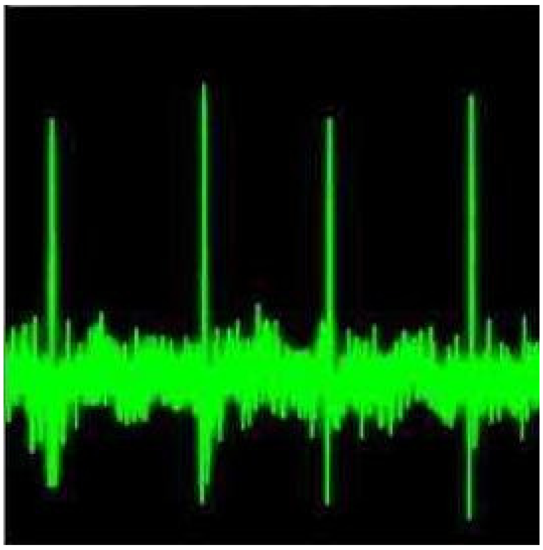
The connectionist modelling of language acquisition



A computational research project

for the degree of MA in Applied Linguistics

Geoff Cockayne

University of Birmingham

July 2008

But what if God himself can be simulated, that is to say, reduced to the signs which attest his existence? Then the whole system becomes weightless; it is no longer anything but a gigantic simulacrum: not unreal, but a simulacrum, never again exchanging for what is real, but exchanging in itself, in an uninterrupted circuit without reference or circumference.

Jean Baudrillard (1995) *Simulacra and Simulation*

Front cover illustration: A noisy spiking neuron.

Reike (1999) *Exploring the neural code*

Table of Contents

Abstract

For over 150 years computers have been considered capable of representing mental processes through the manipulation of symbols, but conventional artificial intelligence models do not account for brain physiology. Connectionist models are networks of artificial neurons and synapses which seek to simulate low-level processing in the brain, characterised by patterns of electro-chemical activity which may be expressed mathematically.

Models which reflect the diversity of brain physiology have been constructed and used to simulate limited aspects of language acquisition, including past tense morphology, and semantic and syntactic categorization. This has led to controversial claims that connectionist models can account for language acquisition without recourse to traditional notions of rules, innate or otherwise.

In this project a model is constructed which simulates individual word morphology. It is particularly successful at capturing infix morphology of the form common in Semitic languages such as Arabic. However, it is not able to simulate word morphology when presented with multiple patterns in the manner described in connectionist literature and known to occur in the brain.

1 Introduction

How do children learn their first language? There is agreement among linguists that they do so in stages (Fromkin et al 2007: 322) (Aitchison 2008: 80), but there is disagreement about the mental mechanisms involved. Some scholars propose that language is like sight: anatomically localised and physiologically modular (Smith 1999: 17). It is further claimed that one such language module contains a specification, called universal grammar (UG), from which all human languages are *acquired* rather than learned (Smith 1999: 45). However, although brain imaging studies provide some support for language modularity, there seems to be no neurological evidence for a universal grammar module (Ward 2006: 211).

Some scholars, who deny the existence of UG, propose that language learning is a consequence of two general cognitive skills: the ability to understand that others have intentions which may be manipulated; and the ability of the child to find patterns in the language he hears (Tomasello 2003: 3-4). Language learning is thus an interactive process whereby input from the child's environment is processed by general cognitive skills.

To determine which of these hypotheses is true, or nearer to the truth, it is useful to investigate the functioning of the brain. If there is a UG module it may be possible to isolate and experiment upon it. Such invasive procedures on people are, however, severely restricted for legal and ethical reasons. Computer models, based on what is known about brain function are therefore a useful alternative. Fortunately , the lower ethical boundary on invasive procedures on non-human species, and advances in non-invasive imaging techniques have led to a substantial increase in our understanding of brain function (Ward 2006: 34, 57). Moreover, computer models, which may be ethically experimented upon, may complement the results of imaging technology (Ward 2006: 75) .

This project seeks to to investigate computational models of language acquisition and in particular connectionist models which are based on what is known about the low level functioning of the brain. The research element of the project is the construction of a connectionist model and its testing using linguistic and non-linguistic data. A central theme of the project is that, notwithstanding the need for simplification, models of language acquisition should be based on physiologically adequate representations of brain behaviour.

The project begins with an overview of the development of computer models of mind and brain: the top-down symbolic or artificial intelligence approach and the bottom-up connectionist or artificial neural network (ANN) approach. In section 3 the workings of connectionist models are described, including the principal expressions used and the types of models constructed. The major research in which connectionist models were applied to linguistic phenomena, including the ground breaking work of Rumelhart and McClelland, are summarised in section 4.

Section 5 describes the construction of the model which was used for the research component of this project and the benefits of the Object Oriented approach for models of the brain. This is followed by the description and analysis of the testing of the model in section 6. The conclusions in section 7 suggest development of the model, and other research, and some possible further developments of connectionism within the framework of cognitive linguistics.

2 Computational models of cognition

Attempts to model human cognition on computers fall into two categories: the symbolic approach, and the connectionist approach. The former views cognition as a form of behaviour *per se*, and seeks to build models which replicate that behaviour. The latter sees cognition as the product of brain physiology and tries to build models based on interconnected networks of artificial neurons (Artificial Intelligence 2008).

2.1 The symbolic approach

The symbolic approach represents a core concept in computer science and linguistics. A computer is more than a calculator because it can manipulate symbols by representing them as numbers (Forouzan and Mosharraf 2008: 43). For linguists, language is more than communication because its users are able to manipulate symbols, in the form of phonemes, combining them to represent an infinity of meanings, which are somehow stored as concepts in the mind (Evans and Green 2006: 476).

2.1.1 Early artificial intelligence

Computer science is said to have been born when Ada Lovelace wrote, in 1843, that Babbage's proposed Analytical Engine established:

> a link . . between between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science.
>
> (Lovelace 2008)

The Analytical Engine, is perhaps the first example of artificial intelligence (AI) because it could make decisions and it was able to 'learn'. Its decision making ability was that of conditional branching: transferring execution to a different part of the program depending on the value of a particular variable. The Engine learned in the sense that it could modify the program which controlled it (Analytical Engine 2008). Had it been built it could have carried out most morphological operations by distinguishing between regular and irregular forms using the 'words and rules' hypothesis (Pinker 1999: 18-19), as shown in figure 1.

```
if rootForm isStored          // irregular is stored as
      get irregularForm       // paired data with root
else                          // rule is explicitly
      applyRule               // coded
```

Figure 1: Word and rules, psuedocode

Such an algorithm cannot be regarded as intelligent, since all new words and rules must be explicitly coded. However, the ability of the Engine to modify its program allows the coding of 'meta-rules' which would write new rules based on patterns of input and error correction. Figure 2, a hypothetical exchange between the program and a user, shows how this might work:

```
program     > the past tense of cry is cryed.
program     > Is this correct?
user        > No
program     > Please enter the correct past form.
user        > cried
program     > Is the rule delete 'y' and append 'ied'?
user        > yes
program     > the past tense of try is tried
program     > is this correct
user        > yes
program     > New rule coded.
```

Figure 2: Inducing past tense rules

It is thus possible to hypothesise a self-modifying computer program capable of inducing all the rules of all languages by interaction with native speakers, and thus, given sufficient time and memory, perfectly emulating native speaker language production. The only limitation to such a program is the proposition that language 'makes infinite use of finite media' (Pinker 1994: 84). That is, if the program were asked to produce every possible sentence in a given language, it would never stop.

2.1.2 The universal Turing machine and the Turing test

The foregoing hypothesis is a linguistics rendering of two computer science hypotheses: the *Universal Turing Machine* (UTM) and the *Turing Test.* The first proposes essentially that all solvable mathematical problems may be encoded and executed on a UTM, and that unsolvable problems will cause the program to run indefinitely (Turing Machine 2008a, 2008b) (Penrose 1989: 67-75). The UTM, given concrete form as the von Neumann machine, is the basis of all modern computers (Forouzan and Mosharraf 2008: 2-4).

The Turing Test proposes that it is possible to construct a computer program which can communicate with a human user such that it is indistinguishable from another person. A program which passes the Turing Test is claimed actually to *think* (Penrose 1989: 6-8). Such claims are of obvious interest to linguists, however no program has yet come anywhere near passing the test (Turing Test 2008). Nevertheless, proponents of 'strong AI' hold that to the extent that a program, or any other device, can simulate human cognitive behaviour, it is by definition intelligent (Artificial intelligence 2008) (Boden 1988: 7-8). This leads to the proposition that any information processing device, such as a thermometer, is intelligent.

However, the thermometer and the computer program lack a property which distinguishes them from the mind: intentionality; the proposition that actions are not merely consequent upon physical states but are done for purposes which are linked to values and ideas (Thompson 2003: 78-79). In the case of all computer programs, the intentionality is contained within the programmer; intentionality can only be demonstrated to exist in biological systems (Searle 1980: 14). Thus a computer program may *duplicate* syntax and morphology, but it can only *simulate* meaning since this is dependent on intentionality (Searle 1980: 11), and in particular on understanding the intentions of others (Tomasello 2003: 23). Those programs, such as ELIZA, which come closest to passing the Turing Test, operate on the basis of manipulating syntax in order to simulate intentionality and intentionality reading (Fromkin et al 2007: 389).

2.1.3 Symbols and semantics

Cognitive linguists hold that language is represented in the mind as a collection of symbolic assemblies, each assembly consisting of two parts. For example the phonetic form [kæt] is likely, for native English speakers raised in the West, to be paired with a mental image of the domestic cat. Such concepts are derived from our perceptions of the real world (Evans and Green 2006: 6). Other symbolic assemblies link the concept of the domestic cat to its various characteristics. This type of linking is understood in computer science and represented in what are known as semantic networks, which use the classical notion of a shared set of semantic primitives to categorise objects (Forouzan and Mosharraf 2008: 468).

Unfortunately, this type of categorisation is inadequate, since language categories are characterised not by necessary and sufficient properties but by exemplar prototypes which have fuzzy boundaries (Taylor 1989: 40-41) and are subject to exceptions (Evans and Green 2006: 253). For example, the same object may be considered to be a bowl if it is intended to contain rice, or a cup if it is intended to contain tea. And a cup which has lost its handle is unlikely to be reclassified as a bowl. Such aspects of cognition do not sit well with the conventional algorithmic approach of computer science, although given enough programming time and computer memory it would be possible to encode all symbolic assemblies for a given language. Such an enterprise would, however, not be sufficient to pass the Turing Test, much less provide an adequate model of language, for the reasons outlined in the preceding section, and the following paragraph.

Computer programs store and access information by reference, as shown in figures 1 and 2 above. This approach corresponds with the *referential theory of linguistic meaning*, which is the view that words and phrases represent labels to things and states in the real world (Lycan 2000: 5). This view is common sense since we define words and phrases in dictionaries and suppose that we make meaning by syntactic and morphological manipulation. The referential theory, however fails to adequately explain how we actually use language. For example the following statement:

these walls are rather thin (1)

may be uttered ironically in an open-plan office where there are no walls to be thin, or by a visitor to a medieval castle which has walls which are 5 metres thick. Even non ironic meanings are problematic: utterance (1) may be a polite request to a neighbour to turn his music down, or a husband's attempt to avoid putting up a shelf. Such utterances are understood because language meaning is a function of the context in which it is used (Lycan 200: 189). Language users develop a vast encyclopaedic knowledge to which particular words, phrases and texts are contextualised 'access-points' (Evans and Green 2006: 221). Moreover, these access points are developed through language use and depend on an understanding of others' intentions (Tomasello 2003: 23).

2.2 From AI to connectionism – what can be simulated

The use of the top down, symbolic approach to construct software which attempts to simulate natural language, seems to have failed because conventional algorithms cannot simulate either the intentionality of language users or the context in which utterances are made. Connectionism seeks to construct highly simplified models of the brain starting from its basic components, the neuron and the synapse. Connectionist models are models, albeit simplified, of biological systems. The question thus arises as to whether such models may be able, in addition to simulating particular aspects of linguistic and other behaviour, to simulate intentionality and thereby give access to the higher cognitive skills, including discourse, which are predicated on intentionality.

A computer simulation is a mathematical model of some phenomenon represented in the form of an information processing computer program. A simulation of a hurricane is clearly not a hurricane, and a simulation of the brain is not a brain. Thus the presence of information processing does not imply a relationship between the program and the brain (Searle 1980: 6). However, if the outputs of a computer model of a hurricane were connected to a very large wind generator, it would be reasonable to suggest that the system, taken as a whole, represents something closer to a real hurricane than the computer alone. It could be described as an artificial hurricane.

Connectionist models manipulate electro-numeric patterns in a manner which is substantially closer to the electro-chemical pattern manipulation of the brain than conventional computer programs, which are in turn closer than mechanical automata.

> *if* thought processes consist in the manipulations of *patterns* of substances, then we only need to build a system capable of creating these identical patterns through its physical organization, without requiring the same set of substances. [Italics are in the original]
>
> (Dyer 1990: 8).

Dyer is directly contradicting Searle's assertion above that thought may only arise in biological systems. However, it seems reasonable to suggest that the closer a model comes to resembling that which it models, the more realistically it will be able to simulate the original. Thus if conventional AI is considered to be the 'wrong architecture', connectionist models are perhaps a move towards the right architecture (Churchland and Churchland 1990: 35).

Connectionist models have been designed for example, which, to a limited degree, simulate linguistic categorisation (Marcus 2001: 93). As such models come to resemble actual brain operation more closely, and, since intentionality either arises from the physical operations of the brain or it is externally ordained, it is possible that a system without explicit rules may give rise to simulated intentionality. Nevertheless, it remains a simulation: *actual* intentionality can only arise from something which is biological and embodied.

For cognitive linguistics the wider architecture within which processing occurs, the human body, is fundamental to linguistic analysis (Evans and Green 2006: 44-45). Thus a connectionist model interfaced to an artificial body, that is a robot with appropriate sensory devices, spatial awareness and the ability to move, might be considered closer to a real person than the model alone. It could perhaps be described as an artificial person.

2.3 The connectionist approach

Where the symbolic approach is goal directed and represents top-down design, the connectionist approach is pattern oriented and represents bottom-up design (McLeod et al 1998: 11-12). All connectionist models seek to simulate the functioning of the brain starting from its basic component, the neuron. But since the human brain has at least 100 billion neurons, each of which may connect to 10,000 other neurons, no current model comes close to representing the brain's connective complexity (Ward 2006:17).

2.3.1 The physiological basis of connectionism

Although neurons show considerable physiological variability, they share a number of common features which may be described mathematically and thus represented in a computer model. Neurons typically have multiple inputs, called dendrites, which transmit electro-chemical signals from other neurons, see figure 3. If the signals arriving at the neuron from its dendrites are sufficient to overcome the neuron's resistance, known as the axon hillock, the neuron will 'fire': that is, generate a signal called an action potential. The action potential is transmitted along a single output known as the axon (Ward 2006: 18-19).

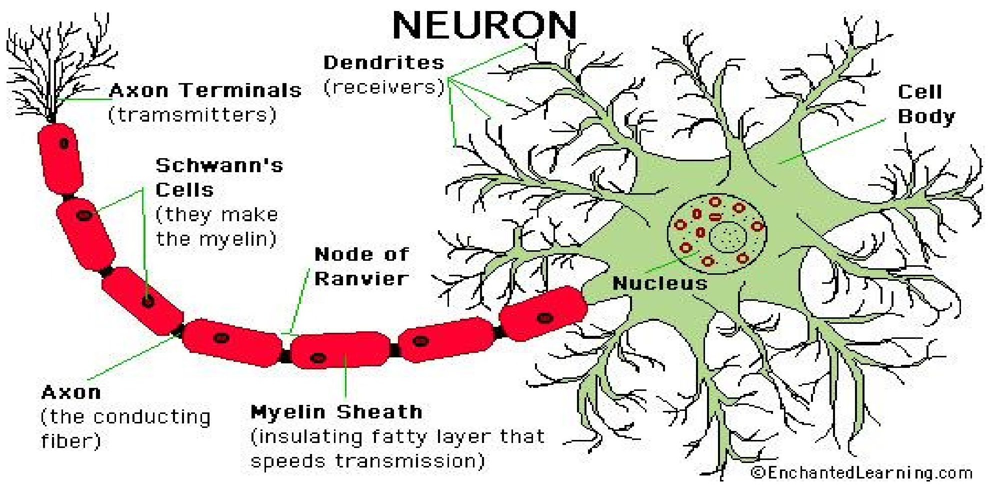

NEURON
Dendrites (receivers)
Axon Terminals (tramsmitters)
Cell Body
Schwann's Cells (they make the myelin)
Node of Ranvier
Nucleus
Axon (the conducting fiber)
Myelin Sheath (insulating fatty layer that speeds transmission)
©EnchantedLearning.com

Figure 3: A typical neuron with multiple dendrites and single axon (Enchanted Learning)

The axon has multiple terminals which connect its output to the dendrites of other neurons via a synapse, as shown in figure 4. The signal migrates across the synaptic cleft, in the form of chemical neurotransmitters, to the connected dendrite and thus on to the next neuron (Gurney 1997: 10). The strength of the neurotransmitters changes with use, which influences the probability that the post-synaptic neuron will fire (McLeod et al 1998: 14). This influence may be positive (excitatory) or negative (inhibitory). If no signal is present at the pre-synaptic connection the synapse is inactive and no signal is transmitted (Ward 2006: 70). Learning is represented by the changing strengths of synapses in response to stimulation by the pre-synaptic signal (McLeod 1998: 14).

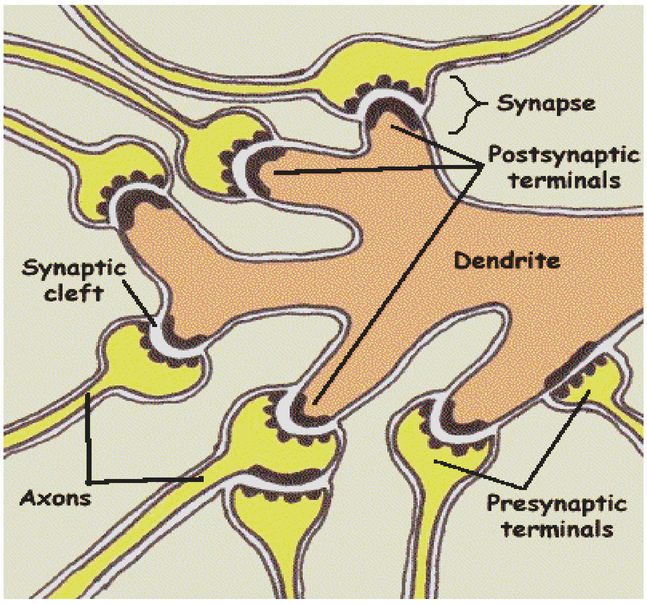


Figure 4: Synaptic connections (www.mult-sclerosis.org)

In general, information processing in the brain is achieved through patterns of responses (or patterns of activity) in groups of neurons (Ward 2006: 35). These patterns are transmitted in parallel through layers of neurons and thus through the different regions of the brain (McLeod 1998: 12). The common sense notion, which underpins the symbolic approach, that the brain stores chunks of knowledge in the manner of a dictionary, or a computer's memory, is not reflected in the low-level physiology of the brain. Knowledge at this level is a distributed set of pattern activations and no item of knowledge can be said to be stored in a particular location (McLeod et al 1998: 31). This form of distributed storage gives rise both to the brain's ability to overcome physical damage and, perhaps, the higher cognitive skills (Ward 2006: 37).

2.3.2 Symbolic and connectionist representations at neuronal level

If processing in the brain were symbolic, we might expect to see a neuronal organisation of the type shown in figure 5 with each neuron representing, for example, a phoneme. Generating the regular past tense of 'work' simply requires strengthening one synapse, shown as a blue diamond, to achieve a representation of the phoneme \t\. Repeated presentations of similar patterns leads to the discovery of the rule using UG, or its emergence using pattern finding skills.

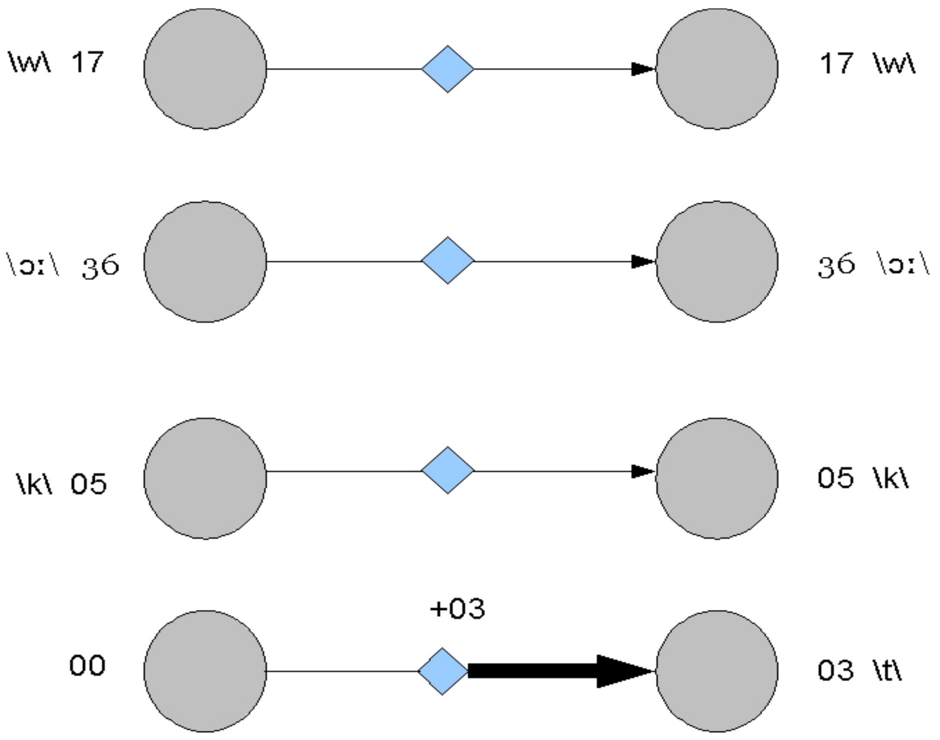


Figure 5: Representation of a 'symbolic' neural network

The values shown are based on the IPA codes used in the Headway course books (Soars & Soars 2003: 159).

However, the brain does not work this way. Neurons have multiple connections to each other as shown in figure 6.

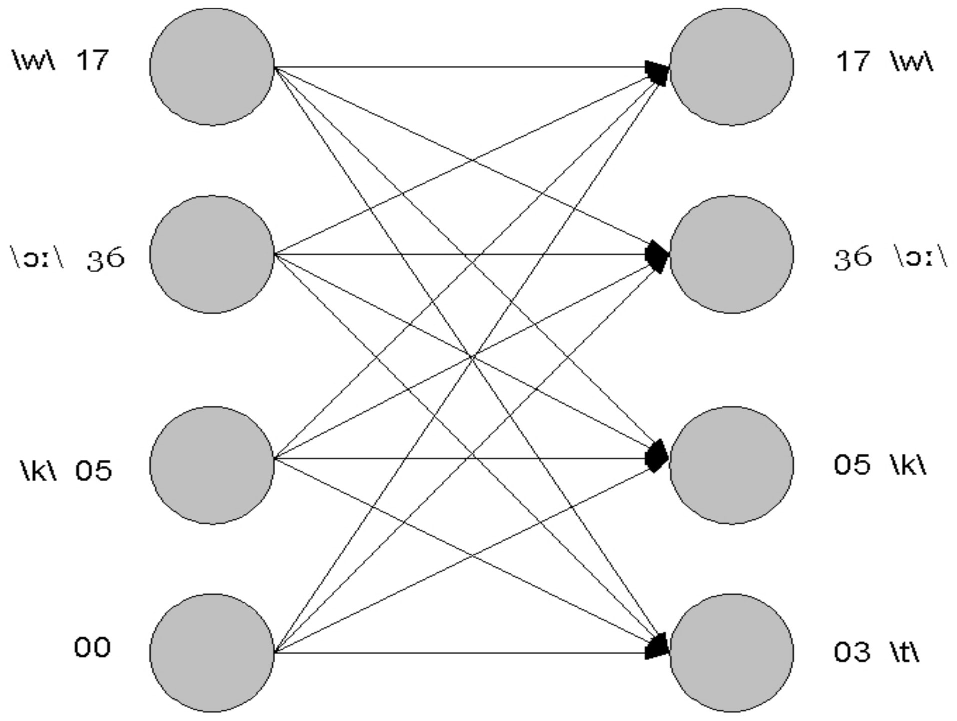


Figure 6: A connectionist neural network

In such a network, where the initial strengths are the same and the values shown are presented, the output values are all the same because each neuron in the output layer is being presented with the same values[1]. In order to achieve the desired output it is necessary to 'train' the network many times, comparing the desired output with the actual output and making small adjustments to the individual synapse strengths (McLeod et al 1998: 19). Thus learning is characterised by progressive and interrelated changes to network connections, and not by any individual change.

1 In this case *f* (58), the sum of the input values, where *f* is the product of the synapse strength and the activation function as described in the following section.

3 How Connectionist models work

An Artificial Neural Network (ANN or connectionist model) consists of two or more layers of neurons with each neuron connected to all those in the preceding layer; as shown in figure 6 above. The artificial neurons are referred to as Threshold Logic Units (TLUs) (Gurney 1997: 13-15), computational units or simply units (McLeod 1998: 11-12). It appears that the TLU conflates the computational behaviour of both the neuron and the synapse (Gurney 1997: 15). In this project the neuronal and synaptic computation have been kept separate since this seems to represent more closely the brain's physiology.

3.1 The Threshold Logic Unit

The TLU simulates the behaviour of the real neuron and synapse by summating the input values of the dendrites connected to it and comparing their value to that of a specified threshold, representing the axon hillock described in section 2.3.1. The inputs consist of the output of each neuron (TLU) in the preceding layer, multiplied by a *weight*, which represents the synapse strength. This is expressed in figure 7.

$$a = \sum_{i=1}^{n} w_i x_i$$

Where:

a	**is the activity or net input of this neuron**
w(i)	**is the weight of the connection from neuron(i) in the preceding layer to this neuron.**
x(i)	**is the output or activity of neuron(i) in the preceding layer.**

Figure 7: Net input to TLU (Gurney 1997: 15)

3.2 Activation functions

'Activation' is used to describe the firing of a neuron: the generation of an action potential. 'Activity' describes a particular value at some place in the network. Net input is the sum of the activities presented to neuron by its connected dendrites. 'Threshold' represents the axon hillock: the value which must be met, or exceeded, in order for the neuron to fire. The output of a neuron (its activity) is determined by an activation function; the most common in connectionist models being the binary threshold function and the sigmoid function.

3.2.1 The binary threshold Function

If the net input is equal to or greater than the threshold, an activation is generated and the neuronal activity is set to 1. Otherwise the activity is set to zero. This function seems to be a simplification of neuron physiology (Ward 2006: 20), it is justified in order to explore the principles of pattern association (McLeod et al 1998: 17-18). Firing of the neuron is expressed by:

$y = 1$ if $a >= \theta$
$y = 0$ if $a < \theta$ (Gurney 1997: 15)

Where,

y	is the output (activity) of the neuron.
a	is the net input to the neuron
θ	is the threshold

3.2.2 The sigmoid function

This function is said to be a more physiologically valid representation of neuron behaviour because changes in the net input at the lowest and highest input values have relatively little effect on the output value (McLeod 1998: 103). It is also the function most commonly used in connectionist models of language acquisition (McLeod 1998: 18 and Chp 9). The sigmoid function is represented by the expression shown in figure 8, producing an output of the form shown in figure 9.

$$y = \frac{1}{1 + e^{-(a-\theta)/\rho}}$$

Where,

y is the output value passed to the axon.
e is a mathematical constant (Euler's number), approximately, 2.71828183.
a is the net input from the dendrites.
θ (theta) is the threshold
ρ (rho) is a variable which may be used to change the shape of the function. Smaller values 'squash' the function, bringing it closer to the the binary threshold function (Gurney 1997: 19).

Figure 8: The Sigmoid function (Gurney 1997: 19)

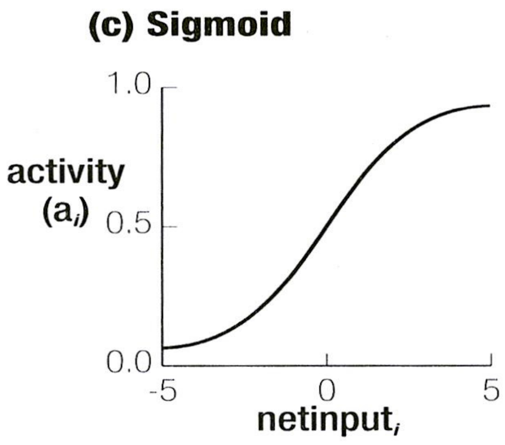


Figure 9: Graphical representation of the Sigmoid Function (McLeod et al 1998: 18)

3.3 Learning rules

The manner of learning in a connectionist model is critical to its success. The strengthening of neurotransmitters at the synapse described in section 2.3.1 is represented computationally by applying a learning rule in order to adjust the weights in the TLU. The two most common rules are described below. The connectionist literature also describes a *bias node* or *bias unit* which adjusts the threshold as if it were a weight (McLeod et al 1998: 20, Gurney 1997: 37). No physiological justification is provided for this operation; neuroscience texts seem to suggest that the threshold (axon hillock) is actually constant, firing at about -50mV (Ward 2006: 20).

3.3.1 The Hebb rule

This is a simple rule used for pattern association. Typically, a model is presented with both the input pattern and the desired output pattern, and the Hebb Rule is used to adjust the synapse strengths (weights) across the network using the expression:

$$\Delta w_{ij} = \varepsilon \, a_i \, a_j$$ (McLeod et al 1998: 54)

where

Δw_{ij}	is the change in strength for this synapse.
ε	is the learning rate.
a_i	is the desired output value (activity) of the post-synaptic neuron.
a_j	is the actual output of the pre-synaptic neuron.

Since the right hand side of the expression is constant for a given pair of patterns, the Hebb Rule is also constant: either a positive value or 0. Its effect is to continually increase the strength of active connections.

3.3.2 The delta rule

This rule seeks to adjust the synapse strengths in order to minimise the errors at the output neurons. It does this by subtracting the output value obtained from the output value desired:

$$\Delta w_{ij} = [a_{i(\text{desired})} - a_{i(\text{obtained})}] \, a_j \, \varepsilon$$ (McLeod et al 1998: 54)

Where,

$a_{i(\text{obtained})}$ is the actual value of the post-synaptic neuron on the previous iteration.

The value of Δw_{ij} may be positive or negative and changes on each iteration until the actual output value is equal to the desired value.

3.4 The development of connectionist modelling

The first connectionist network was based on the McCulloch and Pitts (MCP) neuron devised in 1943. It was a logic-state device capable of simulating the Boolean functions AND, OR and NOT and storing the result (McLeod et al 1998: 314). However the network did not incorporate synaptic functionality and it was thus not possible for the MCP network to simulate learning.

3.4.1 Hebb Learning and the perceptron

Psychologist Donald Hebb first proposed that learning was effected by the strengthening of connections between neurons and and this idea was implemented in an electro-mechanical device built by Edmonds and Minsky in 1951 (Rumelhart & McClelland 1986: 152-153) . The tendency of the Hebb rule, noted in section 3.3.1, to increase synaptic strength without limit led to a number of modifications, including setting an overall limit to synapse strengthening in a network(McLeod et al 1998: 319). These modifications were based on what was discovered about the brain's physiology (McLeod et al 1998: 320).

These modifications were incorporated into the perceptron, the first pattern association model, developed by Rosenblatt in the late 1950s (Rumelhart & McClelland 1986: 156). Pattern association models simulate a simple form of stimulus-response memory in which the presentation of a particular pattern of input generates a single, regular output (Rolls and Treves 1998: 23). A key insight of this development was neural operations are based on probabilistic rather than logical processing (McLeod et al 1998: 320).

Although the perceptron model had some success in classifying patterns according to similarity of shape (Rumelhart & McClelland 1986: 155) Minsky & Papert were able to show that it was unable to discriminate patterns requiring, ironically, XOR logic, where 0 & 1 = 1, but both 0 & 0 and 1 & 1 = 0 (McLeod et al 1986: 323-325).

Rosenblatts's perceptron model contained only two layers, input and output, and it soon became clear that the solution to the XOR problem was the addition of intervening *hidden layers*. However there was no known way of training such models. A solution was found with the backpropagation method in which the learning rule is applied backwards through the network (McLeod et al 1986: 325). Although successful, it was used in Rumelhart and McClelland's work on past tense verbs (McClelland and Rumelhart 1986: 225), backpropagation is considered physiologically implausible (Gluck and Myers 2001: 109).

3.5 Modelling memory

Developments during the 1980s, particularly the work of Hinton and Anderson on auto-association and competitive networks, and McClelland on content addressable memory, led to the ability to model more sophisticated forms of memory, and thus, it is claimed, the prospect of simulating higher cognitive functions (McLeod et al 1998: 326).

3.5.1 Auto-association models

Auto-association models are similar to pattern association networks, with the addition of a connection back to each source neuron, referred to as a recurrent connection, as shown in figure 10. These models have the ability to store multiple stimulus-response type patterns (McLeod et al 1998: 73). They are also able to generate a complete pattern from partial input, by looking for pattern similarities, and produce a correct pattern when the network has been lesioned (Rolls and Treves 1998: 46-47).

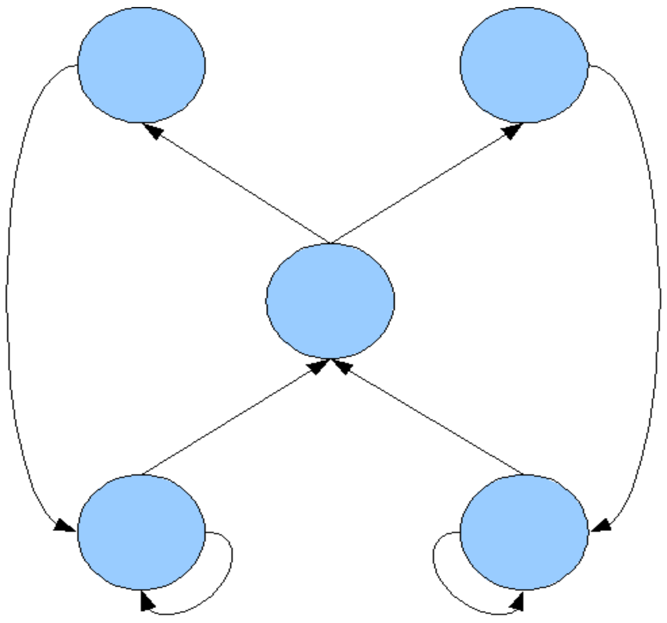


Figure 10: An auto-associative network with recurrent connections (McLeod et al 1998: 141)

3.5.2 Competitive models

In a competitive network, of the type shown in figure 11, outputs are connected to an inhibitory cluster: a set of neurons each of which inhibits those in the same layer when activated. Thus the neuron which receives the greater number of activations will eventually suppress the activation of its neighbours and win the competition. This competitive behaviour leads to the clustering of similar patterns of input (McLeod et al 1998: 127-128) and was used by Elman in his work on syntactic and semantic clustering (see 4.3.1 below).

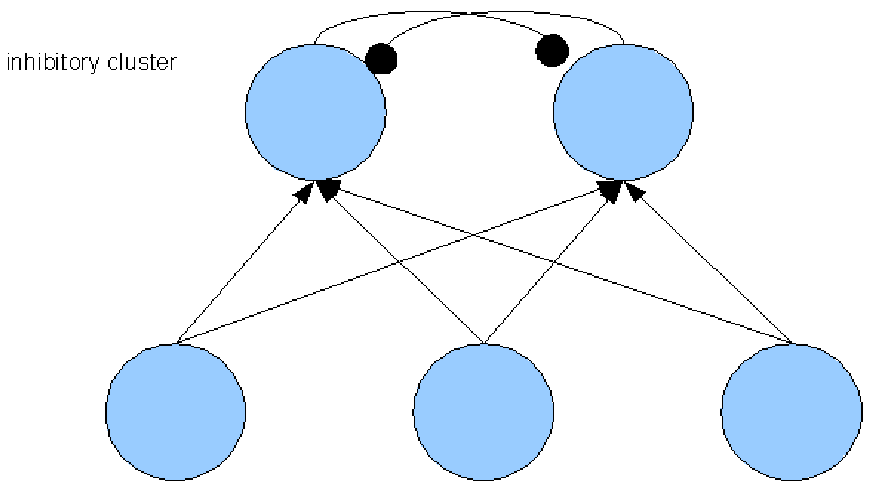


Figure 11: A competitive network with inhibitory cluster (McLeod et al 1998: 128)

3.5.3 Content addressable memory

Computers and dictionaries store information by reference: a dictionary cannot handle questions such as, “what's that word beginning with 't' that means boring?” It would be necessary to search every entry under 't', a somewhat tedious process. A computer database faces the same problem but is able to iterate through its contents quickly. Yet the brain handles such questions with ease because information is stored by content rather than reference (McLeod et al 1998: 34).

McClelland built a connectionist model which was similarly content addressable, by combining the auto-associative model and the competitive model described above (McLeod et al 1998: 35) (Rolls and Treves 1998: 42). Figure 12 shows how this works.

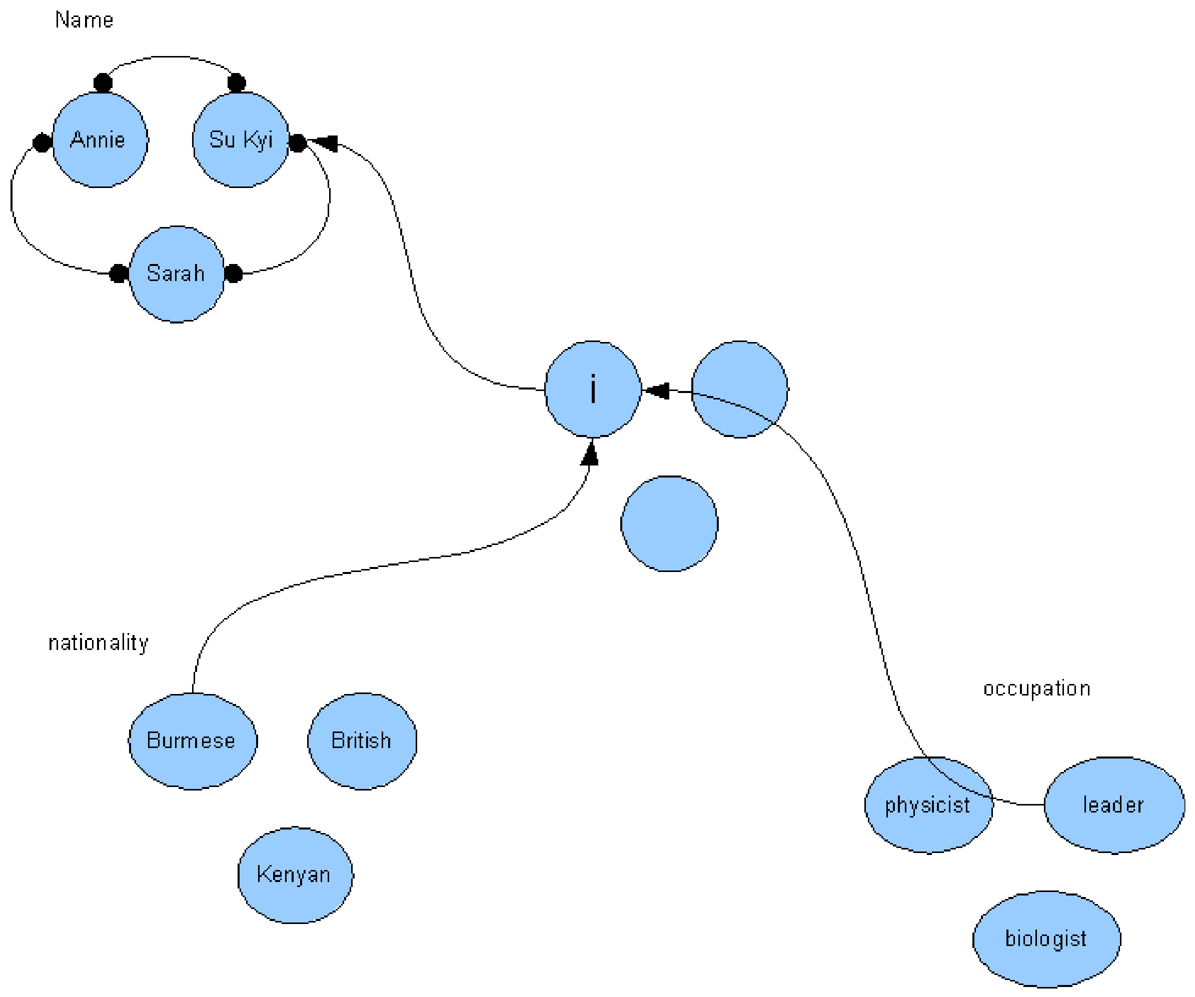


Figure 12: content addressable memory (from McLeod et al 1998: 40)

The clusters labelled name, occupation, and nationality represent knowledge about a group of people. Each unit in the cluster is mutually inhibitory with respect to each of the others (shown only for 'name' in the figure). The central cluster connects particular instances of knowledge representing what one knows about a specific individual; these connections are excitatory (McLeod 1998: 41).

If one asks the question 'Who is that Burmese leader . . . ?' the nationality and occupation clusters are activated for 'Burmese' and 'leader', which suppress the other units in the respective clusters. This combination excites unit *i* in the central cluster, which in turn excites the name 'Su Kyi' in the name cluster (McLeod et al 1998: 41).

3.6 Recent developments

There are two important criticisms of connectionism: that it does not adequately represent brain behaviour and that it does not account for higher cognitive functions. Recent attention has been given to these areas, as described in the next two sections.

3.6.1 Development of physiologically adequate networks

The output from neurons is characterised by spiking (Ward 2006: 20) and expressions which model this behaviour, for example the Hodgkin-Huxley equations (Trappenberg 2002: 24) have been incorporated into ANNs. Signals in dendrites deteriorate with distance and the signals of some axons are boosted by a being sheathed with a fatty material called myelin (Ward 2006: 19, 21). Expressions known as cable equations have been developed which capture this behaviour (Trappenberg 2002: 29).

Models which account for brain modularity have also been developed including: the pre-frontal cortex and hippocampus associated with episodic memory (Rolls and Treves 1998: 97) (Trappenberg 2002: 270); the amygdala and orbiofrontal cortex associated with emotion and motivation (Rolls and Treves 1998: 137); and the cerebellum and basal ganglia associated with motor functions (Rolls and Treves 1998: 190-191).

The proposition that computers themselves may be the 'wrong architecture' (Searle: 1980 passim) has led to work on constructing a machine which more closely resembles the architecture of the brain. (Furber et al 2006). The machine, known as SpiNNaker, is based on a large array of neuron-like components, which will, it is claimed, simulate networks of up to a billion neurons.

3.6.2 Accounting for higher cognitive functions

Connectionist models have also been criticised for failing to account for higher cognitive functions, in particular that they 'do not provide an account that includes communicative function or meaning' (Tomasello 2003: 191). Connectionists might argue that, in contrast with AI research, their goal is to explore cognition at low level particularly with respect to the innateness proposition mentioned in the introduction. Nevertheless, models have been built which are moving toward an explanation of the higher mental processes.

Models have been constructed which learn to specialise on different cognitive aspects of a particular phenomenon. They are modular in design, consisting of a network of competitive networks (Jacobs et al 1991: 227-228). A model has been trained to recognise both *what* an object is and *where* it is, with the modules competing for these tasks. The modules are not pre-assigned to either task. Moreover, it is found that different module architectures are successful at different tasks (Jacobs et al 1991: 239).

Models which are capable of a limited form of prediction have also been built, by incorporating into a recurrent network a *context layer* which accounts for previous patterns of network activity (Elman et al 1996: 81). This temporal element allows the network to predict, for example the correct third person form of the verb 'kick' in the following:

boy who chases dogs kicks ball (2) (McLeod et al 1998: 197)

Child development, including language development, is observed to occur in stages rather than continuously. (Gross 1996: 631). This might seem to present a particular problem for connectionism since its learning strategy is based on the gradual changes of synaptic strengths. However, both models of language, and other forms of development, are able to demonstrate staged behaviour (McClelland and Rumelhart 1986: 252) (McLeod et al 1998: 217, 231). This behaviour is inherent to the nature of processing within connectionist networks. Learning in the form of the delta rule is error minimising and represents a multi-dimensional gradient as shown in figure 13. The consequence of this is that small changes in weights can produce large changes, up or down, in overall error.

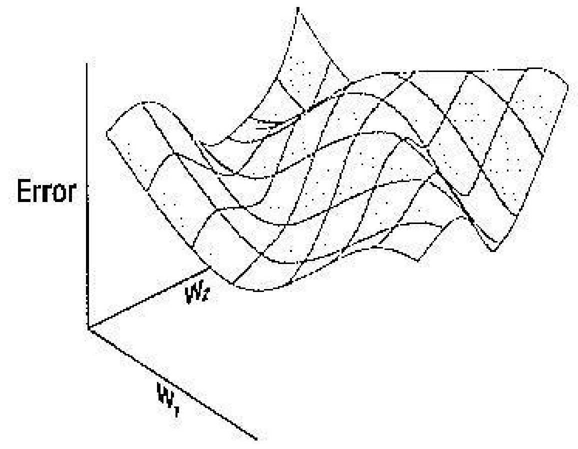


Figure 13: error gradient during learning (McLeod et al 1998: 235)

4 Connectionist research in linguistics

Connectionist models simulate learning and so their application to linguistics implies claims about the nature of language acquisition in children (McLeod et al 1998: 178). Such models have generated considerable controversy since Rumelhart and McClelland first trained a model which simulated the learning of past tense verbs (McClelland and Rumelhart 1986: chapter 18).

4.1 Learning the past tense of English verbs

Learning of the past tense is characterised by 'U' shaped behaviour in which children having learned a correct irregular such as 'went' are observed to produce utterances such as:

the alligator goed kerplunk (3) (Pinker 1999: 17)

Some scholars propose that this is evidence of UG: the effect of identifying the regular '-ed' rule being so powerful that the child over-uses it until at a later stage she learns to 'block' the rule for irregulars (Pinker 1999: 19) . Other scholars observe that such errors are relatively rare; occur with irregular as well as regular forms, and are inversely related to the frequency with which the child hears them (Tomasello 2003: 233). Such data supports a single, rather than dual, learning process in which words are clustered according to phonological similarity and token frequency; errors occur where words are placed in the wrong cluster (Tomasello 2003: 238).

4.1.1 Rumelhart and McClelland 1986

Rumelhart and McClelland entered the fray, building a model which consisted of a two layer competitive pattern associator (McClelland and Rumelhart 1986: 222). This was connected to phonological encoder/decoder which employed *wickelfeatures*, whereby the encoding of a phoneme accounted for its adjacent phonemes as well as its own phonemic properties (McClelland and Rumelhart 1986: 234). The model used the sigmoid activation function, and a variation of the delta rule (McClelland and Rumelhart 1986: 224, 225).

The model was presented with a total of 506 verbs, divided into three sets of 10, 410 and 86, based on frequency of use by young children. The first set was presented and trained and then the second group added to the first and trained. The final group of verbs was presented without training (McClelland and Rumelhart 1986: 240-241).

During training the model demonstrated 'U' shaped behaviour and achieved a success rate of over 95% after 200 trials (McClelland and Rumelhart 1986: 242). When the final group of 86 verbs was presented, without training, the success rate was 92% for regulars and 84% for irregulars (McClelland and Rumelhart 1986: 261). Rumelhart and McClelland concluded that they had :

> shown that a reasonable account of the acquisition of past tense can be provided without recourse to the notion of a "rule" as anything more than a *description* of the language. [quotes and italics are original]
>
> (Rumelhart and McClelland 1986: 267)

The model was subsequently criticised on an number of grounds. (1) The onset of 'U' shaped behaviour was attributed to the timing of the presentation of the second set of verbs which contained a far greater proportion of regulars than the first set (Pinker and Prince 1988: 138). (2) The use of competitive networks in the decoder tended to obscure the actual output "chosen" by the model (Pinker and Prince 1988: 94, 123). (3) The model failed to account for the syntactic behaviour of the past tense in utterances such as 'I helped her leave' versus 'I know she left' (Pinker and Prince 1988: 85). (4) The model did no more than could be done more easily by a symbolic, rule following, model (Pinker and Prince 1988: 81).

The first three criticisms were answered in subsequent research described below. The fourth criticism is certainly true and is demonstrated in figure 1. However, it rather misses the point that in a Universal Turing Machine correct results are always generated provided the rules are correctly encoded by the programmer; such models reveal nothing about how language arises in the brain. Connectionist models are claimed to demonstrate that a rule-like behaviour is an emergent property of a simple brain-like pattern finding mechanism (Elman et al 1998: 110-115).

4.1.2 Plunkett and Marchman 1993

In a revision of the Rumelhart and McClelland model, Plunkett and Marchman sought to answer the specific criticisms of Pinker and Price. Their model included hidden units, which sit between the input and output units (Plunkett and Marchman 1993: 29), and a phonological encoding system which does not appear to use wickelfeatures (Plunkett and Marchman 1993: 31). Most importantly an initial training set of ten verbs stems was increased one verb at a time rather than in bulk (see criticism (1) above), (Plunkett and Marchman 1993: 33).

The model demonstrated 'U' shaped behaviour for both regular and irregular verbs, though at different stages (Plunkett and Marchman 1993: 39). A number of irregular verbs, having been correctly mapped to their past form correctly during early training, were latter regularised; for example 'comed', 'seed', 'blowed' (Plunkett and Marchman 1993: 47). The model began to generalise once the proportion of regulars in the set exceeded 50%; prior to this generalisation was 'blocked' (Plunket and Marchman 1993: 55). Furthermore, the onset of regularisation was 'relatively sudden' (Plunkett and Marchman 1993: 52).

Plunkett and Marchman concluded that their model suggested:

a *single mechanism* learning system may offer an alternative account of the transition from rote-learning processes to system building in children's acquisition of English verb morphology. [italics in original]

Plunkett and Marchman 1993: 58

4.2 Semantic categorization and prototype formation

Categorization is a core cognitive skill which contributes to the ability of the human mind to form concepts from phenomena we observe in the world. The concept which defines a category is constructed, not by a set of defined properties but as an abstract, flexible and multi-feature prototype. The prototype is then used as a measure of category membership (Evans and Green 2006: 249). The sophisticated quality of prototypes enables us, for example, to easily recognise a cross-bred labrador-poodle, or a bull-terrier with no tail, as belonging to the category 'dog'. This sophistication causes problems for children during their early lexical development (McLeod et al 1998: 189). They are observed to both over-extend and under-extend their use of words: calling all animals 'dog' or calling only the family pet 'dog' (Crystal 2003: 246-247).

4.2.1 Plunkett (1992)

Plunkett constructed a model which sought to simulate early lexical development. It consisted of a multi-layer auto-associative network with inputs and outputs separately assigned to images and labels, as shown in figure 14.

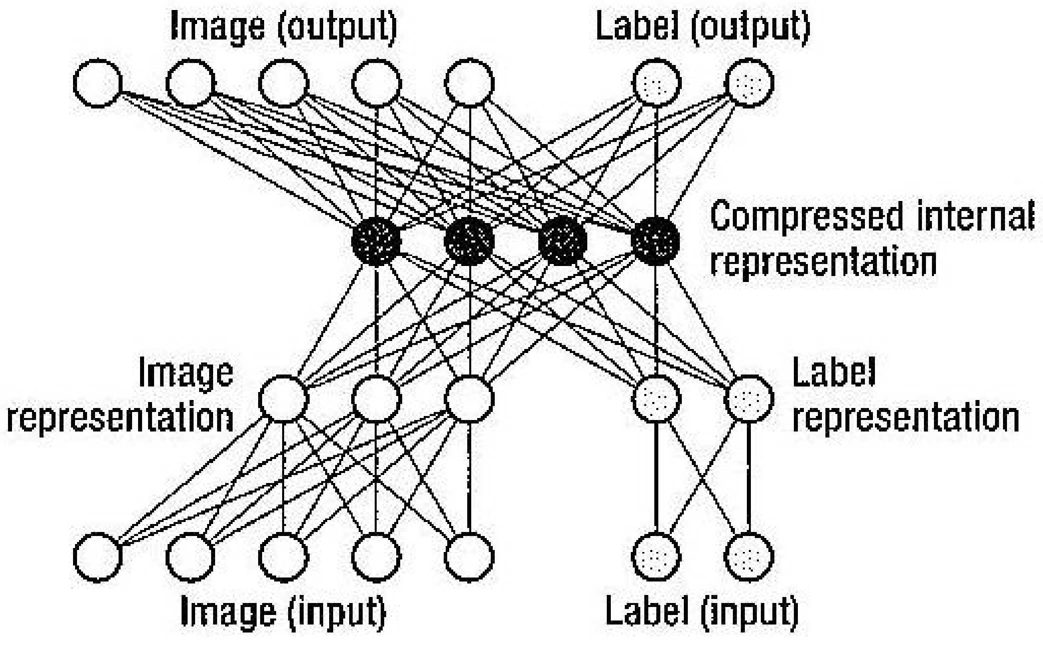


Figure 14: auto-associative network with dual inputs and outputs (McLeod et al 1998: 190)

The model is trained in three stages: (1) an image is presented until trained to the image output; (2) its label is similarly presented; (3) both are presented until trained to both outputs (McLeod et al: 1998: 190-191) . The images consist of groups of dot patterns based on a prototype, using a method previously tested on human subjects. Out of nine dots in each pattern, only two are in common with the prototype; the others are randomised. The labels are represented as a single active bit within a 32 bit pattern as shown below (Plunkett et al 1998: 189). It is not clear from the original article why this method was chosen.

00000000000000000000000000000100

During training the prototype is not presented to the model nor is the model trained to produce an image output given a label input, or a label output given an image input (McLeod et al 1998: 191). During training the model is tested for comprehension: the ability to correctly generate an image when a label is presented; and production: the ability to correctly generate a label when an image is presented. The model demonstrated over-extension and under-extension for both comprehension and production (McLeod et al 1998: 191-192).

For cognitive linguists this model may appear to be of greater importance than the past tense models described above, since it appears to include a claim to simulate the higher cognitive skill of categorisation, including prototyping. It may seem surprising therefore that it does not appear to have sparked the type of controversy which followed Rumelhart and McClelland's work. Moreover no reference is made to this model in standard texts such as Evans & Green (2006), Croft and Cruse (2004) or Tomasello (2003). The model appears to fall short of the cognitive linguists' view of prototypes since it does not appear to be capable of simulating their taxonomic characteristics, whereby a category such as 'dog' may be regarded as more cognitively 'rich' than those such as 'mammal' or 'spaniel' (Croft and Cruse 2004: 83).

4.3 Learning word boundaries and syntactic classification

Linguistic input is temporal but comprehension seems to necessitate its 'chunking' into meaningful symbols. When one first hears an unfamiliar language it can seem to be a meaningless stream of sound. The same problem must face children learning their first language: just how they learn to mark the boundaries of words is a puzzle for linguists. A similar temporal problem must also occur with syntax acquisition: how is it that a native speaker of English recognises the following sentence as 'grammatical' though meaningless?

Colorless green ideas sleep furiously. (Chomsky 1957: 15)

Some scholars suggest that because syntactical relationships are characterised by 'long-distance dependencies', it is impossible for word classes to be acquired by merely analysing the temporal input stream. (Pinker 1994: 97). They thus propose an innate, tree-like, phrase structure grammar in which sentences are analysed and hierarchically deconstructed into their constituent parts (Pinker 1994: 98).

Connectionists reject this approach and seek to show that syntax can be acquired directly from the child's linear, temporal input.

4.3.1 Elman 1990, 1993

Elman built an auto-associative model with additional *context units* which served as a form of dynamic short-term memory, as shown in figure 15 (Elman 1990: 182).

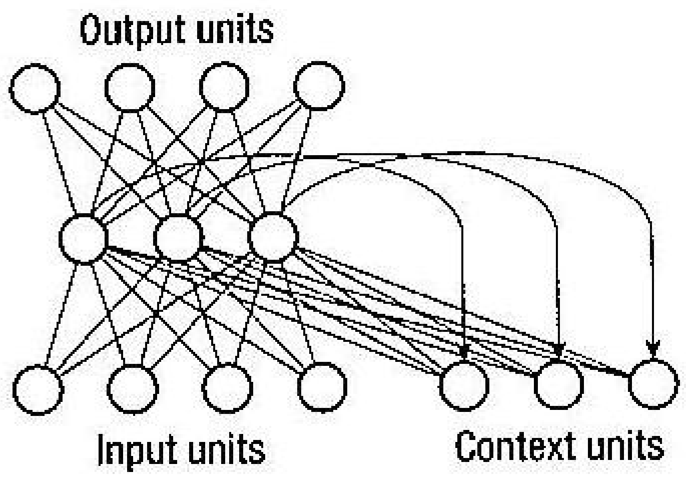


Figure 15: auto-associative network with context units (McLeod et al 1998: 197)

The model was first presented with a 1000 unit string consisting of randomised consonant/vowels combinations constructed from the units 'ba, dii, guu'. As the string was presented the network was asked to predict the next character in the sequence. Unsurprisingly, its ability to predict vowels following consonants was significantly higher than for consonants themselves . However, it was this very disparity that allowed the boundary between the 'words' to be established, as shown in figure 16.

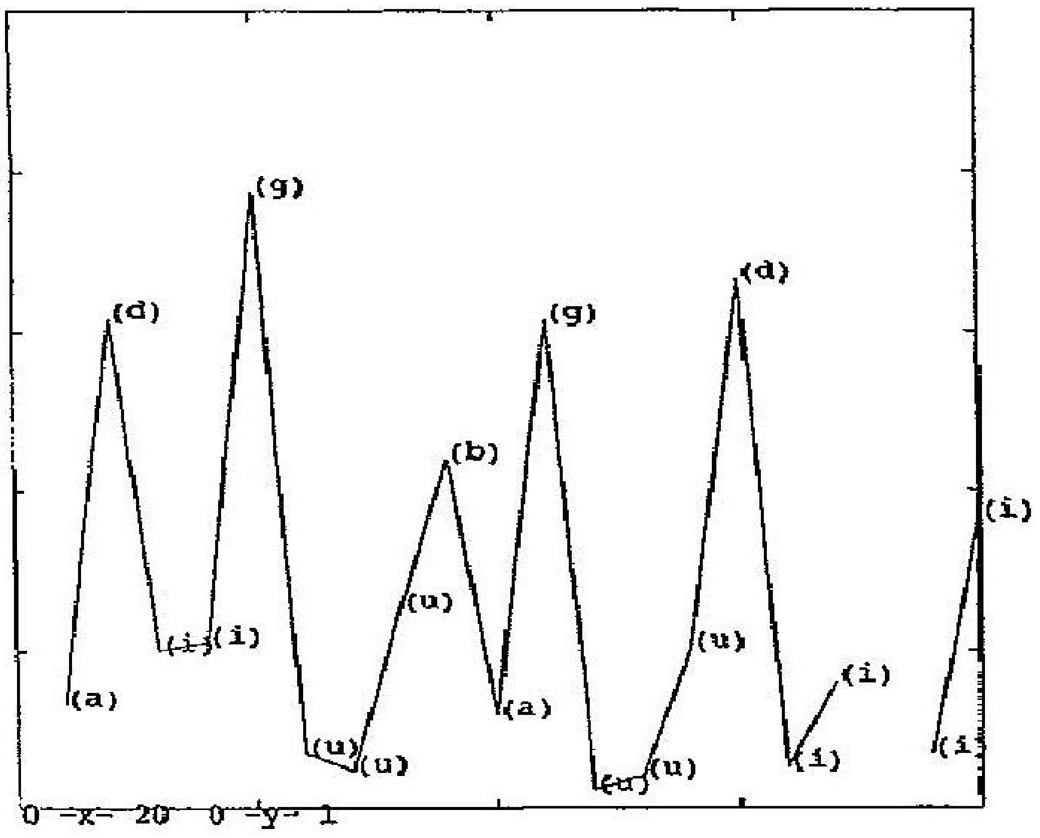


Figure 16: root mean squared error rate in letter prediction (Elman 1990: 189)

The model was then re-run using sample English sentences. It produced a similar pattern, except that it was able to partition words beginning with both consonants and vowels, as shown below:

many/years/ago/aboy/and/girl/lived/by/the/sea/they/played/happily

(Elman 1990: 194)

A larger version of the same model was then presented with an input stream consisting of two and three word sequences in the form:

man eat food book break woman destroy plate (Elman 1990: 194)

Although the input words were bounded, the sentences were not, and no information was given concerning word classes. The model was required to cluster the input words hierarchically (Elman 1990: 196). The results are shown in figure 17. The added labels indicate that the model is able to distinguish word classes to some detail: 'like' and 'chase' require a direct object. It is simultaneously capable of semantic classification: 'monster', 'lion', and 'dragon are [+animate] [-human] [+large] (Elman 1990: 200)

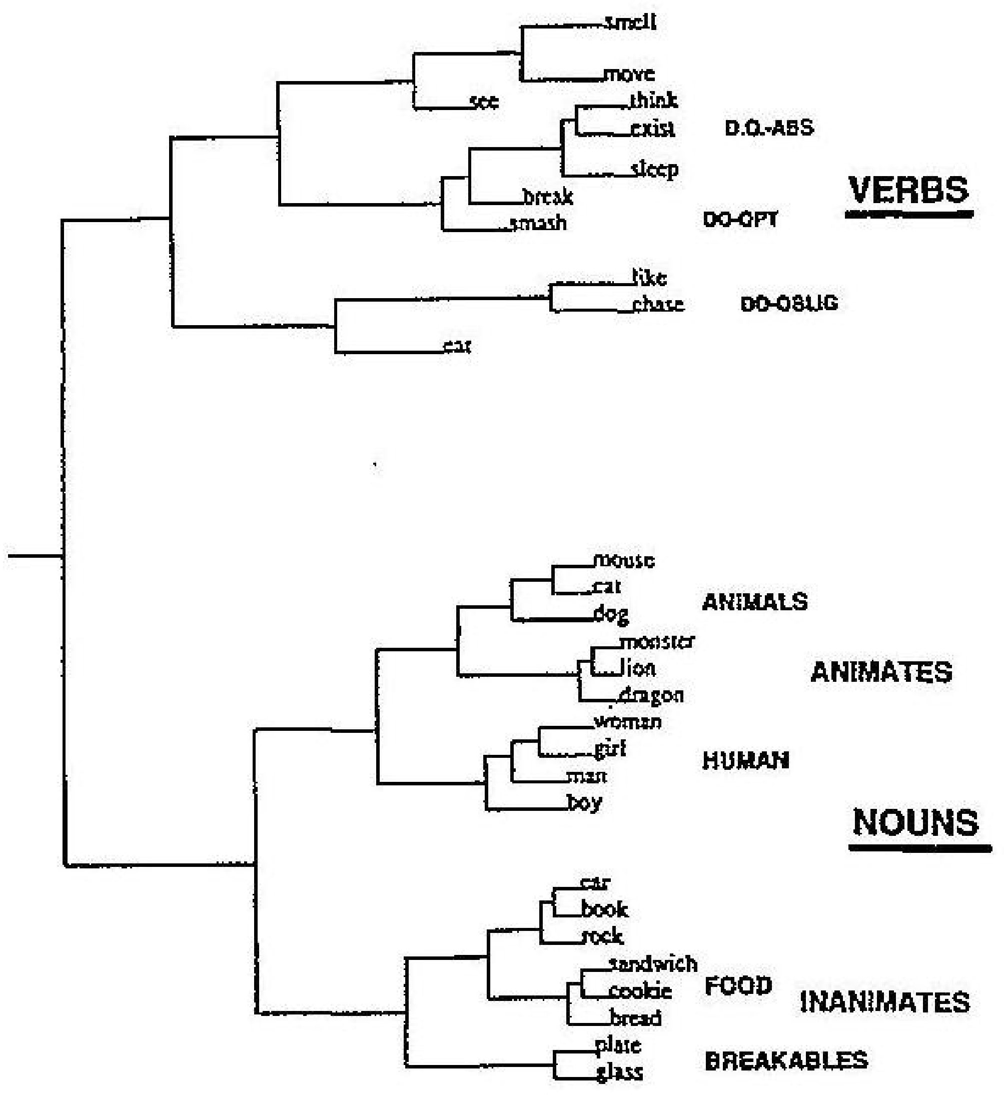


Figure 17: Syntactic clustering of hidden unit activation vectors (Elman 1990: 200)

4.4 The auxiliary inversion question

One of the strongest arguments for an innate universal grammar is the 'poverty of stimulus' proposition in which children are claimed to 'know more than they learn' (Smith 1991: 40). One example of this is the manner in which children are able to generate the correct question form in an utterance containing two auxiliaries such as:

the man who is exiled from Tibet is the Dalai Lama (1)

Given previous encounters with simpler question forms, children would be expected sometimes to invert the first auxiliary and produce:

is the man who exiled from Tibet is the Dalai Lama? *(2)

However, research shows that they rarely make such mistakes and thus it is claimed that children must have innate grammatical knowledge which recognises 'the man who is exiled from Tibet' as a noun phrase thus requiring the inversion of the second rather than the first auxiliary when forming the question (Pinker 1994: 41-42). A Connectionist account by contrast claims that a neural network could generate the correct form without prior knowledge of phrase structures.

4.4.1 Lewis and Elman 2001

A model similar to that described in 4.3.1 above was presented with increasingly complex sentences from CHILDES, a corpus of child directed speech. Each word in an utterance was supplied sequentially and the model was required to predict the class of the next word (Lewis and Elman 2001: 554). A sample prediction is shown in figure 18; the vertical bars show the strength of prediction.

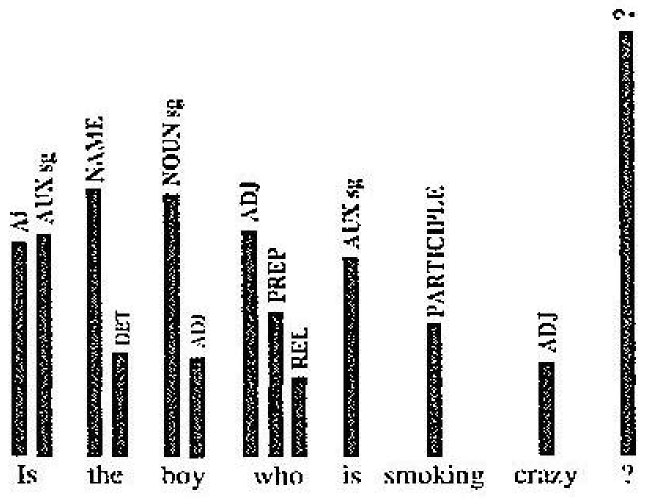


Figure 18: strength of predictions for next word (Lewis and Elman 2001: 554)

The model correctly predicts a singular auxiliary following the relative pronoun 'who' and an adjective – rather than an auxiliary - following the participle 'smoking' . At no time does the model predict a form corresponding to the ungrammatical utterance (2) above (Lewis and Elman 2001: 554).

5 Building a connectionist model

To build a connectionist model requires programming skills. Since linguists do not generally have such skills it is usually necessary to make use of software, such as 'tlearn', which is supplied on disk by McLeod et al (1998). Such models are 'black-boxes': their internal workings are not available to investigation by the researcher. For researchers with some programming skills it is possible to copy code supplied by programmers, for example Rogers (1997). This allows the researcher to investigate the code; nevertheless, one is reliant on the design and implementation methodology of the programmer. For this research it was decided to design and construct a new model, because it was thereby possible to:

- Construct a model based on classes which aim to be physiologically adequate; in particular separating the functionality of neurons, synapses and the network which connects them.
- Test the model progressively as it is developed; in particular the mathematical expressions on which it relies.

5.1 OOP and classes

Object Oriented Programming (OOP) is a computer science paradigm in which objects and concepts from the real world are defined as such in software. A class is a blueprint containing properties and behaviour. An object is created as an instance of a class. A key concept of OOP is encapsulation: objects may communicate only by sending messages to each other. An object may not directly interfere with the properties or behaviour of an other object (Troelsen 2007: 147).

Two other important concepts in OOP are inheritance and polymorphism. Inheritance allows the creation of child classes which extend the properties and behaviour of the parent class, whereas polymorphism permits their modification or restriction (Troelsen 2007: 145-146). This approach seems suitable for modelling the brain since one may begin with a simple set of classes based on the neuron and the synapse, and subsequently extend the classes to capture brain diversity and modularity.

In any OOP project, great care must be taken to establish the appropriate classes. With an engineered system such as car this relatively straightforward, since it consists of a set of discrete components. A biological system is more problematic because it may not be clear where one object ends and other begins; moreover, the properties and behaviour of the system's components may not be perfectly understood.

For this project three classes were constructed:

1. The neuron class, encapsulating the summation of the dendrites values and the generation of an action potential at the axon hillock.
2. The synapse class, calculating and storing the strength of the connection between its source and target neurons.
3. The network, organising the creation and management of a network of neurons and synapses, in accordance with values passed from the user.

The decision to define these classes was taken in answer to the question 'where does the main functionality of the system lie?' Thus there is no class for dendrites or axons because, in this simplified model, they merely transmit values: they do not modify them. In the brain signals may in fact be affected by the anatomical features of particular dendrites and axons (Ward 2006: 19). Also, although the axon hillock is often described in the literature as anatomically separate from the neuron (Alberts et al 1989: 1087), it seemed unnecessary to it define as separate class since its behaviour is computationally integral with the neuron.

Each class is described below; the full code is listed and documented in appendixes A and B.

5.2 The neuron class

The function of the neuron class is to receive inputs from its connected dendrites, summate them, apply the appropriate activation function and pass the output to its axon, as shown in figure 19.

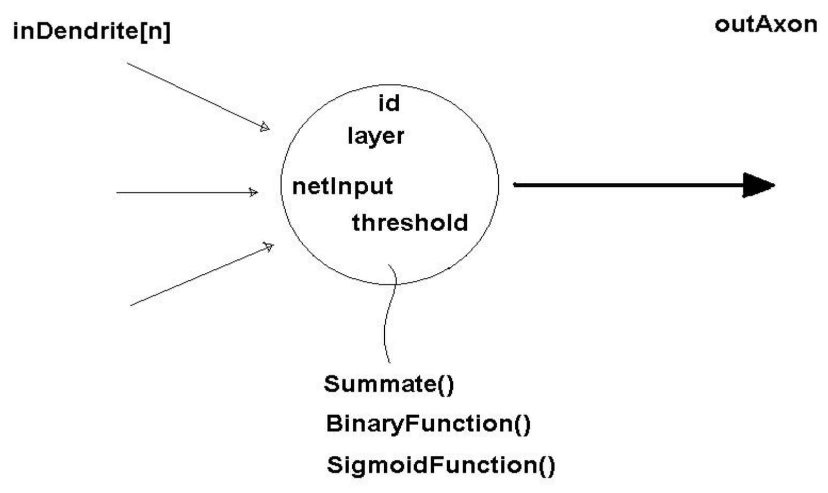


Figure 19: Schematic representation of the Neuron class

Neuron properties

- id. Each neuron has a unique id number, assigned when the neuron object is created.
- layer. Each neuron is in a layer, assigned when the object is created (McLeod et al 1998: 12-13).
- inDendrite[n]: The neuron may have any number of input dendrites. They are represented as an array (Gurney 1997: 8). The values may be positive (excitatory) or negative (inhibitory) (McLeod 1998: 11). Each dendrite is connected to the output of a synapse object.
- threshold. A constant value passed from the Network class (Gurney 1997: 11, 14).
- outAxon: Each neuron has one axon, the value of which is determined by the summation of the input dendrites and the application of the binary or sigmoid function described below (McLeod et al 1998: 11) (Gurney 1997: 14-15).

Neuron behaviour

- Summate(). Adds the values of the the input dendrites, to the variable *netInput*, which is then passed to the appropriate activation function (McLeod et al 1998: 17) .
- binaryFunction(). Described in section 3.2.1 above. The value of the output variable *outAxon* is 0 or 1.

- sigmoidFunction(). Described in section 3.22. The value of outAxon is always >0 and <1.

5.3 The synapse class

The function of the Synapse class is to receive an input from a pre-synaptic neuron, multiply it by the synapse strength and output the result to the post-synaptic neuron, as shown in figure 20. During learning the synapse strength is adjusted in accordance with the Hebb rule or the delta rule.

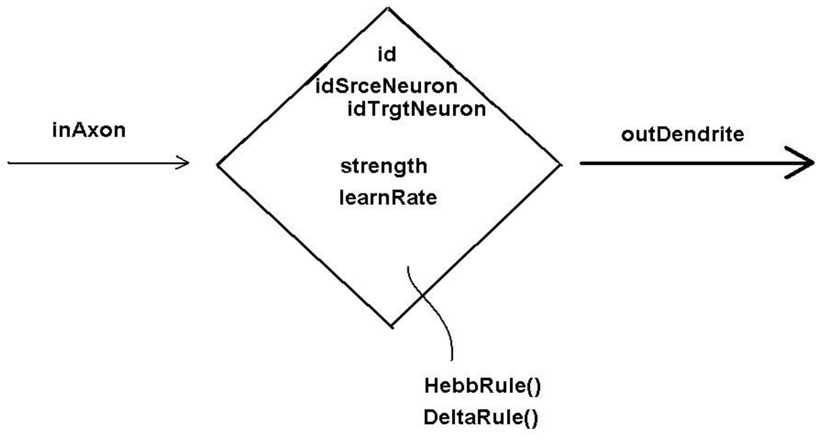


Figure 20: Schematic representation of Synapse class

Synapse properties

- id. A unique id number assigned when the synapse is created.
- idSrceNeuron. The id number of the source neuron from which the input value in inAxon is passed. Assigned by the Network class when the synapse object is created.
- idTrgtNeuron. The id number of the post-synaptic neuron to which the output value outDendrite is passed.
- inAxon. The value of the input from the pre-synaptic neuron.
- outDendrite. The value passed by the synapse to the post-synaptic neuron.
- strength. The value by which the value of inAxon is multiplied and then passed to outDendrite (McLeod et al 1998: 17). This value is adjusted during learning (McLeod et al 1998: 18-19).

- learnRate. A constant assigned by the Network class when the synapse object is created. It determines the rate at which the synapse strength is adjusted during learning (Gurney 1997: 43).

Synapse behaviour

- Hebb Rule(). Implemented as described in section 3.4.1 above, using the code:

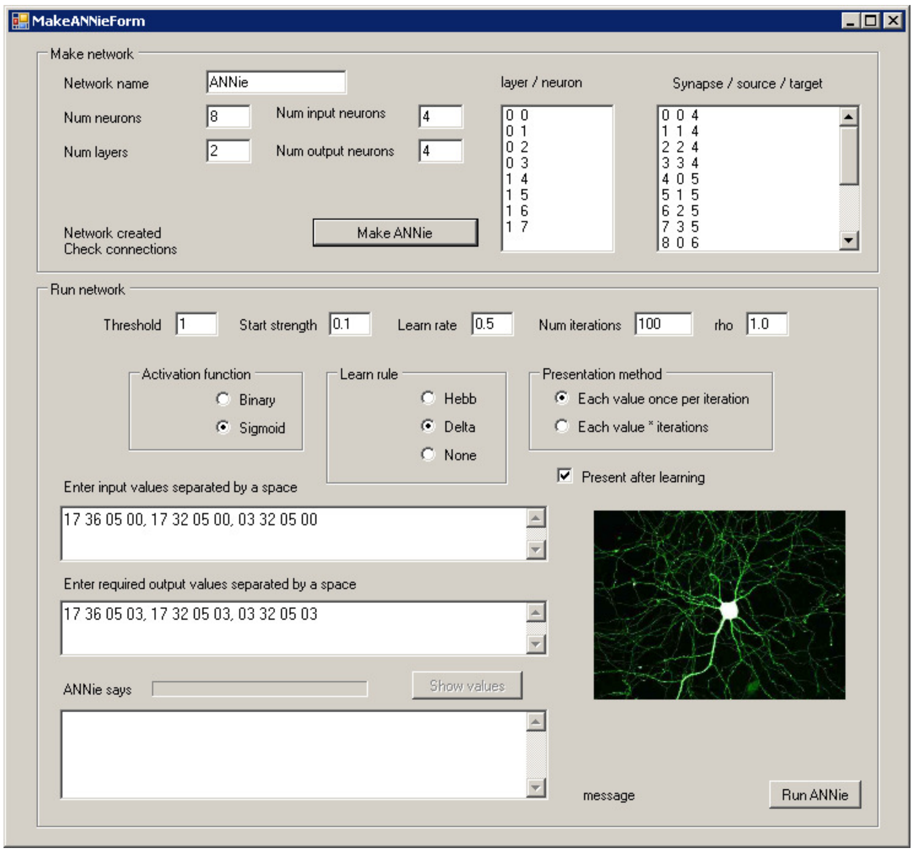
```
strengthChange = learnRate * inAxon * targetActivity
```

- Delta Rule(). Implemented as described in See section 3.4.2 above, using the code:

```
strengthChange = (targetActivity - obtainedActivity) *
        sourceActivity * learnRate;
```

5.4 The network class

The Network class creates and runs a network object consisting of neurons and synapses using specification input from the user via the handler form, MakeANNie shown in figure 21. The design adopted permits the creation of multiple network objects of varying size. This would enable the simulation of brain modularity through the creation of a network of connected networks. Moreover, through the creation of subclasses, it would be possible to incorporate auto-association, recurrent and other network designs into a single modular 'super' network.


MakeANNieForm

Make network

Network name: ANNie
Num neurons: 8
Num layers: 2
Num input neurons: 4
Num output neurons: 4

layer / neuron
```
0 0
0 1
0 2
0 3
1 4
1 5
1 6
1 7
```

Synapse / source / target
```
0 0 4
1 1 4
2 2 4
3 3 4
4 0 5
5 1 5
6 2 5
7 3 5
8 0 6
```

Network created
Check connections

Make ANNie

Run network

Threshold: 1
Start strength: 0.1
Learn rate: 0.5
Num iterations: 100
rho: 1.0

Activation function: Binary [✓ Sigmoid]
Learn rule: Hebb [✓ Delta] None
Presentation method: [✓ Each value once per iteration] Each value * iterations
[✓ Present after learning]

Enter input values separated by a space
17 36 05 00, 17 32 05 00, 03 32 05 00

Enter required output values separated by a space
17 36 05 03, 17 32 05 03, 03 32 05 03

ANNie says

Show values

message

Run ANNie

Figure 21: MakeANNie handler form

Functionally, the class consists of two parts: 'Make network' and 'Run network', representing respectively, anatomy and physiology. There is a case to split the Network class along these lines but they have been kept together for the moment for the convenience of passing variables to the instantiated network object.

5.4.1. The make network function

The 'make network' function of the Network class is shown schematically in figure 22.

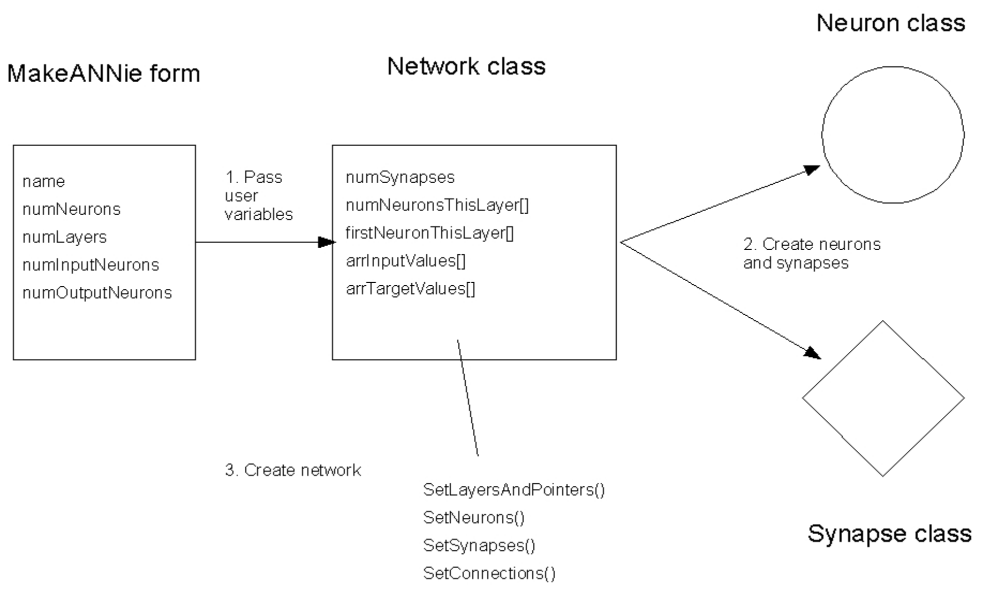


Figure 22: Make network function of Network class

It creates a network consisting of the number of neurons specified by the user. If the number of layers is greater than two, the number of neurons per hidden layer is calculated. The number of synapses is calculated according to the expression:

$$ns = \Sigma\, nn_i * nn_{i+1}$$

Where:

ns	is the total number of synapses
nn_i	is the number of neurons in the current layer
nn_{i+1}	is the number of neurons in the next layer

5.4.2 Creating neurons, synapses and connections

Neurons are instantiated with the appropriate layer number and number of dendrites. The number of dendrites per neuron is calculated according to the number of neurons in the preceding layer. Synapses are instantiated having calculated the number required as described above.

Making the connections requires iterating through each neuron in each layer and then connecting through a synapse to each neuron in the following layer. Each synapse stores the id number for its source and target neuron. This is done using the SetConnections() method as outlined in the following psuedocode:

```
thisSynapse = 0;
      for each targetLayer[i]
            for each targetNeuron[j] in targetLayer[i]
                  for each sourceNeuron[k] in sourceLayer[i-1]
                        thisSynapse.sourceNeuron = k;
                        thisSynapse.targetNeuron = j;
                        increment thisSynapse;
```

Figure 23: Set connections psuedocode

5.4.3 The run network function

The run function of the Network class receives the runtime values from the MakeANNie form and runs the network object using the methods shown in figure 24, under the control of methods in the MakeANNie form.

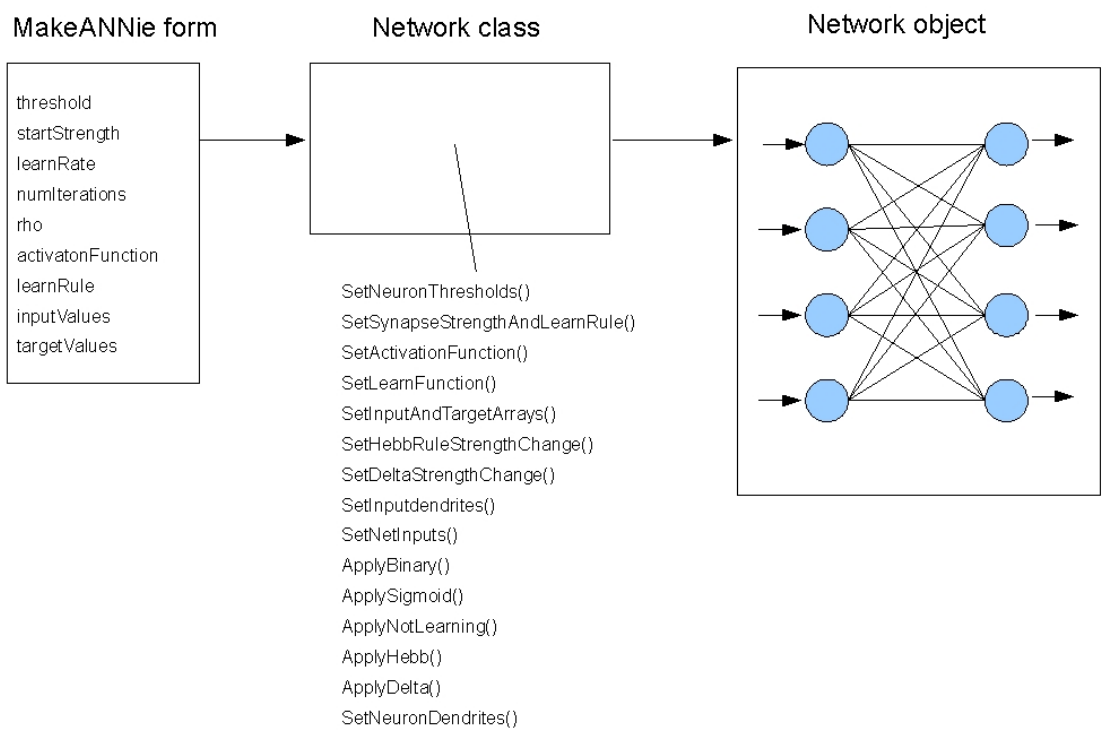


Figure 24: 'Run network' function of Network class

The running of the network object is outlined below. The code is listed and documented in Appendix B.

1. Set the user specified neuron thresholds, synapse start strengths and synapse learn rates. These values remain constant during the run.
2. Assign the user specified activation function (binary or sigmoid) and learning rule (Hebb or delta).
3. For each iteration:

   3.1 For each input word:

      3.1.1 If Hebb learning, set synapse strength change (constant for each character).

      3.1.2 present each character of the input word to the dendrites of the input layer.

      3.1.3 For each layer:

         Set the net inputs of the neurons in the layer.

         Apply the activation function and pass output values to neuron axons.

Pass axon values to synapses and apply learning function .

Pass synapse output values to dendrites of next layer.

3.1.4 Process output layer and display output values.

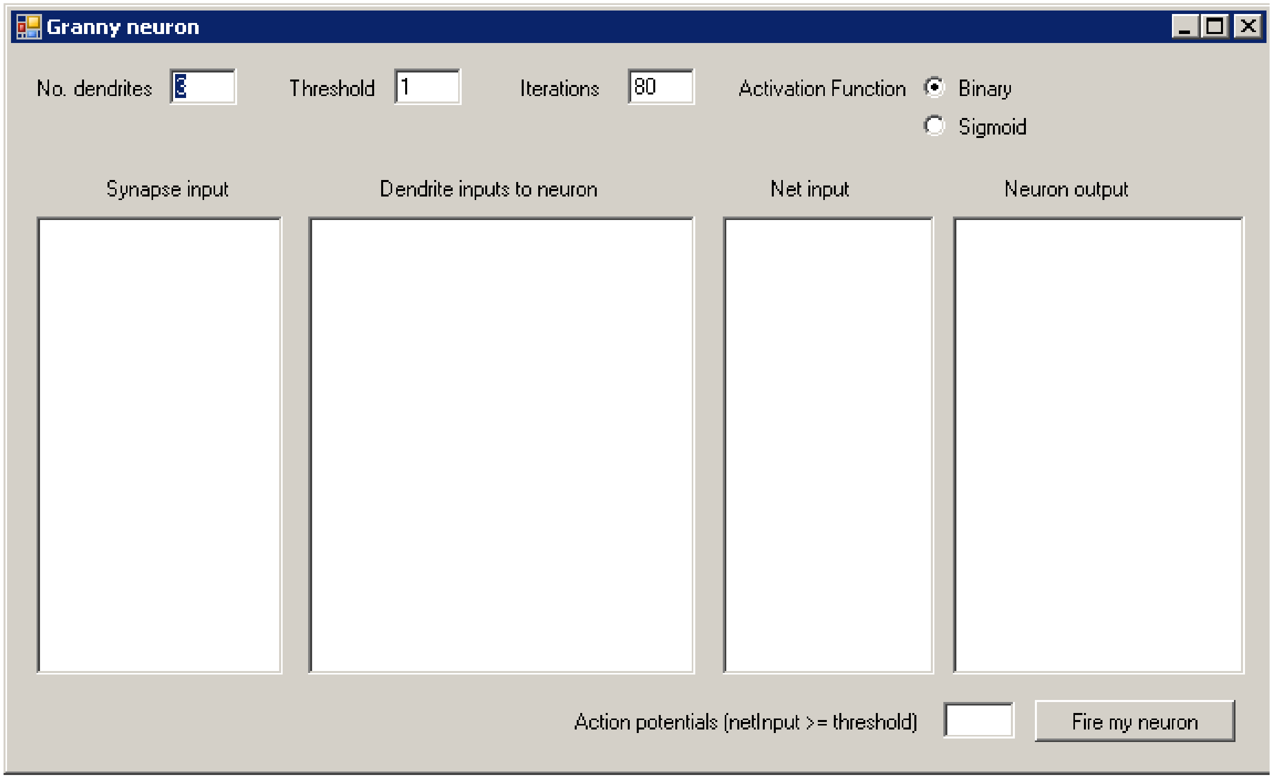
3.1.5 If delta learning, set synapse strength change for next word/iteration.

6 Testing the model

The objective of testing the model is to determine if it can simulate aspects of language acquisition. However, it is first necessary to determine if the network components, the neurons and synapses, behave in accordance with their specifications.

6.1 Testing a single neuron

A handler form was created, shown in figure 25 below, which instantiated a single neuron, shown schematically in figure 26. The number of dendrites (and thus instantiated synapses), the threshold, iterations and activation function are user determined. The input values were entered in the source code, though it would have been better to include these in the user input.


Granny neuron

No. dendrites 3 Threshold 1 Iterations 80 Activation Function [✓ Binary] Sigmoid

Action potentials (netInput >= threshold)

Figure 25: Handler form for single neuron and synapse testing

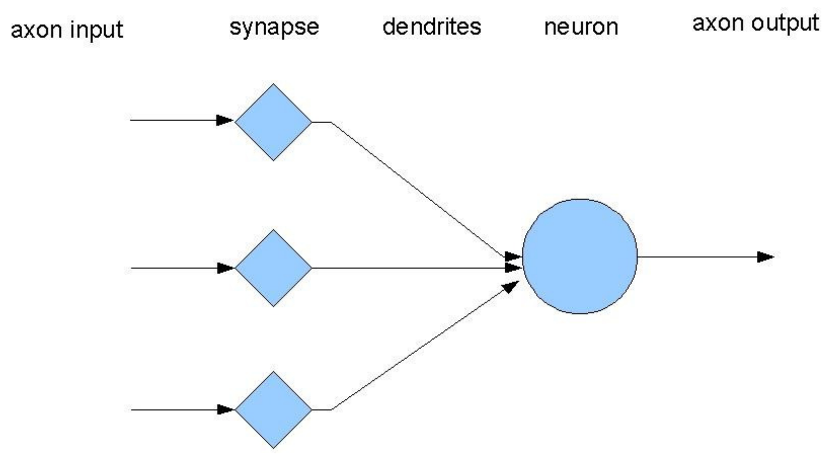


Figure 26: Testing a single neuron

Four sets of inputs were presented to the neuron as shown in table 1 below. Detailed results are shown in appendix C.

The Hebb rule (`strengthChange = learnRate * inAxon * targetActivity`) was used, with the learn rate fixed at 0.25 and the target value fixed at 1.0. This rule continuously increases the synapse strength providing all values in the expression are greater than zero.

Table1: Neuron testing; dendrites=3, threshold=1, iterations=100/150, binary function

Inputs	**Predicted outputs**	**Reason for prediction**	**Actual outputs**
All = 0	Always = 0	The synapses are never strengthened since the Hebb Rule is multiplicative; thus the threshold is never achieved.	As predicted
Sum >= 1	Always = 1	The threshold is always achieved.	As predicted
All = 0.1	Initially = 0; then always = 1	Positive inputs to the synapses are gradually strengthened until the threshold is achieved.	As predicted
Each >0.1<0.25	Initially = 0 then = 0 or 1; then always = 1	As previous but random input implies a brief intermediate period.	As predicted, but see comments in text.

The fourth test result, where the input values are randomised with in a fairly narrow range, is particularly interesting since it probably more closely models actual neural behaviour than the others. Neurons are known to exhibit 'noise', that is, frequent changes in the signal outputs at low value levels (McLeod et al 1998: 32-33, 44) (Ward 2006: 38, 42).

The output patterns during the intermediate period of test four varied surprisingly. The details are shown in appendix C.

6.2 Testing the network with a single input pattern

The most basic operation of a neural network is pattern association: when one pattern is presented as input another pattern is generated as output. This simple pattern association is said to provide a cognitive mechanism for the psychological phenomenon of stimulus-response (see sections 3.4.1 and 3.5 above and McLeod 1998: 52).

The model was first presented with a number of simple binary patterns and in each case trained to generate a different pattern. The model was then presented with more complex patterns representing a set of phonemes and trained to produce a modified phonemic pattern representing an inflection of the original. In each case the original pattern was presented *after* training to ensure that the association had been learned.

6.2.1 Binary pattern association

This stage of testing was carried out using the binary activation function and the Hebb rule (Rolls and Treves 1998: 6-7). The model was two layer with four input and four output neurons. The initial threshold was initially set to 1.0, with a start strength of 0.1, and a learn rate of 0.5. Initial training was for 100 iterations. The results are summarised in table 2.

Table 2: Binary pattern association

Input pattern	Target pattern	Actual output2
1.00 0 1.00 0	0 1.00 0 1.00	0.31 1.00 0.31 1.00
1.00 1.00 0 0	0 0 1.00 1.00	0.33 0.33 1.00 1.00
1.00 0 0 0	1.00 0 0 1.00	1.00 0.33 0.33 1.00
0 1.00 0 0	0 1.00 1.00 1.00	0.33 1.00 1.00 1.00

An examination of the intermediate output values showed that where the target value was 1.00, the corresponding actual output incremented rapidly to match. However, where the target value was 0, the actual output remained constant. The intermediate values for the first seven iterations of the first pattern are shown in table 3.

Table 3: Intermediate values for first pattern in table 2

Iteration	Output value
0	0.31 0.55 0.31 0.55
1	0.31 0.77 0.31 0.77
2	0.31 0.90 0.31 0.90
3	0.31 0.96 0.31 0.96
4	0.31 0.99 0.31 0.99
5	0.31 0.99 0.31 0.99
6	0.31 1.00 0.31 1.00

The values of the threshold, start strength and learn rate were then individually adjusted to determine their effect on the output pattern. The results are summarised in tables 4, 5 and 6.

Table 4: Binary pattern association: threshold=6, others as table 2

Input pattern	Target pattern	Actual output
1.00 0 1.00 0	0 1.00 0 1.00	0 1.00 0 1.00
1.00 1.00 0 0	0 0 1.00 1.00	0 0 1.00 1.00
1.00 0 0 0	1.00 0 0 1.00	1.00 0 0 1.00
0 1.00 0 0	0 1.00 1.00 1.00	0 1.00 1.00 1.00

Adjusting the threshold to 6, had the effect of matching the actual output to the target output. Examination of the intermediate values showed that on the first iteration where

2 Values are input with two decimal places, represented internally with 14 decimal places, and rounded back to two decimal place for output

the target value was 0 the actual output was also 0, but where the target value was 1.00 the output was 0.01. However, in order to achieve this result it was necessary to run the model with a threshold of 1. If the network was instantiated and immediately run with a threshold of 6, all 0 results are returned.

Table 5: Binary pattern association: start strength=0.25, others as table 2

Input pattern	**Target pattern**	**Actual output**
1.00 0 1.00 0	0 1.00 0 1.00	0.45 1.00 0.45 1.00
1.00 1.00 0 0	0 0 1.00 1.00	0.45 0.45 1.00 1.00
1.00 0 0 0	1.00 0 0 1.00	1.00 0.45 0.45 1.00
0 1.00 0 0	0 1.00 1.00 1.00	0.45 1.00 1.00 1.00

Table 6: Binary pattern association: learn rate=1.0, others as table 2

Input pattern	**Target pattern**	**Actual output**
1.00 0 1.00 0	0 1.00 0 1.00	0.34 1.00 0.34 1.00
1.00 1.00 0 0	0 0 1.00 1.00	0.34 0.34 1.00 1.00
1.00 0 0 0	1.00 0 0 1.00	1.00 0.34 0.34 1.00
0 1.00 0 0	0 1.00 1.00 1.00	0.34 1.00 1.00 1.00

Adjusting the start strength or the learn rate increases the actual output value corresponding to the 0 target value. Various other values of the start strength and learn rate were tested: in each case the higher the variable value the higher the non 1.0 output value.

6.2.2 Non-binary pattern association

In this section the same model is presented with the nine 'high frequency' verbs used by McClelland and Rumelhart (1986: 240). The binary coded *Wickelfeatures* used by McClelland and Rumelhart were avoided since they had been criticised for failing to give a distinct, that is symbolic, representation to the phonemes being manipulated (Pinker and Prince 1988: 175). The input and output values in this model are thus given distinct symbolic form.

The phonetic forms were converted into numeric codes, shown in table 7, using a table in a student course book (Soars and Soars 2003: 159). This coding was chosen because the phonemes are grouped in a useful manner: codes 1 to 24 are consonants, 25 to 37 are vowels and 38 to 45 are diphthongs. Voiced and unvoiced phonemes are adjacent, as are short and long sounds. Thus if the model produces errors, the degree of error is to some degree measurable according to its variance from the target value.

Table 7: Verb codes presented to the model

Written		IPA		IPA code	
come	came	/kʌm/	/keɪm/	05 35 12 00	05 38 12 00
get	got	/get/	/gɒt/	06 28 03 00	06 31 03 00
give	gave	/gɪv/	geɪv	06 26 08 00	06 38 08 00
look	looked	/lʊk/	/lʊkt/	11 33 05 00	11 33 05 03
take	took	/teɪk/	/tʊk/	03 38 05 00	03 33 05 00
go	went	/gəʊ/	/went/	06 39 00 00	17 28 13 03
have	had	/hæv/	/hæd/	14 29 08 00	14 29 04 00
live	lived	/lɪv/	/lɪvd/	11 26 09 00	11 26 09 04
feel	felt	/fiːl/	/felt/	07 25 11 00	07 28 11 03

The model was presented with the code for each verb individually, the synapse values being reset to the value of *startStrength* between verbs: the model was *tabula rasa* for each verb. Initially, the model was run for 100 learning iterations, using the sigmoid function and delta learning rule. The phoneme values shown in bold are those that change to form the past tense (for example, /ʌ/ to /eɪ/ for come --> came).

Table 8: Finding the past tense of high frequency verbs.

Verb (present tense)	**Target value (past tense)**	**Value after 100 iterations**	**No of iterations to capture past phoneme(s)**	**No of iterations to capture target value**
come	05 **38** 12 00	08 **38** 13 05	70	1135
get	06 **31** 03 00	09 **31** 07 06	37	1202
give	06 **38** 08 00	09 **38** 10 05	73	1192
look	11 33 05 **03**	13 33 08 **07**	374	374
take	03 **33** 05 00	07 **33** 08 05	53	1153
go	**17 28 13 03**	**18 28 14 07**	376	376
have	14 29 **04** 00	15 29 **08** 05	316	1145
live	11 26 09 **04**	13 26 11 **08**	322	322
feel	07 **28** 11 **03**	10 **28** 13 **07**	385	385

The model showed a tendency to quickly capture the past phoneme for the verbs 'come', 'get', 'give' and 'take'. In each case the phoneme change is a single vowel preceded by a consonant. Also, in each case the changed phoneme was the second phoneme in the word. This could indicate either some interesting pattern seeking behaviour or, a fault in the design of model biasing it toward the second number in each value. The model was therefore re-made with six input and six output neurons, thus shifting the position of the change vowel from 2 to 3. The model was then re-run with the same data, as shown in table 9.

Table 9: Effect of shifting position of past phoneme

Verb (present tense)	**Target value (past tense)**	**Value after 100 iterations**	**No of iterations to capture past phoneme(s)**
come	00 05 **38** 12 00 00	04 07 **38** 13 04 04	47
get	00 06 **31** 03 00 00	04 08 **31** 06 04 04	6
give	00 06 **38** 08 00 00	04 08 **38** 09 04 04	49
look	00 11 33 05 **03** 00	04 12 33 07 **06** 04	265
take	00 03 **33** 05 00 00	04 06 **33** 07 04 04	29
go	00 17 **28 13 03** 00	04 **17 28 14 06** 04	816
have	00 14 29 **04** 00 00	04 14 29 **06** 04 04	224
live	00 11 26 09 **04** 00	04 12 26 10 06 04	227
feel	00 07 **28** 11 **03** 00	04 08 **28** 12 **06** 04	270

The results were similar to those in table 8; with the exception of 'go' the past phonemes were captured rather quicker than previously.

The verbs whose phoneme changes were quickly captured are examples of infix inflections, in contrast with the suffix inflections of regular verbs in English. Infix inflexions are common in Arabic: the root form *ktb,* for example, is infixed with vowels to form words connected with writing (Fromkin et al 2008: 515). The model was therefore tested with a sample of such infixes, and the results are shown in table 10. The input root form was infixed with the weak vowel /ə/. Only the first five are actual Arabic words: the rest were assigned random vowels.

Table 10: Testing infix inflections on root /kətəb/

Target word	**Target value**	**Value after 100 iterations**	**No of iterations to capture infix phoneme(s)**	**No of iterations to capture target value**
/kætæb/	05 **29** 03 **29** 02	07 **29** 06 **29** 05	14	346
/kɪtæb/	05 **26** 03 **29** 02	07 **26** 06 **29** 05	44	346
/æktɪb/	**29** 05 03 **26** 02	**29** 07 06 **26** 05	44	346
/kɪtaːb/	05 **26** 03 **30** 02	07 **26** 06 **30** 05	44	346
/kʊtʊb/	05 **33** 03 **33** 02	07 **33** 06 **33** 05	35	346
/kʌta:b/	05 **35** 03 **30** 02	07 **35** 06 **30** 05	44	346
/ki:tɒb/	05 **25** 03 **31** 02	07 **25** 06 **31** 05	49	346
/kətu:b/	05 **37** 03 **34** 02	07 **37** 06 **34** 05	49	346
/ketɜːb/	05 **27** 03 **36** 02	07 **27** 60 **36** 05	47	346
/ku:təb/	05 **34** 03 **37** 02	07 **34** 06 **37** 05	49	346

The infixes were captured very quickly, an average of 42 iterations, compared with an average 223 iterations for the English verbs in table 8. In one case, the declension /æktıb/ (I write), the vowel pattern is *prefix* + *infix*; nevertheless the past phonemes are captured similarly quickly to the *infix* + *infix* pattern.

6.3 Testing the network with multiple input patterns

In the various models described in section 4, a single network is trained using a set of input values. The network is subsequently presented with both trained and novel values. The following section attempts to repeat these experimental results using binary and non-binary values.

6.3.1 Binary pattern association

Sets of binary patterns were trained and then presented, using the Hebb rule and the Binary activation function. The results are shown in table 11 below.

Table 11: Binary pattern association with multiple values

Input set	**Target set**	**Output set**
1.0 0 1.0 0, 0 1.0 1.0 0	0 1.0 0 1.0, 1.0 1.0 0 0	1.0 1.0 0 1.0, 1.0 1.0 0 1.0
1.0 1.0 0 0, 0 0 1.0 0	0 0 1.0 1.0, 1.0 0 0 1.0	1.0 0 1.0 1.0, 1.0 0 1.0 1.0
0 0 0 0, 1.0 1.0 1.0 1.0	1.0 0 0 0, 1.0 1.0 1.0 0	1.0 1.0 1.0 0 1.0 1.0 1.0 0

In each case the output value was the same for each value presented, and, apart from the second value in row three, the output set failed to match the target set.

6.3.2 Non-binary pattern association

The nine verb codes from table seven were presented as a set, using the delta rule and the sigmoid activation function. The results are shown in table 12.

Table 12: High frequency verbs presented as a set

Verb (present tense)	**Target value (past tense)**	**Value after 10 iterations**	**Value after 100 iterations**
come	05 38 12 00	11 31 10 07	09 31 08 02
get	06 31 03 00	12 31 11 07	09 31 08 02
give	06 38 08 00	12 31 11 07	09 31 08 02
look	11 33 05 03	11 31 11 07	09 31 08 02
take	03 33 05 00	11 31 11 07	09 31 08 02
go	17 28 13 03	11 31 11 07	09 31 08 02
have	14 29 04 00	11 31 10 07	09 31 08 02
live	11 26 09 04	12 31 11 07	09 31 08 02
feel	07 28 11 03	12 31 11 07	09 31 08 02

Again, the output values failed to match the target values. They showed some slight variations in the early iterations but by 100 iterations were identical. The model was tested with a reduced set of values, and various threshold, start strength, learn rate, and rho values were used. However, the results in each case were similar.

6.4 Analysis and interpretation of results

In section 6.1 a single neuron was tested using the binary activation function and the Hebb rule. The results from this testing was in accordance with predicted behaviour and suggests that the expressions were coded correctly. Further tests with the sigmoid function, shown in Appendix C.6, demonstrated similar behaviour.

In section 6.2.1 binary pattern matching with the Hebb rule was successful where the threshold value was set to 6.0. At lower threshold values the initial neuron outputs were always greater than zero and since the Hebb rule is positively cumulative, the output would never match zero values. The effect of the higher threshold setting is to suppress all initial output values to zero enabling the connections to the 1.0 target outputs to be incremented, leaving the zero target outputs unchanged. This behaviour suggests that the connectivity of the model was correctly coded. Whether it says anything about brain behaviour is uncertain since it is not clear if the simplifications of the Hebb rule and binary function are represented in actual neural behaviour.

In section 6.2.2 the model was presented with non-binary values representing English present tense phoneme strings. The model always generated the appropriate past tense form. The model generated the past tense more quickly where the morphological change was a vowel change in the form *consonant+vowel*. When the position of the *consonant+vowel* pair was shifted, similar results were obtained, suggesting that the results were not the consequence of a fault in the model biasing the results to the second neuron.

Testing the model with infix morphological changes of the type *consonant+vowel+consonant+vowel+consonant*, characteristic of Arabic and other Semitic languages, produced the most interesting results. The past phoneme change was in each case captured in less than 50 iterations and the full morphological change was captured in exactly 346 iterations for each value. This compares with 374 and 322 iterations respectively to capture the past phoneme change in the English regulars 'look' and 'live', although in these cases the past phonemes were the last to be captured. The model demonstrates an ability to find *infix* + *infix* and *prefix* + *infix* patterns more quickly than other morphological forms.

However, some caution is necessary in interpreting these results since the model failed to capture multiple inputs in the manner of the McClelland and Rumelhart and other models described in section 4. It is possible that there is some fault in the coding of the classes, but this seems unlikely given the testing described above. It is more likely that this failure is due to the symbolic rather than binary form in which the values were presented, which may explain the use of binary input in several of the models described, including the wickelfeatures used by Rumelhart and McClelland. Alternatively, the failure may be due to the manner in which the delta rule was implemented. Figure 13 on page 24 shows an error gradient which represents *global* rather than *local* minima (Gurney 1997: 69) and it may be necessary to modify the expressions in the model to achieve this, using the linear algebra methods described in Gurney (1997: chapter 3).

7 Conclusions

7.1 The research model

The model constructed for this research produced some interesting results especially with respect to infix morphology. It may be that this type of morphology is particularly suitable to the pattern finding abilities of connectionist models, which implies that children may acquire infixes more quickly than other morphology. However, no reference has been found to such a phenomenon in the literature.

To research this possibility further it is first necessary to develop the model so that it successfully captures multiple as well as single input patterns, as discussed in the previous section. The model could then be presented with appropriate words, not restricted to verbs, in a cross-linguistic study of English and Arabic, or of the Germanic and Semitic families. If the results found in this study were confirmed, then a cross-linguistic study of native English and native Arabic speaking children could be carried out to determine if they acquire infix morphology more quickly than other forms.

The model uses Object Oriented design, which allows any number of network objects to be constructed. Sub-classes may also be created incorporating the various network designs described in section 3.5. Further, it permits the construction of a network of networks which more closely simulates brain functionality, with the possibility of attempting to simulate higher cognitive functions. It may even be possible to construct a network of networks which is constrained by some kind of content addressable parameters (see section 3.5.3), to provide a limited simulation of an innate language acquisition device.

7.2 The value of connectionist models

This research has shown that connectionist models may be of practical value to scholars in the field of language acquisition, since they may generate hypotheses which can then be used in child development studies. However, this proposition comes with two important caveats. Firstly, the models themselves must be physiology adequate. Simplification is justified on the grounds that it allows the most important characteristics of the model to be observed clearly: in this case the behaviour of synapses during the learning process. However, more complex models, which come closer to representing real brain behaviour, are necessary if connectionism is to achieve more than the simulation of relatively narrow aspects of language acquisition; and in particular to offer some account of higher cognitive functions such as linguistic prototyping. Modelling techniques, perhaps including back-propagation, which do not seem to be physiologically adequate, should be avoided.

The second caveat is that the connectionist modelling of language acquisition should be carried out by linguists who are competent in software development and have an adequate understanding of physiology of the brain. It may well be that a dearth of such expertise, combined with the particular challenges that language presents modellers, accounts for the relative lack of research in the field since the work of Rumelhart and McClelland, Elman and others in the 1980s and 1990s. It may also explain why connectionism is often characterised as being in opposition to other branches of linguistics, including cognitive linguistics.

Notwithstanding the bold assertion of McClelland and Rumelhart (section 4.1.1 above), subsequent researchers appear to have been cautious about the claims they make for connectionism. Linguistic rules may be characterised as emergent properties of models, which is not to claim that the rules themselves do not exist. Connectionists seem to support the general proposition of cognitive linguists, and usage theorists such as Tomasello, that language acquisition is a function of general cognitive skills rather than any innate language module such as UG.

One potential area of collaborative research between cognitive linguists and connectionists, is that of simulated embodiment. A network of networks, based on the auto-associative model described in section 4.2.1, could be connected to a number of devices which simulate the human senses. Further, such a network, incorporating motor function simulation described in section 3.6.1, could be connected to a device with spatial awareness and movement functions: that is, a robot.

Connectionism, taken to the extreme, appears to represent a reductionist epiphenomenalism in which all of human behaviour, including language, has a purely physical cause: the complex combinations of the electro-chemical outputs of neurons. By this account, mental events are a consequence, not a cause, of physical events. Such a proposition seems to dismiss human conciousness as the mere froth of neuronal activity. The spectre of epiphenomenalism may account for some of the hostility towards connectionism. However, no connectionist model comes close to such an account and theorists such as Elman (see section 4.1.1) seem content to describe connectionism as a vehicle for the study of human development as conceptual emergence.

word count = 12,923

References

Aitchison, J. (2008) *The Articulate Mammal*. London: Routledge.

Alberts, B. et al (1998) *Molecular Biology of the Cell: second edition*. London: Garland Publishing.

Analytical Engine (2008). *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-216027/computer> (Accessed 10 May 2008).

Artificial intelligence (2008). *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-219085/artificial-intelligence> (Accessed 10 May 2008).

Bennett, MR. & Hacker, PMS (2007) *Philosophical Foundations of Neuroscience*. Oxford: Blackwell.

Churchland, PM, and Churchland, PS. (1990) 'Could a Machine Think?'. *Scientific American*, edition January 1990 (pp. 32-37). Washington DC: Scientific American.

Chomsky, N. (1957) *Syntactic Structures*. The Hague: Mouton.

Crystal, D. (2003) *The Cambridge Encyclopedia of Language*. Cambridge: Cambridge University Press.

Dyer, M. (1990) 'Intentionality and Computationalism'. *Journal of Experimental and Theoretical Artificial Intelligence, 2(4), 303-319.*

Elman, JL. et al. (1998) *Rethinking Innateness*. London: MIT Press.

Elman, JL. (1990) Finding Structure in Time. *Cognitive Science* (14) , 179-211.

Evans, V. and Green, M. (2006) *Cognitive Linguistics An Introduction*. Edinburgh: Edinburgh University Press.

Forouzan, B. and Mosharraf, F. (2008) *Foundations of Computer Science*. London: Thomson.

Fromkin, V. et al (2007) *An Introduction to Language*. Boston: Thomson Wadsworth.

Furber, S. et al (2006) 'High-Performance Computing for Systems of Spiking Neurons', *AISB'06 workshop on GC5: Architecture of Brain and Mind*. Bristol, 3-4 April 2006. Vol.2, pp 29-36. Online: http://intranet.cs.man.ac.uk/apt/publications/papers_05.php (Accessed 10 June 2008).

Gluck, M. and Myers, C. (2001) *Gateway to Memory: An Introduction to Neural Network Modeling of the Hippocampus and Learning (Issues in Clinical and Cognitive*

Neuropsychology). London: MIT Press.

Gross, R. (1996) *Psychology The Science of Mind and Behaviour*. London: Hodder & Stoughton.

Gurney, K. (1997) *An Introduction to Neural Networks*. London: CRC press.

Jacobs, RA. (1991) 'Task decomposition through competition in a modular connectionist architecture'. *Cognitive Science*, 15(2), 219-250.

Lerer, S. (1998) 'Modern Linguistics and the Politics of Language' Lecture 35 in audio course, *The History of the English Language*. Chantilly: The Teaching Company.

Lewis, JD. and Elman, JL. (2001) A connectionist investigation of linguistic arguments from poverty of the stimulus: Learning the unlearnable. *Proceedings of the 23rd Annual Conference of the Cognitive Science Society,* pp. 552-557, Erlbaum.

Lovelace (2008). *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-216028/computer> (Accessed 10 May 2008).

Lycan, WG. (2000) *Philosophy of Language*. Abingdon: Routledge.

McClelland, J. & Rumelhart, D. (1986) *Parallel Distributed Processing Volume 2: Psychological and Biological Models*. London: MIT Press.

McLeod, P. et al (1997) *Introduction to Connectionist Modelling of Cognitive Processes*. Oxford: Oxford University Press.

Marcus, G. (2001) *the Algebraic Mind*. Cambridge: MIT Press.

Penrose, R. (1989) *The Emperor's New Mind*. London: Vintage.

Pinker, S. (1994) *The Language Instinct*. London:Penguin.

Pinker, S. (1999) *Words and Rules*. London: Phoenix.

Pinker, S. and Prince, A. (1988). 'On Language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition,* 28, 73-193.

Plunkett, K. and Marchman, V. (1993) 'From rote learning to system building: acquiring verb morphology in children and connectionist nets'. *Cognition,* 48, 21-69.

Plunkett, K. et al (1992) Symbol grounding or the emergence of symbols? Vocabulary growth in children and a connectionist net. *Connection Science,* 4, 293-312.

Rolls, E, and Treves, A. (1998) *Neural Networks and Brain Function*. Oxford: Oxford University Press.

Rogers, J. (1997) *Object-Oriented Neural Networks in C++*. London: Academic Press

Rumelhart, D. and McClelland, J. (1986) *Parallel Distributed Processing Volume 1: Foundations*. London: MIT Press.

Searle, JR. (1980) 'Minds, brains, and programs'. *Behavioral and Brain Sciences,* 3 (3), 417-457.

Sejnowski and Rosenberg (1987) 'Parallel Networks that Learn to Pronounce English Text'. *Complex Systems,* 1, 145-168.

Sapolsky, R. (2005) *Biology and Human Behaviour: The Neurological Origins of Individuality*. (Audio course). Chantilly: The Teaching Company.

Smith, N. (1999) *Chomsky Ideas and Ideals*. Cambridge: Cambridge University Press.

Soars, L. and Soars, J. (2003) *New Headway Intermediate Student's Book: the third edition*. Oxford: Oxford University Press.

Taylor, JR. (1989) *Linguistic Categorization*. Oxford: Oxford University Press.

Thompson, M. (2003) *Philosophy of* Mind. London: Hodder Education.

Troelsen, A. (2007) *Pro C# with .NET 3.0*. Berkeley: Apress.

Turing Machine (2008a) *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-216036/computer> (Accessed 11 May 2008).

Turing Machine (2008b) *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-9001510/Turing-machine> (Accessed 17 May 2008).

Turing Test (2008) *Encyclopædia Britannica*. Online. <http://www.britannica.com/eb/article-219090/artificial-intelligence> (Accessed 17 May 2008).

Tomasello, M. (2005) *Constructing a Language*. London: Harvard University Press.

Ward, J. (2006) *The Student's Guide to Cognitive Neuroscience*. Hove: Psychology Press.

Appendix A: The neuron and synapse Classes

The documented code for the Neuron and Synapse classes is listed on the following pages. The code was written in C# using the Microsoft Visual C# 2008 Express Edition.

A.1 Neuron class code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ANNie
{
    class Neuron
        // The Neuron receives input via the array inDendrite[], summates the values
        // of the dendrites in SetNetInput(), processes them in BinaryFunction()
        // or SigmoidFunction() and asigns the result to outAxon (and thus to Synapse).
        // The layer property is calculated at run time
        // The actionPotential property is set if netInput >= threshold (Gurney 1997: 11)
        // The properties id, layer and threshold are assigned by the Network class
        // when the ANN is created. Neurons in the input layer have one only one dendrite.
        //
    {
        protected int _id;          // each neuron has a unique id 0-n
        protected int _layer;       //  the layer number of the neuron
        protected int _numDendrites;    // number of dendrites
        protected internal double[] inDendrite = new double[] {}; // see notes 1 & 2 below
        protected double _outAxon;      // the output from the neuron
        protected double _threshold;    // the threshold at which the neuron activates
        protected double _netInput;     // sum of inDendrites[i], passed to activation function
        protected int _actionPotential;       // set to 1 if netInput >= threshold;
        protected int _numActionPotentials;   // number of activations in current run
        protected double _rho;  // used to change slope of sigmoid
        // Get Set methods
        public int id
        {
            get { return _id; }
            set { _id = value; }
        }
        public int layer
        {
            get { return _layer; }
            set { _layer = value; }
        }
        public int numDendrites
        {
            get { return _numDendrites; }
            set { _numDendrites = value; }
        }
        public double this[int index]
        {
            get { return inDendrite[index] ; }
            set { inDendrite[index] = value; }
        }
        public double outAxon
        {
            get { return _outAxon; }
            set { _outAxon = value; }
        }
        public double threshold
        {
            get { return _threshold; }
            set { _threshold = value; }
        }
        public double netInput
        {
            get { return _netInput; }
            set { _netInput = value; }
        }
        public int actionPotential
        {
            get { return _actionPotential; }
            set { _actionPotential = value; }
        }
        public int numActionPotentials
        {
            get { return _numActionPotentials; }
            set { _numActionPotentials = value; }
        }
        public double rho
        {
            get { return _rho; }
            set { _rho = value; }
        }

        //                  Constructors
        //
        // This one is called from Network class
        public Neuron(int idNum, int layNum, int numConnections)
        {
            id = idNum;
            layer = layNum;
            numDendrites = numConnections;
            initDendrites();
            outAxon = 0.0;
            threshold = 0.0;
            actionPotential = 0;
            numActionPotentials = 0;
        }
        // This one is called from grannyForm
        public Neuron(int idNum, int layNum, int numConnections, double theta)
        {
            id = idNum;
            layer = layNum;
            numDendrites = numConnections;
            initDendrites();
            outAxon = 0.0;
            threshold = theta;
            actionPotential = 0;
            numActionPotentials= 0;
        }
        // Called from constructor
        // Set the inDendrite[] array size to the number of dendrites
        // and initialise the inDendrite[] array variables to zero
        public void initDendrites()
        {
            Array.Resize(ref inDendrite, numDendrites);
            foreach (int connection in inDendrite)
              {inDendrite[connection] = 0.0;}
        }
        //
        //                      Class methods
        //
        // Summates the input dendrites to netInput
        public void SetNetInput()
        {
            netInput = 0.0;
            for (int j = 0; j < numDendrites; j++)
                netInput += inDendrite[j];
        }
        // Set actionPotential and iterate numActionPotentials if netInput >= threshold
        // Used for reporting purposes only.
        public void SetActionPotential()
        {
            if (netInput >= threshold)
            {
                actionPotential = 1;
                numActionPotentials++;
            }
            else
                actionPotential = 0;
        }
        // Binary function. (McLeod et al 1998: 17-19) Output is either 0 or 1
        // .. ! else '0' produces permanent 0 output with Hebb Rule! Need to look at this.
        // Hebb works OK with the sigmoid function
        public void BinaryFunction()
        {
            if (netInput >= threshold)
                outAxon = 1.0;
            //else
            //    outAxon = 0.0;
        }
        // Sigmoid function (Gurney 1997: 19) An 'S' shaped function which
        // accentuates input values around the threshold value, flattens values
        // which are higher or lower. Rho changes the slope of the sigmoid and is
        // set by the user at runtime using the Network.SetSigmoidRho() method
        //
        public void SigmoidFunction()
        {
            //double rho = 1.0;   // need to Set this from Network class
            outAxon = 1.0 / (1.0 + Math.Exp(-(netInput - threshold) / rho));
        }
    }

}
//Note 1: inDendrite is an array of the inputs to the neuron.
// Dendrite, because the inputs to any neuron, apart from those in layer 0,
// are via dendrites.
//
// Note 2: protected internal int[] inDendrite = new int[] {};
//
// C# does not permit the creation of an array field in the class definition
// in the same way as for non-array variables. It must be created using
// an 'indexer'. It does not permit the use of the protected keyword alone.
// This means that inDendrite is exposed and directly accessible to code,
// unlike the other fields. It also does not permit the use of pointer addresing
// in the field definitions. Odd. I have decided to live with this for the time
// being since I want the array inDendrite to be accessible in the same way as the
// other properties (neuron.inDendrite[i]). The alternative, presumably, would be
// to create the array outside the class, in the source code, which I'm not keen on.
//
```

A.2 Synapse class code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ANNie
{
    class Synapse
        // Neurons are connected to each other via a synapse. The synapse knows which
        // neurons it is connected to by the idSrceNeuron and idTrgtNeuron properties.
        // Each synapse has a stored strength which is changed according to the Hebb or
        // Delta Rule, when learning is taking place. When no learning is taking place
        // the strength property is unchanged.
        // Data values arrive at the synapse inAxon property, from the source neuron
        // are strengthened, and passed to outDendrite and on to the target neuron.
    {
        protected int _id;              // unique id of the synapse object
        protected int _idSrceNeuron;    // unique id of source neuron
        protected int _idTrgtNeuron;    // unique id of target neuron
        protected double _inAxon;       // input value from axon of source neuron
        protected double _outDendrite;  // output value to dendrite of target neuron
        protected double _learnRate;    // Used in Hebb and Delta Rule
        protected double _strength;     // Strength of the synapse calculated by learning rule
        protected double _strengthChange;   // Calculated by Hebb or Delta rule
        protected double _targetActivity;   // Used by the delta Rule.
        protected double _obtainedActivity; // Delta Rule
        // Get Set methods
        public int id
        {
            get { return _id; }
            set { _id = value; }
        }
        public int idSrceNeuron
        {
            get { return _idSrceNeuron; }
            set { _idSrceNeuron = value; }
        }
        public int idTrgtNeuron
        {
            get { return _idTrgtNeuron; }
            set { _idTrgtNeuron = value; }
        }
        public double inAxon
        {
            get { return _inAxon; }
            set { _inAxon = value; }
        }
        public double outDendrite
        {
            get { return _outDendrite; }
            set { _outDendrite = value; }
        }
        public double learnRate
        {
            get { return _learnRate; }
            set { _learnRate = value; }
        }
        public double strength
        {
            get { return _strength; }
            set { _strength = value; }
        }
        public double strengthChange
        {
            get { return _strengthChange; }
            set { _strengthChange = value; }
        }
        public double targetActivity
        {
            get { return _targetActivity; }
            set { _targetActivity = value; }
        }
        public double obtainedActivity
        {
            get { return _obtainedActivity; }
            set { _obtainedActivity = value; }
        }
        //
        //              Constructors
        //
        // This one is called from Network class
        public Synapse()
        {
            id = 0;
            idSrceNeuron = 0;
            idTrgtNeuron = 0;
            learnRate = 0.0;
            inAxon = 0.0;
            outDendrite = 0.0;
            strength = 0.0;
            strengthChange = 0.0;
            targetActivity = 0.0;
        }
        // This one is called from grannyForm
        public Synapse(double synapseLearnRate, double synapseStrength)
        {
            id = 0;
            idSrceNeuron = 0;
            idTrgtNeuron = 0;
            learnRate = synapseLearnRate;
            inAxon = 0.0;
            outDendrite = 0.0;
            strength = synapseStrength;
            strengthChange = 0.0;
        }
        //
        //             Class methods
        //
        // Not learning, McLeod(18-19). Since learning is is represented by the Hebb or
        // Delta rule, I assume that the during the non-learning phase no synapse
        // strengthening takes place. Thus the output from the synapse is constant
        // for a given input, changing only in response to changed input and the previously
        // learned strength.
        // I wonder about the physiological validity of simply turning learning on and off.
        // I suspect it is not quite so simple in the brain.
        //
        public void NotLearning()
        {
            outDendrite = inAxon * strength;
        }
        // Hebb Rule, McLeod (54). The synapse is strengthened on each iteration in
        // accordance with the strengthChange property, which is constant for each run.
        // StrengthChange is set by the Network method SetHebbRuleStrengthChange() and is
        // described in more detail there. Here, on each iteration, the strength is
        // incremented by the strength change, multiplied by the inAxon value and passed
        // to the outDendrite.
        //
        public void HebbRule()
        {
            strength += strengthChange;
            outDendrite = inAxon * strength;
        }
        // Delta Rule (McLeod: 19 expression 1.3) The required strength change is
        // calculated here on each iteration in contrast with the Hebb Rule.
        // The change in the synapse strength is the desired (target) activity minus the
        // obtained activity, multiplied by the source activity (input) and the learning rate.
        // This method is called from Network.SetDeltaStrengthChange() which determines
        // the targetValue, obtainedValue and sourceValue for each synapse. The actual
        // synapse processing takes place here.
        //
        public void DeltaRule()
        {
            double sourceActivity = inAxon;
            strengthChange = (targetActivity - obtainedActivity) * sourceActivity * learnRate;
            strength += strengthChange;
            outDendrite = inAxon * strength;
        }
    }
}
```

Appendix B: The network class

The documented code for the Network classes is listed on the following pages. The code was written in C# using the Microsoft Visual C# 2008 Express Edition.

B.1 Network class code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ANNie
{
    class Network
        // Creates and runs a network of neurons and synapses. There is a case to split
        // this class since creation is anatomy and the run functions are physiology.
        // At creation time the first set of properties, before the comment marker,
        // are passed directly from the MakeANNie Form to the Initialsie() method.
        // The other properties are calculated as indicated by Network. methods as
        // indicated.
        //
        // The four arrays have to be defined as 'indexers' rather than properties in C#
        // and the Get/Set methods require the use of the 'this' keyword. Two indexers of
        // the same value type are not permitted so I have used int/short and single/double
        // to overcome this. I have checked the passing of values to and from these arrays
        // and can find no ill effects.
    {
        protected int _id;
        protected string _name;
        protected int _numNeurons;
        protected int _numLayers;
        protected int _numInputNeurons;
        protected int _numOutputNeurons;
        protected double _threshold;
        protected double _rho;  // used to adjust the slope of the sigmoid
        //
        protected int _numSynapses; // See SetSynapses()
        protected int _activationFunction; // See SetActivationFunction()
        protected int _learnFunction; // See SetLearnFunction()
        protected internal int[] numNeuronsInThisLayer = new int[] { }; // See
SetLayersAndPointers()
        protected internal short[] firstNeuronInThisLayer = new short[] { }; // See
SetLayersAndPointers()
        protected internal Single[] arrInputValues = new Single[] { }; // See
SetInputAndTargetArrays()
        protected internal double[] arrTargetValues = new double[] { }; // See
SetInputAndTargetArrays()
        //
        //                  Get Set methods
        //
        public int id
        {
            get { return _id; }
            set { _id = value; }
        }
        public string name
        {
            get { return _name; }
            set { _name = value; }
        }
        public int numNeurons
        {
            get { return _numNeurons; }
            set { _numNeurons = value; }
        }
        public int numSynapses
        {
            get { return _numSynapses; }
            set { _numSynapses = value; }
        }
        public int numLayers
        {
            get { return _numLayers; }
            set { _numLayers = value; }
        }
        public int numInputNeurons
        {
            get { return _numInputNeurons; }
            set { _numInputNeurons = value; }
        }
        public int numOutputNeurons
        {
            get { return _numOutputNeurons; }
            set { _numOutputNeurons = value; }
        }
        public double threshold
        {
            get { return _threshold; }
            set { _threshold = value; }
        }
        public double rho
        {
            get { return _rho; }
            set { _rho = value; }
        }
        public int activationFunction
        {
            get { return _activationFunction; }
            set { _activationFunction = value; }
        }
        public int learnFunction
        {
            get { return _learnFunction; }
            set { _learnFunction = value; }
        }
        public int this[int indexNum]
        {
            get { return numNeuronsInThisLayer[indexNum]; }
            set { numNeuronsInThisLayer[indexNum] = value; }
        }
        public short this[short indexFirst]
        {
            get { return firstNeuronInThisLayer[indexFirst]; }
            set { firstNeuronInThisLayer[indexFirst] = value; }
        }
        public Single this[Single indexInp]
        {
            get { return arrInputValues[Convert.ToInt16(indexInp)]; }
            set { arrInputValues[Convert.ToInt16(indexInp)] =value; }
        }
        public double this[double indexTrgt]
        {
            get { return arrTargetValues [Convert.ToInt16(indexTrgt)]; }
            set { arrTargetValues[Convert.ToInt16(indexTrgt)] = value; }
        }
        //
        //                  Constructor
        // This doesn't do much since the Network object needs to be instantiated
        // at the asame time as MakeANNie form in order to expose it to the MakeANNie form
        // methods. See Initialise().
        public Network(int netId)
        {
            id = netId;
        }
        //
        //                  Class methods
        //              1. Make Network methods
        //
        // Initialise Network object. Called from MakeANNieForm.button1_Click
        // label: "Make ANNie". Arrays containing layer values are initialsed by the
        // the two sub methods.
        //
        public void Initialise(string netName, int netNumNeurons, int netNumLayers, int
netNumInputNeurons,           int netNumOutputNeurons)
        {
            name = netName;
            numNeurons = netNumNeurons;
            numSynapses = 0;
            numLayers = netNumLayers;
            numInputNeurons = netNumInputNeurons;
            numOutputNeurons = netNumOutputNeurons;
            InitNumNeuronsInThisLayer();    // See below
            InitFirstNeuronInThisLayer();
        }
        //
        // Set the size of the arrays numNeuronsIn this layer[] and firstNeuronInThisLayer[]
        // They are used when the network is run, to allow iteration through a layer.
        // Values are assigned to the arrays in SetLayersAndPointers
        public void InitNumNeuronsInThisLayer()
        {
            Array.Resize(ref numNeuronsInThisLayer, numLayers);
            for (int i=0; i<numNeuronsInThisLayer.Length; i++)
                numNeuronsInThisLayer[i] = 0;
        }
        public void InitFirstNeuronInThisLayer()
        {
            Array.Resize(ref firstNeuronInThisLayer, numLayers);
            for (int i=0; i<firstNeuronInThisLayer.Length; i++)
                firstNeuronInThisLayer[i] = 0;
        }
        // Assign values to numNeuronsInthisLayer[] and firstNeuronInThisLayer[]
        //
        public void SetLayersAndPointers()
        {
            // Input layer
            firstNeuronInThisLayer[0] = 0;
            numNeuronsInThisLayer[0] = numInputNeurons;
            // Output layer
            firstNeuronInThisLayer[numLayers - 1] = Convert.ToInt16(numNeurons - numOutputNeurons);
            numNeuronsInThisLayer[numLayers - 1] = numOutputNeurons;
            // Hidden layers, if there are any
            if (numLayers > 2)
            {
                // calculate number of neurons per hidden layer.
                int numNeuronsPerHiddenLayer =
                    (numNeurons - (numInputNeurons + numOutputNeurons)) / (numLayers - 2);
                // Assign number of neurons per layer and first neuron in layer
                for (int i = 1; i < numLayers - 1; i++)
                {       // each layer has the same number of neurons
                    numNeuronsInThisLayer[i] = numNeuronsPerHiddenLayer;
                    firstNeuronInThisLayer[i] = Convert.ToInt16(firstNeuronInThisLayer[i - 1] +
                                    numNeuronsPerHiddenLayer - 1);
                }
            }
        }

        // Create neuron objects. Set neuron id, layer number, number of dendrites
        // and threshold.
        public void SetNeurons(ref Neuron[] arrNeuron)
        {
            // Initialise neurons with id, layer, number of dendrites and threshold
            //
            int layer = 0;  // set to input layer
            int numDendrites = 1; // neurons in the input layer have one dendrite
            // Loop to instantiate neuron objects and initialise
            for (int i = 0; i < numNeurons; i++)
            {
                // Increment the layer but don't overshoot array bound
                // This seems a bit obscure. Simplify?
                if (layer < numLayers - 1 && i == firstNeuronInThisLayer[layer + 1])  //increment
if next
//layer
                {
                    numDendrites = numNeuronsInThisLayer[layer]; // ready for next layer
                    layer++;
                }
                // Create neuron object, numDendrites is the number of neurons
                // in the preceding layer
                arrNeuron[i] = new Neuron(i, layer, numDendrites);
            }
        }
        // Calculate the number of synapses required and then resize the array
        // created in the calling form. Instantiate the synapse objects and assign
        // an id number to each.
        public void SetSynapses(ref Synapse[] arrSynapse)
        {
            // Calculate total number of synapses: for each layer, the number of neurons in
            // this layer * the number of neurons in the next layer
            for (int i = 0; i < numLayers - 1; i++)
                numSynapses += numNeuronsInThisLayer[i] * numNeuronsInThisLayer[i + 1];
            Array.Resize(ref arrSynapse, numSynapses);
            // Instantiate synapse objects and assign id
            for (int i = 0; i < numSynapses; i++)
            {
                arrSynapse[i] = new Synapse();
                arrSynapse[i].id = i;
            }
        }
        // Assign the source and target neurons to each synapse
        // Maybe break this up
        public void SetConnections(ref Synapse[] arrSynapse)
        {
            int thisSynapse = 0; // holds the current synapse
            // iterate by layer
            for (int i = 1; i < numLayers; i++)
            {     // Set start and end neurons for target layer
                int startNeuronTrgt = firstNeuronInThisLayer[i];
                int endNeuronTrgt = firstNeuronInThisLayer[i] + numNeuronsInThisLayer[i];
                // iterate for neurons in target layer
                for (int j = startNeuronTrgt; j < endNeuronTrgt; j++)
                {
                    // Set start and end neurons for source layer
                    int startNeuronSrc = firstNeuronInThisLayer[i - 1];
                    int endNeuronSrc = firstNeuronInThisLayer[i - 1] + numNeuronsInThisLayer[i -
1];
                    // Iterate for neurons in source layer
                    for (int k = startNeuronSrc; k < endNeuronSrc ; k++)
                    {
                        arrSynapse[thisSynapse].idSrceNeuron = k;
                        arrSynapse[thisSynapse++].idTrgtNeuron = j; // iterate synapse for next
iteration
                    }
                }
            }
        }
        // -------------------   2. Run Network methods    ------------------------
        //
        // Set all neurons to the value passed from user input. Both McLeod (20)
        // and Gurney (40) employ a bias node which dynamically changes the threhold as
        // if it were a synaptic weight (strength). This seems to be physiologically
        // implausible since the neuron is said to depolarise and thus generate an
        // action potential, at around -50mV (Ward 20). I can find no reference to such
        // behaviour in standard biology texts (eg Curtis & Barnes 1986)
        //
        public void SetNeuronThresholds(double netThreshold, ref Neuron[] arrNeuron)
        {
            for (int i = 0; i < numNeurons; i++)
                arrNeuron[i].threshold = netThreshold;
        }
        // Set rho, the slope of the sigmoid, for each neuron
        public void SetSigmoidRho(double netRho, ref Neuron[] arrNeuron)
        {
            for (int i = 0; i < numNeurons; i++)
                arrNeuron[i].rho = netRho;
        }

        // Set the initial strength and learn rate for all synapses. Passed from user
        // input.
        public void SetSynapseStrengthAndLearnRate(double netStartStrength, double netLearnRate,
                      ref Synapse[] arrSynapse)
        {
            for (int i=0; i < numSynapses; i++)
            {
                arrSynapse[i].strength = netStartStrength;
                arrSynapse[i].learnRate = netLearnRate;
            }
        }
        // The user input for the activation and learn functions gives is a set of
        // boolean values. They are converted to two integer variables for convenience.
        public void SetActivationFunction(bool binary, bool sigmoid)
        {
            if (binary)
                activationFunction = 0;
            if(sigmoid)
                activationFunction = 1;
        }
        public void SetLearnFunction(bool notLearning, bool hebb, bool delta)
        {
            if (notLearning)
                learnFunction = 0;
            if (hebb)
```

```
                learnFunction = 1;
            if (delta)
                learnFunction = 2;
        }
        // The input and target values are received from the user as strings.
        // They are converted and assigned to the numeric arrays arrInputValues[] and
        // arrTargetValues[]. The arrays are used by the two learn functions but not when
        // there is no learning (see SetInputDendrites()).
        //
        // numValuesInOutArray is set to numNeurons since I wish to align its subscripts
        // to those of the output layer, for convenience. This means that the subscripts
        // < startValueOutArray are unused. I wanted to null them but C# won't have it.
        //
        public void SetInputAndTargetArrays(string[] inputValues, string[] targetValues)
        {
            if (learnFunction == 1 || learnFunction == 2) // Is Hebb or Delta
            {
                int numValuesInOutArray = numNeurons;
                //int numValuesInOutArray = firstNeuronInThisLayer[numLayers - 1]
                //    + numNeuronsInThisLayer[numLayers - 1];
                int startValueOutArray = firstNeuronInThisLayer[numLayers - 1];
                Array.Resize(ref arrInputValues, numInputNeurons);
                Array.Resize(ref arrTargetValues, numValuesInOutArray);
                // First set all Target values to -999
                for (int i = 0; i < arrTargetValues.Length; i++)
                    arrTargetValues[i] = -999.0;        // Can't null a double!
                // Set the user input and output string values to the two arrays
                for (int i = 0; i < inputValues.Length; i++)
                    arrInputValues[i] = Convert.ToSingle(inputValues[i]) / 100;
                for (int i = 0; i < targetValues.Length; i++)
                    arrTargetValues[i + startValueOutArray] = Convert.ToDouble(targetValues[i]) /
100;
            }
        }
        //
        // SetHebbRuleStrengthChange() -- Implements Hebb Rule:
        // strengthChange = learnRate * targetOutputValue * inputValue
        // McLeod (54) expression 3.1. Since the values in the expression do not change
        // throughout the run this methods is executed once only per run.
        public void SetHebbRuleStrengthChange(ref Synapse[] arrSynapse)
        {
            // First neuron in output layer
            int startValueTargetArray = firstNeuronInThisLayer[numLayers - 1];
            // Iterate synapses
            for (int currSynapse = 0; currSynapse < numSynapses; currSynapse++)
            {
                // Iterate input array
                for (int j = 0; j < arrInputValues.Length; j++)
                {
                    // Iterate output array
                    for (int i = startValueTargetArray; i < arrTargetValues.Length; i++)
                    {
                        if (arrSynapse[currSynapse].idSrceNeuron == j
                            && arrSynapse[currSynapse].idTrgtNeuron == i)
                        {
                            // The strength change is the input value at neuron[j]
                            // * output(target) value at neuron[i] * learnRate
                            arrSynapse[currSynapse].strengthChange = arrInputValues[j]
                                * arrTargetValues[i] * arrSynapse[currSynapse].learnRate;
                        }
                    }
                }
            }
        }
        //
        // SetDeltaStrengthChange() -- Implements Delta Rule Mcleod (19) expr 1.3
        // This method is run at the end of each iteration to capture the obtained values.
        // For each synapse change the strength by subtracting the obtained activation
        // from the desired (target) activation: a[i](target) - a[i](obtained)
        // multply the result by the input (source) activation and the learnRate.
        // Obtained activation is the set of values at the outAxon of
        // the output layer. When the target Synapse is identified the values are passed
        // to the Synapse properties targetActivity and obtainedActivity, and applied
        // by the ApplyDelta() method on the next iteration.
        public void SetDeltaStrengthChange(ref Synapse[] arrSynapse, ref Neuron[] arrNeuron)
        {
            // First neuron in output layer
            int startValueTargetArray = firstNeuronInThisLayer[numLayers - 1];
            // Iterate synapses
            for (int currSynapse = 0; currSynapse < numSynapses; currSynapse++)
            {
                // Iterate input array
                for (int j = 0; j < arrInputValues.Length; j++)
                {
                    // Iterate output array
                    for (int i = startValueTargetArray; i < arrTargetValues.Length; i++)
                    {
                        // currSynapse connects neuron[j] to neuron[i]
                        int srceNeuron = arrSynapse[currSynapse].idSrceNeuron;
                        int trgtNeuron = arrSynapse[currSynapse].idTrgtNeuron;
                        if (srceNeuron == j && trgtNeuron == i)
                        {
                            // Set the target, obtained and source values
                            double targetValue = arrTargetValues[i];
                            // Actual output of neuron[i]
                            double obtainedValue = arrNeuron[trgtNeuron].outAxon;
                            // Output from source neuron[j]
                            //double sourceValue = arrNeuron[srceNeuron].outAxon;
                            // Set target and obtained values for use in Delta Rule
                            // sourceValue is neuron outAxon value
                            //
                            // .. ! this once needs to be set once for each input word
                            arrSynapse[currSynapse].targetActivity = targetValue;
                            arrSynapse[currSynapse].obtainedActivity = obtainedValue;

                            //arrSynapse[currSynapse].DeltaRule(targetValue, obtainedValue,
sourceValue);
                        }
                    }
                }
            }

        }
        // Set user input values array to input layer dendrites
        // Value is /100 because sigmoid function generates values >0<1
        public void SetInputDendrites(string[] inputValuesArray, ref Neuron[] arrNeuron)
        {
            for (int i = 0; i < inputValuesArray.Length ; i++)
                arrNeuron[i].inDendrite[0] = Convert.ToDouble(inputValuesArray[i]) / 100;
        }
        // Set netInput for each neuron in current layer
        public void SetNetInputs(int thisLayer, ref Neuron[] arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            for (int i = startNeuron; i < endNeuron; i++)
                    arrNeuron[i].SetNetInput();
        }
        // Apply Binary function to each neuron in current layer
        public void ApplyBinary(int thisLayer, ref Neuron[] arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            for (int i = startNeuron; i < endNeuron; i++)
                arrNeuron[i].BinaryFunction();
        }

        // Apply Sigmoid function to each neuron in current layer
        public void ApplySigmoid(int thisLayer, ref Neuron[] arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            for (int i = startNeuron; i < endNeuron; i++)
                arrNeuron[i].SigmoidFunction();
        }

        // Apply Not Learning function to each source synapse in current layer
        // Get value from source neuron outAxon, set to source neuron synapse inAxon
        // Apply not learning function
        public void ApplyNotLearning(int thisLayer, ref Synapse[] arrSynapse, ref Neuron[]
arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            // Iterate neurons in current layer
            for (int i = startNeuron; i < endNeuron; i++)
            {
                // Iterate synapses
                for (int j = 0; j < numSynapses; j++)
                    // if this synapse is connected to neuron i
                    if (arrSynapse[j].idSrceNeuron == i)
                    {
                        // pass the value from the neuron to the synapse
                        // apply not learning function
                        arrSynapse[j].inAxon = arrNeuron[i].outAxon;
                        arrSynapse[j].NotLearning();
                    }
            }
        }

        // For each neuron in this layer set synapse input axon
        // and apply Hebb Rule
        public void ApplyHebb(int thisLayer, ref Synapse[] arrSynapse, ref Neuron[] arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            for (int i = startNeuron; i < endNeuron; i++)
            {
                for (int j = 0; j < numSynapses; j++)
                    if (arrSynapse[j].idSrceNeuron == i) // if this synapse is connected to neuron
i
                    {
                        arrSynapse[j].inAxon = arrNeuron[i].outAxon; // pass the value from the
neuron to the synapse
                        arrSynapse[j].HebbRule();
                    }
            }
        }
        // ApplyDelta() -- Implements Delta Rule Mcleod (19) expr 1.3
        // The target and obtained activity required for the Delta Rule are set by
        // SetDeltaStrengthChange() method at the end of each iteration.
        // This method could be merged with ApplyHebb()
        public void ApplyDelta(int thisLayer, ref Synapse[] arrSynapse, ref Neuron[] arrNeuron)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            for (int i = startNeuron; i < endNeuron; i++)
            {
                for (int j = 0; j < numSynapses; j++)
                    if (arrSynapse[j].idSrceNeuron == i) // if this synapse is connected to neuron
i
                    {
                        arrSynapse[j].inAxon = arrNeuron[i].outAxon; // pass the value from the
neuron to the synapse
                        arrSynapse[j].DeltaRule();
                    }
            }
        }

        // Pass Synapse dendrite outputs to target Neuron inputs
        public void SetNeuronDendrites(int thisLayer, ref Neuron[] arrNeuron, ref Synapse[]
arrSynapse)
        {
            int startNeuron = firstNeuronInThisLayer[thisLayer];
            int endNeuron = firstNeuronInThisLayer[thisLayer] + numNeuronsInThisLayer[thisLayer];
            int targetDendrite = 0;
            for (int i = startNeuron; i < endNeuron; i++)
            {
                for (int j = 0; j < numSynapses; j++)
                {
                    if (arrSynapse[j].idTrgtNeuron == i)
                    {
                        // targetDendrite is the location of the source neuron
                        // minus the value of the first neuron in this layer
                        targetDendrite = arrSynapse[j].idSrceNeuron -
firstNeuronInThisLayer[thisLayer - 1];
                        arrNeuron[i].inDendrite[targetDendrite] = arrSynapse[j].outDendrite;
                    }
                }
            }
        }
    }
}
```

B.2 The MakeANNie form handler code

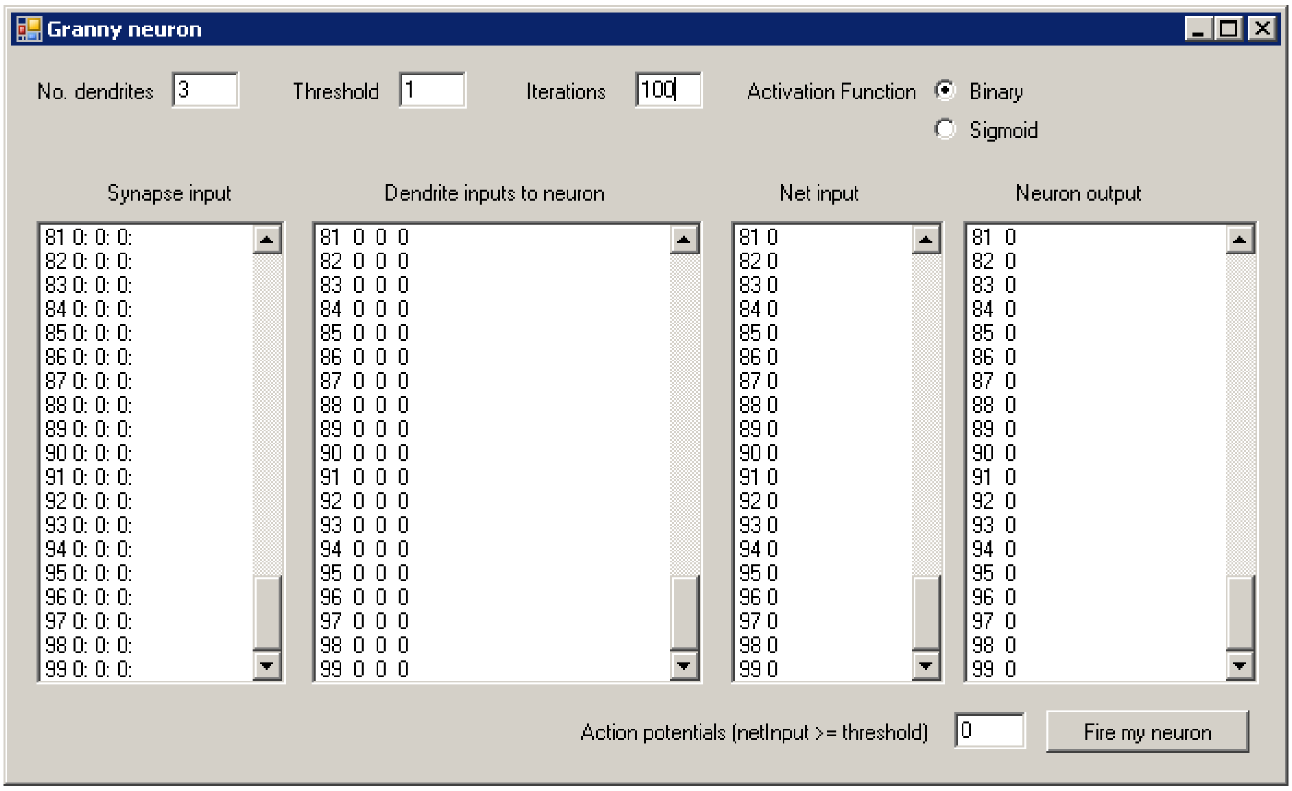
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace ANNie
{
    public partial class MakeANNieForm : Form
        // Create a network of neurons and synapses, connect them and run network
    {
        // Create here and resize and initialise later in order to expose to other methods
        Network aNetwork = new Network(0);
        Neuron[] aNeuron = new Neuron[0];
        Synapse[] aSynapse = new Synapse[0];
        string outFile = @".\OutValues.txt"; // used to pass values to outputForm
        //
        public MakeANNieForm()
        {
            InitializeComponent();
        }
        //
        //                  Make Network
        //
        private void button1_Click(object sender, EventArgs e)
        {
            string netName = textBox6.Text;
            int numNeurons = Convert.ToInt16(textBox1.Text);
            int numLayers = Convert.ToInt16(textBox2.Text);
            int numInputNeurons = Convert.ToInt16(textBox4.Text);
            int numOutputNeurons = Convert.ToInt16(textBox5.Text);
            //
            listBox1.Items.Clear();
            listBox2.Items.Clear();
            label7.Text = "";
            //
            // Do some initial checking here to make sure the numbers work
            // e.g. In a two layer network the input and output neurons must equal
            // the total number of neurons. Hidden layers must be of equal size.
            //
            if (numNeurons <= 0 || numLayers <= 0 || numInputNeurons <= 0 || numOutputNeurons <= 0)
                label7.Text = "Values must be positive";
            if (numLayers <= 1)
                label7.Text = "Minimum 2 layers";
            if (numInputNeurons + numOutputNeurons > numNeurons)
                label7.Text = "Not enough neurons!";
            else
            {
                //
                //  Network
                //
                // Initialise main values, constructor was run above to expose object
                // in order to make network object public
                aNetwork.Initialise(netName, numNeurons, numLayers, numInputNeurons,
numOutputNeurons);
                // Calulate number of layers and neurons required from user spec
                // and assign to network object
                aNetwork.SetLayersAndPointers();
                label7.Text = "Network layers and pointers created";
                //
                // Neurons
                //
                // Resize neuron array. aNeuron was created on form opening above
                // to make array object public
                Array.Resize(ref aNeuron, numNeurons);
                aNetwork.SetNeurons(ref aNeuron);  // Create neuron objects and set number
dendrites
                label7.Text = "Neurons and dendrites created";
                //
                // Synapses
                //
                // Calcualte total number of synapses.
                // In each layer the number of synapses is equal to the number of neurons
                // multiplied by the number of neurons in the prevous layer
                aNetwork.SetSynapses(ref aSynapse);
                label7.Text = "Synapses created.";
                // Make the connections by setting the appropriate source and target
                // neuron ids in each synapse.
                aNetwork.SetConnections(ref aSynapse);
                label7.Text = "Network created \r\nCheck connections";
                groupBox2.Enabled = true;
                // Output neuron and synapse ids to form
                for (int i = 0; i < numNeurons; i++)
                    listBox1.Items.Add(Convert.ToString(aNeuron[i].layer) + "  " +
                        Convert.ToString(aNeuron[i].id));
                for (int i = 0; i < aSynapse.Length; i++)
                    listBox2.Items.Add(Convert.ToString(aSynapse[i].id) + "  " +
                        Convert.ToString(aSynapse[i].idSrceNeuron) + "  " +
                        Convert.ToString(aSynapse[i].idTrgtNeuron));
            }
        }
        //
        // -------------------    Run Network     -------------------------
        //
        // Get run values from user and process
        //
        private void button3_Click(object sender, EventArgs e)
        {
            double threshold = Convert.ToDouble(textBox11.Text);
            double startStrength = Convert.ToDouble(textBox10.Text);
            double learnRate = Convert.ToDouble(textBox9.Text);
            int numIterations = Convert.ToInt16(textBox8.Text);
            double rho = Convert.ToDouble(textBox13.Text); // the slope of the sigmoid
            bool isBinary = radioButton1.Checked;
            bool isSigmoid = radioButton2.Checked;
            bool isHebbLearn = radioButton3.Checked;
            bool isDeltaLearn = radioButton4.Checked;
            bool isNotLearning = radioButton8.Checked;
            bool isPresentAfterLearning = checkBox1.Checked;
            //bool isOutputFinal = radioButton5.Checked;
            //bool isOutputInter = radioButton6.Checked;
            //bool isOutputAll = radioButton7.Checked;
            string strInputValues = textBox7.Text; // IPA code
            string strInputAll = textBox7.Text.Trim();  // used for mutiple word input
            string strTargetAll = textBox3.Text.Trim();
            //
            // method here to get each input value for presentation to the network
            // Need to modify the classes and form code as little as possible
            // csv between values, spaces inside values. Need to distinguish between
            // phoneme: a single number, and word: a collection of phonemes.
            //
            // strInputAll contains the entire input as a string
            // wrdInputCollection is a collection of words
            string[] wrdInputCollection = new string[] { }; // new an array of input words
            string[] wrdTargetCollection = new string[] { }; // new
            wrdInputCollection = strInputAll.Trim().Split(','); // new assign input words to
collection
            int numInputWords = wrdInputCollection.Length; //
            wrdTargetCollection = strTargetAll.Trim().Split(','); // new assign target words to
collection

            // Do some exception checking, make sure arrays are the same size
            // and what not

            //
            TidyUp(numIterations);  // Pre-run housekeping
            // Pre-iteration processing of network run values
            SetNeuronsAndSynapses(threshold, startStrength, learnRate, rho); // Set start values
for all                                                                                    // neurons and
synapses
            // Set boolean values to numeric and assign to class properties
            SetActivationAndLearnFunctions(isBinary, isSigmoid, isNotLearning, isHebbLearn,
isDeltaLearn);
            //
            //       Loop for each iteration
            //
            for (int currIteration = 0; currIteration < numIterations; currIteration++)
            {
                if (progressBar1.Value >= progressBar1.Maximum)
                    progressBar1.Value = 0;
                else
                    progressBar1.Value += 1;
                File.AppendAllText(outFile, "Iteration " + currIteration + " ----------------------
\r\n");
                textBox12.Text = textBox12.Text + "Iteration " + currIteration + "-------------
\r\n";
                label10.Text = Convert.ToString(currIteration);
                this.Update();
                //
                //          loop for each word
                //
                for (int currInputWord = 0; currInputWord < numInputWords; currInputWord++)
                {
                    // Set next input and target words from word collections
                    String[] strArrInputValues = wrdInputCollection[currInputWord].Trim().Split('
'); // mod
                    String[] strArrTargetValues = wrdTargetCollection[currInputWord].Trim().Split('
');
                  // Initiaise arrays containing input and target values for use in Hebb and Delta
functions
                    aNetwork.SetInputAndTargetArrays(strArrInputValues, strArrTargetValues);
                    // Hebb strength change is fixed for the run and applied on each iteration
                    if (isHebbLearn)
                        aNetwork.SetHebbRuleStrengthChange(ref aSynapse);
                    // Output current word to textbox and file
                    textBox12.Text = textBox12.Text + "Word " + currInputWord + " "
                        + wrdInputCollection[currInputWord] + "\r\n";
                    File.AppendAllText(outFile, "Word " + currInputWord + " "
                        + wrdInputCollection[currInputWord] + "\r\n");
                        // Set input values from this Form to input layer
                        aNetwork.SetInputDendrites(strArrInputValues, ref aNeuron);
                        // process each layer
                        for (int currLayer = 0; currLayer < aNetwork.numLayers; currLayer++)
                            ProcessLayer(currLayer);
                        // Final processing of output layer
                        aNetwork.SetNetInputs(aNetwork.numLayers - 1, ref aNeuron);
                        aNetwork.ApplySigmoid(aNetwork.numLayers - 1, ref aNeuron);
                        // Display values to this Form and output to file
                        OutputToFormAndFile(currIteration);

                        // values for strength change
                        if (isDeltaLearn)
                            aNetwork.SetDeltaStrengthChange(ref aSynapse, ref aNeuron);
                    }
                // display last values to textBox12
                this.Update(); // repaint form
            }
            //
            //              Present input vlaues without learning
            //
            if (isPresentAfterLearning)
            {
                // strength changes have no effect when no learning but to make sure
                for (int i = 0; i < aNetwork.numSynapses; i++)
                    aSynapse[i].strengthChange = 0.0;
                textBox12.Text = "";
                // Present each word to the network
                for (int thisWord = 0; thisWord < numInputWords; thisWord++)
                {
                    File.AppendAllText(outFile, "Presentation without learning\r\n");
                    string[] arrInputValues = wrdInputCollection[thisWord].Trim().Split(' '); //
mod
                    aNetwork.learnFunction = 0;
                    aNetwork.SetInputDendrites(arrInputValues, ref aNeuron);
                    // process each layer
                    for (int currLayer = 0; currLayer < aNetwork.numLayers; currLayer++)
                        ProcessLayer(currLayer);
                    // Final processing of output layer
                    aNetwork.SetNetInputs(aNetwork.numLayers - 1, ref aNeuron);
                    aNetwork.ApplySigmoid(aNetwork.numLayers - 1, ref aNeuron);
                    // Display values to this Form and output to file
                    OutputToFormAndFile(0);
                }
            }
            label10.Text = ("Processing competed");
        }
        // -------------------- Form Methods ------------------------
        //
        // Pre-run housekeeping
        private void TidyUp(int iterations)
        {
            progressBar1.Maximum = iterations;
            progressBar1.Value = 0;
            textBox12.Text = ""; // output box
            label10.Text = "";
            button2.Enabled = false;
            this.Update();  // repaint form
            if (File.Exists(outFile))
                File.Delete(outFile);    // delete previous output file - but it comes back!
        }
        // Set start values for all neurons and synapses
        private void SetNeuronsAndSynapses(double netThreshold, double netStartStrength,
               double netLearnRate, double netRho)
        {
            aNetwork.SetNeuronThresholds(netThreshold, ref aNeuron);
            aNetwork.SetSynapseStrengthAndLearnRate(netStartStrength, netLearnRate, ref aSynapse);
            aNetwork.SetSigmoidRho(netRho, ref aNeuron);
        }
        // Set Network.activationFunction to 0, 1
        // Set Network.learnFunction to 0, 1, 2
        private void SetActivationAndLearnFunctions(bool binary, bool sigmoid, bool notLearn, bool
hebb,          bool delta)
        {
            aNetwork.SetActivationFunction(binary, sigmoid);
            aNetwork.SetLearnFunction(notLearn, hebb, delta);
        }
        // Process a layer. I haven't put this method in the Network class
        // as some pre and post processing is required
        public void ProcessLayer(int thisLayer)
        {
            // Summate dendrite inputs to Netinput
            aNetwork.SetNetInputs(thisLayer, ref aNeuron);
            // Neuron processing
            if (aNetwork.activationFunction == 0)
                aNetwork.ApplyBinary(thisLayer, ref aNeuron);
            if (aNetwork.activationFunction == 1)
                aNetwork.ApplySigmoid(thisLayer, ref aNeuron);
            // Synapse processing
            if (aNetwork.learnFunction == 0)
                aNetwork.ApplyNotLearning(thisLayer, ref aSynapse, ref aNeuron);
            if (aNetwork.learnFunction == 1)
                aNetwork.ApplyHebb(thisLayer, ref aSynapse, ref aNeuron);
            if (aNetwork.learnFunction == 2)
                aNetwork.ApplyDelta(thisLayer, ref aSynapse, ref aNeuron);

            // assign synapse outputs to next layer dendrite inputs
            // outAxon is multiplied by 100 since sigmoid generates values >0<1
            aNetwork.SetNeuronDendrites(thisLayer, ref aNeuron, ref aSynapse);
        }
        //
        public void OutputToFormAndFile(int thisIteration)
        {
            int startNeuron = 0;
            int endNeuron = 0;
            double outputValue = 0.0;
            double roundedOutputValue = 0.0;
            string outLine = "";
            string roundedOutLine = "";
            string outText = "";
            //if (File.Exists(outFile))
            //    File.Delete(outFile);    // tidyUp() does this but it mysteriously reappears
            //aNeuron[0].netInput
            for (int thisLayer = 0; thisLayer < aNetwork.numLayers; thisLayer++)
            {
                startNeuron = aNetwork.firstNeuronInThisLayer[thisLayer];
                endNeuron = startNeuron + aNetwork.numNeuronsInThisLayer[thisLayer];
                if (thisLayer == 0)
                {
```

```
                for (int i = 0; i < endNeuron; i++)
                    outLine += Convert.ToString(aNeuron[i].netInput) + " ";
                File.AppendAllText(outFile, "inWord: " + outLine + "\r\n");
                outLine = "";
            }
            for (int i = startNeuron; i < endNeuron; i++)
            {
                outputValue = aNeuron[i].outAxon * 100;
                roundedOutputValue = Math.Round(outputValue, 0);
                outLine += Convert.ToString(outputValue) + " ";
                roundedOutLine += Convert.ToString(roundedOutputValue) + " ";
            }
            outText = thisIteration + " " + thisLayer + ". " + outLine + "\r\n";
            File.AppendAllText(outFile, outText);

            if (thisLayer == aNetwork.numLayers -1)
                textBox12.Text += thisIteration + ". " + roundedOutLine + "\r\n";
            outLine = "";
            roundedOutLine = "";
            outText = "";
        }
        double sumStrengthChange = 0.0;
        for (int i = 0; i < aNetwork.numSynapses; i++)
        {
            roundedOutputValue = Math.Round(aSynapse[i].strength, 4);
            outLine = outLine + "(" + aSynapse[i].idSrceNeuron + "," +
            aSynapse[i].idTrgtNeuron + ")" +
            Convert.ToString(roundedOutputValue) + "  ";
            outLine = outLine + Math.Round(aSynapse[i].strengthChange, 4);
            sumStrengthChange += aSynapse[i].strengthChange;
        }
        outText = outText + outLine + "\r\n" + "Sum of strength changes = " +
            sumStrengthChange;
        File.AppendAllText(outFile, outText + "\r\n\r\n");
        button2.Enabled = true;
    }

    // Does nothing but the compiler protests if I delete it
    public void MakeANNieForm_Load(object sender, EventArgs e)
    {
       //Neuron[] testNeuron = new Neuron[0]; // create array of neurons
    }
    // Create output form and show output values from file
    private void button2_Click(object sender, EventArgs e)
    {
        ShowOutputForm showForm = new ShowOutputForm();
        showForm.Show();
    }
  }
}
```

Appendix C: Testing a single neuron

C1 All synapse inputs set to zero

Code: aSynapse[j].inAxon = 0.0;



No. dendrites: 3 Threshold: 1 Iterations: 100 Activation Function: [✓ Binary] Sigmoid

Synapse input
```
81 0: 0: 0:
82 0: 0: 0:
83 0: 0: 0:
84 0: 0: 0:
85 0: 0: 0:
86 0: 0: 0:
87 0: 0: 0:
88 0: 0: 0:
89 0: 0: 0:
90 0: 0: 0:
91 0: 0: 0:
92 0: 0: 0:
93 0: 0: 0:
94 0: 0: 0:
95 0: 0: 0:
96 0: 0: 0:
97 0: 0: 0:
98 0: 0: 0:
99 0: 0: 0:
```

Dendrite inputs to neuron
```
81  0  0  0
82  0  0  0
83  0  0  0
84  0  0  0
85  0  0  0
86  0  0  0
87  0  0  0
88  0  0  0
89  0  0  0
90  0  0  0
91  0  0  0
92  0  0  0
93  0  0  0
94  0  0  0
95  0  0  0
96  0  0  0
97  0  0  0
98  0  0  0
99  0  0  0
```

Net input
```
81 0
82 0
83 0
84 0
85 0
86 0
87 0
88 0
89 0
90 0
91 0
92 0
93 0
94 0
95 0
96 0
97 0
98 0
99 0
```

Neuron output
```
81  0
82  0
83  0
84  0
85  0
86  0
87  0
88  0
89  0
90  0
91  0
92  0
93  0
94  0
95  0
96  0
97  0
98  0
99  0
```

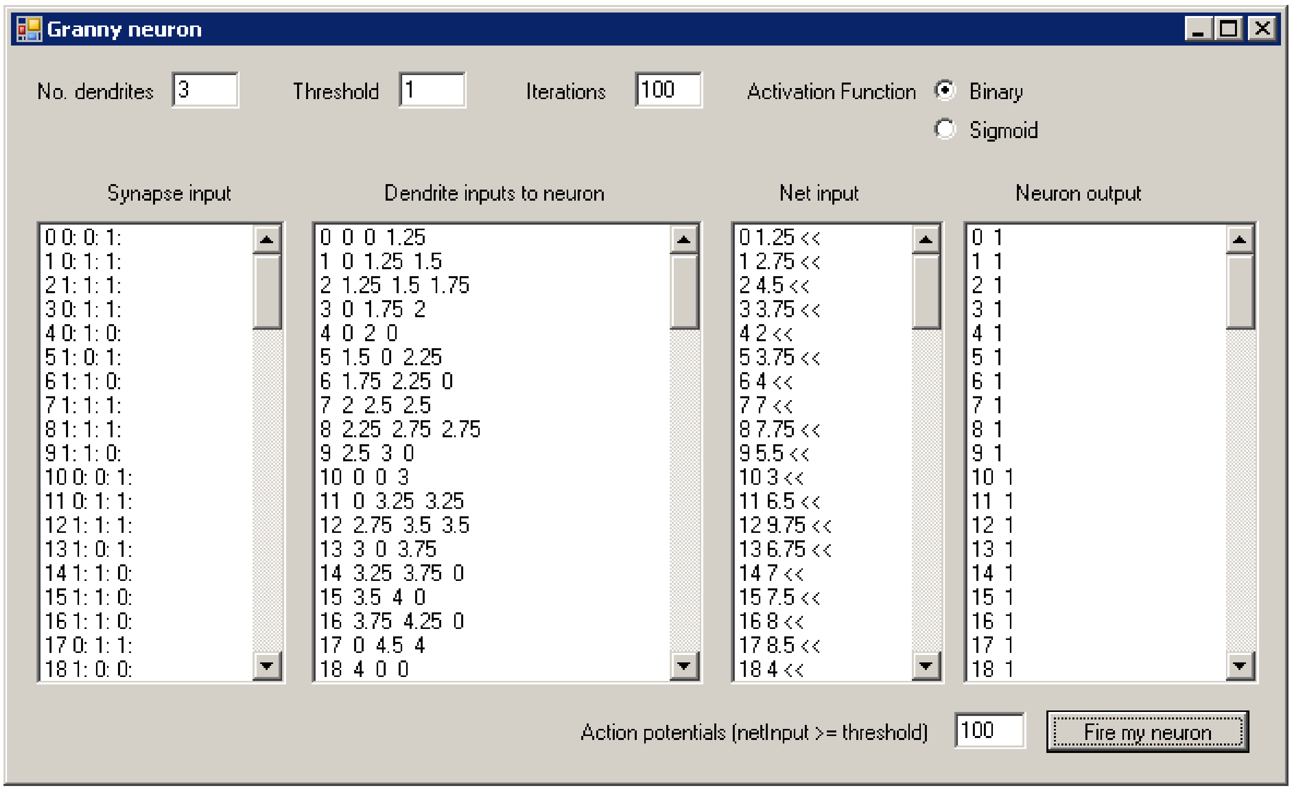
Action potentials (netInput >= threshold): 0

Comment: Output values are always zero.

C2 Total synapse values always >= threshold

Code:

```
aSynapse[j].inAxon = Convert.ToDouble(rnd.Next(0, 2));
if (synapseValues == "0: 0: 0: ")          // in case all three are zero
          aSynapse[j].inAxon = 1;
```



Granny neuron

No. dendrites 3 Threshold 1 Iterations 100 Activation Function [✓ Binary] Sigmoid

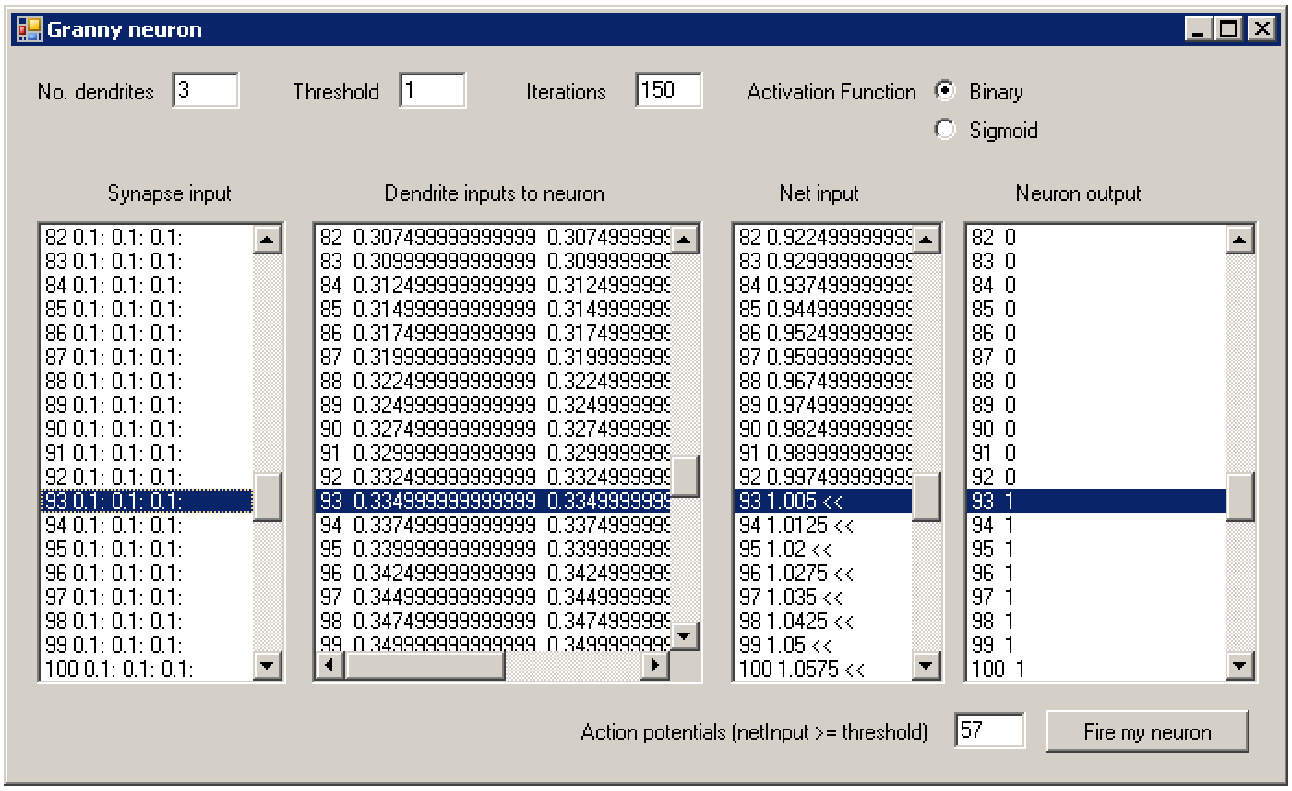
Synapse input	Dendrite inputs to neuron	Net input	Neuron output
0 0: 0: 1:	0 0 0 1.25	0 1.25 <<	0 1
1 0: 1: 1:	1 0 1.25 1.5	1 2.75 <<	1 1
2 1: 1: 1:	2 1.25 1.5 1.75	2 4.5 <<	2 1
3 0: 1: 1:	3 0 1.75 2	3 3.75 <<	3 1
4 0: 1: 0:	4 0 2 0	4 2 <<	4 1
5 1: 0: 1:	5 1.5 0 2.25	5 3.75 <<	5 1
6 1: 1: 0:	6 1.75 2.25 0	6 4 <<	6 1
7 1: 1: 1:	7 2 2.5 2.5	7 7 <<	7 1
8 1: 1: 1:	8 2.25 2.75 2.75	8 7.75 <<	8 1
9 1: 1: 0:	9 2.5 3 0	9 5.5 <<	9 1
10 0: 0: 1:	10 0 0 3	10 3 <<	10 1
11 0: 1: 1:	11 0 3.25 3.25	11 6.5 <<	11 1
12 1: 1: 1:	12 2.75 3.5 3.5	12 9.75 <<	12 1
13 1: 0: 1:	13 3 0 3.75	13 6.75 <<	13 1
14 1: 1: 0:	14 3.25 3.75 0	14 7 <<	14 1
15 1: 1: 0:	15 3.5 4 0	15 7.5 <<	15 1
16 1: 1: 0:	16 3.75 4.25 0	16 8 <<	16 1
17 0: 1: 1:	17 0 4.5 4	17 8.5 <<	17 1
18 1: 0: 0:	18 4 0 0	18 4 <<	18 1

Action potentials (netInput >= threshold) 100

Comment: Always generates an action potential. Synaptic strengthening has no effect.

C3 Synapse values set to 0.1

Code:
aSynapse[j].inAxon = 0.1;



Granny neuron

No. dendrites 3 Threshold 1 Iterations 150 Activation Function [✓ Binary] Sigmoid

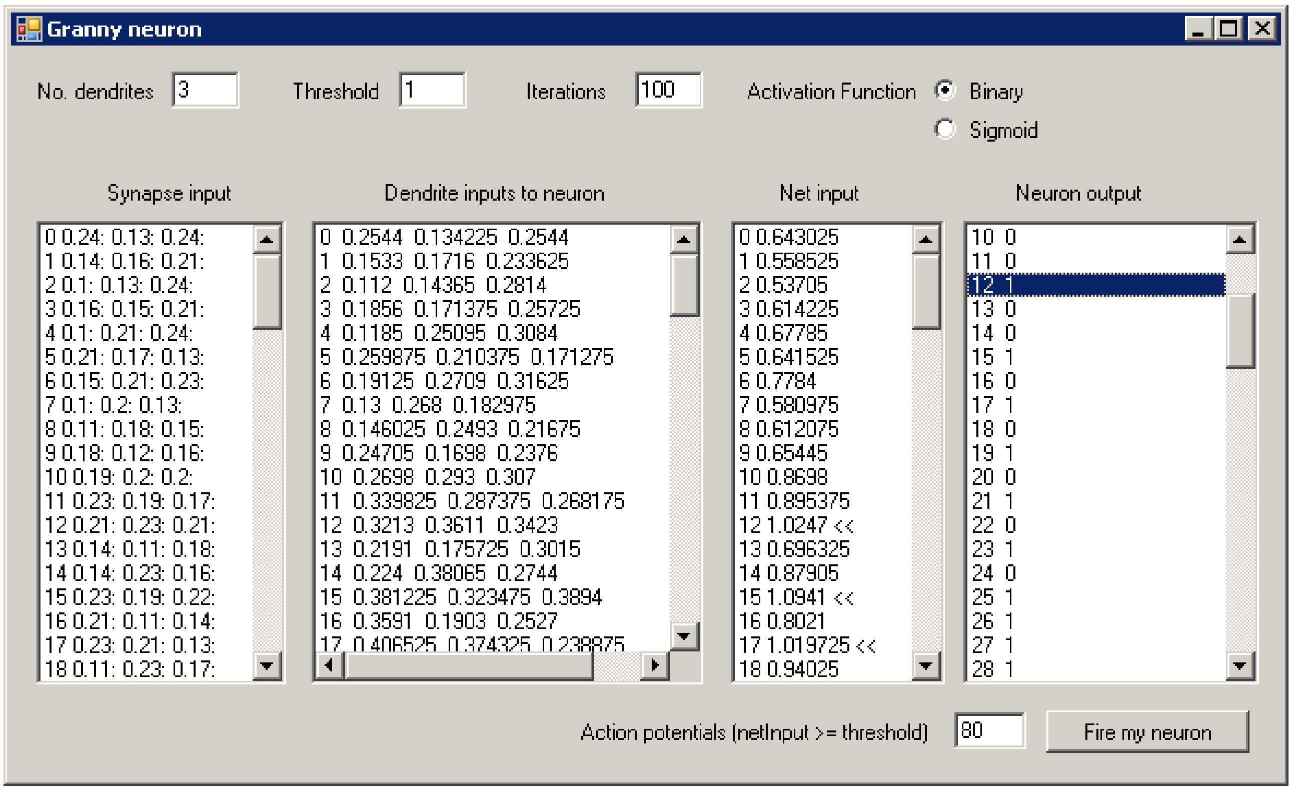
```
Synapse input        Dendrite inputs to neuron                 Net input            Neuron output
82 0.1: 0.1: 0.1:    82  0.307499999999999  0.30749999         82 0.9224999999      82  0
83 0.1: 0.1: 0.1:    83  0.309999999999999  0.30999999         83 0.9299999999      83  0
84 0.1: 0.1: 0.1:    84  0.312499999999999  0.31249999         84 0.9374999999      84  0
85 0.1: 0.1: 0.1:    85  0.314999999999999  0.31499999         85 0.9449999999      85  0
86 0.1: 0.1: 0.1:    86  0.317499999999999  0.31749999         86 0.9524999999      86  0
87 0.1: 0.1: 0.1:    87  0.319999999999999  0.31999999         87 0.9599999999      87  0
88 0.1: 0.1: 0.1:    88  0.322499999999999  0.32249999         88 0.9674999999      88  0
89 0.1: 0.1: 0.1:    89  0.324999999999999  0.32499999         89 0.9749999999      89  0
90 0.1: 0.1: 0.1:    90  0.327499999999999  0.32749999         90 0.9824999999      90  0
91 0.1: 0.1: 0.1:    91  0.329999999999999  0.32999999         91 0.9899999999      91  0
92 0.1: 0.1: 0.1:    92  0.332499999999999  0.33249999         92 0.9974999999      92  0
93 0.1: 0.1: 0.1:    93  0.334999999999999  0.33499999         93 1.005 <<          93  1
94 0.1: 0.1: 0.1:    94  0.337499999999999  0.33749999         94 1.0125 <<         94  1
95 0.1: 0.1: 0.1:    95  0.339999999999999  0.33999999         95 1.02 <<           95  1
96 0.1: 0.1: 0.1:    96  0.342499999999999  0.34249999         96 1.0275 <<         96  1
97 0.1: 0.1: 0.1:    97  0.344999999999999  0.34499999         97 1.035 <<          97  1
98 0.1: 0.1: 0.1:    98  0.347499999999999  0.34749999         98 1.0425 <<         98  1
99 0.1: 0.1: 0.1:    99  0.349999999999999  0.34999999         99 1.05 <<           99  1
100 0.1: 0.1: 0.1:                                             100 1.0575 <<        100  1
```

Action potentials (netInput >= threshold) 57

Comment: There is no action potential until iteration 93. All subsequent iterations generate an action potential. The strengthening is the same for all synapses thus netInput is >= threshold once the neuron dendrite inputs are each >= 1/3.

C4 Input values random number between 0.1 and 0.25

Code: `aSynapse[j].inAxon = rnd.Next(10, 25) / 100.0;`



```
Granny neuron

No. dendrites [3]    Threshold [1]    Iterations [100]    Activation Function (●) Binary
                                                                              ( ) Sigmoid

Synapse input            Dendrite inputs to neuron           Net input          Neuron output
0 0.24: 0.13: 0.24:      0  0.2544  0.134225  0.2544         0 0.643025         10  0
1 0.14: 0.16: 0.21:      1  0.1533  0.1716  0.233625         1 0.558525         11  0
2 0.1: 0.13: 0.24:       2  0.112  0.14365  0.2814           2 0.53705          12  1
3 0.16: 0.15: 0.21:      3  0.1856  0.171375  0.25725        3 0.614225         13  0
4 0.1: 0.21: 0.24:       4  0.1185  0.25095  0.3084          4 0.67785          14  0
5 0.21: 0.17: 0.13:      5  0.259875  0.210375  0.171275     5 0.641525         15  1
6 0.15: 0.21: 0.23:      6  0.19125  0.2709  0.31625         6 0.7784           16  0
7 0.1: 0.2: 0.13:        7  0.13  0.268  0.182975            7 0.580975         17  1
8 0.11: 0.18: 0.15:      8  0.146025  0.2493  0.21675        8 0.612075         18  0
9 0.18: 0.12: 0.16:      9  0.24705  0.1698  0.2376          9 0.65445          19  1
10 0.19: 0.2: 0.2:       10  0.2698  0.293  0.307            10 0.8698          20  0
11 0.23: 0.19: 0.17:     11  0.339825  0.287375  0.268175    11 0.895375        21  1
12 0.21: 0.23: 0.21:     12  0.3213  0.3611  0.3423          12 1.0247 <<       22  0
13 0.14: 0.11: 0.18:     13  0.2191  0.175725  0.3015        13 0.696325        23  1
14 0.14: 0.23: 0.16:     14  0.224  0.38065  0.2744          14 0.87905         24  0
15 0.23: 0.19: 0.22:     15  0.381225  0.323475  0.3894      15 1.0941 <<       25  1
16 0.21: 0.11: 0.14:     16  0.3591  0.1903  0.2527          16 0.8021          26  1
17 0.23: 0.21: 0.13:     17  0.406525  0.374325  0.238875    17 1.019725 <<     27  1
18 0.11: 0.23: 0.17:                                         18 0.94025         28  1

                         Action potentials (netInput >= threshold) [80]    [Fire my neuron]
```

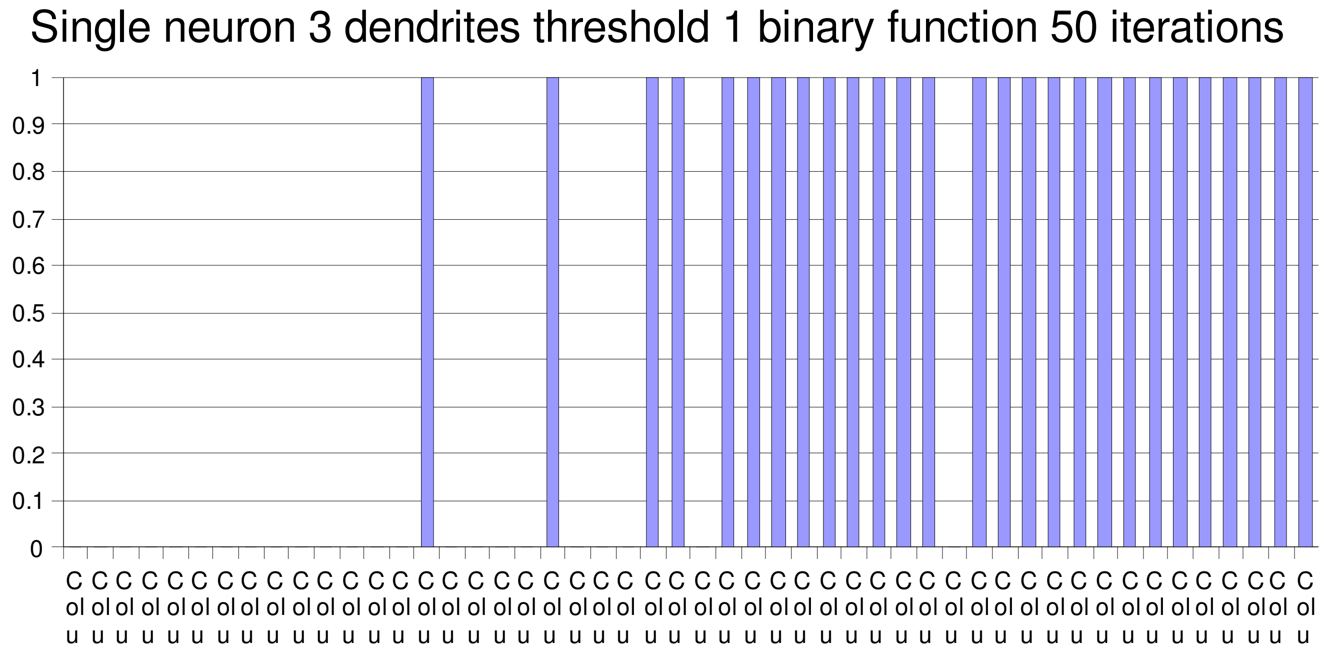
Comment: There is no action potential until iteration 12. Action potentials are consistent from iteration 40. Between iterations 12 and 40 the neuron demonstrates an increasing though inconsistent tendency to generate an action potential.

This model was re-run ten times. The patterns are shown below, truncated at 80 for clarity.

```
1   000000000000 1001010101010111111111111110 1111111111111111111111111111111111111111
2   00000000000000000 111101011000 111111111111111111111111111111111111111111111111111
3   00000000000000000000 1111110 11111111111111111111111111111111111111111111111111111
4   00000000 10000100000001100010 1111111111111111111111111111111111111111111111111111
5   000000000000000 100010111110 11111111111111111111111111111111111111111111111111111
6   000000000000000000 10001111111110 111111111111111111111111111111111111111111111111
7   00000000000000000 1100100000011010 11111111111111111111111111111111111111111111111
8   000000000000000000 11110100 111111111111111111111111111111111111111111111111111111
9   00000000000000000 1000010101111111111111111111110 11111111111111111111111111111111
10  00000000000000000 10101001111110 1111111111111111111111111111111111111111111111111
```

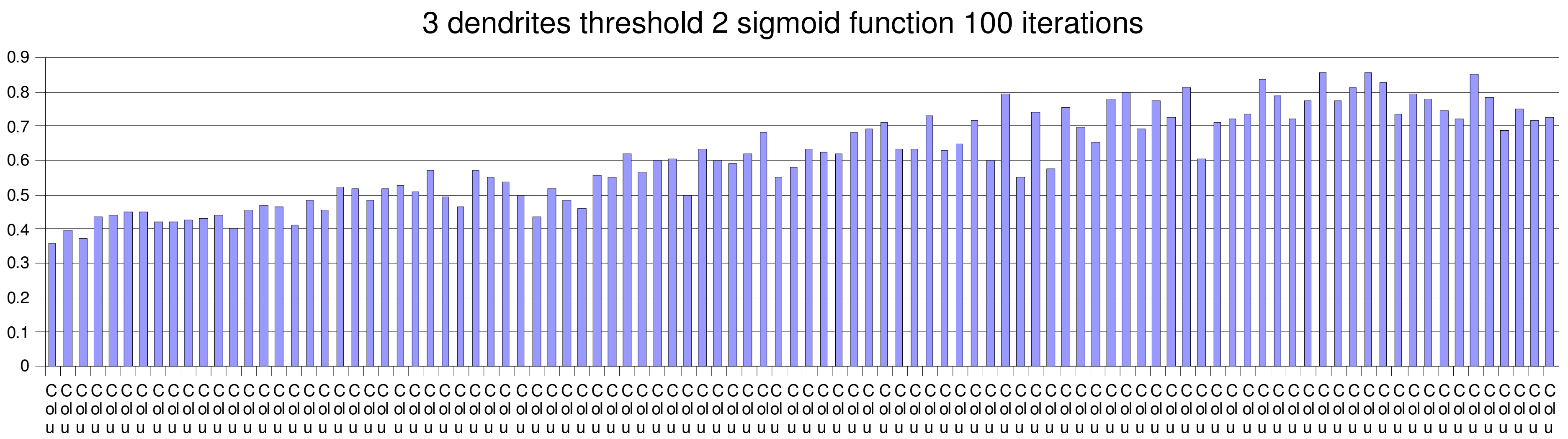
C5 Graphical representation of neuron output

The horizontal axis is the number of iterations



C.6 Neuron outputs with sigmoid function.

An action potential occurs when the output is greater than 0.5, the horizontal axis is the number of iterations.



Appendix D: Testing networks of neurons

The following tables show the results of testing the model with a network of connected neurons and synapses.

D.1 Binary pattern association

The network was presented with a number of input patterns and target patterns in binary form. In each case the model is two layer with four input and four output neurons. The results are summarised in section 6.2.1 of the main text.

Table D.1.1: input pattern 1010, target pattern 0101. Binary activation and Hebb rule. Threshold 1, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0.31 0.55 0.31 0.55	25. 0.31 1 0.31 1	50. 0.31 1 0.31 1	75. 0.31 1 0.31 1
1. 0.31 0.77 0.31 0.77	26. 0.31 1 0.31 1	51. 0.31 1 0.31 1	76. 0.31 1 0.31 1
2. 0.31 0.9 0.31 0.9	27. 0.31 1 0.31 1	52. 0.31 1 0.31 1	77. 0.31 1 0.31 1
3. 0.31 0.96 0.31 0.96	28. 0.31 1 0.31 1	53. 0.31 1 0.31 1	78. 0.31 1 0.31 1
4. 0.31 0.99 0.31 0.99	29. 0.31 1 0.31 1	54. 0.31 1 0.31 1	79. 0.31 1 0.31 1
5. 0.31 0.99 0.31 0.99	30. 0.31 1 0.31 1	55. 0.31 1 0.31 1	80. 0.31 1 0.31 1
6. 0.31 1 0.31 1	31. 0.31 1 0.31 1	56. 0.31 1 0.31 1	81. 0.31 1 0.31 1
7. 0.31 1 0.31 1	32. 0.31 1 0.31 1	57. 0.31 1 0.31 1	82. 0.31 1 0.31 1
8. 0.31 1 0.31 1	33. 0.31 1 0.31 1	58. 0.31 1 0.31 1	83. 0.31 1 0.31 1
9. 0.31 1 0.31 1	34. 0.31 1 0.31 1	59. 0.31 1 0.31 1	84. 0.31 1 0.31 1
10. 0.31 1 0.31 1	35. 0.31 1 0.31 1	60. 0.31 1 0.31 1	85. 0.31 1 0.31 1
11. 0.31 1 0.31 1	36. 0.31 1 0.31 1	61. 0.31 1 0.31 1	86. 0.31 1 0.31 1
12. 0.31 1 0.31 1	37. 0.31 1 0.31 1	62. 0.31 1 0.31 1	87. 0.31 1 0.31 1
13. 0.31 1 0.31 1	38. 0.31 1 0.31 1	63. 0.31 1 0.31 1	88. 0.31 1 0.31 1
14. 0.31 1 0.31 1	39. 0.31 1 0.31 1	64. 0.31 1 0.31 1	89. 0.31 1 0.31 1
15. 0.31 1 0.31 1	40. 0.31 1 0.31 1	65. 0.31 1 0.31 1	90. 0.31 1 0.31 1
16. 0.31 1 0.31 1	41. 0.31 1 0.31 1	66. 0.31 1 0.31 1	91. 0.31 1 0.31 1
17. 0.31 1 0.31 1	42. 0.31 1 0.31 1	67. 0.31 1 0.31 1	92. 0.31 1 0.31 1
18. 0.31 1 0.31 1	43. 0.31 1 0.31 1	68. 0.31 1 0.31 1	93. 0.31 1 0.31 1
19. 0.31 1 0.31 1	44. 0.31 1 0.31 1	69. 0.31 1 0.31 1	94. 0.31 1 0.31 1
20. 0.31 1 0.31 1	45. 0.31 1 0.31 1	70. 0.31 1 0.31 1	95. 0.31 1 0.31 1
21. 0.31 1 0.31 1	46. 0.31 1 0.31 1	71. 0.31 1 0.31 1	96. 0.31 1 0.31 1
22. 0.31 1 0.31 1	47. 0.31 1 0.31 1	72. 0.31 1 0.31 1	97. 0.31 1 0.31 1
23. 0.31 1 0.31 1	48. 0.31 1 0.31 1	73. 0.31 1 0.31 1	98. 0.31 1 0.31 1
24. 0.31 1 0.31 1	49. 0.31 1 0.31 1	74. 0.31 1 0.31 1	99. 0.31 1 0.31 1

Table D.1.2: input pattern 1100, target pattern 0011. Binary activation and Hebb rule. Threshold 1, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0.31 0.31 0.55 0.55	25. 0.31 0.31 1 1	50. 0.31 0.31 1 1	75. 0.31 0.31 1 1
1. 0.31 0.31 0.77 0.77	26. 0.31 0.31 1 1	51. 0.31 0.31 1 1	76. 0.31 0.31 1 1
2. 0.31 0.31 0.9 0.9	27. 0.31 0.31 1 1	52. 0.31 0.31 1 1	77. 0.31 0.31 1 1
3. 0.31 0.31 0.96 0.96	28. 0.31 0.31 1 1	53. 0.31 0.31 1 1	78. 0.31 0.31 1 1
4. 0.31 0.31 0.99 0.99	29. 0.31 0.31 1 1	54. 0.31 0.31 1 1	79. 0.31 0.31 1 1
5. 0.31 0.31 0.99 0.99	30. 0.31 0.31 1 1	55. 0.31 0.31 1 1	80. 0.31 0.31 1 1
6. 0.31 0.31 1 1	31. 0.31 0.31 1 1	56. 0.31 0.31 1 1	81. 0.31 0.31 1 1
7. 0.31 0.31 1 1	32. 0.31 0.31 1 1	57. 0.31 0.31 1 1	82. 0.31 0.31 1 1
8. 0.31 0.31 1 1	33. 0.31 0.31 1 1	58. 0.31 0.31 1 1	83. 0.31 0.31 1 1
9. 0.31 0.31 1 1	34. 0.31 0.31 1 1	59. 0.31 0.31 1 1	84. 0.31 0.31 1 1
10. 0.31 0.31 1 1	35. 0.31 0.31 1 1	60. 0.31 0.31 1 1	85. 0.31 0.31 1 1
11. 0.31 0.31 1 1	36. 0.31 0.31 1 1	61. 0.31 0.31 1 1	86. 0.31 0.31 1 1
12. 0.31 0.31 1 1	37. 0.31 0.31 1 1	62. 0.31 0.31 1 1	87. 0.31 0.31 1 1
13. 0.31 0.31 1 1	38. 0.31 0.31 1 1	63. 0.31 0.31 1 1	88. 0.31 0.31 1 1
14. 0.31 0.31 1 1	39. 0.31 0.31 1 1	64. 0.31 0.31 1 1	89. 0.31 0.31 1 1
15. 0.31 0.31 1 1	40. 0.31 0.31 1 1	65. 0.31 0.31 1 1	90. 0.31 0.31 1 1
16. 0.31 0.31 1 1	41. 0.31 0.31 1 1	66. 0.31 0.31 1 1	91. 0.31 0.31 1 1
17. 0.31 0.31 1 1	42. 0.31 0.31 1 1	67. 0.31 0.31 1 1	92. 0.31 0.31 1 1
18. 0.31 0.31 1 1	43. 0.31 0.31 1 1	68. 0.31 0.31 1 1	93. 0.31 0.31 1 1
19. 0.31 0.31 1 1	44. 0.31 0.31 1 1	69. 0.31 0.31 1 1	94. 0.31 0.31 1 1
20. 0.31 0.31 1 1	45. 0.31 0.31 1 1	70. 0.31 0.31 1 1	95. 0.31 0.31 1 1
21. 0.31 0.31 1 1	46. 0.31 0.31 1 1	71. 0.31 0.31 1 1	96. 0.31 0.31 1 1
22. 0.31 0.31 1 1	47. 0.31 0.31 1 1	72. 0.31 0.31 1 1	97. 0.31 0.31 1 1
23. 0.31 0.31 1 1	48. 0.31 0.31 1 1	73. 0.31 0.31 1 1	98. 0.31 0.31 1 1
24. 0.31 0.31 1 1	49. 0.31 0.31 1 1	74. 0.31 0.31 1 1	99. 0.31 0.31 1 1

Table D.1.3: input pattern 1000, target pattern 1001. Binary activation and Hebb rule. Threshold 1, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0.4 0.29 0.29 0.4	25. 1 0.29 0.29 1	50. 1 0.29 0.29 1	75. 1 0.29 0.29 1
1. 0.52 0.29 0.29 0.52	26. 1 0.29 0.29 1	51. 1 0.29 0.29 1	76. 1 0.29 0.29 1
2. 0.65 0.29 0.29 0.65	27. 1 0.29 0.29 1	52. 1 0.29 0.29 1	77. 1 0.29 0.29 1
3. 0.75 0.29 0.29 0.75	28. 1 0.29 0.29 1	53. 1 0.29 0.29 1	78. 1 0.29 0.29 1
4. 0.83 0.29 0.29 0.83	29. 1 0.29 0.29 1	54. 1 0.29 0.29 1	79. 1 0.29 0.29 1
5. 0.89 0.29 0.29 0.89	30. 1 0.29 0.29 1	55. 1 0.29 0.29 1	80. 1 0.29 0.29 1
6. 0.93 0.29 0.29 0.93	31. 1 0.29 0.29 1	56. 1 0.29 0.29 1	81. 1 0.29 0.29 1
7. 0.96 0.29 0.29 0.96	32. 1 0.29 0.29 1	57. 1 0.29 0.29 1	82. 1 0.29 0.29 1
8. 0.97 0.29 0.29 0.97	33. 1 0.29 0.29 1	58. 1 0.29 0.29 1	83. 1 0.29 0.29 1
9. 0.98 0.29 0.29 0.98	34. 1 0.29 0.29 1	59. 1 0.29 0.29 1	84. 1 0.29 0.29 1
10. 0.99 0.29 0.29 0.99	35. 1 0.29 0.29 1	60. 1 0.29 0.29 1	85. 1 0.29 0.29 1
11. 0.99 0.29 0.29 0.99	36. 1 0.29 0.29 1	61. 1 0.29 0.29 1	86. 1 0.29 0.29 1
12. 1 0.29 0.29 1	37. 1 0.29 0.29 1	62. 1 0.29 0.29 1	87. 1 0.29 0.29 1
13. 1 0.29 0.29 1	38. 1 0.29 0.29 1	63. 1 0.29 0.29 1	88. 1 0.29 0.29 1
14. 1 0.29 0.29 1	39. 1 0.29 0.29 1	64. 1 0.29 0.29 1	89. 1 0.29 0.29 1
15. 1 0.29 0.29 1	40. 1 0.29 0.29 1	65. 1 0.29 0.29 1	90. 1 0.29 0.29 1
16. 1 0.29 0.29 1	41. 1 0.29 0.29 1	66. 1 0.29 0.29 1	91. 1 0.29 0.29 1
17. 1 0.29 0.29 1	42. 1 0.29 0.29 1	67. 1 0.29 0.29 1	92. 1 0.29 0.29 1
18. 1 0.29 0.29 1	43. 1 0.29 0.29 1	68. 1 0.29 0.29 1	93. 1 0.29 0.29 1
19. 1 0.29 0.29 1	44. 1 0.29 0.29 1	69. 1 0.29 0.29 1	94. 1 0.29 0.29 1
20. 1 0.29 0.29 1	45. 1 0.29 0.29 1	70. 1 0.29 0.29 1	95. 1 0.29 0.29 1
21. 1 0.29 0.29 1	46. 1 0.29 0.29 1	71. 1 0.29 0.29 1	96. 1 0.29 0.29 1
22. 1 0.29 0.29 1	47. 1 0.29 0.29 1	72. 1 0.29 0.29 1	97. 1 0.29 0.29 1
23. 1 0.29 0.29 1	48. 1 0.29 0.29 1	73. 1 0.29 0.29 1	98. 1 0.29 0.29 1
24. 1 0.29 0.29 1	49. 1 0.29 0.29 1	74. 1 0.29 0.29 1	99. 1 0.29 0.29 1

Table D.1.4: input pattern 0100, target pattern 0111 Binary activation and Hebb rule. Threshold 1, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0.29 0.4 0.4 0.4	25. 0.29 1 1 1	50. 0.29 1 1 1	75. 0.29 1 1 1
1. 0.29 0.52 0.52 0.52	26. 0.29 1 1 1	51. 0.29 1 1 1	76. 0.29 1 1 1
2. 0.29 0.65 0.65 0.65	27. 0.29 1 1 1	52. 0.29 1 1 1	77. 0.29 1 1 1
3. 0.29 0.75 0.75 0.75	28. 0.29 1 1 1	53. 0.29 1 1 1	78. 0.29 1 1 1
4. 0.29 0.83 0.83 0.83	29. 0.29 1 1 1	54. 0.29 1 1 1	79. 0.29 1 1 1
5. 0.29 0.89 0.89 0.89	30. 0.29 1 1 1	55. 0.29 1 1 1	80. 0.29 1 1 1
6. 0.29 0.93 0.93 0.93	31. 0.29 1 1 1	56. 0.29 1 1 1	81. 0.29 1 1 1
7. 0.29 0.96 0.96 0.96	32. 0.29 1 1 1	57. 0.29 1 1 1	82. 0.29 1 1 1
8. 0.29 0.97 0.97 0.97	33. 0.29 1 1 1	58. 0.29 1 1 1	83. 0.29 1 1 1
9. 0.29 0.98 0.98 0.98	34. 0.29 1 1 1	59. 0.29 1 1 1	84. 0.29 1 1 1
10. 0.29 0.99 0.99 0.99	35. 0.29 1 1 1	60. 0.29 1 1 1	85. 0.29 1 1 1
11. 0.29 0.99 0.99 0.99	36. 0.29 1 1 1	61. 0.29 1 1 1	86. 0.29 1 1 1
12. 0.29 1 1 1	37. 0.29 1 1 1	62. 0.29 1 1 1	87. 0.29 1 1 1
13. 0.29 1 1 1	38. 0.29 1 1 1	63. 0.29 1 1 1	88. 0.29 1 1 1
14. 0.29 1 1 1	39. 0.29 1 1 1	64. 0.29 1 1 1	89. 0.29 1 1 1
15. 0.29 1 1 1	40. 0.29 1 1 1	65. 0.29 1 1 1	90. 0.29 1 1 1
16. 0.29 1 1 1	41. 0.29 1 1 1	66. 0.29 1 1 1	91. 0.29 1 1 1
17. 0.29 1 1 1	42. 0.29 1 1 1	67. 0.29 1 1 1	92. 0.29 1 1 1
18. 0.29 1 1 1	43. 0.29 1 1 1	68. 0.29 1 1 1	93. 0.29 1 1 1
19. 0.29 1 1 1	44. 0.29 1 1 1	69. 0.29 1 1 1	94. 0.29 1 1 1
20. 0.29 1 1 1	45. 0.29 1 1 1	70. 0.29 1 1 1	95. 0.29 1 1 1
21. 0.29 1 1 1	46. 0.29 1 1 1	71. 0.29 1 1 1	96. 0.29 1 1 1
22. 0.29 1 1 1	47. 0.29 1 1 1	72. 0.29 1 1 1	97. 0.29 1 1 1
23. 0.29 1 1 1	48. 0.29 1 1 1	73. 0.29 1 1 1	98. 0.29 1 1 1
24. 0.29 1 1 1	49. 0.29 1 1 1	74. 0.29 1 1 1	99. 0.29 1 1 1

The threshold was then changed to 6.0 and the same patterns presented.

Table D.1.5: input pattern 1010, target pattern 0101. Binary activation and Hebb rule. Threshold 6, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0 0.01 0 0.01	25. 0 1 0 1	50. 0 1 0 1	75. 0 1 0 1
1. 0 0.02 0 0.02	26. 0 1 0 1	51. 0 1 0 1	76. 0 1 0 1
2. 0 0.06 0 0.06	27. 0 1 0 1	52. 0 1 0 1	77. 0 1 0 1
3. 0 0.14 0 0.14	28. 0 1 0 1	53. 0 1 0 1	78. 0 1 0 1
4. 0 0.31 0 0.31	29. 0 1 0 1	54. 0 1 0 1	79. 0 1 0 1
5. 0 0.55 0 0.55	30. 0 1 0 1	55. 0 1 0 1	80. 0 1 0 1
6. 0 0.77 0 0.77	31. 0 1 0 1	56. 0 1 0 1	81. 0 1 0 1
7. 0 0.9 0 0.9	32. 0 1 0 1	57. 0 1 0 1	82. 0 1 0 1
8. 0 0.96 0 0.96	33. 0 1 0 1	58. 0 1 0 1	83. 0 1 0 1
9. 0 0.99 0 0.99	34. 0 1 0 1	59. 0 1 0 1	84. 0 1 0 1
10. 0 0.99 0 0.99	35. 0 1 0 1	60. 0 1 0 1	85. 0 1 0 1
11. 0 1 0 1	36. 0 1 0 1	61. 0 1 0 1	86. 0 1 0 1
12. 0 1 0 1	37. 0 1 0 1	62. 0 1 0 1	87. 0 1 0 1
13. 0 1 0 1	38. 0 1 0 1	63. 0 1 0 1	88. 0 1 0 1
14. 0 1 0 1	39. 0 1 0 1	64. 0 1 0 1	89. 0 1 0 1
15. 0 1 0 1	40. 0 1 0 1	65. 0 1 0 1	90. 0 1 0 1
16. 0 1 0 1	41. 0 1 0 1	66. 0 1 0 1	91. 0 1 0 1
17. 0 1 0 1	42. 0 1 0 1	67. 0 1 0 1	92. 0 1 0 1
18. 0 1 0 1	43. 0 1 0 1	68. 0 1 0 1	93. 0 1 0 1
19. 0 1 0 1	44. 0 1 0 1	69. 0 1 0 1	94. 0 1 0 1
20. 0 1 0 1	45. 0 1 0 1	70. 0 1 0 1	95. 0 1 0 1
21. 0 1 0 1	46. 0 1 0 1	71. 0 1 0 1	96. 0 1 0 1
22. 0 1 0 1	47. 0 1 0 1	72. 0 1 0 1	97. 0 1 0 1
23. 0 1 0 1	48. 0 1 0 1	73. 0 1 0 1	98. 0 1 0 1
24. 0 1 0 1	49. 0 1 0 1	74. 0 1 0 1	99. 0 1 0 1

Table D.1.6: input pattern 1100, target pattern 0011. Binary activation and Hebb rule. Threshold 6, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0 0 0 0	25. 0 0 1 1	50. 0 0 1 1	75. 0 0 1 1
1. 0 0 0.01 0.01	26. 0 0 1 1	51. 0 0 1 1	76. 0 0 1 1
2. 0 0 0.01 0.01	27. 0 0 1 1	52. 0 0 1 1	77. 0 0 1 1
3. 0 0 0.02 0.02	28. 0 0 1 1	53. 0 0 1 1	78. 0 0 1 1
4. 0 0 0.04 0.04	29. 0 0 1 1	54. 0 0 1 1	79. 0 0 1 1
5. 0 0 0.06 0.06	30. 0 0 1 1	55. 0 0 1 1	80. 0 0 1 1
6. 0 0 0.09 0.09	31. 0 0 1 1	56. 0 0 1 1	81. 0 0 1 1
7. 0 0 0.14 0.14	32. 0 0 1 1	57. 0 0 1 1	82. 0 0 1 1
8. 0 0 0.22 0.22	33. 0 0 1 1	58. 0 0 1 1	83. 0 0 1 1
9. 0 0 0.31 0.31	34. 0 0 1 1	59. 0 0 1 1	84. 0 0 1 1
10. 0 0 0.43 0.43	35. 0 0 1 1	60. 0 0 1 1	85. 0 0 1 1
11. 0 0 0.55 0.55	36. 0 0 1 1	61. 0 0 1 1	86. 0 0 1 1
12. 0 0 0.67 0.67	37. 0 0 1 1	62. 0 0 1 1	87. 0 0 1 1
13. 0 0 0.77 0.77	38. 0 0 1 1	63. 0 0 1 1	88. 0 0 1 1
14. 0 0 0.85 0.85	39. 0 0 1 1	64. 0 0 1 1	89. 0 0 1 1
15. 0 0 0.9 0.9	40. 0 0 1 1	65. 0 0 1 1	90. 0 0 1 1
16. 0 0 0.94 0.94	41. 0 0 1 1	66. 0 0 1 1	91. 0 0 1 1
17. 0 0 0.96 0.96	42. 0 0 1 1	67. 0 0 1 1	92. 0 0 1 1
18. 0 0 0.98 0.98	43. 0 0 1 1	68. 0 0 1 1	93. 0 0 1 1
19. 0 0 0.99 0.99	44. 0 0 1 1	69. 0 0 1 1	94. 0 0 1 1
20. 0 0 0.99 0.99	45. 0 0 1 1	70. 0 0 1 1	95. 0 0 1 1
21. 0 0 0.99 0.99	46. 0 0 1 1	71. 0 0 1 1	96. 0 0 1 1
22. 0 0 1 1	47. 0 0 1 1	72. 0 0 1 1	97. 0 0 1 1
23. 0 0 1 1	48. 0 0 1 1	73. 0 0 1 1	98. 0 0 1 1
24. 0 0 1 1	49. 0 0 1 1	74. 0 0 1 1	99. 0 0 1 1

Table D.1.7: input pattern 1000, target pattern 1001. Binary activation and Hebb rule. Threshold 6, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0 0 0 0	25. 1 0 0 1	50. 1 0 0 1	75. 1 0 0 1
1. 0.01 0 0 0.01	26. 1 0 0 1	51. 1 0 0 1	76. 1 0 0 1
2. 0.01 0 0 0.01	27. 1 0 0 1	52. 1 0 0 1	77. 1 0 0 1
3. 0.02 0 0 0.02	28. 1 0 0 1	53. 1 0 0 1	78. 1 0 0 1
4. 0.04 0 0 0.04	29. 1 0 0 1	54. 1 0 0 1	79. 1 0 0 1
5. 0.06 0 0 0.06	30. 1 0 0 1	55. 1 0 0 1	80. 1 0 0 1
6. 0.09 0 0 0.09	31. 1 0 0 1	56. 1 0 0 1	81. 1 0 0 1
7. 0.14 0 0 0.14	32. 1 0 0 1	57. 1 0 0 1	82. 1 0 0 1
8. 0.21 0 0 0.21	33. 1 0 0 1	58. 1 0 0 1	83. 1 0 0 1
9. 0.31 0 0 0.31	34. 1 0 0 1	59. 1 0 0 1	84. 1 0 0 1
10. 0.43 0 0 0.43	35. 1 0 0 1	60. 1 0 0 1	85. 1 0 0 1
11. 0.55 0 0 0.55	36. 1 0 0 1	61. 1 0 0 1	86. 1 0 0 1
12. 0.67 0 0 0.67	37. 1 0 0 1	62. 1 0 0 1	87. 1 0 0 1
13. 0.77 0 0 0.77	38. 1 0 0 1	63. 1 0 0 1	88. 1 0 0 1
14. 0.85 0 0 0.85	39. 1 0 0 1	64. 1 0 0 1	89. 1 0 0 1
15. 0.9 0 0 0.9	40. 1 0 0 1	65. 1 0 0 1	90. 1 0 0 1
16. 0.94 0 0 0.94	41. 1 0 0 1	66. 1 0 0 1	91. 1 0 0 1
17. 0.96 0 0 0.96	42. 1 0 0 1	67. 1 0 0 1	92. 1 0 0 1
18. 0.98 0 0 0.98	43. 1 0 0 1	68. 1 0 0 1	93. 1 0 0 1
19. 0.99 0 0 0.99	44. 1 0 0 1	69. 1 0 0 1	94. 1 0 0 1
20. 0.99 0 0 0.99	45. 1 0 0 1	70. 1 0 0 1	95. 1 0 0 1
21. 0.99 0 0 0.99	46. 1 0 0 1	71. 1 0 0 1	96. 1 0 0 1
22. 1 0 0 1	47. 1 0 0 1	72. 1 0 0 1	97. 1 0 0 1
23. 1 0 0 1	48. 1 0 0 1	73. 1 0 0 1	98. 1 0 0 1
24. 1 0 0 1	49. 1 0 0 1	74. 1 0 0 1	99. 1 0 0 1

Table D.1.8: input pattern 0100, target pattern 0111. Binary activation and Hebb rule. Threshold 6, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0 0.01 0.01 0.01	25. 0 1 1 1	50. 0 1 1 1	75. 0 1 1 1
1. 0 0.01 0.01 0.01	26. 0 1 1 1	51. 0 1 1 1	76. 0 1 1 1
2. 0 0.01 0.01 0.01	27. 0 1 1 1	52. 0 1 1 1	77. 0 1 1 1
3. 0 0.02 0.02 0.02	28. 0 1 1 1	53. 0 1 1 1	78. 0 1 1 1
4. 0 0.04 0.04 0.04	29. 0 1 1 1	54. 0 1 1 1	79. 0 1 1 1
5. 0 0.06 0.06 0.06	30. 0 1 1 1	55. 0 1 1 1	80. 0 1 1 1
6. 0 0.1 0.1 0.1	31. 0 1 1 1	56. 0 1 1 1	81. 0 1 1 1
7. 0 0.15 0.15 0.15	32. 0 1 1 1	57. 0 1 1 1	82. 0 1 1 1
8. 0 0.23 0.23 0.23	33. 0 1 1 1	58. 0 1 1 1	83. 0 1 1 1
9. 0 0.33 0.33 0.33	34. 0 1 1 1	59. 0 1 1 1	84. 0 1 1 1
10. 0 0.45 0.45 0.45	35. 0 1 1 1	60. 0 1 1 1	85. 0 1 1 1
11. 0 0.57 0.57 0.57	36. 0 1 1 1	61. 0 1 1 1	86. 0 1 1 1
12. 0 0.69 0.69 0.69	37. 0 1 1 1	62. 0 1 1 1	87. 0 1 1 1
13. 0 0.79 0.79 0.79	38. 0 1 1 1	63. 0 1 1 1	88. 0 1 1 1
14. 0 0.86 0.86 0.86	39. 0 1 1 1	64. 0 1 1 1	89. 0 1 1 1
15. 0 0.91 0.91 0.91	40. 0 1 1 1	65. 0 1 1 1	90. 0 1 1 1
16. 0 0.94 0.94 0.94	41. 0 1 1 1	66. 0 1 1 1	91. 0 1 1 1
17. 0 0.96 0.96 0.96	42. 0 1 1 1	67. 0 1 1 1	92. 0 1 1 1
18. 0 0.98 0.98 0.98	43. 0 1 1 1	68. 0 1 1 1	93. 0 1 1 1
19. 0 0.99 0.99 0.99	44. 0 1 1 1	69. 0 1 1 1	94. 0 1 1 1
20. 0 0.99 0.99 0.99	45. 0 1 1 1	70. 0 1 1 1	95. 0 1 1 1
21. 0 1 1 1	46. 0 1 1 1	71. 0 1 1 1	96. 0 1 1 1
22. 0 1 1 1	47. 0 1 1 1	72. 0 1 1 1	97. 0 1 1 1
23. 0 1 1 1	48. 0 1 1 1	73. 0 1 1 1	98. 0 1 1 1
24. 0 1 1 1	49. 0 1 1 1	74. 0 1 1 1	99. 0 1 1 1

D.2.1 Non-binary pattern association 1

The network was presented with a number of input patterns and target patterns representing the high frequency verbs used by McClelland and Rumelhart. The point at which the model captures the past phoneme(s) is indicated. The model is two layer with four input and four output neurons. The results are summarised in section 6.2.2 (table 9) of the main text.

Table D.2.1.1: verb: 'come'. input pattern 05 35 12 00, target pattern 05 38 12 00. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 15 35 19 13	60. 11 37 15 8	90. 9 38 14 6
1. 28 30 29 28	31. 15 35 19 13	61. 11 37 15 8	91. 9 38 14 6
2. 28 30 28 27	32. 15 36 18 12	62. 10 37 15 8	92. 8 38 14 6
3. 27 30 27 26	33. 14 36 18 12	63. 10 37 15 8	93. 8 38 14 6
4. 26 31 27 25	34. 14 36 18 12	64. 10 37 15 8	94. 8 38 14 5
5. 25 31 26 25	35. 14 36 18 12	65. 10 37 15 7	95. 8 38 13 5
6. 25 31 26 24	36. 14 36 18 11	66. 10 37 15 7	96. 8 38 13 5
7. 24 31 25 23	37. 14 36 18 11	67. 10 37 15 7	97. 8 38 13 5
8. 23 32 25 22	38. 14 36 17 11	68. 10 37 15 7	98. 8 38 13 5
9. 23 32 25 22	39. 13 36 17 11	69. 10 37 15 7	99. 8 38 13 5
10. 22 32 24 21	40. 13 36 17 11	70. 10 38 15 7 <<<	150. 7 38 12 4
11. 22 32 24 20	41. 13 36 17 11	71. 10 38 14 7	200. 6 38 12 3
12. 21 33 23 20	42. 13 36 17 10	72. 10 38 14 7	250. 6 38 12 2
13. 21 33 23 19	43. 13 36 17 10	73. 10 38 14 7	300. 5 38 12 2
14. 20 33 23 19	44. 13 37 17 10	74. 9 38 14 7	400. 5 38 12 1
15. 20 33 22 18	45. 12 37 17 10	75. 9 38 14 7	500. 5 38 12 1
16. 19 33 22 18	46. 12 37 16 10	76. 9 38 14 7	600. 5 38 12 1
17. 19 34 22 17	47. 12 37 16 10	77. 9 38 14 7	700. 5 38 12 1
18. 19 34 21 17	48. 12 37 16 9	78. 9 38 14 6	800. 5 38 12 1
19. 18 34 21 16	49. 12 37 16 9	79. 9 38 14 6	900. 5 38 12 1
20. 18 34 21 16	50. 12 37 16 9	80. 9 38 14 6	1000. 5 38 12 1
21. 18 34 21 16	51. 12 37 16 9	81. 9 38 14 6	1134. 5 38 12 1
22. 17 34 20 15	52. 11 37 16 9	82. 9 38 14 6	1135. 5 38 12 0
23. 17 34 20 15	53. 11 37 16 9	83. 9 38 14 6	1136. 5 38 12 0
24. 17 35 20 15	54. 11 37 16 9	84. 9 38 14 6	
25. 16 35 20 14	55. 11 37 16 9	85. 9 38 14 6	
26. 16 35 19 14	56. 11 37 15 8	86. 9 38 14 6	
27. 16 35 19 14	57. 11 37 15 8	87. 9 38 14 6	
28. 16 35 19 13	58. 11 37 15 8	88. 9 38 14 6	
29. 15 35 19 13	59. 11 37 15 8	89. 9 38 14 6	

Table D.2.1.2: verb: 'get'. input pattern 06 28 03 00, target pattern 06 31 03 00. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 16 30 15 13	60. 12 31 10 8	90. 9 31 8 6
1. 28 29 28 28	31. 16 30 14 13	61. 11 31 10 8	91. 9 31 8 6
2. 28 29 27 27	32. 16 30 14 13	62. 11 31 10 8	92. 9 31 7 6
3. 27 29 27 26	33. 15 30 14 13	63. 11 31 10 8	93. 9 31 7 6
4. 26 29 26 26	34. 15 30 14 12	64. 11 31 9 8	94. 9 31 7 6
5. 26 30 25 25	35. 15 30 13 12	65. 11 31 9 8	95. 9 31 7 6
6. 25 30 24 24	36. 15 31 13 12 <<<	66. 11 31 9 8	96. 9 31 7 6
7. 24 30 24 23	37. 15 31 13 12	67. 11 31 9 8	97. 9 31 7 6
8. 24 30 23 23	38. 14 31 13 12	68. 11 31 9 8	98. 9 31 7 6
9. 23 30 23 22	39. 14 31 13 11	69. 11 31 9 7	99. 9 31 7 6
10. 23 30 22 21	40. 14 31 13 11	70. 11 31 9 7	150. 8 31 6 4
11. 22 30 21 21	41. 14 31 12 11	71. 11 31 9 7	200. 7 31 5 3
12. 22 30 21 20	42. 14 31 12 11	72. 11 31 9 7	300. 6 31 4 2
13. 21 30 20 20	43. 14 31 12 11	73. 10 31 9 7	400. 6 31 3 1
14. 21 30 20 19	44. 13 31 12 10	74. 10 31 9 7	600. 6 31 3 1
15. 21 30 20 19	45. 13 31 12 10	75. 10 31 9 7	800. 6 31 3 1
16. 20 30 19 18	46. 13 31 12 10	76. 10 31 8 7	1000. 6 31 3 1
17. 20 30 19 18	47. 13 31 11 10	77. 10 31 8 7	1201. 6 31 3 1
18. 19 30 18 17	48. 13 31 11 10	78. 10 31 8 7	1202. 6 31 3 0
19. 19 30 18 17	49. 13 31 11 10	79. 10 31 8 7	1203. 6 31 3 0
20. 19 30 18 16	50. 13 31 11 10	80. 10 31 8 7	
21. 18 30 17 16	51. 13 31 11 9	81. 10 31 8 7	
22. 18 30 17 16	52. 12 31 11 9	82. 10 31 8 6	
23. 18 30 17 15	53. 12 31 11 9	83. 10 31 8 6	
24. 18 30 16 15	54. 12 31 11 9	84. 10 31 8 6	
25. 17 30 16 15	55. 12 31 10 9	85. 10 31 8 6	
26. 17 30 16 14	56. 12 31 10 9	86. 10 31 8 6	
27. 17 30 15 14	57. 12 31 10 9	87. 10 31 8 6	
28. 16 30 15 14	58. 12 31 10 9	88. 10 31 8 6	
29. 16 30 15 14	59. 12 31 10 8	89. 10 31 8 6	

Table D.2.1.3: verb: 'give'. input pattern 06 26 08 00, target pattern 06 38 08 00. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 16 35 17 13	60. 12 37 13 8	90. 9 38 11 6
1. 28 30 28 28	31. 16 35 17 13	61. 11 37 13 8	91. 9 38 11 6
2. 28 30 28 27	32. 16 35 17 13	62. 11 37 13 8	92. 9 38 11 6
3. 27 30 27 26	33. 15 35 16 13	63. 11 37 12 8	93. 9 38 11 6
4. 26 30 27 26	34. 15 36 16 12	64. 11 37 12 8	94. 9 38 11 6
5. 26 31 26 25	35. 15 36 16 12	65. 11 37 12 8	95. 9 38 11 6
6. 25 31 25 24	36. 15 36 16 12	66. 11 37 12 8	96. 9 38 11 6
7. 24 31 25 23	37. 15 36 16 12	67. 11 37 12 8	97. 9 38 11 6
8. 24 31 24 23	38. 14 36 15 11	68. 11 37 12 7	98. 9 38 10 6
9. 23 32 24 22	39. 14 36 15 11	69. 11 37 12 7	99. 9 38 10 5
10. 23 32 23 21	40. 14 36 15 11	70. 11 37 12 7	150. 8 38 9 4
11. 22 32 23 21	41. 14 36 15 11	71. 11 37 12 7	200. 7 38 9 3
12. 22 32 22 20	42. 14 36 15 11	72. 11 37 12 7	250. 7 38 8 2
13. 21 33 22 20	43. 14 36 15 11	73. 10 37 12 7	300. 6 38 8 2
14. 21 33 22 19	44. 13 36 15 10	74. 10 38 12 7 <<<	400. 6 38 8 1
15. 21 33 21 19	45. 13 36 14 10	75. 10 38 12 7	600. 6 38 8 1
16. 20 33 21 18	46. 13 36 14 10	76. 10 38 12 7	800. 6 38 8 1
17. 20 33 20 18	47. 13 37 14 10	77. 10 38 11 7	1000. 6 38 8 1
18. 19 33 20 17	48. 13 37 14 10	78. 10 38 11 7	1191. 6 38 8 1
19. 19 34 20 17	49. 13 37 14 10	79. 10 38 11 7	1192. 6 38 8 0
20. 19 34 19 16	50. 13 37 14 9	80. 10 38 11 7	1193. 6 38 8 0
21. 18 34 19 16	51. 12 37 14 9	81. 10 38 11 6	
22. 18 34 19 16	52. 12 37 14 9	82. 10 38 11 6	
23. 18 34 19 15	53. 12 37 13 9	83. 10 38 11 6	
24. 17 34 18 15	54. 12 37 13 9	84. 10 38 11 6	
25. 17 35 18 15	55. 12 37 13 9	85. 10 38 11 6	
26. 17 35 18 14	56. 12 37 13 9	86. 10 38 11 6	
27. 17 35 18 14	57. 12 37 13 9	87. 10 38 11 6	
28. 16 35 17 14	58. 12 37 13 8	88. 10 38 11 6	
29. 16 35 17 14	59. 12 37 13 8	89. 9 38 11 6	

Table D.2.1.4: verb: 'look'. input pattern 11 33 05 00, target pattern 11 33 05 03. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 18 32 15 14	60. 15 33 11 10	90. 13 33 9 7
1. 29 29 28 28	31. 18 32 15 14	61. 14 33 11 9	91. 13 33 9 7
2. 28 30 28 27	32. 18 32 15 14	62. 14 33 11 9	92. 13 33 9 7
3. 27 30 27 27	33. 18 32 15 14	63. 14 33 10 9	93. 13 33 8 7
4. 27 30 26 26	34. 18 32 14 13	64. 14 33 10 9	94. 13 33 8 7
5. 26 30 25 25	35. 17 32 14 13	65. 14 33 10 9	95. 13 33 8 7
6. 26 30 25 24	36. 17 32 14 13	66. 14 33 10 9	96. 13 33 8 7
7. 25 30 24 24	37. 17 32 14 13	67. 14 33 10 9	97. 13 33 8 7
8. 25 30 23 23	38. 17 32 14 13	68. 14 33 10 9	98. 13 33 8 7
9. 24 30 23 22	39. 17 32 13 12	69. 14 33 10 9	99. 13 33 8 7
10. 24 30 22 22	40. 17 32 13 12	70. 14 33 10 9	150. 12 33 7 5
11. 24 31 22 21	41. 16 32 13 12	71. 14 33 10 9	200. 11 33 6 5
12. 23 31 21 21	42. 16 32 13 12	72. 14 33 10 9	250. 11 33 6 4
13. 23 31 21 20	43. 16 32 13 12	73. 14 33 10 8	300. 11 33 5 4
14. 22 31 20 20	44. 16 32 13 12	74. 14 33 10 8	350. 11 33 5 4
15. 22 31 20 19	45. 16 32 12 11	75. 14 33 9 8	373. 11 33 5 4
16. 22 31 20 19	46. 16 32 12 11	76. 14 33 9 8	374. 11 33 5 3 <<<
17. 21 31 19 18	47. 16 32 12 11	77. 13 33 9 8	375. 11 33 5 3
18. 21 31 19 18	48. 16 32 12 11	78. 13 33 9 8	
19. 21 31 18 18	49. 16 32 12 11	79. 13 33 9 8	
20. 21 31 18 17	50. 15 32 12 11	80. 13 33 9 8	
21. 20 31 18 17	51. 15 32 12 11	81. 13 33 9 8	
22. 20 31 17 17	52. 15 32 12 10	82. 13 33 9 8	
23. 20 31 17 16	53. 15 33 11 10	83. 13 33 9 8	
24. 20 31 17 16	54. 15 33 11 10	84. 13 33 9 8	
25. 19 32 17 16	55. 15 33 11 10	85. 13 33 9 8	
26. 19 32 16 15	56. 15 33 11 10	86. 13 33 9 8	
27. 19 32 16 15	57. 15 33 11 10	87. 13 33 9 8	
28. 19 32 16 15	58. 15 33 11 10	88. 13 33 9 7	
29. 18 32 15 15	59. 15 33 11 10	89. 13 33 9 7	

Table D.2.1.5: verb: 'take'. input pattern 03 38 05 00, target pattern 03 33 05 00. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 14 32 15 13	60. 10 33 11 8	90. 7 33 9 6
1. 28 29 28 28	31. 14 32 15 13	61. 10 33 11 8	91. 7 33 9 6
2. 27 30 28 27	32. 14 32 15 13	62. 9 33 11 8	92. 7 33 9 6
3. 27 30 27 26	33. 14 32 15 12	63. 9 33 10 8	93. 7 33 8 6
4. 26 30 26 25	34. 13 32 14 12	64. 9 33 10 8	94. 7 33 8 6
5. 25 30 25 25	35. 13 32 14 12	65. 9 33 10 8	95. 7 33 8 6
6. 24 30 25 24	36. 13 32 14 12	66. 9 33 10 7	96. 7 33 8 5
7. 24 30 24 23	37. 13 32 14 11	67. 9 33 10 7	97. 7 33 8 5
8. 23 30 23 22	38. 13 32 14 11	68. 9 33 10 7	98. 7 33 8 5
9. 22 30 23 22	39. 12 32 13 11	69. 9 33 10 7	99. 7 33 8 5
10. 22 30 22 21	40. 12 32 13 11	70. 9 33 10 7	150. 5 33 7 4
11. 21 31 22 20	41. 12 32 13 11	71. 9 33 10 7	200. 5 33 6 3
12. 21 31 21 20	42. 12 32 13 10	72. 9 33 10 7	300. 4 33 5 2
13. 20 31 21 19	43. 12 32 13 10	73. 8 33 10 7	400. 3 33 5 1
14. 20 31 20 19	44. 12 32 13 10	74. 8 33 10 7	600. 3 33 5 1
15. 19 31 20 18	45. 11 32 12 10	75. 8 33 10 7	800. 3 33 5 1
16. 19 31 20 18	46. 11 32 12 10	76. 8 33 9 7	1000. 3 33 5 1
17. 18 31 19 17	47. 11 32 12 10	77. 8 33 9 7	1153. 3 33 5 1
18. 18 31 19 17	48. 11 32 12 10	78. 8 33 9 7	1154. 3 33 5 0
19. 18 31 18 17	49. 11 32 12 9	79. 8 33 9 6	1155. 3 33 5 0
20. 17 31 18 16	50. 11 32 12 9	80. 8 33 9 6	
21. 17 31 18 16	51. 11 32 12 9	81. 8 33 9 6	
22. 17 31 17 15	52. 10 32 12 9	82. 8 33 9 6	
23. 16 31 17 15	53. 10 33 11 9 <<<	83. 8 33 9 6	
24. 16 31 17 15	54. 10 33 11 9	84. 8 33 9 6	
25. 16 32 17 14	55. 10 33 11 9	85. 8 33 9 6	
26. 15 32 16 14	56. 10 33 11 8	86. 8 33 9 6	
27. 15 32 16 14	57. 10 33 11 8	87. 8 33 9 6	
28. 15 32 16 14	58. 10 33 11 8	88. 8 33 9 6	
29. 15 32 15 13	59. 10 33 11 8	89. 7 33 9 6	

Table D.2.1.6: verb: 'go'. input pattern 06 39 00 00, target pattern 17 28 13 03. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 22 28 19 14	60. 19 28 16 10	90. 18 28 15 7
1. 29 29 29 28	31. 21 28 19 14	61. 19 28 16 10	91. 18 28 14 7
2. 28 29 28 27	32. 21 28 19 14	62. 19 28 16 9	92. 18 28 14 7
3. 28 29 28 27	33. 21 28 19 14	63. 19 28 16 9	93. 18 28 14 7
4. 28 29 27 26	34. 21 28 19 13	64. 19 28 16 9	94. 18 28 14 7
5. 27 29 27 25	35. 21 28 19 13	65. 19 28 16 9	95. 18 28 14 7
6. 27 29 26 24	36. 21 28 18 13	66. 19 28 16 9	96. 18 28 14 7
7. 27 29 26 24	37. 21 28 18 13	67. 19 28 16 9	97. 18 28 14 7
8. 26 29 25 23	38. 21 28 18 13	68. 19 28 15 9	98. 18 28 14 7
9. 26 29 25 22	39. 21 28 18 12	69. 19 28 15 9	99. 18 28 14 7
10. 26 29 25 22	40. 20 28 18 12	70. 19 28 15 9	150. 17 28 13 5
11. 25 29 24 21	41. 20 28 18 12	71. 18 28 15 9	200. 17 28 13 5
12. 25 29 24 21	42. 20 28 18 12	72. 18 28 15 9	250. 17 28 13 4
13. 25 29 23 20	43. 20 28 17 12	73. 18 28 15 8	300. 17 28 13 4
14. 25 29 23 20	44. 20 28 17 12	74. 18 28 15 8	350. 17 28 13 4
15. 24 29 23 19	45. 20 28 17 11	75. 18 28 15 8	375. 17 28 13 4
16. 24 29 23 19	46. 20 28 17 11	76. 18 28 15 8	376. 17 28 13 3 <<<
17. 24 29 22 18	47. 20 28 17 11	77. 18 28 15 8	377. 17 28 13 3
18. 24 29 22 18	48. 20 28 17 11	78. 18 28 15 8	
19. 23 29 22 18	49. 20 28 17 11	79. 18 28 15 8	
20. 23 29 21 17	50. 20 28 17 11	80. 18 28 15 8	
21. 23 29 21 17	51. 20 28 17 11	81. 18 28 15 8	
22. 23 29 21 17	52. 19 28 17 10	82. 18 28 15 8	
23. 23 29 21 16	53. 19 28 17 10	83. 18 28 15 8	
24. 23 29 21 16	54. 19 28 16 10	84. 18 28 15 8	
25. 22 29 20 16	55. 19 28 16 10	85. 18 28 15 8	
26. 22 28 20 15	56. 19 28 16 10	86. 18 28 15 8	
27. 22 28 20 15	57. 19 28 16 10	87. 18 28 15 8	
28. 22 28 20 15	58. 19 28 16 10	88. 18 28 15 8	
29. 22 28 20 15	59. 19 28 16 10	89. 18 28 15 7	

Table D.2.1.7: verb: 'have'. input pattern 14 29 08 00, target pattern 14 29 04 00. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 20 29 15 13	60. 17 29 10 8	90. 15 29 8 6
1. 29 29 28 28	31. 20 29 14 13	61. 17 29 10 8	91. 15 29 8 6
2. 28 29 28 27	32. 20 29 14 12	62. 17 29 10 8	92. 15 29 8 6
3. 28 29 27 26	33. 19 29 14 12	63. 16 29 10 8	93. 15 29 8 6
4. 27 29 26 25	34. 19 29 14 12	64. 16 29 10 8	94. 15 29 8 6
5. 27 29 25 25	35. 19 29 14 12	65. 16 29 10 8	95. 15 29 8 5
6. 26 29 24 24	36. 19 29 13 12	66. 16 29 10 7	96. 15 29 8 5
7. 26 29 24 23	37. 19 29 13 11	67. 16 29 9 7	97. 15 29 8 5
8. 26 29 23 22	38. 19 29 13 11	68. 16 29 9 7	98. 15 29 8 5
9. 25 29 23 22	39. 19 29 13 11	69. 16 29 9 7	99. 15 29 8 5
10. 25 29 22 21	40. 18 29 13 11	70. 16 29 9 7	150. 14 29 6 4
11. 24 29 22 20	41. 18 29 13 11	71. 16 29 9 7	200. 14 29 5 3
12. 24 29 21 20	42. 18 29 12 10	72. 16 29 9 7	250. 14 29 5 2
13. 24 29 21 19	43. 18 29 12 10	73. 16 29 9 7	300. 14 29 5 2
14. 23 29 20 19	44. 18 29 12 10	74. 16 29 9 7	315. 14 29 5 2
15. 23 29 20 18	45. 18 29 12 10	75. 16 29 9 7	316. 14 29 4 2 <<<
16. 23 29 19 18	46. 18 29 12 10	76. 16 29 9 7	317. 14 29 4 2
17. 23 29 19 17	47. 18 29 12 10	77. 16 29 9 7	400. 14 29 4 1
18. 22 29 18 17	48. 18 29 11 9	78. 16 29 9 6	600. 14 29 4 1
19. 22 29 18 17	49. 18 29 11 9	79. 16 29 9 6	800. 14 29 4 1
20. 22 29 18 16	50. 17 29 11 9	80. 16 29 9 6	1000. 14 29 4 1
21. 22 29 17 16	51. 17 29 11 9	81. 16 29 8 6	1144. 14 29 4 1
22. 21 29 17 15	52. 17 29 11 9	82. 16 29 8 6	1145. 14 29 4 0
23. 21 29 17 15	53. 17 29 11 9	83. 16 29 8 6	1146. 14 29 4 0
24. 21 29 16 15	54. 17 29 11 9	84. 16 29 8 6	
25. 21 29 16 14	55. 17 29 11 9	85. 15 29 8 6	
26. 21 29 16 14	56. 17 29 11 8	86. 15 29 8 6	
27. 20 29 15 14	57. 17 29 10 8	87. 15 29 8 6	
28. 20 29 15 13	58. 17 29 10 8	88. 15 29 8 6	
29. 20 29 15 13	59. 17 29 10 8	89. 15 29 8 6	

Table D.2.1.8: verb: 'live'. input pattern 11 26 09 00, target pattern 11 26 09 04. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 18 27 17 15	60. 15 26 13 10	90. 13 26 11 8
1. 29 29 29 28	31. 18 27 17 15	61. 15 26 13 10	91. 13 26 11 8
2. 28 29 28 28	32. 18 27 17 14	62. 14 26 13 10	92. 13 26 11 8
3. 27 29 27 27	33. 18 27 17 14	63. 14 26 13 10	93. 13 26 11 8
4. 27 29 27 26	34. 18 27 17 14	64. 14 26 13 10	94. 13 26 11 8
5. 26 29 26 25	35. 17 27 16 14	65. 14 26 13 10	95. 13 26 11 8
6. 26 29 25 25	36. 17 27 16 14	66. 14 26 13 10	96. 13 26 11 8
7. 25 29 25 24	37. 17 27 16 13	67. 14 26 13 10	97. 13 26 11 8
8. 25 28 24 23	38. 17 27 16 13	68. 14 26 13 10	98. 13 26 11 8
9. 24 28 24 23	39. 17 27 16 13	69. 14 26 13 9	99. 13 26 11 8
10. 24 28 23 22	40. 17 27 16 13	70. 14 26 13 9	150. 12 26 10 6
11. 24 28 23 22	41. 17 27 15 13	71. 14 26 12 9	200. 11 26 9 5
12. 23 28 23 21	42. 16 27 15 13	72. 14 26 12 9	300. 11 26 9 5
13. 23 28 22 21	43. 16 27 15 12	73. 14 26 12 9	321. 11 26 9 5
14. 22 28 22 20	44. 16 27 15 12	74. 14 26 12 9	322. 11 26 9 4 <<<
15. 22 28 21 20	45. 16 27 15 12	75. 14 26 12 9	323. 11 26 9 4
16. 22 28 21 19	46. 16 27 15 12	76. 14 26 12 9	
17. 21 28 21 19	47. 16 27 15 12	77. 14 26 12 9	
18. 21 28 20 18	48. 16 27 14 12	78. 13 26 12 9	
19. 21 28 20 18	49. 16 27 14 11	79. 13 26 12 9	
20. 21 28 20 18	50. 16 27 14 11	80. 13 26 12 9	
21. 20 28 19 17	51. 15 27 14 11	81. 13 26 12 9	
22. 20 27 19 17	52. 15 27 14 11	82. 13 26 12 9	
23. 20 27 19 17	53. 15 27 14 11	83. 13 26 12 8	
24. 20 27 19 16	54. 15 26 14 11	84. 13 26 12 8	
25. 19 27 18 16	55. 15 26 14 11	85. 13 26 12 8	
26. 19 27 18 16	56. 15 26 14 11	86. 13 26 12 8	
27. 19 27 18 16	57. 15 26 14 11	87. 13 26 12 8	
28. 19 27 18 15	58. 15 26 13 10	88. 13 26 12 8	
29. 19 27 18 15	59. 15 26 13 10	89. 13 26 11 8	

Table D.2.1.9: verb: 'feel'. input pattern 07 25 11 00, target pattern 07 28 11 03. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	30. 16 28 18 14	60. 12 28 15 10	90. 10 28 13 8
1. 28 29 29 28	31. 16 28 18 14	61. 12 28 15 10	91. 10 28 13 7
2. 28 29 28 27	32. 16 28 18 14	62. 12 28 15 10	92. 10 28 13 7
3. 27 29 27 27	33. 16 28 18 14	63. 12 28 14 9	93. 10 28 13 7
4. 26 29 27 26	34. 16 28 18 14	64. 12 28 14 9	94. 10 28 13 7
5. 26 29 26 25	35. 15 28 18 13	65. 12 28 14 9	95. 10 28 13 7
6. 25 29 26 24	36. 15 28 17 13	66. 12 28 14 9	96. 10 28 13 7
7. 25 29 25 24	37. 15 28 17 13	67. 11 28 14 9	97. 10 28 13 7
8. 24 29 25 23	38. 15 28 17 13	68. 11 28 14 9	98. 10 28 13 7
9. 23 29 24 23	39. 15 28 17 13	69. 11 28 14 9	99. 10 28 13 7
10. 23 29 24 22	40. 14 28 17 12	70. 11 28 14 9	150. 8 28 12 5
11. 23 29 24 21	41. 14 28 17 12	71. 11 28 14 9	200. 8 28 11 5
12. 22 29 23 21	42. 14 28 16 12	72. 11 28 14 9	250. 7 28 11 4
13. 22 29 23 20	43. 14 28 16 12	73. 11 28 14 9	300. 7 28 11 4
14. 21 29 23 20	44. 14 28 16 12	74. 11 28 14 9	350. 7 28 11 4
15. 21 29 22 19	45. 14 28 16 12	75. 11 28 14 8	384. 7 28 11 4
16. 20 29 22 19	46. 14 28 16 11	76. 11 28 14 8	385. 7 28 11 3 <<<
17. 20 29 22 19	47. 13 28 16 11	77. 11 28 14 8	386. 7 28 11 3
18. 20 29 21 18	48. 13 28 16 11	78. 11 28 14 8	
19. 19 29 21 18	49. 13 28 16 11	79. 11 28 13 8	
20. 19 29 21 17	50. 13 28 16 11	80. 11 28 13 8	
21. 19 29 20 17	51. 13 28 15 11	81. 11 28 13 8	
22. 18 29 20 17	52. 13 28 15 11	82. 10 28 13 8	
23. 18 29 20 16	53. 13 28 15 11	83. 10 28 13 8	
24. 18 29 20 16	54. 13 28 15 10	84. 10 28 13 8	
25. 18 29 19 16	55. 13 28 15 10	85. 10 28 13 8	
26. 17 28 19 16	56. 12 28 15 10	86. 10 28 13 8	
27. 17 28 19 15	57. 12 28 15 10	87. 10 28 13 8	
28. 17 28 19 15	58. 12 28 15 10	88. 10 28 13 8	
29. 17 28 19 15	59. 12 28 15 10	89. 10 28 13 8	

D.2.1 Non-binary pattern association 2

The network was presented with a number of input patterns and target patterns representing infix morphology found in Semitic languages such as Arabic. The point at which the model captures the past phonemes is indicated with arrows. The model is two layer with five input and five output neurons. The results are summarised in section 6.2.2 (table 10) of the main text.

Table D.2.2.1: root /*k*ətəb/ target /kætæb/. input pattern 05 37 03 37 02, target pattern 05 29 03 29 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 29 12 29 12	60. 9 29 8 29 7	90. 7 29 6 29 6
1. 29 30 29 30 29	31. 13 29 12 29 12	61. 9 29 8 29 7	91. 7 29 6 29 5
2. 28 30 27 30 27	32. 13 29 12 29 11	62. 9 29 8 29 7	92. 7 29 6 29 5
3. 27 30 26 30 26	33. 13 29 12 29 11	63. 9 29 8 29 7	93. 7 29 6 29 5
4. 26 30 25 30 25	34. 12 29 11 29 11	64. 9 29 8 29 7	94. 7 29 6 29 5
5. 25 30 24 30 24	35. 12 29 11 29 11	65. 9 29 8 29 7	95. 7 29 6 29 5
6. 24 30 23 30 23	36. 12 29 11 29 10	66. 9 29 7 29 7	96. 7 29 6 29 5
7. 23 30 23 30 22	37. 12 29 11 29 10	67. 9 29 7 29 7	97. 7 29 6 29 5
8. 22 30 22 30 21	38. 12 29 11 29 10	68. 9 29 7 29 7	98. 7 29 6 29 5
9. 22 30 21 30 21	39. 12 29 10 29 10	69. 9 29 7 29 7	99. 7 29 6 29 5
10. 21 30 20 30 20	40. 11 29 10 29 10	70. 8 29 7 29 7	150. 6 29 5 29 4
11. 20 30 20 30 19	41. 11 29 10 29 10	71. 8 29 7 29 7	200. 6 29 4 29 3
12. 20 30 19 30 19	42. 11 29 10 29 9	72. 8 29 7 29 6	250. 5 29 4 29 3
13. 19 30 18 30 18	43. 11 29 10 29 9	73. 8 29 7 29 6	300. 5 29 3 29 3
14. 19 29 18 29 18 <<<	44. 11 29 10 29 9	74. 8 29 7 29 6	345. 5 29 3 29 3
15. 18 29 17 29 17	45. 11 29 10 29 9	75. 8 29 7 29 6	346. 5 29 3 29 2
16. 18 29 17 29 17	46. 11 29 9 29 9	76. 8 29 7 29 6	347. 5 29 3 29 2
17. 17 29 16 29 16	47. 10 29 9 29 9	77. 8 29 7 29 6	
18. 17 29 16 29 16	48. 10 29 9 29 9	78. 8 29 7 29 6	
19. 17 29 16 29 15	49. 10 29 9 29 9	79. 8 29 7 29 6	
20. 16 29 15 29 15	50. 10 29 9 29 8	80. 8 29 7 29 6	
21. 16 29 15 29 14	51. 10 29 9 29 8	81. 8 29 7 29 6	
22. 15 29 15 29 14	52. 10 29 9 29 8	82. 8 29 7 29 6	
23. 15 29 14 29 14	53. 10 29 9 29 8	83. 8 29 6 29 6	
24. 15 29 14 29 13	54. 10 29 9 29 8	84. 8 29 6 29 6	
25. 15 29 14 29 13	55. 10 29 8 29 8	85. 8 29 6 29 6	
26. 14 29 13 29 13	56. 10 29 8 29 8	86. 8 29 6 29 6	
27. 14 29 13 29 13	57. 9 29 8 29 8	87. 8 29 6 29 6	
28. 14 29 13 29 12	58. 9 29 8 29 8	88. 8 29 6 29 6	
29. 14 29 13 29 12	59. 9 29 8 29 7	89. 7 29 6 29 6	

Table D.2.2.2: root /*k*ətəb/ target /kɪtæb/. input pattern 05 37 03 37 02, target pattern 05 26 03 29 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 27 12 29 12	60. 9 26 8 29 7	90. 7 26 6 29 6
1. 29 30 29 30 29	31. 13 27 12 29 12	61. 9 26 8 29 7	91. 7 26 6 29 5
2. 28 30 27 30 27	32. 13 27 12 29 11	62. 9 26 8 29 7	92. 7 26 6 29 5
3. 27 29 26 30 26	33. 13 27 12 29 11	63. 9 26 8 29 7	93. 7 26 6 29 5
4. 26 29 25 30 25	34. 12 27 11 29 11	64. 9 26 8 29 7	94. 7 26 6 29 5
5. 25 29 24 30 24	35. 12 27 11 29 11	65. 9 26 8 29 7	95. 7 26 6 29 5
6. 24 29 23 30 23	36. 12 27 11 29 10	66. 9 26 7 29 7	96. 7 26 6 29 5
7. 23 29 23 30 22	37. 12 27 11 29 10	67. 9 26 7 29 7	97. 7 26 6 29 5
8. 22 29 22 30 21	38. 12 27 11 29 10	68. 9 26 7 29 7	98. 7 26 6 29 5
9. 22 29 21 30 21	39. 12 27 10 29 10	69. 9 26 7 29 7	99. 7 26 6 29 5
10. 21 28 20 30 20	40. 11 27 10 29 10	70. 8 26 7 29 7	150. 6 26 5 29 4
11. 20 28 20 30 19	41. 11 27 10 29 10	71. 8 26 7 29 7	200. 6 26 4 29 3
12. 20 28 19 30 19	42. 11 27 10 29 9	72. 8 26 7 29 6	250. 5 26 4 29 3
13. 19 28 18 30 18	43. 11 27 10 29 9	73. 8 26 7 29 6	300. 5 26 3 29 3
14. 19 28 18 29 18	44. 11 26 10 29 9 <<<	74. 8 26 7 29 6	345. 5 26 3 29 3
15. 18 28 17 29 17	45. 11 26 10 29 9	75. 8 26 7 29 6	346. 5 26 3 29 2
16. 18 28 17 29 17	46. 11 26 9 29 9	76. 8 26 7 29 6	347. 5 26 3 29 2
17. 17 28 16 29 16	47. 10 26 9 29 9	77. 8 26 7 29 6	
18. 17 28 16 29 16	48. 10 26 9 29 9	78. 8 26 7 29 6	
19. 17 28 16 29 15	49. 10 26 9 29 9	79. 8 26 7 29 6	
20. 16 27 15 29 15	50. 10 26 9 29 8	80. 8 26 7 29 6	
21. 16 27 15 29 14	51. 10 26 9 29 8	81. 8 26 7 29 6	
22. 15 27 15 29 14	52. 10 26 9 29 8	82. 8 26 7 29 6	
23. 15 27 14 29 14	53. 10 26 9 29 8	83. 8 26 6 29 6	
24. 15 27 14 29 13	54. 10 26 9 29 8	84. 8 26 6 29 6	
25. 15 27 14 29 13	55. 10 26 8 29 8	85. 8 26 6 29 6	
26. 14 27 13 29 13	56. 10 26 8 29 8	86. 8 26 6 29 6	
27. 14 27 13 29 13	57. 9 26 8 29 8	87. 8 26 6 29 6	
28. 14 27 13 29 12	58. 9 26 8 29 8	88. 8 26 6 29 6	
29. 14 27 13 29 12	59. 9 26 8 29 7	89. 7 26 6 29 6	

Table D.2.2.3: root /*k*ətəb/ target /æktɪb/. input pattern 05 37 03 37 02, target pattern 29 05 03 26 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 29 13 12 27 12	60. 29 9 8 26 7	90. 29 7 6 26 6
1. 30 29 29 30 29	31. 29 13 12 27 12	61. 29 9 8 26 7	91. 29 7 6 26 5
2. 30 28 27 30 27	32. 29 13 12 27 11	62. 29 9 8 26 7	92. 29 7 6 26 5
3. 30 27 26 29 26	33. 29 13 12 27 11	63. 29 9 8 26 7	93. 29 7 6 26 5
4. 30 26 25 29 25	34. 29 12 11 27 11	64. 29 9 8 26 7	94. 29 7 6 26 5
5. 30 25 24 29 24	35. 29 12 11 27 11	65. 29 9 8 26 7	95. 29 7 6 26 5
6. 30 24 23 29 23	36. 29 12 11 27 10	66. 29 9 7 26 7	96. 29 7 6 26 5
7. 30 23 23 29 22	37. 29 12 11 27 10	67. 29 9 7 26 7	97. 29 7 6 26 5
8. 30 22 22 29 21	38. 29 12 11 27 10	68. 29 9 7 26 7	98. 29 7 6 26 5
9. 30 22 21 29 21	39. 29 12 10 27 10	69. 29 9 7 26 7	99. 29 7 6 26 5
10. 30 21 20 28 20	40. 29 11 10 27 10	70. 29 8 7 26 7	150. 29 6 5 26 4
11. 30 20 20 28 19	41. 29 11 10 27 10	71. 29 8 7 26 7	200. 29 6 4 26 3
12. 30 20 19 28 19	42. 29 11 10 27 9	72. 29 8 7 26 6	250. 29 5 4 26 3
13. 30 19 18 28 18	43. 29 11 10 27 9	73. 29 8 7 26 6	300. 29 5 3 26 3
14. 29 19 18 28 18	44. 29 11 10 26 9 <<<	74. 29 8 7 26 6	345. 29 5 3 26 3
15. 29 18 17 28 17	45. 29 11 10 26 9	75. 29 8 7 26 6	346. 29 5 3 26 2
16. 29 18 17 28 17	46. 29 11 9 26 9	76. 29 8 7 26 6	347. 29 5 3 26 2
17. 29 17 16 28 16	47. 29 10 9 26 9	77. 29 8 7 26 6	
18. 29 17 16 28 16	48. 29 10 9 26 9	78. 29 8 7 26 6	
19. 29 17 16 28 15	49. 29 10 9 26 9	79. 29 8 7 26 6	
20. 29 16 15 27 15	50. 29 10 9 26 8	80. 29 8 7 26 6	
21. 29 16 15 27 14	51. 29 10 9 26 8	81. 29 8 7 26 6	
22. 29 15 15 27 14	52. 29 10 9 26 8	82. 29 8 7 26 6	
23. 29 15 14 27 14	53. 29 10 9 26 8	83. 29 8 6 26 6	
24. 29 15 14 27 13	54. 29 10 9 26 8	84. 29 8 6 26 6	
25. 29 15 14 27 13	55. 29 10 8 26 8	85. 29 8 6 26 6	
26. 29 14 13 27 13	56. 29 10 8 26 8	86. 29 8 6 26 6	
27. 29 14 13 27 13	57. 29 9 8 26 8	87. 29 8 6 26 6	
28. 29 14 13 27 12	58. 29 9 8 26 8	88. 29 8 6 26 6	
29. 29 14 13 27 12	59. 29 9 8 26 7	89. 29 7 6 26 6	

Table D.2.2.4: root /*k*ətəb/ target /kɪtaːb/. input pattern 05 37 03 37 02, target pattern 05 26 03 30 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 27 12 30 12	60. 9 26 8 30 7	90. 7 26 6 30 6
1. 29 30 29 30 29	31. 13 27 12 30 12	61. 9 26 8 30 7	91. 7 26 6 30 5
2. 28 30 27 30 27	32. 13 27 12 30 11	62. 9 26 8 30 7	92. 7 26 6 30 5
3. 27 29 26 30 26	33. 13 27 12 30 11	63. 9 26 8 30 7	93. 7 26 6 30 5
4. 26 29 25 30 25	34. 12 27 11 30 11	64. 9 26 8 30 7	94. 7 26 6 30 5
5. 25 29 24 30 24	35. 12 27 11 30 11	65. 9 26 8 30 7	95. 7 26 6 30 5
6. 24 29 23 30 23	36. 12 27 11 30 10	66. 9 26 7 30 7	96. 7 26 6 30 5
7. 23 29 23 30 22	37. 12 27 11 30 10	67. 9 26 7 30 7	97. 7 26 6 30 5
8. 22 29 22 30 21	38. 12 27 11 30 10	68. 9 26 7 30 7	98. 7 26 6 30 5
9. 22 29 21 30 21	39. 12 27 10 30 10	69. 9 26 7 30 7	99. 7 26 6 30 5
10. 21 28 20 30 20	40. 11 27 10 30 10	70. 8 26 7 30 7	150. 6 26 5 30 4
11. 20 28 20 30 19	41. 11 27 10 30 10	71. 8 26 7 30 7	200. 6 26 4 30 3
12. 20 28 19 30 19	42. 11 27 10 30 9	72. 8 26 7 30 6	250. 5 26 4 30 3
13. 19 28 18 30 18	43. 11 27 10 30 9	73. 8 26 7 30 6	300. 5 26 3 30 3
14. 19 28 18 30 18	44. 11 26 10 30 9 <<<	74. 8 26 7 30 6	345. 5 26 3 30 3
15. 18 28 17 30 17	45. 11 26 10 30 9	75. 8 26 7 30 6	346. 5 26 3 30 2
16. 18 28 17 30 17	46. 11 26 9 30 9	76. 8 26 7 30 6	347. 5 26 3 30 2
17. 17 28 16 30 16	47. 10 26 9 30 9	77. 8 26 7 30 6	
18. 17 28 16 30 16	48. 10 26 9 30 9	78. 8 26 7 30 6	
19. 17 28 16 30 15	49. 10 26 9 30 9	79. 8 26 7 30 6	
20. 16 27 15 30 15	50. 10 26 9 30 8	80. 8 26 7 30 6	
21. 16 27 15 30 14	51. 10 26 9 30 8	81. 8 26 7 30 6	
22. 15 27 15 30 14	52. 10 26 9 30 8	82. 8 26 7 30 6	
23. 15 27 14 30 14	53. 10 26 9 30 8	83. 8 26 6 30 6	
24. 15 27 14 30 13	54. 10 26 9 30 8	84. 8 26 6 30 6	
25. 15 27 14 30 13	55. 10 26 8 30 8	85. 8 26 6 30 6	
26. 14 27 13 30 13	56. 10 26 8 30 8	86. 8 26 6 30 6	
27. 14 27 13 30 13	57. 9 26 8 30 8	87. 8 26 6 30 6	
28. 14 27 13 30 12	58. 9 26 8 30 8	88. 8 26 6 30 6	
29. 14 27 13 30 12	59. 9 26 8 30 7	89. 7 26 6 30 6	

Table D.2.2.5: root /*k*ətəb/ target /kʊtʊb/. input pattern 05 37 03 37 02, target pattern 05 33 03 33 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 32 12 32 12	60. 9 33 8 33 7	90. 7 33 6 33 6
1. 29 30 29 30 29	31. 13 32 12 32 12	61. 9 33 8 33 7	91. 7 33 6 33 5
2. 28 30 27 30 27	32. 13 32 12 32 11	62. 9 33 8 33 7	92. 7 33 6 33 5
3. 27 30 26 30 26	33. 13 32 12 32 11	63. 9 33 8 33 7	93. 7 33 6 33 5
4. 26 31 25 31 25	34. 12 32 11 32 11	64. 9 33 8 33 7	94. 7 33 6 33 5
5. 25 31 24 31 24	35. 12 33 11 33 11 <<<	65. 9 33 8 33 7	95. 7 33 6 33 5
6. 24 31 23 31 23	36. 12 33 11 33 10	66. 9 33 7 33 7	96. 7 33 6 33 5
7. 23 31 23 31 22	37. 12 33 11 33 10	67. 9 33 7 33 7	97. 7 33 6 33 5
8. 22 31 22 31 21	38. 12 33 11 33 10	68. 9 33 7 33 7	98. 7 33 6 33 5
9. 22 31 21 31 21	39. 12 33 10 33 10	69. 9 33 7 33 7	99. 7 33 6 33 5
10. 21 31 20 31 20	40. 11 33 10 33 10	70. 8 33 7 33 7	150. 6 33 5 33 4
11. 20 31 20 31 19	41. 11 33 10 33 10	71. 8 33 7 33 7	200. 6 33 4 33 3
12. 20 31 19 31 19	42. 11 33 10 33 9	72. 8 33 7 33 6	250. 5 33 4 33 3
13. 19 31 18 31 18	43. 11 33 10 33 9	73. 8 33 7 33 6	300. 5 33 3 33 3
14. 19 32 18 32 18	44. 11 33 10 33 9	74. 8 33 7 33 6	345. 5 33 3 33 3
15. 18 32 17 32 17	45. 11 33 10 33 9	75. 8 33 7 33 6	346. 5 33 3 33 2
16. 18 32 17 32 17	46. 11 33 9 33 9	76. 8 33 7 33 6	347. 5 33 3 33 2
17. 17 32 16 32 16	47. 10 33 9 33 9	77. 8 33 7 33 6	
18. 17 32 16 32 16	48. 10 33 9 33 9	78. 8 33 7 33 6	
19. 17 32 16 32 15	49. 10 33 9 33 9	79. 8 33 7 33 6	
20. 16 32 15 32 15	50. 10 33 9 33 8	80. 8 33 7 33 6	
21. 16 32 15 32 14	51. 10 33 9 33 8	81. 8 33 7 33 6	
22. 15 32 15 32 14	52. 10 33 9 33 8	82. 8 33 7 33 6	
23. 15 32 14 32 14	53. 10 33 9 33 8	83. 8 33 6 33 6	
24. 15 32 14 32 13	54. 10 33 9 33 8	84. 8 33 6 33 6	
25. 15 32 14 32 13	55. 10 33 8 33 8	85. 8 33 6 33 6	
26. 14 32 13 32 13	56. 10 33 8 33 8	86. 8 33 6 33 6	
27. 14 32 13 32 13	57. 9 33 8 33 8	87. 8 33 6 33 6	
28. 14 32 13 32 12	58. 9 33 8 33 8	88. 8 33 6 33 6	
29. 14 32 13 32 12	59. 9 33 8 33 7	89. 7 33 6 33 6	

Table D.2.2.6: root /kətəb/ target /kʌtɑːb/. input pattern 05 37 03 37 02, target pattern 05 35 03 30 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 34 12 30 12	60. 9 35 8 30 7	90. 7 35 6 30 6
1. 29 30 29 30 29	31. 13 34 12 30 12	61. 9 35 8 30 7	91. 7 35 6 30 5
2. 28 30 27 30 27	32. 13 34 12 30 11	62. 9 35 8 30 7	92. 7 35 6 30 5
3. 27 31 26 30 26	33. 13 34 12 30 11	63. 9 35 8 30 7	93. 7 35 6 30 5
4. 26 31 25 30 25	34. 12 34 11 30 11	64. 9 35 8 30 7	94. 7 35 6 30 5
5. 25 31 24 30 24	35. 12 34 11 30 11	65. 9 35 8 30 7	95. 7 35 6 30 5
6. 24 31 23 30 23	36. 12 34 11 30 10	66. 9 35 7 30 7	96. 7 35 6 30 5
7. 23 32 23 30 22	37. 12 34 11 30 10	67. 9 35 7 30 7	97. 7 35 6 30 5
8. 22 32 22 30 21	38. 12 34 11 30 10	68. 9 35 7 30 7	98. 7 35 6 30 5
9. 22 32 21 30 21	39. 12 34 10 30 10	69. 9 35 7 30 7	99. 7 35 6 30 5
10. 21 32 20 30 20	40. 11 34 10 30 10	70. 8 35 7 30 7	150. 6 35 5 30 4
11. 20 32 20 30 19	41. 11 34 10 30 10	71. 8 35 7 30 7	200. 6 35 4 30 3
12. 20 32 19 30 19	42. 11 34 10 30 9	72. 8 35 7 30 6	250. 5 35 4 30 3
13. 19 32 18 30 18	43. 11 34 10 30 9	73. 8 35 7 30 6	300. 5 35 3 30 3
14. 19 33 18 30 18	44. 11 35 10 30 9	74. 8 35 7 30 6	345. 5 35 3 30 3
15. 18 33 17 30 17	45. 11 35 10 30 9	75. 8 35 7 30 6	346. 5 35 3 30 2
16. 18 33 17 30 17	46. 11 35 9 30 9	76. 8 35 7 30 6	347. 5 35 3 30 2
17. 17 33 16 30 16	47. 10 35 9 30 9	77. 8 35 7 30 6	
18. 17 33 16 30 16	48. 10 35 9 30 9	78. 8 35 7 30 6	
19. 17 33 16 30 15	49. 10 35 9 30 9	79. 8 35 7 30 6	
20. 16 33 15 30 15	50. 10 35 9 30 8	80. 8 35 7 30 6	
21. 16 33 15 30 14	51. 10 35 9 30 8	81. 8 35 7 30 6	
22. 15 33 15 30 14	52. 10 35 9 30 8	82. 8 35 7 30 6	
23. 15 34 14 30 14	53. 10 35 9 30 8	83. 8 35 6 30 6	
24. 15 34 14 30 13	54. 10 35 9 30 8	84. 8 35 6 30 6	
25. 15 34 14 30 13	55. 10 35 8 30 8	85. 8 35 6 30 6	
26. 14 34 13 30 13	56. 10 35 8 30 8	86. 8 35 6 30 6	
27. 14 34 13 30 13	57. 9 35 8 30 8	87. 8 35 6 30 6	
28. 14 34 13 30 12	58. 9 35 8 30 8	88. 8 35 6 30 6	
29. 14 34 13 30 12	59. 9 35 8 30 7	89. 7 35 6 30 6	

Table D.2.2.7: root /kətəb/ target /kiːtɒb/. input pattern 05 37 03 37 02, target pattern 05 25 03 31 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 26 12 31 12	60. 9 25 8 31 7	90. 7 25 6 31 6
1. 29 30 29 30 29	31. 13 26 12 31 12	61. 9 25 8 31 7	91. 7 25 6 31 5
2. 28 30 27 30 27	32. 13 26 12 31 11	62. 9 25 8 31 7	92. 7 25 6 31 5
3. 27 29 26 30 26	33. 13 26 12 31 11	63. 9 25 8 31 7	93. 7 25 6 31 5
4. 26 29 25 30 25	34. 12 26 11 31 11	64. 9 25 8 31 7	94. 7 25 6 31 5
5. 25 29 24 30 24	35. 12 26 11 31 11	65. 9 25 8 31 7	95. 7 25 6 31 5
6. 24 29 23 30 23	36. 12 26 11 31 10	66. 9 25 7 31 7	96. 7 25 6 31 5
7. 23 29 23 30 22	37. 12 26 11 31 10	67. 9 25 7 31 7	97. 7 25 6 31 5
8. 22 28 22 30 21	38. 12 26 11 31 10	68. 9 25 7 31 7	98. 7 25 6 31 5
9. 22 28 21 30 21	39. 12 26 10 31 10	69. 9 25 7 31 7	99. 7 25 6 31 5
10. 21 28 20 30 20	40. 11 26 10 31 10	70. 8 25 7 31 7	150. 6 25 5 31 4
11. 20 28 20 30 19	41. 11 26 10 31 10	71. 8 25 7 31 7	200. 6 25 4 31 3
12. 20 28 19 30 19	42. 11 26 10 31 9	72. 8 25 7 31 6	250. 5 25 4 31 3
13. 19 28 18 30 18	43. 11 26 10 31 9	73. 8 25 7 31 6	300. 5 25 3 31 3
14. 19 28 18 31 18	44. 11 26 10 31 9	74. 8 25 7 31 6	345. 5 25 3 31 3
15. 18 27 17 31 17	45. 11 26 10 31 9	75. 8 25 7 31 6	346. 5 25 3 31 2
16. 18 27 17 31 17	46. 11 26 9 31 9	76. 8 25 7 31 6	347. 5 25 3 31 2
17. 17 27 16 31 16	47. 10 26 9 31 9	77. 8 25 7 31 6	
18. 17 27 16 31 16	48. 10 26 9 31 9	78. 8 25 7 31 6	
19. 17 27 16 31 15	49. 10 25 9 31 9	79. 8 25 7 31 6	
20. 16 27 15 31 15	50. 10 25 9 31 8	80. 8 25 7 31 6	
21. 16 27 15 31 14	51. 10 25 9 31 8	81. 8 25 7 31 6	
22. 15 27 15 31 14	52. 10 25 9 31 8	82. 8 25 7 31 6	
23. 15 27 14 31 14	53. 10 25 9 31 8	83. 8 25 6 31 6	
24. 15 27 14 31 13	54. 10 25 9 31 8	84. 8 25 6 31 6	
25. 15 26 14 31 13	55. 10 25 8 31 8	85. 8 25 6 31 6	
26. 14 26 13 31 13	56. 10 25 8 31 8	86. 8 25 6 31 6	
27. 14 26 13 31 13	57. 9 25 8 31 8	87. 8 25 6 31 6	
28. 14 26 13 31 12	58. 9 25 8 31 8	88. 8 25 6 31 6	
29. 14 26 13 31 12	59. 9 25 8 31 7	89. 7 25 6 31 6	

Table D.2.2.8: root /kətəb/ target /kətuːb/. input pattern 05 37 03 37 02, target pattern 05 37 03 34 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 36 12 33 12	60. 9 37 8 34 7	90. 7 37 6 34 6
1. 29 30 29 30 29	31. 13 36 12 33 12	61. 9 37 8 34 7	91. 7 37 6 34 5
2. 28 31 27 30 27	32. 13 36 12 33 11	62. 9 37 8 34 7	92. 7 37 6 34 5
3. 27 31 26 31 26	33. 13 36 12 33 11	63. 9 37 8 34 7	93. 7 37 6 34 5
4. 26 31 25 31 25	34. 12 36 11 33 11	64. 9 37 8 34 7	94. 7 37 6 34 5
5. 25 32 24 31 24	35. 12 36 11 33 11	65. 9 37 8 34 7	95. 7 37 6 34 5
6. 24 32 23 31 23	36. 12 36 11 33 10	66. 9 37 7 34 7	96. 7 37 6 34 5
7. 23 32 23 31 22	37. 12 36 11 33 10	67. 9 37 7 34 7	97. 7 37 6 34 5
8. 22 32 22 31 21	38. 12 36 11 33 10	68. 9 37 7 34 7	98. 7 37 6 34 5
9. 22 33 21 31 21	39. 12 36 10 33 10	69. 9 37 7 34 7	99. 7 37 6 34 5
10. 21 33 20 32 20	40. 11 36 10 34 10	70. 8 37 7 34 7	150. 6 37 5 34 4
11. 20 33 20 32 19	41. 11 36 10 34 10	71. 8 37 7 34 7	200. 6 37 4 34 3
12. 20 33 19 32 19	42. 11 36 10 34 9	72. 8 37 7 34 6	250. 5 37 4 34 3
13. 19 33 18 32 18	43. 11 36 10 34 9	73. 8 37 7 34 6	300. 5 37 3 34 3
14. 19 34 18 32 18	44. 11 36 10 34 9	74. 8 37 7 34 6	345. 5 37 3 34 3
15. 18 34 17 32 17	45. 11 36 10 34 9	75. 8 37 7 34 6	346. 5 37 3 34 2
16. 18 34 17 32 17	46. 11 36 9 34 9	76. 8 37 7 34 6	347. 5 37 3 34 2
17. 17 34 16 32 16	47. 10 36 9 34 9	77. 8 37 7 34 6	
18. 17 34 16 32 16	48. 10 36 9 34 9	78. 8 37 7 34 6	
19. 17 34 16 33 15	49. 10 37 9 34 9	79. 8 37 7 34 6	
20. 16 35 15 33 15	50. 10 37 9 34 8	80. 8 37 7 34 6	
21. 16 35 15 33 14	51. 10 37 9 34 8	81. 8 37 7 34 6	
22. 15 35 15 33 14	52. 10 37 9 34 8	82. 8 37 7 34 6	
23. 15 35 14 33 14	53. 10 37 9 34 8	83. 8 37 6 34 6	
24. 15 35 14 33 13	54. 10 37 9 34 8	84. 8 37 6 34 6	
25. 15 35 14 33 13	55. 10 37 8 34 8	85. 8 37 6 34 6	
26. 14 35 13 33 13	56. 10 37 8 34 8	86. 8 37 6 34 6	
27. 14 35 13 33 13	57. 9 37 8 34 8	87. 8 37 6 34 6	
28. 14 35 13 33 12	58. 9 37 8 34 8	88. 8 37 6 34 6	
29. 14 36 13 33 12	59. 9 37 8 34 7	89. 7 37 6 34 6	

Table D.2.2.9: root /kətəb/ target /ketɜːb/. input pattern 05 37 03 37 02, target pattern 05 27 03 36 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 28 12 35 12	60. 9 27 8 36 7	90. 7 27 6 36 6
1. 29 30 29 30 29	31. 13 28 12 35 12	61. 9 27 8 36 7	91. 7 27 6 36 5
2. 28 30 27 31 27	32. 13 28 12 35 11	62. 9 27 8 36 7	92. 7 27 6 36 5
3. 27 30 26 31 26	33. 13 28 12 35 11	63. 9 27 8 36 7	93. 7 27 6 36 5
4. 26 29 25 31 25	34. 12 28 11 35 11	64. 9 27 8 36 7	94. 7 27 6 36 5
5. 25 29 24 31 24	35. 12 28 11 35 11	65. 9 27 8 36 7	95. 7 27 6 36 5
6. 24 29 23 32 23	36. 12 28 11 35 10	66. 9 27 7 36 7	96. 7 27 6 36 5
7. 23 29 23 32 22	37. 12 27 11 35 10	67. 9 27 7 36 7	97. 7 27 6 36 5
8. 22 29 22 32 21	38. 12 27 11 35 10	68. 9 27 7 36 7	98. 7 27 6 36 5
9. 22 29 21 32 21	39. 12 27 10 35 10	69. 9 27 7 36 7	99. 7 27 6 36 5
10. 21 29 20 32 20	40. 11 27 10 35 10	70. 8 27 7 36 7	150. 6 27 5 36 4
11. 20 29 20 33 19	41. 11 27 10 35 10	71. 8 27 7 36 7	200. 6 27 4 36 3
12. 20 29 19 33 19	42. 11 27 10 35 9	72. 8 27 7 36 6	250. 5 27 4 36 3
13. 19 29 18 33 18	43. 11 27 10 35 9	73. 8 27 7 36 6	345. 5 27 3 36 3
14. 19 28 18 33 18	44. 11 27 10 35 9	74. 8 27 7 36 6	346. 5 27 3 36 2
15. 18 28 17 33 17	45. 11 27 10 35 9	75. 8 27 7 36 6	347. 5 27 3 36 2
16. 18 28 17 33 17	46. 11 27 9 35 9	76. 8 27 7 36 6	
17. 17 28 16 34 16	47. 10 27 9 36 9	77. 8 27 7 36 6	
18. 17 28 16 34 16	48. 10 27 9 36 9	78. 8 27 7 36 6	
19. 17 28 16 34 15	49. 10 27 9 36 9	79. 8 27 7 36 6	
20. 16 28 15 34 15	50. 10 27 9 36 8	80. 8 27 7 36 6	
21. 16 28 15 34 14	51. 10 27 9 36 8	81. 8 27 7 36 6	
22. 15 28 15 34 14	52. 10 27 9 36 8	82. 8 27 7 36 6	
23. 15 28 14 34 14	53. 10 27 9 36 8	83. 8 27 6 36 6	
24. 15 28 14 34 13	54. 10 27 9 36 8	84. 8 27 6 36 6	
25. 15 28 14 34 13	55. 10 27 8 36 8	85. 8 27 6 36 6	
26. 14 28 13 34 13	56. 10 27 8 36 8	86. 8 27 6 36 6	
27. 14 28 13 35 13	57. 9 27 8 36 8	87. 8 27 6 36 6	
28. 14 28 13 35 12	58. 9 27 8 36 8	88. 8 27 6 36 6	
29. 14 28 13 35 12	59. 9 27 8 36 7	89. 7 27 6 36 6	

Table D.2.2.10: root /*k*ətəb/ target /kuːtəb/. input pattern 05 37 03 37 02, target pattern 05 34 03 37 02. Sigmoid activation and delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0.

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 30 30 30 30 30	30. 13 33 12 36 12	60. 9 34 8 37 7	90. 7 34 6 37 6
1. 29 30 29 30 29	31. 13 33 12 36 12	61. 9 34 8 37 7	91. 7 34 6 37 5
2. 28 30 27 31 27	32. 13 33 12 36 11	62. 9 34 8 37 7	92. 7 34 6 37 5
3. 27 31 26 31 26	33. 13 33 12 36 11	63. 9 34 8 37 7	93. 7 34 6 37 5
4. 26 31 25 31 25	34. 12 33 11 36 11	64. 9 34 8 37 7	94. 7 34 6 37 5
5. 25 31 24 32 24	35. 12 33 11 36 11	65. 9 34 8 37 7	95. 7 34 6 37 5
6. 24 31 23 32 23	36. 12 33 11 36 10	66. 9 34 7 37 7	96. 7 34 6 37 5
7. 23 31 23 32 22	37. 12 33 11 36 10	67. 9 34 7 37 7	97. 7 34 6 37 5
8. 22 31 22 32 21	38. 12 33 11 36 10	68. 9 34 7 37 7	98. 7 34 6 37 5
9. 22 31 21 33 21	39. 12 33 10 36 10	69. 9 34 7 37 7	99. 7 34 6 37 5
10. 21 32 20 33 20	40. 11 34 10 36 10	70. 8 34 7 37 7	150. 6 34 5 37 4
11. 20 32 20 33 19	41. 11 34 10 36 10	71. 8 34 7 37 7	200. 6 34 4 37 3
12. 20 32 19 33 19	42. 11 34 10 36 9	72. 8 34 7 37 6	250. 5 34 4 37 3
13. 19 32 18 33 18	43. 11 34 10 36 9	73. 8 34 7 37 6	300. 5 34 3 37 3
14. 19 32 18 34 18	44. 11 34 10 36 9	74. 8 34 7 37 6	345. 5 34 3 37 3
15. 18 32 17 34 17	45. 11 34 10 36 9	75. 8 34 7 37 6	346. 5 34 3 37 2
16. 18 32 17 34 17	46. 11 34 9 36 9	76. 8 34 7 37 6	347. 5 34 3 37 2
17. 17 32 16 34 16	47. 10 34 9 36 9	77. 8 34 7 37 6	
18. 17 32 16 34 16	48. 10 34 9 36 9	78. 8 34 7 37 6	
19. 17 33 16 34 15	49. 10 34 9 37 9	79. 8 34 7 37 6	
20. 16 33 15 35 15	50. 10 34 9 37 8	80. 8 34 7 37 6	
21. 16 33 15 35 14	51. 10 34 9 37 8	81. 8 34 7 37 6	
22. 15 33 15 35 14	52. 10 34 9 37 8	82. 8 34 7 37 6	
23. 15 33 14 35 14	53. 10 34 9 37 8	83. 8 34 6 37 6	
24. 15 33 14 35 13	54. 10 34 9 37 8	84. 8 34 6 37 6	
25. 15 33 14 35 13	55. 10 34 8 37 8	85. 8 34 6 37 6	
26. 14 33 13 35 13	56. 10 34 8 37 8	86. 8 34 6 37 6	
27. 14 33 13 35 13	57. 9 34 8 37 8	87. 8 34 6 37 6	
28. 14 33 13 35 12	58. 9 34 8 37 8	88. 8 34 6 37 6	
29. 14 33 13 36 12	59. 9 34 8 37 7	89. 7 34 6 37 6	

D.3 Multiple input patterns

In the models described in section 4 of the text, a single network is able to the correct target form for multiple inputs. This was attempted with the model built for this project with the results shown below.

Table D.3.1: input patterns 1010 0110, target patterns 0101 1100. Binary activation and Hebb rule. Threshold 1, start strength 0.1, learn rate 0.5

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 0 1 0 1	26. 100 100 0 100	52. 100 100 0 100	78. 100 100 0 100
0. 1 2 0 1	26. 100 100 0 100	52. 100 100 0 100	78. 100 100 0 100
1. 1 6 0 2	27. 100 100 0 100	53. 100 100 0 100	79. 100 100 0 100
1. 2 15 0 2	27. 100 100 0 100	53. 100 100 0 100	79. 100 100 0 100
2. 2 33 0 6	28. 100 100 0 100	54. 100 100 0 100	80. 100 100 0 100
2. 6 57 0 6	28. 100 100 0 100	54. 100 100 0 100	80. 100 100 0 100
3. 6 79 0 15	29. 100 100 0 100	55. 100 100 0 100	81. 100 100 0 100
3. 15 91 0 15	29. 100 100 0 100	55. 100 100 0 100	81. 100 100 0 100
4. 15 96 0 33	30. 100 100 0 100	56. 100 100 0 100	82. 100 100 0 100
4. 33 99 0 33	30. 100 100 0 100	56. 100 100 0 100	82. 100 100 0 100
5. 33 100 0 57	31. 100 100 0 100	57. 100 100 0 100	83. 100 100 0 100
5. 57 100 0 57	31. 100 100 0 100	57. 100 100 0 100	83. 100 100 0 100
6. 57 100 0 79	32. 100 100 0 100	58. 100 100 0 100	84. 100 100 0 100
6. 79 100 0 79	32. 100 100 0 100	58. 100 100 0 100	84. 100 100 0 100
7. 79 100 0 91	33. 100 100 0 100	59. 100 100 0 100	85. 100 100 0 100
7. 91 100 0 91	33. 100 100 0 100	59. 100 100 0 100	85. 100 100 0 100
8. 91 100 0 96	34. 100 100 0 100	60. 100 100 0 100	86. 100 100 0 100
8. 96 100 0 96	34. 100 100 0 100	60. 100 100 0 100	86. 100 100 0 100
9. 96 100 0 99	35. 100 100 0 100	61. 100 100 0 100	87. 100 100 0 100
9. 99 100 0 99	35. 100 100 0 100	61. 100 100 0 100	87. 100 100 0 100
10. 99 100 0 100	36. 100 100 0 100	62. 100 100 0 100	88. 100 100 0 100
10. 100 100 0 100	36. 100 100 0 100	62. 100 100 0 100	88. 100 100 0 100
11. 100 100 0 100	37. 100 100 0 100	63. 100 100 0 100	89. 100 100 0 100
11. 100 100 0 100	37. 100 100 0 100	63. 100 100 0 100	89. 100 100 0 100
12. 100 100 0 100	38. 100 100 0 100	64. 100 100 0 100	90. 100 100 0 100
12. 100 100 0 100	38. 100 100 0 100	64. 100 100 0 100	90. 100 100 0 100
13. 100 100 0 100	39. 100 100 0 100	65. 100 100 0 100	91. 100 100 0 100
13. 100 100 0 100	39. 100 100 0 100	65. 100 100 0 100	91. 100 100 0 100
14. 100 100 0 100	40. 100 100 0 100	66. 100 100 0 100	92. 100 100 0 100
14. 100 100 0 100	40. 100 100 0 100	66. 100 100 0 100	92. 100 100 0 100
15. 100 100 0 100	41. 100 100 0 100	67. 100 100 0 100	93. 100 100 0 100
15. 100 100 0 100	41. 100 100 0 100	67. 100 100 0 100	93. 100 100 0 100
16. 100 100 0 100	42. 100 100 0 100	68. 100 100 0 100	94. 100 100 0 100
16. 100 100 0 100	42. 100 100 0 100	68. 100 100 0 100	94. 100 100 0 100
17. 100 100 0 100	43. 100 100 0 100	69. 100 100 0 100	95. 100 100 0 100
17. 100 100 0 100	43. 100 100 0 100	69. 100 100 0 100	95. 100 100 0 100
18. 100 100 0 100	44. 100 100 0 100	70. 100 100 0 100	96. 100 100 0 100
18. 100 100 0 100	44. 100 100 0 100	70. 100 100 0 100	96. 100 100 0 100
19. 100 100 0 100	45. 100 100 0 100	71. 100 100 0 100	97. 100 100 0 100
19. 100 100 0 100	45. 100 100 0 100	71. 100 100 0 100	97. 100 100 0 100
20. 100 100 0 100	46. 100 100 0 100	72. 100 100 0 100	98. 100 100 0 100
20. 100 100 0 100	46. 100 100 0 100	72. 100 100 0 100	98. 100 100 0 100
21. 100 100 0 100	47. 100 100 0 100	73. 100 100 0 100	99. 100 100 0 100
21. 100 100 0 100	47. 100 100 0 100	73. 100 100 0 100	99. 100 100 0 100
22. 100 100 0 100	48. 100 100 0 100	74. 100 100 0 100	
22. 100 100 0 100	48. 100 100 0 100	74. 100 100 0 100	
23. 100 100 0 100	49. 100 100 0 100	75. 100 100 0 100	
23. 100 100 0 100	49. 100 100 0 100	75. 100 100 0 100	
24. 100 100 0 100	50. 100 100 0 100	76. 100 100 0 100	
24. 100 100 0 100	50. 100 100 0 100	76. 100 100 0 100	
25. 100 100 0 100	51. 100 100 0 100	77. 100 100 0 100	
25. 100 100 0 100	51. 100 100 0 100	77. 100 100 0 100	

The model was subsequently run with various thresholds, start strengths, learn rates and using the sigmoid and delta rules. In each case the results were similar.

The model was then tested using the IPA codes shown in table 7 in the main text. A sample run is shown below.

Table D.3.2: input patterns 05 35 12 00 (/kʌm/), 06 28 03 00 (/get/), target patterns 05 38 12 00, (/keɪm/), 06 31 03 00 (/gɒt/). Sigmoid activation and Delta rule. Threshold 1, start strength 0.1, learn rate 0.5, rho 1.0

Iteration. Output	Iteration. Output	Iteration. Output	Iteration. Output
0. 29 29 29 29	25. 12 34 13 9	50. 8 34 10 5	75. 7 34 8 4
0. 28 30 29 28	25. 12 34 13 9	50. 9 34 10 5	75. 7 34 9 4
1. 28 30 28 27	26. 12 34 13 9	51. 8 34 10 5	76. 7 34 8 4
1. 27 30 27 26	26. 12 34 13 9	51. 9 34 10 5	76. 7 34 9 4
2. 26 30 26 25	27. 12 34 13 9	52. 8 34 10 5	77. 7 34 8 4
2. 25 30 26 25	27. 12 34 13 9	52. 9 34 10 5	77. 7 34 9 4
3. 25 30 25 24	28. 11 34 12 8	53. 8 34 9 5	78. 7 34 8 3
3. 24 30 25 23	28. 12 34 13 9	53. 8 34 10 5	78. 7 34 9 4
4. 24 31 24 22	29. 11 34 12 8	54. 8 34 9 5	79. 7 34 8 3
4. 23 31 24 22	29. 11 34 13 8	54. 8 34 10 5	79. 7 34 9 4
5. 22 31 23 21	30. 11 34 12 8	55. 8 34 9 5	80. 7 34 8 3
5. 22 31 23 21	30. 11 34 12 8	55. 8 34 10 5	80. 7 34 9 4
6. 21 31 22 20	31. 11 34 12 8	56. 8 34 9 5	81. 7 34 8 3
6. 21 31 22 20	31. 11 34 12 8	56. 8 34 10 5	81. 7 34 9 4
7. 21 31 21 19	32. 11 34 12 8	57. 8 34 9 5	82. 7 34 8 3
7. 20 32 21 19	32. 11 34 12 8	57. 8 34 10 5	82. 7 34 9 4
8. 20 32 20 18	33. 10 34 12 7	58. 8 34 9 5	83. 7 34 8 3
8. 20 32 20 18	33. 11 34 12 8	58. 8 34 10 5	83. 7 34 9 3
9. 19 32 20 17	34. 10 34 11 7	59. 8 34 9 4	84. 7 34 8 3
9. 19 32 20 17	34. 10 34 12 7	59. 8 34 9 5	84. 7 34 9 3
10. 18 32 19 16	35. 10 34 11 7	60. 8 34 9 4	85. 7 34 8 3
10. 18 32 19 16	35. 10 34 12 7	60. 8 34 9 5	85. 7 34 9 3
11. 18 32 18 15	36. 10 34 11 7	61. 8 34 9 4	86. 7 34 8 3
11. 18 32 18 15	36. 10 34 11 7	61. 8 34 9 5	86. 7 34 9 3
12. 17 32 18 15	37. 10 34 11 7	62. 8 34 9 4	87. 7 34 8 3
12. 17 32 18 15	37. 10 34 11 7	62. 8 34 9 5	87. 7 34 8 3
13. 16 33 17 14	38. 10 34 11 7	63. 8 34 9 4	88. 7 34 8 3
13. 16 33 17 14	38. 10 34 11 7	63. 8 34 9 4	88. 7 34 8 3
14. 16 33 17 13	39. 10 34 11 6	64. 7 34 9 4	89. 7 34 8 3
14. 16 33 17 14	39. 10 34 11 7	64. 8 34 9 4	89. 7 34 8 3
15. 15 33 16 13	40. 9 34 11 6	65. 7 34 9 4	90. 7 34 8 3
15. 15 33 17 13	40. 10 34 11 7	65. 8 34 9 4	90. 7 34 8 3
16. 15 33 16 12	41. 9 34 11 6	66. 7 34 9 4	91. 7 34 8 3
16. 15 33 16 13	41. 10 34 11 6	66. 8 34 9 4	91. 7 34 8 3
17. 15 33 16 12	42. 9 34 10 6	67. 7 34 9 4	92. 6 34 8 3
17. 15 33 16 12	42. 9 34 11 6	67. 8 34 9 4	92. 7 34 8 3
18. 14 33 15 12	43. 9 34 10 6	68. 7 34 9 4	93. 6 34 8 3
18. 14 33 15 12	43. 9 34 11 6	68. 8 34 9 4	93. 7 34 8 3
19. 14 33 15 11	44. 9 34 10 6	69. 7 34 9 4	94. 6 34 8 3
19. 14 33 15 11	44. 9 34 11 6	69. 8 34 9 4	94. 7 34 8 3
20. 13 33 14 11	45. 9 34 10 6	70. 7 34 9 4	95. 6 34 8 3
20. 14 33 15 11	45. 9 34 10 6	70. 7 34 9 4	95. 7 34 8 3
21. 13 33 14 10	46. 9 34 10 6	71. 7 34 9 4	96. 6 34 8 3
21. 13 33 14 11	46. 9 34 10 6	71. 7 34 9 4	96. 7 34 8 3
22. 13 34 14 10	47. 9 34 10 5	72. 7 34 9 4	97. 6 34 8 3
22. 13 33 14 10	47. 9 34 10 6	72. 7 34 9 4	97. 7 34 8 3
23. 13 34 14 10	48. 9 34 10 5	73. 7 34 9 4	98. 6 34 8 3
23. 13 34 14 10	48. 9 34 10 6	73. 7 34 9 4	98. 7 34 8 3
24. 12 34 13 9	49. 8 34 10 5	74. 7 34 8 4	99. 6 34 8 3
24. 12 34 14 10	49. 9 34 10 6	74. 7 34 9 4	99. 7 34 8 3

This model was also run with various input and target values, and the threshold, start strength, learn rate and rho value adjusted. In each case the results were similar.