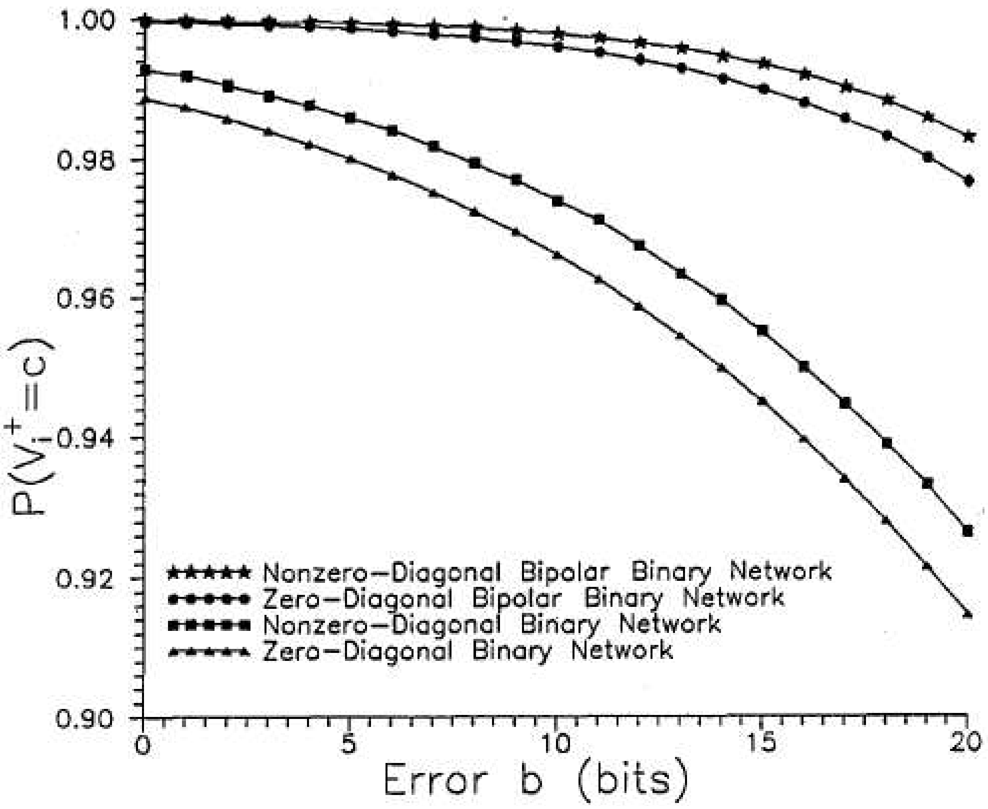


Fig. 4.18 - Gmitro's Data for Bipolar and Unipolar Networks

Here the probability an individual bit will be correct after update is plotted as a function of the number of bits that are corrupted in a memory vector. As can be seen, not only are the cases with nonzero diagonal terms better in terms of performance in both binary bipolar and binary unipolar representations, but Gmitro showed that for the standard HNN, binary bipolar networks performed better overall than binary unipolar networks. DeWilde[51] showed that while adding a small positive diagonal term to the memory matrix did improve performance in some cases when compared to a zero diagonal matrix (although sometimes within the error bars of each others' results), adding nonzero terms along the diagonal definitely increased the number of spurious states in the network. DeWilde explains that this is from states that were not stable for a zero diagonal memory becoming stable due to positive self-feedback.

Relaxation of the Symmetry Condition for Memory Matrix Elements

Hopfield's original paper[21] held that the elements of the memory matrix for a HNN had to be symmetric ($T_{ij} = T_{ji}$). However, it has been pointed out that biological neurons do not obey this property[52]. Further, some researchers since Hopfield have had interesting results experimenting with the removal of this symmetry condition. For example, Chengxiang[53] et al found that when an asymmetric component to the memory matrix was introduced, the number of random inputs that converged to memory states was increased. However, the size of the attraction basin for a given memory did not show a change in size. Chengxiang concluded that this apparent contradiction is due to the destabilization of the spurious memory attractors by the new memory components.

Chen[54] and Amari derived analytical results for HNNs with asymmetric connections and found conditions under which the network would be locally and/or globally stable. Four years later, Zhao[55] was able to use principles of the stability of basins of attraction to design optimal networks by tuning the "degree of stability" of the memory matrix elements, also defining a constant of symmetricity equal to

$$\sigma = \frac{2\Gamma}{d(d-1)}$$

where Γ is the number of connections which are symmetric in a given memory matrix. Therefore, $0 \leq \Gamma \leq 1$, with 0 meaning the matrix is completely antisymmetric and 1 meaning it is completely symmetric. Issues in this area are still under consideration, as Zheng[56] found that HNNs with asymmetric elements and some matrix components equal to zero still perform reasonably well compared to fully connected symmetrically weighted memory matrices; he pointed out that this could be useful in hardware implementations as it can be used to reduce fabrication difficulty.

Altered Memory Models

While Hopfield's original memory model was to store memories that were encoded in the formation of the memory matrix, progress has been made since the inception of the HNN in novel applications and configurations of memory.

For example, Dotsenko[57],[58] proposed a model of memory which layered multiple Hopfield nets in the same working substance. In his example, Dotsenko proposed using an Ising model of spins. He realized that it was possible to divide the system of spins into a hierarchy of clusters, each containing several spins. Each of the clusters would serve as an independent HNN, and presumably function as part of a larger HNN at a higher hierarchical level. Further progress in hierarchical models of HNN was made when Cortes[59] was able to induce hierarchically ordered memories by using an update scheme involving updated weights using a Parisi[60] function. The "Parisi function" is a way of simulating the breaking of an order parameter in a matrix which describes the behavior of a spin glass. This is in contrast to Dotsenko's implementation, which required previous knowledge of all previously input patterns to input new information.

Other memory models may involve novel vectors or memory matrices. Amit[61] experimented with HNN vectors where the set of patterns input was biased in some arbitrary way, by setting the sum of all vector components equal to some small constant.

4d. Measurements of Convergence – Energy Landscapes

It has been mentioned in previous sections that the formation of the memory matrix in an HNN produces an "attractor" for each memory, along with a set of "spurious states", which are attractors to which convergence is possible, but were not encoded intentionally in the memory. A quantitative treatment of this issue is now presented.

First, it is useful to address how someone might describe a function for the dynamics of a HNN. A physical system with properties similar to neural networks already exists in spin glasses. A spin glass is a system of atoms (molecules, etc), each with a magnetic moment. These magnetic moments are "frustrated", in the sense that the structure of the system prevents the collapse to a single minimal-energy state. In the case of a spin-glass, this frustration is augmented by stochastic disorder, meaning that ferromagnetic and antiferromagnetic configurations of spins are distributed randomly throughout the structure of the material. A model of a spin-glass which could be exactly solved was introduced by Sherrington[62]. In Sherrington's paper, the Hamiltonian of a spin-glass system is given by

$$\widetilde{H} = -\frac{1}{2}\sum_{i \neq i} J_{ij} S_i S_j$$

where S=±1. Hopfield[21] recognized that this same type of equation could be applied to his form of neural networks, where the magnets of the system are replaced by computing nodes, and the interactions between magnets are replaced by connection strengths in the weight matrix. This then allows for a general form of energy for a HNN. Note that this is not "energy" in any physical sense of a neural network. The term is a carryover from the spin-glass model. However, the dynamics used and mathematical machinery employed in this spin-glass analysis can be applied to an analysis of the HNN. The energy of a HNN is a scalar valued function with the equation

$$E = -\frac{1}{2}\vec{v}^{\,t}\widetilde{T}\vec{v} - \vec{i}^{\,t}\vec{v} + \vec{t}^{\,t}\vec{v}$$

Since external inputs (i) and threshold constants (t) for the network can be set to zero without a loss of generality, this function is frequently represented as

$$E = -\frac{1}{2}\vec{v}^{\,t}\widetilde{T}\vec{v}$$

Notice the similarity with the spin-glass Hamiltonian. A graph of this function for the case where $T_{ii} = 0$ is shown here

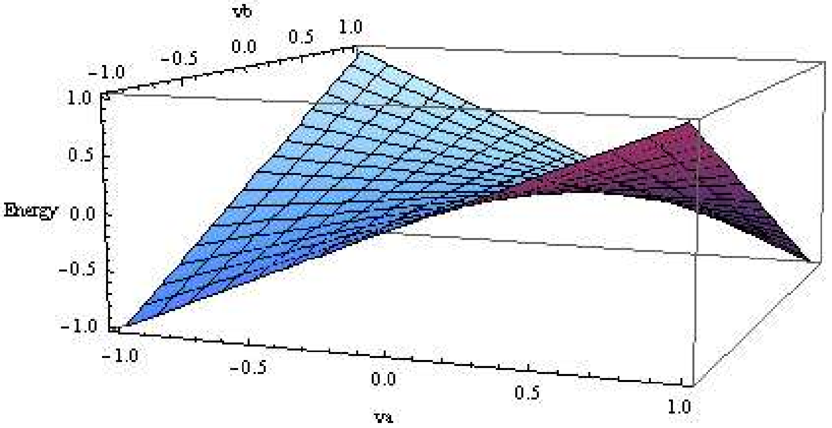


Fig. 4.19 – Energy Landscape for a Two Dimensional Memory Matrix

As can be seen from the above plot, energy minima exist when $v_a=v_b=-1$ and when $v_a=v_b=1$. If, however, the $T_{ii}=0$ is not enforced, the energy landscape is altered. The equation for the energy of the network is altered from

$$E=-v_a v_b$$

to

$$E=-v_a v_b+v_a^2+v_b^2$$

However, it was pointed out by Gindi[49] that for a binary bipolar valued neural nets, each component is ±1. Therefore, the energy can be reduced to

$$E=-v_a v_b+nd$$

where ‘n’ is the number of vectors, and ‘d’ is the dimensionality of each vector (in the example here, d=2). Therefore the values of energy might change, but the number of energy minima is not altered.

Lyapunov Functions

The general class of functions which is employed to prove the stability of certain types of systems (such as a spin-glass) in the manner mentioned is called a Lyapunov function. This is a function based on the system state. To get an idea of how a Lyapunov function is found[30], consider a system governed by a set of first order differential equations (either linear or nonlinear):

$$\begin{aligned} \dot{x}_1 &= f_1(\vec{x}) \\ \dot{x}_2 &= f_2(\vec{x}) \\ &\cdots \\ \dot{x}_n &= f_n(\vec{x}) \end{aligned}$$

Further, assume that $f_n(0)=0$ for all 'n'. A condition for the state vector to migrate inevitably toward this minimum can be formulated. If a positive definite function $E(\vec{x})$ can be found with the properties that (closely following Zurada[30])

1) E is continuous with respect to all components x_i
2) $\dfrac{dE(\vec{x}(t))}{dt} < 0$

then the result of this function will converge toward a minimum. This is the definition of a Lyapunov function. It is possible to have more than one Lyapunov function for a given system. Further, there is no known unique and best method for the identification of such a function. Frequently these functions are found by physical insight, similar to what was done by Hopfield in comparing the HNN to a spin-glass.

Network Updates

The gradient of the Energy function, in general, is

$$\vec{\nabla}E(\vec{v}) = -\frac{1}{2}\vec{\nabla}(\vec{v}^{\,t}\tilde{T}\vec{v}) = -\frac{1}{2}(\tilde{T}\vec{v} + \tilde{T}^{\,t}\vec{v})$$

However, if Hopfield's original prescription is followed for construction of the memory matrix, $T_{ij}=T_{ji}$ and this can be simplified to

$$\vec{\nabla}E(\vec{v}) = -\frac{1}{2}\vec{\nabla}(\vec{v}^{\,t}\tilde{T}\vec{v}) = -\tilde{T}\vec{v}$$

It was shown by Petsche[63] that this gradient is a linear function of the Hamming Distance between the vector in the energy function and each of the memories encoded in the memory matrix.

4e. Measurements of Convergence – Hypercubes

A second measure of the convergence of a HNN is the distance from the desired memory in a state space. This space is frequently represented as the volume in 'd' dimensions bounded by ±1. For example, for the model shown above, where d=2, the state space can be represented as the vertices below

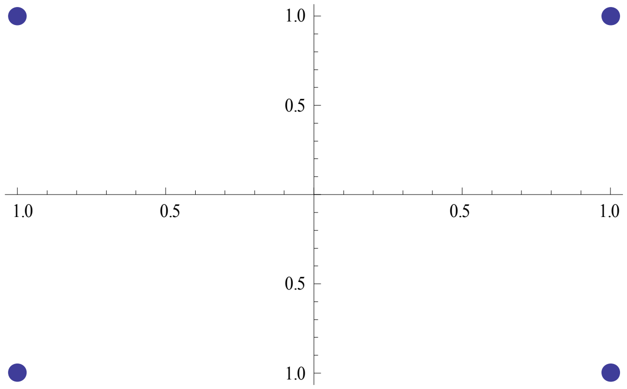


Fig. 4.20 – State Space for a Two Dimensional Neural Network

A few items of note regarding the state space:

1) Network structures using a hard limiting function will by necessity have states constrained to the vertices shown above, since they are saturated to ±1 (if f(0) =1)

2) Network structures using soft limiting functions are constrained to be within the perimeter of the square shown above.

3) The distance between the given output vector and the desired output vector in the case of networks with a hard limiting function is frequently expressed as the "Hamming Distance" between the two vectors. This is defined to be the number of places in which their vector entries differ. In terms of the state diagram, this is equal to the number of edges between the two vectors.

4f. Measurements of Convergence – Statistical Approaches

Another possible measurement of the effectiveness of a HNN is to determine, statistically speaking, what the likelihood is of a vector converging to the correct memory. In this case, I will illustrate using a "clue vector" (a vector with some components missing, so they are set to zero). To assess the efficacy of a Hopfield Neural Network (HNN) at memory recall as a function of clue length (q), vector length (d), number of vectors encoded (n), and number of iterations to gain statistics (I), it is instructive to first demonstrate the recall of a HNN for toy models, since I will be addressing considerably more advanced networks in this manner in Chapter 6.

As a simple example to demonstrate that convergence is indeed possible, consider the following set of vectors:

v^1 = {-1,1,-1,-1,1,-1,-1,1,1,-1}

v^2 = {1,1,-1,1,1,-1,1,-1,1,-1}

v^3 = {-1,1,1,1,1,-1,-1,-1,1,-1}

Taking the outer product of these vectors with themselves and summing them, then removing diagonal elements results in the memory matrix (T):

$$\widetilde{T} = \begin{bmatrix} 0 & -1 & -1 & 1 & -1 & 1 & 3 & -1 & -1 & 1 \\ -1 & 0 & -1 & 1 & 3 & -3 & -1 & -1 & 3 & -3 \\ -1 & -1 & 0 & 1 & -1 & 1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 & 1 & -1 & 1 & -3 & 1 & -1 \\ -1 & 3 & 1 & -1 & 0 & 3 & 1 & 1 & -3 & 3 \\ 1 & -3 & 1 & -1 & -3 & 0 & 1 & 1 & -3 & 3 \\ 3 & -1 & -1 & 1 & -1 & 1 & 0 & -1 & -1 & 1 \\ -1 & -1 & -1 & -3 & -1 & 1 & -1 & 0 & -1 & 1 \\ -1 & 3 & -1 & 1 & 3 & -3 & -1 & -1 & 0 & -3 \\ 1 & -3 & 1 & -1 & -3 & 3 & 1 & 1 & -3 & 0 \end{bmatrix}$$

To test if an individual clue vector will converge to a desired memory, it is necessary to use one of the original memory vectors with components missing (represented as zero entries in the input vector). The Memory Matrix is then multiplied by the input clue vector, and a hard limiting function, Sign, is applied to the result, giving a vector O. Success is achieved if the result (O) is equal to the original vector encoded in the matrix (v).

Trying this for the above case, setting q=1, gives the following clue vectors (u):

U^1 = {-1,0,0,0,0,0,0,0,0,0}

U^2 = {1,0,0,0,0,0,0,0,0,0}

U^3 = {-1,0,0,0,0,0,0,0,0,0}

The problematic nature of recall can already been seen here. Since there are only two choices (1, -1) for any individual vector component entry, the clue is quite possibly too short to successfully determine which is the desired output vector. Letting the memory matrix act on each of these clues and then following with a hard limiting function gives:

Sign[T.U^1] = {0,1,1,-1,1,-1,-1,1,1,-1}

Sign[T.U^2]= {0,-1,-1,1,-1,1,1,-1,-1,1}

Sign[T.U^3]= {0,1,1,-1,1,-1,-1,1,1,-1}

It can readily be seen here that the length of the clue is inadequate to converge completely to memories encoded in the HNN. The degree of convergence can be quantified by using the Hamming Distance.

As mentioned in the previous section, the Hamming Distance between two vectors is the number of entries in which they differ. So, for example, the vectors:

$$v^1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$v^2 = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$$

have a Hamming Distance (HD) of 2, since their entries differ in 2 places. Comparing the produced output vector (O^n) to the desired output vector (v^n) by calculating their HD can be illustrative. For complete convergence, HD = 0. For the above scenario, the HDs are as follows:

q=1	
HD(O^1,v^1)	2
HD(O^2,v^2)	6
HD(O^3,v^3)	3

Fig. 4.21 – Table of Hamming Distances for Output Vectors and Original Vectors q=1

A helpful way to view this data is as a plot of HD vs. q. For the above data, this yields:

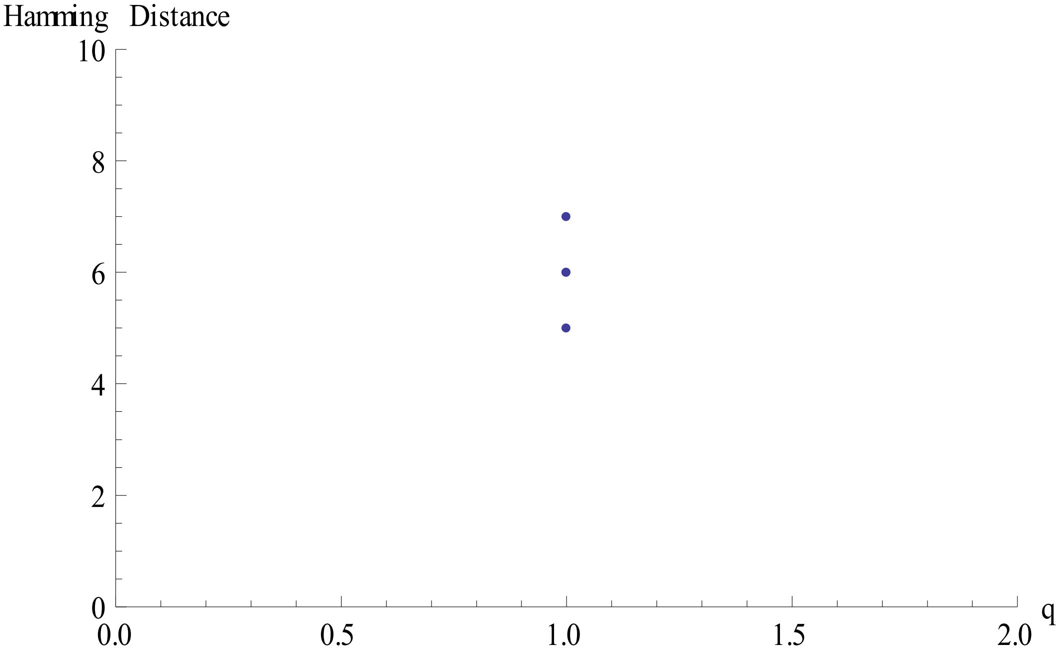


Fig. 4.22 – Plot of Hamming Distances for Output Vectors and Original Vectors q=1

In this case, there are three points, one each for the Hamming Distance between the output and the desired vector for a particular input clue of length equal to one. Repeating the above procedure, this time with q=2 gives:

q=2	
HD(O^1,v^1)	3
HD(O^2,v^2)	2
HD(O^3,v^3)	3

Fig. 4.23 – Table of Hamming Distances for Output Vectors and Original Vectors q=2

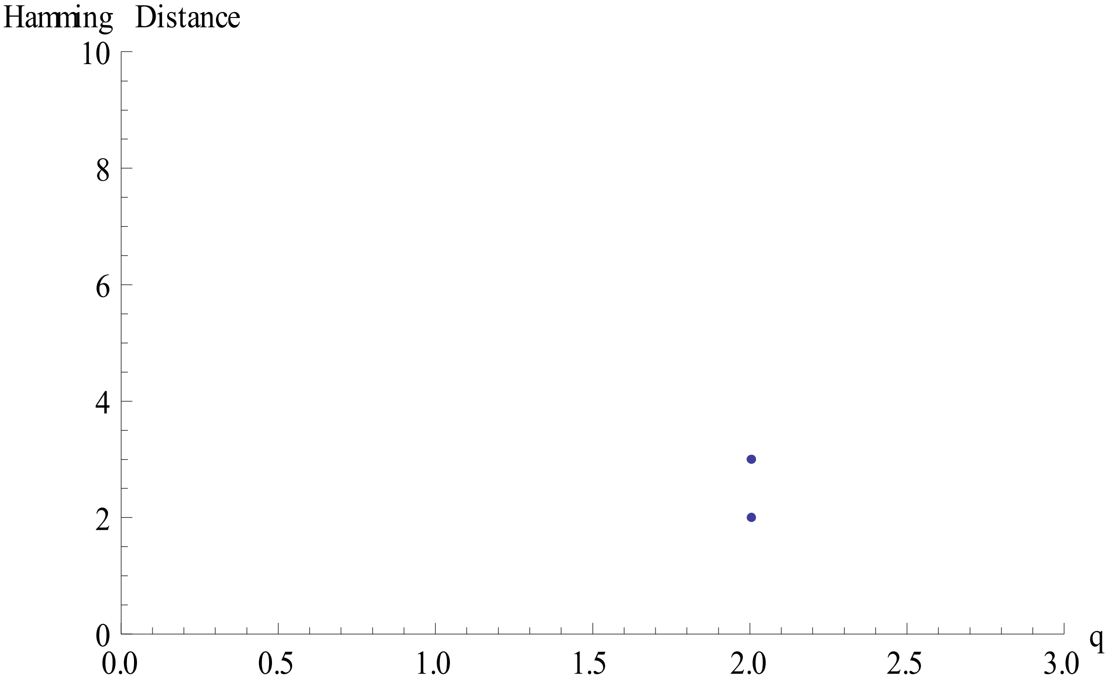


Fig. 4.24 – Plot of Hamming Distances for Output Vectors and Original Vectors q=2

Combining this with information on all clue lengths gives the following data table:

	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
HD(O^1,v^1)	2	3	2	0	1	1	1	0	0	0
HD(O^2,v^2)	6	2	2	0	1	1	0	0	0	0
HD(O^3,v^3)	3	3	2	0	1	1	1	0	1	1

Fig. 4.25 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, First Iteration

A plot of this information is shown:

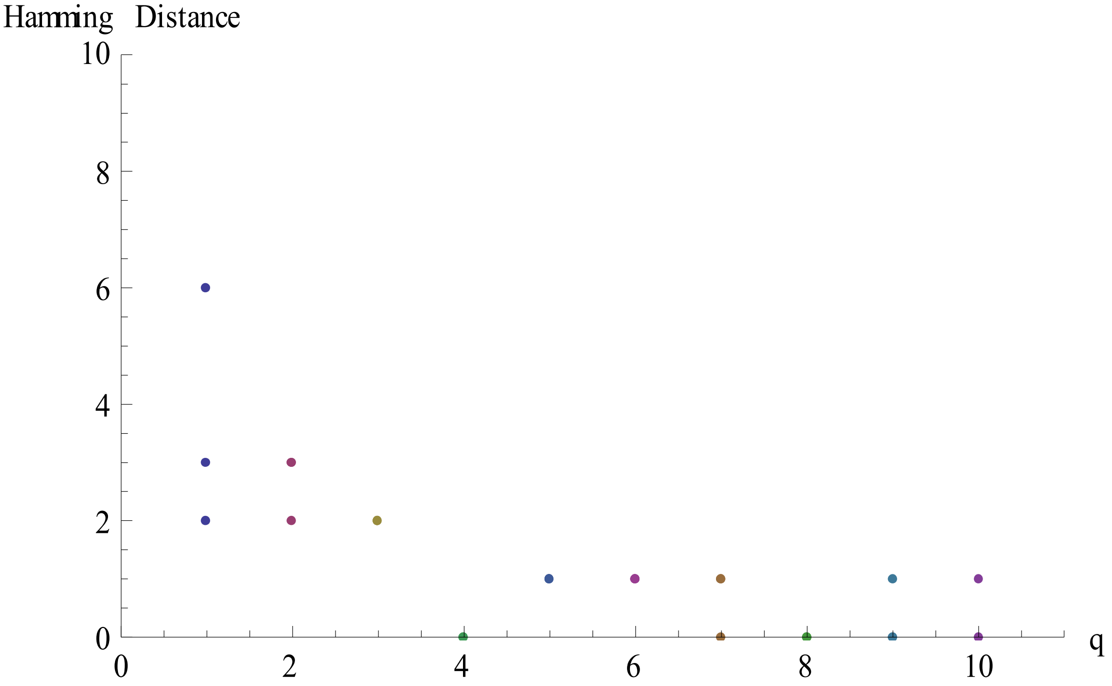


Fig. 4.26 – Plot of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, First Iteration

As can be seen here, while the Hamming Distance between the output vector and the target vector does decrease with increasing ‘q’, in some cases it remains greater than zero – even when the clue is equal to the length of the target vector itself.

A mapping of the functional relationship, then, between:

Clue length (q)

Number of vectors (v)

Dimension of vectors (d)

Number of iterations (I)

is the objective of this section. This will give a foundation for Chapters 6 and 7, where this technique is used at length.

Setting the number of iterations to two involves feeding the output from the above operation back into the memory matrix and checking for convergence, again after allowing the output to be acted upon by the saturating function.

Performing this operation gives the following results:

I=2	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
HD(O_1,v_1)	2	2	2	0	1	1	1	0	0	0
HD(O_2,v_2)	8	1	1	0	1	1	0	0	0	0
HD(O_3,v_3)	1	1	1	1	1	1	1	0	1	1

Fig. 4.27 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Second Iteration

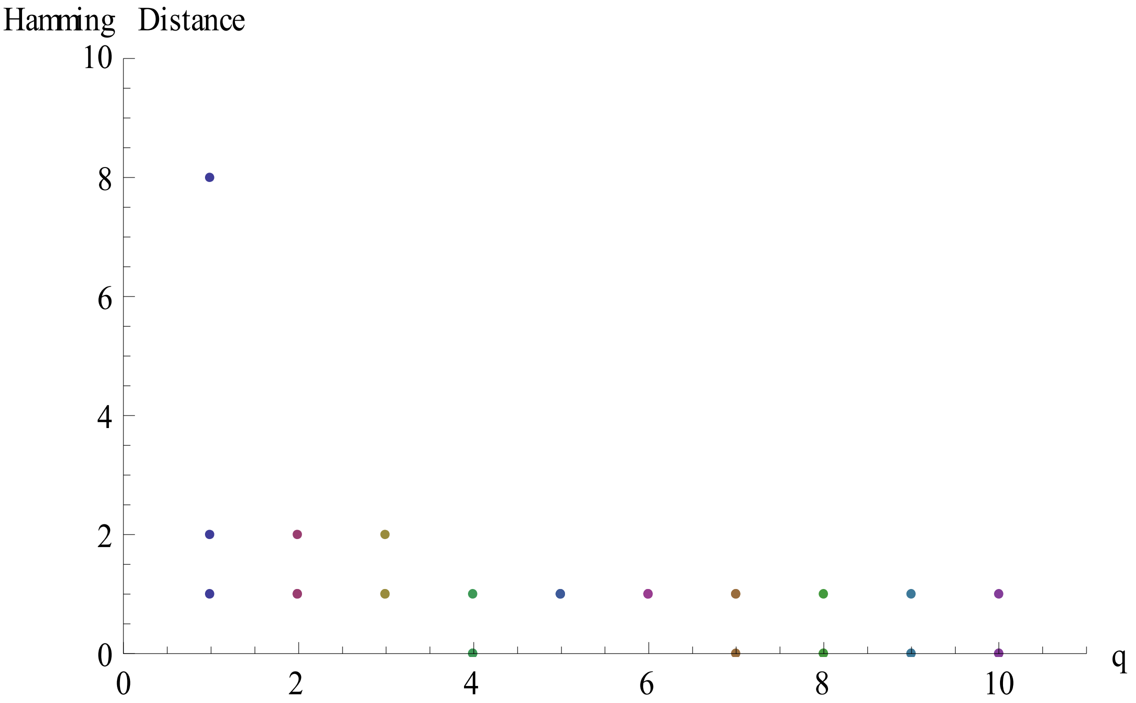


Fig. 4.28 – Plot of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Second Iteration

It is evident that the Hamming Distance, overall, has dropped (especially for low values of q). Since convergence for all values was not achieved, even for perfect input vectors, it is possible that no amount of iterations will result in complete convergence for all values.

Repeating this process with I=3 yields:

I=3	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
HD(O_1,v_1)	2	2	2	0	1	1	1	0	0	0

$HD(O_2,v_2)$	8	1	1	0	1	1	0	0	0	0
$HD(O_3,v_3)$	1	1	1	1	1	1	1	1	1	1

Fig. 4.29 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Third Iteration

This shows that there is no further convergence. A few points of interest:

1) So far, even with $n < d$, convergence is not assured, even if $q=d$

2) There reaches a point at which the system will not improve convergence, even as the number of iterations (I) increases

3) The T_{ii} condition causes some elements to be removed. It may be useful to repeat this procedure, this time with T_{ii} not equal to zero.

Repeating Convergence Assessment without T_{ii} = 0 Condition

The convergence rates shown above were established with the condition that the diagonal elements of the memory matrix (T) be set to zero. It may be illustrative to see if convergence occurs when T_{ii} is not zero.

I=1

I=1 (No T_{ii} Condition)	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
$HD(O^1,v^1)$	1	3	0	0	0	0	0	0	0	0
$HD(O^2,v^2)$	5	0	0	0	0	0	0	0	0	0
$HD(O^3,v^3)$	2	3	0	0	0	0	0	0	0	0

Fig. 4.30 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, First Iteration, T_{ii} Not Equal to Zero

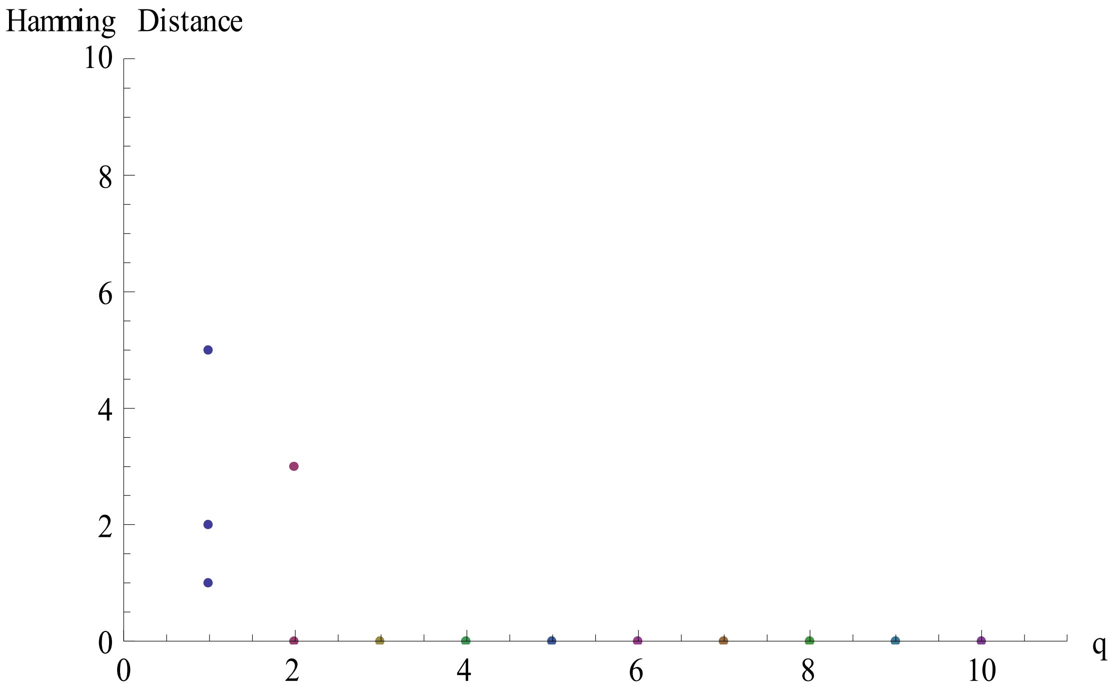


Fig. 4.31 – Plot of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, First Iteration, T_{ii} Not Equal to Zero

The above graph clearly shows a faster convergence when the condition that $T_{ii} = 0$ is removed, in the case of I=1, n=3, d=10.

Repeating this for the I=2 case:

I=2 (No $T_{ii} = 0$ Condition)	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
$HD(O^1,v^1)$	0	2	0	0	0	0	0	0	0	0
$HD(O^2,v^2)$	6	0	0	0	0	0	0	0	0	0
$HD(O^3,v^3)$	3	1	0	0	0	0	0	0	0	0

Fig. 4.32 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Second Iteration, T_{ii} Not Equal to Zero

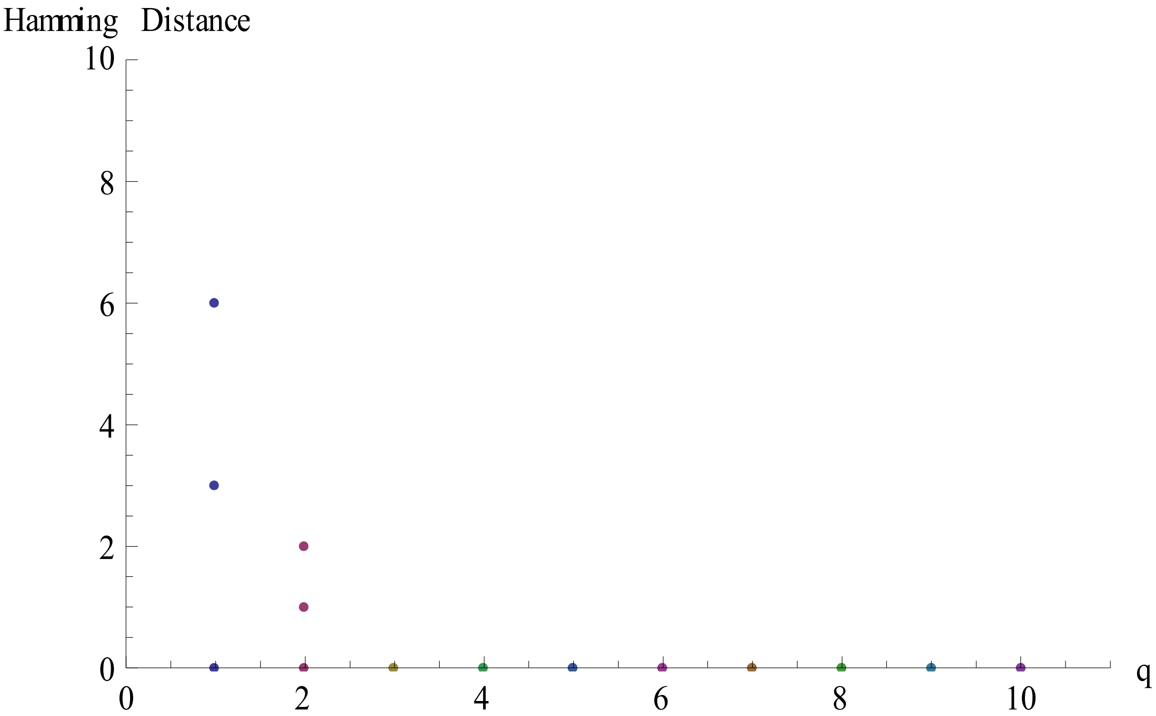


Fig. 4.33 – Plot of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Second Iteration, T_{ii} Not Equal to Zero

\

Repeating for I=3:

I=3 (No T_{ii} =0 Condition)	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8	q=9	q=10
HD(O^1,v^1)	0	2	0	0	0	0	0	0	0	0
HD(O^2,v^2)	6	0	0	0	0	0	0	0	0	0
HD(O^3,v^3)	3	1	0	0	0	0	0	0	0	0

Fig. 4.34 – Table of Hamming Distances for Output Vectors and Original Vectors for all Clue Lengths, Third Iteration, T_{ii} Not Equal to Zero

Again showing a stability identical to I=2.

This illustrates an intriguing possibility. It is possible that convergence is considerably more likely in the case where T_{ii} is not equal to zero, instead of where T_{ii} is equal to zero. This criterion, and several others, will be tested in Chapter 6.

Chapter 5 – DNA based computation

5a. The Hamiltonian Path and SAT Problems

In 1994 Leonard Adleman introduced the idea of DNA computation[64]. As mentioned in Chapter 1, computation can be effectively performed with any working substance, so long as the inputs and outputs are meaningful. In Adleman's original work, the Hamiltonian path problem was solved using selective filtration of DNA molecules.

To get an idea of the novelty involved in Adleman's publication, it is first necessary to understand the Hamiltonian path problem (aka the "Travelling Salesman" Problem). In this problem, a salesman needs to visit a number of cities on a map, all connected by roads. The question is "What is the optimal route to take, such that each city is only visited once?". In terms of the Hamilton Path problem, the cities are represented by nodes in a graph, and the highways by edges. The reformulation then involves constructing a directed path which intersects each node in the graph once and only once. The graph is said to contain a Hamiltonian path if it meets this condition.[65]

It is helpful at this juncture for a brief digression into computational complexity. Consider the Travelling Salesman problem. If asked to calculate a route that goes through 'n' cities, there are (n-1)! possible routes. For a small set of cities, for example 10, this is a reasonable number of itineraries to check (9! = 362,880). However, if the number of cities is increased, the problem quickly gets out of hand. For example, if there are 40 cities to visit, 39! itineraries are possible. This is approximately 10^{45}, which is well out of range for a computer to check even in the lifetime of the Universe. This problem, then, is intractable. How to determine which problems are tractable and which ones aren't is the subject of computational complexity. Computer scientists have formulated different classes for problems to determine their tractability:

Polynomial (also called Class 'P') – These are problems where the number of steps required to solve the problem is a polynomial of the length of the input. For example, if it takes twice as many steps to solve a problem every time the input grows by one, the complexity of this problem would be y=2x (where 'x' is the length of the input). This is clearly a polynomial time problem.

Nondeterministic Polynomial (also called Class 'NP') – This class of problems cannot be solved in polynomial time on a deterministic Turing machine. However, they can actually be solved in polynomial time on a nondeterministic Turing machine.

Nondeterministic Polynomial Complete (also called Class 'NP Complete') – This is the most difficult class of problems. An NP Complete problem is one in which no polynomial time algorithm is thought to exist, either for deterministic or nondeterministic Turing machines.

It can be shown that the traveling salesman problem is NP complete, meaning that there should exist no solution for the problem which can be solved in polynomial time[66]

However, Adleman proposed a method of solution which was easily scalable. In his representation, he first came up with a method of sifting solutions, so that only those meeting the criteria to be a Hamiltonian path were left after the algorithm was completed. These steps were[64]:

1) Generate random paths through the graph

2) Keep only those paths that begin with the desired starting and ending nodes

3) If the graph has "n" nodes, keep only those solutions which enter only "n" nodes

4) Keep only those solutions that enter all of the nodes of the graph at least once

5) If any paths remain, say "Yes". Otherwise, say "No".

To see the elegance of Adleman's solution, inspect the graph shown below:

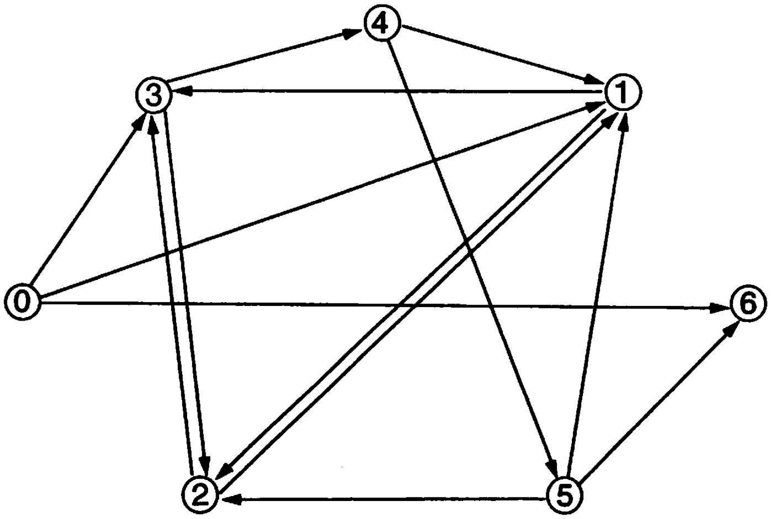


Fig. 5.1 – Hamiltonian Path Diagram

This graph only has 7 nodes, along with the given allowed paths for this graph. The implementation of these steps in DNA, as described by Adleman, were:

1) Generate random paths through the graph

To accomplish this, Adleman first chose to represent each of the nodes in the graph by a randomly generated 20mer (O_i). Nodes 2, 3, and 4 are shown below. Each directed edge (0 → 1, 1 → 2, etc) was represented by a 20mer which had 10 bases from the 3' end of O_i connected to 10 bases from the 5' of O_j. In cases where i->j had i=0 or j=6, all i, or j respectively, was used. The paths between node pairs (2,3) and (3,4) are shown below. Combining all of these components together in solution makes it possible to encode all possible paths as strings of DNA.

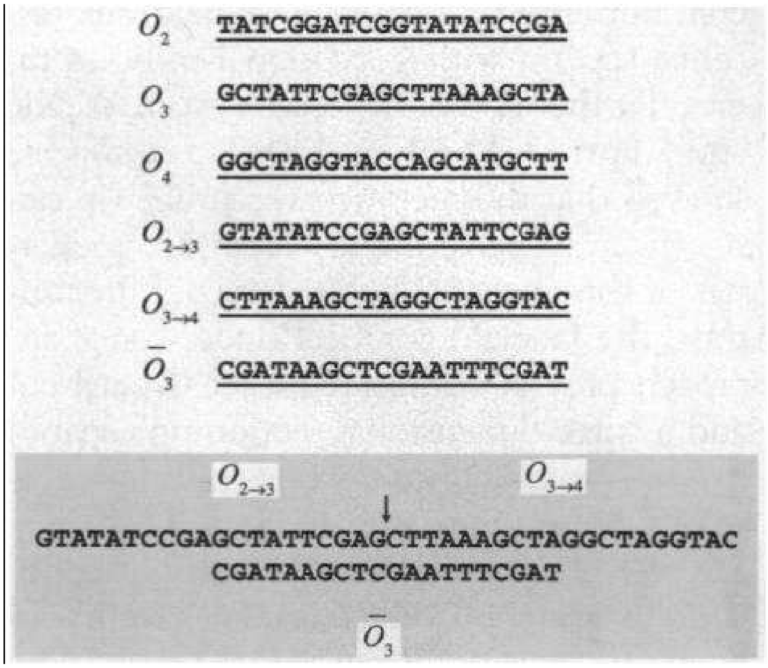

O_2 TATCGGATCGGTATATCCGA
O_3 GCTATTCGAGCTTAAAGCTA
O_4 GGCTAGGTACCAGCATGCTT
$O_{2\to3}$ GTATATCCGAGCTATTCGAG
$O_{3\to4}$ CTTAAAGCTAGGCTAGGTAC
$\bar{O}_3$ CGATAAGCTCGAATTTCGAT
$O_{2\to3}$
$O_{3\to4}$
GTATATCCGAGCTATTCGAGCTTAAAGCTAGGCTAGGTAC
CGATAAGCTCGAATTTCGAT
$\bar{O}_3$

Fig. 5.2 – Nodes and Paths in DNA

Similar encoding is performed for all nodes and paths in the graph that is to be traveled.

Once the oligomers have linked, representing all possible paths through the graph, heating the mixture to a suitable temperature results in the denaturing of the "bridge" pieces. The long oligomers left over now give the desired complete set of all possible random paths.

2) Keep only those paths that begin with the desired starting and ending nodes

This step is rather straightforward. Since the oligomer sequence for the desired start and end nodes is known, to filter out all sequences not having those start and end nodes, Adleman performed a polymerase chain reaction on the oligomer set, using the complementary oligomers to the start and end nodes as primers. This amplified everything meeting the criterion of desired start and end nodes, and had no effect on other sequences.

3) If the graph has “n” nodes, keep only those solutions which enter only “n” nodes

To implement this step, Adleman realized that solutions having “n” nodes would be 20*n bases long. All other oligomers that didn’t meet the “’n’ nodes criterion” would be longer or shorter. Adleman performed agarose gel electrophoresis, which separated the oligomers by length. He then proceeded to extract from the gel the oligomers of the appropriate length. This extracted product now represents solutions which have only the correct number of nodes, and still have the desired starting and ending nodes.

4) Keep only those solutions that enter all of the nodes of the graph at least once

The penultimate step in Adleman’s recipe required the oligomers to be tagged with magnetic beads. Accomplishing this involved first denaturing the DNA, then incubating O_i bar with magnetic beads attached with the oligomers. Only oligomers with O_i in them would be tagged. This process was then repeated with O_j and O_k.

5) If any paths remain, say “Yes”. Otherwise, say “No”.

The final step, reading out a “Yes” or “No” answer, involved another PCR and gel electrophoresis. Since Adleman’s original work, many different models for DNA computing have been devised, along with solutions for previously open problems. Here I delineate relevant properties of DNA computation, some interesting ideas for implementing schemes to make use of these properties, and some practical limitations one might find.

The first property to note is that a DNA environment can be used not only to solve the Travelling Salesman problem, but any problem which is NP complete. Lipton[67] was able to demonstrate this on what is referred to as the SAT (or "satisfaction") problem. The SAT problem is both simple to understand and NP-complete. In this problem, consider the formula:

$$F = (x \vee y) \wedge (\bar{x} \vee \bar{y})$$

First, assume that both x and y are Boolean (quite frequently it is convenient to regard 1 as "true" and 0 as "false"). In the above equation, "$\vee$" is the logical operator representing "OR', and "$\wedge$" is the logical operator representing "AND". The SAT problem then is to find the values for "x" and "y" that make the formula F true. For the above equation, x = 0 and y=1 is a valid solution, as is x=1 and y=0. The generalized form of the above function (F) is C_1 AND C_2 AND C_3 AND.. C_n, where each C_i (i = 1..n) is a clause of the form v_1 OR v_2 OR v_3.. OR v_n. In this formula, v_i is a variable or its negation. The generalized SAT problem, then, is to find values for x and y which make the whole function (F) true. In terms of current algorithms, the best method essentially tries all 2^n choices for the n variables in the equation.

Lipton began solving this problem in DNA by encoding binary numbers into DNA. To do this, he first represented each number by a graph, and each possible path through the graph by an oligomer. The similarity to Adleman's original approach is noticed! In this case, however, the graph would proceed from an initial vertex, a_1. It would then diverge with edges leading to two nodes, x and x'. Each of these nodes would then have an edge coming together at a node a_2. The process would then repeat, generating the graph shown below:

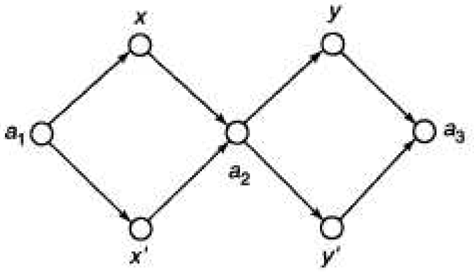


Fig. 5.3 – Graph for Representing Numbers

Representation of a binary number is then accomplished by observing which path is taken through the graph. If the edge between a_1 and x is taken, a "1" is inferred. If the edge between a_1 and x' is taken, a "0" is inferred. So, for example, the number 00 is represented by the path a_1 -> x' -> a_2 -> y' -> a_3. To translate this into DNA, Lipton assigned each vertex a random sequence of the form $a_i = p_iq_i$, for all a, x, y, etc. He then generated all sequences of the form q_i(bar)p_j(bar), for all i and j. Finally, he added two more sequences to the tube: one which is complementary to the first half of the initial vertex and one which is complementary to the last half of the final vertex. Combining all of these pieces in a tube and allowing them to anneal generated an alphabet of all possible binary numbers that are 'n' digits long.

Once all possible binary numbers were encoded into a tube (t_o), it was possible to find only those pieces which were solutions to the problem through the appropriate combination of extraction and recombination of relevant elements between tubes. For example, to find the solution to the SAT problem mentioned above, an experimenter would have combinations in the initial tube:

2 Bit Number	Graph Representation	Oligomer Representation
00	$a_1 \rightarrow x' \rightarrow a_2 \rightarrow y' \rightarrow a_3$	$p_1q_1p_{x'}q_{x'}p_2q_2p_{y'}q_{y'}p_3q_3$
01	$a_1 \rightarrow x' \rightarrow a_2 \rightarrow y \rightarrow a_3$	$p_1q_1p_{x'}q_{x'}p_2q_2p_yq_yp_3q_3$
10	$a_1 \rightarrow x \rightarrow a_2 \rightarrow y' \rightarrow a_3$	$p_1q_1p_xq_xp_2q_2p_{y'}q_{y'}p_3q_3$
11	$a_1 \rightarrow x \rightarrow a_2 \rightarrow y \rightarrow a_3$	$p_1q_1p_xq_xp_2q_2p_yq_yp_3q_3$

Fig. 5.4 – Combinations of DNA for the SAT Problem

From this tube (t_o), it would be possible to extract only those sequences which have a "1" in the first bit of the sequence (denoted $E(t_0,1,1)$). Call this tube t_1. The remainder, $t_{1'}$, is then $E(t_0,1,0)$. From $t_{1'}$, extract those elements having a 1 in the second bit of the sequence ($E(t_{1'}, 2, 1)$) and label it t_2. Combine t_2 and t_1 into another tube and label it t_3 (which is actually $E(t_0,1,1)$ and $E(t_{1'},2,1)$). Next create $t_4 = E(t_3,1,0)$ and $t_{4'}=E(t_3,1,1)$. From t_4, create $t_5 = E(t_{4'},2,0)$. Finally, combine t_4 and t_5 into a tube to create t_6, which contains the solutions to the problem. The underlying thinking for this solution can be recognized by realizing that t_3 consists of all those sequences that satisfy the first clause: 01, 10, 11. t_6 consists of all of the solutions that also satisfy the second clause: 01, 10. These are the correct answers to the original problem.

Generalizing this procedure to solve a SAT problem with more clauses is straightforward. And with the ability to solve this generalization comes to ability to solve most examples of NP problems. This extension is accomplished by considering problems that correspond to any Boolean formula; they can consist of variables along with the operators of negation, OR, and AND. As Lipton concludes "This SAT problem for formulas can be solved in a number of DNA experiments that are linear in the size of the formula."

5b. Computing with DNA Tiles

A model designed and constructed by Winfree et al.[68] demonstrates the possibility of using a set of specially designed tiles to effect DNA computation. Winfree points out that a specific type of tiles – Wang tiles – can be designed so that they mimic the operation of a Turing machine.[69] A Wang tile is a type of tile in which the tile has two edges with coloration. One of these tiles can be placed next to another only if their edges are identically colored where they touch. A simple of example of this is shown here:

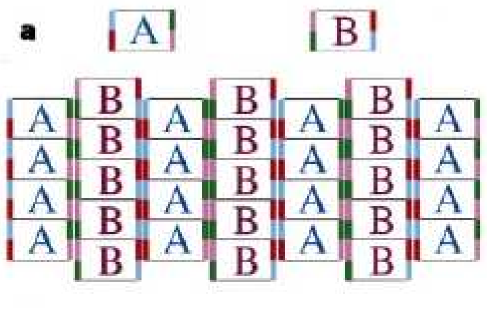


Fig. 5.5 – Wang Tiles

The tiles ‘A’ and ‘B’ can only be lined up to form an alternating pattern of columns; there is no way to set one column of ‘A’ tiles next to another. In addition, the columns of ‘A’ tiles and ‘B’ tiles are “out of phase”, meaning that each ‘B’ is not immediately adjacent to an ‘A’, but rather offset from it by half of a tile height. Given more colors with which to work, more complex tile patterns are possible. Consider, for example, the following:

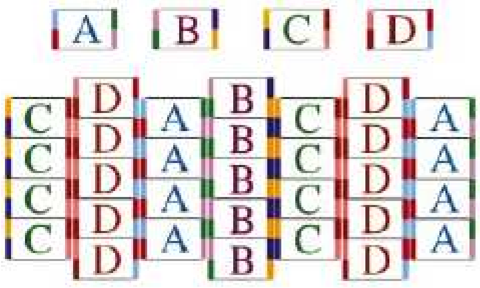


Fig. 5.6 – More Wang Tiles

In the above set of tiles, adding more colors (and consequently different tile types) has resulted in lattice with an increased periodicity.

Winfree et al. used the antiparallel DX motif to construct nanocrystals meeting the requisite conditions for Wang tiles. A DX motif is a form of DNA with a double-crossover[68]. This consists to two double-stranded helices, side by side, which are linked at two crossover junctions. Only two DX motifs are stable in small molecules: the DAO (double crossover, antiparallel, odd spacing) and DAE (double crossover, antiparallel, even spacing). A diagram of these two structures, along with the other three unstable forms (DPE, DPOW, DPON) are shown here[70]:

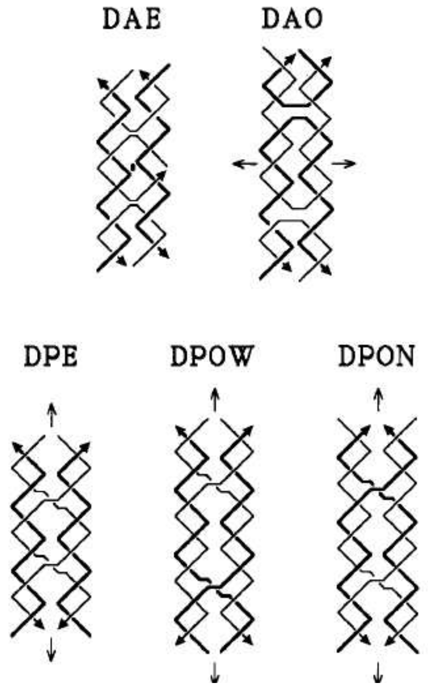


Fig 5.7 – Forms of Double Crossover DNA Molecules

While both DAO and DAE molecules exist as B-forms of the DNA double helix, there are differences between them. DAO have an odd number of half turns between crossover points. These molecules have four strands of DNA, all of which participate in both of the helices in the crystal. DAE molecules have an even number of half turns. They have five strands instead of four. However, only three of these strands participate in both helices, and two strands that do not.

All DX units have a single-stranded sticky end. Since this end is unique, association of DX units can be accomplished by designing ends that will hybridize with one another. To ensure that mishybridizations between strands is unlikely, Winfree designed sequences that had no unnecessary 6-base subsequences complementary to other 6-bases subsequences. Further, the occasional 5-base subsequence with complementarity was rare.

5c. The Maximal Clique Problem

In computational complexity theory, the Maximal Clique Problem[71] is the question of how to find the largest clique in a given graph. To illustrate this problem, consider the diagram below:

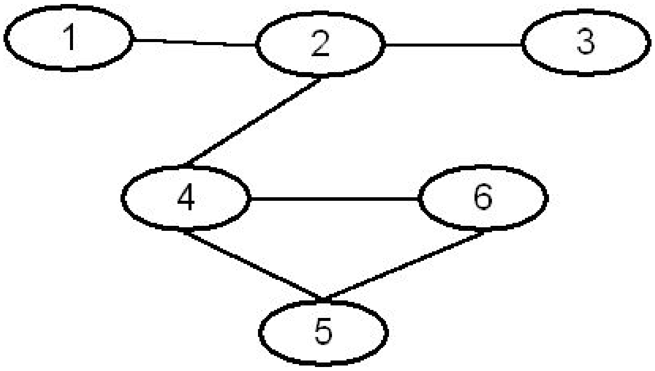


Fig. 5.8 – Example of a Clique

In the above figure, the vertices 4, 5, and 6 are connected into a "clique", meaning a subgraph in which each vertex is connected to each other vertex in that group. The Maximal Clique Problem, then, is to find the largest such subgraph in a given set of N vertices. This problem has been proven to be NP-complete[72]. However, Ouyang et al[73] have shown a solution to this problem using DNA as a computing material.

Ouyang's method involves first constructing what he refers to as a "data pool". This pool consists of all possible combinations of possible vertices and their edges. Ouyang began by first representing every possible clique arrangement for an N digit graph with an N digit binary number. In this scheme, a vertex which is a member of a clique will have a value of 1, whereas a vertex which is not a member of a clique will have a value of zero. The binary number is read from right to left. So in the figure above, where vertices 4, 5, and 6 are a clique, the binary representation would be 111000. To translate this set of all possible cliques into DNA, Ouyang generated a set of oligomers with V_{ij} representing a vertex, with j = 0,1. The oligomers, however, did not only have value information (V).

They also had position information for the strand. So the complete set of generated oligomers had:

$P_iV_iP_{i+1}$ (for i = even)

$P_{i+1}V_iP_i$ (bar) (for i=odd)

This gives the ability, then, to generate long oligomers with all possible combinations of V_{ij}. For example, one possible generated strand would be:

$P_6V_5P_5V_4P_4V_3P_3V_2P_2V_1P_1$

This oligomer represents 000000, which would be a set of vertices with no connections. In contrast, the oligomer:

$P_6P_5P_4P_3P_2P_1$

would represent the binary number 111111, which is a set of six vertices, with every vertex connected to all of the others.

Ouyang's system represented a V_{i0} with an oligomer consisting of 10 bases, and V_{i1} with an oligomer consisting of 0 bases. Each position segment was 20 bases long. This means, then, that a segment like 000000 had 60 bases for the set of V_i, along with 140 more for position fragments, for a total of 200 bases in the oligomer. A segment like

111111 contained no bases for the set of V_i, and 140 bases serving as position fragments, giving an oligomer with a total of 140 bases.

With the set of all possible cliques represented in DNA, it was necessary to filter out solutions that were undesirable. To do this, Ouyang first constructed the "complementary graph" to the one in question. This graph consists of a vertices that do not have connections between them. For the graph presented in Figure 4.7, the complementary graph would be:

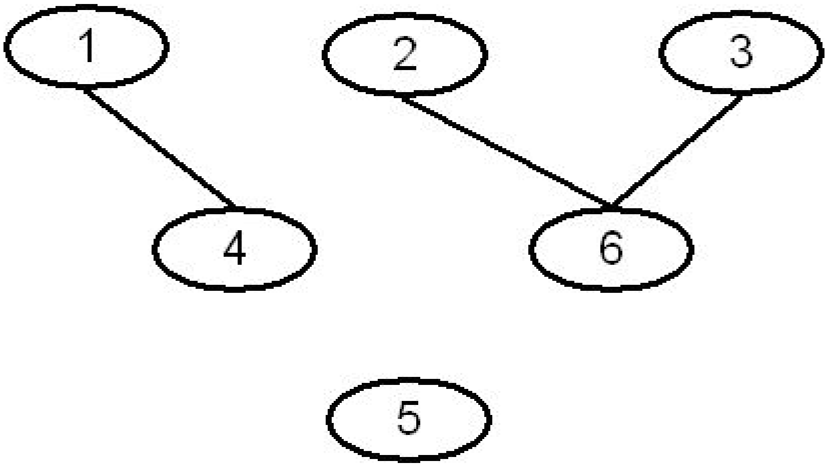


Fig. 5.9 – Complementary Clique Graph

This graph shows which vertices are not connected to each other, and are therefore not both members of a clique. For example, since there is a connection between vertices 1 and 4 in the complementary graph above, there can be no clique from the original graph which contains both 1 and 4.

In order to remove to strands which contained a connection between nodes 1 and 4, the solution was first separated into two tubes, denoted “t_0” and “t_1”. The DNA in t_0 was then acted upon with a restriction enzyme set to cut the DNA at a specific site if $V_1 = 1$. The DNA in t_1 was acted upon similarly, with a cut occurring if $V_4 = 1$. The remaining DNA from these tubes was then recombined, leaving a solution free of the connection between nodes 1 and 4.

For the above graph, the same process would be repeated for node connections between 2 and 6; as well as 3 and 6. From this point, the tube is free of problem solutions which are invalid. Reading the length of the maximal clique is straightforward, since in this scheme the maximal clique corresponds to the minimum length DNA fragment remaining. Performing a polyacrylamide gel electrophoresis easily demonstrated that the shortest DNA fragment corresponded to a clique length of four vertices.

Ouyang pushed the solution to this problem a step further by showing it was possible to not only determine the length of the maximal clique, but to know conclusively which nodes were the components of this clique. He accomplished this by amplifying the DNA solution and sequencing it. Errors in Ouyang’s method could include the production of ssDNA during PCR reactions (which cannot be cut by the restriction enzymes used), and incomplete cutting by restriction enzymes, which could lead to incorrect answers.

The first of these problems was solved by digesting the DNA solution with S_1 nuclease prior to restriction digestions, thereby eliminating the ssDNA. With regard to the second problem, Ouyang hypothesized that repeating the digestion step followed by the PCR step should increase the signal-to-noise ratio, since the PCR step amplifies the number of copies that were mistakenly uncut in the prior step. Cutting the solution again should help eliminate the unwanted strands.

The novel approaches above to solving problems, each being interesting in their own right, still display some common elements in their implementation. Each of these solutions requires encoding candidate solutions in DNA, then somehow filtering out the solutions which are incorrect. A few potential problems with this methodology have been pointed out.

As described by Boneh et al[74], there are several different types of molecular computing schemes that are possible. As Boneh notes, most results are of the following form "Given enough strands of DNA and certain biological operations, one can simulate some classic model of computation efficiently. Some compare to formulas, some to circuits, others to 1-tape nondeterministic Turing machines." A lucid summary from Boneh detailing both the complexity of the implementation in Big O notation, as well as the number of strands required, is shown below:

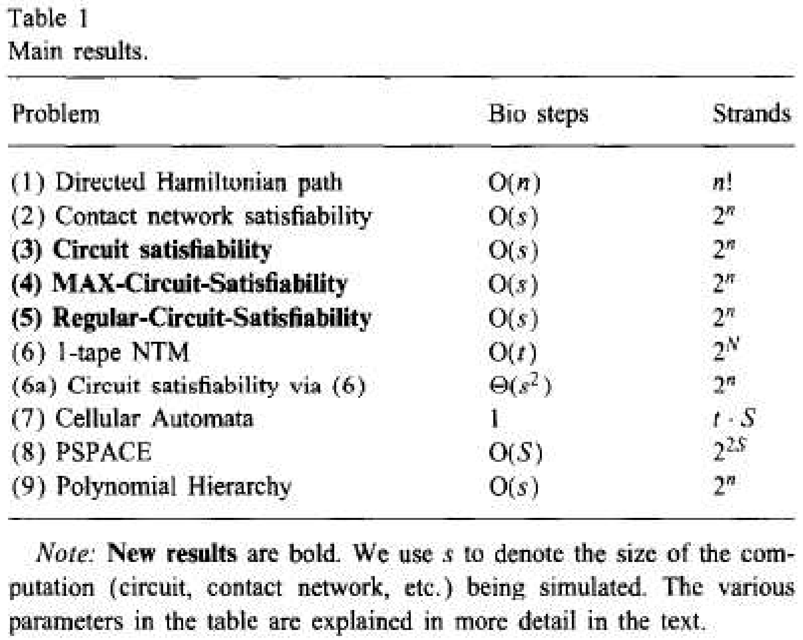


Table 1
Main results.

Problem	Bio steps	Strands
(1) Directed Hamiltonian path	$O(n)$	$n!$
(2) Contact network satisfiability	$O(s)$	2^n
(3) Circuit satisfiability	$O(s)$	2^n
(4) MAX-Circuit-Satisfiability	$O(s)$	2^n
(5) Regular-Circuit-Satisfiability	$O(s)$	2^n
(6) 1-tape NTM	$O(t)$	2^N
(6a) Circuit satisfiability via (6)	$\Theta(s^2)$	2^n
(7) Cellular Automata	1	$t \cdot S$
(8) PSPACE	$O(S)$	2^{2S}
(9) Polynomial Hierarchy	$O(s)$	2^n

Note: **New results** are bold. We use s to denote the size of the computation (circuit, contact network, etc.) being simulated. The various parameters in the table are explained in more detail in the text.

Fig. 5.10 – Table of Problems Solvable in DNA

However, Boneh also notes in his paper regarding the above methods "…we assume that the biological operations are perfect.. we note that several researchers have already begun to take steps to make DNA algorithms more noise tolerant." It would turn out that this noise tolerant condition would be requisite for successful, scalable DNA computation.

5d. Requirements for Implementing DNA Computing Schemes

One potential source of error for a computation performed using DNA is the potential for mishybridization. Deaton et al[75] pointed out that hybridization chemistry could be relevant, in that prior to his publication, it was assumed that hybridization between Watson-Crick pairs was error free. This was (and is) not necessarily a valid assumption, as hybridization (and mishybridization) can be a function of a number of parameters in a given reaction. Deaton pointed out that the fraction of "G" and "C" bases in a given strand are relevant to the melting temperature for that strand, and that at an incorrect temperature, hybridization between strands could still occur, even though some of the bases in the strand were "mismatched". To help overcome this, Deaton suggested that the oligomers used in reactions be a certain Hamming Distance from each other, to help minimize undesired hybridizations.

Certain schemes of DNA computation involve solutions represented as individual strands of DNA. In this case, it is frequently possible that solutions will contain errors (due to DNA mishybridization). If the experimental procedure involves the use of polymerase chain reaction to amplify DNA, then these errors could be amplified as well, giving incorrect output. Taq Polymerase is frequently used in Polymerase Chain Reaction (PCR), and has been measured to have an error rate of approximately 1 in 10^4.[76] Consequently, it is necessary to find solutions for computational problems that are fault tolerant. That is, even in the presence of hybridization errors, the fidelity of computation can be ensured.

5e. Precursors to the Mills, Yurke & Platzmann (MYP)Model Hopfield Neural Network

In contrast to the above implementations, the scheme proposed by MYP[91] entails the construction of a neural network architecture from DNA molecules. This model retains the property of massive parallelism, while possibly being more fault tolerant than earlier proposed models. Calculations of the convergence of this form of network are measured in Chapter 7.

One of the ideas necessary for the construction of a HNN in DNA is how to store memories as oligomers. An early form of this idea was proposed by Baum[77]. His proposal entailed the encoding of a word as a set of binary values. Each site in a DNA sequence would consist of two parts – one which held the information about the position in question, and the other which held a subsequence representing either one or zero. Each word encoded in memory then would be a distinct oligomer.
Retrieval would be accomplished by the introduction of a clue vector. This clue would consist of complementary sequences to portions of the original word, each attached to a magnetic bead. Today this could be accomplished with a fluorescent tag. The correct word to be recalled, then, would be the one with the most tags attached to it after the introduction of the clue vector. This strand could then be removed from solution and sequenced, giving the original word in its entirety.

Another prerequisite for the implementation of the MYP scheme is the ability to multiply matrices together in DNA. A method for performing this operation was proposed by Oliver[78]. In this paper, Oliver first recalled how two Boolean matrices and their product could be represented by a directed graph[79,80] . Observe the two Boolean matrices and their product in part A given below:

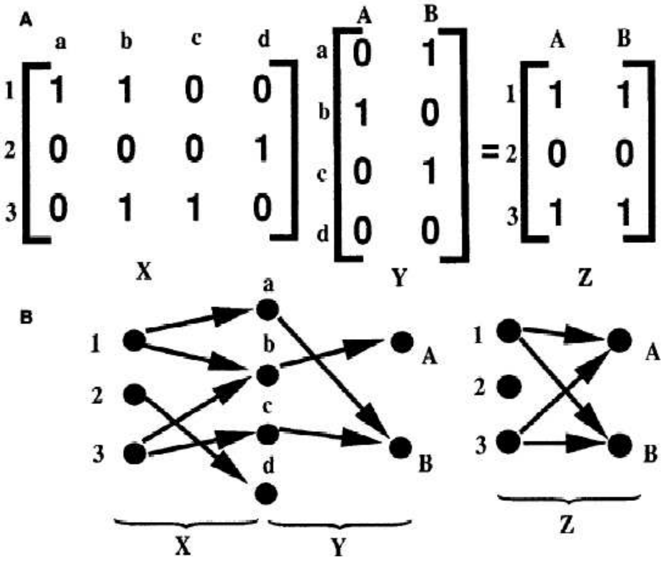


Fig. 5.11 – Boolean Matrix Algebra[78]

In Part B of the figure, one can see the representation of the matrices and their product as a graph. The first layer of nodes on the graph represents the row numbers of the first matrix. The second layer of nodes represents the column entries of the first matrix (which by necessity have the same number of entries as the rows of the second matrix). The terminal row of the graph represents both the column numbers of the second matrix and the column numbers of the product matrix. If an entry exists in a particular row and column of the first matrix (for example, row 1, column a of matrix X), then a directed edge is drawn in the graph between the vertex labeled '1' in the first layer of the graph and the vertex labeled 'a' in the second layer of the graph. In the same manner, if there exists a nonzero entry in the second matrix (for example, row a, column B of matrix Y), there will be a directed edge between vertex 'b' from the second layer of the graph to vertex ‘B’ in the third layer of the graph. This translation allows nonzero entries in the product matrix to be read off as complete paths, going from a vertex in the first layer to a vertex in the third layer. If there is not an unbroken path from the first layer in the graph to the vertex of interest in the final graph, the entry for that row and column in the product matrix is equal to zero.

To implement this idea using DNA, Oliver points out that it is first necessary to:

1) Design and synthesize DNA representing edges and vertices in the graph

2) Cause the constructed DNA to represent paths through the graph (via reactions)

3) Analyze the reaction to identify paths between initial and final vertices by restriction enzyme digestions

According to Oliver’s method, each complete path through the graph is represented by a strand of dsDNA. To perform this operation, each path from an initial node to an intermediate vertex is represented by dsDNA with a single strand hanging from the end that is specific to the intermediate vertex. For example, the connection between vertex 1 and vertex a in the graph in Fig. 5.11 might be represented by a strand of dsDNA with a single strand extending 10 bases beyond the double stranded portion. The edge connecting the intermediate layer to the final layer is then represented by a different double stranded oligomer, with a piece of ssDNA complementary to the above mentioned ssDNA hanging from the end opposite of that mentioned above for the first/intermediate layer. In this way, it is possible for reactants to connect making long strands of dsDNA by allowing them to react in solution with ligase. These strands are all the same length, and each represents a different possible complete path through the graph (and consequently a nonzero entry in the product matrix).

In the construction of the original set of oligomers, each piece of DNA representing a vertex contained in its structure a restriction site, where an enzyme is capable to cutting it. To find out which of the paths through the graph is complete, one needs to separate the mixture, with an aliquot placed in each tube in a set. The number of tubes is the same as the number of entries in the product matrix.

In this case, an oligomer representing a path from a starting vertex to a terminal vertex will have two available restriction sites. Two restriction enzymes corresponding to the row and column of the product matrix are placed into one of the tubes of solution. If there is an available path through the graph, both ends of the DNA piece of interest will be cut, and therefore be shorter than the original DNA in solution. This then corresponds to an entry of 1 in that row and column of the product matrix. If no appropriately shortened piece of DNA is found, there exists a 0 in the row and column of interest in the product matrix.

This method can be extended to matrices involving real numbers, with a few modifications. Representing matrix multiplication with real numbers in a graph is similar to what was shown previously, with the exception that now each element encoded as an edge must also have a “transmission factor”[81] The transmission factor for a path is defined to be the product for the transmission factors for each of the edges in that path. Any particular element in the product matrix is then the sum of the transmission factors for all possible paths connecting the initial and terminal vertices of interest. An example of this sort of graph is shown below:

$$\begin{bmatrix} 5 & 9 & 0 & 0 \\ 0 & 0 & 0 & 25 \\ 0 & 2 & 7 & 0 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 17 & 0 \\ 0 & 32 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 163 & 15 \\ 0 & 0 \\ 34 & 224 \end{bmatrix}$$

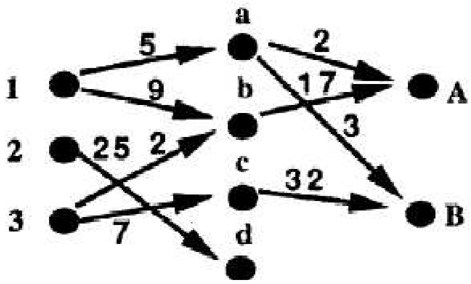


Fig. 5.12 – More Boolean Matrix Algebra

Representing this graph in DNA involves using concentrations of DNA to represent the transmission factors in the original graph. This way, the elements of the product matrix will be proportional to the sum of the concentrations of the associated paths formed. To form a path from two edges requires the annealing of complementary pieces of ssDNA on the end of dsDNA representing edges (as was performed previously for Boolean matrices).

With the ability to produce operations such as matrix multiplication, MYP recognized the ability to implement mathematical computations which relied on this kind of operation. Specifically, MYP proposed to construct a neural network using an extension of the ideas presented above. Their model will be presented in detail in Chapter 7.

Chapter 6 – Optimal Conditions for Convergence of a Hopfield Neural Network

The function of a HNN can vary. Hopfield's original intent, however, was the recall of patterns already encoded in the network from a noisy input. This application has had the most research and study as investigators attempt to both assess and improve upon Hopfield's model. Since one of the main applications of this sort of memory is recall, it may be possible to improve upon the process of retrieving a memory from an incomplete memory by measuring not only the Hamming Distance from the clue to the correct memory, but by measuring the Hamming Distance from the clue to each possible memory, and then choosing the memory with the shortest Hamming Distance (similar to Baum's method[77]).

Here I assess the effect of varying several parameters in both a duobinary HNN and corrupted input HNN, including removing the diagonal (T_{ii}=0) condition, varying the saturation function used for decision making, varying whether an asynchronous or a synchronous update scheme is used in the network, and varying when the hard limiting function is applied.

6a. Vectors Used and Measurement of Convergence

In my simulation, I used vectors having a length d=100, with n=10 vectors with random ±1 entries stored in the memory matrix as prescribed by the sum of outer products rule. For all cases of using clues as input vectors, I started with a clue of length q=1, which was the first component of the desired output vector, with the remaining entries set equal to zero. I proceeded to extend the length of the clue vector by one memory entry until the length of the clue was equal to the length of the desired output vector. This approach allowed me to observe the response of the system to every possible clue vector. For corrupted vectors, I started with a correctness factor (c) which determined the number of entries with correct values (signs not flipped) in a vector. I began with every vector entry except the first half of the vector corrupted. This simulates a completely "noisy" vector, since incorrect entries are equally mixed with correct ones. To maintain the continuity of an increasing amount of correct terms being plotted along the 'x' axis, the corrupted vector graphs are plotted as c/d along the 'x' axis, to show effects as the amount of "correct" vector is presented to the network.

The fidelity of HNN convergence was quantified by examining both the number of iterations required for complete convergence (if it is achieved at all), and the "correctness" of the output when compared to the original input vector. This comparison was accomplished by measuring the Hamming Distance (HD) between the input and output vectors. The Hamming Distance is defined to be the number of places between the two compared vectors where their entries differ. So, for example, a HD of zero implies that the input and output vectors are identical in all components, while a HD of 'd' (where 'd' is the length of the vector being examined) implies they differ in every possible entry. The HD is normalized in my comparison, so that $0 \leq HD \leq 1$.

6b. Removal of the $T_{ii} = 0$ Condition

According to Hopfield[21], the diagonal elements of the memory matrix (T_{ii}) must be set equal to zero to insure convergence of the HNN. As mentioned in Chapter 4, Gindi[49] found that for bipolar nodes and positive diagonal terms, an autoassociative network with outer product formed elements performed nearly identically to a memory matrix with zero valued diagonal terms. Gindi's simulations were performed for d=100 and n=10, and involved testing convergence for both synchronous and asynchronous update schemes, using corrupted memories as inputs. Gmitro[50] verified this result and attempted to extend it using both binary bipolar and binary unipolar models, followed by Marom[82], who suggested that removal of the T_{ii} condition lead to a modest increase in convergence, but a loss of error correction ability. Here I extend this work to assess not only the convergence results when clue vectors are used instead of corrupted vectors, but to quantify both the mean and standard deviation of the convergence as a function of clue length, to see if there is a difference between when $T_{ii} = 0$ and when $T_{ii} \neq 0$

While convergence may not be guaranteed for $T_{ii} \neq 0$, it is interesting to explore how much convergence varies for systems where, after the selection of vectors, convergence is assessed where $T_{ii} =0$ in the memory matrix, and again where $T_{ii} \neq 0$.

For this simulation, vectors (d=100, n=10) were randomly generated and tested for convergence, first with T_{ii} =0. Each vector was tested by creating a clue (q) or correctness (c) as prescribed in section 6a then allowing that clue or corrupted vector to be acted on by the memory matrix, which was then acted on by the threshold function (for this purpose a Heaviside function). This output was compared to the desired memory vector and the difference between them quantified using the Hamming distance. This was repeated for all relevant clue and correctness lengths, for every vector, until q=d, or c=d. Then the same vectors were used to create another memory matrix, this time with T_{ii} elements from the outer product left intact. This gives a robust set of results, including not just convergence for a few possible vectors in a system, but a more realistic measure of the average convergence of a set of vectors, along with the variance in that convergence. To determine if recall fidelity gives a faster convergence result than a direct comparison of the clue vector with other possible vectors, Fig 6.1 shows the Mean Hamming Distance (HD) between each clue length and the vectors with which the clue should not be equal.

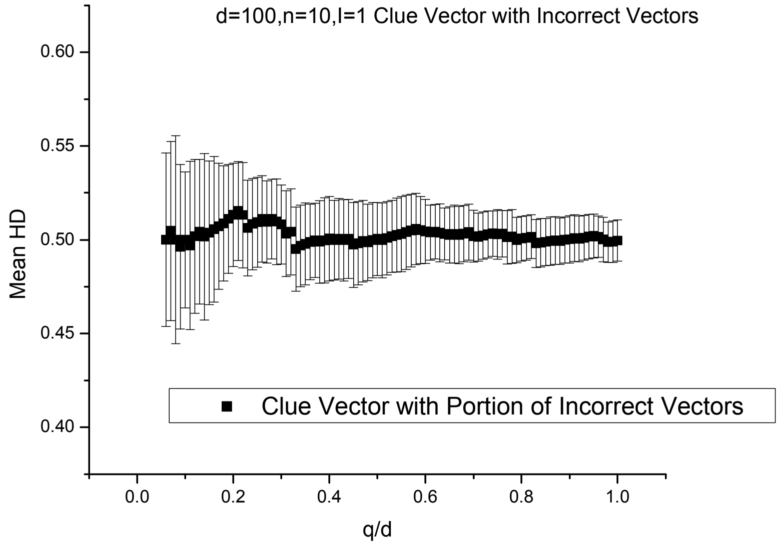
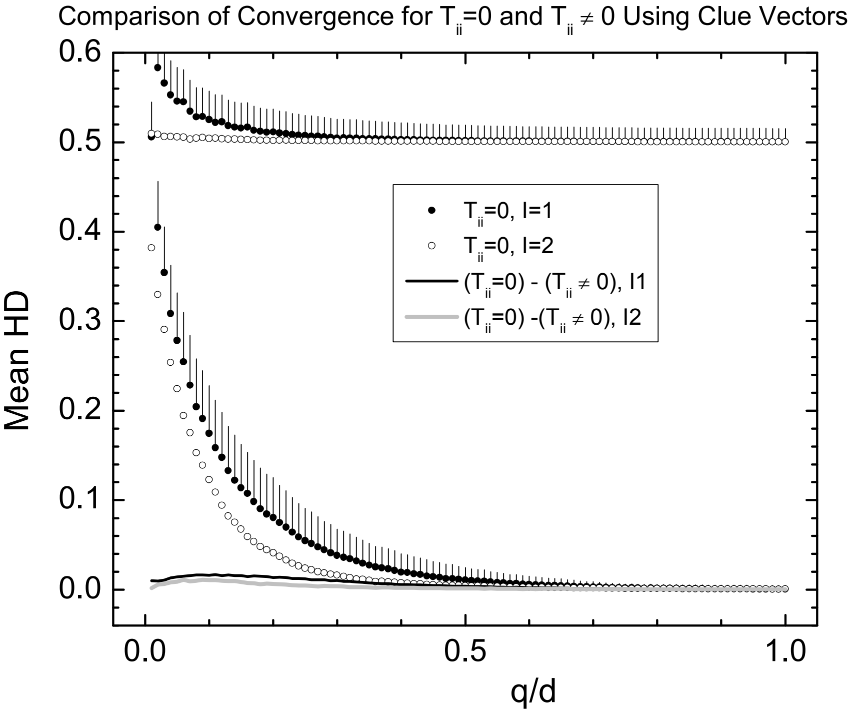


Fig. 6.1 – Comparison of normalized clue vectors with each incorrect vector

The first item to note is that the Mean Hamming Distance should be around .5, which would correspond to a clue vector of a reasonable length having half of its entries differ in sign from a selected incorrect vector to which it is compared. Further, as will be shown in the following plots, this convergence is well above the convergence for the HNN. Analysis of direct comparison methods may be realized in physical systems, such as a biological library. Such a system involves the cloning of molecular fragments through insertion into and subsequent replication of bacteria (BAC) or yeast (YAC). Retrieval of a desired genomic fragment is accomplished via the introduction of a “clue” fragment to the system, which then can be amplified using rolling circle polymerase chain reaction to select the desired result from among many possibilities in solution.

The first item tested in the condition that $T_{ii} = 0$. Results for convergence, measured as normalized Hamming Distance (HD) vs. normalized clue vector length (q/d) and normalized corrupted vector (c/d) are shown below

a)



b)

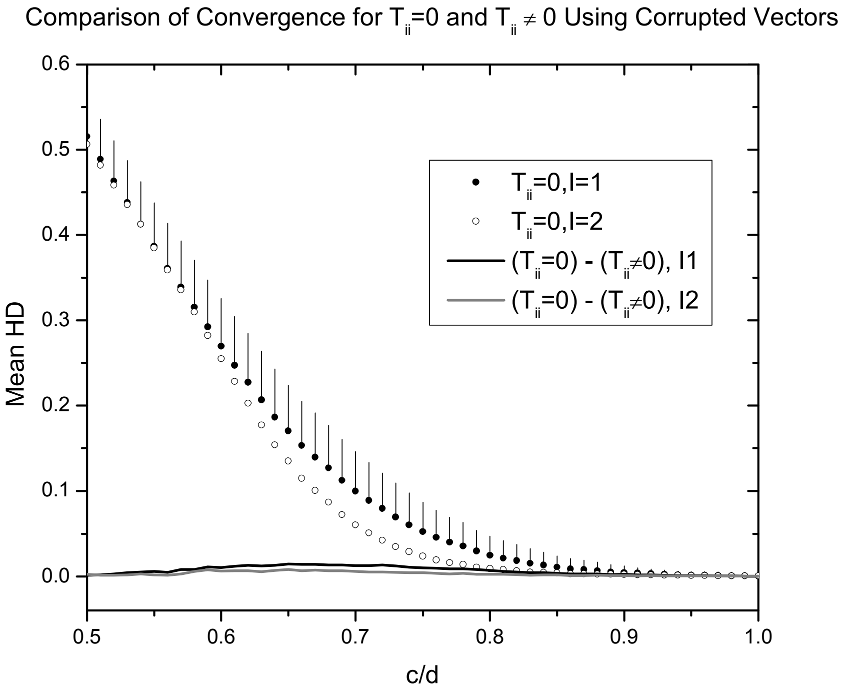


Fig. 6.2 – a) Convergence measured as Hamming Distance vs. normalized clue length for T_{ii}=0 and T_{ii} ≠0. The mean Hamming Distance between the Output Vector and undesired vectors for clue vector inputs can be seen in Fig. 1a. b) Convergence measured as Hamming Distance vs. normalized correctness length for T_{ii}=0 and T_{ii}≠0.

As can be seen from the above graphs, the mean value of convergence for when the diagonal condition is enforced or not enforced is almost identical in all cases. When the diagonal condition is relaxed, convergence is slightly improved (up to a maximum of .011 for clue vector inputs and .014 for corrupted vector inputs) for the first iteration (I=1).

The simulation was run again for I=3. No improvement in convergence was seen from I=2 to I=3. However, there is an improvement in results from I=1 to I=2 when T_{ii} is not forced to be zero. This is in agreement with the derivation of Gindi[49], who found similar results and Marom[82], who found a small improvement in convergence when the diagonal condition was relaxed.

6c. Variation of the Saturation Function for Decision Making

In Hopfield's original scheme, the saturating function given for determining the output values of a HNN was given by a "hard limiting function" (the Heaviside function), which produces an output of -1 for an input less than zero, and 1 for an output greater than zero.

Hopfield[33] was able to study the effects of graded decision functions, and found in his paper that there were normally fewer states that were stable, but that the memory overall functioned the same. When Macukow[83] took up this question briefly, he found that for some vectors attempted, a threshold condition which included zero being mapped to positive one was more useful for convergence than one in which the input had to be greater than zero to be mapped to one. However, his attempts were for a limited number of vectors, and did not address whether this rule was applicable globally.

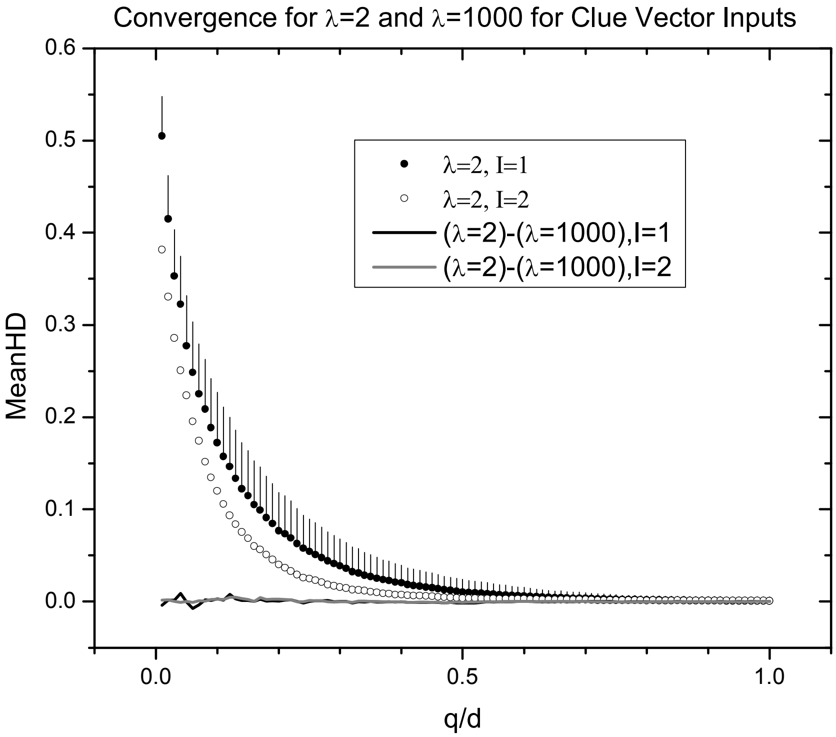
In many neural network implementations, the decision function is not constrained to be a Heaviside function, but may be something which still produces an output in a nonlinear fashion (such as a hyperbolic tangent function). In this section, I assess the convergence of a HNN as a function of the variance of this decision mapping.

To assess the efficacy of using different possible values for a limiting function, the general form

$$f(Mem.u) = \tanh\left(\frac{\lambda(Mem.u)}{2}\right)$$

was used. It can be seen that the overall "steepness" of the decision function is given by the parameter λ, with the function's steepness increasing as $\lambda \to \infty$.

Below are two plots, each showing the convergence for $\lambda=2$ and $\lambda=1000$. The first plot uses clue vectors as inputs to the memory matrix, and the second plot uses corrupted vectors as inputs to the memory matrix.


Convergence for λ=2 and λ=1000 for Clue Vector Inputs
MeanHD
0.6
0.5
0.4
0.3
0.2
0.1
0.0
q/d
0.0
0.5
1.0
● λ=2, I=1
○ λ=2, I=2
(λ=2)-(λ=1000),I=1
(λ=2)-(λ=1000),I=2

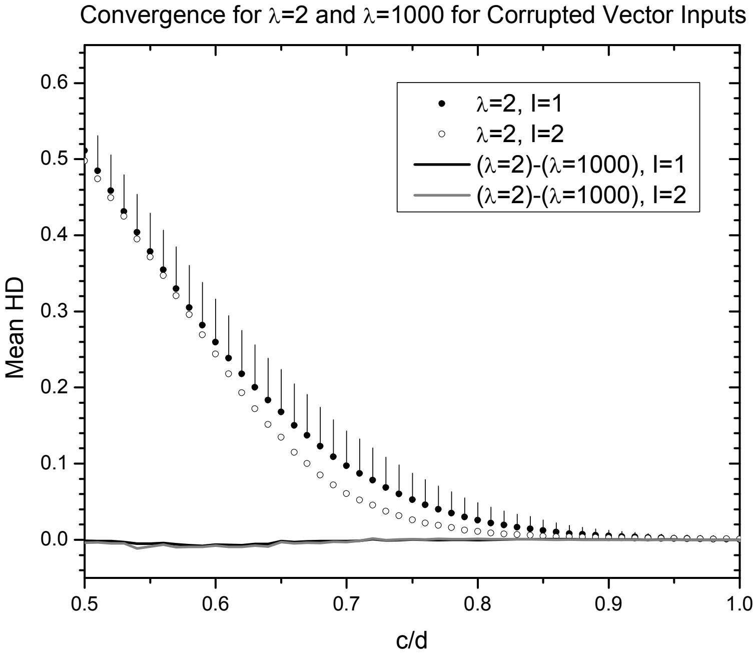


Fig 6.3 a) Mean Hamming Distance vs. Normalized Clue Length for I=1 and I=2. The difference in convergence between λ=2 and λ=1000 is also plotted for I=1 and I=2. b) Mean Hamming Distance vs. Corruption for I=1 and I=2. The difference in convergence between λ=2 and λ=1000 is also plotted for I=1 and I=2.

As the above plots demonstrate, the changes in the value of λ have no statistically significant influence on convergence rate, both when clue vectors and corrupted vectors are presented to the memory matrix. Further, while convergence is marginally improved from I=1 to I=2, and not from I=2 to I=3, (for both clue and corrupted vectors) this is expected and agrees with previous results.

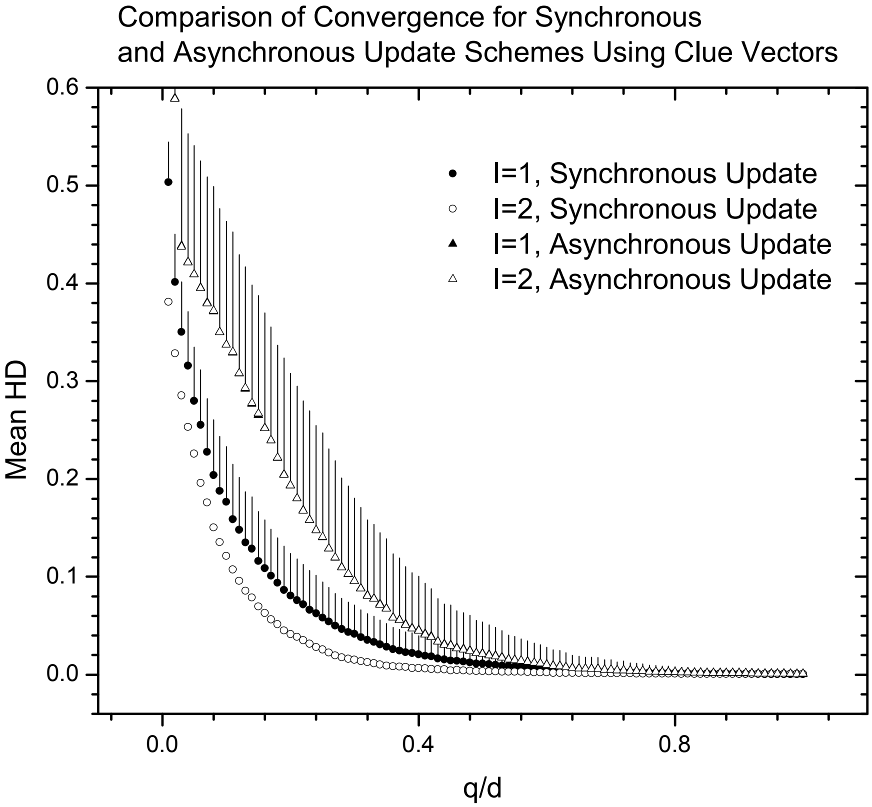
6d. Synchronous vs. Asynchronous Update Schemes for a HNN

Many NNs allow for the possibility of a synchronous update scheme. However, in Hopfield's original paper[21], the update scheme was asynchronous – only one neuron (randomly selected) updated at a time. However, much work has been done on application of updates synchronously[84],[85],[86]. Grondin[87] found that asynchronous update schemes had restrictions on the length of limit cycles not seen in synchronous networks, and that synchronous updates did not, in fact, change the nature of stable states of the system.

Here I investigate the implications of varying the update scheme on the convergence of a HNN. Cheung[88] contrasted the performance of synchronous vs. asynchronous HNNs, and found it was possible in synchronous update schemes (but not in asynchronous update schemes) to have updates resulting in positive energy changes as well as oscillations between two different energy states.

Below are plots for the normalized mean Hamming distance versus the normalized clue length and normalized corruption length for d=100, n=10 for both synchronous and asynchronous update schemes for both sets of vectors.

a)


Comparison of Convergence for Synchronous
and Asynchronous Update Schemes Using Clue Vectors
Mean HD
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.4
0.8
q/d
● I=1, Synchronous Update
○ I=2, Synchronous Update
▲ I=1, Asynchronous Update
△ I=2, Asynchronous Update

b)

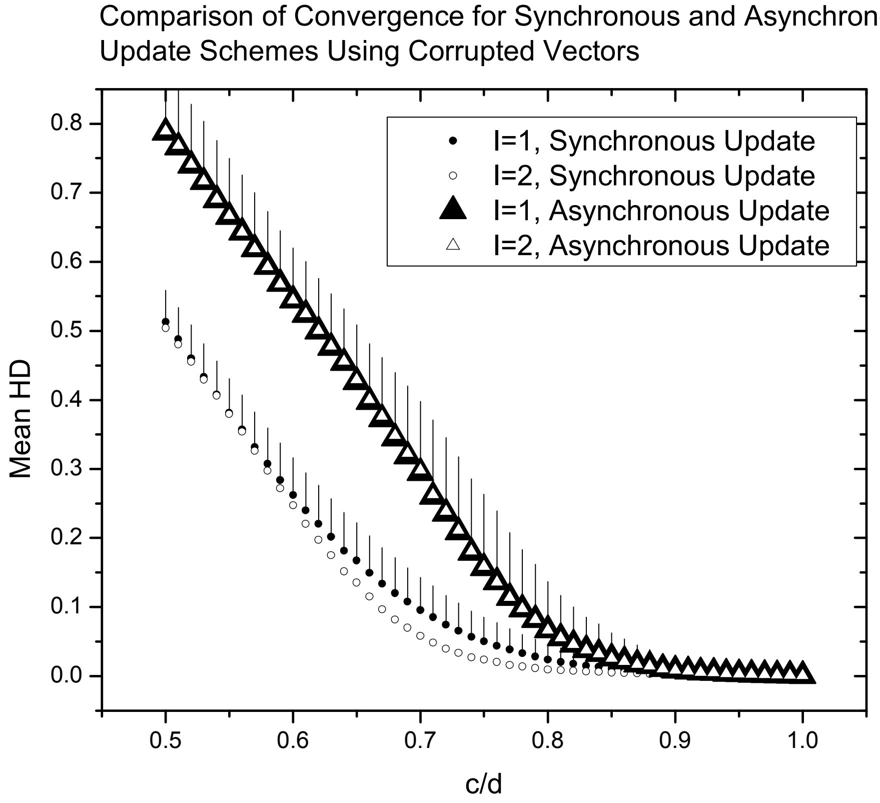


Fig. 6.4 a) Convergence for Synchronous vs. Asynchronous Update Schemes. Plots for Synchronous Update and Asynchronous Update Schemes are plotted for two iterations (I=1, I=2) for Clue Vector inputs. b) Results of both a Synchronous and Asynchronous Update Scheme for I=1 and I=2 using Corrupted Vector inputs.

In the case of clue vector inputs, complete convergence is seen in both modes of update for q/d ≈ .7. For a given clue length, a synchronous update scheme has better convergence. The deviation of values from the mean is also smaller for a system using synchronous update mode. This shows that the actual range of Hamming distances encountered for a set of vectors at a given clue length is less when the update scheme is applied synchronously.

Further, the difference between convergence values for synchronous vs. asynchronous schemes using clue vectors is most significant for values of $q< .4$. Apparently the synchronous update scheme converges faster at this range of clue lengths.

For corrupted vectors, complete convergence is not seen in either case until $c/d = .9$. As with the clue vector case, using synchronous updates instead of asynchronous ones causes an improvement in convergence.

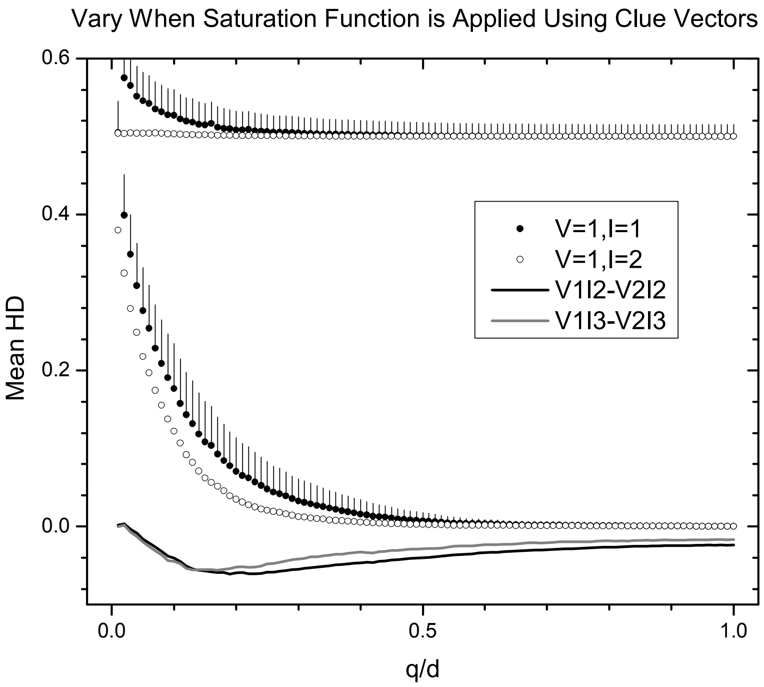
6e. Variation of when the saturating function is applied

In Hopfield's implementation, the saturating function is applied after each step. Here I alter this parameter, allowing for an update to occur at any particular step in the iterated process.

Given three iterations, several combinations of when the saturation function is applied are possible. Here I first apply the saturation function after each iteration, followed by applying the function on the second and third iterations but not the first. I conclude with a simulation applying the saturation function only after the third iteration, and not on the first or second.

6e.1 Application of Saturation Function

Below are the graphs with the saturation function applied after every iteration (V=1), with the results compared to when the saturation function is applied after the second iteration (V=2).



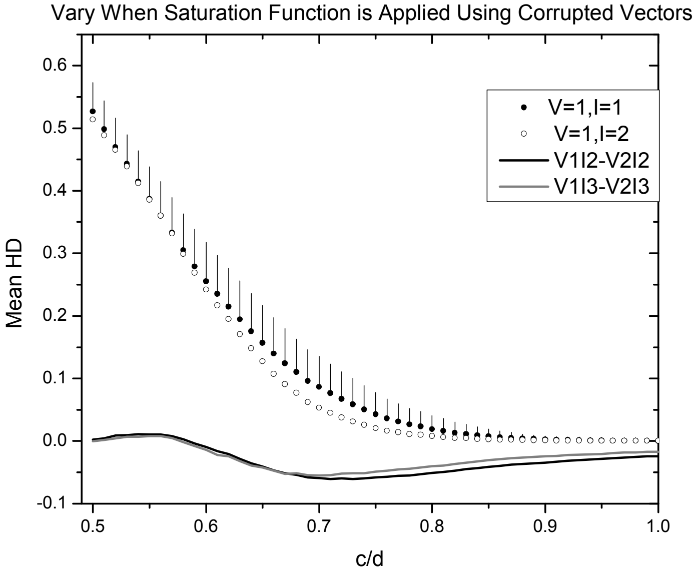


Fig. 6.5 a) Convergence dependence on when Saturation Function is applied for clue vector inputs. The Convergence when the Saturation Function is applied every iteration (V) is measured. Also plotted is the difference in convergence on the second iteration when the saturation function was applied after both iterations and after only the second iteration. Finally, the convergence for the third iteration is plotted when saturation is applied after every iteration and when it is applied only after the second iteration. b) Same information as 5a, with the exception that corrupted vectors are used as inputs.

From the above plot it is demonstrated that the convergence is slightly altered by the choice of when the saturation function is applied. The difference between V=1, I=2 and V=2, I=2 is at most -.04 for clue vectors and -.06 for corrupted vectors, which is a small effect.

The test was repeated with the choice to apply the saturation function after the third iteration. No statistically significant change in convergence was observed.

6f. Analytical Assessment of Convergence

In this section I compare my simulation of convergence with an analytical calculation by Mills[89].

For reference I record his unpublished calculations, which are a slightly improved version of a similar calculation performed by Mills, Yurke and Platzmann[91] previously, as follows: "The elements of memory in a duobinary Hopfield network are represented as m-component vectors $\vec{V} = \sum_{i=1}^{d} V_i \hat{e}_i$ in a space with basis vectors e_i (i=1, 2, …, d). The items of memory, a set of vectors $\vec{V}^{(a)}$ (with a = 1,2, …, m and each component having a value equal to ± 1) representing different experiences, are stored in memory by summing the outer product matrices of the memory vectors:

$$T_{ij} = \sum_{a=1}^{m} V_i^{(a)} V_j^{(a)}$$

I assume that the $V_i^{(a)}$ are part of a nearly orthogonal set, so that the components of two different vectors are nearly uncorrelated. A particular experience $V_i^{(b)}$, imperfectly represented by a truncated "clue" vector $U_i^{(b)} = V_i^{(b)}$ for $i \le q$ and $U_i^{(b)} = 0$ for $i > q$, is recalled by iteration of the nonlinear equations

$$^{b}X_i^{(1)} = S\left\{\sum_{j=1}^{m} T_{ij} U_j^{(b)}\right\} = S\left\{qV_j^{(b)} \pm \sqrt{q(m-1)}\right\} = S\left\{V_j^{(b)} \pm \sqrt{(m-1)/q}\right\},$$

where the square root represents the assertion that the individual components of $V_i^{(b)}$ are being perturbed by the addition of uncorrelated contributions having a Gaussian distribution with standard deviation equal to $\sigma = \sqrt{(m-1)/q}$. The probability that one of the components changes sign is

$$P = \int_1^\infty \frac{1}{\sqrt{2\pi\sigma}} \exp\{-x^2/2\sigma^2\}dx = \frac{1}{2}\int_1^\infty \frac{2}{\sqrt{\pi}} \exp\{-x^2/2\sigma^2\}\frac{dx}{\sqrt{2\sigma}} = \frac{1}{2}\frac{2}{\sqrt{\pi}}\int_{1/\sqrt{2\sigma}}^\infty \exp\{-y^2\}dy =$$
$$\frac{1}{2} erfc\{1/\sqrt{2\sigma}\} = \frac{1}{2} erfc\{\sqrt{q/2(m-1)}\}$$

The Hamming distance between $^{b}X_i^{(1)}$ and the correct memory is thus

$$h(^{b}X_i^{(1)}, V_i^{(b)}) = \frac{d}{2} erfc\{\sqrt{q/2(m-1)}\}.$$

After the next iteration I have

$${}^{b}X_i^{(2)} = S\left\{\sum_{j=1}^{m} T_{ij} X_i^{(1)}\right\}$$

and so forth. Here the function *S*(*x*) is a hard saturating function such as tanh(λx) with λ>>1 acting separately on each component of its vector argument. If the $V_i^{(a)}$ are sufficiently different, i.e. are part of a nearly orthogonal set, the system will settle into a state closely resembling $V_i^{(b)}$."

A plot of these results for convergence as a function of normalized clue length is shown here

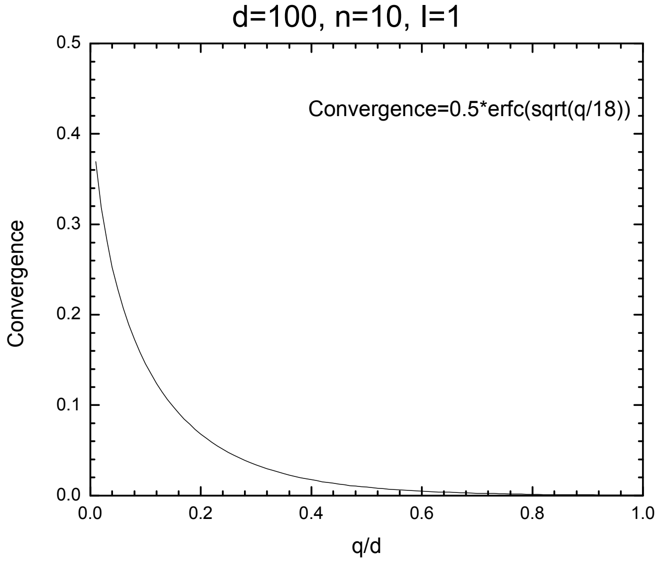


Fig. 6.6 An analytical estimate of convergence of a HNN as a function of normalized clue length (q/d)

This plot of analytical results can be compared to Fig. 6.2a to demonstrate the excellent agreement with computational results for the first iteration of a HNN with $T_{ii} = 0$.

6g. Conclusion

I have tested five possible variables which could have impacted the performance of a HNN: the diagonal condition (setting $T_{ii} = 0$ vs. $T_{ii} \neq 0$), the steepness of the saturating function (hard limiting or more relaxed), the update mode (synchronous vs. asynchronous), when the decision function is applied (every iteration vs. after some number of iterations), and the use of clue vs. corrupted vectors as inputs to the network.

The diagonal condition ($T_{ii} = 0$) was first proposed by Hopfield as a condition for guaranteed stability in the recall algorithm when dealing with corrupted vectors as inputs. I found that convergence with the relaxation of this condition ($T_{ii} \neq 0$) is actually slightly better (.02 change in mean Hamming Distance) than with the rule enforced ($T_{ii} = 0$), both for corrupted and clue vectors as inputs.

When the saturation function is varied, no statistically significant effect on convergence is found. This is the case whether clue vectors or corrupted vectors are presented to the network. There is a slight improvement in convergence between I=1 and I=2, and no significant improvement in convergence from I=2 to I=3, as seen in other sections.

I found that for the synchronous vs. asynchronous update mode, the synchronous update mode displayed better convergence as a function of normalized clue length in the regime where (q/d) < .4 (only for clue vectors). This result also occurred with (c/d) < .9 for corrupted vectors. Further, the variance in convergence is less in synchronous update mode for both clue and corrupt vector implementations, implying that for a given clue (or correctness) length, the range of convergences is less when the update scheme is applied synchronously.

Finally, when the saturation function is applied on different iterations for clue vectors, it was found that application of the saturation function after every iteration shows a slight (at most .05 when q=.17) improvement over application on other possible iterations. For values of (c/d), saturation after every iteration starts to improve convergence as (c/d) increases from (c/d)=.55, up to a limit of (c/d) = .7. From (c/d) = .7 to (c/d) = 1, saturating after every iteration continues performing slightly better than saturation after every second or third iteration, with the effect asymptotically going zero as (c/d) goes to 1.

All of this information combined leads to the conclusion that for a HNN using either clue vectors or corrupted vectors as inputs, the optimum network configuration is one which uses a synchronous update scheme, with $T_{ii} \neq 0$, having a saturation function which is applied after every iteration. Further, systems using clue vectors as inputs have an overall better performance (≈ .05 - .1) at short clue/correctness lengths.

Chapter 7 – Differences Between Conventional Hopfield Neural Networks and DNA Based Hopfield Neural Networks

As discussed in Chapter 5, Hopfield neural networks offer an intriguing example of a content-addressable memory. However, research to date has involved implementations of these networks in silicon. Here I delineate and explore some of the differences between models of HNNs in silicon and DNA.

MYP[90,91] proposed to implement a Hopfield neural network in DNA. As discussed previously, a Hopfield network is one which has the ability to store memories in the connections between nodes. Once stored, the memories can be recalled by the introduction of a "clue vector". A clue vector is a vector with some of the entries missing. In this way, the network functions as a content addressable memory.

7a. Corruption and Completeness Limitations in HNNs

In a set of memory matrices which are not summed to give the usual result for a memory matrix in a HNN, it is interesting to note properties of convergence when a clue or corrupt vector is applied. Consider the following vector and the memory matrix derived from it

$$V^1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad M^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Now assume that a corrupted vector is to be acted upon by the memory matrix. The first entry of the output vector (O) is:

$$O_1^1 = Sign(\sum_j M_{1j}^1 U_j^1)$$

If the original encoded vector had been presented to the network, an entry in the output vector would be equal to Sign(d), where 'd' is the length of the vector. However, given a number of entries which are corrupted (cor), the output now becomes

$$O_1^1 = Sign(d - 2 * cor)$$

since each corrupted entry reduces the positive nature of the argument of the Sign function by two (cancelling out the value it took over, and the need for one other entry to cancel its effect). Therefore, if cor > d/2 , the sign will be reversed from the desired value, and one incorrect entry will be output in the vector.

If clues (vectors with '0' entries for missing information) are presented to the network instead of corrupted vectors, any arbitrary clue length will give a correct answer, since each entry the row of the matrix under consideration helps to "pull" the output entry toward the positive if the desired output is '1' and toward the negative if the desired output is '-1'. Therefore, any positive clue length will reproduce the correct vector.

7b. MYP Model of a Hopfield Neural Network

Construction of MYP's proposed network involves creating both vectors and matrices in DNA.

7b.1 Construction of Input Vectors

To create input vectors, it is necessary to find a means of converting digital and analog data to a meaningful representation in DNA. MYP's original idea for this was to use a chip design, with each cell containing dsDNA. Each of the dsDNA oligomers is random and distinct. Once two of these chips are prepared, the image that the user wishes to encode in DNA is positioned over one of the assemblies. Each cell in the image has a corresponding cell on the chip. In this implementation, the image to be recorded into the DNA is a black and white picture. The white cells are set up to allow light through, while the black ones block it. MYP proposed then placing an ultraviolet light source above the input picture. This would allow cells on the chip corresponding to white cells to receive ultraviolet light, while cells on the input picture which are black would be protecting the underlying DNA. If left on for a period of time, this ultraviolet exposure would cause the DNA on the chip corresponding to the white cells to denature. Once denatured, the ssDNA could be siphoned off and placed into a tube. This would then be a representation of the input picture. Each of the strands in solution has a primer on each end (P,Q) which facilitates reactions such as ligation and PCR.

7b.2 Formation of the Memory Matrix

Creating the matrix of connections that would exist in a Hopfield network (aka – memory matrix) is now a straightforward matter. Since the matrix for the Hopfield network is given by:

$$T_{ij} = \sum_{a=1}^{m} V_i^a V_j^a$$

(where "a" is the number of memories, a "V" is a memory vector) it is possible to form an outer product matrix using oligomers from each of the input vectors. This equation can be realized in DNA, since each component of each vector is a distinct DNA sequence with some unit presence in the solution. Since an entry in an outer product matrix is the result of multiplication of two vector components, this operation can be represented by the ligation of two relevant pieces of DNA. Creating the entire memory matrix requires the ligation of all vector components to all other vector components (for a given vector) in solution. This is accomplished by creating a "linker" strand, consisting of the complement to primer P and the complement to primer Q. This linker is allowed to hybridize in solution with the oligomers, giving strands connected in pairs. Once this has been accomplished, ligase acts on the strands, connecting them permanently. Finally, the solution is heated so that the original linker falls off, leaving the newly created matrix strands. The memory matrix is constructed using the complements of each memory vector component ligated to another "output vector space", which has oligomers representing the equivalent of each input space vector oligomer.

7b.3 Memory Recall Operation

To effect the recall operation of the network, a “clue vector” is presented in solution. This representation is distinct from a “corrupted vector” recall since this scheme is designed to represent a “lack” of information by a zero, and the presence of information by the appropriately signed binary bipolar entry. Since the vector is a set of oligomers with unit concentrations, a clue vector is a set of oligomers with some of the entries missing. To represent this effectively, the missing elements in the input vector are represented by zeros. If I wanted to recall v^1, the clue vector might be something like this:

$$u^1 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} AACGTC \\ GGTCGC \\ \ \end{pmatrix}$$

where I have shown both the mathematical form and constituent DNA strand form. Once put into solution, the memory matrix is allowed to act on these input oligomers. Since the input oligomers are complementary to the oligomers on the 5’ end of the DNA strand, they will anneal in solution to the matrix strands at this end. Once this has been achieved, polymerase is added to the solution, which will grow the strand toward the 3’ end, resulting in dsDNA.

It now becomes necessary isolate and denature the output strand from the matrix strand/input strand combination. Nickase is introduced to the solution, and set to recognize the point at the junction between P$_{\text{bar}}$ and Q$_{\text{bar}}$. The nickase will cut at this point, and create an opening for polymerase to act again, growing along the matrix strand toward the 3’ end. This growing will force the existing strand to denature. These cut off strands are the desired output. This output process can be repeated if necessary using Strand Displacement Amplification[92] (SDA) to obtain increased quantities of output product.

7b.4 Reading the output strand

The output strands can be read using the second chip. This chip is identical to the original chip, with the exception that the output chip has all ssDNA. In effect, this chip is the equivalent of the first chip, except all possible strands have been denatured. What remains on the chip is a set of ssDNA, each strand of which is complementary to one of the possible output strands from the memory matrix. To make visible readout possible, the output strands from the memory are “tagged” with a fluorescent molecule which will output in the visible portion of the spectrum when excited with ultraviolet light. The output strands, once introduced to the chip, will bind to their single stranded components on the chip. The cells with dsDNA oligomers will fluoresce, indicating the original white cells on the image. The cells without dsDNA oligomers will not fluoresce when stuck with the ultraviolet light, thereby showing the cells which are intended to represent black in the original image.

7c. Directions for Future Research

7c.1 DNA Oligomer Length as a Basis Vector in Hopfield Neural Networks

In previously discussed models of neural networks in DNA, the vectors were represented by sets of oligomers, with each oligomer serving as a basis vector. This had the advantage of a large number of possible basis vectors being readily generated by the selection of a sample of randomly generated oligomers. However, it was necessary to attach each oligomer to a chip, which could serve both to provide the initial data for information vectors, as well as the output working substance for recalling information. This system suffered statistical limitations provided by the possible lack of amplification of a particular oligomer, leading to the potential loss of a basis vector.

Another potential model involves representing basis vectors by unit concentrations of oligomers, with different length oligomers representing different basis vectors. A possible implementation of this system is shown here.

Input for this method of information processing would first involve reading in the desired vectors for later recall. This would be accomplished by having a program map each piece of information to be recalled to a number on a scale. For example, if the data was a set of grayscale pictures, a scale of one to twenty four might be used. If the data were a set of letters comprising a text, a scale of twenty seven possible values could be used, each value corresponding to different letters of the alphabet. Once each piece of information is mapped to the scale, this is translated to an amount of an oligomer to be used for that basis vector, with different basis vectors corresponding to different length oligomers. For example, suppose the picture below was read into memory:

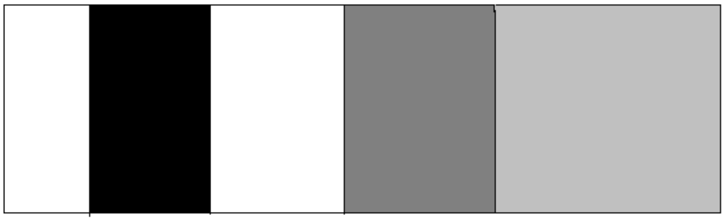


Fig. 7.1 – Possible Grayscale Input Picture

The above picture has the following sequence:

white cell, black cell, white cell, dark grey cell, light grey cell

On a scale of one to twenty four, black might correspond to zero, which would make white correspond to twenty four. The values of grey would be in between, giving a representation in this scale as:

24, 0, 24, 10, 19

Each of these values is then forwarded to a program operating a device, such as a robopippettor, which is able to extract appropriate amounts of oligomers from tubes of random DNA, with each tube containing oligomers of differing lengths. So the above scale might then correspond to:

24 nL of 20 bases ssDNA

0 nL of 30 bases ssDNA

24 nL of 40 bases ssDNA

10 nL of 50 bases ssDNA

19 nL of 60 bases ssDNA

Construction of the memory matrix for one of these vectors involves ligation of the same amounts of each input vector component with other components in that vector. As shown previously, this outer product matrix:

$$T_{ij} = \sum_{a=1}^{m} V_i^a V_j^a$$

will allow memories to be encoded in DNA concentrations. It should, however, be noted that the implementation of this scheme does not allow for scalability, since the number of possible oligomers increases as n, whereas in MYP's proposal, the number increased as 4^n. However, this implementation might be more useful for smaller scale implementations, as results can be read out using polyacrylamide gel electrophoresis rather than a readout chip[93], which might be more cost prohibitive.

7c.2 Models of Feedforward Networks using DNA as a Computing Substance

Previous suggested implementations of neural networks in DNA, such as that proposed by MYP, are predicated on the encoding of connection strengths ligated oligomers as a working substance. It is a straightforward matter to envision extending this idea to other neural network models.

For example, a feed forward network, such as a classifier, involves taking an input vector and determining to which class the input vector belongs.

If it were the case that a neural network had already taken the time to train itself on a silicon system, these matrix values could then be input into solution as concentrations of matrix oligomers connecting layers in the feed forward architecture. This being the case, the DNA system could then sort very large patterns much more quickly than could be accomplished with a conventional computational system.

With an idea in mind of how to implement an HNN in DNA, it is of interest to examine limitations on an MYP HNN computationally, such as relaxing constraints on the network, quantifying the clue length required for recall, etc. A simulation exploring these constraints is the subject of the next section.

7d. Convergence of Clues in Separated Matrices Using DNA

In a conventional HNN, the memory matrix is constructed by using the sum of the outer product matrices generated from the memory vectors. However, it is possible to effect reactions in DNA for the formation of a HNN which do not rely upon this process. If this implementation is used, different rules or conditions for convergence apply compared to a conventional HNN.

7d.1 Minimum Constraints on Amount of Reactants

The first step to the successful realization of a HNN in DNA is the presence of an adequate amount of reactant sample to ensure fidelity at the output step. It is possible to produce increased product amounts using ILA. However, if this step is to be eliminated, a minimum quantity of input reactant is required to ensure successful recall. Suppose there only exists one memory in DNA to be encoded, given by the vector:

$$V^1 = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{pmatrix}$$

then the memory matrix encoding this vector is given by

$$T^1 = \begin{bmatrix} O_{1b}I_{1b} & O_{1b}I_{2b} & O_{1b}I_{3b} & O_{1b}I_{4b} \\ O_{2b}I_{1b} & O_{2b}I_{2b} & O_{2b}I_{3b} & O_{2b}I_{4b} \\ O_{3b}I_{1b} & O_{3b}I_{2b} & O_{3b}I_{3b} & O_{3b}I_{4b} \\ O_{4b}I_{1b} & O_{4b}I_{2b} & O_{4b}I_{3b} & O_{4b}I_{4b} \end{bmatrix}$$

(where ‘b’ in the subscript denotes the strand which is complementary to the original strand. For example, I_{1b} is the complement of I_1).

Assume that a minimum amount (a) of output material is required to display the output component successfully. Further, assume that a clue vector is given to the system:

$$U^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I_1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The first question, then, is how much of this input needs to be presented to the network to ensure recall of the whole output vector? If I_1 is presented to the memory matrix shown above, the output would be:

$$O^1 = \begin{pmatrix} O_1 \\ O_2 \\ O_3 \\ O_4 \end{pmatrix}$$

since each of components in the first column of the memory matrix interact with the clue. So if a given output vector is 'd' components long, and 'a' is the minimum amount of each component required for successful readout, it is imperative to have, in this case,

$$Input_{\min} = a * d$$

However, the minimum aliquot of each entry is reduced if the clue is longer. For example, for a clue of

$$U^1 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ 0 \\ 0 \end{pmatrix}$$

interacting with the memory matrix would produce the same output as before, but this time through interaction with the first two columns of the memory matrix instead of only the first column. This reduces the amount of material required for each input entry by 'q', where 'q' is the length of the clue vector. This leads to a minimum input aliquot for each clue entry of

$$Input_{\min} = \frac{a*d}{q}$$

assuming there is only one vector in the system. If there is more than one vector, the amount of input reactant must be multiplied by the number of vectors in the system (n) to ensure each memory matrix is allowed to react with the input oligomers.

7d.2 Lack of Equivalence Between Standard Matrices and DNA Matrices

At first it may be easy to assume that representing DNA vectors by their numerical counterparts would be the most straightforward way to proceed. However, it can easily be demonstrated that valuable information is lost in the transition. Consider the following set of vectors and their memory matrices

$$V^1 = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad V^2 = \begin{pmatrix} I_{1b} \\ I_{2b} \\ I_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$$

$$T^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} O_{1b}I_{1b} & O_{1b}I_{2b} & O_{1b}I_{3b} \\ O_{2b}I_{1b} & O_{2b}I_{2b} & O_{2b}I_{3b} \\ O_{3b}I_{1b} & O_{3b}I_{2b} & O_{3b}I_{3b} \end{bmatrix}$$

$$T^2 = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} O_1I_1 & O_1I_2 & O_1I_{3b} \\ O_2I_1 & O_2I_2 & O_2I_{3b} \\ O_{3b}I_1 & O_{3b}I_2 & O_{3b}I_{3b} \end{bmatrix}$$

Note the lack of equivalence between these two scenarios. In the case of ordinary matrices, the negative values in the vector cancel each other out in the formation of the memory matrix. However, since the type of data is different for DNA vectors, this does not occur. As a demonstration, consider the presentation of a clue vector in the form

$$U^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I_1 \\ 0 \\ 0 \end{pmatrix}$$

Allowing the numerical vector to be acted upon by the memory matrices and saturating with a Sign function gives:

$$O^1 = Sign\left[\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}\right] = \begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix}$$

but with the DNA representation, the output becomes

$$O^1 = Sign\begin{pmatrix} O_1 \\ O_2 \\ O_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

This is because the DNA doesn’t interact with the matrix oligomers in the second memory, only the first. However, the numerical model does allow for a successful operation with the second memory matrix.

7e. Mathematical Representation of DNA Based Hopfield Neural Networks

Since it is now evident that the typical mathematical operations represented in matrix algebra do not correspond exactly with the outputs from a DNA based neural network, it is a natural extension to explore which mathematics, in fact, do accurately represent the operations of these systems.

The mathematics of the oligomer interactions in these networks can be realized by thinking of each matrix oligomer as a function. The inputs to an individual matrix oligomer can be represented by either a "positive" oligomer ($I_n = 1$) or by a "negative" oligomer (its complement in input space: $I_{nb} = -1$). This oligomer can then produce an output which is an equivalent positive oligomer in output space or a negative oligomer in output space ($O_n = \pm 1$). This is a function which takes ±1 as input and produces ±1 as output.

However, the situation is more complicated by the fact that each matrix oligomer will bind selectively with only complements on the right hand side of itself. For example, in the matrix given by:

$$T^2 = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} O_1 I_1 & O_1 I_2 & O_1 I_{3b} \\ O_2 I_1 & O_2 I_2 & O_2 I_{3b} \\ O_{3b} I_1 & O_{3b} I_2 & O_{3b} I_{3b} \end{bmatrix}$$

The entries in the first column will bind only with the first entry in a clue vector. Likewise, the matrix entries in the second column will bind only with the second entry in a clue vector, etc. This leads to a condition $\delta(r,c)$ (a Kronecker Delta function) on each oligomer, meaning that the output will be zero unless the row number in the clue vector (r) is equal to the column number of a given memory matrix (c).

The output will be determined by the left side of the matrix oligomer under consideration. For example, if a particular oligomer has O_{nb} as the left side, it will produce O_n in reactions, which means the output will be +1. If, however, the oligomer has O_n as the left side, it will produce -1 as the output.

These two rules in conjunction allow for an exhaustive list of functions, each of which describes one possible matrix oligomer. A table listing these values is

DNA Oligomer	Mapping	Kronecker Delta Function Representation
O_nI_n	$-1 \rightarrow -1$	$-\delta(r_n,c_n)\ \delta(x_n,-1)$
O_nI_{nb}	$1 \rightarrow -1$	$- \delta(r_n,c_n)\ \ \delta(x_n,1)$
$O_{nb}I_n$	$-1 \rightarrow 1$	$\delta(r_n,c_n)\ \delta(x_n,-1)$
$O_{nb}I_{nb}$	$1 \rightarrow 1$	$\delta(r_n,c_n)\ \delta(x_n,1)$

Fig. 7.2 – Representation of DNA Interactions as a Function

Using these functions, it is possible to map, for a given input, the mathematical output. Consider the matrix shown above and its function equivalent:

$$T^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} O_{1b}I_{1b} & O_{1b}I_{2b} & O_{1b}I_{3b} \\ O_{2b}I_{1b} & O_{2b}I_{2b} & O_{2b}I_{3b} \\ O_{3b}I_{1b} & O_{3b}I_{2b} & O_{3b}I_{3b} \end{bmatrix} =$$

$$\begin{bmatrix} \delta_{11} & \delta_{12} & \delta_{13} \\ \delta_{21} & \delta_{22} & \delta_{32} \\ \delta_{31} & \delta_{32} & \delta_{33} \end{bmatrix}$$

Presenting a "DNA clue" in the form (1,0,0) as before, to this set of functions, gives:

$$T^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{bmatrix} O_{1b}I_{1b} & O_{1b}I_{2b} & O_{1b}I_{3b} \\ O_{2b}I_{1b} & O_{2b}I_{2b} & O_{2b}I_{3b} \\ O_{3b}I_{1b} & O_{3b}I_{2b} & O_{3b}I_{3b} \end{bmatrix} \begin{pmatrix} I_1 \\ 0 \\ 0 \end{pmatrix} =$$

$$\begin{bmatrix} \delta_{11}\delta(x_1,1) & \delta_{12}\delta(x_2,1) & \delta_{13}\delta(x_3,1) \\ \delta_{21}\delta(x_1,1) & \delta_{22}\delta(x_2,1) & \delta_{23}\delta(x_3,1) \\ \delta_{31}\delta(x_1,1) & \delta_{32}\delta(x_2,1) & \delta_{33}\delta(x_3,1) \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \delta(1,1) \\ \delta(1,1) \\ \delta(1,1) \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} O_1 \\ O_2 \\ O_3 \end{pmatrix}$$

As can be seen here, this representation faithfully recalls information in encoded in the DNA neural network.

Extension of this idea into a general function is straightforward. Any matrix of this nature can be broken down into elements consisting of ±1. As such, when it acts upon a vector entry, the only constraint is that the row number of the vector input be the same as the column number of the matrix. A comparison of the DNA function representation with the usual numerical matrix representation is given here, where each function is for a single entry in one memory matrix

DNA Oligomer	Numerical Representation	Numerical Function Representation	DNA Function Representation
O_nI_n	1	$\delta(r_n,c_n)$	$-\delta(r_n,c_n)\ \delta(x_n,-1)$
O_nI_{nb}	-1	$-\delta(r_n,c_n)$	$-\ \delta(r_n,c_n)\ \ \delta(x_n,1)$
$O_{nb}I_n$	-1	$-\delta(r_n,c_n)$	$\delta(r_n,c_n)\ \delta(x_n,-1)$
$O_{nb}I_{nb}$	1	$\delta(r_n,c_n)$	$\delta(r_n,c_n)\ \delta(x_n,1)$

Fig. 7.3 – Comparison of DNA and Numerical Matrix Entries

The above chart shows that there is more information in the DNA network, since oligomers can retain function information properties that numbers cannot (such as when there are two different ways to create a DNA matrix oligomer which would have a value of '-1'). As Golomb[36] has said, a Shannon bit "is the amount of information gained (or entropy removed) upon learning the answer to a question whose two possible answers were equally likely, a priori." If one examines the numerical function representation above and compare it to the DNA function representation, a difference in the amount of possible questions to ask to reach an answer is seen. In the numerical representation case, if a "±1" is presented to the network and the memory matrix entry is "1", the only question to determine the output is something akin to "Is there a negative sign in front of the Kronecker Delta function representing this entry?" This is suggestive of there being one "Shannon Bit" of information in that matrix element. However, if the matrix entry is a DNA oligomer, and a "±1" is present to the network in the form of an input oligomer, two questions must be asked to determine the output of a particular matrix entry. These would include something such as "Is there a negative sign in front of the Kronecker Delta function representing this entry?". This would be followed by another question, such as "Does the second Kronecker function in the matrix entry function include '-1' as one of its arguments?". From these two "Yes/No" questions, it is possible to exhaustively list the outputs from the memory matrix element. Therefore, each entry has not one, but two Shannon bits of information in it. This means that the memory matrix capacity, instead of being some number of memory vectors 'n' as in the case of a conventional HNN, should instead be '2n'.

Chapter 8 – Results of MYP Neural Network Simulation

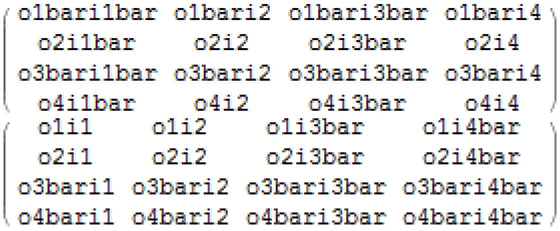
8a. Experiment Convergence

Karabay et al[94] have attempted to implement a small scale model of an MYP neural network. In this model, the authors attempted to form a content addressable memory by the ligation of four input vectors (d=4, n=2) to form a set of outer product matrices. In this implementation, as described in other sections, vector entries were represented by oligomers of DNA, and matrix entries were represented by ligated vector entries. This resulted in a set of four matrices which were allowed to interact with a clue vector, which consisted of one of the components of the first memory encoded in the neural network. The experimental results were confirmed by me using a simulation.

Since the oligomers used in this experiment are designed to effect computation akin to a Hopfield Neural Network (HNN), it might at first appear that simulation of this experiment *in silico* involves the construction of a memory matrix like that prescribed by Hopfield – the sum of the outer product matrices formed by each vector with itself. However, this is not accurate (as mentioned in Chapter 7), since in this representation DNA does not add like a scalar. For example, consider the formation of a memory matrix for the following vectors

$$V^1 = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} I_1 \\ I_{2b} \\ I_3 \\ I_{4b} \end{pmatrix} \qquad V^2 = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} I_{1b} \\ I_{2b} \\ I_3 \\ I_4 \end{pmatrix}$$

The two outer product matrices would appear as in Fig. 9.1 below



$$\begin{pmatrix} \mathrm{o1bari1bar} & \mathrm{o1bari2} & \mathrm{o1bari3bar} & \mathrm{o1bari4} \\ \mathrm{o2i1bar} & \mathrm{o2i2} & \mathrm{o2i3bar} & \mathrm{o2i4} \\ \mathrm{o3bari1bar} & \mathrm{o3bari2} & \mathrm{o3bari3bar} & \mathrm{o3bari4} \\ \mathrm{o4i1bar} & \mathrm{o4i2} & \mathrm{o4i3bar} & \mathrm{o4i4} \end{pmatrix}$$

$$\begin{pmatrix} \mathrm{o1i1} & \mathrm{o1i2} & \mathrm{o1i3bar} & \mathrm{o1i4bar} \\ \mathrm{o2i1} & \mathrm{o2i2} & \mathrm{o2i3bar} & \mathrm{o2i4bar} \\ \mathrm{o3bari1} & \mathrm{o3bari2} & \mathrm{o3bari3bar} & \mathrm{o3bari4bar} \\ \mathrm{o4bari1} & \mathrm{o4bari2} & \mathrm{o4bari3bar} & \mathrm{o4bari4bar} \end{pmatrix}$$

Fig. 8.1 – Ligated Structures in an MYP Neural Network

Unlike for the usual model for a HNN, the two outer product DNA memory matrices must not be simply added together due to the likelihood of cross hybridization, and any attempt to model the DNA network without taking this into account will give incorrect predictions. Instead, the memory matrix is preserved as four components (O_iI_j, $O_iI_j$bar, $O_i$barI_j $O_i$bar$I_j$bar) which will act separately on the query before adding the results together, at which point strands representing opposite polarities can hybridize and thus cancel in whole or in part.

For example, in this experiment, the clue vector consisting of {i1, 0, 0, 0} was input to the network.

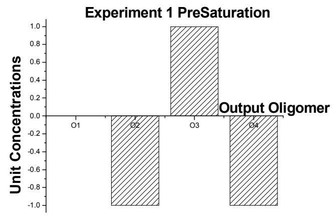
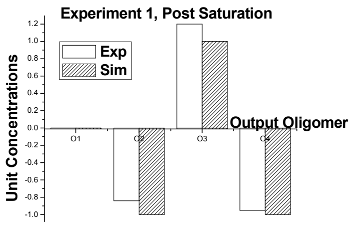
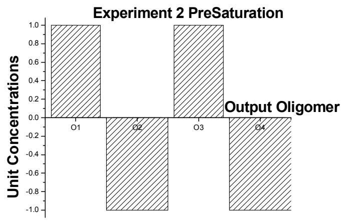
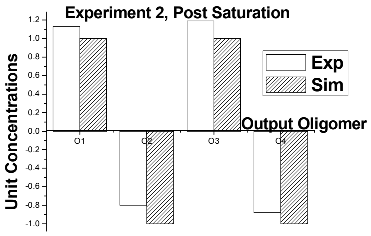
To determine the output of the HNN on this clue vector in DNA:

1) Search for the complement to each clue entry by examining the "right hand side" of each memory oligomer for the complement to the clue entry. In this case, the first column of the first matrix is the only set of entries that have the complement ("i1bar").

2) Determine the complement to the "left hand side" of those oligomers which match. In this situation, the left hand sides of the matching oligomers are o1bar, o2, o3bar, and o4. Therefore the complements are o1, o2bar, o3, and o4bar.

These are the outputs from the HNN implemented in DNA. In order to do this more efficiently, I wrote a Mathematica code which allows the entry of arbitrary vectors, constructs the appropriate set of memory matrices, then considers an input clue and generates the appropriate output.

For each of the experiments above, I (and others) obtained outputs for both simulations and experiments indicated in Fig. 8.2. Experimental data is not shown for presaturation values, since different vector components represented by ssDNA cannot be distinguished from each other in gel electrophoresis.

a) Output Values Before Saturation	b) Output Values After Saturation
 Experiment 1 PreSaturation 	 Experiment 1, Post Saturation
 Experiment 2 PreSaturation 	 Experiment 2, Post Saturation

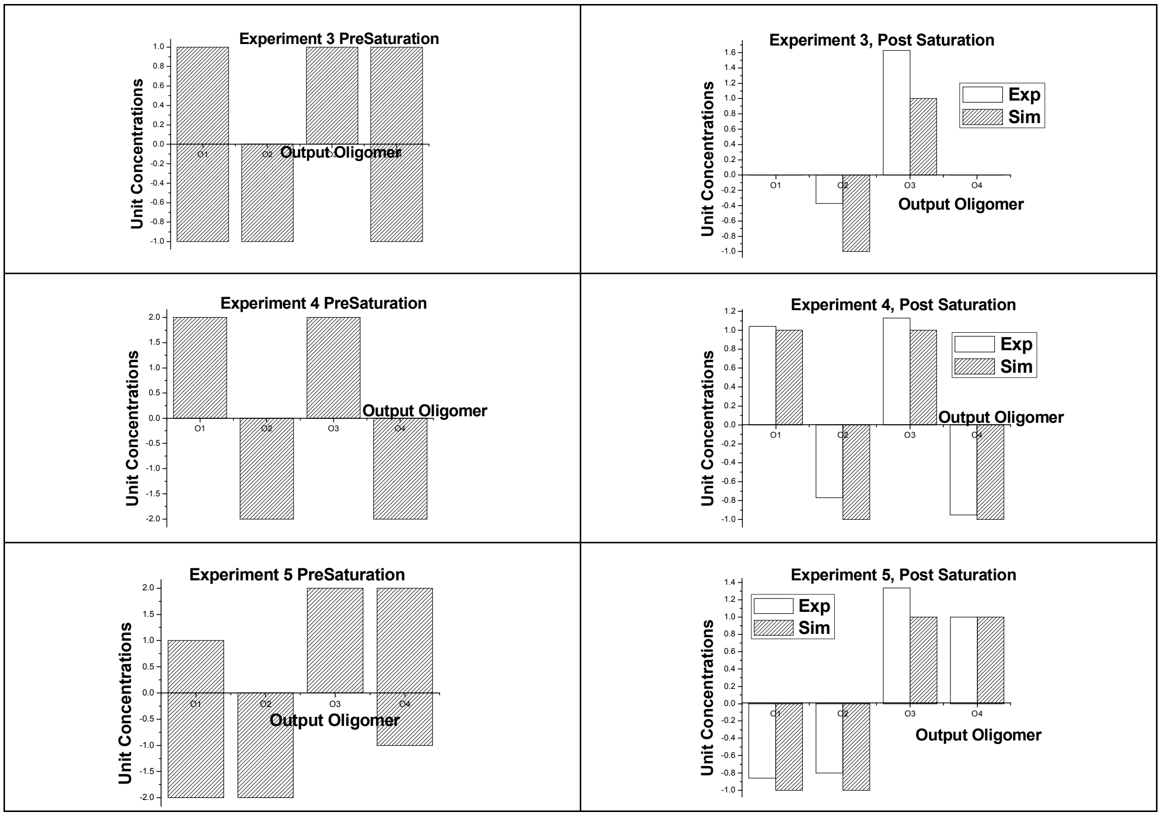


Fig. 8.2 – Comparison of Experimental and Simulated Values of Output from MYP Neural Network

8b. Simulation Convergence

It is my intent here to explore the difference in convergence between an ideal HNN and a MYP model HNN. To see the difference in difference in convergence for these two systems on a small scale, a plot is shown below where the average convergence as a function of clue length is plotted on the same graph with the same experiment performed using a MYP network.

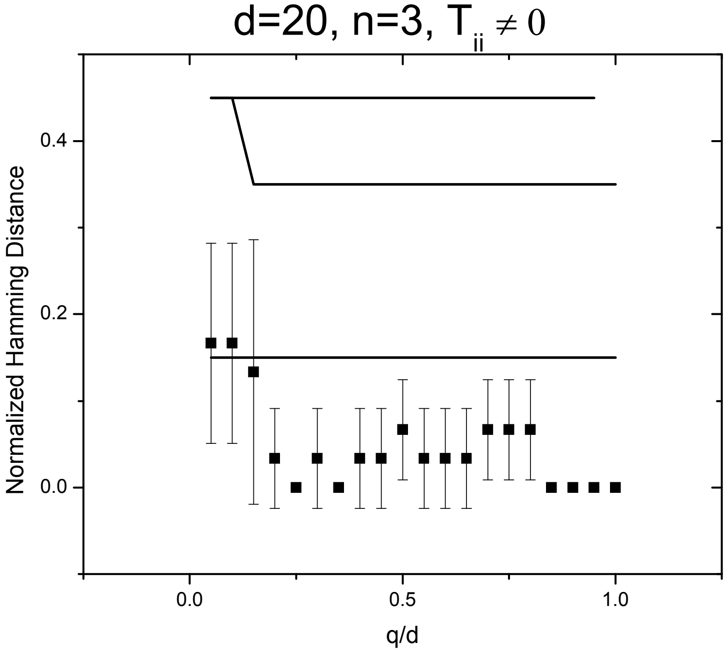


Fig. 8.3 – Comparison of MYP Neural Network with Standard HNN

In the above plot, a simulation was run for a vector length of twenty (d=20) with three vectors encoded in the memory matrix (n=3). Each of the lines above represents the Hamming Distance of the output from one of the DNA vectors used as a clue to the system, while the set of points with error bars represents the average Hamming Distance found from the output of an idealized HNN with the same length idealized clues used as inputs. In this small simulation, the idealized HNN performed better overall.

A table showing the exhaustive output from the DNA vectors is shown here

q/d	Output	Desired Output	Hamming Distance
.05	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,0,0,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-	.45

		1,1,1}	
.1	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,0,0,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.45
.15	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.2	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.25	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.3	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.35	{1,1,-1,0,0,0,1,-1,1,1,0,0,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.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}	.35
.45	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.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}	.35
.55	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.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}	.35
.65	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.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}	.35
.75	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.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}	.35
.85	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.9	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
.95	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35
1	{1,1,-1,-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,1,1,1}	{-1,1,-1,1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,1,-1,1,1}	.35

Fig. 8.4 – Results of Output from MYP Neural Network

From these results it is apparent that, in some cases, the idealized HNN converges more effectively than an MYP network. The reason for this, and its dependence on vector length (d) and number of encoded vectors (n) is unclear and requires further investigation.

Chapter 9 – Determination of the Spurious Energy States of a Hopfield Neural Network

As mentioned previously, Hopfield[21] noted that in the construction of the memory matrix, it was possible to encoded states which were stable endstates and yet not original memories encoded in the neural network. It is the undertaking of this section to offer a quantification of the states, in terms of their number and distance from encoded memories. For the purposes of this section, a spurious state is defined as a stable output which is not exactly the same as a stored memory. In fact, in many cases these states are very close in Hamming Distance to the stored memories in problems examined (ie: d=100, n=10) The properties of the Hamming Distance dependence on vector length (d) and number of vectors encoded in memory (n) will be demonstrated in Section 8e.

9a. Vectors Used and Measurement of Convergence

In this section I ran a simulation to determine both the number of spurious states as a function of vector length (d) and number of vectors encoded in the memory matrix (n). The simulation was set up by generating a random set of vectors (n), each with the same length (d). These vectors were then used to construct a memory matrix, using the sum of outer products rule. A separate set of vectors (r) (each with length 'd') were then generated randomly and acted upon by the memory matrix in a synchronous update scheme. Once acted upon by the memory matrix, the vectors were saturated using a Heaviside function. This output was then compared to the vectors encoded in the memory matrix, to check for convergence. If the output vector matched one of the vectors used to construct the memory matrix, a “hit” for convergence on the first iteration was recorded. If the vector did not converge, it was acted upon by the memory matrix again and saturated, then compared with both the original vectors used to construct the memory and the last iteration of itself. If it matched one of the original vectors, a “hit” for convergence on the second iteration was recorded. If this output vector did not match one of the original vectors used to construct the memory matrix, but did match the prior iteration of itself, it was counted as a spurious memory, since the vector converged to a stability point but did not converge to one of the desired memories.

If neither of these conditions were met, the process of acting on the memory vector by the memory matrix and saturating was repeated, followed by the comparison with both encoding vectors and prior iterations of output vectors. This process was repeated for up to ten iterations, recording when the vector converged to either a spurious state or one of the desired vectors that was used to construct the memory matrix.

The output from each iteration was also compared to the outputs from other iterations. In doing so, it was possible to check for “limit cycles”. These are output results where the output oscillates between 2 (or more) output vectors in a nonterminal sequence.

9b. Comparison with Exhaustive Search Results

As a baseline to assess how well statistical methods compared to exhaustive searches, an exhaustive construction was made for vectors with reasonably short values ($2 \leq d \leq 9$).

In this system, a simulation was run to set up every possible combination of vectors of d-tuples. For example, for d=3, my simulation constructed all eight possible vectors with a length of 3. It then used these vectors, for n=2, to generate a set of all 28 possible matrices which could be constructed from 2 vectors. Each matrix was then allowed to act on each of the 8 possible vectors as inputs, in a manner similar to the simulation listed in section 8a above. This exhaustive manner of searching the possible vector space is fruitful for being conclusive regarding results. Unfortunately, as the size of the vector grows, so does the computational time required for a solution. Therefore, for large values of "d", it is not practical to have such a system to calculate spurious states, and one is left with a statistical sampling model.

Below is a comparison of the results for an exhaustive search for $2 \leq d \leq 9$ along with the simulation run with statistical sampling.

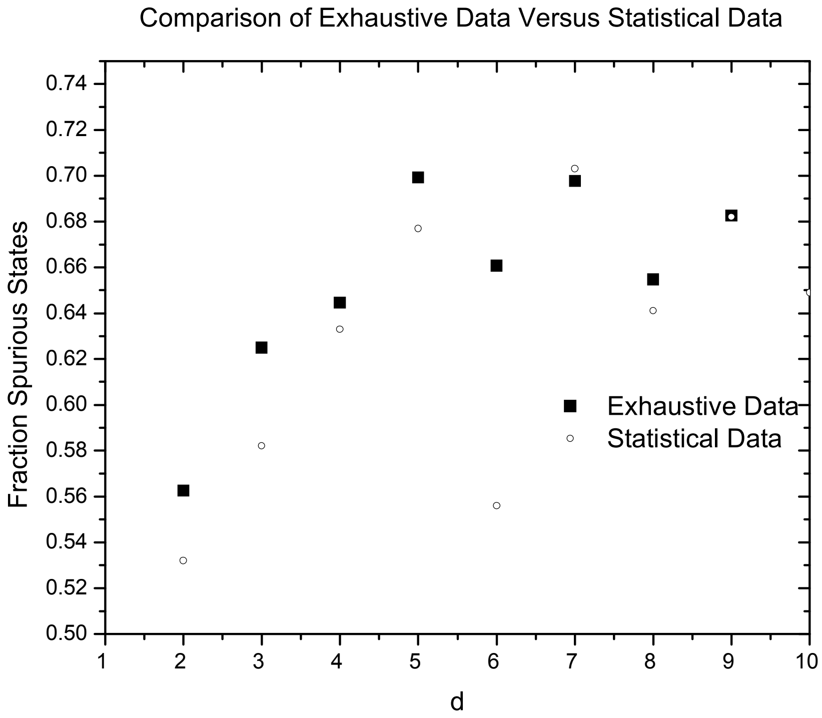


Fig. 9.1 Comparison of the number of Spurious States as a fraction of the total number of system states for short vector lengths (2 ≤ d ≤ 9). Points are plotted both for an exhaustive search and a statistical search. The statistical search entailed a number of matrices tried (M=10) as well as a number of different possible trial runs with different input vectors (r=100)

9c. Monte Carlo Simulation Procedure

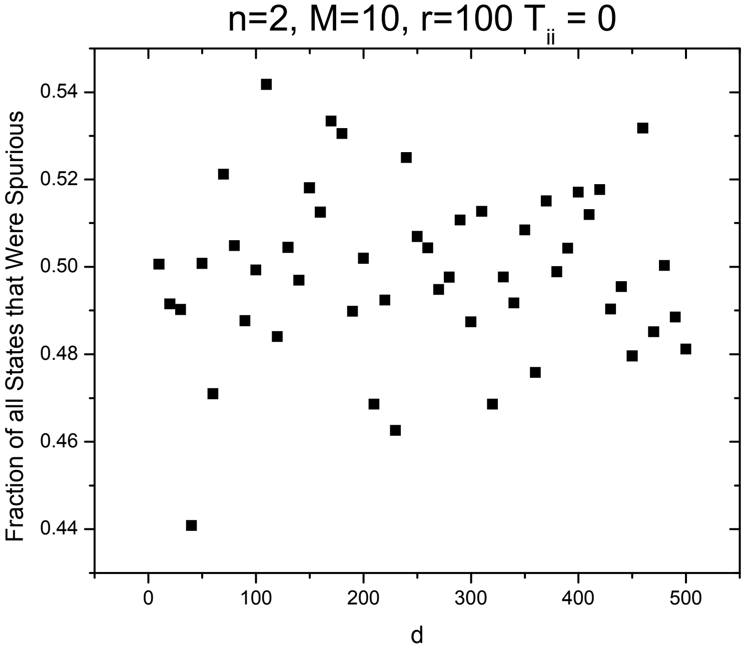
The simulation was run for a wide range of vector lengths (10 ≤ d ≤ 500) in increments of 10. The simulation was tested with numbers of vectors encoded in the memory matrix (2 ≤ n ≤ 30). The procedure for this simulation is given here:

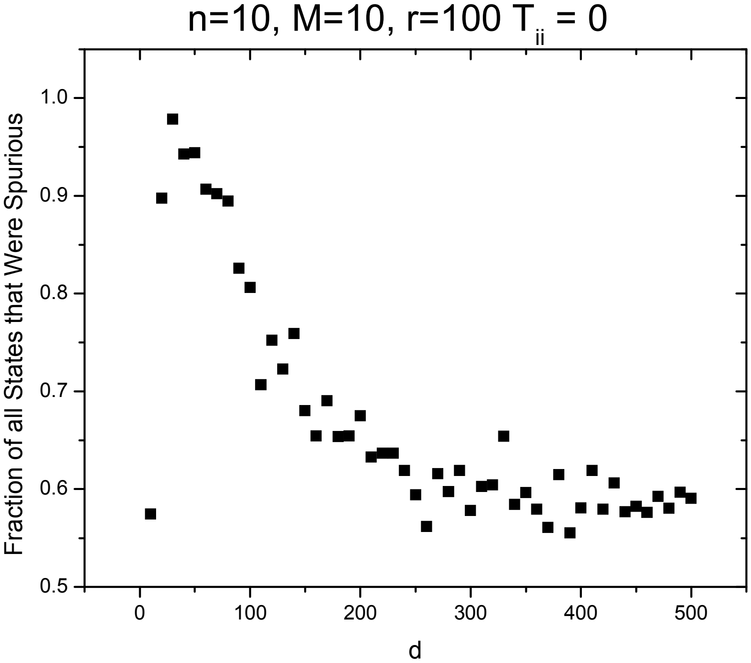
1) A set of vectors (n), each with length ‘d’ was chosen at random. The sum of the outer products of these vectors was taken to construct a memory matrix.
2) A set of input vectors (r) (each with length ‘d’) was selected at random. These input vectors were each acted on by the memory matrix, and then saturated using a Heaviside function.
3) This process of iterating the output vector by feeding it through the memory matrix and allowing the saturating function to act was repeated ten times, so ten iterations for each input vector were recorded.
4) These ten iterations were examined to determine when the output vector converged to one of the original vectors used to construct the memory matrix. If the vectors did not converge to one of the original vectors, it was examined across all iterations to determine when it stabilized into a spurious state or a limit cycle.
5) After all vectors in ‘r’ were examined, a new matrix (M) was constructed out of a new set of ‘n’ random vectors, and the process was repeated.

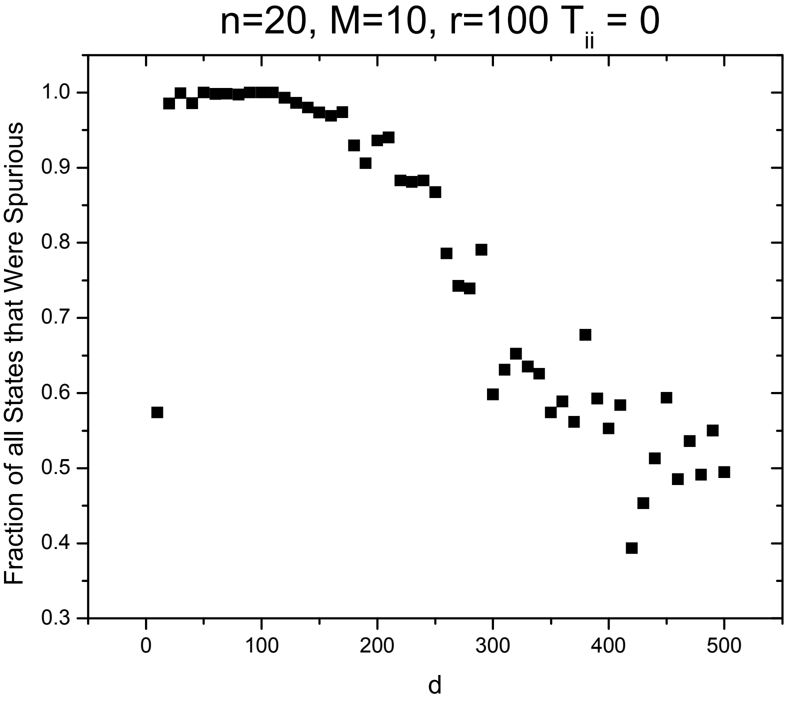
This method of randomly generating vectors and matrices allows for a reasonable statistical ensemble to be constructed, showing a representative set of states which are spurious and convergent for a given vector length (d) and number of vectors in the memory matrix (n). After this process was complete, the simulation was run again, this time with the condition that the diagonal elements of the memory matrix not set to zero, but rather be permitted to retain their values assigned in the sum of outer products construction procedure.

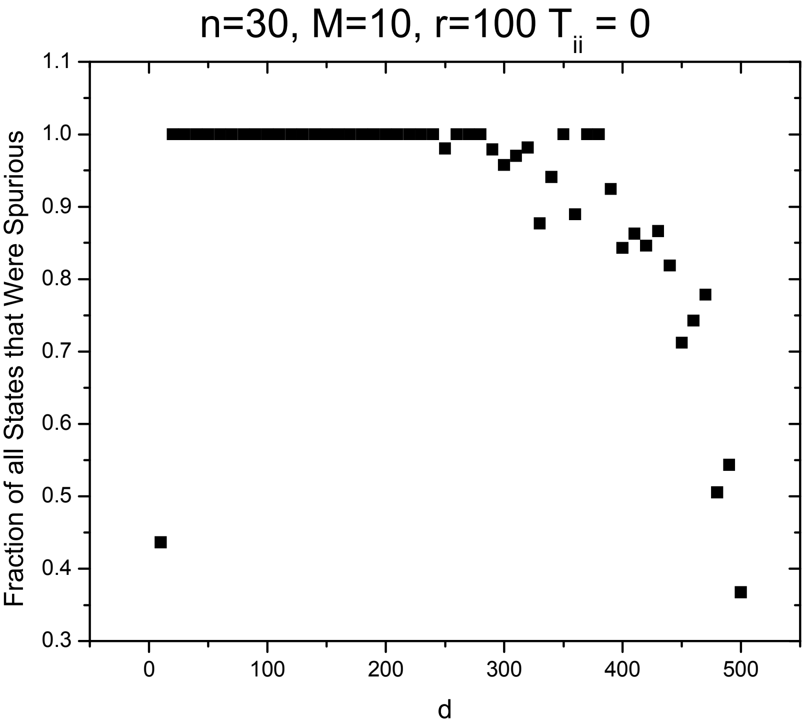
9d. Distribution of Spurious States as a Function of Vector Length and Number of Vectors Encoded

Here are plots of the fraction of spurious states as a function of vector length (d) for a given number of vectors encoded in the memory matrix (n), with the number of different matrices used (M) equal to ten. There were 100 different random vectors (r) tried against each memory matrix.

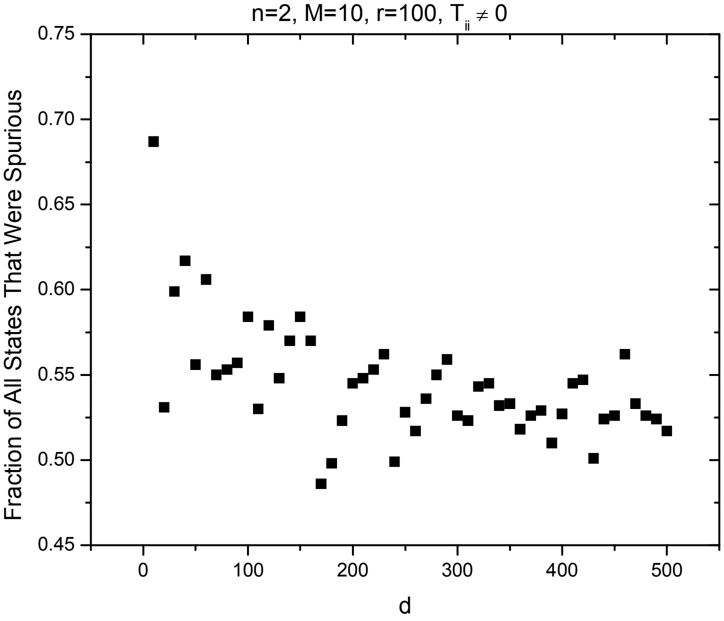

n=2, M=10, r=100 T_{ii} = 0
Fraction of all States that Were Spurious
0.54
0.52
0.50
0.48
0.46
0.44
0
100
200
300
400
500
d

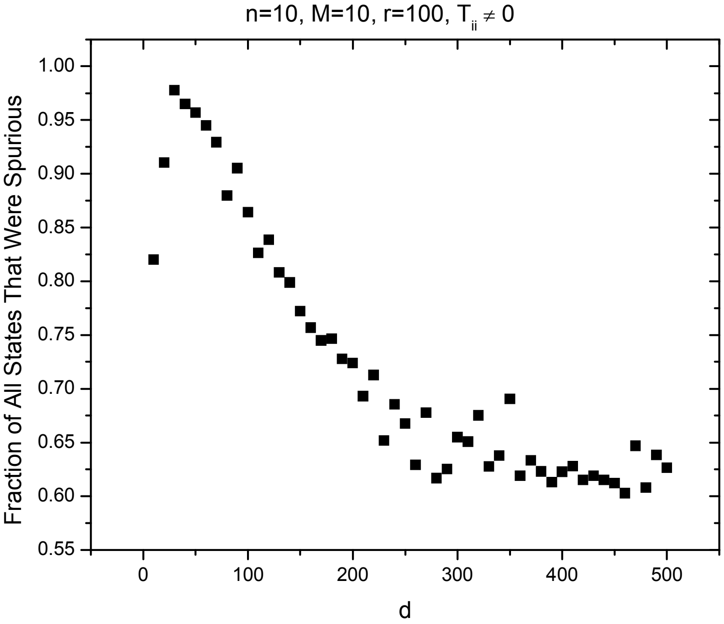

n=10, M=10, r=100 T_{ii} = 0
Fraction of all States that Were Spurious
1.0
0.9
0.8
0.7
0.6
0.5
0
100
200
300
400
500
d


n=20, M=10, r=100 T_{ii} = 0
Fraction of all States that Were Spurious
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
100
200
300
400
500
d


n=30, M=10, r=100 T_{ii} = 0
Fraction of all States that Were Spurious
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
100
200
300
400
500
d

b)


n=2, M=10, r=100, $T_{ii} \neq 0$
Fraction of All States That Were Spurious
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
100
200
300
400
500
d


n=10, M=10, r=100, $T_{ii} \neq 0$
Fraction of All States That Were Spurious
0.55
0.60
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
100
200
300
400
500
d

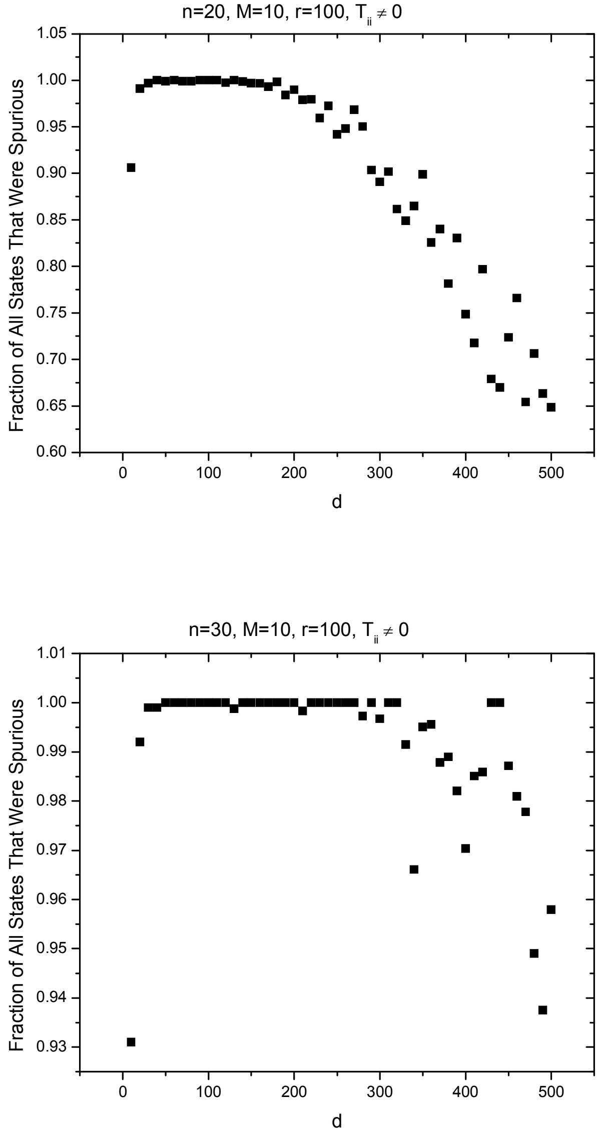


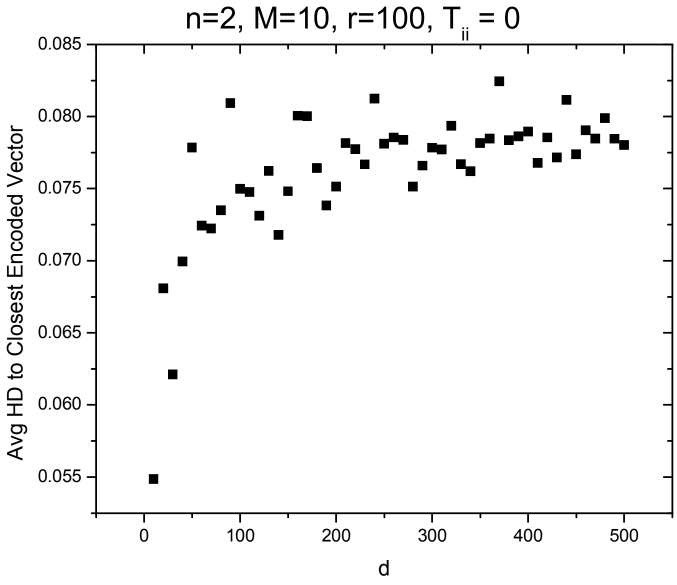
Fig. 9.2 a) Plots of the fraction of total states that are spurious as a function of vector length (d), number of vectors in the memory matrix (n), number of random vectors tried (r), and number of total memory matrices attempted (M). These plots all have the diagonal condition enforced ($T_{ii} = 0$). b) Same, but with $T_{ii} \neq 0$

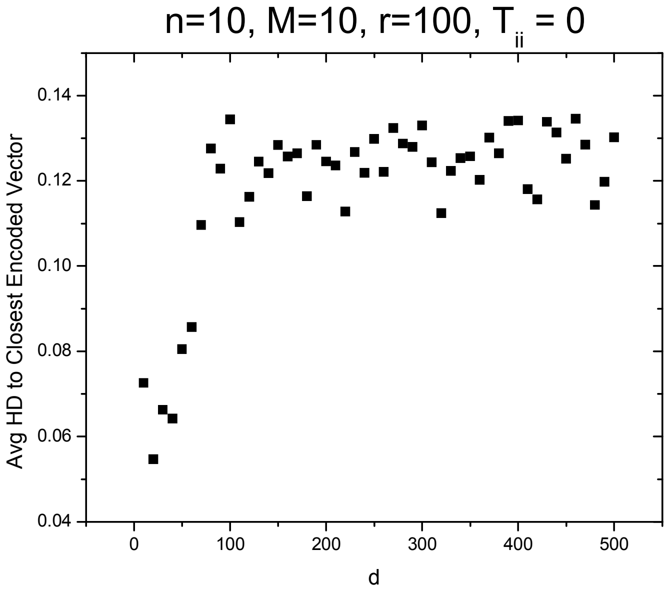
From these graphs (and others not included), I am able to conclude that the number of spurious states for a given vector length first starts to settle down to a minimum as the number of vectors in the matrix is increased. Past a certain point, however, the number of spurious states climbs quickly until almost all states in the network are spurious.

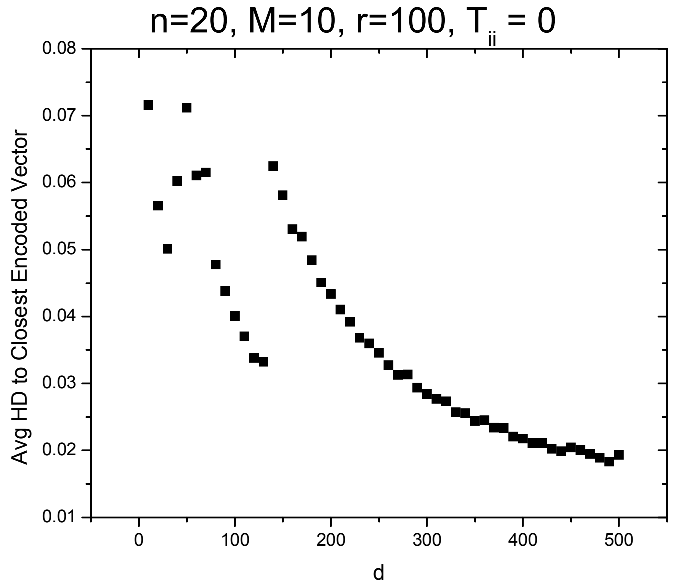
9e. Distances of Spurious States to Closest Input Vector as a Function of Vector Length and Number of Vectors Encoded in the HNN

A further quantification of the nature of spurious states can be accomplished by determining the distribution of Hamming Distances to the closest encoded memory as a function of vector length (d), number of memories encoded in the memory matrix (n), number of matrices tried (M), and number of random vectors attempted against each matrix (r). Below are plots showing this average Hamming Distance for several different numbers of vectors encoded in the memory matrix (n)

a)


n=2, M=10, r=100, T_{ii} = 0
Avg HD to Closest Encoded Vector
0.085
0.080
0.075
0.070
0.065
0.060
0.055
0
100
200
300
400
500
d


n=10, M=10, r=100, T_{ii} = 0
Avg HD to Closest Encoded Vector
0.14
0.12
0.10
0.08
0.06
0.04
0
100
200
300
400
500
d



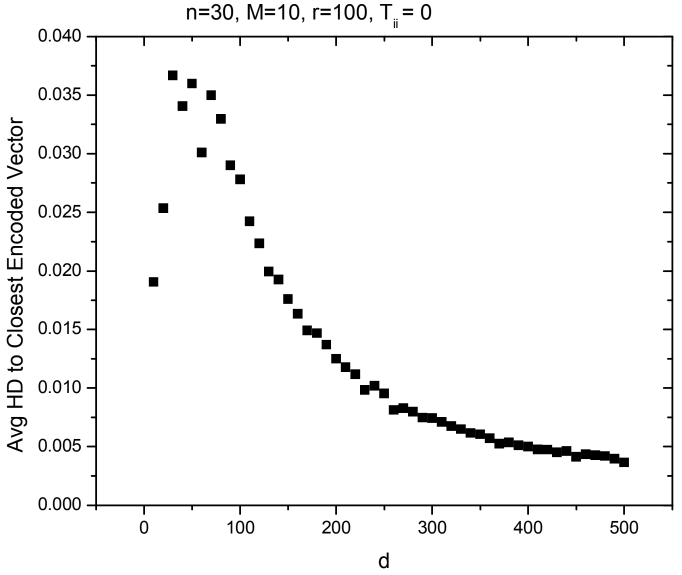
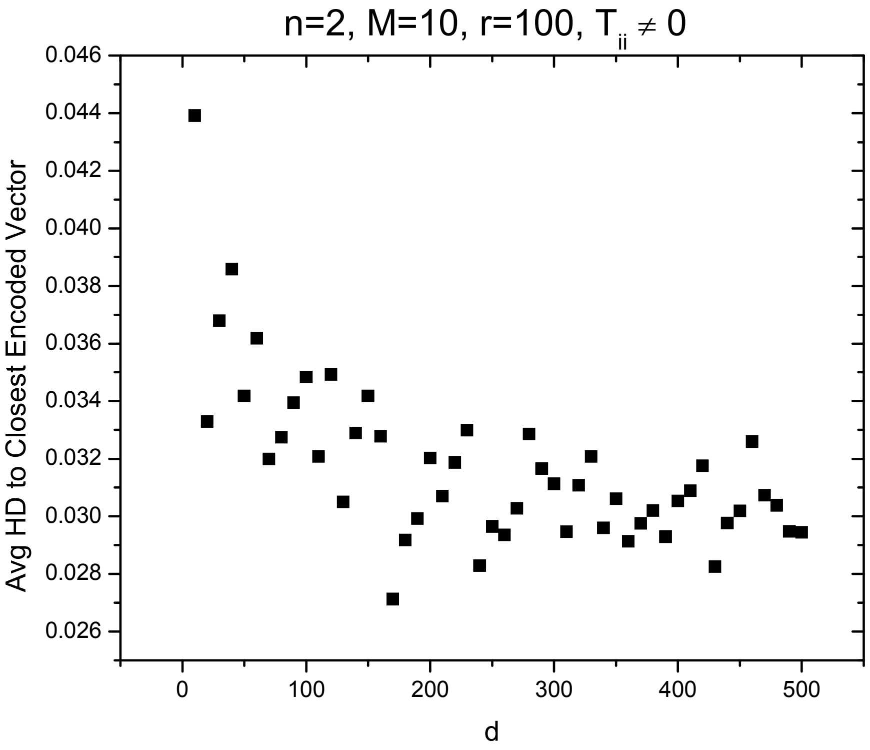
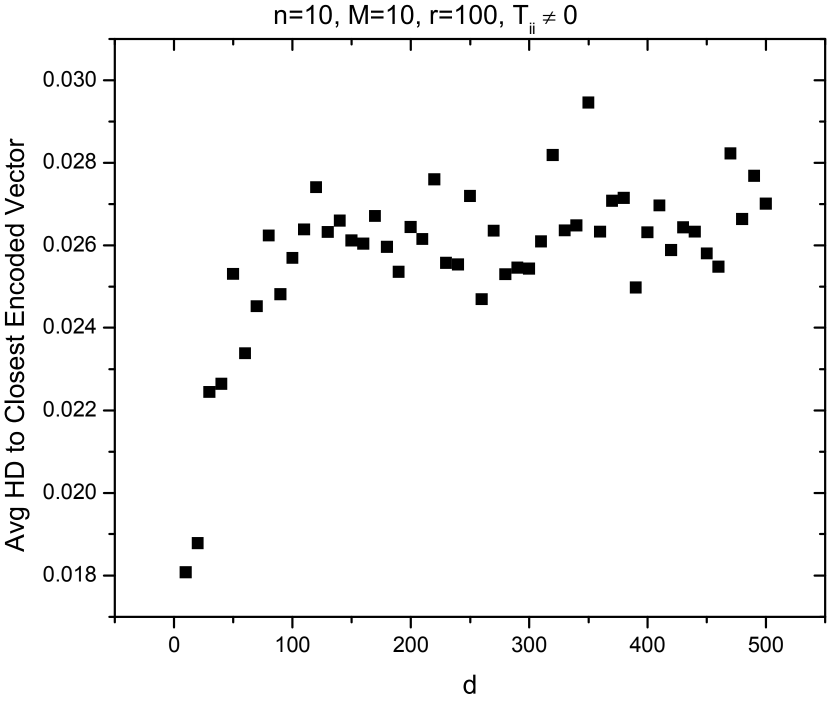


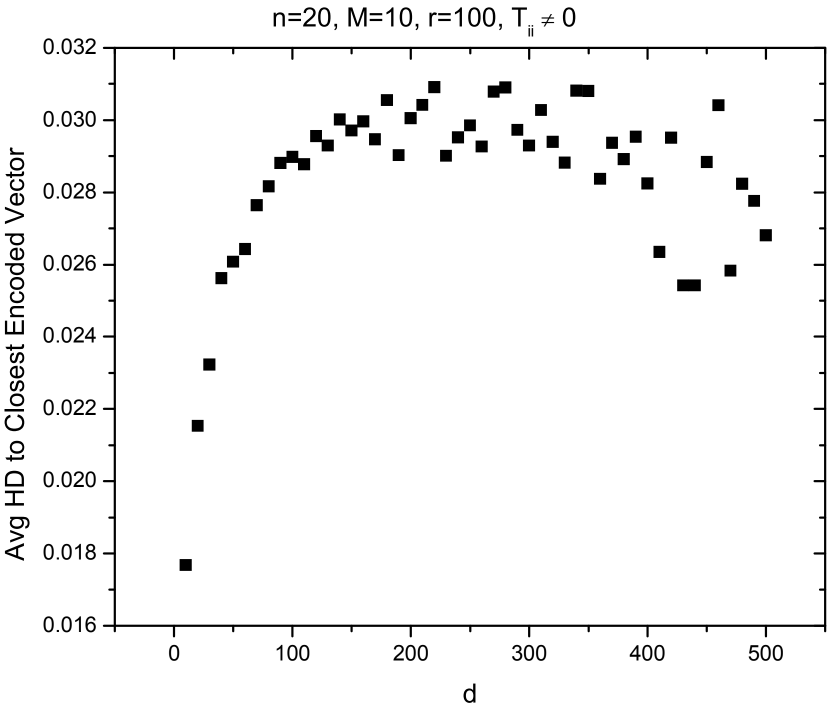
Fig. 9.3 Average Normalized Hamming Distance to the Closest Encoded Vector as a Function of Vector Length (d) and Number of Vectors Encoded in the Memory Matrix (n) for $T_{ii} = 0$

The general shape of the above graph, once established, continues as higher numbers of memories are encoded in the matrix (at least n=60), with a peak average HD to the closest encoded vector of between .035 and .040, and then tapering off as ‘d’ increases.

From the above plots it is demonstrated that as the number of vectors encoded in the memory matrix increases, the average Hamming Distance to the closest vector starts to follow a (k/d) dependence, decreasing in value as the length of the vector increases. Below are shown plots for the same situation, this time with $T_{ii} \neq 0$:




n=10, M=10, r=100, $T_{ii} \neq 0$
Avg HD to Closest Encoded Vector
0.030
0.028
0.026
0.024
0.022
0.020
0.018
d
0
100
200
300
400
500


n=20, M=10, r=100, $T_{ii} \neq 0$
Avg HD to Closest Encoded Vector
0.032
0.030
0.028
0.026
0.024
0.022
0.020
0.018
0.016
0
100
200
300
400
500
d

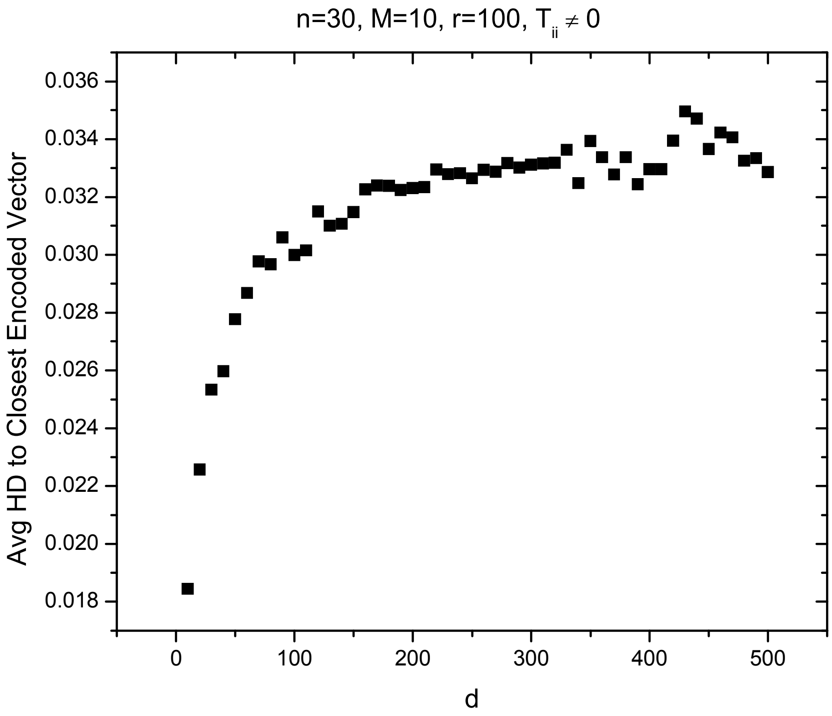


Fig. 9.4 Average Hamming Distance to the Closest Encoded Vector as a Function of Vector Length (d) and Number of Vectors Encoded in the Memory Matrix (n) for $T_{ii} \neq 0$

These plot demonstrate that when $T_{ii} \neq 0$, the distribution of Hamming Distances to the closest encoded vector as a function of 'n' is considerably different. Instead of trending toward a "k/d" behavior, the function approaches a constant value for large 'd'.

9f. Distribution of Converged and Spurious States

While the plots in the previous sections indicate the total fraction of states that were spurious for a given vector length and the Hamming Distance to the closest encoded vector, they do not indicate the distribution of spurious states as a function of iteration. To see the state into which each vector is likely to fall, a bar graph is plotted below, showing the portion of the total number of spurious states into which each iteration fell, for d=10, d=500 for n=2:

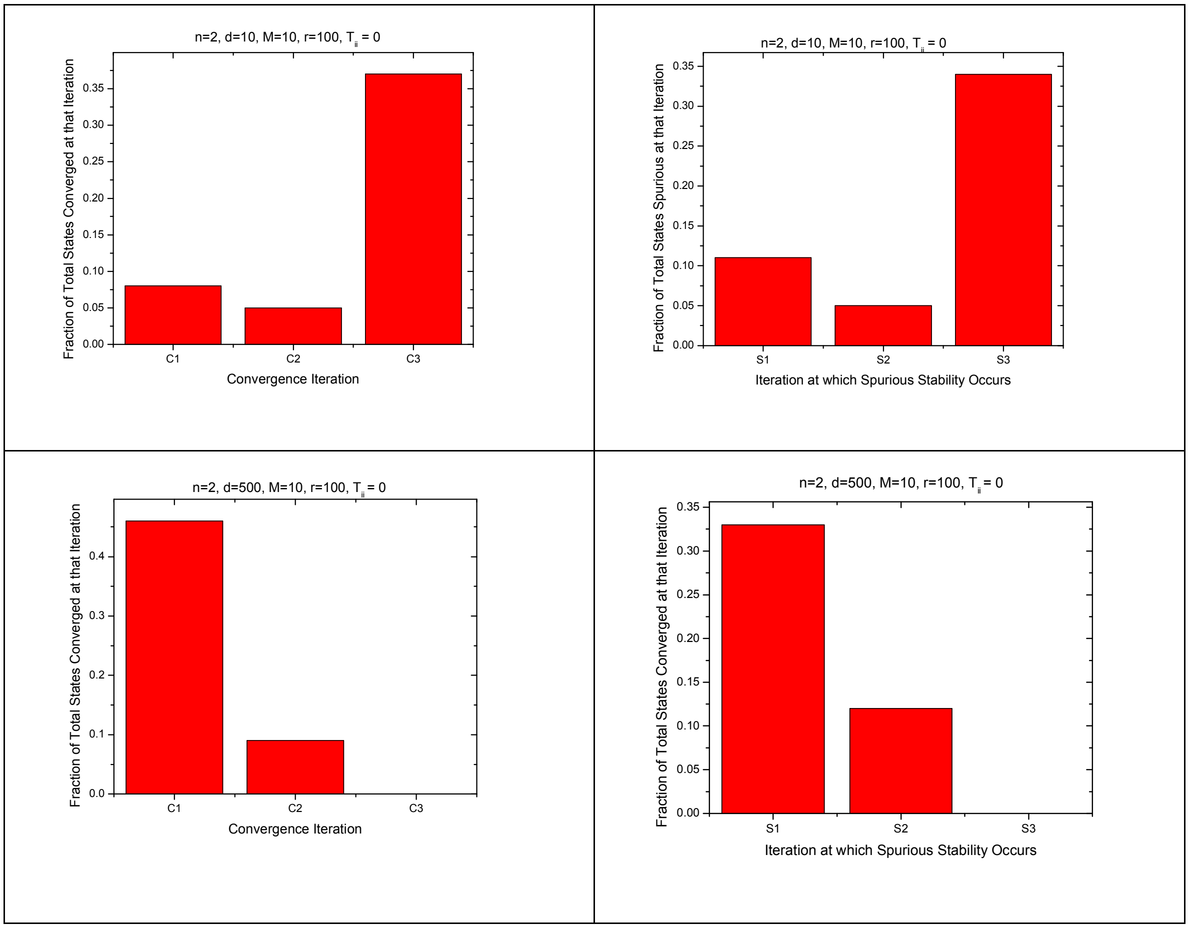


Fig. 9.5 Distribution of Spurious and Converged States for n=2, d=10 and n=2, d=500

For both the spurious states and the converged states, the fraction of input vectors converged changed in distribution. This distribution of converged vectors increased toward the first iteration as 'd' increased. This is suggestive of a possible trend and requires further investigation.

9g. Number of Vectors Converged

As the number of vectors in the system increased, or as the dimensionality of the vectors encoded increased, it began to become apparent that fewer vectors were being recalled within ten iterations. As a sample run, a set of vectors (n=10) were randomly generated, with each vector having a length of 500 (d=500). These vectors were used to construct a memory matrix by the usual sum of outer products rule, and the T_{ii} entries were left to their original values. A corrupted vector was created by giving each entry in one of the original encoded vectors a fifty percent chance of becoming corrupted. This vector was then acted upon by the memory matrix and saturated. This process was repeated, each time examining the Hamming Distance between the output number in question and its immediate predecessor to determine if the vector had settled into a stable state (either one of the original encoded vectors or a spurious memory). Results for twenty nine iterations are shown here

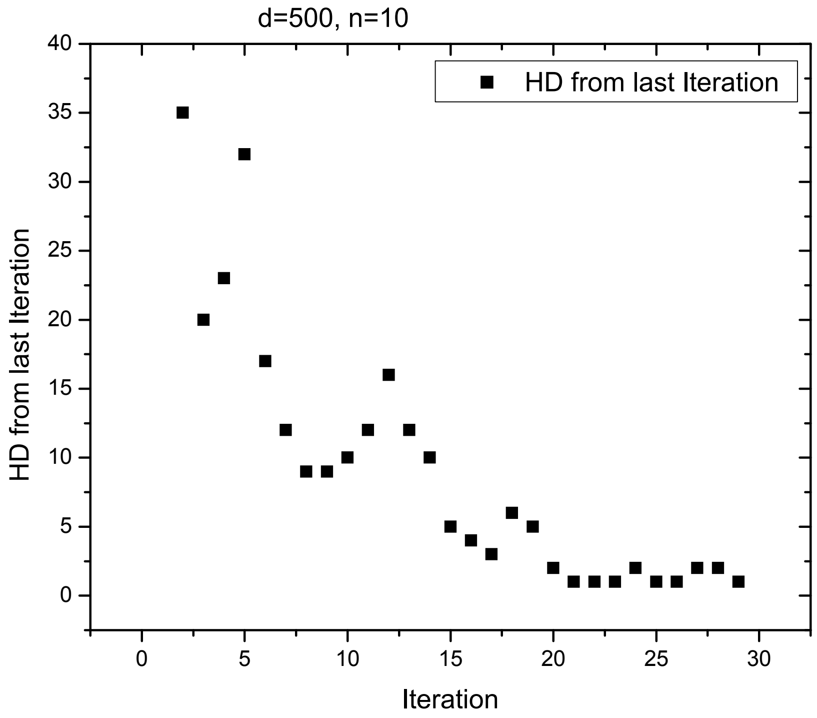
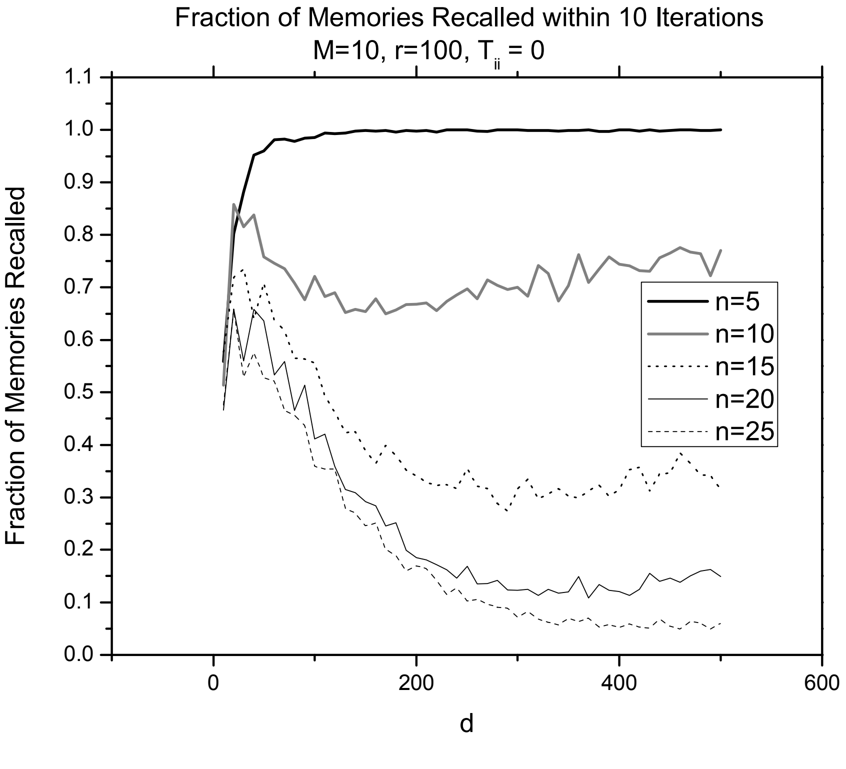


Fig. 9.6 – Hamming Distance difference between iterations for an input corrupted vector (c/d ≈ .5) as a function of iteration number.

As can be seen from the above plot, even at the twenty ninth iteration, the vector is still updating slowly, changing an entry or two at a time. It is possible this is due to the large number of updates to possible entries which will cause a further convergence. When a vector has very few entries, it has few ways to come closer to convergence. However, as 'd' increases, it may take more iterations to cause convergence of the vector to a stable state.

As a test to determine the likelihood of convergence by the tenth iteration, simulations were run for up to ten iterations (I=10) to ensure capturing the most data possible regarding final states and/or limit cycles (outputs which oscillated between multiple repeating output vector values). However, as the number of vectors in the matrix 'n' increased, I found that fewer and fewer vectors had converged either to a stable state or a spurious state by I=10. Below is a plot of the total number of vectors converged as a function of 'n' and 'd'



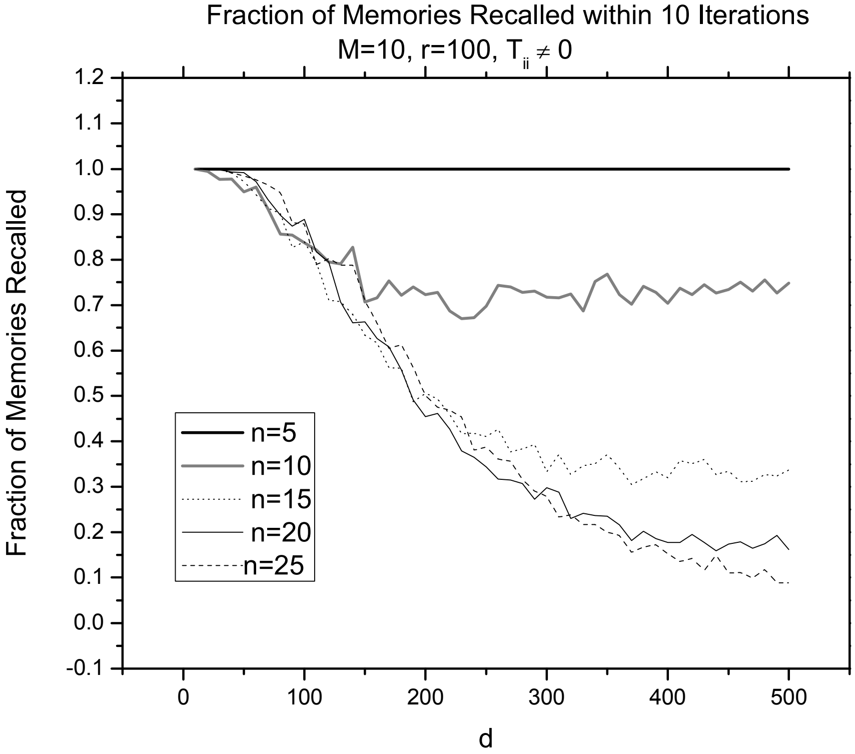


Fig. 9.7 – Fraction of Memories Recalled to Either a Spurious or Converged State within 10 Iterations as a Function of Vector Length (d) and Number of Memories Encoded in Network (n)

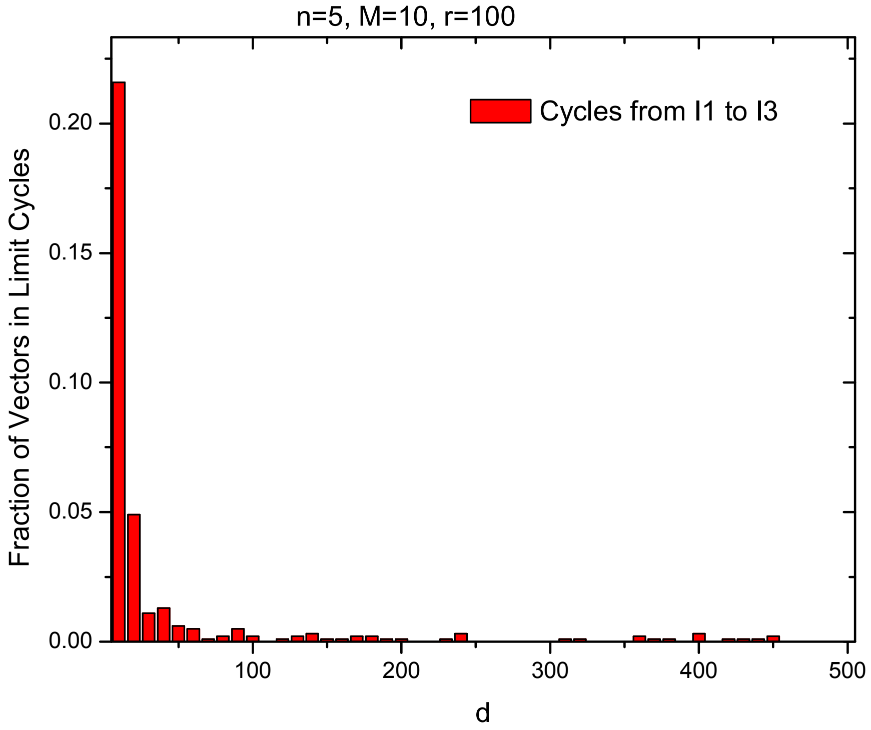
From these examples, it becomes clear that the fraction of states that converge to either a single correct or incorrect memory decreases as the number of memories encoded in the memory matrix increases.

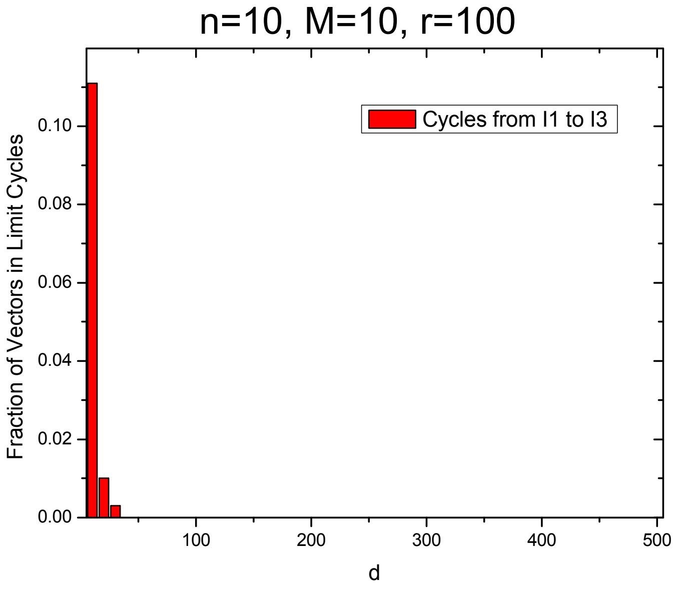
9h. Distribution of Limit Cycles

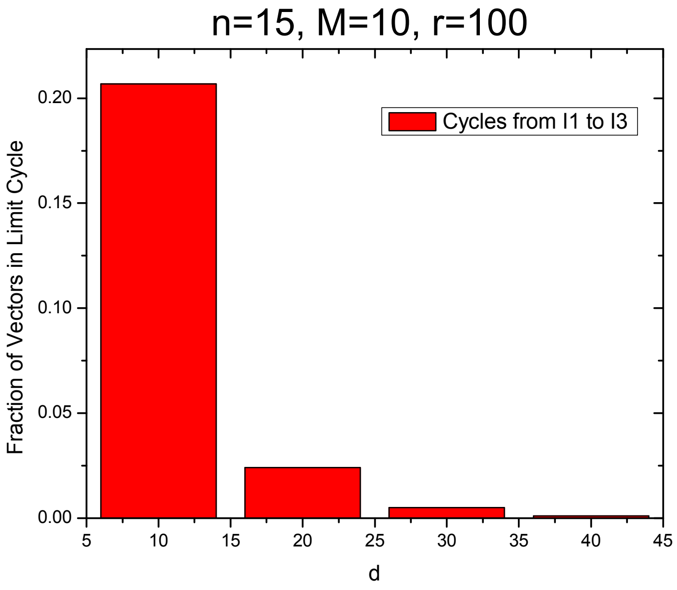
It is interesting to consider the case of "limit cycles" in the evolution of a random corrupt vector used as an input to a HNN. Grondin[87] investigated limit cycles in synchronous and asynchronous update schemes, and found that there were limits on the length of limit cycles presented by the irreversible nature of the asynchronous update scheme. In synchronous update systems (which are used here), Grondin remarked that the nonlinearity of the Heaviside function in decision making “makes it difficult to establish necessary or sufficient conditions for the existence of the solution vector.”

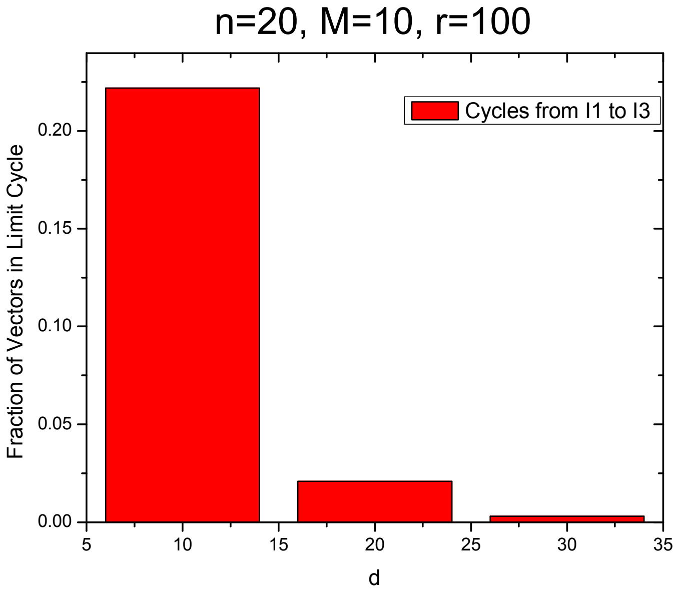
In these simulations, all vectors were able to converge to either a correct state or a spurious state when $T_{ii} \neq 0$. However, in the case where $T_{ii} = 0$, a number of limit cycles were found. These were found by tracking cases where an output vector was equal to a prior value of itself, but not the value immediately preceding the iteration under consideration. For example, if an output vector after several matrix and saturation operations creates a set of output vectors, one after each operation sets (O_1, O_2, O_3, O_4… O_n), a cycle can be determined by examining each O_i for equality. If output O_1 was not equal to O_2, but was equal to O_3, then this output has stabilized with cycle that alternates between output vectors O_1 and O_3 if O_5=O_1=O_3. With each set of iterations, the output vector was recorded and compared to previous iterations to determine if limit cycles existed. Interestingly, I found that limit cycles existed, all of length 1 (and all starting at the first output iteration). Below are plots of these cycles for each number of vectors tested (n=5,10,15,20):

n=5


n=5, M=10, r=100
Fraction of Vectors in Limit Cycles
0.00
0.05
0.10
0.15
0.20
100
200
300
400
500
d
Cycles from I1 to I3


n=10, M=10, r=100
Fraction of Vectors in Limit Cycles
0.00
0.02
0.04
0.06
0.08
0.10
100
200
300
400
500
d
Cycles from I1 to I3


n=15, M=10, r=100
Fraction of Vectors in Limit Cycle
0.00
0.05
0.10
0.15
0.20
🟥 Cycles from I1 to I3
5
10
15
20
25
30
35
40
45
d


n=20, M=10, r=100
Fraction of Vectors in Limit Cycle
0.00
0.05
0.10
0.15
0.20
🟥 Cycles from I1 to I3
5
10
15
20
25
30
35
d

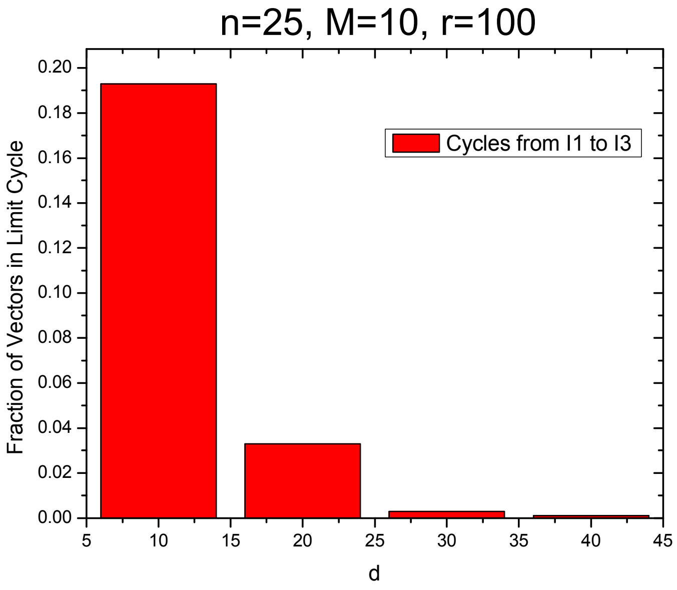


Fig. 9.8 Limit Cycles Found as a Function of Vector Length (d) for n=5, 10, 15,20,25

In each case, I note that the limit cycles all had a length of one; that is, they skipped only one iteration value. This leads to a theorem:

- No cycles with lengths longer than one are found in the output of an HNN with synchronous updating. Cycles are found only if $T_{ii} \neq 0$.

9i. Conclusion

As the number of memories stored in a HNN increases, the number of stable states (either correct memories or spurious ones) that are reached within a given number of iterations decreases. However, the number of states that are reached which are spurious first decreases as the number of vectors encoded in the HNN increases, then increases for both $T_{ii} = 0$ and $T_{ii} \neq 0$. The average Hamming Distance from the spurious state to the closest encoded vector increases as 'n' increases, then falls into a stable distribution (k/d if $T_{ii} = 0$ and approaching a constant value as 'd' increases if $T_{ii} \neq 0$). These spurious states appear to cluster closer to the first iteration as the vector length (d) increases.

Limit cycles exist in HNNs with $T_{ii} \neq 0$. I found that they all had a length of one (meaning there were two output vector values oscillating between each other). Whether there is a more quantifiable distribution of these cycles as a function of 'd' and 'n' is open to further research.

Chapter 10 – Closing Thoughts

This dissertation explored the properties of ideal Hopfield Neural Networks, as well as the possibilities of implementing a HNN in DNA, as suggested by MYP[91].

While most assessments of neural networks have been performed using corrupted vectors (vectors with some components missing), my analysis of varying properties of an ideal HNN involved assessments involving both corrupted vectors and clue vectors (vectors with some components missing). In these ideal HNNs, I found that using a synchronous update scheme is preferred over asynchronous schemes, both for their improved overall convergence, as well as smaller spread in output values. Further, relaxing the $T_{ii} = 0$ condition improves convergence slightly. While Hopfield specified that $T_{ii} = 0$ for unconditional convergence (presumably meaning convergence to one fixed endstate), relaxing this condition results in net improvement in convergence, despite the development of "limit cycles" in output results.

In contrast to these improvements, I found that varying the steepness of the saturation function did not change convergence at all, and that varying when the saturation is applied showed a very slight convergence in both clue and corrupted vector systems.

All of this information combined leads to the conclusion that for a HNN using either clue vectors or corrupted vectors as inputs, the optimum network configuration is one which uses a synchronous update scheme, with $T_{ii} \neq 0$, having a saturation function which is applied after every iteration. Further, systems using clue vectors as inputs have an overall better performance (≈ .05 - .1) at short clue/correctness lengths.

When examining the quantification and distribution of spurious states in a HNN, it was found that the number of spurious states found in a given network first decreases then increases as the number of vectors encoded in the network is increased. The average Hamming Distance from the spurious state to the closest encoded vector increases as ‘n’ increases, then falls into a stable distribution (k/d if $T_{ii} = 0$ and approaching an asymptote if $T_{ii} \neq 0$). These spurious states appear to cluster closer to the first iteration as the vector length (d) increases.

Further, there are limit cycles in HNNs when the memory matrix is not constrained to have zero diagonal elements. This result agrees with Hopfield’s assessment that T_{ii} should be equal to zero to insure unconditional convergence. However, the nature of these cycles has not been explored to date. All cycles in my simulation are all started on the first output iteration and have a length equal to one.

When attempting to implement a HNN in DNA, the ordinary rules of matrix multiplication are inadequate to describe the system output. Instead, the usual matrix and vector multiplications must be replaced by a set of Kronecker Delta functions, describing the interaction and subsequent output from DNA operations. From my analysis, I was able to determine that the information content in each memory matrix oligomer implemented in DNA is twice that of a memory matrix oligomer in an ideal HNN.

A HNN is only one possible model of a neural network, albeit an interesting one. As the first method to use a content addressable memory in a neural network, it offers new opportunities for research in both artificial intelligence and neuroscience. Many open questions remain regarding the comparability of these networks to biological structures, as well as physical design of HNN systems. By offering a small contribution to the most efficient implementation of HNNs in both ideal and DNA models, it is hoped that this dissertation has been helpful in this endeavor.

[1] Sheffer, Henry Maurice. “A Set of Five Independent Postulates for Boolean Algebras with Application to Logical Constants.” *Transactions of the American Mathematical Society* 4.14 (1913): 481-488. Print.

[2] Feynman, Richard. *The Feynman Lectures on Computation*. Reading, MA: Perseus Books, 1996. Print.

[3] Risch, Robert H. “The Solution of the Problem of Integration in Finite Terms.” *Transactions of the American Mathematical Society* 139 (1969): 167-189. Print.

[4] Minsky, Marvin. *Computation: Finite and Infinite Machines.* Englewood Cliffs, NJ: Prentice-Hall, 1967. Print

[5] Turing, A.M., “On Computable Numbers, with an Application to the Entscheidungsproblem.” *Proceedings of the London Mathematical Society* 42 (1937): 230-265. Print.

[6] Godel, K. “Uber formal unentscheidbare Satze der Principia Mathematica und verwandter System.” *I. Monatshefte fur Mathematik und Physik* 38 (1931): 173-198. Print.

[7] Peano, Giuseppe. *Selected works of Giuseppe Peano*. Trans Hubert C. Kennedy. Toronto, ON: University of Toronto Press, 1973. Print.

[8] McCulloch, Walter S. and Pitts.Walter. “A logical calculus of the ideas immanent in nervous activity.” *Bulletin of Mathematical Biophysics* 5 (1943): 115-133. Print

[9] Hebb, Donald. *The Organization of Behavior: A Neuropsychological Theory*. New York, NY: Wiley, 1949. Print.

[10] Rochester, N., et al. “Tests on a cell assembly theory of the action of the brain, using a large digital computer.” *IRE Transactions on Information Theory* IT-2 (1956): 80-93. Print.

[11] Uttley, A.M. “A theory of the mechanism of learning based on the computation of conditional probabilities.” *Proceedings of the First International Conference on Cybernetics* (1956)

[12] Taylor, W.K. “Electrical simulation of some nervous system functional activities.” *Information Theory* 3 (1956): 314-328. Print.

[13] Rosenblatt, Frank. “The Perceptron – A Probabilistic Model for Information Storage and Organization in the Brain.” *Psychological Review* 65.6 (1958): 386-408. Print

[14] Minsky, Marvin and Papert, Seymour. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969. Print.

[15] Grossberg, Stephen. “Contour enhancement, short term memory, and constancies in reverberating neural networks.” Studies in Applied Mathematics 52.3 (1973): 213-257. Print.

[16] Grossberg, Stephen. “How does a brain build a cognitive code?” *Psychological Review* 87 (1980): 1-51. Print.

[17] Minsky, M.L. and Selfridge, O.G. “Learning in Random Nets.” *Information Theory, Fourth London Symposium*, London: Butterworths

[18] Amari, S.I. “A theory of adaptive pattern classifiers.” *IEEE Transactions on Electronic Computers* EC16 (1967): 299-307. Print.

[19] Amari, S.I. “Neural theory of association and concept formation.” Biological Cybernetics 26 (1977): 175-185. Print.

[20] Rumelhart, David, Hinton, Geoffrey and Williams, Ronald. “Learning Internal Representations by Back-Propagating Errors.” *Nature* 323 (1986): 533-536. Print.

[21] Hopfield, J.J. "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences of the USA* 79.8 (1982): 2554-2558. Print.

[22] Linsker, R. “Self-organization in a perceptual network.” *Computer* 21 (1988): 105-117. Print.

[23] Broomhead, DS and Lowe, D. “Multivariable functional interpolation and adaptive networks.” *Complex Systems* 2 (1988): 321-355. Print.

[24] A.B. Hale, Web. August 20, 2009. <http://www2.cedarcrest.edu/academic/bio/hale/bioT_EID/lectures/>

[25] Haykin, Simon. *Neural Networks A Comprehensive Foundation*. Hamilton, Ontario: Prentice Hall, 1999. Print.

[26] Little, W.A. "The existence of persistent states in the brain." *Mathematical Biosciences* 19 (1974): 101-120. Print.

[27] Eberhart, Russell C. and Shi, Yuhui. *Computational Intelligence: Concepts to Implementations*. Burlington, MA: Elsevier, 2007. Print.

[28] Stubbs, D."Neurocomputers." *M.D. Computing* 5.3 (1988): 14-24. Print.

[29] Mendel, JM and McLaren RW. "Reinforcement-learning control and pattern recognition systems." *Adaptive, Learning, and Pattern Recognition Systems: Theory and Applications*. Mendel JM and Fu KS. New York: Academic Press, 1970. 287-318. Print.

[30] Zurada, Jacek. *Introduction to Artificial Neural Systems*. St. Paul, MN: West Publishing Company, 1992. Print.

[31] McClelland, TL, Rumelhart, DE et al. *Parallel Distributed Processing*. Cambridge, MA: The MIT Press 1986. Print.

[32] Widrow, B. "Generalization and Information Storage in Networks of Adaline 'Neurons'." Self-Organizing Systems. MC Jovitz, GT Jacobi, G Goldstein. Washington, DC: Spartan Books, 1962. 435-461. Print.

[33] Hopfield, JJ. "Neurons with graded response have collective computational properties like those of two-state neurons." *Proc. Natl. Acad. Sci. USA* 81 (1984): 3088-3092. Print.

[34] Serra, R. and G. Zanarini. *Complex Systems and Cognitive Processes*. Berlin: Springer-Verlag 1990. Print.

[35] Farhat, Nabil, et al. "Optical Implementation of the Hopfield Model." *Applied Optics* 24.10 (1985): 1469-1475. Print.

[36] Golomb, Solomon. "Claude Elwood Shannon." Notices of the American Mathematical Society 49.1 (2002): 1-9. Print.

[37] Forshaw, MRB, "Pattern storage and associative memory in quasi-neural networks." *Pattern Recognition Letters* 4 (1986): 185-198. Print.

[38] Peretto, P. and Niez J.J. "Long Term Memory Storage Capacity of Multiconnect Neural Networks." *Biological Cybernetics* 54.1 (1986): 53-63. Print.

[39] Keeler, J.D."Information Capacity of Outer-Product Neural Networks." *Physics Letters A* 124.1-2 (1987): 53-58. Print.

[40] Longuet-Higgins, HC, and Willshaw, DJ, and Buneman, OP. "Theories of associative recall." *Quart. Rev. Biophys.* 3 (1970): 223-244. Print.

[41] McEleiece, Robert J. et al. "The Capacity of the Hopfield Associative Memory." *IEEE Transactions on Information Theory* IT-33 (1987): 461-482. Print.

[42] Bruce, AD, Gardner, EJ, and Wallace, DJ. “Dynamics and statistical mechanics of the Hopfield model.” *J. Phys. A: Math* 20 (1987): 2909-2934. Print.

[43] Gardner, E.”Structure of metastable states in the Hopfield model.” *J. Phys. A: Math Gen* 19.16 (1986): 1047-1052. Print.

[44] Hopfield, JJ, Feinstein, DI and Palmer, RG. “Unlearning has a stabilizing effect in collective memories.” *Nature* 304 (1983): 148-149. Print.

[45] Potter, TW, Dissertation (State University of New York, Binghamton) (1987)

[46] Kleinfeld, D and Pendergraft, DB. “’Unlearning’ increases the storage capacity of content addressable memories.” *Biophys J.* 51 (1987): 47-53. Print.

[47] Abu-Mostafa, Y.S. and St. Jacques, J.M. “Information Capacity of the Hopfield Model.” *IEEE Transactions on Information Theory* IT-31.4 (1985): 461-465. Print.

[48] Horn, D. and Weyers, J. “Information packing in associative memory models.” *Physical Review A.* 34.3 (1986): 2324-2328. Print.

[49] Gindi, Gene R., Gmitro, Arthur F. and Parthasarathy, K. “Hopfield model associative memory with nonzero-diagonal terms in memory matrix.” *Applied Optics* 27.1 (1988): 129-134. Print.

[50] Gmitro, Arthur F, Keller, Paul E. and Gindi, Gene R. “Statistical performance of outer-product associative memory models.” *Applied Optics* 28.10 (1989): 1940-1948. Print.

[51] DeWilde, P. “The Magnitude of the Diagonal Elements in Neural Networks.” *Neural Networks* 10.3 (1997): 499-504. Print.

[52] Eccles JC *The physiology of synapses*. Berlin: Springer-Verlag, 1964. Print.

[53] Chengxiang, Zhang, Dasgupta, Chandan, and Singh, Manoranjan. “Retrieval Properties of a Hopfield Model with Random Asymmetric Interactions.” *Neural Computation* 12 (2000): 865-880. Print.

[54] Chen, Tianping and Amari, Ichi. “Stablility of Asymmetric Hopfield Networks.” *IEEE Transactions on Neural Networks* 12.1 (2001): 159-163. Print.

[55] Zhao, Hong. “Designing asymmetric neural networks with associative memory.” *Physical Review E* 70 (2004): 066137-1 – 066137-4. Print.

[56] Zheng, Pernsheng, Zhang, Jianxiong and Tang, Wansheng. “Analysis and design of asymmetric Hopfield networks with discrete-time dynamics.” *Biological Cybernetics* 103 (2010): 79-85. Print.

[57] Dotsenko, V. “Hierarchical Model of Memory.” *Physica* 140A (1986): 410–415. Print.

[58] Dotsendo, V.. “Hierarchical Multilayer Model of Memory.” *Pis’ma Zh. Eksp. Teor. Fiz.* 44.3 (1986): 151-153. Print.

[59] Cortes, C., Krogh, A, and Hertz, JA. “Hierarchical associative networks*.” J. Phys A: Math Gen.* 20 (1987): 4449-4455. Print.

[60] Parisi, G. “Infinite Number of Order Parameters for Spin-Glasses.” *Phys. Rev. Lett*. 43 (1979): 1754-1756. Print.

[61] Amit, DJ, Gutfreund, H and Sompolinsky, H. “Information storage in neural networks with low levels of activity.” *Physical Review A* 35.5 (1987) 2293-2303. Print.

[62] Sherrington, David and Kirkpatrick, Scott. “Solvable Model of a Spin-Glass.” Physical Review Letters 35.26 (1975): 1792-1796. Print.

[63] Petsche, T. *Topics in Neural Networks*. Diss. Princeton University, 1988. New Jersey. Print.

[64] Adleman, Leonard M. “Molecular Computation of Solutions to Combinatorial Problems.” *Science* 266 (1994): 1021-1024. Print.

[65] Garey, M.R. and Johnson, D.S. *Computers and Intractability*. San Francisco: Freeman, 1979. Print.

[66] Karp, Richard. *Complexity of Computer Computations*. New York: Plenum Press, 1972. Print.

[67] Lipton, Richard J. “DNA Solution of Hard Computational Problems.” *Science* 268 (1995): 542-545. Print.

[68] Winfree, Erik, et al. “Design and self-assembly of two-dimensional DNA crystals.” *Nature* 394 (1998): 539-544. Print.

[69] Grunbaum, Branko and Shephard, G.C. *Tilings and Patterns*. New York: Freeman, 1986. Print.

[70] Fu, Tsu-Ju and Seeman, Nadrian. “DNA Double-Crossover Molecules.” *Biochemistry* 32 (1993): 3211-3220. Print.

[71] Karp, Richard. *Reducibility Among Combinatorial Problems*. Berkeley, CA: University of California, 1972. Print.

[72] Garey, M.R. and Johnson, D.S. *Computers and Intractability*. San Francisco: Freeman, 1979. Print.

[73] Ouyang, Qi., et al. “DNA Solution of the Maximal Clique Problem.” *Science* 278 (1997): 446-449. Print.

[74] Boneh, Dan. et al. “On the Computational Power of DNA.” *Discrete Applied Mathematics* 71 (1996): 79-94. Print.

[75] Deaton, R. et al. “Reliability and Efficiency of a DNA-based Computation.” *Physical Review Letters* 80.2 (1998): 417-420. Print.

[76] Tindall, KR and Kunkel, TA. “Fidelity of DNA Synthesis by the Thermus Aquaticus DNA Polymerase.” *Biochemistry* 27 (1988): 6008-6013. Print.

[77] Baum, Eric. “Building an Associative Memory Vastly Larger Than The Brain, Science.” *Science* 268 (1995): 583-585. Print.

[78] Oliver, John. “Matrix Multiplication with DNA.” *Journal of Molecular Evolution* 45 (1997): 161 – 167. Print.

[79] Kim, KH . *Boolean matrix theory and applications*. New York: Marcel Dekker, 1982. Print.

[80] Robinson DF and Foulds LR. *Digraphs: theory and techniques*. New York: Gordon and Breach, Science Publishers, Inc., 1980. Print.

[81] Nielson, F. *Applications of Graph Theory*. London: Academic Press Inc, 1979. Print.

[82] Marom, Emanuel. "Associative Memory Neural Networks with Concatenated Vectors and Nonzero Diagonal Terms." *Neural Networks* 3 (1990): 311-318. Print.

[83] Macukow, Bohdan and Arsenault, Henri H. "Modification of the threshold condition for a content-addressable memory based on the Hopfield model."*Applied Optics* 26.1 (1987): 34-36. Print.

[84] Amari, SI. "Characteristics of randomly connected threshold element networks and network systems." *Proc. IEEE* 59 (1971): 35-47. Print.

[85] Amari, SI. "Learning patterns and pattern sequences by self-organizing nets of threshold elements." *IEEE Trans. Comput*. C-21 (1972): 1197-1206. Print.

[86] Amari, SI. "A method of statistical neurodynamics." *Kybernetik* 14 (1974): 201-215. Print.

[87] Grondin, R.O. et al. "Synchronous and Asynchronous Systems of Threshold Elements." *Biological Cybernetics* 49 (1983): 1-7. Print.

[88] Cheun, KF, Atlas, LE and Marks, RJ II. "Synchronous vs. asynchronous behavior of Hopfield's CAM neural net." *Applied Optics* 26.22 (1987): 4808-4813. Print.

[89] Mills, Allen. Personal Interview. July 2010.

[90] Mills, AP Jr. "Experimental Aspects of DNA Neural Network Computation." *Soft Computing* 5.1 (2001): 10-18. Print.

[91] Mills, AP Jr., Yurke, B. and Platzman, P. "DNA Analog Vector Algebra and Physical Constraints on Large-Scale DNA-based Neural Network Computation." *DIMACS Series in Discrete Mathematics* 54 (2000): 65-73. Print.

[92] Walker, GT. "Isothermal Invitro Amplification of DNA by a Restriction Enzyme DNA-Polymerase System." *Proceedings of the National Academy of Sciences* 89 (1992): 392-396. Print.

[93] Akin, HE. "Electronic Microarrays in DNA Computing." *Journal of Nanoscience and Nanotechnology* 10 (2010): 1-7. Print.

[94] Karabay, D, Hughes, BST., Mills, AP Jr. "Experimental Implementation of a Hopfield Neural Network using DNA Molecules." Not Yet Published.