5-2002

Synesthetic Sensor Fusion via a Cross-Wired Artificial Neural Network.

Stephen Samuel Seneker
East Tennessee State University

Synesthetic Sensor Fusion Via A Cross-Wired Artificial Neural Network

---

A thesis
presented to
the faculty of the Department of Liberal Studies
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Master of Arts in Liberal Studies

---

by
Stephen S. Seneker
May 2002

Keywords: Synesthesia, Sensor Fusion, Robotics, Neural Networks

ABSTRACT

Synesthetic Sensor Fusion Via A Cross-Wired Artificial Neural Network

by

Stephen S. Seneker

The purpose of this interdisciplinary study was to examine the behavior of two artificial neural networks cross-wired based on the synesthesia cross-wiring hypothesis. Motivation for the study was derived from the study of psychology, robotics, and artificial neural networks, with perceivable application in the domain of mobile autonomous robotics where sensor fusion is a current research topic. This model of synesthetic sensor fusion does not exhibit synesthetic responses. However, it was observed that cross-wiring two independent networks does not change the functionality of the individual networks but allows the inputs to one network to partially determine the outputs of the other network in some cases. Specifically, there are measurable influences of network A on network B, and yet network B retains its ability to respond independently.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

A human being perceives and recognizes an environment through the use of many sources of sensory information. The integration of sensory information can be complimentary, enhancing the response of one sense in response to another sense, or it can compensate for a deficiency in a sense. Robotics researchers are using sensor fusion to overcome the limitations of individual sensor shortcomings that limit their applicability. Fused sensory information can be more veridical than that provided by a single sensor because individual sensor measurements may be uncertain, erroneous, and incomplete, whereas the fusion of multiple sensors results in a more reliable percept. In the field of robotics, the potential benefits of sensor fusion has motivated research, yet no general-purpose method for accomplishing sensor fusion across perceptual levels has been proposed (Arkin, 1998; Murphy, 1996, 1999, 2000).

Sensor fusion is a broad term used to describe any process in which sensor information or percepts are combined from multiple sensors into a single percept. Motivation for robotic sensor fusion stems from three basic combinations of sensors: *redundant* (or competing), *complimentary,* and *coordinated.* Redundant sensors generate a percept in one sensory modality. Complimentary sensors provide disjoint information about a percept. Whereas, coordinated sensor fusion, sequences of sensors are used, often for cue-ing or providing focus-of-attention (Arkin, 1998; Murphy, 2000).

According to Arkin (1998), Murphy drew on studies from cognitive psychology and neurophysiology showing that behavioral sensor fusion occurs in animals and, therefore, should be part of a robot's behavioral repertoire. Murphy (1996) reports that Stein and Meredith (1993) offer a neurological model of sensor fusion derived mainly from studies of the superior colliculus in the feline brain. The superior colliculus in cats is similar to that of most mammals, including humans, thus the studies are accepted as representative of the general phenomenon of sensor fusion.

Stein and Meredith (1993) show that different sense stimuli are initially segregated at the neural level. That is, neurons associated with one sense do not interact with neurons originating from other senses until they are transmitted to the brain. In the brain sensory signals converge on the same target, the superior colliculus. In addition to receiving inputs from the senses, the superior colliculus also receives signals from the cerebral cortex, which modulates or influences behavior. It is important to note that while the majority of neurons entering the superior colliculus are sense specific, the majority of neurons leaving the superior colliculus, estimated at 75%, are multisensory. The output of these neurons may be greater when multiple contributing neurons experience weak stimuli than if one undergoes a strong stimulus. Even though outputs go to many other structures, multisensory neurons appear mostly to form pathways to muscles and behavior control (Murphy, 1996).

The neurological model of sensor fusion describes several aspects important to robotic sensor fusion. Neurological studies of the superior colliculus integration of sensory inputs that influence motor control suggests sensor fusion is purposeful and not an artifact or by-product of the central nervous system. Sensor fusion couples perception with action and incorporates contextual information. Multisensory neurons in the superior colliculus include inputs from the cerebral cortex, which in turn modulates behavior. An aspect of particular importance is the observation that multisensory neurons can respond more to multiple weak stimuli suggesting that sensor evidence accrues as opposed to being averaged. Thus, for example, an agent can ascertain danger from multiple weak clues. In the case of robotics, accrual of sensory signals from several inexpensive coarse sensors may be used instead of a single expensive fine-grain sensor. Sensor fusion also allows a robotic perceptual system to be modular - where sensors can be added or removed from a sensor suite without impacting the operation of other sensors (Murphy, 1996).

Research into the perceptual phenomenon of synesthesia motivates a unique approach to sensory fusion. Synesthesia is a fascinating condition in which an otherwise normal person who experiences sensation in one sensory modality results in involuntary perception in another sensory modality simultaneously. A common example is *colored hearing* cases, where a person experiences colors when listening to a particular sound. The synesthesias are typically specific and stable. For example, separate instruments might evoke different visual sensations, such as hues and forms (Ramsøy, 2001).

Ramachandran and Hubbard (2001) review experiments they performed and assert they clearly establish for the first time that synesthesia is genuinely sensory. For example, in one experiment they showed that synesthetically induced colors lead to pop-out. Subjects were presented with displays composed of graphemes, such as a matrix of randomly placed computer-generated ‘2’s. Within this display was embedded a shape, such as a triangle composed of other graphemes, computer-generated ‘5’s. In this case the ‘5’s are mirror images of ‘2’s made up of identical features. Non-synesthetic subjects found it difficult to detect the embedded shape composed of ‘5’s. Whereas, synesthetic subjects, who see ‘2’s as one color and ‘5’s as a different color, see the display as a red triangle amidst a background of green ‘2’s. Performance measures show that synesthetic subjects were significantly better at detecting the embedded shape than non-synesthetic subjects.

The idea synesthesia may be the result of neural cross-wiring has been around for at least 100 years. A neuro-imaging study by Paulesu et al. (1995) used Positron Emission Tomography (PET) to investigate if a neural basis for synesthesia exist. In this study word-color synesthetes were presented with pure tones or single words. Regional cerebral blood flow measurements were taken during tone listening and word listening. Areas of the posterior inferior temporal cortex and parieto-occipital junction, but not early visual areas such as V1, V2, or V4, were activated significantly more during word listening than during tone listening in synesthetic subjects but not in controls. Due to the resolution of the technique, no precise anatomical localization was possible. Yet, the results suggest synesthesia results from activity in brain areas that deal with language and visual feature integration, and conscious visual experience occurs with activation of the primary visual cortex (Paulesu et al., 1995).

The cross-wiring hypothesis is supported by anatomical, physiological, and imaging studies in both humans and monkeys that implicate the fusiform gyrus. Ramachandran and Hubbard (2001) have identified different subtypes of number-color synesthesia and propose they are caused by hyperconnectivity between color and number brain areas at different stages in processing. It is speculated that this hyperconnectivity may be caused by a genetic mutation resulting in defective pruning of connections between brain maps and this is related to the neonatal synesthesia hypothesis, which proposes that all humans go through a normal synesthetic period during development. Lower synesthetes may have cross-wiring (or cross-activation) within the fusiform gyrus, whereas higher synesthetes may have cross-activation in the angular gyrus. The fusiform gyrus of the human brain holds the functional regions responsible for color, identification of face, and recognition of facial expression, Figure 1 blue, green, and red colored areas.

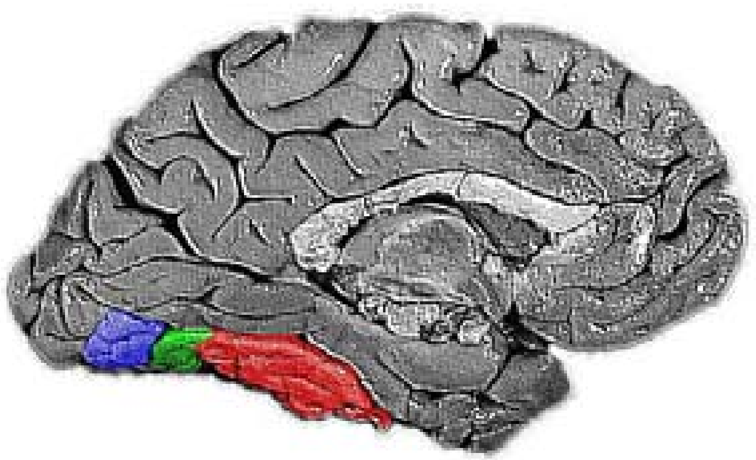

Figure 1. Location of the fusiform gyrus in the brain.

The fusiform gyrus of the brain contains the functional regions responsible for color (blue), identification of face (green), and recognition of facial expression (red). Damage to any of these brain areas leads to a deficit for that mode of visual function. Ramachandran and Hubbard question whether it is a coincidence that the most common form of synesthesia involves graphemes and colors and the brain areas corresponding to these are next to each other. They suggest that synesthesia is caused by cross-wiring (cross-activation) between these two areas, and in such a way as to be analogous to the cross-activation of the hand area by the face in amputees with phantom arms (Ramachandran & Hubbard, 2001).

Synesthesia is a concrete sensory phenomenon whose neural basis is just beginning to be understood, and this can be used as an experimental lever for developing polysensory mechanisms for robotics applications. The hypothesis tested in this study, based on the synesthesia cross-wiring hypothesis, is that cross-wired artificial neural networks would exhibit synesthetic sensory fusion.

The remainder of this thesis details the interdisciplinary background and motivation for the research conducted. The chapters are organized as follows. Chapter 2 provides a review of synesthesia. Chapter 3 is a historical perspective of relevant work by Grey Walter who took a unique approach to modeling brain function. Chapter 4 presents Donald Olding Hebb's seminal ideas on how the neurons in the brain learn. Chapter 5 introduces behavior-based robotics and the concept of sensor fusion. Chapter 6 examines the relationship between biological and artificial neural networks. Chapter 7 details the experimental design, and Chapter 8 concludes with the experimental results and their interpretation.

CHAPTER 2
SYNESTHESIA

The phenomenon of synesthesia derives its name from the Greek *syn*, together, and *aisthesis,* to perceive, and means to perceive together. Synesthetes have perceptual experiences in which they typically perceive two sensory modalities together. That is, sensation in one sense modality induces involuntary perception in another sensory modality simultaneously. For example, people with this condition see sounds, smell colors, and taste shapes. The most common type is experiencing color when hearing sounds.

According to Richard Cytowic, a clinical neurologist studying synesthesia, when we speak "we all intermingle the five senses all the time. We say that red is a 'warm' color, but green is 'cool'; her voice is 'sweet,' or sadness is 'blue.'" However, for synesthetes these are more than just mere metaphors, they are perceived as vivid real experiences. Through clinical testing Cytowic found that a true synesthete will repeatedly affirm that B-flat is green or that roast beef feels like an archway. If synesthesia were due to merely the creative use of language, associations would vary over time. Instead what has been found is that synesthetic associations do not vary over time (as cited in Lemley, 1984).

John Locke first described synesthesia in 1690 when he wrote about a blind man who claimed to understand what the color scarlet was because it was like the sound of a trumpet. Later in 1710 it was described in medical terms by Thomas Whoolhouse, and in 1869 Francis Galton noted synesthesia, and it has periodically received attention since that time (Cytowic, 1993). Even though synesthesia was a topic of scientific interest well over a hundred years ago, by the 1940s interest had faded due to the rise of behaviorism. Investigations of synesthesia depended on introspection, which relies on self-report data from subjects. Introspection was no longer considered a worthy avenue of data collection in experimental psychology; therefore, interest in synesthesia plummeted (Baron-Cohen, Burt, Smith-Laittan, Harrison, & Bolton, 1996).

Little was known about synesthesias underlying physiological causes until progress in the development of brain imaging technology, electrophysiological recording, DNA analysis, and other techniques became available. Renewed interest in synesthesia has followed the attention it was given in the early 1980s by Richard Cytowic (1993).

The following sections of this chapter describe who synesthesia affects and details characteristics of synesthesia and relates several accounts of what synethetes perceptually experience. Next, theory concerning synesthesia is considered with a progression to an examination of research directly and indirectly related to synesthesia.

Synesthetes

Regardless of the senses joined in a given synesthete, the similar histories synesthetes share are uncanny. Synesthetes are typically surprised to learn others do not perceive words, numbers, sounds, taste, and etceteras as they do. They recall always having idiosyncratic perceptions as earlier as they can remember, and that mentioning them at an early age often resulted in ridicule and disbelief (Cytowic, 1995).

Synesthesia runs in families with a pattern that is either autosomal or x-linked dominant transmission, meaning it can be inherited from either parent. More women than men have synesthesia, and in the United States the ratio is 3:1 (Cytowic, 1989), while in the United Kingdom women out number men in a ratio of 6:1. Approximately one in 2,000 people are synesthetes (Baron-Cohen et al., 1996). However, some experts suspect that as many as one in 300 people have a variation of the condition (Carpenter, 2001).

Synesthetes are preponderantly non-right-handed. They are normal in the conventional sense, appear intelligent, and come from all walks of life. They typically exhibit superior performance on the Wechsler Memory Scale. However, within their overall high intelligence, synesthete’s cognitive skills are uneven. A minority are dyscalculic, an inability to conceptualize arithmetic facts, such as numbers, numeric relationships, and outcomes of numerical operations, e.g., estimating the answers to numerical problems before performing calculations. However, the majority of synesthetes have subtle mathematical deficiencies such as lexical-to-digit transcoding. Many exhibit allochiria, right-left confusion, as well as a poor sense of direction for vector rather than network maps (Cytowic, 1995).

Synesthetic relationships are typically unidirectional meaning for instance a particular synesthete’s sight may induce touch perception, but touch does not induce visual perception. This means the number of permutations for synesthetic experiences, if perception of movement is included, is 30. However, the senses of sight and sound are involved considerably more often than others, and it is rare for smell and taste to be either the trigger or the synesthetic response (Cytowic, 1995).

The strangest synesthesia is perhaps *audiomotor*. An adolescent boy with this condition positioned his body in different postures according to the sounds of different words. The boy claimed English and nonsense sounds had defined physical movements, which he would demonstrate with various poses. The physician who described this boy re-tested him 10 years later without warning, and he assumed without hesitation the identical postures of a decade earlier (Cytowic, 1995).

Synesthetic perceptions are generic and durable, never pictorial or elaborated. *Durable* refers to the fact that synesthetic cross-sensory associations do not change over time, and this has been verified by test-retest sessions given even decades apart without warning. *Generic* means the synesthetic experience is unelaborated. While a nonsynesthete may *imagine* a rich floral landscape while listening to classical music, synesthetes perceptions are characterized as blobs, lines, spirals, and lattice shapes; feeling smooth or rough textures; an agreeable or disagreeable taste such as salty, sweet, or metallic (Cytowic, 1995).

The synesthetic experience is emotional and is accompanied by a sense of certitude that is perceived as real and valid. The following accounts relate how a few synesthetes describe their synesthetic perceptions.

Michael Watson

Michael Watson a New York City stage-lighting designer describes his sense of taste as feeling geometric forms, which can be considerably complex, pressing against his face and hands. He describes his perception of spearmint as follows.

> I can reach my hand out and rub it along the backside of a curve. I can't feel where the top and bottom end: so it's like a column. It's cool to the touch, as if it were made of stone or glass. What is so wonderful about it, though, is its absolute smoothness. Perfectly smooth. I can't feel any pits or indentations in the surface, so it must not be made of granite or stone. Therefore, it must be made of glass (Cytowic, 1993).

Carol Crane

Carol Crane a psychologist loves most kinds of music, but concerts have an unusual affect on her. “The sound of guitars always feels like someone is blowing on my ankles. The piano presses on me right here,” she says, tapping her chest just over her heart. “And New Orleans-type jazz hits me all over like heavy, sharp raindrops.” In addition she has reactions to letters and numbers. For instance the letter *b* is a navy blue, *c* is tawny crimson, and the numeral *4* causes her to see tomato red (Lemley, 1999).

Sean Day

Sean Day a linguistics professor describes his sense of taste as being colored in Technicolor. The taste of beef, such as a steak, produces a rich blue. Mango sherbet appears as a wall of lime green with thin wavy strips of cherry red. Steamed gingered squid produces a large glob of bright orange foam, about four feet away, directly in front of me (Carpenter, 2001).

Julie Roxburgh

Julie Roxburgh's synesthesia differs from the preceding accounts in that not only does she see color when she hears sound, but has the reverse, she hears sound whenever she sees color. However, for her it is not a pleasant experience and has resulted in a great deal of suffering. This form of synesthesia leads to substantial interference, stress, dizziness, an overwhelming feeling of information overload, and a need to avoid situations that are too noisy or too colorful. Unlike other cases, synesthesia for Roxburgh has lead to social withdrawal and disrupts everyday life (Baron-Cohen, 1996).

Phenomenology

A problem with synesthesia is that is has almost exclusively been characterized in phenomenological terms. Diagnosis relies heavily on phenomenological evidence, which is subjective. Yet, without compelling physiological or anatomical substantiation, synesthesia was destined to be treated with scientific skepticism and caution (Costa, 1996)

A growing body of evidence supports the fact that cross-modal associations take place in the mammalian brain. The following theories and research highlight recent developments in the understanding of synesthesia.

Neonatal Synesthesia Hypothesis

Maurer's (1993) developmental theory of synesthesia states all human neonates are synesthetic. The Neonatal Synesthesia (NS) hypothesis argues that in early infancy, up to about 4 months of age, all babies experience sensory input in an undifferentiated manner. In contrast the Cross-Modal Transfer (CMT) hypothesis suggest objects can be recognized in more than one modality because infants are able to represent objects in an abstract form. There is evidence in support of the CMT hypothesis. For example, Rose, Gottfried, and Bridger (1978) found that 12 month olds looked longer at an object they had just orally explored.

The NS hypothesis builds on CMT evidence and suggests there is an anatomical basis for neonatal synesthesia if one considers transient connections between neural structures in neonates of other species. For example, the neonatal hamster has transient connections between the retina and main somatosensory and auditory nuclei of the thalamus. Maurer suggests the same may be true of human neonates. It has been found that only during early infancy, evoked responses to spoken language are detectable over the temporal cortex, as expected, and they are also found over the occipital cortex simultaneously. This suggest that the primary sensory cortex is not as specialized in infants as it is in adults (Baron-Cohen, 1996).

Synesthesia and Concepts

A study done by researchers at the University of Waterloo indicates that for one synesthete, color experiences associated with digits could be induced even if the digits were not present. Researchers presented a synesthete with simple arithmetic problems, such as "5 + 2." The experiment showed solving an arithmetic problem activated the concept of 7, leading the synesthete to perceive the color associated with 7. The significance of this experiment is that it shows synesthetic experiences can be elicited by activating the concepts of digits, and suggest color experiences are associated with a digit's meaning and not merely its form (Dixon, Smilek, Cudahy, & Merikle, 2000).

Physiology of Colored Hearing

Neuroimaging studies suggest synesthesia has a biological basis. Evidence comes from studies showing that synesthesia can be induced in normal individuals through the use of hallucinogenic drugs such as LSD and mescaline (Cytowic, 1993). Positron Emission

Tomography (PET) has been used to investigate if a neural basis for synesthesia exist. In a study by Paulesu et al. (1995) six synesthetic women were compared with six matched controls.

PET detects brain activity as changes in regional cerebral blood flow (rCBF). Auditory words, not tones, were used as stimulation to trigger synesthesia. When hearing words while blindfolded, the synesthetes showed abnormal activation, as a measure of rCBF, in areas of the visual association cortex. In both groups word stimulation, as opposed to tone, activated language areas of the perisylvian regions. However, in synesthetes a number of additional visual associative areas were activated, including the posterior inferior temporal cortex and the parieto-occipital junctions.

The inferior temporal cortex has been implicated in the integration of color with shape, as well as in verbal tasks that require attention to visual features of named objects. Synesthetes also had activation in the right prefrontal cortex, insula, and superior temporal gyrus, but no significant activity was detected in the lower visual areas, including V1, V2, and V4. The results from their study suggest synesthesia results from activity in brain areas that deal with language and visual feature integration, and conscious visual experience occurs with activation of the primary visual cortex (Paulesu et al., 1995).

The Limbic System

June 29, 1981, Cytowic in an experiment with the synesthete Michael Watson (MW), measured cortical metabolism by means of radioactive ^{133}Xenon inhalation while he was undergoing a synesthetic experience. Use of this technique yielded interesting, yet unclear, results. The results showed that hemispheric regional Cerebral Blood Flow (rCBF) dropped a full 18% in the left hemisphere during the trial. This is unusual because it is expected that during an active state activity in the cortical areas should rise. MWs mean hemispheric flows were already low and inhomogeneous, yet he showed an additional decrease of 18%, which is impossible to obtain in a normal person. Such a drastic drop would make a normal subject candidate for paralysis or some other visibly disabling condition, and such a condition is not obtainable in a normal person even with a drug (Cytowic, 1993, 1995; Lovelace 1999).

According to Cytowic, MWs metabolism dropped to such a low point during synesthesia that he should have been blind, paralyzed, or shown some conventional sign of a brain lesion. MWs left hemispheric flows were almost three standard deviations below the labs established acceptable limits of normal. However, his thinking and neurological exams were unimpaired (Cytowic, 1995). Cytowic took these results to mean that it is the limbic system and not the cortex that figures heavily in producing synesthetic responses. Based on a single subject he concluded that the limbic system has dominance over the cortex (Cytowic, 1993).

Building on these findings Cytowic cites the hippocampus as the most important structure for producing these responses because limbic epilepsy, which is usually observed as centering around the hippocampus, is known to evoke the same type of cross-modal perceptions synesthetes experience (Cytowic, 1995). However, Lovelace (1999) criticizes this conclusion based on the following observation.

MWs observed levels of deactivation were more than 3 standard deviations below the normal mean. The subject had a history of alcoholism and does not have a left posterior-communicating artery. He speculates with due logic that the observed results could be due to either miscalibrated instrumentation or attributable to physiological abnormalities in MW unrelated to synesthesia. Given the immediacy that concurrent sensations appear and that stimulus-induced changes in blood flow take seconds to occur, it is unlikely such a slow mechanism could explain a percept with such a brief onset latency (Lovelace, 1999).

Binding of Visual Features

What form does information stored in the human mind below the level of awareness take? In the field of visual perception there is an ongoing debate that centers around the question of whether things seen are stored as fragments and features or as an integrated whole. Are features of an object *unbound* when one is not aware of them, becoming *bound* together as a whole only when attention is employed? Mattingley, Rich, Yelland, and Bradshaw (2001) address this by studying synesthetes who experience color when they see certain *graphemes*, such as letters or digits.

Neurobiological evidence shows that separate features of visual information are projected to different cortical regions of the human brain. Relatively early in the processing of visual stimuli, color and shape are separate, and the brain can encode these features without awareness. This work supports the idea of modularity in the human cortex (Mattingley et al., 2001).

It is possible that color-graphemic synesthesia results from a flaw in the modular organization of the brain. Mattingley et al.'s (2001) results agree with the possibility that cortical regions for processing shape and color are abnormally linked, but only during awareness. These finding suggest that *attention* signals associated with awareness are required to produce normal binding (Robertson, 2001).

Visual Auditory Illusion

Vision is believed to dominate perception; however, Shams, Kamitani, and Shimojo (2000) have challenged this established view by showing that auditory information can alter the perception of a visual stimulus to create a visual illusion. They have discovered a visual illusion induced by sound. When a visual flash is accompanied by multiple auditory beeps, the single flash is incorrectly perceived as multiple flashes. In their experiment observers consistently reported incorrectly seeing multiple flashes whenever a single flash was accompanied by more than one beep. The illusion persisted even in informed subjects aware of the fact that there was only one flash.

Their results indicate illusory flashing is caused by an alteration of visual perception by auditory stimuli. Modification of visual perception by sound is not categorical but selective. Their results also showed that sound did not have a fusing effect when multiple flashes were accompanied by a single beep, and suggest the direction of cross-modal interactions is partially dependent on the type of stimulus. They propose that the perception of a continuous stimulus in one modality is rendered more malleable by a discontinuous stimulus in another modality than vice versa (Shams et al., 2000).

Tactile Discrimination

The visual cortex in blind humans is known to be involved in nonvisual perception, which has been attributed to neural plasticity resulting from visual deprivation. Zangaladze, Epstein, Grafton, and Sathlan (1999) showed that discrimination of the orientation of a grating on the fingerpad is associated with subjective reports of visual imagery. Positron emission tomography (PET) shows an increase in regional cerebral blood flow (rCBF) relative to what is seen during discrimination of grating texture in a contralateral region of extrastriate visual cortex near the parieto-occipital fissure. In a study using functional magnetic resonance imaging (fMRI), it was found that visual cortical areas are also active during tactile object recognition, compared with texture discrimination. Processing in the visual cortex may reflect top-down activation of visual representation used to facilitate tactile discrimination of orientation or shape. Alternatively the observed activity may be an epiphenomenon.

In order to distinguish between these two possibilities, researchers used transcranial magnetic stimulation (TMS) over the occipital scalp to block visual perception by disrupting function in the extrastriate visual cortex. The results show that TMS interferes with the tactile discrimination of grating orientation. Its time course and spatial restriction illustrate the specificity of the effect over the scalp. It is also shown by the failure of occipital TMS to affect detection of electrical stimulus applied to the fingerpad or tactile discrimination of grating texture. In contrast TMS applied to the somatosensory cortex blocked discrimination of grating texture and orientation.

The findings indicate that the visual cortex is involved in tactile discrimination of orientation. The findings also demonstrate that the visual cortex is necessary for normal tactile perception in normally sighted subjects (Zangaladze et al., 1999).

Excogitare

Synesthesia is an idiosyncratic condition that is not maladaptive, except in rare cases. For the synesthete, their perceptions are as normal and as real as a non-synesthete. Introspective reports have been verified by means of test-retest scenarios showing that synesthete's perceptions are not attributable to imagination. In light of the fact that synesthesia has been documented as a real phenomenon since 1690, substantial research was not undertaken until physiological investigations implicated a neural basis for synesthesia.

Research has shown that synesthetes differ neurophysiologically. Maurer theorized that synesthesia is a natural developmental state all humans go through in which transient connections exist between neural structures. Neurophysiologial evidence from other species and recordings of evoked potentials in humans during early infancy show cross-sensory activation in the temporal cortex and occipital cortex. Research has also documented that in some synesthetes that the mere thought of the concept of a number results in a synesthetic perception. In normal individuals synesthesia can be drug induced with hallucinogenics such as LSD or mescaline. Neuroimaging studies using PET and fMRI suggest synesthesia results from activity in brain areas that deal with language and visual feature integration. Additionally research has shown that a visual illusion can be induced by audition in normal individuals.

But why do polysensory mechanisms exist in the brain? Considered in terms of evolutionary theory, one avenue of reasoning is they exist because the ability to pair perceptions has survival value. This implies that in an unconscious manner all humans have cross-sensory mechanisms, but synesthetes have cross-modal perceptions. However, this researcher speculates, based on the findings of research directly and indirectly related to synesthesia, that the functioning of the human brain, and other brains, is an efficient cooperative process in which neural structures are reused. Simply stated, it is efficient to reuse an existing mechanism.

The Zangaladze et al. (1999) investigation showed that the visual cortex is involved in the process of tactile discrimination of orientation. One does not have to ask why, but must ask what advantage does this offer? If vision and touch were absolutely separate systems, then each would have to posses its own mechanism for determining orientation. This would essentially be a replication of function. Instead, why not let the two systems share an orientation discriminator, which is what this research suggests. Because vision is dominant and has a well-developed orientation discriminator, it may be that tactions uses a resource normally devoted to vision. Based on research findings, it is obvious that the brain is modular, and that these modules work together in a cooperative manner.

CHAPTER 3

HISTORICAL PERSPECTIVE: *Machina speculatrix*

Imagine a simple machine that explores its environment, is attracted to light, and maneuvers around obstacles. A robot that behaves like an animal sounds like something from science fiction or the latest in artificial life. Valentino Braitenberg, in his 1984 book *Vehicles: Experiments In Synthetic Psychology*, presents a series of thought experiments in which vehicles, simple systems of increasing complexity are constructed from elementary mechanical and electronic components that exhibit complex life-like behaviors. Each of these imaginary vehicles in some way mimics intelligent behavior. Vehicles in the series incorporate essential features of earlier models, and as these vehicles evolve; they are attributed with qualities such as aggression, love, logic, foresight, concept formation, creative thinking, personality, and free will. Yet, if one did not know the principles behind the vehicles' construction, these qualities would not be attributed to the simple control mechanisms that generate their behaviors.

Braitenberg presents these vehicles as evidence for what he calls the "law of uphill analysis and downhill invention," meaning that it is significantly more difficult to try to guess internal structure from mere observation of behavior than it is to create the structure that results in the behavior. Stated another way, it is easier to design a mechanism anew to do something, than it is to figure out how nature has evolved to do it. This suggests that perhaps nature's way is not really insuperably complicated.

Braitenberg's basic machine is a motor connected to a sensor that controls activation of the motor. These simple machines hardly qualify as brutes, and he considers them for no more than a few pages. Next he adds multiple sensors and multiple motors, crossing their wires, and makes some of the connections inhibitory. The results are extremely simple machines, creatures, an observer has no difficulty in attributing with fear, aggression, love, and affection along with a wandering eye; behavior resembling much of what is expected from biological systems.

Braitenberg pursued thought experiments, however, more than 30 years earlier, W. Grey Walter, a British neurophysiologist, used the same idea of complex behavior arising from simple components to build what he called "imitations of life" (as cited in Levy, 1992). This chapter describes Walter's robotic research, detailing the first robots, Elmer and Elsie, that he built between Easter 1948 and Christmas 1949. Next a robot that learns through classical condition, an evolution of the original is described.

W. Grey Walter

Over 50 years ago Dr. Grey Walter at the Burden Neurological Institute was engaged in pioneering research on mobile autonomous robots, building three-wheeled autonomous robotic vehicles as part of his quest to model brain function. Walter, well known for his work on the electroencephalogram, was deeply interested in investigating electromechanical models of simple reflexes exhibited by living creatures. He wanted to study the basis of simple reflex actions and to test his theory on complex behavior arising from neural interconnections. According to Sabbatini (1999), Walter was convinced that even organisms with extremely simple nervous systems could exhibit complex and unexpected behaviors.

Walter had a reputation as an interdisciplinarian genius, a pioneer who explored the interface between electronics and biology. Exceptionally skilled and possessing an understanding of these two sciences, Walter, with uncanny insight, created the first autonomous robotic *animals.* These three-wheeled robots he called *tortoise* after an "Alice in Wonderland" character (Sabbatini, 1999). He nicknamed the first two built Elmer and Elsie, after the initials of the terms describing them – ELectro MEchanical Robots, Light Sensitive, with Internal and External stability (Walter, 1950).

The electronics and mechanics of these robotic animals were simple, consisting of a light sensor, touch sensor, drive motor, steering motor, and two miniature vacuum tubes. The robot's drive train consisted of three wheels arranged in a tricycle-like formation. The front wheel provided propulsion and steering, each function controlled by separate motors. The *sense organs* were simple, a light sensor and a contact sensor. Power was supplied by a miniature hearing aid B battery and a six-volt storage battery mounted at the back of the assembly, and a Perspex, an acrylic plastic, shell protected and covered the complete assembly.

The *nervous system* of these creatures, a simple artificial neural network, consisted of just two neurons, an analog circuit built using two vacuum tubes; a pair of interlinked amplifiers controlled the wheel motors and the direction using information from the sensors. The turtles were designed to perform two actions. First, they *knew* how to avoid large obstacles, retreating if they hit one. Second, they would seek out a light source, and if the light was of sufficient intensity they would move away from instead of toward the source; like moths they are *phototropic animals*. With this simple design, Walter demonstrated that his robotic creations exhibited a variety of complex behaviors. Based on their behavior, Walter named these robots with the mock-biological name, *Machina speculatrix,* because "it explores its environment actively, persistently, systematically as most animals do" (Walter, 1950).

Machina speculatrix

Machina speculatrix was unlike other robotic creations preceding them, and its uniqueness stems from the fact that they did not have a fixed behavior. Instead the robots had reflexes, and through interaction with their environment, behaviors resulted that were never exactly repeated in the same manner, but rather followed a general pattern much in the same way animals do. This emergent life-like behavior was an early form of what is now called Artificial Life. These robots were the first artificial mechanical creatures having some of the typical properties of living creatures, such as behavior and self-organization.

Walter's interest in Artificial Life stemmed from his work in neurophysiology. In order to understand the complexities of the brain, he proposed building electronic models. However, he recognized that the sheer number of neurons present in the brain was an obstacle towards understanding. Walter, in his book, *The Living Brain,* (1953) states, “If the secret of the brain’s elaborate performance lies there, in the number of its units, that would be indeed the only road, and that road would be closed” (p. 118). Therefore, his focus was based on the assumption that it was not the sheer number of cells in the brain but that the “richness of their interconnections” was more so important (Ward, 1998). Walter (1950) states, “The fact that only a few richly interconnected elements can provide practically infinite modes of existence suggests that there is no logical or experimental necessity to invoke more than *number* to account for our subjective conviction of freedom of will and our objective awareness of personality in our fellow men” (p. 44).

Walter built his first robot in 1948. Its *brain,* so to speak, consisted of two neurons, a pair of interlinked amplifiers connecting its two sensors to two motors. The light sensor was attached to the spindle of the steering column so that it always faced in the same direction of the single front drive wheel. One motor steered the machine by turning the spindle, while the other drove the wheel. The other sensor was a contact switch that closed whenever the robot shell bumped into something and this tipped one of the amplifiers into oscillation (Ward, 1998).

Emerging from these simple connections, a panoply of behaviors was exhibited. Normally the light sensor, a photocell, scanned round and round while the drive wheel revolved at half speed, sending the robot in a series of curves in search of dim lights, a cycloidal gait. If light was detected, it would stop scanning and race towards it, but if the light was too bright, dazzled, the robot would begin scanning again, turning away from the light. If an object was hit, the contact reflex would switch the robot between its normal and dazzled states; thus it would repeatedly backup and turn until an obstacle had been negotiated.

When the robot was in the presence of a single light source, it would circle around it in a complex path of advance and withdrawal. However, if there were another light source farther away, the robot would first visit one and then the other and would continually stroll back and forth between the two. Due to the nature of their design these robots elegantly solved the dilemma of Buridan's ass, who scholastic philosophers said would die of starvation when placed between two barrels of hay if it did not possess a transcendental free will. However, this should be considered in the context of the details of the robots operation. When the photocell detects a light, both tubes amplify the signal. If the light is very weak, a change of illumination is the effective signal, but a stronger signal is amplified without loss of its absolute level. The effect in either case is to halt the steering wheel so the robot moves toward the light with the least amount of difficulty. When the light intensity increases to a sufficient level, the signal becomes strong enough to operate a relay in the first tube, which has the opposite effect on the second tube. This results in the steering mechanism operating at double speed causing the robot to abruptly sheer away in favor of gentle stimulation (Walter, 1950).

Walter's robots were designed to seek out a hutch where they normally stayed when recharging their batteries. Inside of the hutch were a 20-watt lamp and a battery charger. When the batteries were sufficiently charged, the robot would be attracted to light from distant sources, but, at threshold, the brilliance is so that it acts as a repellent, causing the robot to wander off and explore its environment. However, when the batteries' charge was low, the effect enhanced the sensitivity of the amplifier so the attraction of light was increased. In this way the robot could locate its hutch and be attracted home, because the bright light was no longer dazzling. Once inside the hutch and connected to the battery charger, the flow of current effectively put the robot to sleep, because power was disconnected from its *nervous system*. After the batteries charged, the internal circuits would be automatically reconnected. Now light that had attracted the robot to its home repels it away (Walter, 1950).

Inevitably this peripatetic robot encountered objects that it could not see, even though it avoided obstacles that cast a shadow when approaching a light source. However, these robots were equipped with a device enabling them to navigate around obstacles. The robot's Perspex shell was suspended on a single rubber mount and had enough flexibility that allowed it to move and close a ring contact. When the ring contact closed it converted the two-stage amplifier into a multivibrator whose oscillations rhythmically opened and closed the relays that controlled the application of power to the motors for steering and crawling, and at the same time ignored signals from the photocell. When contact was made with an obstacle, regardless of operation mode, all stimuli were ignored and the gait was transformed into a succession of butts, withdrawals, and sidesteps until the obstacle was either pushed aside or circumvented. Oscillation continued for about a second after the obstacle has been cleared and during this period the robot attempts to move to a sufficient distance for maneuvering (Walter, 1950).

Machina speculatrix displayed a diverse array of behaviors that emerged from a simple set of elementary components. For instance, Walter fitted a small flash-lamp on Elmer's head that turned-off whenever the light sensor received an adequate light signal. Quickly the robot homed in on a mirror hung in the room and a dance of oscillations ensued. Exposure to light reflected from the indicator lamp was sufficient to operate the circuit controlling the robot's light response, thus, causing the machine to be attracted to its own reflection. "The model flickers and jigs at its reflection in a manner so specific that were it an animal a biologist would be justified in attributing to it a capacity for self-recognition" (Walter, 1950, p. 45). This behavior is due to the reflected light resulting in the indicator light being switched off, and darkness in turn switches it on again, resulting in an oscillation of light being set up. When Elmer and Elsie were placed in the same room, each with an indicator lamp, they engaged in a complex dance of attraction and repulsion. Yet, each attracted by the light of the other, both extinguished its own source of attraction. This resulted in the two becoming involved in a mutual oscillation that eventually led to a stately retreat (Walter, 1950, 1953).

Machina docilis

Walter building on the *Machina speculatrix* model constructed another robot that behaved even more like an animal. He gave this robot the mock-biological name *Machina docilis*, from the Latin word meaning *teachable*, because it could be trained using classical conditioning much like Pavlov's dog. *Machina docilis* was an evolved *Machina speculatrix* that had what Walter called the Conditioned Reflex Analogue (CORA). This mechanism created a connection between the robot's light reflex or contact reflex and another stimuli, such as a whistle. The robot could be trained by blowing the whistle and then kicking it to trigger the contact reflex. "After a dozen kicks the model [robot] will know that a whistle means trouble, and it can thus be guided away from danger by its master" (Walter, 1951, p. 62). In *Machina docilis* the associative memory is formed by oscillations in a feedback circuit. The deterioration of these oscillations is analogous to forgetting. The central part of CORA was a capacitor connected to the inputs. If the shell were kicked after a whistle blow, the capacitor charged until reaching a threshold. At this point the capacitor would discharge causing an electronic gate to open that allowed the whistle to stimulate the same response as kicking the machine. Additionally, if conditioning was not reinforced, extinction occurred and CORA shut the gate (Walter, 1953).

Walter was not the first to attempt constructing a machine that imitated a living creature's behavior as distinguished from appearance versus performance. Thomas Ross in 1938 made a machine that could find its way out of a maze. Through trial and error it could *learn* to find its way to a goal on a system of toy train tracks (Walter, 1953). Yet, Walter's tortoises may be considered the first autonomous mobile robots.

Excogitare

While many of Walter's theories have since been abandoned, the significance of the ideas at the foundation of his robots is now only being realized. According to Owen Holland, Walter built *Machina speculatrix* for a specific purpose, "He wanted to prove that rich connections between a small number of brain cells produces very rich behaviour" (as cited in Ward, 1998, p. 55). This is an idea that has a modern feel. According to Ward (1998) Rodney Brooks of the

Massachusetts Institute of Technology used the idea to lay the foundation of a field that has become known as *behavior-based robotics*.

Unlike earlier *intelligent* robots that used large control programs for decision making that limited them to very specific tasks, Brooks went against the *top-down* control paradigm in favor of the *bottom-up* approach in which control is delegated to very simple elements. For instance, each leg of his six-legged walking robot, Genghis, controlled its own actions, and the robot could walk and avoid objects or clamber over them (Brooks, 1999).

Brooks, in formulating his approach, drew on Walter's work. The animal-like behavior of *Machina speculatrix* is a trait singled out by adherents of behavior-based robotics. The emergence of unpredictable behavior, a hallmark of the natural world, is key to their claims they are progressing towards genuinely lifelike artificial organisms.

Walter was optimistic others would create similar creatures, imitating life with yet more complex machines. In the near future he predicted that similar imitations of life would be capable of repairing and reproducing themselves. However, as classical Artificial Intelligence imposed the top-down paradigm as dogma on experimental robotics and demanded creations possess a human-like grasp of logic, the supple, animal-like behaviors of Elmer and Elsie were destined for the curiosity heap (Levy, 1992).

The destiny of Elmer and Elsie may have been the curiosity heap, but Walter's optimism has been realized. Researchers such as Rodney Brooks, Ronald Arkin, and Robin Murphy have created robots that in essence imitate life. Like Brooks, Walter's work has provided motivation for this researcher. Moravec (1988) wrote, "I believe that robots with human intelligence will be common within fifty years" and that, "… I now expect to see a general-purpose robot usable in the home within ten years." Even though Moravec (1999) states, "There are still no mobile utility robots to help us around the house," and in light of his revised prediction that by 2010 *utility* robots will be common, this researcher does not share his optimism.

CHAPTER 4

CELL ASSEMBLIES

Grey Walter like Donald Hebb was interested in how the brain functions. Walter built his first robot in 1948 shortly before Hebb published his neurophysiological postulate. Other than sharing similar interests, it is not stated in the literature whether or not a mutual influence between their work existed. While Walter was primarily interested in how behavior arose from neural interconnections, Hebb postulated about how learning amongst neurons occurred and is the topic of this chapter.

D. O. Hebb

Donald Olding Hebb (1904-1985) was an extraordinarily influential figure in the discipline of psychology. His opposition to radical behaviorism and emphasis on understanding what happens between stimulus and response helped pave the way for the cognitive revolution. Viewing psychology as a biological science, his neurophysiological *cell assembly* proposal renewed interest in physiological psychology. Since his death, Hebb's seminal ideas continue to exert a growing influence on those interested in mind, brain, and how brains bring about mind (Klein, 1999).

Hebb's influence extends beyond the domain Psychology. For example, in 1943, mathematicians Warren McCulloch and Walter Pitts showed that it was possible to compute with a neural network. Six years later Hebb showed how a neural net could learn. If there is a core to the field of Artificial Intelligence today, it is most likely the connectionist school of neural networks. In this sense, McCulloch, Pitts, and Hebb can be considered the founding fathers of Artificial Intelligence (Susac, 1997).

Cell Assemblies and Phase Sequences

Hebb's fundamental idea, his neurophysiological postulate, was to assume that the brain is constantly making changes at the synapses. Hebb stated this assumption in his 1949 book *The Organization of Behavior*. "*When an axon of cell A is near enough to excite cell 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, 1949, p. 62). This was a bold assumption on his part because at the time he had no evidence to support it whatsoever. Having made this assumption, however, he argued that these synaptic changes were in fact the basis of learning and memory. For example, a sensory impulse coming in from the eyes leaves a trace on the neural network by strengthening all the synapses that are along its path. As a result, Hebb said, a network that was initially random would rapidly organize itself. Experience would accumulate through a kind of positive feedback, where strong, frequently used synapses grow stronger, while weak, infrequently used synapses atrophy. The favored synapses would eventually become so well established that the memories would be locked in due to structural change. These memories, in turn, would be widely distributed over the brain, with each one corresponding to a complex pattern of synapses involving potentially millions of neurons.

Hebb's second assumption was that the selective strengthening of the synapses would cause the brain to self organize into *cell assemblies*, where subsets of thousands of neurons in which circulating nerve impulses would reinforce themselves and continue to circulate. Hebb considered these cell assemblies the basic building blocks of information in the brain, each corresponding for example to a sound, an image, or a fragment of an idea. Yet, these assemblies would not necessarily be physically distinct, overlapping with neurons belonging to other cell assemblies. Because of this overlap, activation of one cell assembly would influence the activation of others, and this activation would lead to the activation of yet others, and so these fundamental building blocks would quickly organize themselves into larger concepts and complex behaviors. Thus, for Hebb, the cell assembly is the fundamental quanta of thought.

Just as cell assemblies are formed as aspects of an object become neurologically interrelated, in a similar fashion cell assemblies become neurologically interrelated to form phase sequences. A phase sequence is “a temporally integrated series of assembly activities; it amounts to one current in the stream of thought” (Hebb, 1959). Once developed, a phase sequence, like a cell assembly, can be fired internally, externally, or by a combination of internal and external stimulation. When any single cell assembly or combination of assemblies in a phase sequence is fired, the entire phase sequence tends to fire. When a phase sequence fires, one experiences a stream of thought, i.e., a series of ideas arranged in some logical order (Hergenhahn & Olson, 2000).

When considering Hebb’s cell assemblies, a shortcoming should be noted. Without inhibiting factors, learning would strengthen synaptic connections until all neurons fired continuously, hence it would make the system useless. In 1950 Nathaniel Rochester and his colleagues at the IBM research laboratory in Poughkeepsi, New York, observed this in a computer model of the cell assembly. Hebb himself never used a computer to test his idea that random nerve nets could organize themselves to store and retrieve information. In spite of this, Hebb’s work later inspired many computer models, from the perceptron to parallel distributed processing (Milner, 1993).

Research

Hebb proposed 53 years ago that animals perceive objects and carry out actions due to the collective ability of large assemblies of brain cells. However, since then, brain researchers have tended to focus on the responses of one or a few neurons at a time.

Two investigations appearing in the February 4th, 1999, *Nature* go against the experimental grain and bolster Hebb’s notion. Both indicate that human perception and learning arise from synchronized activity of clusters of neurons. Large numbers of nerve cells may briefly align the peaks and valleys of their electrical outbursts in order to render unified scenes and meanings from diverse sensations (Bower, 1999).

Some studies have used microelectrodes implanted in the brain. These studies have linked synchronized neural firing in cats and other nonhuman animals to perception and memory (Bower, 1998).

New efforts instead rely on measuring brainwaves by means of electrodes placed on the scalp. Brain waves arising from the synchronized neural activity known as gamma waves are the result of thousands of neurons firing at around forty hertz (Bower, 1999).

A study by Pulvermüller investigated the psychophysiology of word processing. In this investigation he adopted Hebb's theoretical position that cell assemblies are the building blocks of cognitive functions. He suggests these assemblies are not necessarily restricted to small cortical locus but may be dispersed over distant cortical areas. Different assembly topographies can be postulated for different kinds of words. Evidence from evoked potentials and gamma-band electrocortical responses elicited by lexical material supports a cell assembly model of language and other higher cognitive functions (Pulvermüller, 1996).

Rodriguez

Rodriguez et al. (1999) conducted a study in which 10 adults looked at images of either human faces or abstract shapes. To indicate what they saw they pressed one of two computer keys.

Brain tissue involved in vision exhibited gamma activity for an instant when volunteers scrutinized faces but not when they viewed the shapes. These synchronized responses are considered by Rodriguez's group to be crucial for integrating related sensations into a vision of a face, but they dissipated before any key was pressed. A second gamma burst arose in the motor areas of the brain as participants pressed a key, which may have helped coordinate an appropriate reaction (Rodriguez et al., 1999).

Miltner

Wolfgang Miltner of Fredrich Schiller University in Jena, Germany, has performed an investigation that indicates learning fosters synchronized neural activity. For this investigation 16 volunteers in a series of trials saw a flash of colored light that was immediately followed by a mild shock to the third finger of either the right or left hand. After the participants were conditioned, the flashing light alone-evoked surges of gamma activity in brain areas devoted to vision and to representation of the finger that had been shocked. This gamma activity was not observed once the participants learned not to expect a shock to a finger after seeing the flashing light (Miltner, Braun, Arnold, White, & Taub, 1999).

According to Wolf Singer of the Max Planck Institute for Brain Research in Frankfurt, Germany, although these studies do not demonstrate any precise functions for synchronized neural responses, gamma activity, they “could well be the mechanism that binds neurons into functionally coherent assemblies” (as cited in Bower, 1999).

Pulvermüller

According to Pulvermüller (1996) some ideas included in the Hebbian framework are now common features of brain theories about language and other cognitive functions. The assumption that neuron assemblies can be distributed over wide cortical areas is shared by recent large-scale neuronal theories. Apparently, there is a broad consensus that neurons of distant cortical areas can work together as functional units. The Hebbian framework, in addition to postulating there are large-scale neuronal networks, also provides criteria for the formation of cell assemblies, and thus an explanation for assembly topographies (Pulvermüller, 1996).

Within the Hebbian framework, Pulvermüller asks not where cognitive entities are represented in the cortex but where in the cortex correlated neuronal activity takes place when cognitive entities are being learned. Cell assemblies representing phonological word forms are localized in the perisylvian cortices, associative learning must take place to store the rules that determine sequencing of words in well-form sentences and the meaning of word forms. In the Hebbian framework, frequently coactivated neurons strengthen their connections and develop into a higher order assembly representing the phonological word form together with its meaning. As such, theses assemblies related to meaningful words may actually be distributed over the entire cortex (Pulvermüller, 1996).

Studies of neurological patients with linguistic deficits revealed that lesions involving areas outside the perisylvian language cortices of the left hemisphere could lead to problems in processing words of particular categories. Lesions in the frontal lobe in the vicinity of Broca's area tend to result in problems in accessing verbs. Whereas, lesions in the inferior temporal lobe and the temporo-occipital areas tend to selectively affect the ability to access nouns (Pulvermüller, 1996).

In a positron emission tomography study by Martin, Haxby, Lalonde, Wiggs, and Ungerleider (1995) had subjects generate verbs and color words. These researchers found increased metabolic turnover in the ventral temporal lobe when color words were generated, and generation of verbs activated more superior temporal and inferior frontal areas but no additional motor cortices.

Data from evoked potentials have provided evidence that words of similar length and frequency can have different cortical counterparts. It has been suggested that these cortical representations are Hebbian cell assemblies. Although these ideas provide a tentative explanation of findings concerning high-frequency cortical responses related to cognitive processes, future theoretical and experimental research is necessary to further validate them (Pulvermüller, 1996).

CHAPTER 5
BEHAVIOR-BASED ROBOTICS AND SENSOR FUSION

Grey Walter built his robots to explore how behavior arose from rich interconnections amongst a small group of neurons and Donald Hebb postulated about how learning amongst neurons occurred. Walter's and Hebb's theories about brain functions inform those involved in robotics research. Rodney Brooks of the Massachusetts Institute of Technology influenced by the work of Walter used the idea of organizing a robot controller as a set of behaviors to lay the foundation of a field that has become known as *behavior-based robotics* the first topic of this chapter.

Behavior-Based Robotics

Two important characteristics of classical Artificial Intelligence (AI) methodology are the ability to represent hierarchical structure by abstraction and the use of *strong* knowledge that employs explicit symbolic representational assertions about an environment. The influence of AI on robotics was the idea that knowledge and its representation is central to intelligence and may have been a result of AI's preoccupation with human-level intelligence. Behavior-based robotics reacted against these traditions.

According to Rodney Brooks (1987), "Planning is just a way of avoiding figuring out what to do next." Even though this paradigm shift was initially resisted, the notion of sensing and acting within an environment has grown in preeminence in AI related robotics research over the previous focus on knowledge representation and planning. Advances in robotics and sensor hardware made it feasible to test behavior-based robotic hypotheses, and the results enamoured the imagination of AI researchers (Arkin, 1998).

The behavior-based approach to robotics has been evolving since about 1984 in a number of laboratories. Instead of modularizing perception, environment modeling, planning, and execution, this approach builds intelligent control systems where individual modules each directly generate part of the behavior of the robot. Along with an arbitration scheme, an integrated part of the framework, that controls which behavior producing modules has control of part of the robot at any given instance.

The behavior-based approach is an interdisciplinary effort that draws inspirations from neurobiology, ethology, psychophysics, and sociology. This approach grew out of dissatisfaction with traditional robotics and artificial intelligence that seemed unable to deliver real-time performance in dynamic environments. The central idea of the new approach is to advance both AI and robotics by considering the problems of building an autonomous agent that is physically an autonomous mobile robot that carries out some useful task in an environment that has not been specially engineered for it (Brooks, 1991).

A number of researchers starting about 1984 began rethinking the general problem of organizing intelligence. A reasonable requirement was that intelligence should be reactive to dynamic aspects of the environment, and that a mobile autonomous robot should operate on time scales similar to those of animals and humans. In addition intelligence should be able to generate robust behavior in light of uncertain sensors in an unpredictable environment.

Some of the key realizations about the organization of intelligence were that most of what people do in their day-to-day lives is not problem-solving or planning, but are routine activities in a relatively benign yet dynamic environment. Representations an agent uses of objects in its environment do not have to rely on naming those objects with symbols the agent possesses but can be defined through interactions of the agent with its environment. An observer can talk about an agent's beliefs and goals in light of the fact that the agent does not manipulate symbolic data structures. Brooks has argued that in order to really test ideas of intelligence it is important to build complete agents that operate in dynamic environments using real sensors. Internal world models, complete representations of an environment are impossible to obtainfs and are not necessary for agents to act competently. Further, many of the actions of an agent are separable, and coherent intelligence can emerge from independent subcomponents interacting with the environment (Brooks, 1991).

Two key ideas have led to solutions that use behavior-producing modules, situatedness and embodiment. When a robot is situated in its environment, it does not have to deal with abstract descriptions but instead deals with the *here* and *now* of the environment that directly influences the behavior of the system. Embodiment means that robots have bodies and experience their environment directly. Robot actions are part of a dynamic with an environment, and its actions have immediate feedback on the robots' own sensations.

The following example highlights the issues of situatedness and embodiment. A current generation industrial spray-painting robot is embodied but is not situated. That is, it has a physical extent and its servo routines correct for its interactions with gravity and noise present in the system. Yet, it does not perceive any aspects of the shape of an object presented to it for painting and merely goes through a pre-programmed series of actions (Brooks, 1991).

Subsumption Architecture

Brooks (1991) states that some success is achieved in the control of autonomous robots by combining sets of behavioral modules running in parallel. Classic approaches to robotics do not typically decompose control into separate behaviors. Instead, control is decomposed into functional modules that process information from sensation to output in serial stages. A module for perception attempts to reconstruct from visual input a representation of what things are and their location. This representation is then used by another module to make plans to reach the robot's objectives. Next commands from planning are passed on to an effector module that executes them.

Brooks (1986) explores an alternative that de-emphasizes classical approaches construction of internal representations. In the subsumption architecture control is decomposed into separate modules that guide simple behaviors. Each module uses sensory input to find features relevant for controlling its behavior. For example, object avoidance modules that use sonar sensors placed around the robot body to steer it away from nearby objects. Individual modules operate in parallel and detect cues from sensory input and generate commands to move the robot. When commands from several modules are combined, they can produce a robot capable of navigating an uncertain environment without collision.

Early work in behavior-based robotics used a fixed priority scheme called *subsumption architecture* to combine commands from behavior modules. In the subsumption architecture, behaviors are organized into ascending levels of competence. A behavior at the lowest level executes its commands with no awareness of the other behaviors above it. Yet, even at the lowest level of competence the robot can execute meaningful actions for its survival. A basic behavior, such as obstacle avoidance can still function even in the absence of navigation goals by moving the robot out of the path of approaching vehicles. As higher level behaviors are added, they impose additional constraints on the robot's behavior. Higher level behaviors take as input both sensory information and the outputs from lower level. When necessary, high levels modify the output from lower levels and substitute their own commands. For example, a behavior that moves the robot towards landmarks can replace the movement suggested by obstacle avoidance with an alternative that still avoids the obstacle but also moves closer to the landmark. Robots based on the subsumption architecture can be constructed with more sophisticated behavior by incrementally adding higher levels of competence (Brooks, 1989).

Robotic Sensor Fusion

Neurophysiologists and cognitive psychologists are studying sensory integration and intersensory perception in order to generate an accurate model of perception, while engineers, computer scientists, and robotics researchers are building robots, which require mechanisms, not theoretical models, for performing sensor fusion. Single sensor systems have not been completely successful for demanding tasks in navigation, target or goal recognition, and general scene interpretation. The primary disadvantage of a single sensor system is its inability to reduce uncertainty. Uncertainty occurs when features are missing or when the sensor cannot measure all the relevant attributes of a percept and when the observation is ambiguous (Murphy, 1996). Yet, no complete theory of sensor fusion has been presented in the cognitive and biological literature explaining how sensors influence and dominate each other while producing more accurate or confident perception (Murphy, 1994).

Sensor fusion is also an interest for those in the Artificial Intelligence (AI) and Artificial Neural Network (ANN) who work with autonomous mobile robots. The issue of interest is how to use information from one sensor to focus attention of another, and how to combine information from multiple sensors to improve measurement accuracy or confidence in recognition. Additionally, the demands of an unpredictable environment necessitate the use of multiple sensors to provide robustness in light of one sensor's shortcomings (Murphy, 1994). According to Arkin (1998) and Murphy (2000) behavior-based systems can organize perceptual information in three general ways: sensor fission, action-oriented sensor fusion, and perceptual sequencing or sensor fashion, Figure 2 through 4 illustrate these concepts. Sensor fission is easily understood, for example, a motor behavior requires a specific stimulus to produce a response, thus a dedicated perceptual model is used to channel its output directly to the behavior.

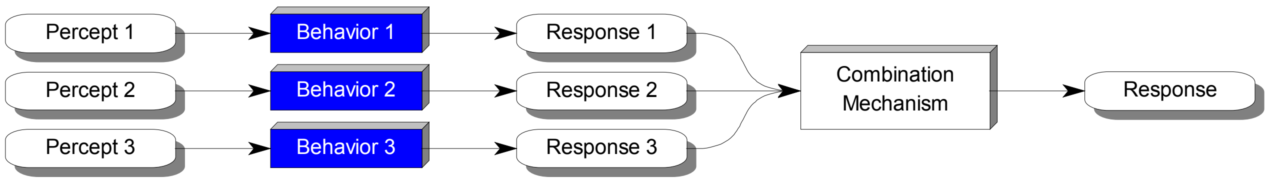

Figure 2. How percepts are combined in sensor fission.

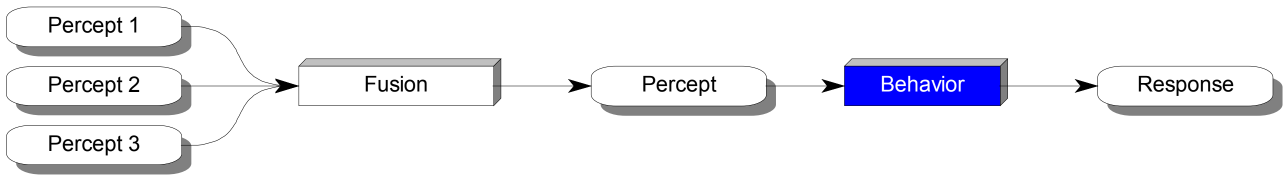

Figure 3. How percepts are combined in sensor fusion.

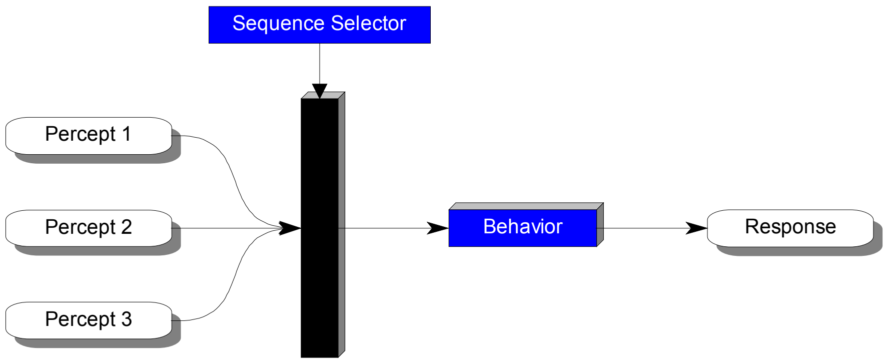

Figure 4. How percepts are combined in sensor fashion.

Action-oriented sensor fusion facilitates the construction of temporary representations (percepts) that are locals to behaviors. Increased robustness is achieved by restricting the final percept to the requirements of a particular behavior's requirements and context as well as retaining the advantage of reactive control while permitting more than one sensor to provide input (Arkin, 1998).

Fixed-action patterns sometime require varying stimuli to support their operation over time and space. Different sensors or different views of an environment may modulate a behavioral response as it unfolds. Perceptual sequencing allows the coordination of multiple perceptual algorithms over time in support of a single behavioral activity. Based on the needs and environment, an agent's perceptual algorithms are phased in and out. The phrase *sensor fashion* describes the significance of differing perceptual modules, changing over time and space (Arkin, 1998).

Sensor Fusion Effects Architecture

According to Murphy (1994) forays by the AI community into sensor fusion have essentially ignored cognitive and behavioral psychology. Murphy's work coalesced in the development of the Sensor Fusion Effects (SFX) architecture, which consist of three generic mechanisms derived from her study of cognitive psychology and neurophysiology. Figure 5 shows the neurophysiological influenced cognitive model of sensing based on studies of sensing in cats that SFX is modeled after.

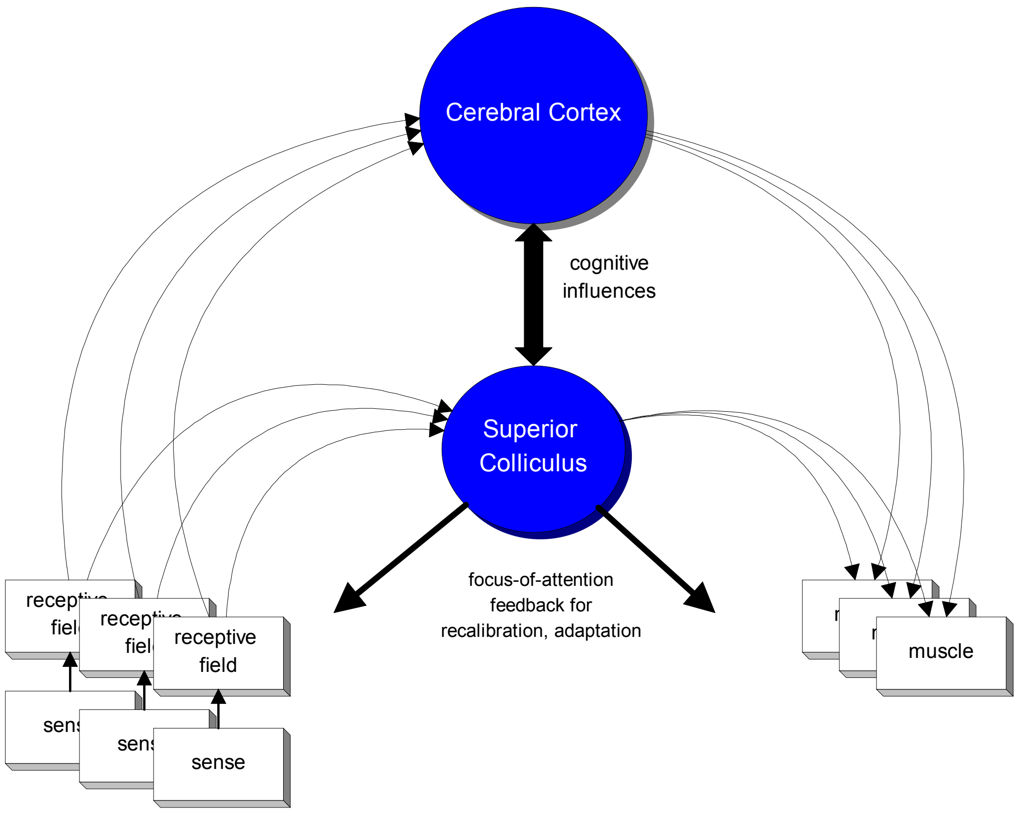

Figure 5. Cognitive model of sensing used in SFX.

The cognitive model suggests sensory processing is initially local to each sensor and may have its own sense-dependent field. According to Murphy (2000) the model is consistent with reactive robotic behaviors and at least with the motivation for sensor fission. Sensor processing then appears to branch where duplicates go to the *superior colliculus* and to the *cerebral cortex*, where branching allows the same sensor stream to be used in multiple ways. In SFX the equivalent superior colliculus functions are implemented in a reactive layer, and cortical activities are implemented in a deliberative layer. Perceptual branching is accomplished through the use of whiteboards, a global cognitive data structure common in many AI systems.

The SFX model has three states:

State 1. *Complete Sensor Fusion:* All sensors cooperate with each other in determining a valid percept.

State 2. *Fusion with the possibility of discordance and resultant recalibration of dependent perceptual sources:* Recalibration of suspect sensors occurs rather than the forced integration of their potentially spurious readings into the derived percept.

State 3. *Fusion with the possibility of discordance and a resultant suppression of discordant perceptual sources:* Spurious readings are entirely ignored by suppressing the output stream of the sensor(s) in question (Arkin, 1998).

SFX uses Dempster-Shafer theory to combine and propagate evidence, where evidence accrues like biological neural network models of sensor fusion advocated by Stein and Meredith (1993). SFX defines a perceptual process capable of performing sensor fusion that executes in two phases, an *investigatory* phase and a *performatory* phase, both derived from the study of orienting behavior. The investigatory phase relies on a *configuration* mechanism, while the performatory phases uses an *execution* and *exception handling* mechanisms (Murphy, 1996).

The responsibility of the configuration mechanism is to select the most appropriate sensing plan based on current operating conditions and the activated plan guides the execution of the execution mechanism that collects, processes, and fuses observations and evidence. Task-specific perceptual schemas for sensor fusion yield percepts directly related to the needs of a motor behavior. Perceptual schemas feed their parent schema and support higher-level schemas.

Each source of sensor data eventually grounds this recursive formulation. A parent perceptual schema combines the incoming subschema information using statistical techniques to produce a percept and a measure of its belief that is used in the motor schema (Arkin, 1998).

<u>Sensor Fusion in a Time-Triggered Network</u>

Elmenreich and Pitzek (2001) state that sensor fusion technologies are advantageous for systems that interact with their environment via a set of sensors, and that sensor fusion combined with smart transducer technologies leads to an effective system in terms of cost, robustness, decomposability and maintainability. An architectural model that supports a break down of a sensor fusion application into three levels, a node level, cluster level, and control application level, is used in the construction of a mobile robot. Communication between these levels is performed by means of a well-defined interface system.

In this study a mobile robot, a smart car, equipped with a suite of pivoted distance sensors, an electric drive, and a steering unit was used. Each sensor is represented as Time Triggered Protocol/Architecture (TTP/A) nodes, where each node is implemented on a separate low-cost microcontroller equipped with a smart transducer interface. The network also has a master node and a data processing node. The robot's distance sensors are able to scan the area in front of it because they are mounted on swivels moved by a servo motors. The distance sensors generate a value corresponding to the distance of an object.

The stream of data generated by the distance sensors is used by the data processing node that fuses the perceptions from the distance sensors with a model of the robot's environment. In the model studied, the shapes of obstacles were stored and assigned a probability value that decreases with the progression of time and increases when the object is re-scanned. The robot has 16 slave nodes and one master node, where navigation and sensor fusion is hosted on a single node. Sensor fusion techniques were used to establish a hardware independent interface to a control application. The Elmenreich and Pitzek (2001) study proved that openness to changes or extensions of sensor nodes and modifications of the control program result in a reduction of the system complexity at the cluster and control application level.

Just-in-Time Sensor Fusion

Rekleitis, Dudek, and Freedman (1996) describe an approach to combine range data from a set of sonar sensors and a directional laser range finder to take advantage of the characteristics of both devices when exploring and mapping unknown environments. The approach is described as “just in time” because it uses the more accurate yet constrained laser range sensor only as needed based on interpretation of sonar data. The key to their approach is that one sensor provides a large-scale but low-resolution depiction of the environment while a second sensor provides a more costly but higher resolution view. Research in sensor fusion has tended to focus on issues of how best to combine measurements from different sensors or how to best extract data with a single sensor and fuse the measurements over time. This research differs in that it is shown how to selectively extract measurements from different types of sensors.

Experimentation with a mobile robot equipped with sonar and a laser range finder demonstrated that judicious usage of the more accurate but more complex laser range finder was able to deal with known ambiguity that arises in sonar data. The algorithm used is based on knowledge of how sensor errors manifest themselves as well as how the environment is typically structured. It is this knowledge that allows the informed selection of locations to be probed with the more accurate sensor. The result was better mapping of a space at little additional computational expense (Rekleitis et al., 1996).

Neural Network Sensor Fusion

Davis and Stentz (1995) use a neural network paradigm to perform simulated and real-world navigation tasks that require the use of multiple sensing modalities. Their research uses backpropagation neural networks because when properly trained a neural network can automatically select and weigh the most important features of an environment, with the added capability of being able to tackle a significantly different environment by just changing the training data.

The goal of the research was to achieve autonomous navigation using the Carnegie Mellon University autonomous navigation High Mobility Multi-Wheeled Vehicle (HMMWV), a four-wheel-drive military ambulance. The goal was for the vehicle to safely wander around while avoiding obstacles. Testing was done in two environments, a square kilometer virtual world and on a 2-kilometer by 2-kilometer outdoor testing site. Two monolithic neural network architectures were used that performed almost identically in addition to a modular network architecture, the Modular Architecture Multi-Modal Theory network (MAMMOTH) – a network architecture and a training paradigm. MAMMOTH consists of two segments, a feature level and a task level. The feature level is a set of feature neural networks trained to recognize specific features in any sensory modality serving as an input source. The task level uses information from the feature level network's hidden layers as input to a network trained to perform the navigation task. The implications for sensor fusion in general are the ease with which new sensor modalities can be added to a given task. Results showed that a monolithic neural network is capable of learning to fuse sensing modalities (Davis & Stentz, 1995).

CHAPTER 6

BIOLOGICAL AND ARTIFICIAL NEURAL NETWORKS

Biological Neurons and Networks

Neural network architectures are motivated by models of the brain and nerve cells. Individual neurons are complicated and have a myriad of parts, sub-systems, and control mechanisms. Neurons communicate information by means of a variety of electrochemical pathways. There are over 100 different classes of neurons, depending on the method of classification. Collectively neurons and their connections form a process that is not binary, stable, nor synchronous (Anderson, 1995).

The brain is a dense neural network consisting of an estimated 100 billion neurons that use biochemical processes to receive, process and transmit information. A diagram of a nerve cell typical of those in the brain is shown in Figure 6. The output area of the neuron is a long branching fibre called the axon. The input area of the neuron is a set of branching fibres called dendrites (Dayhoff, 1990; Smock, 1999).

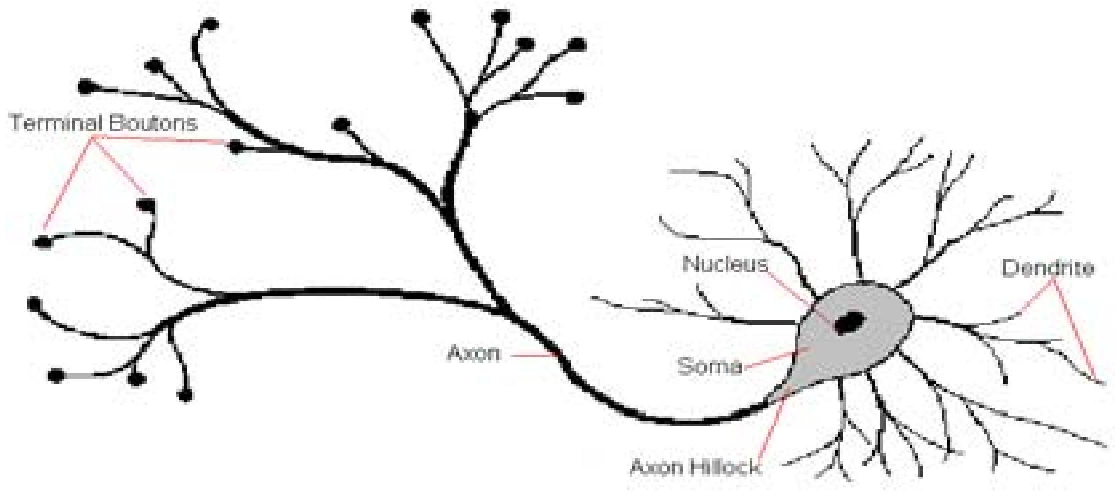

Figure 6. Schematic of a biological nerve cell.

The dendritic tree of a neuron is connected to thousands of other neurons. When one of those neurons fires, a positive or negative charge is received by a dendrite. The strengths of all the received charges are added together through the processes of spatial and temporal summation. Spatial summation occurs when several weak signals are converted into a single large one, while temporal summation converts a rapid series of weak pulses from one source into a large signal. The aggregate input is then passed to the cell body or soma. Yet, the soma and the nucleus do not take an active role in the processing of incoming and outgoing data. Their primary function is the continuous maintenance needed to keep the neuron functional. The axon hillock is the part of the soma that does play a role in determining the output signal of a neuron. If the aggregate input to the neuron is greater than the axon hillock's threshold value, then the neuron fires, i.e., an output signal is generated that is transmitted down the axon. The strength of the output is constant, even if the input was just barely above the threshold, or a thousand times as great. Additionally, the output strength is not affected by the many branches of the axon, the signal reaches each terminal bouton with the same intensity (Smock, 1999). This uniformity of signal is critical in analog devices such as the brain where small errors can multiply and because error correction is more difficult than in digital systems (Dayhoff, 1990).

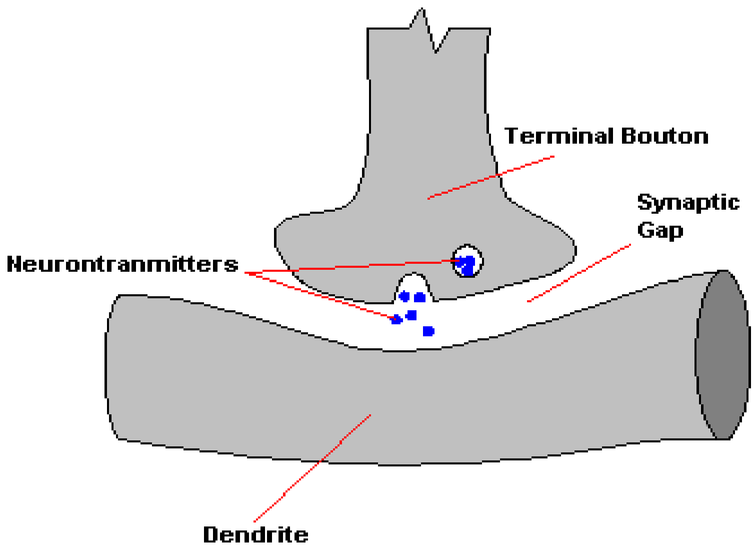

Figure 7. The synapse, a small gap between neurons.

The terminal bouton of one neuron, as illustrated in Figure 7, does not physically make contact with another neuron. Each terminal bouton forms a connection to other neurons across a small gap called a synapse. The neurochemical and physical characteristics determine the strength and polarity of the input signal for each synapse. This is where the brain is the most flexible, and the most vulnerable. Altering the composition of the various neurotransmitter chemicals can increase or decrease the amount of stimulation that the firing axon conveys to the neighboring dendrite. Changing the neurotransmitters can also change whether the stimulation a neuron receives is excitatory or inhibitory (Nicholls, Martin, & Wallace, 1992).

Artificial Neurons and Networks

Neural networks are computational structures inspired by the study of biological neural processing. The field is known by many names, such as connectionism, parallel distributed processing, neuro-computing, natural intelligent systems, machine learning algorithms, and artificial neural networks. An artificial neural network is an attempt to simulate within specialized hardware or by means of simulation software, the multiple layers of simple processing elements of neurons, where each neuron is linked to a number of neighboring neurons with varying coefficients of connectivity that represent the strengths of the connections. Learning is accomplished by adjusting the strength of these connections, causing the overall network to output appropriate results (Haykin, 1999).

The basic components of a neural network are modeled after the structure of the brain. Some neural network structures are not closely related to the brain and some do not have a biological equivalent in the brain. Yet, neural networks have a strong similarity to the biological brain and, therefore, share terminology from neuroscience.

The elemental unit of a neural network is the artificial neuron that simulates the basic functions of biological neurons. Artificial neurons are simpler than their biological counterparts; Figure 8 shows the elements of an artificial neuron.

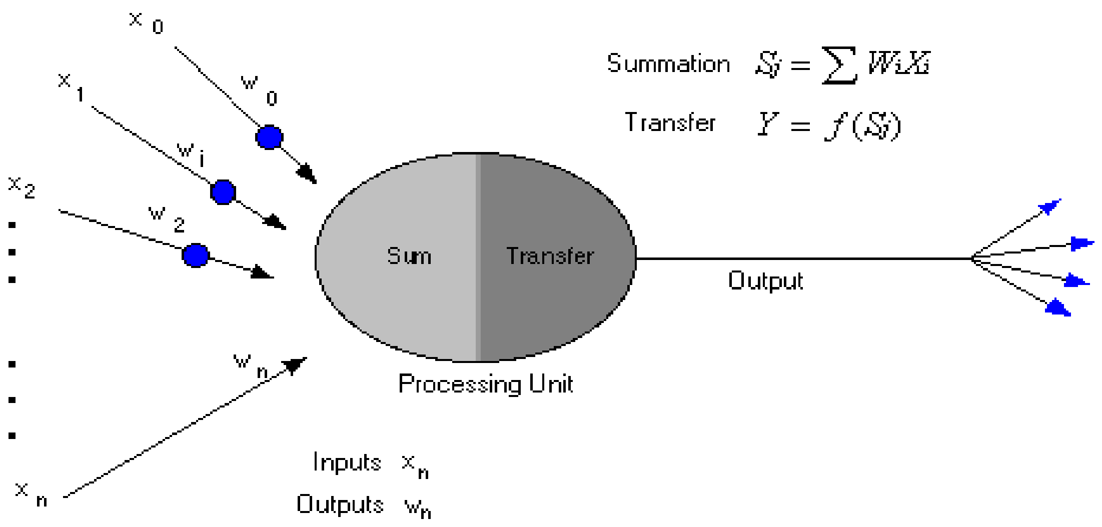

Figure 8. Elements of an artificial neuron.

The inputs to the network are represented by $\mathbf{x_n}$ and each of these inputs is multiplied by a connection weight $\mathbf{w_n}$. In the simplest case, these products are simply summed and processed by a transfer function to generate a result, and then an output. Even though all artificial neural networks are constructed using this basic building block, the fundamentals may vary (Rao & Rao, 1995).

Biological neural networks are constructed in three dimensions from microscopic components. While these neurons appear capable of unrestricted interconnections, this is not true of artificial networks that are the simple clustering of simple artificial neurons. Clustering occurs by creating layers, which may vary, and these are connected to one another. Essentially, all artificial neural networks have a similar topology, where a layer of neurons form external connections to receive inputs from the outside world and another layer of neurons provide the network's outputs to the outside world; all remaining neurons are hidden from view.

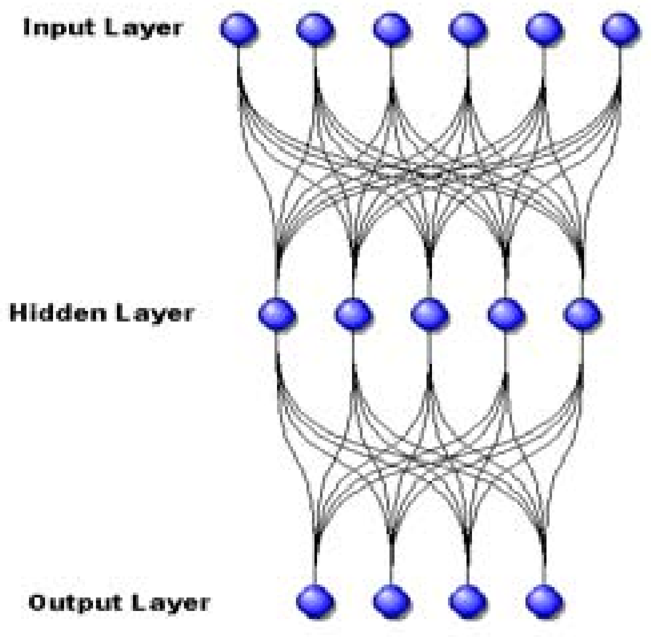

Figure 9. Layers in an artificial neural network.

Figure 9 illustrates how neurons are organized into layers. The input layer consists of neurons receiving input from external sources. The output layer consists of neurons that communicate the results of the network to a user or entity. Additionally there are typically one or more hidden layers between the input and output layers, and layers are usually fully interconnected but are not required to be so (Dayhoff, 1990).

Neurons are connected via a network of connections carrying the output of one neuron as input to other neurons. These paths are normally unidirectional, but there may be a two-way connection between two neurons because there may be another path in the reverse direction. A neuron receives input from many neurons and produces a single output that is input to other neurons. Additionally, the neurons in a layer may communicate with each other, but the neurons of one layer are always connected to at least one other layer (Haykin, 1999).

Learning

Neural networks are sometimes called machine learning algorithms because during training the connection weights are altered to cause the network to learn the solution to a problem. The connection strength between neurons is stored as a weight-value for a specific connection. The network learns by adjusting these connection weights.

Training typically consists of one of three methods, unsupervised learning, reinforcement learning, or back-error propagation. In unsupervised learning the hidden layer neurons determine how to organize themselves without external assistance. In this approach, no exemplars are provided to the network against which it can measure its performance for a given input vector (Haykin, 1999).

In reinforcement learning the connections among the neurons in the hidden layer are randomly set then adjusted as the network is told how close it is to solving the problem. Reinforcement learning is also called supervised learning because it requires a teacher that may be a training set or an observer who rates the performance of the network.

Back-error propagation is a proven, highly successful method used for training multilayered neural networks. In this method the network is given reinforcement and information about errors is also propagated back through the system and used to adjust the connections between the layers (Dayhoff, 1990).

Learning Rules

There are numerous learning rules used for training neural networks. These rules are mathematical algorithms used to update connection weights. The majority of these rules are variations of the most prevalent and oldest learning rules. The understanding of how neurological processing works is limited, and learning is more complex than the simplification represented by learning rules developed for artificial neural networks. A few of the major learning rules are:

- Hebb's Rule

  Donald Olding Hebb introduced the best know learning rule in 1949 in his book *The Organization of Behavior*. The rule states that if a neuron receives input from another neuron, and if both are highly active, the weight between them should be strengthened.

- Hopfield Rule

  This rule is similar to Hebb's Rule with the exception that it specifies the magnitude of the strengthening or weakening. The rule states that if the desired output and the input are both active or inactive, the connection weight is incremented by the learning rate, otherwise the weight is decremented by the learning rate.

- The Delta Rule

  The Delta Rule is a variation of Hebb's Rule, and it is one of the most commonly used. It is based on the idea of continuously modifying the strengths of the input connections to reduce the difference, delta, between desired output value and actual output value of a neuron. This rule changes the connection weights in such a way that it minimizes the mean squared error of the network. The error is propagated back into previous layers one at a time. The process of propagating the errors back into previous layers continues until the first layer is reached. This rule is also known as the Widrow-Hoff Learning Rule and the Least Mean Square Learning Rule.

- Kohonen's Learning Rule

  This rule was developed by Teuvo Kohonen and was motivated by learning in biological systems. In this process neurons compete for the opportunity to learn, i.e., update their weights. The neuron with the largest output is deemed the winner and has the ability to inhibit its competitors as well as exciting its neighbors. Only the winning neuron is permitted output, and only the winner and its neighbors are allowed to update their connection weights. Additionally, this rule does not require knowledge of the desired output (Dayhoff, 1990; Haykin, 1999; Rao & Rao, 1995).

Neural networks are an effective approach for a broad spectrum of applications. They excel at problems involving patterns – pattern mapping, pattern completion, and pattern classification. Neural networks may be applied to translate images into keywords, translate financial data into financial predictions, or map visual images into robotic commands. Neural networks offer an alternative method to analyze data, and to recognize patterns within data, than traditional computing methods. Noisy patterns, such as those with missing segments, may be completed with a neural network trained to recall the completed patterns (Dayhoff, 1990).

CHAPTER 7

EXPERIMENTAL DESIGN

The hypothesis to be tested in the study conducted was motivated by the synesthesia cross-wiring hypothesis, which states synesthetic perceptions are due to neurological cross-wired connections in the brain. Therefore, would cross-wiring two artificial neural networks result in a synesthetic response in one network. Research was conducted by means of computer simulation using software developed to simulate a cross-wired artificial neural network. The software architecture is detailed along with the source code in Appendix A.

Cross-Wired Artificial Neural Network Architecture

Figure 10 depicts the architecture of the cross-wired neural networks. The two networks, referred to as Network A and Network B, are cross-wired in the hidden and output layers of the network. In this design both networks have the same number of layers and same number of cells per layer. Each cell in the hidden and output layer receives input from the corresponding cell in the other network. In the figure, the red lines show the connections from Network A to Network B, and the blue lines show the connections from Network B to Network A.

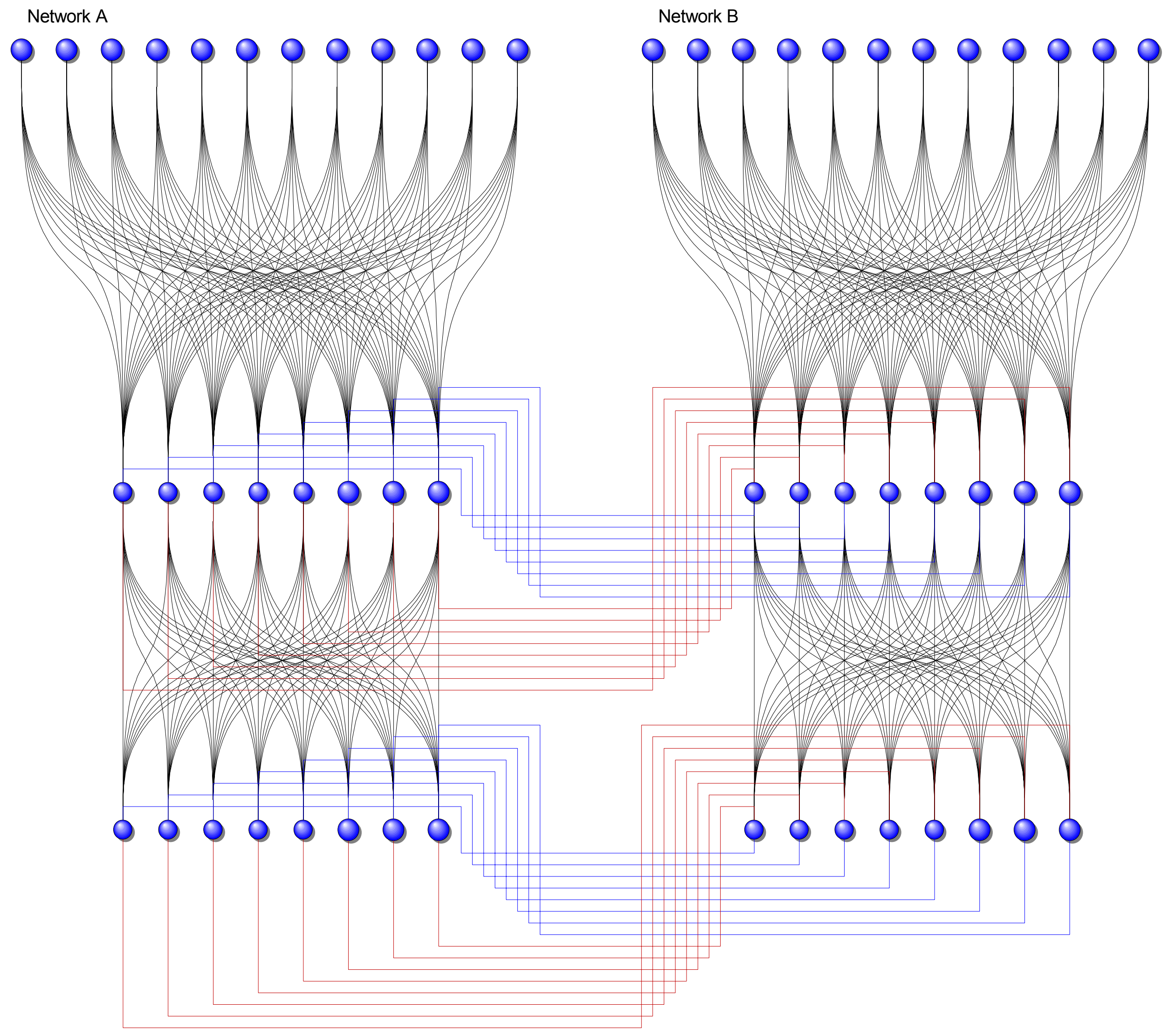

Figure 10. Cross-wired artificial neural network architecture.

Network Training

Training of the networks was performed in two stages. The first stage of training was performed using back-error propagation software by Rao and Rao (1995). In this stage each network, Network A and Network B, was independently trained to map 62 input vectors, patterns, to 62 output vectors, patterns, (see Table 1 and 2 in Appendix B). A root-mean-squared error tolerance of 0.001 was used in this training to evaluate when a network had converged, i.e., learned its training set. Each network was trained with the same input vector set but with different output vector sets. The rationale is that each network is independently coding for a feature, such as a grapheme or a color.

In the second stage of training, software developed for this researched was used to train the simulated cross-wired artificial neural network. Cross-wired connection training consisted of two steps, an initial Hebbian step followed by a Residual Hebbian step. Network A is influenced less by Network B, while Network B is influenced more by Network A because synesthesia is a unidirectional phenomenon.

In the first step, the initial values of the cross-wired connection weights for Network A are determined using a local Hebbian learning rule of the form:

$$W_{n+1} = \frac{\phi}{n} \sum_{i=1}^{n} W_i \qquad (1)$$

where

$$\phi = 0.6180339.$$

The rationale for deriving the initial cross-wired connection weight, W_{n+1}, for a cell is that it should preserve the balance of excitation and inhibition present in the existing weights. The new weight is computed to be the average of the existing connection weights multiplied by ϕ, the Golden Ratio. The factor ϕ was used in determining the initial weights because of the significance attributed to it in nature (Goodwin, 1994).

The Golden Ratio is a concept of elementary geometry that in the past as well as currently holds significant relevance in both human and natural designs. Consider the following line segment:

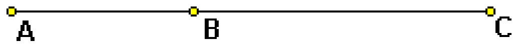

Figure 11. Line segment.

The ratio of the lengths of the two parts of this line segment is the Golden Ratio where:

$$\frac{AB}{BC} = \frac{BC}{AC} = \phi. \tag{2}$$

The initial values for the cross-wired connection weights for Network B are determined by a similar local Hebbian rule as Network A:

$$W_{n+1} = \frac{\Phi}{n} \sum_{i=1}^{n} W_i \tag{3}$$

where

$$\Phi = \frac{1}{\phi} = 1.6180339. \tag{4}$$

In this case, the strength of the cross-wired connection weights in Network B is greater-than those in Network A. Network A is influenced less by Network B, $\phi = 0.6180339$, while Network B is influenced more by Network A, $\Phi = 1.6180339$.

Step two of the second stage of training applies a Residual Hebbian learning rule to adjust the weights. In this step all weights are updated using the following local learning rule:

$$W_{i'} = W_i - \frac{\beta}{n} \sum_{j=1}^{n} W_j \tag{5}$$

where $\beta = \phi \times 0.000015$ for Network A, and $\beta = \Phi \times 0.00015$ for Network B. The average of all weights is multiplied by a small bias factor β, a small fractional constant value, and is subtracted from each weight until the weights converge. The residual leaning can be thought of as a small penalty that is proportional to what the cell already knows.

Testing Scenarios

To test for potential synesthetic responses three scenarios were used to exercise the cross-wired networks. Table 1 and 2 in Appendix B contain the input vectors referred to in each scenario. In the following scenarios Network A receives the same input vectors in all three test scenarios, input vectors 1 through 62. The rationale for this choice in the design of the testing scenarios was that it would simplify the identification of candidate vectors for synesthetic responses because Network A exerts more influence on Network B than Network B exerts on Network A. Thus, based on the response of Network B, an input vector or input vectors to Network A need to be identified in all three scenarios.

Scenario 1

In this scenario 62 *mixed* input vector pairs were presented as input to the networks. Input pairs consisted of input vectors 1 through 62 for Network A paired with input vectors 32 through 62 and input vectors 1 through 31 for Network B, for a total of 62 input vector pairs.

Scenario 2

In this scenario the input vector for Network B was held constant while the input vector for Network A varied. Input vector pairs consisted of input vectors 1 through 62 for Network A paired with two different input vectors for Network B, vector 31 and vector 62, for a total of 62 input vector pairs per run.

Scenario 3

In this scenario the *same* input vector was used for both networks. Sixty-two input vector pairs consisting of the same vector were presented as input to the networks. These input vector pairs consisted of input vectors 1 through 62 for Network A and input vectors 1 through 62 for Network B.

CHAPTER 8
RESULTS

For each scenario the mean of the absolute value of the error per output vector was computed; error is the difference between the non-cross-wired network output vector and the output vector for the network when cross-wired – not the ideal output value in the training set. Plots of the mean absolute error for each output vector were plotted to qualitatively determine if Network A consistently induced a synesthetic response in Network B. A synesthetic response in this context is a significant deviation in the response of Network Bs output vector induced consistently by Network A and associated with a specific input vector of Network A. The error tolerance used during the back-error propagation stage of training was a root-mean-squared error tolerance of 0.001; this value indicates a network has learned its training set. A significant deviation in this case is a mean absolute error greater than 0.001 because each of the eight values in an output vector may vary by a factor of ±0.001 and still be considered within the initial training tolerance.

Scenario 1 Results

In this scenario Network A and Network B receive mixed input patterns.

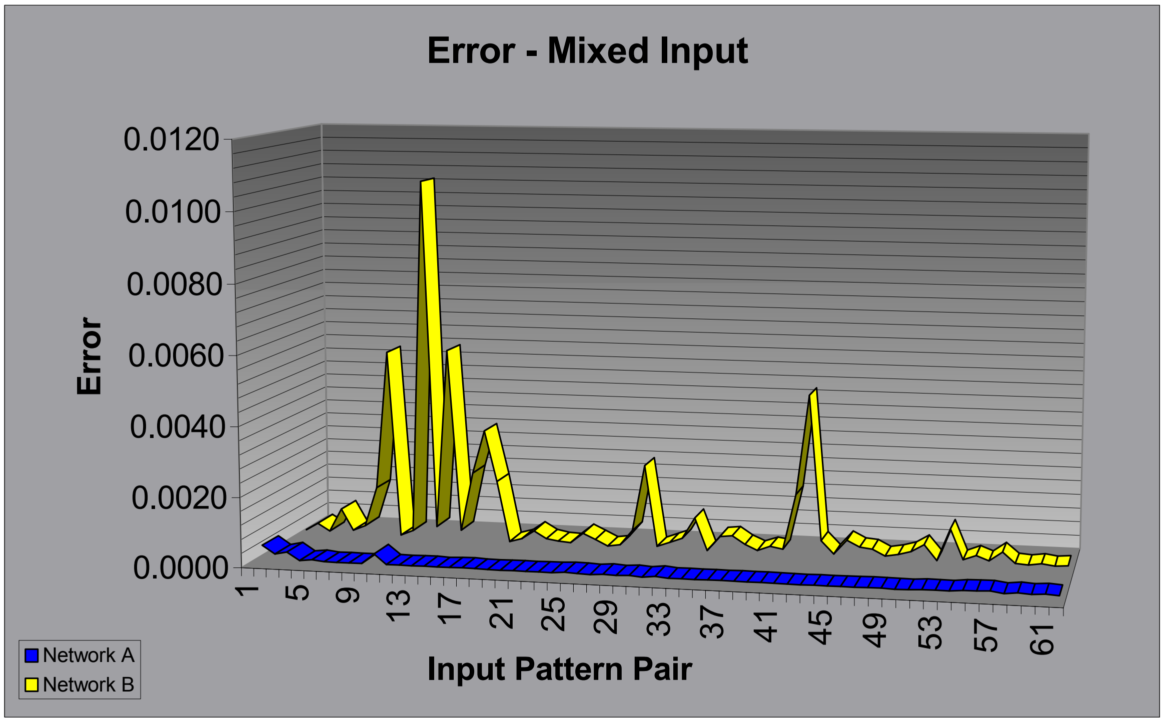

Figure 12. Error mixed input vectors.

By design, Network A receives less influence from Network B, and Network B receives more influence from Network A. Figure 12, the graph of the average absolute error per output vector for each network shows that indeed Network A is influenced very little, while Network B is influenced more. Network B exhibits 9 significant deviations for input vector pairs 8, 11, 12, 15, 16, 17, 29, 41, and 42. Therefore, these vectors are candidates for synesthetic responses.

Scenario 2 Results

In this scenario Network Bs input vector is fixed for two independent runs. Input vector 31 and 62 were used as constant input to Network B.

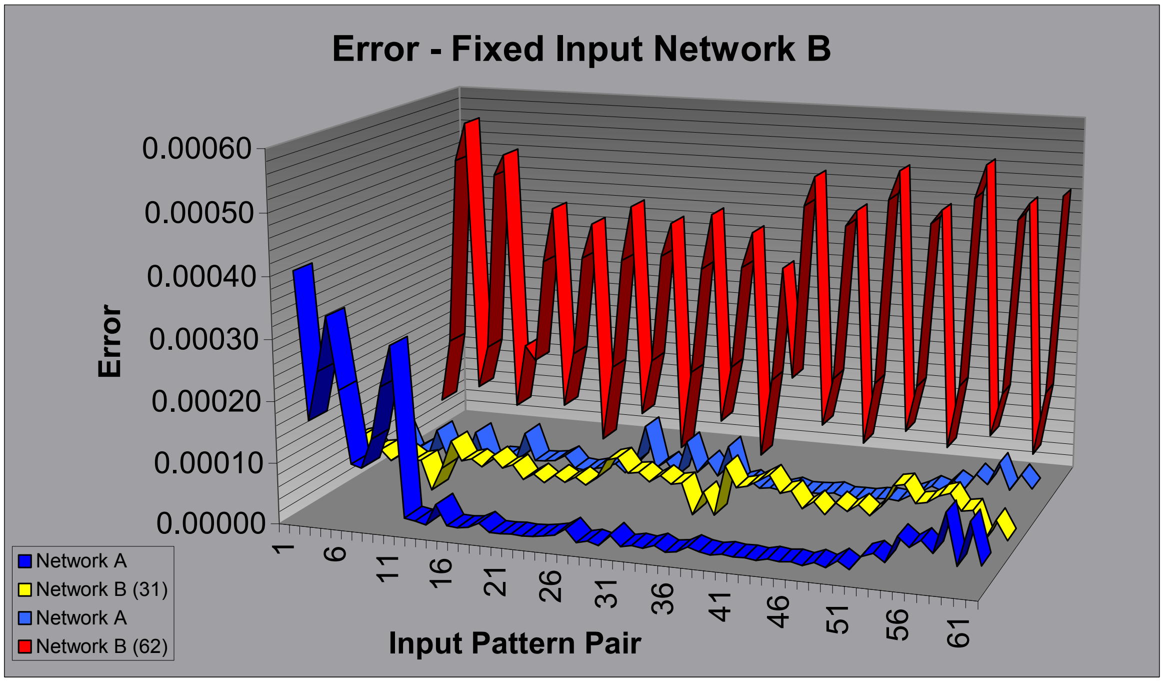

Figure 13. Error fixed input vector Network B.

In this scenario no input vector pair is identified as suspect for synesthetic response because none of the errors are significant. However, the graph in Figure 13 indicates there are small influences in each network by the other.

Scenario 3 Results

In this scenario the input pairs consisted of the same vectors for both networks.

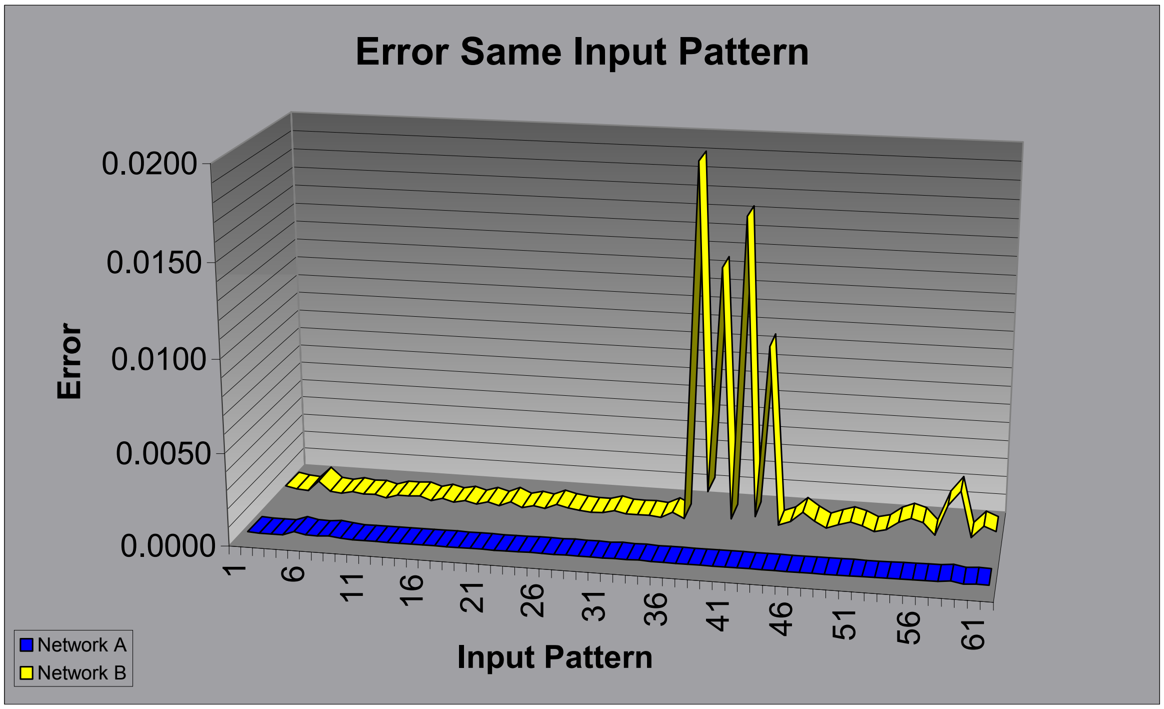

Figure 14. Error Network A and Network B same input vectors.

The results depicted in Figure 14 show that Network A is biased a small amount away from its expected output. Network B exhibits 6 significant deviations for input vector pairs 37, 38, 39, 41, 58, and 59. Yet, none of these vectors coincide with other candidate vectors in the other scenarios except for vector 41 in the first scenario, but the errors are different.

Conclusion

Qualitatively the collective results depicted in Figure 12 through 14 indicate that cross-wiring two artificial neural networks in the manner described does not significantly alter the behavior of the individual networks. It is observed that each network partially determines the output of the other network. Even though each network influences the other, it was observed that each network independently responded to its own inputs.

The purpose of this study was to determine whether or not two cross-wired artificial neural networks would exhibit synesthetic responses. Based on the results, no input vector causes a significant stable alteration in the output vectors of either network in all three scenarios. Thus, it is concluded that no synesthetic response occurs in this design of two cross-wired artificial neural networks.

The results additionally show that cross-wiring two independent artificial neural networks does not significantly alter the functionality of the individual networks, but it does allow inputs to one network to partially determine the outputs of the other network in some cases. That is, there are measurable influences of Network A on Network B, and yet, Network B retains its ability to respond independently to its own inputs.

A benefit of cross-wiring independently trained networks is that it potentially allows for the reuse of previously trained networks without the need to retrain. This implies complex networks may be constructed in a modular fashion. Modularization and reuse are desirable goals as they afford a saving in time as well as foster the reuse of design knowledge.

In the context of robotic sensor fusion, the results indicate that cross-wiring two initially independent networks is a feasible means of fusing sensor data using artificial neural networks. It is speculated, for instance, that this arrangement is operationally feasible for application in autonomous mobile robot navigation. In such an arrangement, it is conceivable that a proximity sensor network could bias a navigation network away from obstacles.

Further research is needed to assess the potential benefits of cross-wiring artificial neural networks. In this study only a single connection between neurons in adjacent layers was considered. Yet, it is conceivable that fully, partially, or sparsely interconnected layers may have merit. Additionally, it is speculated that the use of an interconnecting layer of neurons between networks may yield better results. In this case the interconnecting layer would act much like the hidden layers in a feed forward network, acting as a feature detector, thus allowing for a selective influence.

REFERENCES

Anderson, J. (1995). *An introduction to neural networks.* Cambridge, MA: The MIT Press.

Arkin, R. (1998). *Behavior-based robotics.* Cambridge, MA: The MIT Press.

Baron-Cohen, S. (1996). Is there a normal phase of synaesthesia in development? *Psyche, 2*(27), <http://psyche.cs.monash.edu.au> [February 21, 2002].

Baron-Cohen, S., Burt, L., Smith-Laittan, F., Harrison, J., & Bolton, P. (1996). Synaesthesia: prevalence and familiality. *Perception, 25*, 1073-1079.

Baron-Cohen, S., Harrison, J., Goldstein, L. H., & Wyke, M. (1993). Coloured speech perception: Is synaesthesia what happens when modularity breaks down? *Perception, 22*, 419-426.

Bower, B. (1998). All fired up: Perception may dance to the beat of collective neuronal rythms. *Science News, 153*(8), 120-121.

Bower, B. (1999). Neural ties that bind perception. *Science News, 155*(8), 122.

Braitenberg, V. (1984). *Vehicles, experiments in synthetic psychology.* Cambridge, MA: The MIT Press.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2, 14-23.

Brooks, R. (1987). *Planning is just a way of avoiding figuring out what to do next.* Working paper 303, MIT Artificial Intelligence Laboratory.

Brooks, R. (1989). A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation, 1*, 253-262.

Brooks, R. (1991). New approaches to robotics. *Science, 253*, 1227-1232.

Brooks, R. (1999). *Cambrian intelligence: The early history of the new ai.* Cambridge, MA: The MIT Press.

Carpenter, S. (2001). Everyday fantasia: The world of synesthesia. *Monitor On Psychology, 32*(3), <http://www.apa.org/monitor> [March 7, 2001].

Costa, L. F. (1996). Synesthesia – A real phenomenon? Or real phenomena? *Psyche, 2*(26), <http://psyche.cs.monash.edu.au> [April 21, 2001].

Cytowic, R. (1989). *Synaesthesia: A union of the senses.* Cambridge, MA: The MIT Press.

Cytowic, R. (1993). *The man who tasted shapes.* Cambridge, MA: The MIT Press.

Cytowic, R. (1995). Synesthesia: Phenomenology and neuropsychology. *Psyche, 2*(10), <http://psyche.cs.monash.edu.au> [January 21, 2002].

Davis, I., & Stentz, A. (1995). Sensor fusion for autonomous outdoor navigation using neural network. *Proceedings of the IEEE/RSJ International Conference On Intelligence Robotic Systems, 3*, 338-343.

Dayhoff, J. (1990). *Neural network architectures.* New York: Van Nostrand Reinhold.

Dixon M., Smilek, D., Cudahy, C., & Merikle, P. (2000). Five plus two equals yellow: Mental arithmetic in people with synaesthesia is not coloured by visual experience. *Nature, 406*, 365.

Elmenreich, W., & Pitzek, S. (2001). Using sensor fusion in a time-triggered network. *Proceedings of the 27th Annual Conference of the IEEE Industrial Electronics Society.*

Goodwin, B. (1994). *How the leopard changed its spots: The evolution of complexity.* New York: Charles Scribner's Sons.

Haykin, S. (1999). *Neural networks: A comprehensive foundation.* Upper Saddle River, NJ: Prentice Hall.

Hebb, D. O. (1949). *The organization of behavior.* New York: John Wiley & Sons.

Hebb, D. O. (1959). A neurophysiological theory. In S. Koch (ed.), *Psychology: A study of science* (Vol 1, pp. 629-643). New York: McGraw-Hill.

Hergenhahn, B., & Olson, M. (2000). *An introduction to theories of learning* (6th ed.). Upper Saddle River, NJ: Prentice Hall.

Johnson, J, Schamschula, M., Inguva, R., & Caulfield, H. (1998). Pulse-coupled neural network sensor fusion. *Processing of the international society for optical engineering, USA, 3376*, 219-226.

Klein, R. (1999). The hebb legacy. *Canadian Journal of Experimental Psychology, 53*(1), 1-3.

Lemley, B. (1984). Synesthesia: Seeing is feeling. *Psychology Today*, 65.

Lemley, B. (1999). Do you see what they see? *Discover, 20*(12), 79-87.

Levy, S. (1992). *Artificial life: The quest for a new creation.* New York: Pantheon Books.

Lovelace, C. T., Grossenbacher, P. G., & Crane, C. A. (1999). *Functional connectivity underlying synesthetic perception: theories and data.* <http://www.wfubmc.edu/nba/postdocs/lovelace/cnsposter99/cns.html> [January 24, 2001].

Martin, A., Haxby, J., Lalonde, F., Wiggs, C., & Ungerleider, L. (1995). Discrete cortical regions associated with knowledge of color and knowledge. *Science, 270,* 102-104.

Mattingley, J., Rich, A., Yelland, G., & Bradshaw, J. (2001). Unconscious priming eliminates automatic binding of color and alphanumeric form in synaesthesia. *Nature, 410*, 580-582.

Maurer, D. (1993). Neonatal synesthesia: Implications for the processing of speech and faces. In de Boysson-Bardies, B., de Schonen, S., Jusczyk, P., McNeilage, P., & Morton, J. (eds.) *Developmental Neurocognition: Speech and face processing in the first year of life.* Dordrecht: Kluwer Academic Publishers.

Milner, P. (1993). The mind and Donald O. Hebb. *Scientific American, 268*(1), 124-129.

Miltner, W., Braun, C., Arnold, M., White, H., & Taub, E. (1999). Coherence of gamma-band EEG as a basis for associative learning. *Nature, 397*, 434-436.

Moravec, H. (1988). *Mind children: The future of robot and human intelligence.* Cambridge, MA: Harvard University Press.

Moravec, H. (1999). *Robot: Mere machine to transcendent mind.* New York: Oxford University Press.

Murphy, R., & Arkin R. (1992). SFX: an architecture for action-oriented sensor fusion. *Proceedings of the International Conference on Intelligent Robotics and Systems, (IROS 1992),* 1079-1086.

Murphy, R. (1994). Sensor fusion. In M. Arbib (ed.), *The handbook of brain theory and neural networks.* (pp. 857-860). Cambridge, MA: MIT Press.

Murphy, R. (1996). Biological and cognitive foundations of intelligent sensor fusion. *IEEE Transactions on Systems, Man, and Cybernetics, 20*(1), 42-45.

Murphy, R. (1999). Dempster-shafter theory for sensor fusion in autonomous mobile robots. *IEEE Transcation on Robotics and Automation, 14*(2), 197-206.

Murphy, R. (2000). *Introduction to ai robotics.* Cambridge, MA: MIT Press.

Nicholls, J., Martin, A., & Wallace, B., (1992). *From neuron to brain.* Sunderland, MA: Sinauer Associates.

Paulesu, E., Harrison, J., Baron-Cohen, S., Watson, J., Goldstein, L., Heather, J., et al. (1995). The physiology of colored hearing. *Brain, 118*, 661-676.

Pulvermüller F. (1996). Hebb's concept of cell assemblies and the psychophysiology of word processing. *Psychophysiology, 33*(4), 317-333.

Ramachandran, V., & Hubbard, E. (2001). Synaesthesia - a window into perception, thought and language. *Journal of Consciousness Studies, 8*(12), 3-34.

Ramsøy, T. (2001). Seeing sounds, hearing tastes. *Science & Consciousness Review, 1(1)*, <http://www.geocities.com/science_consciousness_review/AR_103001_ synesthesia.htm> [March 14, 2002].

Rao, V., & Rao, H. (1995). *C++ neural networks & fuzzy logic.* New York: MIS:Press.

Rekleitis, I., Dudek, G., & Freedman, P. (1996). Just-in-time sensing: Efficiently combining sonar and laser range data for exploring unknown worlds. *Proceedings of the IEEE International Conference on Robotics and Automation, 1*, 667-672.

Robertson, L. (2001). Colour my i's blue. *Nature, 410*, 533-534.

Rodriguez, E., George, N., Lachaux, J., Martinerie, J., Renault, B., & Francisco J. (1999). Perception's shadow: Long-distance synchronization of human brain activity. *Nature, 397*, 430-433.

Rose, S., Gottfried, A., & Bridger, W. (1978). Effects of visual, haptic, and manipulator experiences on infants' visual recognition memory of objects. *Developmental Psychology, 17*, 90-98.

Sabbatini, R. (1999). Imitation of life: A history of the first robots. <http://www.epub.org.br/cm/n09/historia/turties_i.htm> [March 18, 2002].

Shams, L., Kamitani, Y., & Shimojo, S. (2000). What you see is what you hear. *Nature, 408*, 788.

Smock, T. (1999). *Physiological psychology: A neuroscience approach.* Upper Saddle River, NJ: Prentice-Hall.

Stein, B., & Meredith, M. (1993). *The merging of the senses.* Cambridge, MA: The MIT Press.

Susac, D. (1997). *More AI history.* <http://ai.about.com/compute/ai/library/weekl/aa080397.htm?iam=dpile&terms=%2B%22dona> [November 16, 2000].

Walter, W. G. (1950). An imitation of life. *Scientific American, 182*(5), 42-45.

Walter, W. G. (1951). A machine that learns. *Scientific American, 184*(8), 60-63.

Walter, W. G. (1953). *The living brain.* New York: W. W. Norton.

Ward, M. (1998). Walter's world. *New Scientist,* 159(2144), 54-55.

Zangaladze, A., Epstein, C. M., Grafton, S. T., & Sathlan, K. (1999). Involvement of visual cortex in tactile discrimination of orientation. *Nature, 401*, 587-590.

APPENDICES

APPENDIX A

Cross-Wired Network Simulation Software

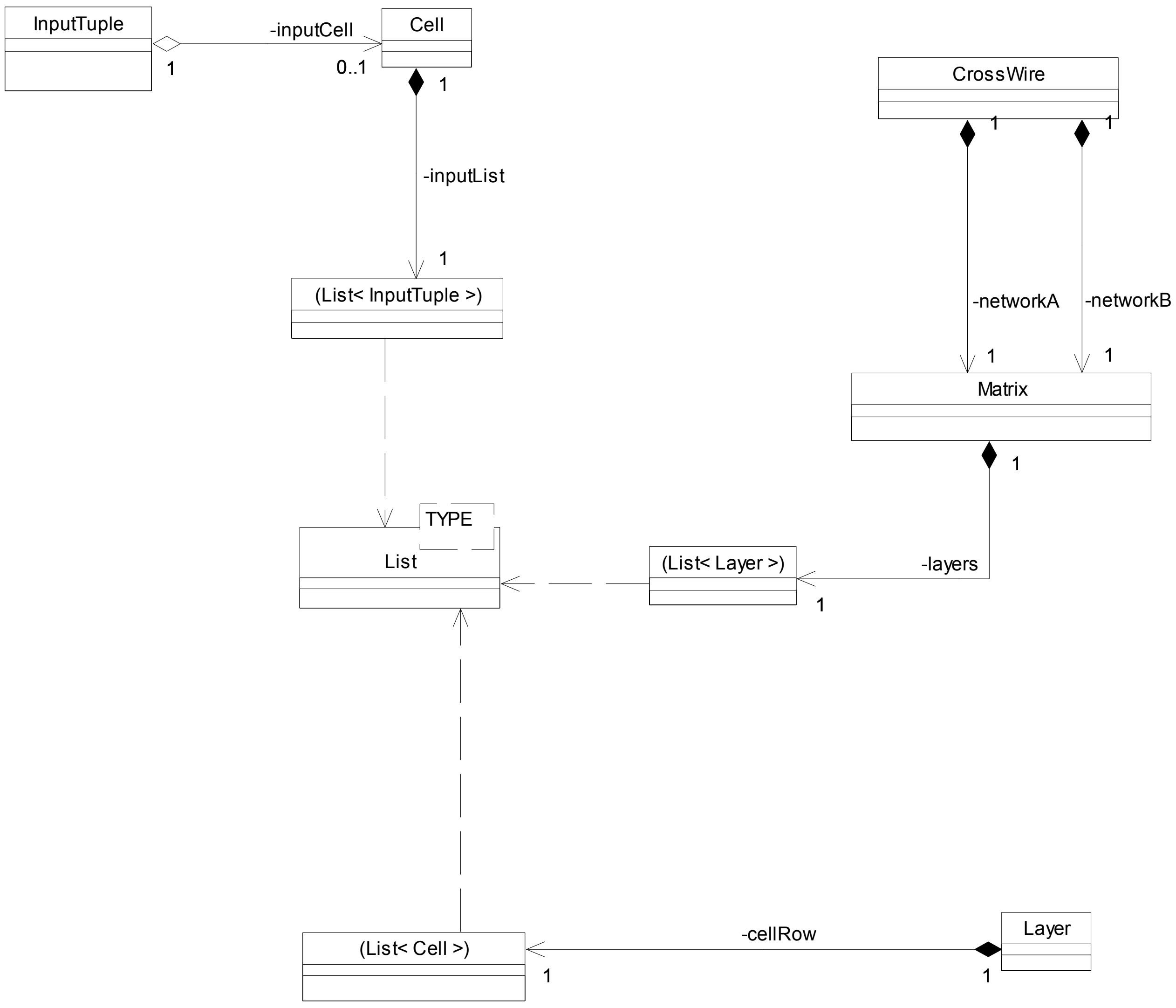

Figure 15. Cross-wired network simulation software UML diagram.

Simulation Software C++ Software Source Code

```
//------------------------------------------------------------------------------
//
// list.h
//
// Template Class - implements basic list container class - UNORDERED Collection
// The list is setup as deque - doubly linked list - but not taken advantage of currently.
//
//------------------------------------------------------------------------------
#ifndef LIST_H
#define LIST_H
#include <_null.h>
#include "truefalse.h"

template <class TYPE>
class List
{
private:
  class NODE
  {
  public:
    TYPE  data;                                    // data stored at node
    NODE* previous;                                // previous node in list
    NODE* next;                                    // next node in list

    NODE(void) { previous = next = NULL; }         // default CTOR
  };

  NODE* head;                                      // head of the list
  NODE* tail;                                      // tail of the list
  NODE* iteratorPosition;                          // position of iterator in the list
  int   iteratorEndOfListFlag;

  int nodeCount;

public:

  List(void)                                       // DEFAULT ctor
  {
    head                  = NULL;
    tail                  = NULL;
    iteratorPosition      = NULL;
    iteratorEndOfListFlag = FALSE;
    nodeCount             = 0;
  };

    List(const List<TYPE> &source);                // copy constructor
   ~List();                                        // dtor

    List &operator=(const List<TYPE> &rvalue);
    TYPE &operator[](int index);
    int  operator==(const List<TYPE> &rvalue) const;
    int  operator!=(const List<TYPE> &rvalue) const;

    int  insert(const TYPE &value);                // insert item into list
    int  insert(void);                             // insert new node - default values - empty
    int  remove(const TYPE &value);                // delete specified item from list
    TYPE iterator(void);                           // iterates over items in list
    void resetIterator(void) { iteratorPosition = head; }   // point to first item in list
    int  getCount(void) const{ return nodeCount; }          // how many items are in list
    void clear(void);                              // clear content of list - empty list
};

#include "list.cpp"                                // add member function implementation

#endif LIST_H
```

// list.cpp

```
// copy constructor
template<class TYPE>
List<TYPE>::List(const List<TYPE> &source)
{
  if(nodeCount)
  {
    NODE* current;

    current = source.head;

    do
    {
        insert(current->data);
        current = current->next;
    }
    while(current);
  }
}


// destructor
template<class TYPE>
List<TYPE>::~List()
{
        if(nodeCount)
        {
                NODE* next;

                do
                {
                        next = head->next;
                        delete head;
                        head = next;
                }
        while(next);
        }
}


template<class TYPE>
List &List<TYPE>::operator=(const List<TYPE> &rvalue)
{
        if(nodeCount)                           // if there are items in the list - clear it before copying...
        clear();

   NODE* current = rvalue.head;

        do
        {
                insert(current->data);
                current = current->next;
         }
         while(current);

   return *this;        // return reference to object pointed to by "this" - not a copy
}
```

```
template<class TYPE>
TYPE& List<TYPE>::operator[](int index)
{
        static TYPE Error;
        if( ((index >= 0) && (index < nodeCount)) && nodeCount )
        {
        NODE* current = head;

                for(int i=0; i<index; i++)
                {
                        current = current->next;
                }

                return current->data;
        }
        else
        {
                return Error;
                // it would be better to throw an exception here!
        }
}


template<class TYPE>
int List<TYPE>::operator==(const List<TYPE> &rvalue) const
{
        int result = FALSE;

  if( nodeCount == rvalue.nodeCount )
  {
        int i;

                NODE* leftList  = head;
    NODE* rightList = rvalue.head;

        for( i = 0; i < nodeCount; i++)
                if( leftList->data != rightList->data )
                {
                        break;
                 }
                else
                 {
                        leftList  = leftList->next;
                        rightList = rightList->next;
                }

                if( i == nodeCount )
                        result = TRUE;
  }
  return result;
}


template<class TYPE>
int List<TYPE>::operator!=(const List<TYPE> &rvalue) const
{
        return !(operator==(rvalue));
}
```

```
template<class TYPE>
int List<TYPE>::insert(const TYPE &value)
{
  int   success = FALSE;                                            // success of operartion
  NODE* newNode = new NODE;                                         // pointer to new node

  if(newNode)
  {
    newNode->data = value;

    if(!nodeCount)                                                  // if the list is empty
    {
      head = tail = newNode;                                        // then head and tail are the same
      iteratorPosition = head;                                      // this is an issue to be addressed - who and when sets this
    }
    else                                                            // add new node to end of list
    {
      newNode->previous = tail;
//    newNode->next     = NULL;                                     // by default - end of list
      tail->next        = newNode;                                  // tail next node point to new node
      tail              = newNode;                                  // tail is now the new node
    }

    ++nodeCount;

    success = TRUE;
  }
  return success;
}


// Insert new node of TYPE - empty
template<class TYPE>
int List<TYPE>::insert(void)
{
   TYPE newType;

   return insert(newType);
}


template<class TYPE>
int List<TYPE>::remove(const TYPE &value)
{
  int found = 0;

  if(head)                                                          // if there are NO nodes in the list then how can one be removed?
  {
    NODE* current = head;
    NODE* temp;

    do
    {
      if( current->data == value )
      {
        ++found;                                                    // number of occurrences deleted
        --nodeCount;

        if(current->previous)                                       // point around node being deleted
          current->previous->next = current->next;
        else
        {
          current->next->previous = NULL;
          head = current->next;
        }

        if(current->next)
          current->next->previous = current->previous;
        else
        {
          current->previous->next = NULL;
          tail = current->previous;
        }

        temp = current;                                        // copy so it can be deleted!
        current = temp->next;                                  // point to next node
        delete temp;                                           // delete node
      }
      else
        current = current->next;
    }
    while(current);
  }

  return found;
}           // remove(const TYPE &value)


// iterator - iterates over items in the list, Calls successively return each item in the list.
// When end of list is reached NULL is returned.   If list is empty NULL is returned.
// Note: revise... better algorithm!
template<class TYPE>
TYPE List<TYPE>::iterator(void)                                 // returns a copy - should retun a reference!
{
  if(iteratorEndOfListFlag)
  {
    iteratorEndOfListFlag = FALSE;
    iteratorPosition      = head;                              // start at the head of the list
    return (TYPE) NULL;
  }

  TYPE* returnItem = NULL;

  if(iteratorPosition)
  {
    returnItem       = &iteratorPosition->data;
    iteratorPosition =  iteratorPosition->next;

    if(!iteratorPosition)
      iteratorEndOfListFlag = TRUE;
  }
  return *returnItem;
}


template<class TYPE>
void List<TYPE>::clear(void)
{
        if(nodeCount)                                          // if there are NO nodes in the list then it is already clear.
        {
                NODE* current = head;
                NODE* temp;

                do
                {
                        temp = current;
                        current = current->next;
                        delete temp;
                }
                while(current);

                head    = NULL;
                tail     = NULL;
                nodeCount = 0;
        }
}
```

```
// cell.h ---------------------------------------------------------------------
//
// Stephen S. Seneker
//
// March 2002
// MALS Thesis Research
//------------------------------------------------------------------------------
//
// This class defines a cell used by a network class - component of an
// artifical neural network.
//
//------------------------------------------------------------------------------

#ifndef CELL_H
#define CELL_H

#include <stdlib.h>
#include <_null.h>
#include "list.h"

class Cell;                                         // forward reference...

class InputTuple
{
private:

        Cell* inputCell;                            // pointer to cell that is an input
        long double weight;                         // weight associated with preceding inputCell

public:

  InputTuple(void) { inputCell = NULL;
  weight =  ( (long double) (rand() % 1000)) / 1000.0; }          // randomize() needs to be called somewhere?

 // copy constructor
  InputTuple(const InputTuple &source) { inputCell = source.inputCell;
                                         weight  = source.weight; }

  InputTuple(const int &value) { inputCell = NULL;
                                 weight    = (long double) value; }

  InputTuple &operator=(const InputTuple &rvalue) {  inputCell = rvalue.inputCell;
                                                     weight    = rvalue.weight;
                                                     return *this; } // return reference to object pointed to by "this" - not a copy

  InputTuple &operator=(Cell* &rvalue) { inputCell = rvalue;
                                         return *this; }

  InputTuple &operator=(long double &rvalue) { weight = rvalue;
                                               return *this; }

  int operator==(const InputTuple &rvalue) const { return inputCell == rvalue.inputCell; }
  int operator==(const void*      &rvalue)     const { return inputCell == rvalue; }
  int operator!=(const InputTuple &rvalue)  const { return inputCell != rvalue.inputCell; }
  int operator!=(const void*      &rvalue)      const { return inputCell != rvalue; }

  long double& getWeight(void)   { return weight; }
  long double* getWeightPtr(void) { return &weight; }
  void   setWeight(const long double &newWeight) { weight = newWeight; }
  void   setCell(Cell* &inCell)  { inputCell = inCell; }
  Cell*  getCell(void)  { return inputCell; }
  void   updateWeight(long double factor)  { weight *= factor; }

};
```

```
class Cell
{
private:

   long double cellState;                         // Current state of the cell - value of its current output
   long double newState;                          // Next State of the cell - value of the output after update

   List<InputTuple> inputList;                    //List of cells this cell receives input from

public:

    Cell(void);                                   // ctor - default
    Cell(const long double &defaultState);        // ctor - initilize current state
    Cell(const Cell &source);                     // ctor - copy
   ~Cell();                                       // dtor

   // ---- Operators -----------------------------------------------------------------
   long double &operator[](const int index);       // return weight for Kth input tuple
   Cell &operator=(const Cell &rvalue);

   int operator==(const Cell  &rvalue) const;
   int operator==(const void* &rvalue) const { return (((cellState == 0.0) && (newState == 0.0) && !inputList.getCount()) && !rvalue); }
   int operator!=(const Cell  &rvalue) const { return !(operator==(rvalue)); }
   int operator!=(const void* &rvalue) const { return !(operator==(rvalue));}

   long double getState(void);                               // return the current state of this cell
   void  setState(const long double &newState);              // set the state of this cell - used for input layer cells
   void  nextState(void);                                    // computer next state of this cell
   void  updateState(void);                                  // update cell to reflect new state (calculated by nextState())
   void  addInputCell(Cell* inputCell);                      // adds a cell to the list of inputs for this cell
   void  addInputCell(Cell* inputCell, long double weight);  // adds a cell to the list of inputs for this cell with an associated weight
   int   getCount(void) { return inputList.getCount(); }     // number of inputs and weights for this cell

};

#endif CELL_H
```

```
// cell.cpp -------------------------------------------------------------------
//
// Implements cell class.
//
// Stephen S. Seneker
//
// March 2002
// MALS Thesis Research
//
//-----------------------------------------------------------------------------

#include <iostream.h>
#include <math.h>
#include "cell.h"


// default CTOR

Cell::Cell(void)
{
   cellState = 0.0;
   newState  = 0.0;
}


// initialize CTOR

Cell::Cell(const long double &defaultState)
{
   cellState = defaultState;
   newState  = 0.0;
}


// copy CTOR

Cell::Cell(const Cell &source)
{
  cellState = source.cellState;
  newState  = source.newState;
  inputList = source.inputList;
}


// DTOR

Cell::~Cell()
{
  inputList.clear();
}


long double& Cell::operator[](const int index)
{
        static long double Error = NULL;

        if( (index >=0) && (index < inputList.getCount()) && inputList.getCount() )
        {
                return inputList[index].getWeight();
        }
        else
        {
                return Error;
        }
}
```

```
Cell& Cell::operator=(const Cell &rvalue)
{
  cellState = rvalue.cellState;
  newState  = rvalue.newState;

  for(int k = 0; k < rvalue.getCount();  k++)
          inputList.insert(rvalue[k]);

  return *this;
}


int Cell::operator==(const Cell &rvalue) const
{
          return ( (inputList == rvalue.inputList) && (cellState == rvalue.cellState) && (newState == rvalue.newState));
}


// Returns the current state of this cell.
long double Cell::getState(void)
{
          return cellState;
}


// sets the state of this cell
// used for input layer cells
void Cell::setState(const long double &newCellState)
{
 cellState = newCellState;
}


// Compute the next state for this cell, i.e., it's next output value.
void Cell::nextState(void)
{
          if( inputList.getCount() )
          {
                    InputTuple tuple;
                    Cell* inputCell;

                    newState = 0.0;
                    while( (tuple = inputList.iterator()) != NULL )
                    {
                              inputCell = tuple.getCell();

                              newState += tuple.getWeight() * inputCell->getState();
                    }
                    // sigmoid Function
                    newState = (1.0 / (1.0 + expl(-1.0 * newState)));
          }
}


// Update the output of this cell.
void Cell::updateState(void)
{
          cellState = newState;
}


// Assigns a cell as input to this cell.
void Cell::addInputCell(Cell* inputCell)
{
  InputTuple newTuple;
  newTuple = inputCell;
  inputList.insert(newTuple);
}
```

```
// Assigns a cell as input to this cell and a default weight.

void Cell::addInputCell(Cell* inputCell, long double weight)
{
    InputTuple newTuple;

    newTuple = inputCell;
    newTuple = weight;

    inputList.insert(newTuple);
}
```

```
// layer.h ------------------------------------------------------------------
//
// Declares Layer Class - a collection of cells.
//
//---------------------------------------------------------------------------

#ifndef LAYER_H
#define LAYER_H

#include "list.h"
#include "cell.h"

class Layer
{
private:

        List<Cell> cellRow;             // A layer is a ROW of cells... a list of cells.

public:

        Layer(void);
        Layer(int count);
        Layer(const Layer &source) { cellRow = source.cellRow; }                // copy constructor
        ~Layer();

        int operator==(const Layer &rvalue) const;
        int operator==(const void* &rvalue) const { return (!cellRow.getCount() && !rvalue); }
        int operator!=(const Layer &rvalue) const { return !(operator==(rvalue)); }
        int operator!=(const void* &rvalue) const { return !(operator==(rvalue)); }

        Cell  &operator[](const int index);
        Layer &operator=(const Layer &rvalue);

        void addCell(void);
        void addCells(int count);                  // layer will consist of count cells
        void nextState(void);
        void updateState(void);
        int  getCount(void)                       { return cellRow.getCount(); }

};

#endif
```

```
//------------------------------------------------------------------------------
//
// layer.cpp
//
//------------------------------------------------------------------------------
//
// Implements layer class - a collection of cells.
//
//------------------------------------------------------------------------------

#include <iostream.h>
#include <iomanip.h>
#include <_null.h>
#include "cell.h"
#include "layer.h"
#include "list.h"
#include "truefalse.h"


Layer::Layer(void)
{
}


Layer::Layer(int count)
{
	addCells(count);
}


Layer::~Layer()
{
	cellRow.clear();
}


// Two layers are equal if all the cells have the same STATE...
int Layer::operator==(const Layer &rvalue) const
{
	int result = TRUE;

	if( getCount() == rvalue.getCount() )
	{
		for(int k = 0; k < getCount(); k++)
		{
			if( cellRow[k].getState() != rvalue.cellRow[k].getState() )
			{
				result = FALSE;
				break;
			}
		}
	}

	return result;
}


Cell& Layer::operator[](const int index)
{
	static Cell Error;

	if( (index >=0) && (index < cellRow.getCount()) && cellRow.getCount() )
	{
		return cellRow[index];
	}
	else
	return Error;					// revision should THROW an exception...
}
```

```
Layer& Layer::operator=(const Layer &rvalue)
{
        cellRow.clear();

        for(int i = 0; i < rvalue.getCount(); i++)
        {
                cellRow.insert( rvalue[i] );
        }

        return *this;
}


void Layer::addCell(void)
{
        cellRow.insert();
}


void Layer::addCells(int count)
{
        if(count > 0)
        {
                for(int i = 0; i < count; i++)
                        addCell();
        }
}


void Layer::nextState(void)
{
  for(int i = 0; i < cellRow.getCount(); i++)
        cellRow[i].nextState();
}


void Layer::updateState(void)
{
  for(int i = 0; i < cellRow.getCount(); i++)
        cellRow[i].updateState();
}
```

```
// matrix.h -----------------------------------------------------------------------------
//
// Declares Matrix Class
//
// A matrix is a collection of layers.
// A layer is a collection of cells.
//
//-----------------------------------------------------------------------------

#ifndef MATRIX_H
#define MATRIX_H

#include "layer.h"
#include "list.h"

class Matrix
{
private:

        List<Layer> layers;

public:

        Matrix(void);
        Matrix(char* filename);
        Matrix(int numberOfLayers);
        ~Matrix();

        Matrix &operator=(const Matrix &source) { //layers = source.layers; - deep copy needs to be DONE for future work.
                                                   return *this; }

        Layer &operator[](const int index);

        Layer &addLayer(void);
        int    addLayers(int count);                         // add count number of layers to the matrix

        void nextState(void);                                // generate next state of matrix
        void updateState(void);                              // update the state of the matrix
        void update(void);                                   // update state of matrix - one layer at time
        int  getCount(void) { return layers.getCount(); }    // number of layers in matrix

        void interconnectLayers(void);                       // FULLY interconnect layers of the matrix

        void loadMatrix(char* filename);                     // load matrix from file
        void saveMatrix(char* filename);                     // save matrix to file

};

#endif
```

```
//------------------------------------------------------------------------------
//
// matrix.cpp
//
//------------------------------------------------------------------------------
//
// Implements matrix class - a network/lattice of cells.
//
//------------------------------------------------------------------------------
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include "matrix.h"

Matrix::Matrix()
{
}


Matrix::Matrix(char* filename)

{
        loadMatrix(filename);
}


Matrix::Matrix(int numberOfLayers)
{
        addLayers(numberOfLayers);
}


Matrix::~Matrix()
{
        layers.clear();
}


Layer& Matrix::operator[](const int index)
{
        static Layer Error;

        if( (index >=0) && (index < layers.getCount()) && layers.getCount() )
        {
                return layers[index];
        }
        else
                return Error;                                   // revision should THROW an exception...
}


Layer& Matrix::addLayer(void)
{
        layers.insert();                                // insert an empty layer
        return layers[layers.getCount() - 1];
}


int Matrix::addLayers(int numberOfLayers)
{

        for(int i = 0; i < numberOfLayers; i++)
        {
                addLayer();
        }

        return TRUE;                                    // optimitic - must handle errors betters!
}
```

```
void Matrix::nextState(void)
{
        for(int i = 1; i < layers.getCount(); i++) // layer 0 is the input layer - skip
                layers[i].nextState();
}


void Matrix::updateState(void)
{
        for(int i = 1; i < layers.getCount(); i++) // layer 0 is the input layer - skip
                layers[i].updateState();
}


void Matrix::update(void)
{
        for(int i = 1; i < layers.getCount(); i++) // layer 0 is the input layer - skip
        {
                layers[i].nextState();          // generate next state for layer i
                layers[i].updateState();        // update state for layer i
        }
}


// Fulley interconnect layers...
void Matrix::interconnectLayers(void)
{
        int clyr;     // current layer
        int plyr;     // previous layer
        int ccel;   // cell in current layer
        int pcel;   // cell in previous layer

        for(clyr = 1; clyr < layers.getCount(); clyr++)
        {
                plyr = clyr - 1;
                for(pcel = 0; pcel < layers[plyr].getCount(); pcel++)
                {
                        for(ccel = 0; ccel < layers[clyr].getCount(); ccel++)
                        {
                                layers[clyr][ccel].addInputCell(&layers[plyr][pcel]);
                        }
                }
        }

}


// load matrix from specified stream/file...
void Matrix::loadMatrix(char *filename)
{
        ifstream wgts(filename);

        if(!wgts)
        {
                cout << "Matrix::loadMatrix - Cannot open file: " << filename << endl;
                exit(0);
        }

        // Create Layers...
        int numberOfLayers;

        wgts >> numberOfLayers;

        addLayers(numberOfLayers);
```

```
	// Add Cells to each layer...
	for(int lyr = 0; lyr < numberOfLayers; lyr++)
	{
		int numberOfCells;

		wgts >> numberOfCells;

		layers[lyr].addCells(numberOfCells);

	}

	// interconnect layers...

	interconnectLayers();

	int lyr, cel, wgt;
	float weight;

	for(lyr = 1; lyr < layers.getCount(); lyr++)
	{
		for(cel = 0; cel < layers[lyr].getCount() ; cel++)
		{
			for(wgt = 0; wgt < layers[lyr][cel].getCount() ; wgt++)
			{
				wgts >> weight;
				layers[lyr][cel][wgt] = weight;
			}
		}
	}

	wgts.close();
}


void Matrix::saveMatrix(char* filename)
{
	ofstream wgts(filename);

	 if(!wgts)
	{
		cout << "Matrix::saveMatrix -Cannot open file: " << filename << endl;
	exit(0);
	}

	wgts.setf(ios::showpoint | ios::fixed);
	wgts.precision(12);

	wgts << layers.getCount() << endl;				// line 0: number of layers

	for(int i = 0; i < layers.getCount(); i++)			// line 1: number of cells in each layer
	{
		wgts << layers[i].getCount() << " ";
	}
	wgts << endl;

	for(int lyr = 1; lyr < layers.getCount(); lyr++)		// weights starting at layer 1 cell 1 - layer zero is input layer
	{
		for(int cel = 0; cel < layers[lyr].getCount(); cel++)
		{
			for(int wgt = 0; wgt < layers[lyr][cel].getCount(); wgt++)
			{
				wgts << layers[lyr][cel][wgt] << " ";
			}
			wgts << endl;
		}
	}
	wgts.close();
}
```

```
// crosswire.h ------------------------------------------------------------
//
// Declares Cross-Wired network class.
// Simulation proper.
//
//-----------------------------------------------------------------------------

#ifndef CROSSWIRED_H
#define CROSSWIRED_H

#include "matrix.h"

class CrossWire
{
private:

        Matrix networkA;
        Matrix networkB;

public:

        CrossWire();
        CrossWire(char* filenameA, char* filenameB);
        ~CrossWire();

        void linkNetworks(void);                                  // link network A and B
        void trainCrossWire(void);                                // train cross-wire weights

        void nextState(void);                                     // compute next state for cross-wired networks
        void updateState(void);                                   // update output of cross-wired networks
        void update(void);                                        // update networks
        void updateStable(void);

        void loadNetworksAB(char* filenameA, char* filenameB);    // without cross-wired connections
        void loadCrossWire(char* filenameA, char* filenameB);     // save cross-wired weights
         void saveCrossWire(char* filenameA, char* filenameB);    // save cross-wried weights

        void displayNetworks(void);                               // display input and ouput for each network
        void setInputs(void);                                     // set inputs for each network - by hand
        int  setInputs(ifstream &wgtsA, ifstream &wgtsB);         // set cross-wird network inputs
        void runCrossWire(char* inFileA, char* inFileB);          // run networks using fileA/fileB as inputs
};


#endif
```

```
//------------------------------------------------------------------------------
// crosswire.cpp
//
//------------------------------------------------------------------------------
//
// Implements cross-wired network class.
// Networks to be cross-wired must have identical layout - same number layers
// with identical rows and columns.
//
//------------------------------------------------------------------------------

#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>

#include <stdlib.h>                     // randomize
#include <time.h>                       // randomize
#include <math.h>                       // sqrt()

#include "crosswire.h"

CrossWire::CrossWire()
{
}


CrossWire::CrossWire(char* filenameA, char* filenameB)
{
        loadCrossWire(filenameA, filenameB);
}

CrossWire::~CrossWire()
{
}


// cross-wire networkA and networkB
// BOTH must have same configuration!
void CrossWire::linkNetworks(void)
{
        // link network A to network B
        for(int lyr = 1; lyr < networkA.getCount(); lyr++)
        {
                for(int cel = 0; cel < networkA[lyr].getCount(); cel++)
                {
                        networkA[lyr][cel].addInputCell(&networkB[lyr][cel]);   // link this cell to correcponding cell in netowrkB
                        networkB[lyr][cel].addInputCell(&networkA[lyr][cel]);  // link this cell to corresponding cell in networkA
                }
        }
}
```

```
void CrossWire::trainCrossWire(void)
{
//----------------------------------------------------------------------------
// Initial Hebbian Training
//----------------------------------------------------------------------------
        // Propportion of new weight to average of existing weights.
        const long double phi = 0.618033987498948482;       // Golden Ratio
        const long double Phi = 1.618039987498948482;

//------- Network A ----------------------------------------------------------
        for(int lyr = 1; lyr < networkA.getCount(); lyr++)
        {
                for(int cel = 0; cel < networkA[lyr].getCount(); cel++)
                {
                        long double numberOfWeights = networkA[lyr][cel].getCount() - 1;
                        long double wgtSum          = 0.0;

                        int wgt;
                        for(wgt = 0; wgt < numberOfWeights; wgt++)
                        {
                                wgtSum += networkA[lyr][cel][wgt];
                        }

                        networkA[lyr][cel][wgt] = (wgtSum / numberOfWeights) * phi;
                }
        }

//------- Network B ----------------------------------------------------------
        for(int lyr = 1; lyr < networkB.getCount(); lyr++)
        {
                for(int cel = 0; cel < networkB[lyr].getCount(); cel++)
                {
                        long double numberOfWeights = networkB[lyr][cel].getCount() - 1;
                        long double wgtSum          = 0.0;

                        int wgt;
                        for(wgt = 0; wgt < numberOfWeights; wgt++)
                        {
                                wgtSum += networkB[lyr][cel][wgt];
                        }

                        networkB[lyr][cel][wgt] = (wgtSum / numberOfWeights) * Phi;
                }
        }
```

```
//-------------------------------------------------------------------------------
// Residual Hebbian Training
//-------------------------------------------------------------------------------
        const long double bias    = phi * 0.000015;
        const long double Bias    = Phi * 0.00015;
        const int    MaxPass = 5500000;

//------- Network A ------------------------------------------------------------
        for(int lyr = 1; lyr < networkA.getCount(); lyr++)
        {
                int crossWeight  = networkA[lyr][0].getCount() - 1;

                // cross-wired connection weight is last weight in cell weight list; each cell in a layer has the same number of inputs
                int cellsInLayer = networkA[lyr].getCount();                // number of cells in the layer
                List<long double> weights;                                  // list of new cross-wired connection weights
                List<long double> lastWeights;                    // weights computed in previous iteration - used to detect convergence

                for(int cel = 0; cel < cellsInLayer; cel++)                 // make a list of cross-wired connection weights for this layer
                {
                        weights.insert(networkA[lyr][cel][crossWeight]);
                }

                int pass = 0;
                do
                {
                        lastWeights = weights;

                        long double wgtSum  = 0.0;
                        long double wgtAvg  = 0.0;
                        long double cellCnt = cellsInLayer; // - 1;

                        for(int i = 0; i < cellsInLayer; i++)
                        {
                                for(int j = 0; j < cellsInLayer; j++)
                                {
                                        //if( j != i )
                                        wgtSum += weights[j];
                                }

                                wgtAvg      = wgtSum / cellCnt;
                                weights[i] -= wgtAvg * bias;

                        }
                        ++pass;
                }
                while( (lastWeights != weights) && (pass < MaxPass) );

        cout << "Passes: " << pass << endl;

        for(int cel = 0; cel < cellsInLayer; cel++)                         // update weights in layer
                        networkA[lyr][cel][crossWeight] = weights[cel];

        } // for() - layer
```

```
//------- Network B -----------------------------------------------------------
        for(int lyr = 1; lyr < networkB.getCount(); lyr++)
        {
                int crossWeight  = networkB[lyr][0].getCount() - 1;

                // cross-wired connection weight is last weight in cell weight list; each cell in a layer has the same number of inputs
                int cellsInLayer = networkB[lyr].getCount();            // number of cells in the layer

                List<long double> weights;                              // list of new cross-wired connection weights
                List<long double> lastWeights;                  // weights computed in previous iteration - used to detect convergence

                for(int cel = 0; cel < cellsInLayer; cel++)                     // make a list of cross-wired connection weights for this layer
                {
                        weights.insert(networkB[lyr][cel][crossWeight]);
                }

                int pass = 0;
                do
                {
                        lastWeights = weights;

                        long double wgtSum  = 0.0;
                        long double wgtAvg  = 0.0;
                        long double cellCnt = cellsInLayer; // - 1;


                        for(int i = 0; i < cellsInLayer; i++)
                        {
                                for(int j = 0; j < cellsInLayer; j++)
                                {
                                        // if( j != i )
                                        wgtSum += weights[j ];
                                }

                                wgtAvg      = wgtSum / cellCnt;
                                weights[i] -= wgtAvg * bias;
                        }

                        ++pass;
                }
                while( (lastWeights != weights) && (pass < MaxPass) );

                cout << "Passes: " << pass << endl;

                for(int cel = 0; cel < cellsInLayer; cel++)                             // update weights in layer
                        networkB[lyr][cel][crossWeight] = weights[cel];

        } // for() - layer
}  // trainCrossWire(void)


void CrossWire::nextState(void)
{
        networkA.nextState();
        networkB.nextState();
}


void CrossWire::updateState(void)
{
        networkA.updateState();
        networkB.updateState();
}
```

```
void CrossWire::update(void)
{
        networkA.nextState();
        networkA.updateState();
        networkB.nextState();
        networkB.updateState();
}


// update - output converges to steady state
void CrossWire::updateStable(void)
{
        Layer lastOutputA;
        Layer lastOutputB;

        int stopFlag = FALSE;
        int outLyr = networkA.getCount() - 1;

        update();                               // prime

        lastOutputA = networkA[outLyr];
        lastOutputB = networkB[outLyr];

        int i = 0;
        do
        {
                update();                       // update state of both networks

                if((lastOutputA == networkA[outLyr]) && (lastOutputB == networkB[outLyr]))
                {
                        stopFlag = TRUE;
                }
                else
                {
                        lastOutputA = networkA[outLyr];
                        lastOutputB = networkB[outLyr];
                }
                ++i;

                if(i==100)
                        stopFlag = TRUE;
        }
        while(!stopFlag);
}


void CrossWire::loadNetworksAB(char* filenameA, char* filenameB)
{
        networkA.loadMatrix(filenameA);
         networkB.loadMatrix(filenameB);
        linkNetworks();
}
```

```
void CrossWire::loadCrossWire(char* filenameA, char* filenameB)
{
        ifstream wgtsA(filenameA);
        ifstream wgtsB(filenameB);

        if(!wgtsA)
        {
                cout << "CrossWire::loadCrossWire - Cannot open fileA: " << filenameA << endl;
                exit(0);
        }

        if(!wgtsB)
        {
                cout << "CrossWire::loadCrossWire - Cannot open fileB: " << filenameB << endl;
                exit(0);
        }

        // Create Layers...
        int numberOfLayersA;
        int numberOfLayersB;

        wgtsA >> numberOfLayersA;
        wgtsB >> numberOfLayersB;

        if( numberOfLayersA != numberOfLayersB )
        {
                cout << "CrossWire::loadCrossWire - LayersA != LayersB" << endl;
                exit(0);
        }

        // add layers...
        networkA.addLayers(numberOfLayersA);
        networkB.addLayers(numberOfLayersB);

        // Add Cells to each layer...
        for(int lyr = 0; lyr < numberOfLayersA; lyr++)
        {
                int numberOfCellsA;
                int numberOfCellsB;

                wgtsA >> numberOfCellsA;
                wgtsB >> numberOfCellsB;

                if( numberOfCellsA != numberOfCellsB )
                {
                        cout << "CrossWire::loadCrossWire - cellsA != cellsB - layer:" << lyr << endl;
                        exit(0);
                }

                networkA[lyr].addCells(numberOfCellsA);
                networkB[lyr].addCells(numberOfCellsB);
        }

        // interconnect layers...

        networkA.interconnectLayers();
        networkB.interconnectLayers();

        // link NetworkA and NetworkB

        linkNetworks();
        // read weights from streams (files)

        int lyr, cel, wgt;

        for(lyr = 1; lyr < networkA.getCount(); lyr++)
        {
                for(cel = 0; cel < networkA[lyr].getCount() ; cel++)
                {
                        for(wgt = 0; wgt < networkA[lyr][cel].getCount() ; wgt++)
                        {
                                wgtsA >> networkA[lyr][cel][wgt];
                                wgtsB >> networkB[lyr][cel][wgt];
                        }
                }
        }

  wgtsA.close();
  wgtsB.close();

}       // loadCrossWire(char* filenameA, char* filenameB)


void CrossWire::saveCrossWire(char* filenameA, char* filenameB)
{
        networkA.saveMatrix(filenameA);
        networkB.saveMatrix(filenameB);
}


void CrossWire::displayNetworks(void)
{
        int outLyrA = networkA.getCount() - 1;
        int outLyrB = networkB.getCount() - 1;

        cout << "Network A: ";
        cout.precision(1);
        for(int i = 0; i < networkA[0].getCount(); i++)
                cout << networkA[0][i].getState() << ", ";
        cout << " :  ";

        cout.precision(6);
        for(int i = 0; i < networkA[outLyrA].getCount(); i++)
                cout << networkA[outLyrA][i].getState() << ", ";
        cout << endl;

//-----------------------------------------------------------------------------

        cout << "Network B: ";
        cout.precision(1);
        for(int i = 0; i < networkB[0].getCount(); i++)
                cout << networkB[0][i].getState() << ", ";
        cout << " :  ";

        cout.precision(6);
        for(int i = 0; i < networkB[outLyrB].getCount(); i++)
                cout << networkB[outLyrB][i].getState() << ", ";
        cout << endl;
}
```

```
// setInputs - user entered
void CrossWire::setInputs(void)
{
        long double cellState;

        cout << "Inputs for networkA(" << networkA[0].getCount() << "): ";
        for(int i = 0; i < networkA[0].getCount(); i++)
        {
                cin >> cellState;
                networkA[0][i].setState(cellState);
        }
        cout << endl;

        cout << "Inputs for networkB(" << networkB[0].getCount() << "): ";
        for(int i = 0; i < networkB[0].getCount(); i++)
        {
                cin >> cellState;
                networkB[0][i].setState(cellState);
        }
        cout << endl;
}


// read inputs from file stream
// networkA and networkB must be same architecture
int CrossWire::setInputs(ifstream &wgtsA, ifstream &wgtsB)
{
        long double cellStateA;
        long double cellStateB;

        int i;
        for(i = 0; (i < networkA[0].getCount()) && !wgtsA.eof() && !wgtsB.eof(); i++)
         {
                wgtsA >> cellStateA;
                wgtsB >> cellStateB;
                networkA[0][i].setState(cellStateA);
                networkB[0][i].setState(cellStateB);
        }

        return !(networkA[0].getCount() - i);
}
```

```
// 1. read input files inFileA/inFileB
// 2. display inputs and outputs
// 3. repeat 4 and 5 for all inputs
void CrossWire::runCrossWire(char* inFileA, char* inFileB)
{
  ifstream inVcA(inFileA);                    // input Vector A
  ifstream inVcB(inFileB);                    // input Vector B

  if(!inVcA)
  {
        cout << "CrossWire::runnCrossWire() - Cannot open input file: " << inFileA << endl;
         exit(0);
  }

  if(!inVcB)
  {
        cout << "CrossWire::runnCrossWire() - Cannot open input file: " << inFileB << endl;
        exit(0);
  }

  while( setInputs(inVcA, inVcB) )
  {

        updateStable();

        displayNetworks();
  }

  inVcA.close();
  inVcB.close();
}



//----------------------------------------------------------------------------
//
// Cross Wire ------------------------------------------------------------------
//
//----------------------------------------------------------------------------
void main(void)
{
        CrossWire networks;

        networks.loadNetworksAB("netA1288s.wgts", "netB1288s.wgts");        // load AND interlink networks

         networks.linkNetworks();
                                                                            // link networks

        networks.trainCrossWire();
                                                                            // train cross-wired connections

        networks.runCrossWire("inputA128.dat", "inputB128.dat");            // run cross-wired with these input lists

        networks.saveCrossWire("testA128.wgts", "testB128.wgts");           // save new weights
}
```

APPENDIX B

Network Training Data

Table 1

<u>Network A Training Set</u>

Pattern	Input Vector												Expected Output Vector							
1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
2	0	1	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0	0	0	1
3	0	1	0	0	0	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0
4	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
5	0	1	1	0	0	0	1	1	0	1	0	0	0	0	1	1	0	1	0	0
6	0	1	1	0	0	0	1	1	0	1	0	1	0	0	1	1	0	1	0	1
7	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1	0
8	0	1	1	1	0	0	1	1	0	1	1	1	0	0	1	1	0	1	1	1
9	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0
10	0	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	1	0	0	1
11	1	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1
12	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0
13	1	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	1
14	1	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
15	1	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1
16	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0	0	0	1	1	0
17	1	0	1	1	0	1	0	0	0	1	1	1	0	1	0	0	0	1	1	1
18	1	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0
19	1	0	1	0	0	1	0	0	1	0	0	1	0	1	0	0	1	0	0	1
20	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	1	0
21	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1
22	1	0	1	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1	0	0
23	1	0	1	0	0	1	0	0	1	1	0	1	0	1	0	0	1	1	0	1
24	1	0	1	0	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0
25	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	1	1	1
26	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0
27	1	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1
28	1	0	0	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0
29	1	0	0	1	0	1	0	1	0	0	1	1	0	1	0	1	0	0	1	1
30	1	0	1	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0
31	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
32	1	0	1	0	0	1	0	1	0	1	1	0	0	1	0	1	0	1	1	0
33	1	0	1	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	1
34	1	0	1	0	0	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0
35	1	0	1	0	0	1	0	1	1	0	0	1	0	1	0	1	1	0	0	1
36	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0
37	1	0	0	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0	0	1
38	1	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0	0	0	1	0
39	1	0	0	1	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	1
40	1	0	1	0	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	0
41	1	0	1	0	0	1	1	0	0	1	0	1	0	1	1	0	0	1	0	1
42	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
43	1	0	1	1	0	1	1	0	0	1	1	1	0	1	1	0	0	1	1	1
44	1	0	1	0	0	1	1	0	1	0	0	0	0	1	1	0	1	0	0	0
45	1	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1
46	1	0	1	0	0	1	1	0	1	0	1	0	0	1	1	0	1	0	1	0
47	1	0	1	1	0	1	1	0	1	0	1	1	0	1	1	0	1	0	1	1
48	1	0	1	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1	0	0
49	1	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1	0	1
50	1	0	1	0	0	1	1	0	1	1	1	0	0	1	1	0	1	1	1	0
51	1	0	1	1	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	1
52	1	1	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0
53	1	1	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	0	0	1
54	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	1	0	0	1	0
55	1	1	0	1	0	1	1	1	0	0	1	1	0	1	1	1	0	0	1	1
56	1	1	1	0	0	1	1	1	0	1	0	0	0	1	1	1	0	1	0	0
57	1	1	1	0	0	1	1	1	0	1	0	1	0	1	1	1	0	1	0	1
58	1	1	1	0	0	1	1	1	0	1	1	0	0	1	1	1	0	1	1	0
59	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
60	1	1	1	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
61	1	1	1	0	0	1	1	1	1	0	0	1	0	1	1	1	1	0	0	1
62	1	1	1	0	0	1	1	1	1	0	1	0	0	1	1	1	1	0	1	0

Table 2

Network B Training Set

Pattern	Input Vector												Expected Output Vector							
1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1
2	0	1	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	1	1
3	0	1	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	1
4	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
5	0	1	1	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	1
6	0	1	1	0	0	0	1	1	0	1	0	1	0	1	0	1	0	0	1	1
7	0	1	1	0	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	1
8	0	1	1	1	0	0	1	1	0	1	1	1	0	1	1	1	0	0	1	1
9	0	1	1	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1
10	0	1	1	0	0	0	1	1	1	0	0	1	1	0	0	1	0	0	1	1
11	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	1	0
12	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	1	0
13	1	0	0	1	0	1	0	0	0	0	1	1	0	0	1	1	0	1	1	0
14	1	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0
15	1	0	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0
16	1	0	1	0	0	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0
17	1	0	1	1	0	1	0	0	0	1	1	1	0	1	1	1	0	1	1	0
18	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1	1	0
19	1	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	0	1	1	0
20	1	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	0
21	1	0	1	1	0	1	0	0	1	0	1	1	1	0	1	1	0	1	1	0
22	1	0	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	1	1	0
23	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0
24	1	0	1	0	0	1	0	0	1	1	1	0	1	1	1	0	0	1	1	0
25	1	0	1	1	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	0
26	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1
27	1	0	0	0	0	1	0	1	0	0	0	1	0	0	0	1	0	1	1	1
28	1	0	0	0	0	1	0	1	0	0	1	0	0	0	1	0	0	1	1	1
29	1	0	0	1	0	1	0	1	0	0	1	1	0	0	1	1	0	1	1	1
30	1	0	1	0	0	1	0	1	0	1	0	0	0	1	0	0	0	1	1	1
31	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1
32	1	0	1	0	0	1	0	1	0	1	1	0	0	1	1	0	0	1	1	1
33	1	0	1	1	0	1	0	1	0	1	1	1	0	1	1	1	0	1	1	1
34	1	0	1	0	0	1	0	1	1	0	0	0	1	0	0	0	0	1	1	1
35	1	0	1	0	0	1	0	1	1	0	0	1	1	0	0	1	0	1	1	1
36	1	0	1	0	0	1	0	1	1	0	1	0	1	0	1	0	0	1	1	1
37	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	1	0	1	0	0
38	1	0	0	0	0	1	1	0	0	0	1	0	0	0	1	0	0	1	0	0
39	1	0	0	1	0	1	1	0	0	0	1	1	0	0	1	1	0	1	0	0
40	1	0	1	0	0	1	1	0	0	1	0	0	0	1	0	0	0	1	0	0
41	1	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	0	1	0	0
42	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	0
43	1	0	1	1	0	1	1	0	0	1	1	1	0	1	1	1	0	1	0	0
44	1	0	1	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	0	0
45	1	0	1	0	0	1	1	0	1	0	0	1	1	0	0	1	0	1	0	0
46	1	0	1	0	0	1	1	0	1	0	1	0	1	0	1	0	0	1	0	0
47	1	0	1	1	0	1	1	0	1	0	1	1	1	0	1	1	0	1	0	0
48	1	0	1	0	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0	0
49	1	0	1	0	0	1	1	0	1	1	0	1	1	1	0	1	0	1	0	0
50	1	0	1	0	0	1	1	0	1	1	1	0	1	1	1	0	0	1	0	0
51	1	0	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0	1	0	0
52	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1
53	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0	1	0	1	0	1
54	1	1	0	0	0	1	1	1	0	0	1	0	0	0	1	0	0	1	0	1
55	1	1	0	1	0	1	1	1	0	0	1	1	0	0	1	1	0	1	0	1
56	1	1	1	0	0	1	1	1	0	1	0	0	0	1	0	0	0	1	0	1
57	1	1	1	0	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1
58	1	1	1	0	0	1	1	1	0	1	1	0	0	1	1	0	0	1	0	1
59	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	0	1
60	1	1	1	0	0	1	1	1	1	0	0	0	1	0	0	0	0	1	0	1
61	1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	0	1	0	1
62	1	1	1	0	0	1	1	1	1	0	1	0	1	0	1	0	0	1	0	1

VITA

STEPHEN S. SENEKER

Personal Data: Date of Birth: March 21, 1971
Place of Birth: Norfolk, Virginia
Marital Status: Single

Education: Northeast State Technical Community College, Blountville, Tennessee;
Computer Engineering Technology, A.A.S., 1996
East Tennessee State University, Johnson City, Tennessee;
Computer Science, B.S., 1999
Liberal Studies, M.A., 2002

Professional Experience: Adjunct Instructor, East Tennessee State University, Department of Computer Science 2001.
Instructor, East Tennessee State University, Upward Bound Program, 2001.

Professional Associations: Phi Kappa Phi