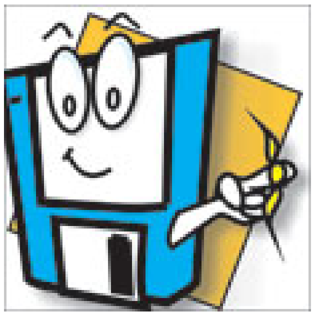
Crossroads
The ACM Student Magazine



# Associative Memory and the Board Game Quarto

by ***Zachary A. Kissel***

## Introduction

**Neural networking** or **Connectionist Theory** dates back to 1943 with the researchers Warren S. McCulloch and William Pitts [5]. This article investigates the basic structure of neural networks and how they relate to associative memory. The article will also study the use of associative memory in construction of a competent computer opponent for the game Quarto. The computer opponent is trained using a modification of the classical Hebbian learning algorithm.

## Associative Memory

**Associative memory** is the ability to recall facts based on description. For example, when asked to name a band that was part of the "British Invasion" and named after an insect, most people name *The Beatles* [1]. The process used to determine this fact from the given clues is known as **association**.

Association is a common phenomenon that humans usually overlook. However, most of our day-to-day living relies heavily on associative memory. The question that needs to be explored regards how associative memory concepts can be used in a computer. Associative memory is used extensively in recognition programs to perform such tasks as handwriting recognition and image recognition.

## Neural Network Basics

In order to understand associative memory, as it exists in the computer, the basics of neural networks must be understood. Neural networks were designed to mimic the mechanisms of the neurons in the human brain. Biological neurons are made up of dendrites, the soma or cell body, an axon, a terminal button, a synaptic gap, and a synapse or point of contact with another neuron (**Figure 1**).

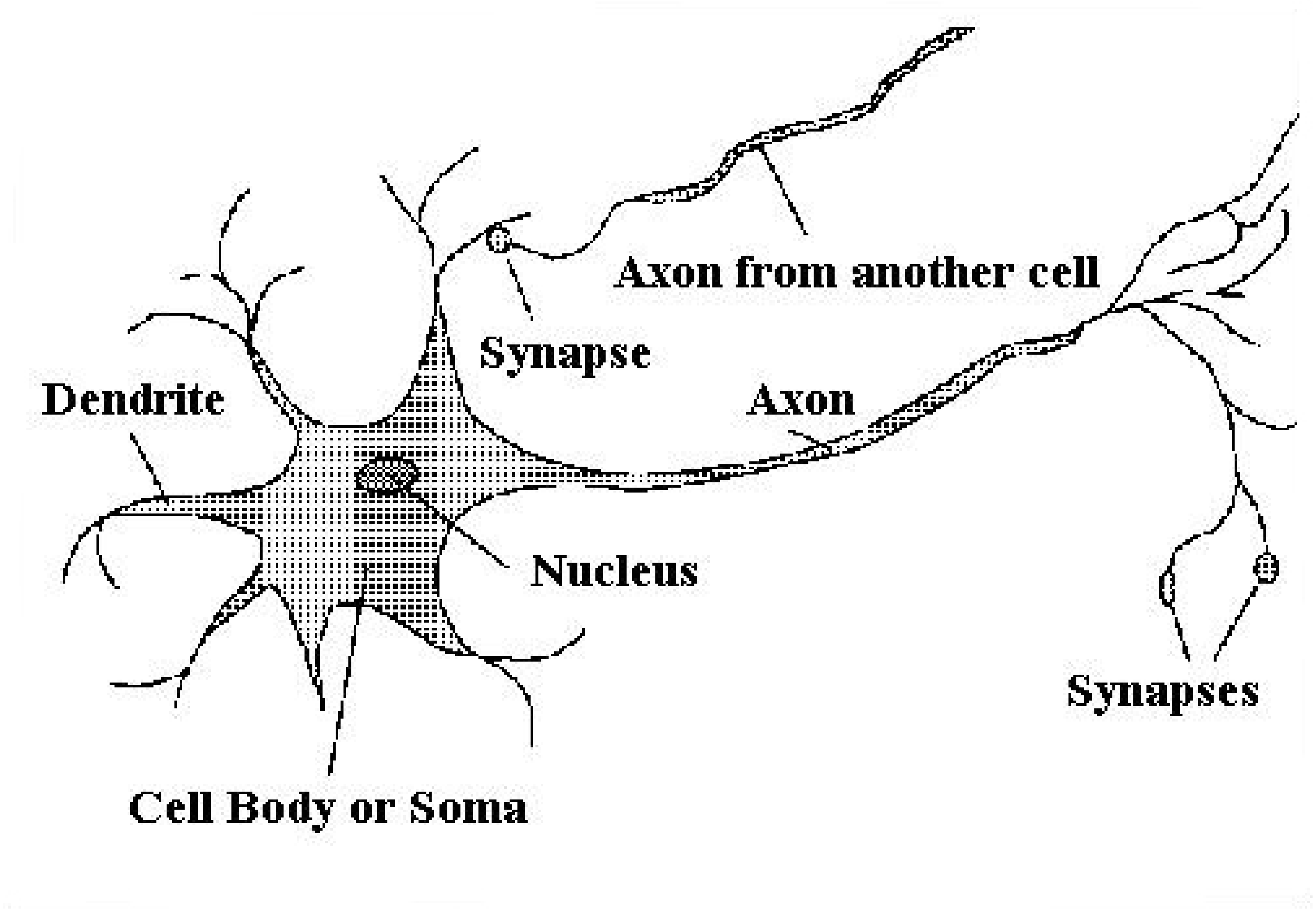


**Figure 1:** A Neuron [2].

A neuron operates through a series of electrochemical pulses. The process is as follows: an electrical pulse is passed into the neuron through the synaptic gap to either the dendrites or the soma. This happens until enough of the electricity amasses in order to overcome what is known as the **threshold** at which the neuron fires.

**Firing** occurs when an electrical pulse is released down the axon through the terminal button toward the synaptic gap, allowing for a message to propagate to subsequent neurons [6]. When this structure is implemented in a computer, the basic neuron structure is simplified. In the computer representation there is no synaptic gap, and the terminal buttons are connected directly to the soma (**Figure 2**).

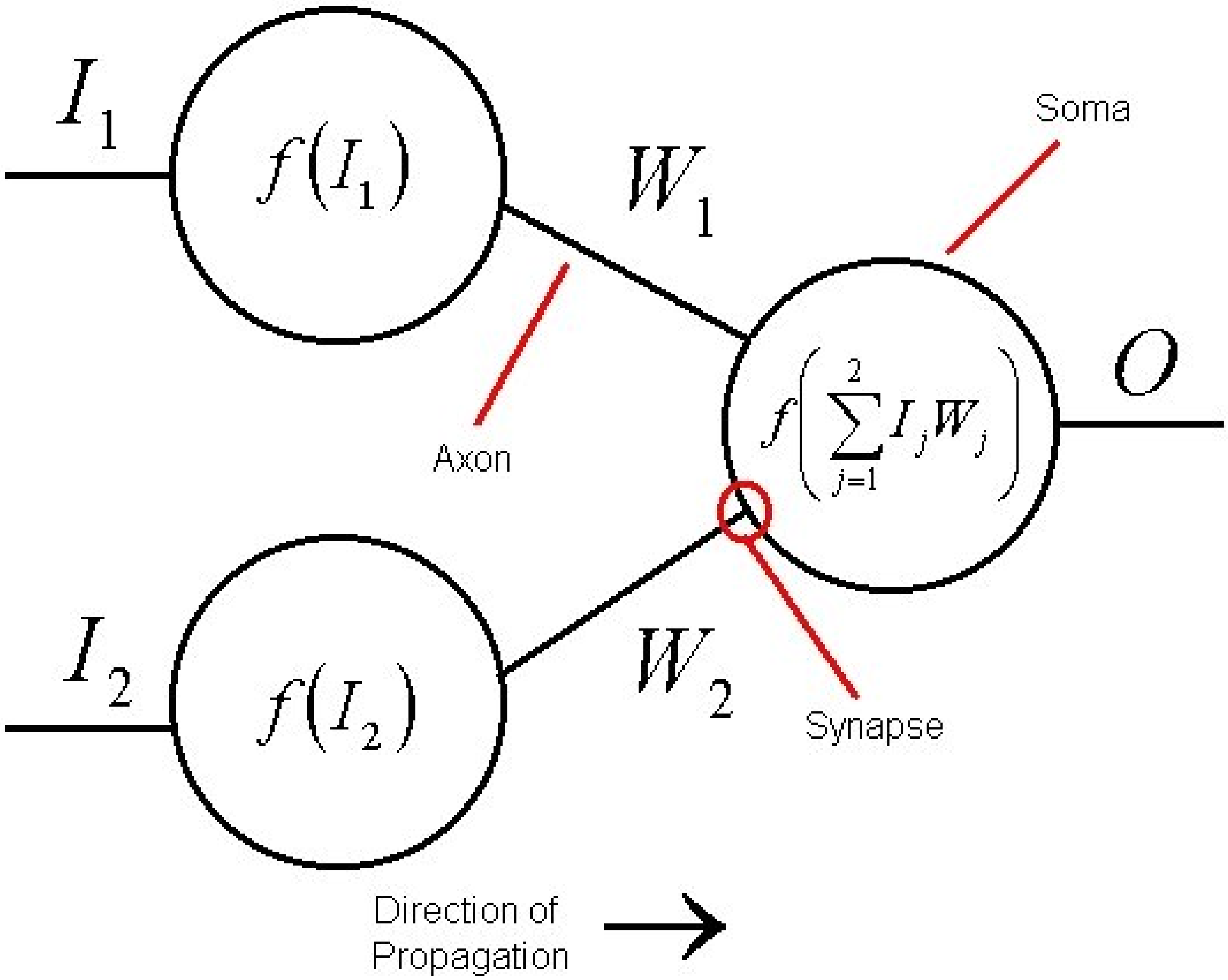


**Figure 2:** A neural network where *f* is some activation function and $W_1$ and $W_2$ are the connection weights.

Neural networks are represented in computers as weighted directed graphs in which the input is entered at the extreme left through the input neurons and propagated through the network toward the right. The sum of all the input weights into a neuron multiplied by the values of its previous neighbors is passed to a firing function, known as the **activation function**. The result of this function then determines the firing of the current neuron.

The weights along the edges of the axons in the neural networks are what allow the network to learn and retain information. Learning occurs by allowing the connection between more common nodes to amass greater strength in the overall network. In order for the computer to learn on its own accord, there needs to be an algorithm to initiate learning. There are many algorithms for training neural networks. One such algorithm is known as the **Hebbian learning** algorithm, and is popular with associative memory experiments.

As an example, consider the AND function from Boolean algebra. This function evaluates to true only when both inputs are true, and false with any other combination of inputs. In order to build a network to solve this problem, the number of input nodes and output nodes must be defined. For this problem there will be two input nodes and one output node, resembling the network in **Figure 2**. Next, the set of values for the input must be defined. This function logically lends itself to **binary encoding**, data encoded as zeros and ones. Finally, the activation function must be defined. For a function as elementary as this, a simple **step function** can be used. The step function is defined as:

```
Step(x)
  if(x=2)
    return(1)
  else
    return(0)
```

The threshold in this step function is set to two because both inputs have to be one for proper firing in the network. In this network, the weights are set to one because a simple activation function can be used to describe the range of the network's operation. It should be noted that there is not a single unique neural network to solve a certain problem. Weights may be described in a plethora of ways. Since the AND function has a binary definition, it is logical to use the aforementioned binary encoding.

## Hebbian Learning

In 1949 a psychologist by the name of Donald Hebb proposed [3]:

> When an axon of cell A is near enough to excite B and repeatedly or persistently takes part 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.

Hebb stated that when one neuron repeatedly causes another neuron to fire, the strength of the connections between the two neurons is increased. This is the postulate that is utilized in the Hebbian learning algorithm.

The Hebbian learning algorithm is traditionally executed on a given **network architecture**, or way in which the network is set up, satisfying the following prerequisites: The network must have the same number of input nodes as output nodes, and every input node must have a connection to every output node. Unlike other neural network architectures, networks trained using the Hebbian learning algorithm do not have any **hidden layers** or levels of network architecture between the input layer and the output layer.

Considering the architecture that has been described, a learning algorithm can be defined. Note that the strength of the connections between the two neurons in a Neural Network is represented by the weights on the edges of the graph. Recall that Hebb theorized that when one neuron causes the firing of another neuron, the strength of the connection between the two neurons is increased. The algorithm is as follows:

```
for i = 0 to Number_of_Input_Nodes
 for j = 0 to Number_of_Output_Nodes
 weight[i][j] = weight[i][j] + gamma * Input[i] * Output[j]
```

where `weight[i][j]` is the weight of the connection between input node i and output node j, `gamma` is the **learning constant**, which is used to keep connections in the network from becoming too strong too quickly, and `Input[i]` is to be associated with `Output[j]`.

The activation function of a Hebbian network is normally a function known as the **bipolar step** or **sign** function. This function is defined as follows:

```
Sign(x)
  if(X<0)
    return (-1)
  else
    return(1)
```

This means that when the network is trained, the network must encode the data using either positive one or negative one, which is known as **bipolar encoding**. Bipolar encoding is a major handicap of this algorithm because not everything that a neural network may be used to recognize can be expressed in a simple bipolar answer. Sometimes a **fuzzy number**, a number on the closed interval [0,1], is used to represent the value of an answer. A viable solution to this handicap is introduced in the section *Construction of a Competent Opponent*.

## Quarto

The game Quarto is nothing more than a slightly more complex version of Tic-Tac-Toe. The game is played on a four by four matrix with sixteen distinct pieces. To win the game, a player must have four pieces in a row that all share the same quality. There exist a total of 322,560 possible winning combinations.

The interesting complexity in the game Quarto is that a player is not allowed to select his or her own piece. Their opponent must select the piece that must be placed on the board. A more complete listing of the rules and other information may be found at the publishers' website [7].

## Construction of a Competent Opponent

The goal of this project was to develop a program that could learn to play the game Quarto after only being shown one turn of the game, consisting of what the board looked like both before and after a piece was played and distributed. The general flow of the program is the following:

1. The computer will initially give the player a piece which to place on the board.
2. The player will select a piece to give to the computer.
3. The computer will be given a piece which to place. This piece along with the current information about the board will be fed through the neural network, and the piece will be placed.
4. The computer will select a piece to give to the player by feeding the current information about the board through a neural network.

This program uses the concept of a **priority board**, which is generated by the neural network responsible for placing a piece on the board. A priority board is an identically sized matrix that is used to hold values that determine how advantageous a certain spot is on the game board [4]. In this case, the numbers turn out to be fuzzy numbers.

The program used a slightly modified version of this algorithm. The algorithm used was still the same greedy algorithm, in which the square with the highest value is selected. Given that there was often a tie for the best piece placement, an alteration had to be made. The alteration made was that the first highest valued square was selected and the computer placed a piece in that location.

First, we develop a method to encode game-play information and utilize the proven effectiveness of a priority board. Previously, the intelligence was composed of two independent neural networks. One network was responsible for giving a piece to the player and the other network was responsible for placing a piece, given to the computer. Each input node and output node in the piece-placing network represented one space on the game board. The piece-placing network also contained an extra input node to store the piece to be placed. The piece-giving network has input nodes that represent each place on the board and output nodes that represent one of the possible pieces.

The deviance from the general standard was both in the architecture of the networks and the activation function itself. Instead of using the traditional activation function, a less commonly used function, the **sigmoid function** was employed:

`Sigmoid(x) = (1/(1+e^(-alpha*x)))`, where alpha is some constant

Using a Sigmoidal Activation Function was advantageous because it allowed for the compression of the architecture. Though the ideal state for Hebbian learning would have an equal number of input nodes and output nodes, our architecture created a satisfactory state because it allowed for an almost one-to-one mapping of the input nodes to the output nodes. The original network had difficulties learning because there existed significantly more input nodes than output nodes in the network responsible for placing a piece. The imbalance that existed in the original architecture was due to the bipolar encoding of the input information, which was needed because of the bipolar step activation function. By using the sigmoid activation function with an alpha of 0.5 and an input encoding consisting of the closed set [0-16], the network responsible for placing a piece on the board was reduced to seventeen input nodes and sixteen output nodes and the network responsible for giving a piece to the player remained fixed at sixteen input nodes and sixteen output nodes. By providing a more orthogonal architecture due to the linear encoding, training became less of a concern.

## Training

Before the training can be discussed in depth, the structure of the networks needs to be discussed. Every piece in the game board was assigned a numeric code between one and sixteen, one value for each of the sixteen pieces. A value of zero was assigned to the empty spaces. For the piece-placing network, the first sixteen input nodes were used for the current setup of the board, with each input node representing a single place on the board. The remaining input node was used to hold the numerically encoded piece that needed to be placed. There are sixteen output nodes, one for each place on the board. In the training process, each of these nodes was assigned a bipolar value: negative one signified not to place the current piece at that spot, while positive one signified to place the current piece at a spot.

The network responsible for giving a piece used a slightly different approach. There were sixteen input nodes and sixteen output nodes. The sixteen input nodes were the sixteen positions on the board. The pieces used the same encoding for the input as the piece-placing network. These encodings were then associated with an output using a number from the set {-1,0,1}. Negative one signified that it was a bad piece to give, zero signified there was a blank, and positive one denoted the piece to give.

The two networks were first trained by giving the computer a complete turn from a human versus human game to memorize. The rest of the training happened during game-play. In order to train during game-play, the network would be told which turn it should memorize. For example, if the game was set to train on the third move, the information about the current move would be encoded and the appropriate network would be trained.

All of the networks were trained using a learning constant (gamma) of 0.451. The learning constant is a variable that has to be adjusted, because there are no fixed rules for determining the value and a better value may exist. The neural networks were then put into the environment of normal game-play and trained on one specific turn per game, with the player's piece-giving move and the player's piece-placing move encoded and trained in the respective networks. There was only one opening move trained and several middle of the game moves trained. The network was trained with a very limited number of inputs in order to help reduce the chances of **statistical overfitting**. Statistical overfitting is a common problem in neural networks where the network memorizes irrelevant information [6].

## Results

The computer has a fairly strong opening game and a very aggressive endgame using the learning constant of 0.451. The networks were able to generalize from trained data to an acceptable performance in game-play. However, adjusting the gamma value will allow the network to generalize to a higher level of play. The networks were able to determine strong positions on the board independently, discovering that winning often comes to those who control the corners of the board, a strategy that was not implemented during training. The networks, as trained, only won thirty percent of the games played, based on a sample of twenty games. Though the number of wins is low, some of this lends itself to faults in training, since the network was trained with both my own strengths and weaknesses in game-play. There are many ways in which these networks can be improved. One way in which to improve the performance of these networks is to look more into adjusting the learning constant. Another way to improve the networks is to graph the data to more accurately see if statistical overfitting is occurring. These and any additional paths that may result in better game-play in the future should be thoroughly explored.

Quarto is but one computer game that can be explored using associative memory. This technology can easily be expanded to related games, such as Tic-Tac-Toe, and other, more complex games. As more of the inner workings of the brain is learned, associative memory in the computing world will change proportionally in order to construct more accurate and powerful models.

## References

**1**
Flake, G. W., *The Computational Beauty of Nature*. MIT Press, Cambridge, MA, 1999 p. 312.

**2**
Kendall, G., "Neural Networks," *Introduction to Artificial Intelligence*, 21 September 2001, < **http://www.cs.nott.ac.uk/~gxk/courses/g5aiai/006neuralnetworks/neural-networks.htm**>.

**3**
Klien, R. M., *The Hebb Legacy*, <**http://www.psych.ualberta.ca/~bbcs99/hebb%20legacy.html** >.

**4**
Matthews, J. "Simple Board Game AI," *Generation5*, 16 January 2000, <**http://www.generation5.org/boardai.shtml** >.

**5**

The Mind Project, *A Computer Model of the Neuron: The McCulloch and Pitts Neuron*, 2000-2003, <**http://www.mind.ilstu.edu/curriculum/perception/mpneuron1.html** >.

**6**

Rojas, R. *Neural Networks: A Systematic Introduction*. Springer-Verlag, New York, NY, 1996. pp. 18-19, 144.

**7**

Gigamic Games, *Quarto!*, <**http://www.gigamic.com/regles/anglais/rquartoe.htm**>.

---

## Biography

Zach Kissel (**kisselz@merrimack.edu**) is a Computer Science major and Mathematics major at Merrimack College. His main Computer Science interests lie in Artificial Intelligence, specifically in the field of Bio-Computation. In his spare time he enjoys mountain biking and running.