Formal Modeling of Connectionism using Concurrency Theory, an Approach Based on Automata and Model Checking*

Li Su, Howard Bowman and Brad Wyble
Centre for Cognitive Neuroscience and Cognitive Systems, University of Kent,
Canterbury, Kent, CT2 7NF, UK
{ls68,hb5,bw5}@kent.ac.uk

Abstract

This paper illustrates a framework for applying formal methods techniques, which are symbolic in nature, to specifying and verifying neural networks, which are sub-symbolic in nature. The paper describes a communicating automata [Bowman & Gomez, 2006] model of neural networks. We also implement the model using timed automata [Alur & Dill, 1994] and then undertake a verification of these models using the model checker Uppaal [Pettersson, 2000] in order to evaluate the performance of learning algorithms. This paper also presents discussion of a number of broad issues concerning cognitive neuroscience and the debate as to whether symbolic processing or connectionism is a suitable representation of cognitive systems. Additionally, the issue of integrating symbolic techniques, such as formal methods, with complex neural networks is discussed. We then argue that symbolic verifications may give theoretically well-founded ways to evaluate and justify neural learning systems in the field of both theoretical research and real world applications.

Key words: formal methods, communicating automata, timed automata, model checking, symbolic, sub-symbolic, neural networks, cognition, learning

* This manuscript is prepared for submission to the Journal of Logic and Computation.

Part 1 Introduction

It is stated in [O'Reilly & Munakata, 2000] that cognitive neuroscience is the study of how cognition is derived from neural physiology. Cognition generally includes attention, perception, memory, language and reasoning [Eysenck,]. Cognitive neuroscience differs from neuroscience because it stresses general principles underlying cognitive phenomena and the neurobiological mechanisms involved. Furthermore these principles can be embedded into models, which are abstractions of such experimentally identified phenomena expressed at a suitable level of biological detail. The models are explicit computational models that help researchers evaluate proposed mechanisms underlying observed human behaviour. Models also help to illustrate the causes of disordered behaviour since modification of the models can help to interpret the underlying mechanisms involved.

There are two major streams of research on cognitive modelling: connectionism and the symbolic paradigms. In this section we introduce both paradigms, point out their limitations for modelling cognitive systems and for real world applications. Finally, we discuss the issue of integrating symbolic computation with connectionism. Note that, in the context of this paper, we use connectionism, neural network and sub-symbolic computation as synonyms.

1.1 Connectionism

In the first part of this section, we briefly explain a popular computational model of cognitive neuroscience: neural networks. In the second part of the section, we point out the limitations of connectionism.

1.1.1 Neural Networks

As recognized in [Zeidenberg, 1990], a neural network is a computational model, which is a directed graph composed of nodes and connections between the nodes. The nodes represent neurons or units. Each node is associated with a number that refers to the activation of the neuron. Another number is also associated with each connection. It is called the weight. Among these nodes, there are some special nodes with their activation set externally. These neurons are called the input neurons. There are some nodes that are distinguished as output neurons. The rest of the nodes are hidden neurons. The behaviour of the network is a function of the activations of nodes and of the weights on the connections.

The activation of each neuron is based on the activation of the nodes that have directed connections to it, the weights on those connections and the neuron's bias. The activations are then updated according to a rule. Therefore many computational models of nerve systems are directly based on such an approach. For example, the membrane potential updating rule of a prominent current connectionist modelling system PDP++ can be found in [O'Reilly & Munakata, 2000]. It is as follows:

$$v_m(t+1) = v_m(t) - dt_{vm}[g_e(t)(v_m(t) - E_e) + g_i(t)(v_m(t) - E_i) + g_l(t)(v_m(t) - E_l)] \qquad (1.1)$$

which is a discrete simplification of the Hodgkin-Huxley equations [Koch, 1999]. Readers should refer to Appendix A for justification of this. In equation 1.1, $v_m(t)$ and $v_m(t+1)$ denote the membrane potential at time t and t+1 respectively. E_e, E_i and E_l denote the reversal potentials of excitatory, inhibitory and leak channels respectively. $g_e(t)$, $g_i(t)$ and $g_l(t)$ denote the excitatory, inhibitory and leak conductance respectively. As it is defined in [O'Reilly & Munakata, 2000], "the time constant $0 < dt_{vm} < 1$ slows the potential change, capturing the corresponding slowing of this change in a neuron primarily as a result of the capacitance of the cell membrane". Note, Equation 1.1 is recursive since the membrane potential is a function of itself. Given a constant input and other parameters, the membrane potential will always settle into a stable value, which is called the equilibrium membrane potential. If we assume that the inputs are presented to the network for a sufficient amount of time in order to allow the membrane potential to settle, we do not have to compute the membrane potential directly from Equation 1.1. Instead, we can compute the equilibrium membrane potential as follows:

$$V_m = \frac{g_e E_e + g_i E_i + g_l E_l}{g_e + g_i + g_l} \quad (1.2)$$

Note, the equilibrium membrane potential is not a function of time. In the rest of the paper, we use V_m instead of computing the membrane potential v_m. This is consistent with the calculation of voltage in [O'Reilly & Munakata, 2000]'s neural network simulations. The kind of computational model we are interested in is a simplification and abstraction of known biology. In addition to Equation 1.2, there are two other equations as follows:

$$g_e = \sum_i (a_i w_{j,i}) + \frac{\beta}{N} \quad (1.3)$$

$$a_j = \frac{1}{1 + e^{\gamma(-V_m + \Theta)}} \quad (1.4)$$

In Equation 1.3, the excitatory input g_e is a weighted sum of all the pre-synaptic activations coming into the neuron. $w_{j,i}$ denotes the weight between post-synaptic neuron *j* and pre-synaptic neuron *i*. β denotes the bias weight, which is divided by the number of inputs *N*. This ensures that the bias weight has the same amount of impact as other inputs. The computation of inhibitory input g_i will be given in the relevant part during the paper. We assume that E_e, E_i, E_l and g_l are constants. Now we can calculate the membrane potential using Equation 1.2. The next step is the calculation of the neuron activation using Equation 1.4, where Θ denotes the nominal threshold, and γ denotes the steepness parameter, which determines the steepness of the output function.

So far we have introduced the updating rule for the activations of each neuron, next we have to address that the activations would be updated individually, thus the neural networks are massively parallel systems. Another aspect of neural networks is their ability to learn, i.e. neural networks adjust their weights according to a learning rule. The aim of learning is to train the network to complete or classify an input pattern or to compute a function of its input. We expect that the network performs badly at the beginning of learning and it performs well afterwards. Neural networks have static architectures and the update and learning rules do not change. Only the weights change and they are not changed after learning unless new knowledge must to be learned. There are a variety of learning rules (e.g. Hebbian Learning (CPCA rule), delta rule, Backpropagation (BP learning) and the generalized recirculation algorithm (GeneRec) [O'Reilly & Munakata, 2000]), which essentially use statistical properties of a set of input/output pairs to generalize. We will explain each of these learning algorithms in the relevant sections.

1.1.2 Limitations of Connectionism

Neural networks are one of the most successful computational models of cognitive neuroscience, especially Parallel Distributed Processing Systems (PDPs) [McClelland & Rumelhart, 1987]. However it has been pointed out by [O'Reilly & Munakata, 2000] that there is much to be discovered about learning, processing, representation in the brain and about how cognition emerges from the underlying neural mechanisms. One issue is the capability of neural networks to handle high-level cognitive tasks, such as natural language processing. Moreover there are many debates on whether neural networks are a suitable representation of the mind. It is argued in [Fodor & Pylyshyn, 1988] that there are three important features lying at the centre of these limitations, which prevent connectionist models from being suitable models of the mind. Thus, from their perspective, connectionism, are best viewed as methods to implement classical cognitive architectures, such as systems based on symbolic paradigms.

- Neural networks have limited compositional structures. It can be seen that connectionist theories acknowledge only casual connectedness, such as activations and weights, which are primitive relations among nodes. If connectionist models do not acknowledge the compositionality of mental representation, they can not explain the systematic and productive features of language and the mind, such as: *Jack shot the deer eating the grass*, which is of the syntactic type $f(a, g(b, c))$. However, classical theories acknowledge not only causal

relations but also structural relations. It is argued in [Fodor & Pylyshyn,1988] that connectionist graphs are not structural descriptions of mental representations but the specifications of causal relations, i.e. an arc from node *A* to node *B* can only refer to the fact that the state of *B* is causally affected by the state of *A*. So it is not true to think that neural networks can represent the mind as a combinatorial constituent structure and a combinatorial semantics.

- Neural networks are not sensitive to structure processing. It is stated in [Fodor & Pylyshyn, 1988] that certain features of the syntactic structures in mental representation correspond systematically to certain of their semantic features. For example: $\neg(P \wedge Q)$ is logically equivalent to $(\neg P \vee \neg Q)$, the operation of distributing negation can be performed if it has the above form and the structural form is the only condition allowing the operation. So one can believe that the correct model of the mind would be able to apply the above inference to any representation with the form $\neg(P \wedge Q)$ and transform it into a representation of the form $(\neg P \vee \neg Q)$. However it is argued by [Fodor & Pylyshyn, 1988] that neural networks do not have such ability because they can not constitute mental processes that are sensitive to the combinatorial structure of the representations on which they operate.

- Finally, it is also difficult to abstract generalized expressions and extract knowledge (rules) from trained neural networks. The most important operation that neural networks have is learning, which is basically a form of statistical modelling. But the structure of the representation of an input is not a factor in determining a corresponding output due to the above limitations. Consequentially, current neural networks are unable to derive an inference through learning, such as: *If Tom jumps, someone jumps*, which is of the syntactic type $f(a) \rightarrow \exists x \cdot f(x)$.

In addition, distributed representations are used in most of the connectionist models, such as PDP++ models. With a distributed representation, a neuron takes part in the representation of a concept or a number of concepts. In contrast, in a localist representation, individual neurons denote specific concepts. A drawback in using PDP++ neural network models to represent the mind is that distributed data in neural networks is hard to interpret. As discussed in [Callan, 1999], in order to learn something new from a trained network or to gain confidence in the network's ability, we should know what knowledge the network is using to perform tasks. In order to view the knowledge, we need to know how the network reorients information. Then there is the question of what it is that individual neurons and weights represent. However, activations in the hidden layer with distributed representations are hardly interpretable and changing the value of a single parameter may have a significant but largely unpredictable effect in the whole network. The difficulty of interpreting distributed data within a neural network creates the problem that researchers may fail to implement a network which should achieve something useful as it was designed to, due to the fact that there are too many parameters which affect the whole network and have extremely complex interdependency. For example, it may take a long time to get the network into a stable state by tweaking parameters if an attracter network falls into an oscillatory behaviour. More seriously, the attractor network may not provide a parameter setting that makes it stable.

1.2 Symbolic Paradigms

There are many symbolic intelligent systems, for example an algorithm that is designed to perform searching, which is a fundamental part of traditional AI. Another class of symbolic system is the rule based production systems such as expert systems and programming languages like Prolog [Bratko, 1986] or Lisp [Norvig, 1992]. There are also some production system based symbolic models of the mind, such as SOAR [Newell, 1990], ACT-R [Anderson, 1993] and EPIC [Meyer & Kieras, 1997]. We are interested in Formal Methods and it is the major symbolic technique we used in the current research to break new ground in cognitive neuroscience. In this part of the section we introduce symbolic processing in general and its limitations in respect of cognitive modelling. In the following section, we also discuss the issue of integrating connectionism with symbolic processing, especially applying Formal Methods to neural networks.

1.2.1 Symbolic Processing

Symbols can be regarded as variables, which can refer to one thing at one moment and another at another. Furthermore, a symbol is a representation of a thing at the conceptual level. Symbolic processing tends to have variables that denote concepts, such as "words". A cognitive model is considered to be symbolic if it acknowledges Newell and Simon's Physical Symbol System Hypothesis [Newell & Simon, 1976]. This hypothesis argues that intelligence is isomorphic with a physical system that manipulates symbols. So intelligent systems, by this definition, explicitly require the following features:

- The systems have elements representing information in the external world.
- Some other elements can manipulate that representation.

This hypothesis states that symbolic processing is the key requirement of high-level cognition. Hence, it is discussed in [Callan, 1999] that traditional AI, such as symbolic paradigms, assumes that it is theoretically possible to construct a mind out of a computer. In order to use computer programmes to derive intelligent systems, research has to account for these topics, i.e. the representation of data, the ability to reason and the ability to learn.

Symbolic systems are good at manipulating, explaining and reasoning about complex data structures. In [Callan, 1999] it is pointed out that there is some attraction to symbolic cognitive systems. Firstly, the knowledge or rules are visible. Secondly, large systems can be composed from small structures. Thirdly, symbolic representations facilitate generalization and the abstraction of knowledge. As we have explained in the previous sections, these points are not true for the mainstream of connectionist models.

1.2.2 Limitations of Symbolic Paradigms

The drawbacks of using symbolic processing to model the mind are as follows:

- The representation of knowledge can be explicit or implicit. Symbolic processing requires knowledge to be represented explicitly using symbol structures. However, understanding how knowledge is represented in the brain is the theoretical basis for symbolic representations. Unfortunately, how symbols emerge from the brain remains unknown. This problem is more significant for representing implicit knowledge. Taking a procedural memory task as an example, it is not possible to produce a document about how to ride a bicycle that a child can read then recall to ride the bicycle successfully. Such knowledge is indeed obtained from experience of training, and it is unclear how such a set of rules could be represented symbolically.

- Symbolic paradigms attempt to use a set of rules to guide programs to perform reasoning thereby handling novel situations and making "rational" decisions. However, a difficulty with this approach is that the knowledge domain to a target problem is sometimes unbounded, namely it is not possible to predict how many rules need to be included. It is hard for a designer to foresee how much knowledge is required and also hard to validate a design that performs adequately. It is even more difficult when we require implicit knowledge to be expressed by symbol structures. Hence, symbolic processing is at present unable to express solutions for all tasks explicitly through the formulation of a suitable computer program.

- As discussed in [Lewis, 1999], symbolic paradigms are distinguished from connectionism because they require that cognitive systems have the ability of composing and manipulating symbols and their structures, extracting novel structures that may be composed and interpreted, and denoting executable processes according to their structures. This hypothesis attempts to explain the emergence of high-level cognition such as rational thinking, logical inference and natural language understanding. However it lacks grounding in biology, namely how symbols and symbolic processing that represents intelligent behaviours emerge from biological systems. It does not take into account how the low-level processes interact with high-level processes.

1.3 Integration of Symbolic and Sub-symbolic Computations

As we have discussed in previous sections, neither symbolic nor sub-symbolic computations are completely appropriate models of the mind. This paper servers as an initial step towards integrating symbolic and neural computation. Our motivations are justified from a cognitive point of view on the one hand, and an engineering application point of view on the other hand.

1.3.1 Cognitive Viewpoint

[Barnard & Bowman, 2004] suggested that one of the motivations of integrating symbolic and sub-symbolic computation is to specify and justify behavior of complex cognitive architectures in an abstract and suitable form.

- Firstly, numerous theories assume that mental modules, or their neural substrates, are processing information at the same time. Hence, any realistic architecture of the mind must be concurrent at some level.

- Secondly, the control of concurrent processing can be distributed. So, cognition should be viewed as the behavior that emerges from interaction amongst independently evolving modules. Most traditional AI approaches fail to acknowledge the requirements of concurrent and distributed control. This is because they tend to be prescriptive and are built around centralized memory and control algorithms [Newell, 1990; Anderson, 1993; Meyer & Kieras, 1997].

- Finally, [Fodor & Pylyshyn, 1988] argued that hierarchical decomposition is needed in order to reflect the characteristics of the mind.

Although connectionist networks are commonly regarded as concurrent and distributed, they are typically limited to only one level of concurrently evolving modules. The primitive elements of neural networks are neurons but not neural networks. To some degree, it is hard to construct and understand large architectures without hierarchical structuring. In certain respects, modeling based on neural networks is low-level in character, i.e. it is hard to relate to primitive constructs and data structures found in high-level notations preferred by the symbolic modeling community. We have stated in the previous sections that a major problem with neural networks is their lack of compositional structure. It has been agued by [Fodor & Pylyshyn, 1988] that compositional structure is one of the key features required in order to represent cognition. For example, box and arrow diagrams are usually used to represent interactions between mental modules or subsystems. The box and arrow diagram representations are essentially compositional structures, i.e. each box represents a mental module, which may be composed to express larger modules.

It has been argued by [Barnard & Bowman, 2000] that computational formalisms and tools are needed for modelling complex mental architectures. Related work has been performed by [Barnard & Bowman, 2000] and [Gibson, 1993] at different levels of abstraction in order to show the power of such formalisms. Although their work has little in common, they both use the same formal language: LOTOS [Bolognesi & Brinksma, 1988]. The former has discussed the relationship between such formalisms and neural networks approaches, particularly the pitfalls of using neural networks to construct big mental architectures such as Interactive Cognitive Subsystems (ICS) [Barnard & Bowman, 2000]. They also provided an illustration of modeling a high-level cognitive architecture using formal methods. Their model contains a set of top-level modules that are connected by communication channels. Modules interact by exchanging data items along channels. Control is distributed and each module evolves independently. They also suggested encoding low-level neural networks using the same method, and formally relating models at different levels of abstraction.

In this paper, key constructs within neural networks are encoded at two levels of description, which have characteristics of symbolic systems. The low-level descriptions use neural networks encoded in communicating automata. We argue that this formalism sits between classical forms of symbolic systems arising from programming languages such as Lisp and Prolog, and connectionist networks. We will explain these models in Part 2 and 3. The high-level descriptions contain a set of properties, which are expressed in logical formulae. They are abstract descriptions of global properties, which do not prescribe internal details of how those properties are realized. We will explain these models in Part 4. Computer scientists have developed a number of theories and tools to automatically justify the relationship between different levels of description within a formal framework. Our models prescribe low-level internal structure, which we hypothesis can be used to explore complex interactions within

neural networks and to justify whether high-level properties can emerge from low-level constructs. Examples of this approach are demonstrated in Part 5.

1.3.2 Application Viewpoint

Symbolic systems are good for manipulating, explaining and reasoning about complex data structures, but neural networks are good at dealing with complex highly non-linear systems, especially in handling catastrophic changes or gradual degradations. It is argued by [Schumann *et al.*, 2003] that neural networks can be applied to extending traditional controllers, which are ineffective in some systems, including aircrafts, spacecrafts, robotics and flexible manufacturing systems. These systems are sometimes called adaptive systems, which mean that their functions evolve over time and they are able to improve their performance through learning. Furthermore, adaptive systems are autonomous, i.e. they have to deal with unforeseen circumstances using local capabilities. Neural network based controllers have demonstrated a superior ability to control adaptive systems, which sometimes includes on-line learning systems [Napolitano et al., 1998]. These controllers are most often used in safety/mission critical domains. However, the correctness of adaptive on-line learning systems must be guaranteed. This is because it is not possible to adapt toward controllable behaviours when the system has changed beyond a critical point. This is also because the system has to dynamically react to changes within short periods of time, e.g. the controller has to output the control signal within the periodical sampling interval. So, this requires that the learning processes converge before a pre-specified deadline.

Unfortunately, the slow speed of learning is one of the greatest limitations of current learning algorithms. For example, the standard BP algorithm often requires the training patterns to be presented hundreds or thousands of times in order to solve a relatively simple task. Furthermore, connectionist networks rarely provide any indication of the accuracy and reliability of their predictions. As long ago as 1988, [Fodor and Pylyshyn, 1988] pointed out that the neural networks approach remained almost entirely experimental. Although a great deal of mathematical work has been done, it is still not sufficient from the analytical point of view to justify that certain configurations of neural networks and their mechanisms are reliable. It is also critical to provide safety, especially stable, guarantees to learning algorithms. Integrating neural networks with symbolic paradigms, e.g. traditional AI, Predicate Calculus or Formal Methods, could overcome these issues, thus in the latter part of this work we try to apply a strongly mathematically based technique, Formal Methods, to encoding and analyzing neural networks. Why we have chosen this approach will become clear shortly.

1.4 Formal Methods and Model Checking

Following the discussion in [Ibrahim et al, 1990], a specification expresses the functional and non-functional requirements of a system, which could be a piece of software or a neural network. It states what the target system is intended to achieve and the attributes or features the system must possess. Such specifications are used for validation and verification purposes. It is validated to assure that the specification represents the intended use of the system and that the system as specified will meet its objectives. Hence, the specification forms a basis for developing a design of a system that can be verified against the validated specification.

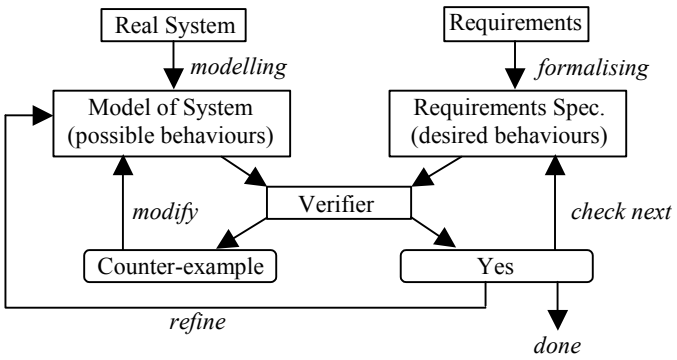


Figure 1 Model Checking

In resent years, model checking has become a useful technique for verifying properties of finite state concurrent systems, i.e. systems in which the variables range over finite domains. The methodology of

traditional model checking is presented schematically in Figure 1. The real systems such as neural networks or cognitive systems modelled in other forms are modelled using a formal language, which describes the possible behaviour of the system. The initial model may not be accurate. We also propose a set of requirements of the system. They are expressed in logical formulae. Computer scientists have developed a number of theories and tools, such as verifiers. The model and requirement specifications are inputted to the verifier. The verifier automatically assesses the relationship between them and outputs either a positive or a negative result, if there are sufficient resources. The verifier also has the facilities for generating traces, such as witnesses/counter-examples.

One question is, can we use formal methods and their rich tool support to develop computational models of neuroscience from the scope of a single neuron to high-level cognitive mechanisms. Moreover is it possible to apply automatic checking and analytical methods such as model checking to understand the complicated behaviour of neural networks? Another question is, can we develop methods, which integrate the advantages of both formal methods and conventional techniques used by neuroscientists, such as box and arrow diagrams and neuron networks. The aim of this paper is not to solve any of these questions completely, but to discuss the possible ways of looking at these questions and determining what is the most critical area, where we need to integrate different techniques.

As a preliminary assessment of our approach, this paper also describes a study, which applies formal methods techniques to evaluating learning using automatic analysis, i.e. model checking. Formal methods are strongly based on logic and they have rich tool support. Formal methods have shown their power in software engineering and various areas, where correctness and effectiveness of computer systems need to be guaranteed. So they can, for example, be used in designing distributed systems [Wu, Chanson & Gao, 1999] and [Bowman & Derrick, 2001], electronic hardware [Silver & Brzozowski, 2003] and traffic signal systems [Damm, 2001]. In these areas, there are similar problems and requirements in respect of modelling complex interactions among components with distributed control. Another advantage of using model checking to validate and verify on-line adaptive systems is that the model checker is autonomous. So, it can be applied along side the learning systems without human interaction. The learning systems may encounter unforeseen circumstances. Hence, any kind of prior verification may be infeasible. The model checker could validate the parts of the system, which will be affected by adaptive learning before the actual learning is committed. We will demonstrate a simplified model in Part 5.

There are also many papers [Fodor & Pylyshyn, 1988] and [Callan, 1999] about the compositionality of neural networks and recursive auto-associative memories (RAAM) [Pollack, 1990]. They consider one or both of the two questions mentioned above. There is also work on integrating neural networks and symbolic rules. In [Garcez, Broda & Gabbay, 2002], symbolic systems provide the neural networks with the background information needed for learning, and the information embedded in neural networks is extracted in symbolic forms. In their work, logic is regarded as a very useful tool in helping to explain a neural network's inference process and in formalising their learning and generalisation mechanisms. A similar framework to our approach was presented by [Smith, 1992] in a general mathematical setting. But his work did not consider automatic simulation or verification.

Part 2 Communicating Automata Model of Neural Networks

In this section we present a general framework for modelling neurons. When modelling cognitive systems, abstractions are highly encouraged to just capture the most significant elements in the real system and omit implementation and biological details. This kind of abstraction is also a basic method in any science including computer science and neuroscience.

In order to model a neural network, it is essential to specify the behaviour of a single neuron. We have chosen formal methods, especially communicating automata to specify neural networks. The system is a composition of communicating automata. Accordingly, a single neuron is modelled as an automaton. This model also needs to be general enough in order to capture the most important elements of a neuron. This formalism of connectionist networks is a symbolic system, which can be used to understand connectionist systems. The models are formalisations of neuron networks, which can be used to see whether this abstraction can perform the computations that real neuron networks can perform.

2.1 Communicating Automata and Notations

Before we present the model, we introduce the specification notation used. It contains features of timed automata [Alur & Dill, 1994], along with features from classic communicating automata, such as data passing communication channels. This kind of notation is a form of communicating automata, which has data passing actions and mathematical equations associated with transitions. This notation includes signatures, which define the data structures of the automata, and a behaviour expression, which is represented in graphs like timed automata. We will explain the graphs at relevant parts during the section.

In the context of this paper, a communicating automaton is a tuple $(L,\bar{l},T)$, where:

L defines a finite set of locations, which are denoted as nodes in diagrams.

$\bar{l}$ is an initial location, which is denoted as a node with a smaller circle inside, see the *Input* location in Figure 2.

T is a transition relation, elements of which have the form $(l,g,a,\mathcal{E},l')$, in which $l,l' \in L$, g is a guard, $a \in \overline{A}$ is an action, $\overline{A} = \{\tau\} \cup (\mathcal{G} \times \overline{\mathcal{D}})$, which defines the set of all data passing declarations, where $\overline{\mathcal{D}} = \mathcal{D}^*$, i.e. all tuples that can be generated from elements of $\mathcal{D}$. $\mathcal{E}$ is an effect set. When guard g holds, action a can be performed and if it is performed, assignments in set $\mathcal{E}$ will be made. a can be either a communication action or an internal action, denoted τ. Communication actions have the following general form:

$$e\,\overline{d}_m,$$

where e is a gate name, i.e. $e \in \mathcal{G}$, $\mathcal{G}$ denotes the set of all gate names, $\overline{d}_m = \langle d_1 \cdots d_m \rangle$ is a vector of data passing declarations. There are two types of declarations: a value declaration and a variable declaration, as follows:

$$\mathcal{D} = Value\_Declaration \cup Variable\_Declaration.$$

A value declaration is defined as the following:

$$Value\_Declaration : !Ex,$$

where Ex is a value expression, e.g. $!(5+3)$, $!(y \times 6)$ and $!(true)$.

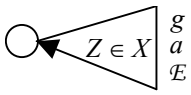
A variable declaration is defined as the following:

$$Variable\_Declaration : ?x ,$$

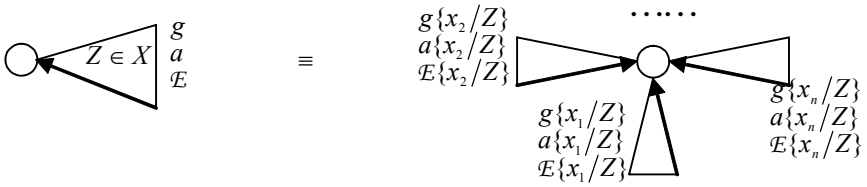
where $x \in Name$ is a name of a variable and *Name* represents all the names of variables appearing in the network.

Note that communication actions are similar to value passing actions in LOTOS [Bolognesi & Brinksma, 1988], but we do not introduce new variables in actions. Data and their types have to be defined in the data declaration. So unlike value passing actions in LOTOS, communication actions in our notation do not reference types.

The notation we use to describe our communicating automata contains a number of conventions. In particular, we use a Parameterised Loop notation, i.e.



where $X = \{x_1, x_2, \cdots, x_n\}$, g is a guard, a is an action, $\mathcal{E}$ is an effect and each of g, a and $\mathcal{E}$ typically contain references to Z. The above notation denotes that there are n self loops; each of them is associated with an instantiation of the variable Z with an element from the set X. In addition, the scope of Z only applies to this self loop and expressions associated with this self loop, such as g, a and $\mathcal{E}$. Thus the following are equivalent.



where $t\{x/y\}$ denotes a substitution, which replaces all occurrences of y in the expression t by x. We will give semantics to the communicating automata after an example, which is shown in the next section.

2.2 Neuron Automaton

In this section, we explain one example of how to model neural networks, inspired by [O'Reilly & Munakata, 2000], using communicating automata. This model is biologically plausible in one important sense, i.e. the neurons are automata and control of the network is distributed into each individual neuron rather than centralized. This is in contrast with traditional neural network simulations, which act over a globally accessible model of the entire network. In other senses, this model is biologically plausible for the following reasons. The excitatory and inhibitory inputs are explicitly separated and weights do not change sign during learning.

2.2.1 Global Declarations

Global declarations define the set of global constants and variables. In our communicating automata model of neural networks we define set G containing global constants that are shared by the whole network. Set G is defined as follows:

$$G ::= \{Ee, Ei, El, gl, \Theta : \mathcal{R}_0^1, \gamma : \mathcal{N}^+\} ,$$

where Ee is the reversal potential of excitatory channels, Ei is the reversal potential of inhibitory channels, El is the reversal potential of leak channels, gl is the leak conductance, Θ is the nominal firing threshold, γ is the steepness parameter, which determines the steepness of the output function, $\mathcal{R}_0^1$ denotes real values in the range 0 to 1, and $\mathcal{N}^+$ denotes the positive natural numbers. There are no global variables in this model. Explanation of these parameters has been presented previously.

2.2.2 Neuron Automaton Definition and Local Variable Declarations

The signature or the "header" of a neuron automaton is defined as follows:

$$NEURON[C, V, E],$$

where C, V, E are three sets of arguments for this automaton. This expression is called the signature of the neuron. We will explain each of these sets in detail.

Set *C* is the set of variables that are evaluated at the construction stage of each neuron automaton. Values in this set can vary between neurons but they can not be changed in the lifetime of each neuron. The lifetime of a neuron process starts from the construction of the process to the termination of the process, also any learning stage that might be included. Set *C* is defined as follows:

$$C ::= \{k : Id, preIds, postIds : \mathcal{P}(Id)\},$$

where *k* is the identity of this neuron automaton. The type of the identity is *Id*, and neuron identities are assumed to be unique. *preIds* is the set of pre-synaptic neuron identities and *postIds* is the set of post-synaptic neuron identities. $\mathcal{P}(Id)$ denotes the power set of *Id*.

Set *V* is the set of variables that are local to each neuron automaton. Values in this set are subject to change over time. These variables are key to describing the dynamics of the neuron. Set *V* is defined as follows:

$$V ::= \{in : Id \to \mathcal{R}_0^1, w : Id \times Id \to \mathcal{R}_0^1, \beta, ge, gi, Vm, h : \mathcal{R}_0^1\},$$

where *in* is a function, the domain of *in* is the set *preIds*, for example, in_{pre_sy} would denote the pre-synaptic activation with respect to a pre-synaptic identity $pre\_sy \in preIds$. *w* is also a function, for example, w_{k,pre_sy} would denote the weight with respect to the identity of this neuron *k* and a pre-synaptic neuron identity *pre_sy*. β is the bias weight that denotes the baseline differences in excitability between different neurons, *ge* is the excitatory input conductance, *gi* is the inhibitory input conductance, *Vm* is the equilibrium membrane potential and *h* is the activation of this neuron. It has been discussed previously that we do not compute the membrane potential directly using Equation 1.1. We use the equilibrium membrane potential in Equation 1.2 instead, because we assume that the inputs are presented to the network for a sufficient amount of time in order to allow the membrane potential to settle.

There are other variables that are used in the automaton but they do not relate to neurobiology. These variables are included because of the requirements of modelling; therefore the elements may be different between implementations. We introduce them in the set *E*, which is defined as follows:

$$E ::= \{tempIds : \mathcal{P}(Id), pre\_sy, post\_sy, i : Id, \cdots\},$$

where *tempIds* is a set of identities used temporally, it is the set of unprocessed pre-/post-synaptic neuron identities. *pre_ sy* is the identity of a specific pre-synaptic neuron and *post_ sy* is the identity of a specific post-synaptic neuron. *i* is used together with quantification. It should be noticed that the specification of this set is not completed because extra elements can be added in order to enable additional functionality to be incorporated, such as learning or other mechanisms, which can be seen later.

2.2.3 The Graph Illustrations

The behaviour expressions of communicating automata are defined in graphs. A neuron automaton can be seen in Figure 2. Initially the arguments are set to be as follows: (Most of these figures are employed in [O'Reilly & Munakata, 2000].)

- k can be assigned to any identity as long as it is distinct from any other neuron;
- *preIds* and *postIds* depend on the connectivity of the network;
- Ee, Ei and El are 1.00, 0.15 and 0.15 respectively;
- gl can be initialised as required in specific models;
- Θ is set to be 0.25 in most of the models;
- γ is 10 for membrane potentials in the range between 0 and 1;
- initially, $\forall i \in preIds \cdot in_i := 0$;
- in most of the models, w_{k,pre_sy} and β are initially randomly distributed around 0.5, which allows the learning rule to effectively focus learning on those neurons which are more labile. These neurons are in the sensitive middle range of the sigmoid function. Consequently, a weight change will have a dramatic effect [O'Reilly & Munakata, 2000];
- ge, gi, Vm and h are zero initially, however it is more plausible to set Vm initially to the resting potential, which has the value of 0.15;
- initially, $tempIds := preIds$; and
- pre_sy and $post\_sy$ are zero initially.

As is shown in Figure 2, the neuron automaton has three locations: *Input*, *Middle* and *Output*. In the initial location, *Input*, a neuron gets all inputs from its input channels. We express value passing actions as follows:

$$port!pre\_sy!k?in_{pre\_sy},$$

where *port* is the gate name, $!pre\_sy$, $!k$ and $?in_{pre\_sy}$ are data passing declarations. The data declarations identify the nature of the communication to be made, i.e. working from left to right, the first declaration denotes the pre-synaptic neuron with respect to this communication, the second declaration denotes the post-synaptic neuron and the third declaration receives the activation value transmitted over the link. In this particular case, $!pre\_sy$ and $!k$ are value declarations, but $?in_{pre\_sy}$ is a variable declaration. Value declarations in this channel are used to achieve value synchronisation, i.e. interactions are only allowed in which pre_sy (the identity of the pre-synaptic neuron) and k (the identity of this neuron) are in the appropriate slots. Then in_{pre_sy} is bound to the appropriate value of the matched pre-synaptic neuron's activation. This is called value passing. (We will come back to this point when we explain the *Output* location, where this value synchronisation and value passing will become clearer.) The associated effect is an assignment, that is,

$$tempIds := tempIds - \{pre\_sy\},$$

which indicates the completion of getting input from the pre-synaptic neuron with identity pre_sy, thus this identity can be subtracted from the set *tempIds*. Once inputs have been made from all pre-synaptic neurons, the following guard will hold.

$$tempIds = \phi$$

Thereby, the neuron automaton takes a transition, computes the excitatory conductance ge and the membrane potential Vm and then moves to the next location *Middle*. The computations of ge and Vm are taken form [O'Reilly & Munakata, 2000]. They are as follows:

$$ge := \sum_{i \in preIds} (in_i \times w_{k,i}) + \frac{\beta}{\#\, preIds} \text{ and} \qquad (2.1)$$

$$Vm := \frac{ge \times Ee + gi \times Ei + gl \times El}{ge + gi + gl}. \qquad (2.2)$$

From Equation 2.1, we can see that the products of activations and weights are computed post-synaptically. If X is any finite set, $\#X$ is a natural number denoting the cardinality of, i.e. the number of elements in, X. The gi in the function is reserved for future use, it is set to zero in this model. Finally, *tempIds* is assigned to *postIds*.

It is also important to know that the main function of a dendritic tree is to integrate information from its pre-synaptic neurons. The integration can be regarded as two basic simultaneous processes, i.e. space integration and time integration. Space integration is that the dendrite summarises the inputs from different pre-synaptic neurons at a given moment, while time integration deduces the result of space integration according to a function of time, typically through time averaging. The model we present ignores time integration. The neuron only "receives" its input at particular points of time, i.e. the moments that two neurons interact via value passing channels. Time integration could be modelled using timed automata, it would increase the state space of the system dramatically and automatic analysis is likely to be intractable.

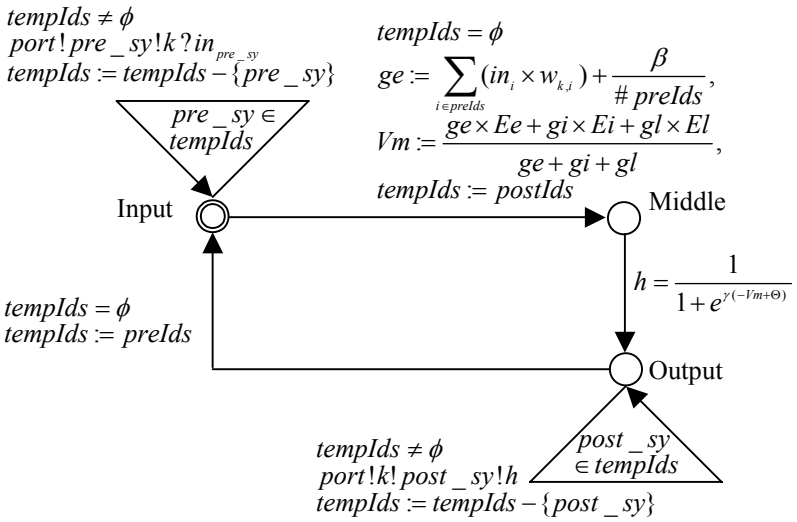


Figure 2 A General Neuron Automaton

The activation h is evaluated using a soft threshold sigmoid non-linear function, as follows:

$$h = \frac{1}{1 + e^{\gamma(-Vm+\Theta)}}. \quad (2.3)$$

This function is illustrated in Figure 3.

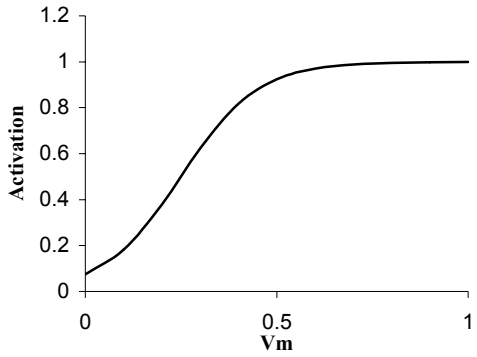


Figure 3 The Soft Threshold Sigmoid Non-linear Function, $\Theta = 0.25, \gamma = 10$

In the next location, *Output*, the neuron sends its activation through output channels. We express the value passing action as follows:

$$port!k!post\_sy!h,$$

Where, as was the case previously, *port* is the gate name, $!k$, $!post\_sy$ and $!h$ are data passing declarations. In this particular case, $!k$, $!post\_sy$ and $!h$ are all value declarations. The value declaration of $!post\_sy$ in this channel is used to achieve value synchronisation, i.e. only the matched post-synaptic neuron that has the identity $post\_sy$ can enable interaction on this channel. Through this synchronization, the value of *k* will be bound to the *pre_ sy* value in the matched post-synaptic neuron and *h* will be bound to in_{pre_sy} in the post-synaptic neuron's input. After passing the values, the identity *post_ sy* is subtracted from the unprocessed identities set *tempIds*, in order that activation is sent to a different post-synaptic neuron on the next cycle through the self-loop. However, the guard:

$$tempIds \neq \phi$$

will be *false* when it finishes outputting to all its post-synaptic neurons. Then it will not be able to perform the self loops any more and will take the alternative transition. The guard of this transition,

$$tempIds = \phi$$

holds at this point of time. Taking this transition involves assigning to the set *tempIds*:

$$tempIds := preIds$$

and returns the neuron automaton to the initial location, *Input*.

2.3 Semantics

We assume our system of neural networks is described as a vector of communicating automata, which we semantically interpret as a labelled transition system $(S, s_0, \rightarrow)$, where

- $S \subseteq (L_1 \times \cdots \times L_i \times \cdots \times L_n) \times \Sigma$ defines the set of states in the labelled transition system. L_i indicates the set of locations of the automaton *i*, and Σ denotes the set of all possible global data states. Explicitly, a state is represented as $\overline{V}_{n,\sigma}$, which denotes a vector of *n* locations, i.e. $\langle l_1, \cdots, l_n \rangle$, applied to σ, $\sigma \in \Sigma$. Thus $\overline{V}_n$ is of the form $\langle l_1, \cdots, l_n \rangle$ and a global state is indicated by the locations of each automaton and a global data state.

- $s_0 = (\langle \bar{l}_1, \cdots, \bar{l}_i, \cdots, \bar{l}_n \rangle, \bar{\sigma})$ defines the initial state. $\bar{l}_i$ indicates the initial location of the automaton *i* and $\bar{\sigma}$ indicates the initial data state.

- $\rightarrow \subseteq S \times \bar{A} \times S$ is the transition relation defined by the following inference rules:

  Internal action rule:

$$\frac{(i,l) \xrightarrow{g,\tau,\mathcal{E}} (i,l') \quad g(\sigma)}{\bar{V}_{n,\sigma} \xrightarrow{\tau} \bar{V}_{n,\mathcal{E}(\sigma)}^{i \to l'}}, \text{ where}$$

  - (i,l) indicates that automaton *i* is in location *l*;
  - guards are interpreted as predicate functions, i.e. $g : \Sigma \rightarrow Boolean$;
  - τ is the internal action;
  - effects are interpreted as functions, i.e. $\mathcal{E} : \Sigma \rightarrow \Sigma$;
  - $\bar{V}_n^{i \to l'}$ defines a substitution notation as follows:

$$\langle l_1, \cdots, l_i, \cdots, l_n \rangle^{i \to l'} = \langle l_1, \cdots, l', \cdots, l_n \rangle .$$

  The consequence of substitution is that the automaton *i* is in location l'.

  Communicating action rule:

$$\frac{\begin{array}{c}(i,l) \xrightarrow{g, e\bar{d}_m, \mathcal{E}} (i,l') \quad (j,r) \xrightarrow{g', e\bar{d}'_m, \mathcal{E}'} (j,r') \\ i \neq j \quad g(\sigma) \quad g'(\sigma) \quad \bar{u}_m = \overline{eval}(\bar{d}_m, \sigma) \quad \bar{u}'_m = \overline{eval}(\bar{d}'_m, \sigma) \quad \bar{u}_m \doteq \bar{u}'_m\end{array}}{\bar{V}_{n,\sigma} \xrightarrow{e.\Omega(\bar{u}_m, \bar{u}'_m)} \bar{V}_{n,\mathcal{E}(\mathcal{E}'(\propto(\bar{u}_m, \bar{u}'_m, \sigma)))}^{i \to l', j \to r'}}$$

  Where:

  $e\bar{d}_m$ denotes a gate name $e \in \mathcal{G}$ and a vector of data passing declarations $\bar{d}_m = \langle d_1, \cdots, d_m \rangle$;

  $\overline{eval} : (\mathcal{D} \times \ldots \times \mathcal{D}) \times \Sigma \rightarrow Partial\_Value\_Tuple$, where $\mathcal{D}$ denotes data passing declarations, which was defined previously and $Partial\_Value\_Tuple$ is the set of all partially evaluated tuples, where elements of the tuples are either variable declarations or values, thus

$$\overline{eval}(\langle d_1, \cdots, d_m \rangle, \sigma) = \langle eval(d_1, \sigma), \cdots, eval(d_m, \sigma) \rangle , \text{ where}$$

$$eval(!Ex, \sigma) = Ex(\sigma) \text{ and}$$
$$eval(?x, \sigma) = ?x ,$$

  where expressions are interpreted as functions from global data states to values, i.e. $Ex : \sigma \rightarrow value$.

  $\doteq$: $Partial\_Value\_Tuple \times Partial\_Value\_Tuple \rightarrow Boolean$, thus partial value tuples:

$$\langle u_1, \cdots, u_m \rangle \doteq \langle u'_1, \cdots, u'_m \rangle \text{ iff}$$
$$\forall j (1 \leq j \leq m) \cdot (\neg(u_j = ?x \wedge u'_j = ?y) \wedge ((u_j \neq ?x \wedge u'_j \neq ?y) \Rightarrow u_j = u'_j)) ;$$

Ω: $Partial\_Value\_Tuple \times Partial\_Value\_Tuple \to Value\_Tuple$,
where $Value\_Tuple = (value \times \ldots \times value)$ and:

$\Omega(\langle u_1, \cdots u_m \rangle, \langle u'_1, \cdots u'_m \rangle) = \langle get\_val(u_1, u'_1), \cdots, get\_val(u_m, u'_m) \rangle$, where

$$get\_val(v, u) = v \text{ and}$$
$$get\_val(?x, v) = v;$$

$\propto$: $Partial\_Value\_Tuple \times Partial\_Value\_Tuple \times \Sigma \to \Sigma$, thus

$$\forall x \in Name \cdot \propto (\langle u_1, \cdots u_m \rangle, \langle u'_1, \cdots u'_m \rangle, \sigma)(x)$$
$$= \begin{cases} v, & if\ \exists j (1 \le j \le m) \cdot ((u_j = ?x \wedge u'_j = v) \vee (u_j = v \wedge u'_j = ?x)) \\ \sigma(x), & otherwise \end{cases}$$

Part 3 Timed Automata Models of Neural Networks

In this section we change the notation from communicating automata to timed automata, which is necessary to enable simulation and verification in Uppaal. These two notations differ in their handling of value passing channels, which Uppaal timed automata do not support [Pettersson, 2000]. They also differ in the data types provided, i.e. Uppaal does not provide real numbers. In this section, leaning algorithms such as the CPCA rule, delta rule, BP learning and GeneRec algorithm [O'Reilly & Munakata, 2000] are explained and applied in these models. In addition, the environment of neural networks and kWTA inhibition are also explained and applied in the relevant models. These models are directly derived from the general neuron automaton that was presented in previous sections. Although these models are composed of neuron-like automata and other automata defining other mechanisms, it is not necessary to specify every system using neuron level units. We also introduce some additional notation for this section.

- All Bold variables, e.g. $\boldsymbol{w}$, $\boldsymbol{test}$ and $\boldsymbol{a}$, represent global variables, which are visible to all automata. These variables are used for value passing between automata, and are employed because timed automata lack data passing channels. Variables written in other fonts are local variables that are only visible to each neuron automaton.

- The expression: $port_Z?$ (or $port_Z!$) is a half action, which denotes a point-to-point channel. See [Pettersson, 2000] for a definition of half actions. We use the notation: $\boldsymbol{port}_Z?$ (or $\boldsymbol{port}_Z!$) for a broadcast channel. Broadcast channels perform one-to-all communication, i.e. when a half action $\boldsymbol{port}_Z!$ is offered, it is available to all the automata in the system. Hence, all the half actions $\boldsymbol{port}_Z?$ are enabled.

- The following defines the meaning of a bounded quantification.

$$\forall i \le N \cdot P \equiv P\{1/i\} \wedge P\{2/i\} \wedge \cdots \wedge P\{N/i\},$$

  where P could be a predicate or an assignment. For example, the expression: $\forall i \le N \cdot take_i$ is a guard, which denotes:

$$take_1 \wedge take_2 \wedge \cdots \wedge take_N,$$

  where $take$ is a function, which returns a Boolean. Another example is a set of assignments: $\forall i \le N \cdot take_i := B$, which denotes the following:

$$take_1 := B \wedge take_2 := B \wedge \cdots \wedge take_N := B,$$

  where B is a Boolean.

- We can also define bounded existential quantification from bounded universal quantification in the usual way:

$$\exists i \le N \cdot P \equiv \neg \forall i \le N \cdot \neg P.$$

3.1 Value Passing and Data Types

3.1.1 Value Passing Channels

It is important to understand how neurons communicate in this model. The timed automata notation does not support value passing channels, so we implement communication using the following methods:

- Each neuron in the network has associated with it a number $\boldsymbol{a}_k$, which refers to the activation of the neuron. This is consistent with the definition of neural networks we explained in the

previous section. These numbers are global to all neurons, however the retrieval of activation is restricted by synchronization. Taking the automaton in Figure 4 as an example, the neuron has to communicate with all its pre-synaptic neurons via channels $\boldsymbol{port}_Z?$ in order to satisfy the precondition in the guard $\forall i \in preIds \cdot take_i$, then it retrieves pre-synaptic activations and computes the overall excitatory input.

- On one hand, the neuron automaton offers $\boldsymbol{port}_k!$ when it outputs. On the other hand, a neuron automaton gets input using parameterised loops, which offer a generalised choice to communicate with different pre-synaptic neurons. At each choice, the neuron can communicate on a broadcast channel $\boldsymbol{port}_Z?$ where ***port*** corresponds to the gate name and Z corresponds to the pre-synaptic identity. Note that the post-synaptic identity is omitted in the communication channel because the broadcasting channel is visible to all neurons.

- Boolean variables $take_Z$ are tokens that are used as an alternative to the set subtraction mechanism used in the previous model. When receiving a pre-synaptic activation, the identity of the pre-synaptic neuron is then subtracted from *tempIds* (which is a copy of the pre-synaptic identities) in the previous model. In this model, a token is set when a pre-synaptic activation is offered. In the previous model, the input terminates when *tempIds* is empty, whereas in this model, it terminates when the neuron has all the tokens.

3.1.2 Data Types

Natural and real numbers are infinite sets, so systems containing natural and real variables can have an infinite number of states. These systems cannot be model checked, hence formal specification languages, such as, Uppaal timed automata, typically only provide bounded integer values. As a result, we use integer values to express weights, activation and other variables in the neural network. So, in the signature of neurons, the types $\mathcal{N}^+$ and $\mathcal{R}_0^1$ are replaced by the type $\mathcal{N}_0^\psi$. This denotes integer numbers ranged between 0 and ψ. The latter of these being the largest integer a system can offer. (In practice, we can choose a suitable number. It is set to 1000 in most of our models, which provides enough precession for the learning algorithms used.) Note that $\mathcal{N}_0^\psi$ is a finite set. If we restrict all data structures of our model to finite sets, we can define a system that has a finite number of states. Consequently, such systems can be model checked. However, there is the problem that an integer greater than 1 is not a unit of multiplication. For any two integers that are both bigger than 1, the product of the two is bigger then either. Therefore, we need to normalise the results after each multiplication, i.e. divide the result by ψ in order to keep the range of the result between 0 and ψ. For simplicity of presentation, we will omit the normalisation in the graph illustrations. In addition, Uppaal timed automata only provide array structures rather than sets. However, it is straightforward to map our sets into Uppaal's arrays. See Appendix D for the formal definition of the number representations.

3.2 Global Declaration and Neuron Automata

In this model, the global declarations consist of a set of constants G and a set of global variables $\boldsymbol{V}$:

$$G ::= \{Ee, El, gl, \Phi, \Gamma : \mathcal{N}_0^\psi\} \text{ and}$$
$$\boldsymbol{V} ::= \{\boldsymbol{a} : \mathcal{N}_0^\psi \to \mathcal{N}_0^\psi, \boldsymbol{w} : \mathcal{N}_0^\psi \times \mathcal{N}_0^\psi \to \mathcal{N}_0^\psi\}.$$

Elements in set G will be explained when they appear in the relevant equations. In set $\boldsymbol{V}$, $\boldsymbol{a}$ is a function, for example, $\boldsymbol{a}_i$ would denote the pre-synaptic activation with respect to a pre-synaptic identity $i \in preIds$, where $preIds \subset \mathcal{N}_0^\psi$ by definition, and $\boldsymbol{a}_k$ denotes the activation of this neuron (see signature below). $\boldsymbol{w}$ is a function, for example, $\boldsymbol{w}_{k,i}$ denotes the weight with respect to this neuron identity k and a pre-synaptic neuron identity i.

The signature of a single neuron is as follows:

$NEURON[C, V, E]$, where

$C ::= \{k : \mathcal{N}_0^{\psi}, preIds : \mathcal{P}(\mathcal{N}_0^{\psi})\}$,

$V ::= \{\beta, ge, Vm : \mathcal{N}_0^{\psi}\}$ and

$E ::= \{take : \mathcal{N}_0^{\psi} \rightarrow Boolean, Z, i : \mathcal{N}_0^{\psi}\}$,

where similarly to the previous section, *k* denotes the identity of this neuron automaton, all neurons are uniquely identified. *preIds* is a set of pre-synaptic neuron identities. Elements in set *V* have been explained previously. In set *E*, *take* is a function; for example, $take_i$ indicates the reception of pre-synaptic activation with respect to a pre-synaptic identity *i*, thereby its domain is *preIds*.

An automaton model of a single neuron is shown in Figure 4. In this model, the neuron automaton has three locations: *Input*, *Middle* and *Output*. In the initial location, *Input*, a neuron is waiting for all its inputs to be ready. There are $\#preIds$ pre-synaptic neurons. When the half action $\boldsymbol{port}_Z?$ is offered, the neuron's *Z*th input is ready. When an input is ready, the neuron produces a token by setting the Boolean variable $take_Z$ to *true*. The parameterised loop does not prevent the neuron from signalling availability of the same pre-synaptic activation twice, this is not a problem since the newly set activation will simply overwrite the previous one. However, the token mechanism prevents the neuron from progressing before all pre-synaptic neurons have set their activations at least once.

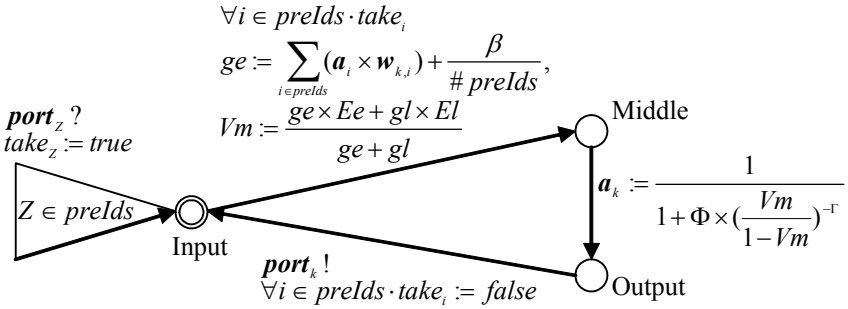


Figure 4 A Neuron Automaton, which Preserves Distributed Control

There is a transition between the *Input* and *Middle* locations. Once the neuron collects all its tokens, in other words, all its inputs are ready, the guard $\forall i \in preIds \cdot take_i$ will hold. The neuron then computes its overall excitatory input conductance ge, which is a sum of the product of its pre-synaptic activations and corresponding weights. This is based on Equation 2.1 and it is shown in Figure 4 as the following:

$$ge := \sum_{i \in preIds} (\boldsymbol{a}_i \times \boldsymbol{w}_{k,i}) + \frac{\beta}{\#preIds}. \qquad (3.1)$$

After computing the overall input conductance ge, the equilibrium membrane potential Vm can be evaluated. The function that determines Vm is based on Equation 2.2 with $gi = 0$ (since we do not consider inhibition in this model) and it is shown in Figure 4 as follows:

$$Vm := \frac{ge \times Ee + gl \times El}{ge + gl}. \quad (3.2)$$

Having taken this transition, the neuron goes to the next location, *Middle*.

In the *Middle* location, the neuron is ready to compute its output. The activation is computed by a soft threshold sigmoid non-linear function, shown in Figure 4 as the following:

$$\boldsymbol{a}_k := \frac{1}{1 + \Phi \times (\frac{Vm}{1 - Vm})^{-\Gamma}}, \quad (3.3)$$

which is an approximation of the previous sigmoid function, i.e. Equation 2.3. We are not able to implement Equation 2.3 in Uppaal because it does not provide real numbers. In Equation 3.3, Φ denotes the gain parameter, Γ denotes the steepness parameter and variable $\boldsymbol{a}_k$ is the output activation of this neuron. We assume that the output activation is 0.5 when $V_m = \Theta$. So, let $\boldsymbol{a}_k = 0.5$ and $V_m = \Theta$ in Equation 3.3, we are able to give the relation between Φ and Θ as the following:

$$\Theta = \frac{\Phi^{\frac{1}{\Gamma}}}{1 + \Phi^{\frac{1}{\Gamma}}} \qquad (3.4)$$

In Figure 5, we plot a set of sigmoid functions using the same steepness parameter $\Gamma = 3$ but different gain parameters, which are shown in the legend. Table 1 maps the gain parameters to thresholds when $\Gamma = 3$.

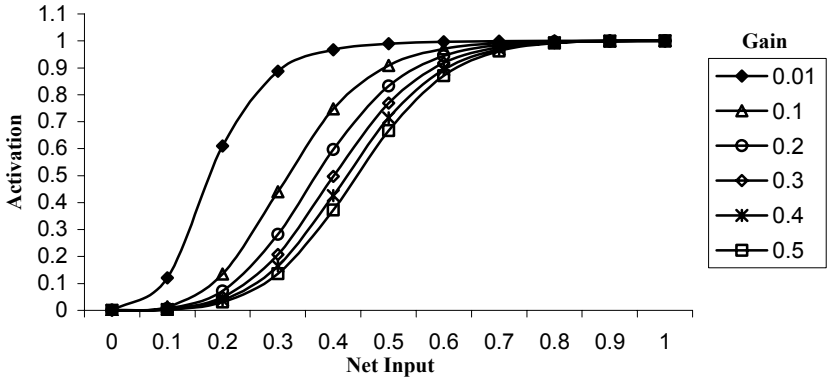


Figure 5 Sigmoid Functions with Different Gain Parameters

Gain Parameter Φ	0.01	0.1	0.2	0.3	0.4	0.5
Threshold Θ	0.177	0.317	0.369	0.401	0.424	0.442

Table 1 Mapping Between the Gain Parameters and the Thresholds, $\Gamma = 3$

From the next location, *Output*, the neuron takes a transition, sends a signal to its post-synaptic neurons via a broadcast channel $\boldsymbol{port}_k!$, which indicates its activation has been computed and can be used, it also resets its tokens and returns to the initial location. Based on this simple model, more complicated networks can be specified, for instance, learning, inhibitory competition and bi-directional activation can be added. In the following subsections we present neural network models that use different learning rules.

3.3 the Conditional Principal Analysis (CPCA) Rule

The conditional principal components analysis rule is a Hebbian form of model learning, which is based on pre/post-synaptic neural activation. Model learning attempts to develop an internal model of the world or the environment of the network. The biological mechanisms underlying model learning are long-term potentiation (LTP) and long-term depression (LTD) [Bliss & Lomo, 1973]. However, details of the biology are not presented in our computational model. The result of model learning is that neurons end up representing correlations, in other words, the general statistical structure in the environment. CPCA also requires a conditionalizing function [O'Reilly & Munakata, 2000]. This is achieved using inhibitory competition, which enables neurons to self-organize. In this model, the environment is a separate automaton sitting outside the network and the inhibitory mechanism, e.g. kWTA inhibition, is also modelled as a separate automaton. In similar fashion, cognitive control [Wyble et al., 2005] and other mechanism can be modelled. However, learning is distributed into each neuron automaton.

The network has two layers and the kWTA inhibition automaton. The input layer is modelled as a single automaton called the *InputLayer* and the Output layer is a set of neuron automata. The reason

why we use the *InputLayer*, rather than specifying each input neuron individually is justified as follows:

- Input neurons can be specified as neuron automata, as is shown in Figure 4 and we give them sufficient excitatory inputs externally in order to have them presenting the corresponding activation patterns. However, there can be a great number of input neurons in a system and their job is often simply to relay their inputs to their outputs. As a result, we would increase the state space dramatically if we specified each input neuron explicitly.

- However, we may have to specify input neurons explicitly when the input layer receives feedback from other layers, because the activations of input neurons are dynamically changing. In those cases where feedback activation is not considered, we do not have to specify input neurons explicitly, instead we use a single automaton to define the overall behaviour of the input layer. Note that the automaton is not a neuron automaton, but it is still a timed automaton. The specification of this automaton is given later.

For similar reasons, this model uses kWTA inhibition rather than lateral inhibition, i.e. because we want to reduce the state space of the system. The kWTA algorithm computes inhibitory input directly using mathematical functions, hence a single automaton that performs such computation is sufficient for our network, and this automaton can be shared by different layers. On the other hand, introducing several pools of inhibitory inter-neurons is computationally much more expensive. Although [O'Reilly & Munakata, 2000] suggest that the kWTA computation could be realised via combinations of feedforward and feedback inhibition. However, their justification for this is only informally made by a number of examples.

3.3.1 Global Declaration

The global declaration is as follows:

$$G ::= \{Ee, Ei, El, gl, \Phi, \varepsilon, \Gamma : \mathcal{N}_0^{\psi}, Patterns : P\} \text{ and}$$
$$\boldsymbol{V} ::= \{\boldsymbol{a}, \boldsymbol{gi}\Theta, \boldsymbol{currentPattern} : \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{w} : \mathcal{N}_0^{\psi} \times \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{gi} : \mathcal{N}_0^{\psi}, \boldsymbol{test} : Boolean\}$$

where ε denotes the learning rate. Boolean variable ***test*** decides if weights are updated in the network. ***currentPattern*** is the current pattern being presented to the network. ***gi*** denotes the inhibitory input for output neurons and ***gi***Θ is used by the kWTA automaton and will be explained in the relevant sections. A pattern is a function of type: $\mathcal{N}^+ \to \mathcal{R}_0^1$ in the case of communicating automata and $\mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}$ in the case of Uppaal timed automata. That is, patterns map neuron identifiers (actually, just input layer identifiers) to activation level. P is the set of all possible sets of patterns, where $P = \mathcal{P}(\mathcal{N}^+ \to \mathcal{R}_0^1)$ in the case of communicating automata and $\mathcal{P}(\mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi})$ in the case of Uppaal timed automata. The neural network is trained with a task *Patterns*, which is an element of P.

3.3.2 Input Layer

Some automata in the network are synchronised with *InputLayer*, see Figure 6. What the *InputLayer* does is to decide which input pattern the network experiences at the current moment. It sends its signals via broadcast channels, but when the system is written in other notations, it could be achieved with multi-way synchronisation. As a result, only the relevant neurons can synchronise with it. The environment can be modelled as a single automaton or a set of automata. In the simplest situation, the environment is just a set of patterns, which can be easily embedded within the *InputLayer*. Hence, for these models, *InputLayer* can also be viewed as the environment and input neurons integrated.

The signature of *InputLayer* is shown as follows:

$$INPUTLAYER[C, E], \text{ where}$$
$$C ::= \{\pi : \mathcal{N}_0^{\psi}, inIds, outIds : \mathcal{P}(\mathcal{N}_0^{\psi})\} \text{ and}$$
$$E ::= \{take : \mathcal{N}_0^{\psi} \to Boolean, Z, i : \mathcal{N}_0^{\psi}, tempPatterns : P\}.$$

In the set C, π denotes the identity of the input layer. *InIds* and *outIds* are the sets of identities of the input and output layers respectively. ***currentPattern*** is the current pattern being presented to the network.

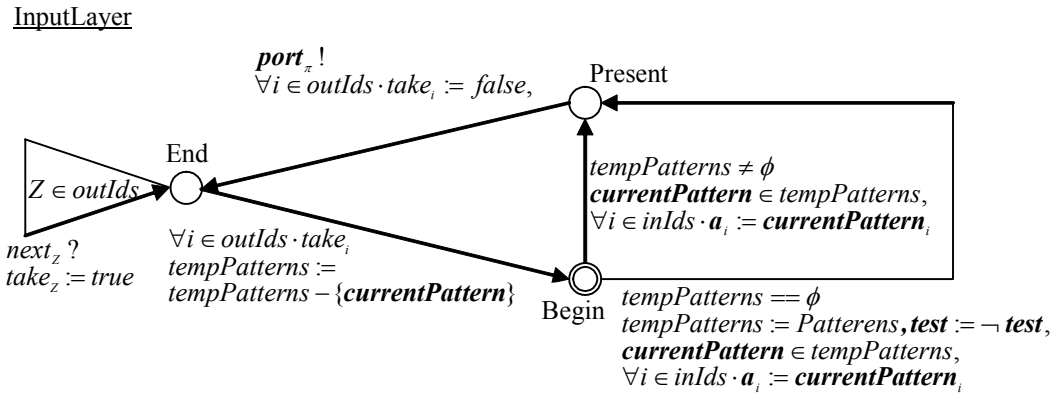


Figure 6 The Input Layer

This automaton is an abstraction of the real input layer and it starts at the *Begin* location, where it decides whether to present the next pattern or to start testing, i.e. it presents the next pattern if not all the patterns have been presented, otherwise it starts to test the network. When the current pattern is presented, the inputs are then assigned as follows:

$$\forall i \in inIds \cdot \boldsymbol{a}_i := \boldsymbol{currentPattern}_i .$$

The next location is *Present*, where the automaton takes a transition signalling the output layer neurons of the availability of the inputs via a broadcast channel $\boldsymbol{port}_\pi !$ and initialises the tokens:

$$\forall i \in outIds \cdot take_i := false .$$

The *InputLayer* then goes to the next location, *End*, where it collects tokens in order to know that all the neurons have completed processing the pattern. The output layer neurons inform the input layer via point-to-point channels $next_Z ?$ when they have evaluated their activations. The *InputLayer* removes the presented pattern from *tempPatterns* and returns to the *Begin* location when it has all the tokens, i.e. when the following guard holds:

$$\forall i \in outIds \cdot take_i$$

To summarise, the *InputLayer* automaton defines how these patterns are presented at the input neurons. ***test*** and ***currentPattern*** are global variables, which are only modified by the *InputLayer*. This automaton is not a neuron automaton but an abstraction of the whole input layer in order to simplify the system. However, input neurons must be modelled explicitly as neuron automata if there are bi-directional activation between the input and hidden layers. Thus, an explicit environment automaton is needed when we model input neurons explicitly or if the environment itself involves complicated structure, because one of the fundamental roles of neural networks is to interact with its environment such as the physical world or bodily states. *InputLayer* is reused in the following sections.

3.3.3 Output Layer

Since the CPCA rule is typically applied in two layer networks, we will not be including a hidden layer here. Thus, we move straight to discussing the output layer. The signature of an output neuron is shown as follows:

$NEURON[C, V, E]$, where

$C ::= \{k : \mathcal{N}_0^\nu, preIds : \mathcal{P}(\mathcal{N}_0^\nu)\}$,

$V ::= \{ge, Vm : \mathcal{N}_0^{\psi}\}$ and

$E ::= \{i : \mathcal{N}_0^{\psi}\}$.

This neuron automaton has four locations, see Figure 7. In the *Input* location, the neuron is ready to compute its input and knows there are new inputs available when a signal $\boldsymbol{port}_{\pi}?$ is broadcasted from *InputLayer*. Then it performs a transition, reads the input values and computes the excitatory input conductance ge. The function for computing the ge is similar to the previous model, except that the bias weight is removed. It also computes $\boldsymbol{gi}\Theta_k$ using the function [O'Reilly & Munakata, 2000]:

$$\boldsymbol{gi}\Theta_k := \frac{ge(Ee-\Theta)+gl(El-\Theta)}{\Theta-Ei}, \quad (3.5)$$

where the $\boldsymbol{gi}\Theta_k$ determines how much inhibition this neuron needs in order to make sure that its membrane potential is just below the threshold, i.e. the neuron is inactive. This function is derived from Equation 1.2 or 2.2, letting $Vm = \Theta$. After it computes all the functions, it goes to the next location, *Input*.

In the *Middle* location, the neuron waits for a signal ***kwta_finish***?, which denotes that the actual inhibition value for this layer $\boldsymbol{gi}$ has been computed by the kWTA automaton. It then computes the equilibrium membrane potential Vm using the following function and moves to the next location: *Output*.

$$Vm := \frac{ge \times Ee + \boldsymbol{gi} \times Ei + gl \times El}{ge + \boldsymbol{gi} + gl}, \quad (3.6)$$

In the *Output* location, the outgoing transition is synchronised with the *InputLayer* by the signal $next_k!$, which informs the *InputLayer* so that the next pattern can be presented.

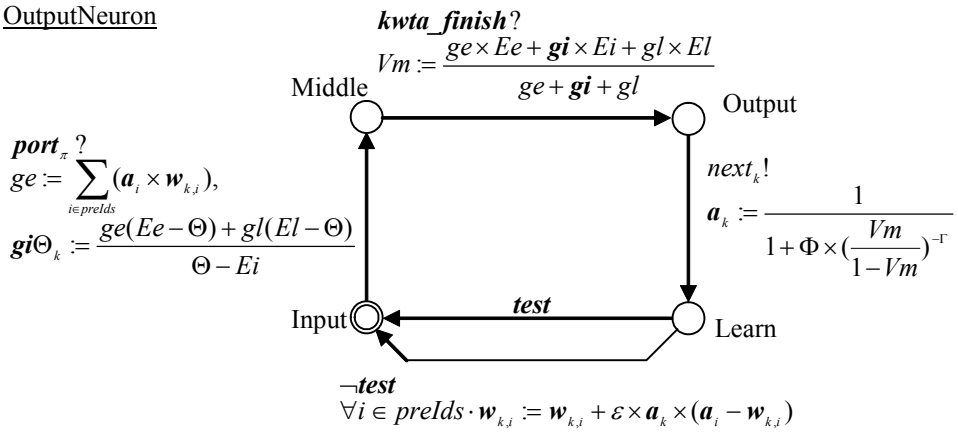


Figure 7 an Output Neuron Automaton

When the neuron is in the *Learn* location and the Boolean variable *test* is *false*, it takes the lower transition and modifies its weights according to the CPCA rule [O'Reilly & Munakata, 2000]:

$$\forall i \in preIds \cdot \boldsymbol{w}_{k,i} := \boldsymbol{w}_{k,i} + \varepsilon \times \boldsymbol{a}_k \times (\boldsymbol{a}_i - \boldsymbol{w}_{k,i}), \quad (3.7)$$

where $\boldsymbol{a}_k$ is the post-synaptic activation, $\boldsymbol{a}_i$ is the pre-synaptic activation and ε is the learning rate. If the Boolean variable *test* is *true*, the network is being tested for its outcomes, hence, the weights are unchanged. These transitions do not need to wait for signals but leave this location straightaway. Finally, the neuron automaton returns to the *Input* location. Note that explicitly encoding the test in a neuron automaton is not biologically plausible. It is a more pragmatic constraint in the model. However, we will discuss this issue in more depth in the case studies.

3.3.3 Other Automata

Some mechanisms can be modelled as separate automata outside the representation of neurons. We just show a kWTA inhibition automaton in Figure 8 as an example to show how mechanisms can be added into the network. The kWTA algorithm is taken from [O'Reilly & Munakata, 2000].

The signature of the kWTA inhibition automaton is shown as follows:

$INHIBITION[C, E]$, where
$C ::= \{q, K, m : \mathcal{N}_0^{\psi}\}$,
$E ::= \{temp, n, i : \mathcal{N}_0^{\psi}\}$.

In the set C, K denotes the number of winners, m denotes the number of neurons in the output layer, and the q parameter determines where exactly to place the inhibition between the Kth and K+1th neurons [O'Reilly & Munakata, 2000]. This automaton is invoked every time the $\boldsymbol{gi}\Theta_i$ s have been computed over a certain layer and it decides the inhibitory input over this layer. The main job of this automaton is to sort the $\boldsymbol{gi}\Theta_i$ values and then use either the basic kWTA or the average-based kWTA function [O'Reilly & Munakata, 2000] to get the final inhibition value $\boldsymbol{gi}$.

The initial location of the kWTA automaton is *Start*. It takes a transition when the signal $\boldsymbol{port}_{\pi}?$ is broadcast from the *InputLayer*. At this point, it also initializes two pointers i and n, and then goes to the next location *L0*. At this location and the next location *L1*, it specifies a sorting algorithm. In the *L0* location, it swaps the order of the ith output neurons $\boldsymbol{gi}\Theta_i$ and the i+1th output neurons $\boldsymbol{gi}\Theta_{i+1}$ if $\boldsymbol{gi}\Theta_i > \boldsymbol{gi}\Theta_{i+1}$, otherwise it maintains their order. The two transitions both go to the *L1* location, where it decides whether the inner pointer i is at the end of the vector, i.e. $i == n-1$. When it is at the end of the array, the last element in the array is the largest $\boldsymbol{gi}\Theta$, and then the automaton resets the inner pointer i and decreases the outer pointer n by 1, in order to find the second biggest element. When it is not the end of the vector it simply increases the inner pointer i.

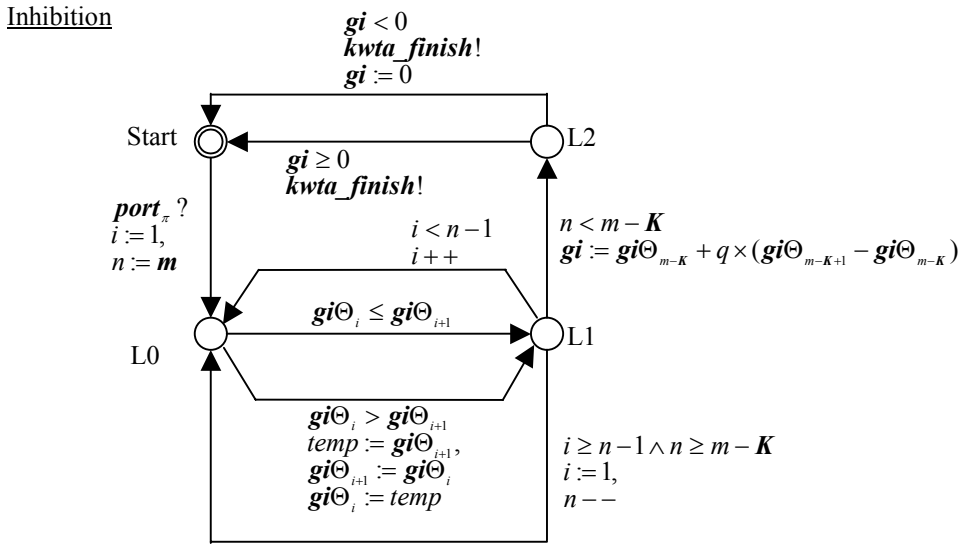


Figure 8 The kWTA Inhibition Automaton

The automaton shown in Figure 8 specifies the basic kWTA function, so the sorting process only needs to indicate the $\boldsymbol{K}$ and $\boldsymbol{K}+1$ th largest $\boldsymbol{gi}\Theta$. As a result, in the *L1* location, it checks if the sorting is completed for the kWTA function. When the guard $n < m - \boldsymbol{K}$ holds, the sorting is finished and the actual inhibitory input for the hidden layer is computed by the function:

$$\boldsymbol{gi} := \boldsymbol{gi}\Theta_{m-\boldsymbol{K}} + q \times (\boldsymbol{gi}\Theta_{m-\boldsymbol{K}+1} - \boldsymbol{gi}\Theta_{m-\boldsymbol{K}}). \qquad (3.8)$$

When we specify the average-based kWTA function, we need to alter the model to sort the whole vector and change the function accordingly. (The average-based kWTA function can be found in [O'Reilly & Munakata, 2000].)

When the final inhibition value ***gi*** is computed, it goes to the next location *L2*. It is possible for some neural networks that the activation in the hidden layer is at a very low level and consequently, the number of activated neurons is less than ***K***. In this situation, the resulting inhibition input ***gi*** computed by the kWTA function is negative. It is not plausible for inhibitory neurons to output negative activation, so in the *L2* location, it needs to check the sign of ***gi***. When it is negative, it will be set to zero. These transitions also send signal ***kwta_finish***! to the *InputLayer* notifying the completion of the process and returns the automaton to the *Start* location.

3.4 The Delta Rule

As we have discussed in the previous section, the objective of model learning is to develop an internal representation of the environment. The CPCA learning rule attempts to discover the regularities (i.e. correlations) within the environment. Task learning is a different kind of learning, which attempts to solve specific problems or tasks. In other words, it attempts to produce appropriate outputs for given inputs. One would expect that a single learning mechanism is able to perform both model learning and task learning, but this is not always possible. For example, we will present several case studies in this paper, which will show some issues concerning the learnability, stability and biological plausibility of different learning algorithms.

In this section, we show a neural network model, which performs error-driven task learning. This learning is also called supervised learning in contrast with the unsupervised model learning. The idea of error-driven learning is to adapt the weights such that the output of the network gets closer to the expected outcome. In order to perform such learning, we firstly need a measure of error. One of the popular measures is the summed squared error (SSE), which is the summed difference between the actual output $\boldsymbol{a}_i$ and target activations $\boldsymbol{currntPattern}_i$, where *i* is the index of the output neurons and *j* is the index of patterns:

$$SSE = \sum_j \sum_i (\boldsymbol{currentPattern}_i - \boldsymbol{a}_i)^2 \qquad (3.9)$$

The goal of error-driven learning is to minimise the SSE and zero SSE indicates a perfect match between the outcome and the task expectation. The delta rule is an error-driven learning approach, which directly computes the derivative of the SSE with respect to the weights. The derivative shows how to change the weights. The detailed calculation of the derivative can be found in [O'Reilly & Munakata, 2000], we only present the learning rule, as follows:

$$\delta := \boldsymbol{currentPattern}_k - \boldsymbol{a}_k, \qquad (3.10)$$

$$\forall i \in preIds \cdot \boldsymbol{w}_{k,i} := \boldsymbol{w}_{k,i} + \varepsilon \times \boldsymbol{a}_i \times \delta \qquad (3.11)$$

$$\text{and } \beta := \beta + \varepsilon \times \delta \qquad (3.12)$$

where δ is an intermediate variable to evaluate error.

3.4.1 Global Declaration

The global declaration is as follows:

$$G ::= \{Ee, El, gl, \Phi, \varepsilon : \mathcal{N}_0^{\psi}, \Gamma : \mathcal{N}_0^{\psi}, Patterns : P\} \text{ and}$$
$$\boldsymbol{V} ::= \{\boldsymbol{a}, \boldsymbol{currentPattern} : \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{w} : \mathcal{N}_0^{\psi} \times \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{test} : Boolean, \delta : \mathcal{N}_0^{\psi}\}$$

where all the parameters have been explained in the previous sections.

3.4.2 Input Layer

As we have explained in the previous section, the input layer in this model is the same as it is in the previous model (see Figure 6).

3.4.3 Output Layer

Again, this network only contains two layers, as is consistent with the focus of the delta rule. Thus, we move to the output layer. The signature of the output neuron is as follows:

$NEURON[C, V, E]$, where
$C ::= \{k : \mathcal{N}_0^{\psi}, preIds : \mathcal{P}(\mathcal{N}_0^{\psi})\}$,
$V ::= \{ge, \beta, Vm : \mathcal{N}_0^{\psi}\}$ and
$E ::= \{i : \mathcal{N}_0^{\psi}\}$.

This neuron automaton, as shown in Figure 9, is similar to the output neuron in the previous model.

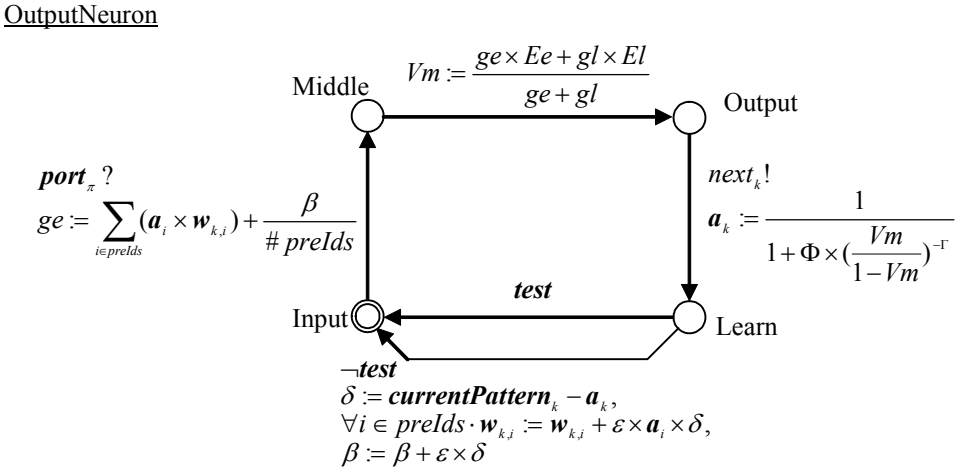


Figure 9 an Output Neuron Automaton

3.5 The Backpropagation (BP) Learning Algorithm

In this section, we model the BP learning algorithm, which is used to train multi-layer neural networks with non-linear activation functions (sigmoids). Furthermore, the algorithm can be applied when the network contains any arbitrary number of layers. Thus, in addition to modelling input and output layers, we will specify a general hidden layer. We can divide the learning procedure into three stages:

- **Forward Propagation of Activation**

  The activations of input layer neurons are determined by the input patterns and the activations of each layer are propagated forward to the next layer until the output layer. We have explained this in previous sections.

- **Backward Propagation of Error**

  The error associated with neuron *k* is evaluated using a strategy called gradient descent, which minimizes the SSE.

$$\delta_k = \begin{cases} \boldsymbol{currentPattern}_k - \boldsymbol{a}_k & \text{if neuron } k \text{ is an output neuron} \\ \sigma'(\eta_k) \sum_j \boldsymbol{w}_{k,j}\, \delta_j & \text{if neuron } k \text{ is a hidden neuron} \end{cases}$$

where $\sigma'(\eta_k) = \boldsymbol{a}_k - \boldsymbol{a}_k^2$, is the derivative of the sigmoid function and *j* ranges over neurons in the successor layer. η_k denotes the net input of the neuron *k*. However, we do not compute this value in our models, so it is not included in the signature. In the graph illustrations, $\sigma'(\eta_k)$ should be replaced by $\boldsymbol{a}_k - \boldsymbol{a}_k^2$.

- **Weight update**

  In this paper, weights are adjusted after processing each pattern:

$$\Delta w_{k,j} = \varepsilon \delta_{\boldsymbol{k}} \boldsymbol{a}_j$$

3.5.1 Global Declaration

The global declaration is as follows:

$G ::= \{Ee, El, gl, \Phi, \varepsilon : \mathcal{N}_0^{\psi}, \Gamma : \mathcal{N}_0^{\psi}, Patterns : P\}$ and
$\boldsymbol{V} ::= \{\boldsymbol{a}, \boldsymbol{d}, \boldsymbol{currentPattern} : \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{w} : \mathcal{N}_0^{\psi} \times \mathcal{N}_0^{\psi} \to \mathcal{N}_0^{\psi}, \boldsymbol{test} : Boolean\}$

where $\boldsymbol{d}$ denotes the error term sent from the output layer to the hidden layer with respect to hidden neuron identities.

3.5.2 Input Layer

The input layer is the same as previous models (see Figure 6).

3.5.3 Hidden Layer

The signature of a hidden neuron is shown as follows:

$NEURON[C, V, E]$, where
$C ::= \{k, \pi : \mathcal{N}_0^{\psi}, postIds : \mathcal{P}(\mathcal{N}_0^{\psi})\}$,
$V ::= \{\beta, ge, Vm : \mathcal{N}_0^{\psi}\}$ and
$E ::= \{take : \mathcal{N}_0^{\psi} \to Boolean, \delta, Z, i : \mathcal{N}_0^{\psi}\}$,

All the variables have been explained previously.

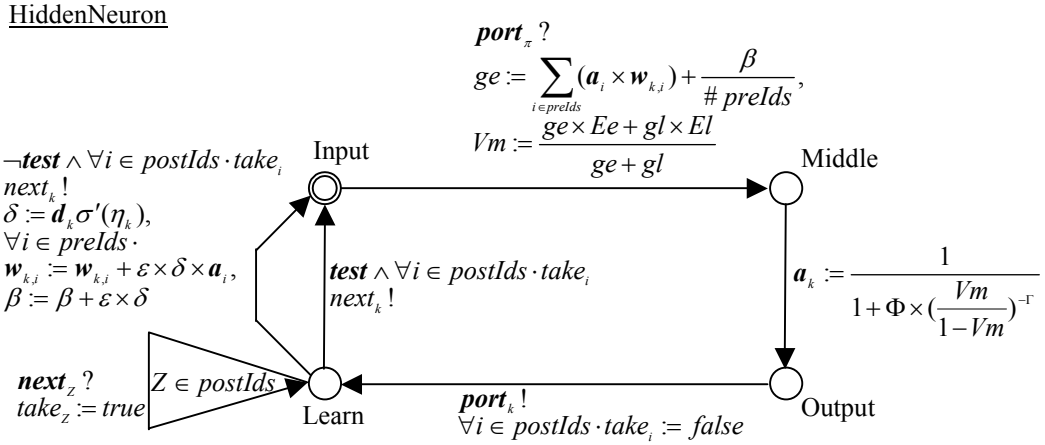


Figure 10 a Hidden Layer Neuron Automaton

The hidden neuron is shown in Figure 10, which is similar to other neuron automata in previous models. However, the essential difference is in the *learn* location, where tokens need to be collected

before the weights are updated. The broadcast signals $\boldsymbol{next}_Z?$ are sent from the output layer neurons, they indicate that error terms $\boldsymbol{d}_z$ are ready (we will explain the error term in the next section). In this model, weights and biases are updated according to the following learning rule [O'Reilly & Munakata, 2000]:

$$\delta := \boldsymbol{d}_k \sigma'(\eta_k), \tag{3.13}$$

$$\forall i \in preIds \cdot \boldsymbol{w}_{k,i} := \boldsymbol{w}_{k,i} + \varepsilon \times \delta \times \boldsymbol{a}_i \tag{3.14}$$

$$\text{and } \beta := \beta + \varepsilon \times \delta . \tag{3.15}$$

3.5.4 Output Layer

The signature of an output neuron is shown as follows:

$NEURON[C, V, E]$, where
$C ::= \{k : \mathcal{N}_0^{\psi}, preIds : \mathcal{P}(\mathcal{N}_0^{\psi})\}$,
$V ::= \{\beta, ge, Vm : \mathcal{N}_0^{\psi}\}$ and
$E ::= \{take : \mathcal{N}_0^{\psi} \to Boolean, \delta, Z, i : \mathcal{N}_0^{\psi}\}$.

The output layer neurons are also neuron automata. One of them is shown in Figure 11. It is clear that the neuron automata in different layers have similar behaviour, so we skip some details. However, the output neurons are different from hidden neurons because they can update weights and biases immediately after outputting, while the hidden neurons need to wait for the error terms to be propagated from later layers.

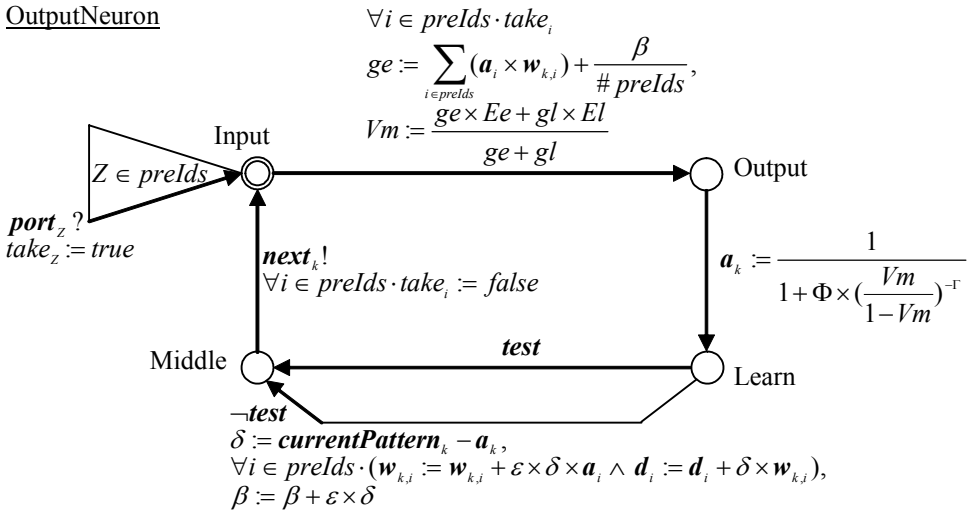


Figure 11 an Output Layer Neuron Automaton

In the *Learn* location, there are also two outgoing transitions. The behaviour of the neurons in this layer is similar to the hidden units except that the output layer neurons update their weights and biases using the following rules [O'Reilly & Munakata, 2000]:

$$\delta := \boldsymbol{currentPattern}_k - \boldsymbol{a}_k , \tag{3.16}$$

$$\forall i \in preIds \cdot \boldsymbol{w}_{k,i} := \boldsymbol{w}_{k,i} + \varepsilon \times \delta \times \boldsymbol{a}_i \tag{3.17}$$

$$\text{and } \beta := \beta + \varepsilon \times \delta . \tag{3.18}$$

Then the error terms, which get back-propagated to the hidden layer, are computed as follows [O'Reilly & Munakata, 2000]:

$$\forall i \in preIds \cdot \boldsymbol{d}_i := \boldsymbol{d}_i + \delta \times \boldsymbol{w}_{k,i}, \qquad (3.19)$$

where $\boldsymbol{d}_i$ is built up for the hidden neuron *i* by summing δ s per output neuron. The broadcast signal $\boldsymbol{next}_k!$ is sent to hidden layer neurons. The signal informs the hidden neurons that the error terms are ready.

3.6 the Generalized Recirculation Algorithm (GeneRec)

GeneRec was considered by [O'Reilly, 1996], as an alternative to gradient algorithms such as the BP algorithm. This algorithm is promising due to its biological plausibility [O'Reilly, 1996]. Although the BP algorithm is powerful in respect of task learning, it suffers from biological implausibility. The essential problem is that the BP algorithm requires the backpropagation of the error term from the dendrite of the post-synaptic neuron to the axon of the pre-synaptic neuron via the synapse. However, there is no biological mechanism, which fulfils such a requirement. On the other hand, GeneRec can approximate BP learning using local activations. It avoids using the problematic error terms. Hence, it is a more plausible task learning algorithm, which could be employed by the brain.

GeneRec requires bi-directional propagation of activations and the learning algorithm includes two activation phases: minus and plus phases.

- In the minus phase, the input activations drive the network's output, which is the actual response. This phase is the same as the forward propagation of activation in BP learning. Variables in this phase are denoted with superscripts, e.g. $\boldsymbol{a}_k^-$.

- In the plus phase, the outputs of the network are externally clamped to expected outcomes. So, both the input and output activations drive the network. Variables in this phase are denoted with superscripts, e.g. $\boldsymbol{a}_k^+$.

Instead of computing the error term, GeneRec calculates the difference of the two activation phases. It has been argued by [O'Reilly, 1996] and [O'Reilly & Munakata, 2000] that the difference of the two stages of activation $\boldsymbol{a}_k^+ - \boldsymbol{a}_k^-$ is a justifiable approximation of the error signal $\delta_k = (\eta_k^+ - \eta_k^-)\sigma'(\eta_k^-)$ used in the BP algorithm, where η denotes the net input:

$$\eta_k = \sum_i \boldsymbol{a}_i \boldsymbol{w}_{k,i} + \sum_j \boldsymbol{a}_j \boldsymbol{w}_{j,k}, \qquad (3.20)$$

where *i* ranges over input neurons and *j* ranges over output neurons. The error term of hidden neurons in BP is based on Equations 3.13, 3.16 and 3.19, i.e.

$$\delta_k = \sum_j (\boldsymbol{currentPattern}_j - \boldsymbol{a}_j)\boldsymbol{w}_{k,j}\sigma'(\boldsymbol{a}_k), \qquad (3.21)$$

where $\boldsymbol{currentPattern}_j - \boldsymbol{a}_j$ is not a local calculation, which is implausible. Assume the bi-directional connections are symmetric, i.e. $\boldsymbol{w}_{k,j} = \boldsymbol{w}_{j,k}$. If we distribute the weights into the sum, we can obtain the following:

$$\begin{aligned}
\delta_k &= (\sum_j \boldsymbol{currentPattern}_j \boldsymbol{w}_{j,k} - \sum_j \boldsymbol{a}_j \boldsymbol{w}_{j,k})\sigma'(\eta_k) \\
&= ((\sum_i \boldsymbol{a}_i \boldsymbol{w}_{k,i} + \sum_j \boldsymbol{currentPattern}_j \boldsymbol{w}_{j,k}) - (\sum_i \boldsymbol{a}_i \boldsymbol{w}_{k,i} + \sum_j \boldsymbol{a}_j \boldsymbol{w}_{j,k}))\sigma'(\eta_k) \\
&= (\eta_k^+ - \eta_k^-)\sigma'(\eta_k) \\
&\approx \boldsymbol{a}_k^+ - \boldsymbol{a}_k^-
\end{aligned} \qquad (3.22)$$

Note that $\boldsymbol{a}_k^+ - \boldsymbol{a}_k^-$ is a perfect approximation of the error signal if the activation function is linear. The steeper the slope, the worse the approximation. Also, the larger $\eta_j^+ - \eta_j^-$, the worse the approximation.

The network is similar to the previous model, the differences are:

- The learning rule is the same for all neurons, as follows:

$$\delta := \boldsymbol{a}_k^+ - \boldsymbol{a}_k^- , \tag{3.23}$$

$$\forall i \in preIds \cdot \boldsymbol{w}_{k,i} := \boldsymbol{w}_{k,i} + \varepsilon \times \delta \times \boldsymbol{a}_i \tag{3.24}$$

$$\text{and } \beta := \beta + \varepsilon \times \delta . \tag{3.25}$$

- The network is bi-directional with symmetric weights.
- There is a separate automaton to implement the settling of the network and the two activation phases. However, we will skip the detail of this automaton and the corresponding changes made to other neuron automata.

Part 4 Requirements Language

The high level behaviour of neural networks is described using a requirements language allowing logical formulae to be expressed. The network of automata evolves through a series of states, which form several paths. The system can evolve through different paths. The requirements language consists of state formulae, which describe individual states and path formulae, which quantify over paths of the model. Assuming φ is a state formula, $\forall$, $\exists$, $\Diamond$ and $\Box$ are operators of path formulae. The property $\forall\Box\varphi$ requires that all states satisfy φ, $\exists\Diamond\varphi$ requires that at least one reachable state satisfies φ, $\exists\Box\varphi$ requires that at least along one path all states satisfy φ and $\forall\Diamond\varphi$ requires that along all paths at least one state eventually satisfies φ.

In this paper we are interested in learning. There is a set of properties, which we want the learning system to satisfy. These properties fall into three categories: Reachability, Safety and Liveness [Behrmann *et al.*, 2004].

- **Reachability Properties**

  These ask whether there eventually exists a state in which something will happen. For example, the formula *success* is *true* when all the output neurons get their activations on the correct side of 0.5. Thus, $\exists\Diamond success$ checks if the learning process could eventually allow the network to output the correct answer. These properties validate the basic behaviour of the model, but do not guarantee the correctness of adaptive systems.

- **Safety Properties**

  These ask whether something "bad" will ever happen. For example, the property *deadlock* evaluates to *true* at a state without successors. Thus, $\forall\Box\neg deadlock$ justifies that the system is free from such situations. Safety properties can always be expressed as reachability properties, such as $\neg\exists\Diamond deadlock$. Assuming approperate formulation of properties, neural networks, cognitive models or any dynamic systems will never reach "bad" states if they satisfy safety properties.

- **Liveness Properties**

  These ask whether the system eventually does something useful. So, we could check liveness properties such as $\forall\Diamond success$, which justifies that the system can meet our requirements along all paths. By verifying these properties over learning algorithms, we can justify if they will eventually converge at desired situations. In order to understand learning algorithms or adaptive systems in general, we are also interested in whether the models can perform something infinitely often.

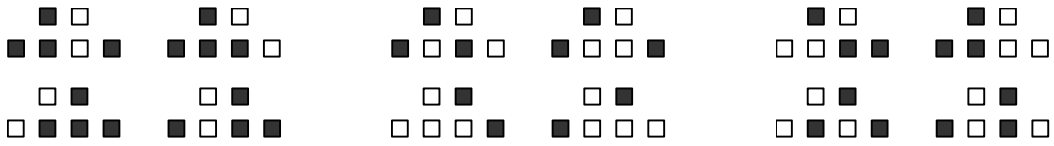
Properties can be expressed by nesting different types of operators, such as $\forall\Diamond\forall\Box success$. This property describes a complex behaviour that we require the learning process to possess. This behaviour is that the adaptive system starts with no knowledge of the task. At this time, success does not hold. During training with examples, it may sometimes show correct answers, i.e. success holds, but it may also show incorrect answers. When this happens, the system has found some solutions to the task but these solutions are not stable during further training. However, starting from some states during the training, it is able to show correct answers invariantly. Thereby, the above property is satisfied. In the next section, we will see that real-time properties can also be expressed.

Part 5 Case Studies

These case studies show how to apply model checking to verify that different learning algorithms can achieve task learning of different difficulties. Firstly, we replicate a preliminary experiment, which has been performed by [O'Reilly & Munakata, 2000] using PDP++ simulations. We begin the investigation with the same tasks, configuration and initialisation of the networks as their work but show our results via model checking rather than simulations. This experiment is not intending to show the advantage of verification over PDP++ simulation, but it demonstrates the methodology of model checking using a simple and rather familiar example. Secondly, we focus on a particular and more interesting bench mark, i.e. the XOR problem. We have chosen the XOR problem due to its historical position and since it requires a small number of nodes and connections. This function is often used to test the ability of learning algorithms and this simple function has been much discussed. The XOR function returns 0 if the inputs are either {1,1} or {0,0}, and returns 1 if the input are either {1,0} or {0,1}. Finally, we relate this verification to a real-time application, which requires model checking of real-time properties expressed in Temporal Logic.

5.1 Verification 1: Model Checking the Learnability of the CPCA Rule, Delta Rule and BP Learning Algorithms

We investigate the learnability of the Conditional Principal Components Analysis (CPCA) rule, delta rule and BP learning algorithms using the following three tasks, which are shown in Figure 12, where dark units represent "0" activation and light units represent "1" activation. Each task contains four patterns. A pattern has four inputs and two outputs. Note, these patterns were used by [O'Reilly & Munakata, 2000] in order to compare the learnability of different learning algorithms using PDP++ simulations. Note, we use BP learning instead of the GeneRec algorithm, which was used by [O'Reilly & Munakata, 2000]. In later sections, we will undertake a more detailed comparison of these two learning algorithms. In addition, we use model checking rather than simulations to obtain our results. Furthermore, this investigation replaces the "Impossible" patter in Figure 11(c) by the XOR problem, since it is in fact embedded within the "Impossible" patter. Thus this can be seen to elaborate on the issue of the class of networks and leaning algorithms that can learn "impossible"-style linearly inseparable problems [Sutherland & Rudy, 1989]. As we have explained previously, the XOR problem requires a small number of neurons and connections. So, it can avoid the scalability problems associated with model checking. We will discuss this problem in later parts.



(a) (b) (c)

Figure 12 (a) the "Easy" Pattern, (b) the "Hard" Pattern and (c) the "Impossible" Pattern

For the sake of simplicity, we skip the detailed specification of these neural networks, but we show the results of the model checking. We test the systems on a standard PC, using Uppaal. We check the neural network using the following formulas:

- $\forall\Box \neg deadlock$: deadlock freedom.
- $\exists\Diamond success$: the network could eventually reach a state where it outputs the correct answer.
- $\forall\Diamond success$: the network can eventually output the correct answer along all possible traces.

where *success* evaluates to *true* when all the output activations are on the correct side of 0.5 at the end of a test trail. The initial weights are set before the verification and the patterns are presented in a fixed order between epochs. As explained previously, the model checker could produce a trace as an example. In this case, Uppaal provides only a single trace and there is no branching in the transition system so produced. The network shows deterministic behaviour, because all communications are completed in the product of the system. Thus, we implement a java program, which generates initial weights randomly from Gaussian distributions with variance of 0.05 and mean of 0, 0.1, 0.2 and 0.3. Then it evokes Uppaal to verify the property $\exists\Diamond success$. This approach aims to cover a broad region of

the initial weights space in order to approximately verify the property of $\forall\Diamond success$. Note, in the rest of the paper we give a percentage for the number of runs that satisfies the property $\exists\Diamond success$, in order to infer to what extend the property $\forall\Diamond success$ will be satisfied. One can vary the mean and the variance of the Gaussian distribution in order to explore different regions of the initial weights space. In addition, some prior knowledge can guide the choice of the regions of exploration. For example, in most of the models, the initial weighs are set to small values.

The result of verification can be seen in Table 2. Firstly, deadlock does not occur in any of these situations. Then we train the networks using the "Easy" patterns in Figure 12 (a). When the network only uses the CPCA learning rule, 80% of trials it learns successfully. However, if we test the same network using the "Hard" patterns in Figures 12 (b) and the XOR problem, the model checker will indicate that the network can learn these tasks for the given initial weights. We then modify the learning rule to the delta rule. The delta rule is used in learning and the "Easy" and the "Hard" patterns are presented to the network. It can be verified that in 76% and 9% of the trials, the delta rule can learn these patterns respectively, in which the "Hard" patterns could not be learned using the CPCA learning rule. We then use the delta rule to train the XOR problem, which is a linearly inseparable problem [O'Reilly & Munakata, 2000]. The result of verification is that it can never find a state in which the learning reaches success, given the initial weights that we use. Finally, we insert a hidden layer of three neurons into the network and use the BP learning algorithm. We skip the "Easy" and the "Hard" patterns because BP learning is a generalisation of the delta rule. So, the result of the delta rule can imply the result of BP learning for these patterns although the success percentage may be different. The verification shows that BP learning is successful in 18% of the trials. We will explain why some of these percentages are very small in the relevant part of the paper.

	"Easy" Patterns			"Hard" Patterns			XOR Problem		
	CPCA	Delta	BP	CPCA	Delta	BP	CPCA	Delta	BP
$\forall\Box\neg deadlock$	√	√	√	√	√	√	√	√	√
$\exists\Diamond success$	80%	76%	—	×	9%	—	×	×	18%

Table 2 √: satisfied or 100%, ×: not satisfied or 0%, —: not relevant

These results are consistent with [O'Reilly & Munakata, 2000]'s simulations. However, their simulations suggest the CPCA rule is not able to learn the "Hard" and "Impossible" patterns and the delta rule is not able to learn the "Impossible" pattern based on the fact that the network has not solved the task by the end of the simulation. Effectively, their simulations really have proved the following: $\neg\exists\Diamond_{\leq N} success$, where N is the number of cycles in the simulation. On the other hand, model checking has mathematically proved that $\neg\exists\Diamond success$ holds. It provides stronger theoretical arguments on the learnability of different learning algorithms. However, this verification is only for a finite set of initial weight settings. In order to obtain a more comprehensive proof, one has to be able to generalise the verification to any initial weight. One possible approach would be using abstract interpretation techniques [Cousot, 2001]. The idea is that the state space may be divided into several regions such that the property is uniform within each region. As a result, the verification can be performed at one initial weight in each region and the property is guaranteed to hold if it holds within every region. We will investigate this idea in future research.

5.2 Verification 2: Model Checking the Learnability and Stability of Learning the XOR Problem using Different Learning Algorithms

In the previous section, we have demonstrated that model checking is able to replicate some experiments that are normally performed using simulations. In this section, we concentrate on one problem, i.e. the XOR problem. This investigation follows from those considered in the previous section.

As shown in Figure 13, the basic network is described as a hierarchy of components. At the top-level it has two modules: *Environment* and *NeuralNet*. Each of which can be composed from a set of modules. For example, *NeuralNet* itself is composed of *InputLayer*, *HiddenLayer* and *OutputLayer*, each of which is also a module composed of a set of neurons. Neurons are fully connected between adjacent layers and BP learning is applied. The *Environment* automaton summarise the environment of the

NeuralNet. In a realistic model of a neural network based controller, this automaton may include the sensory and effecter systems, but in our simplified model these systems are absent.

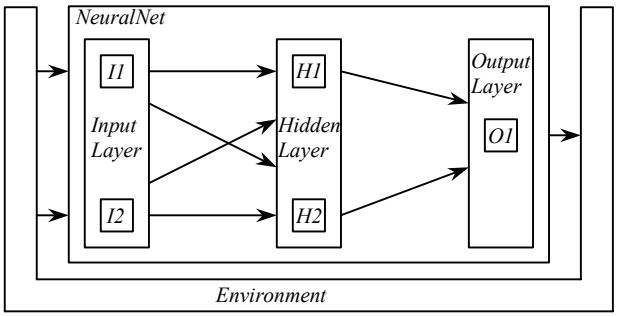


Figure 13 a Neural Network that computes the XOR Problem using the Standard BP learning Algorithm

In following sections, we will investigate this network and different variations of it. They are characterised by the following:

- **BP** - Each weights are integers, which range over [-1000,1000];
- **BP Positive** - The same network, but the weights range over [0,1000];
- **BP kWTA** - Weights range over [0,1000] and there is an additional module: *kWTA*, which computes the inhibitory activation for the hidden layer;
- **BP RNN** - Weights range over [-1000,1000], but the connections between hidden and output layers are bi-directional;
- **GeneRec** - The learning algorithm is the generalised recirculation algorithm [O'Reilly, 1996].

Why consider these classes of networks? To consider whether the class of network can learn linearly inseparable problems is a big scientific question. Especially when issues of biological plausibility are being considered. These different classes of neural network differ with respect to how they employ biological principles. We will elaborate on how these different classes relate to biological plausibility in the relevant sections.

In the previous section, we have investigated the learnability properties using model checking. In this section we extend this investigated with a more complicated property, i.e. the stability property. In order to model check this property using Uppaal, we introduce an additional technique called the stability approximation algorithm.

5.2.1 The Stability Approximation Algorithm

Traditional techniques find it difficult to investigate stability properties, which may be characterised using an error function over learning. It is even more difficult to investigate the oscillatory behaviours of recurrent networks because small weight changes can lead to significantly different activation states. On the one hand, simulations are not able to test the entire state space. Hence, they are limited in their capacity to guarantee that the learning process is able to settle in a minimum and that oscillation can never happen. Most simulations stop training after a certain number of epochs or when a certain number of correct outputs have been obtained. This may not ensure that the network is stable afterwards. Note that turning the learning algorithm off and testing the behaviour of the network with learning off is not a realistic general assumption when considering animal cognition. That is, there is no evidence to support the hypothesis that in the general case animals alternate between phases in which learning is "turned on" and "turned off" (although such phases may arise in the context of specific brain structures and learning functions, see for example [Hasselmo et al., 2002]). A much more viable assumption is that learning is continuously active and, thus, candidate learning algorithms should be stable in the sense being considered here. Readers can also contrast this argument with the plasticity-stability dilemma [Grossberg, 1976], which means animals have to continuously learn new changes in the world and maintain their obtained memories at the same time. The brain has to solve this problem in order to allow us to continue learning new knowledge in a stable fashion without catastrophic interference with our previous learned knowledge.

On the other hand, mathematical analysis is able to prove stability. For example, [Sprinkhuizen-Kuyper & Boers, 1996] has proved that the error surface of the network that computes the XOR problem has a single minimum. The network is trained using the BP algorithm with a single hidden neuron, which is the simplest network that computes the XOR problem. This work also demonstrates how difficult it is to mathematically prove a relatively simple network manually. Moreover, model checking is potentially able to prove some complex properties automatically and with mathematical rigour. We undertake verification and model checking of the following property:

$$\forall\Diamond\forall\Box success$$

which has been discussed in previous sections. This property tests both learnability and stability of learning algorithms. However, the Uppaal model checker does not in general support nesting of temporal operators in properties. The above property not only has nested temporal operators but is also an unbounded liveness property [Behrmann et al., 2004]. So it can not be checked even with a test automaton [Behrmann et al., 2004]. Thus, we introduce an algorithm to verify this property using Uppaal. The algorithm is shown in Figure 14.

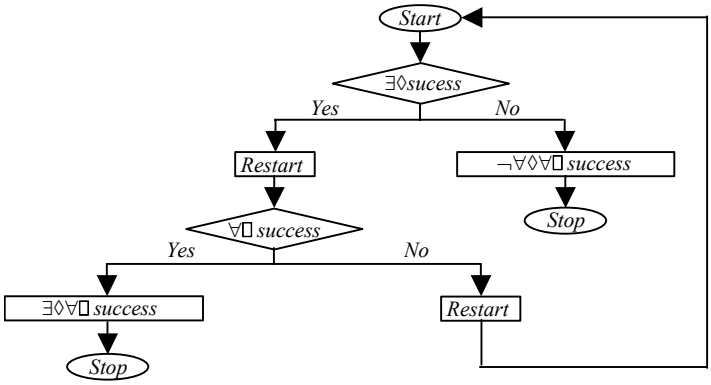


Figure 14 An Algorithm that Partially Verifies the Stability Property

This algorithm is essentially working on an approximation of the statespace. Although Uppaal supports under and over approximate representation of the statespace, it is still unable to verify this property. So we have to extend the Java program, which we used in previous section, to implement our own approximation algorithm. It then invokes the Uppaal model checker. It also refines the trace files generated by Uppaal and outputs an Excel file. Note that this algorithm can not directly check the property $\forall\Diamond\forall\Box success$, but it checks the property $\exists\Diamond\forall\Box success$ from a certain initial weights. We then run this algorithm several times from different initial weights. The result of the verification is a percentage of trials where the property $\exists\Diamond\forall\Box success$ holds.

This algorithm begins at *Start* and invokes Uppaal to check the learnability property:

$$\exists\Diamond success$$

If this property is not satisfied, neither $\exists\Diamond\forall\Box success$ nor $\forall\Diamond\forall\Box success$ holds, otherwise the reached state (which is a possible solution, i.e. *success* is *true*) is identified via the trace file. Then the verification is restarted from the *success* state and checks the following property:

$$\forall\Box success$$

If this property is satisfied, $\exists\Diamond\forall\Box success$ holds, otherwise the reached state (which is not a solution, i.e. *success* is *false*) is identified via the trace file. Then the verification is restarted from this state and returns to the *Start*. It is important to notice that the algorithm may return to the *Start* an infinite number of times if the learning performs some special oscillatory behaviour. Hence, this algorithm does not guarantee to terminate and it is only able to partially verify the stability property. In order to completely verify the stability property, we need to use model checkers that support nesting of operators, such as Kronos [Bozga *et al*., 1998] or LOTOS. However, we restrict the number of

iterations to 20 times. The program resets the initial weights when this algorithm terminates. In the rest of the paper, we will use this algorithm to verify the stability property.

5.2.2 BP

In this network, weights are integers, which range over [-1000,1000]. The gain parameter is $\Phi = 0.15$ and the learning rate is $\varepsilon = 0.01$. The patterns are presented in a randomly selected order, which is unchanged between epochs. This is the same model we used in the previous section, where the verification demonstrated that the network could learn the XOR problem successfully in 18% of trials. We then model check the stability property:

$$\forall\Diamond\forall\Box success$$

The result is that the network is not only able to learn the XOR problem, but the learning is also stable in all successful trials. Furthermore, there are no oscillations of the error term during learning. This result is consistent with the results of [Sprinkhuizen-Kuyper & Boers, 1996]. They have proved that the error surface of the network that computes the XOR problem has a single minimum. Hence, the learning is stable.

5.2.3 BP Positive

In this section, we investigate, the same network as previously, but the weights range over [0,1000]. This neural network is a more biologically plausible model due to the following:

- The weights in the previous section can be positive, zero or negative, but the weights here can only be zero or positive.
- The weights do not change sign during learning.
- Excitatory and inhibitory activations are treated separately from each other.

These principles are more consistent with known neurobiology [O'Reilly & Munakata, 2000]. We then repeat the verification with the initial weights from the same regions as the previous approach and model check the following property:

$$\exists\Diamond success$$

The result is that the network could not learn the XOR problem successfully from the given initial weights. However, this verification is not able to prove that BP learning is unable to learn the XOR problem with only positive weights, because it only verifies the learning from a particular set of initial weight settings. This verification only proves that the standard BP learning algorithm can not learn the XOR problem with a particular initial weights. We will not check the stability property because it will definitely fail to hold if the learnability property is unsatisfied.

5.2.4 BP kWTA

In this section, the neural network is the same as the network in section 5.2.3, but there is an additional module: *kWTA*. This module computes the inhibitory activation for the hidden layer. It has been shown in the previous section that with some initial weights, standard BP learning is not able to learn XOR when the weights are restricted to be zero or positive. So we modify standard BP learning by introducing kWTA inhibition in the hidden layer. We justify that feedforward networks using this modified BP learning rule can learn XOR because the neural networks satisfy the learnability property in 24% of all trials, i.e.

$$\exists\Diamond success$$

The modified BP learning rule is able to learn XOR because:

- When {0,0} is presented to the input neurons, there are no output activations in the input layer neurons. So the activations in hidden layer neurons are zero too. Whichever hidden neuron

wins the competition the kWTA (in this case, being 1WTA) inhibition mechanism does not increase the activations of winners. Hence the output neuron outputs zero.

- When {0,1} or {1,0} is presented to the input neurons, one of the hidden layer neurons is strongly activated and the other activation is close to zero. In this case, there is little competition between the two hidden neurons because their activations are in the two extremes. The kWTA mechanism produces a small inhibitory activation. As a result, the activation of the winner is still strong. Hence, the BP learning can adjust the weights to output 1.
- When {1,1} is presented to the input neurons, both the hidden layer neurons are strongly activated. In this case, there is strong competition between the two hidden neurons because their activations are very close. The kWTA mechanism produces a strong inhibitory pressure between the activations of the two hidden neurons. As a result of strong competition in the hidden layer, the hidden layer activations become much lower. Hence, the BP algorithm can learn to output zero.

We also model check the stability property:

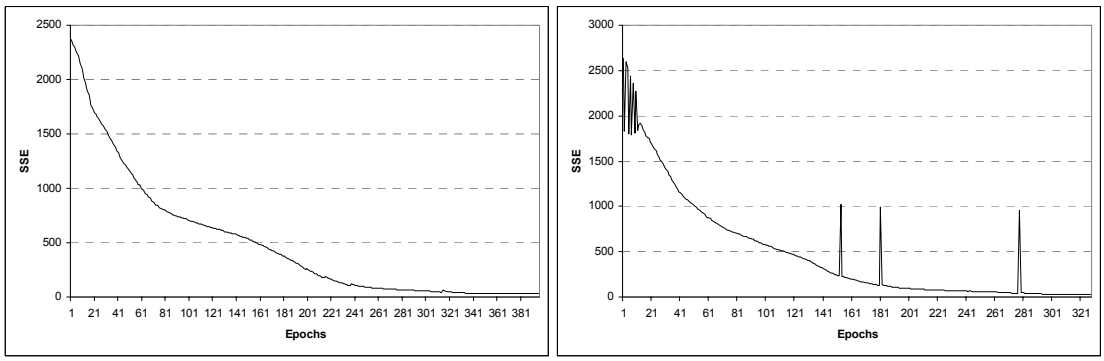
$$\forall\Diamond\forall\Box success$$

The result is that the learning is stable in 83% of the successful trials and unstable in 17% of the successful trials. There is no trail that the program can not determine the stability.

5.2.5 BP RNN

Recurrent neural networks (RNNs) are attractive in biological or cognitive modelling, adaptive control and other important applications. RNNs are dynamic systems, so they need to be analysed for stability, which includes testing the resulting dynamical network behaviour and the learning algorithms. In this section, we concentrate on stability of the learning trajectory. The network is the same as in section 5.2.2 except that the connections between hidden and output layers are bi-directional. The learning task is also XOR. In this section, the learning algorithm is error backpropagation with the symmetry preservation constraint. This means the feedback weights w_{ij} are copied from feedforward weights w_{ji}. In the next section, the learning algorithm we will consider is the generalized recirculation algorithm, which also enforces the symmetry preservation constraint, see [O'Reilly, 1996].

We model check the learning algorithm using the stability approximate algorithm introduced previously. The learning trajectory is measured using summed squared error (SSE), which is explained in previous sections. The results of the verification have three general forms. With some initial weights (87% of all trials), the learning converges to non-solution local minima. Hence, the property $\exists\Diamond success$ is not satisfied. With other initial weights (13% of all trials), the learning converges to solutions, but the leaning trajectories may have two different forms, which are shown in Figure 15. In most of the cases, i.e. about 80% of all the successful trials, the learning trajectory is a smooth, monotonically decreasing, line, as shown in Figure 15 (a). However, there exist some situations, about 20% of all the successful trials, where the leaning trajectory shows oscillations, as illustrated in Figure 15 (b). Note that in both smooth and oscillatory cases, the stability property $\forall\Diamond\forall\Box success$ may hold. If this property holds, it guarantees that there will be either no oscillation from some point of time or no oscillation that can potentially shake the network out of a solution.



(a) (b)

Figure 15 Learning Trajectories for Recurrent Neural Networks that Compute the XOR Problem using the Error Backpropagation Algorithm with the Symmetry Preservation Constraint. (a and b) have Different Initial Weights

So far, we have discovered only 1% of all trials such that the network satisfies the learnability property $\exists\Diamond success$ but not the stability property $\forall\Diamond\forall\Box success$. The cause of this 1% of trials is that the approximation representation algorithm exceeds 20 iterations, i.e. these trials are undecidable by our algorithm. We would be able to perform a more comprehensive verification if we could check the network from any initial weight in the future.

This is a rather striking result because it is well known that the error function generated by RNNs during learning have an arbitrary unstable jumpy character [Bakker et al, 2000]. However, model checking is able to show that some RNNs can converge to stable states. The advantage of model checking as an analytical technique compared with testing based on simulation is the following. Testing a RNN using simulation, one can also obtain learning trajectories such as those shown in Figure 15. However, is that sufficient to say the learning in Figure 15 (a) will not oscillate in further training? Similarly, is that sufficient to say the learning in Figure 15 (b) will oscillate an infinite number of times and not become stable after further training? The answer is that, using simulation, one needs to be very careful in drawing such conclusions.

5.2.6 GeneRec

In this section, we verify the generalized recirculation algorithm (GeneRec). The results of the verification have five different forms:

- In about 70% of all trials, the networks satisfy neither the learnability property $\exists\Diamond success$ nor the stability property $\forall\Diamond\forall\Box success$, which indicates the learning converges to unsuccessful local minima. (In about 30% of all trials, the learnability property is satisfied.)
- In less than 1% of successful trials, the networks satisfy both properties and the learning trajectories are smooth, as shown in Figure 16 (a). However, this is a very rare outcome.
- In about 82% of successful trials, the networks satisfy both properties as shown in Figure 16 (b).
- In about 10% of successful trials, the networks satisfy the learnability property $\exists\Diamond success$ but not the stability property $\forall\Diamond\forall\Box success$, as shown in Figure 16 (c and d).
- In about 7% of successful trials, the stability approximation algorithm is undetermined.

This result is even harder to obtain from simulation without mathematical analysis. For example, the model checking has shown that there are no more "large" spikes after 330 epochs in Figure 16 (b). However, only based on the learning trajectory, one can not obtain such certainty. Looking at the trajectory in Figure 16 (c), between 180 and 420 epochs, one may say the learning is unsuccessful. But based on the trajectory between 420 and 470 epochs, one may say the learning is successful and some simulations will then stop at this point. As the learning continues, it becomes unsuccessful again. This suggests that the results from simulations may be misleading. Another example is shown in Figure 16 (d).

The reason why model checking is able to prove that RNNs can converge to stable states is that model checking is aiming to construct a compact representation of the entire statespace. When the model

checker finds that newly generated states are identical to previously visited states, it can determine that RNNs converge to stable states.

It is argued in [Bakker *et al*., 2000], that after the network gets close to the global minimum, oscillation may occur. A possible reason for this is that RNNs are similar to chaotic systems and are thus very sensitive to the model parameters. He also mentioned noise, which can influence the appearance of the minimum. But our verification has shown that unstable characteristics can be found without introducing any noise.

The oscillation may occur at any stage of learning. The cause of the oscillation may be as follows. The weights are updated after each pattern is presented, so the network is trained to satisfy different sets of constrains at each pattern. For linearly inseparable problems such as XOR, one set of constrains may move the weights in a completely opposite direction to the previous set of constraints. The key issue being that small weight changes have a dramatic affect on the behaviour of the style of recurrent network that GeneRec acts upon. Furthermore, there is no successful but unstable trail in BP RNN but nearly 10% of successful GeneRec trials are unstable. Also 7% of successful GeneRec trials have oscillations, which result in the failure of the stability approximation algorithm. In general, GeneRec has shown a much more unstable character compared with BP RNN learning. In Appendix B, we give a possible explanation of why these two learning algorithms perform differently.

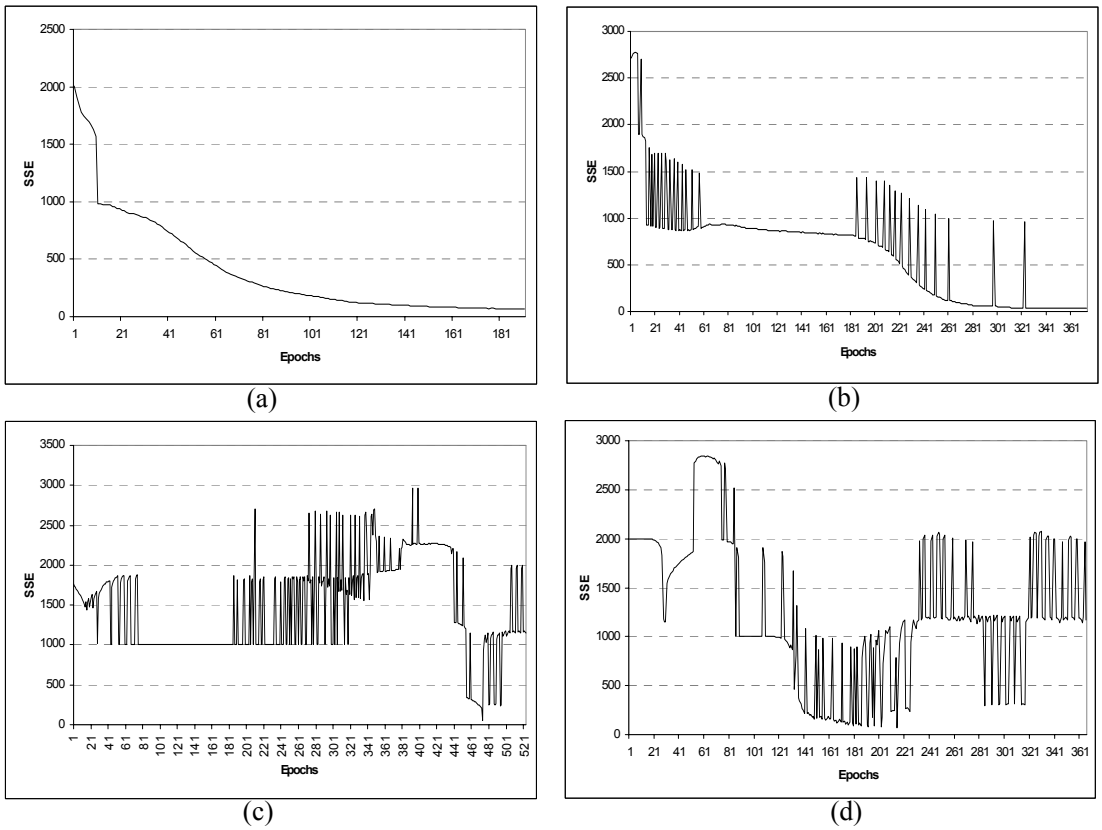


Figure 16 Learning Trajectories for Recurrent Neural Networks that Compute the XOR Problem using the Generalized Recirculation Algorithm with the Symmetry Preservation Constraint. (a, b, c and d) have Different Initial Weights

5.3 Verification 3: Model Checking the Speed of the Backpropagation Algorithm with Application to Neural Network based Controllers

It is stated in [Menzies & Pecheur, 2004] that Artificial Intelligence (AI) is useful in real-world applications but developers must gain confidence of the correctness of such systems. However, it is hard to apply traditional verification and validation (V&V) techniques to AI systems. In their work, several readily available methods are investigated in order to evaluate their applicability to the V&V of AI systems. In their paper, the following techniques are discussed.

- Scenario based testing,
- Run-time monitoring,
- Static analysis,
- Model checking and
- Theorem proving.

They pointed out that testing might be difficult and expensive for complex AI systems. One reason for this is that the input set of AI systems is normally huge and discontinuous. So, small changes in the input set may lead to significantly different performance of the system. Testing is aiming to test a minimum subset of the input set and infer that other inputs will behave similarly. In this case, it may be difficult and expensive to define a minimum subset and test it. Run-time monitoring is also discussed in [Menzies & Pecheur, 2004] and several case studies have been conducted at NASA. For example, [Artho et al., 2003] and [Barringer et al., 2004] have reported that run-time monitoring/verification can be used to monitor some temporal logic properties during a K9 planetary rover operation.

It is reported by [Menzies & Pecheur, 2004] that model checking is widely used for the V&V of AI systems at NASA. For example, some parts of NASA's Remote Agent Experiment (RAX) Executive have been converted to the Spin input language and model checked using the Spin model checker. Five concurrency errors were uncovered and fixed [Havelund et al., 2000]. Furthermore, model checking has also been applied to the K9 rover executive [Giannakopoulou et al., 2002]. They have also pointed out a limitation of model checking, which is that an AI system has to be transformed to the modeling language of a model checker. This transformation may be difficult and expensive. However, the architecture of connectionist networks is often homogeneous, hence, once a neuron has been specified, a bigger network is relatively easy to construct based on it. This is not always true for rule based AI systems. Static analysis and theorem proving are also discussed in [Menzies & Pecheur, 2004], but the former is labour-intensive, and the latter requires expert skills and effort. The other methods are largely automatic. In this section we will focus on model checking neural network based controllers, which perform on-line supervised learning.

In some applications, the control architectures use pre-trained networks, which are numerical approximations of a function. The correctness of such systems can be verified [Rodrigues et al., 2001], but their verification does not consider the adaptability of the system. Other control architectures use on-line training of neural networks. This approach is attractive because it is able to handle dynamic adaptation, but it requires a high level of stability and correctness of the learning process. There are existing approaches to evaluating the performance of neural networks, such as [Schumann et al., 2003], which proposed a layered approach, i.e. run-time monitoring, to verify and validate neural network based controllers. The limitation of their work is that they only focus on monitoring the on-line adaptation but cannot guarantee stability and correctness at system design stages. In addition, it is not always feasible to replace formal stability guarantees with only empirical testing, due to limited resources and time.

As we have explained previously, model checking is an alternative method to the V&V of AI systems, which includes rule-based systems and neural networks. Model checking can be applied along side the learning with little human interaction due to the fact that model checking is automatic. It is useful for deep space explorations, because the communication delay between the spacecraft and the base station may be hours and the on-line learning system has to deal with unforeseen circumstances. Therefore, any kind of prior verification may be infeasible. We propose to model check an abstract model of the on-line learning system before the actual learning is undertaken.

5.3.1 Specification

As already discussed, the BP algorithm is a supervised learning rule widely used in many applications, so study of this algorithm has practical value. We have chosen the XOR problem as our learning task due to its historical position. As previously considered, although it requires a small number of nodes and connections, it is characteristic of difficult linearly inseparable learning tasks. This simple problem is often used to test the ability of learning algorithms and it has been much discussed in this paper and by other researchers, e.g. [O'Reilly, 1996]. In terms of our larger ambition, analyses of neural network based controllers, this XOR verification serves as a preliminary assessment of our approach, which will be extended to realistic applications in future work.

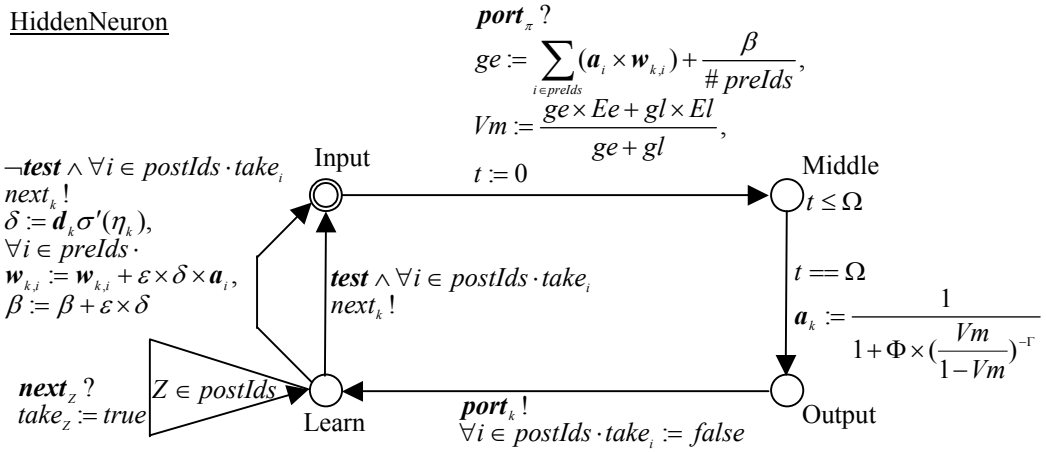


Figure 17 Neural Automaton with Timing Constraints

This network is identical to that shown in Figure 13 except that there is a *Tester* automaton, which is shown in Figure 18. We will explain the *Tester* in the next section. In this section, we focus on the neuron automaton shown in Figure 17, which is similar to the automaton in Figure 10. However, there are timing constraints attached to every neuron automata used here.

The timing constraints in this model are the following. $t \le \Omega$ is an invariant of the *Middle* location, and t is a local clock. Invariants are timing conditions, and automata can only stay in locations while the condition holds. $t == \Omega$ is a guard, which is a condition allowing the transition to be taken. To summarize the time course of the neuron, it stays at the *Input*, *Output* and *Learn* locations while communication is completing, but it stays at the *Middle* location for exactly Ω units of time, which we assume represents the time required to update net input and activation. In this case, Ω is 5 units of time. This assumption is made only for analytical reasons and is neither based on neuron physiology nor hardwire constrains. The timing constraints in output neurons are specified in similar fashion.

5.3.2 Verification

In this section, we verify this model in Uppaal. We assume that the *deadline* is 5000 units of time, the learning rate is 0.05. Note that the *deadline* and Ω are both arbitrary numbers, which would be prescribed by a real application. We check the system for deadlock freedom using the following Temporal Logic formula:

$$\forall\Box \neg deadlock$$

The result is that the system has no deadlocks for the XOR training set. We also verify the stability of the network using the following Temporal Logic formula, which contains timing:

$$\forall\Diamond_{\le deadline} \forall\Box success$$

Satisfaction of this formula means the system always reaches a successful state before the *deadline*, and *success* holds invariantly from that state. However, Uppaal does not support this formula. Hence, we introduce a test automaton in Figure 18. It begins at the *Start* location and moves to the next location, *Deadline*, when clock t equals the parameter *deadline*. When the test automaton is in this location, patterns are still presented to the neural network. If learning is successful when *deadline* becomes *true* and remains stable during further training, the next location, *Fail*, is unreachable. With this test automaton, we are able to verify the previous property using the following Temporal Logic formula:

$$\forall\Box \neg Tester.Fail$$

The result of the verification is that the system satisfies the above property and is guaranteed to learn XOR according to the required timing constraints. It also guarantees the learning process eventually stabilises.

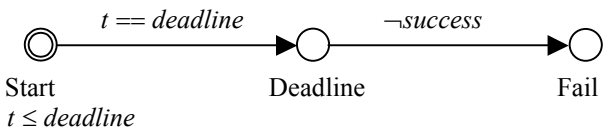


Figure 18 The Test Automaton, *Tester*

If the above properties are not satisfied, the result can be feedback to the controller and parameters can be adjusted accordingly. In critical situations, an alarm can be raised. When adaptation is not possible, the model checker may even abort the mission.

Part 6 Discussion and Future Work

In this paper, we present a framework to specify neural networks and learning algorithms using formal methods, i.e. communicating automata. We argue that this formalism sits between classical forms of symbolic systems arising from programming languages such as Lisp and Prolog, and connectionist networks. In the case study, we verify neural networks over a set of properties expressed in Temporal Logic. Our models and properties can be regarded as symbolic descriptions of the system behaviour at different levels of abstraction. The low-level descriptions use neural networks encoded in communicating automata. The high-level descriptions contain a set of properties, which are expressed in logical formulae. They are abstract descriptions of global properties, which do not prescribe internal details of how those properties are realised. Verifications may give theoretically well-founded ways to evaluate and justify the learning capacity and determine whether cognitive properties can emerge from neural-level architectures. This work also servers as a preliminary assessment of the V&V of neural network based controllers.

In traditional neural network simulations, semantically interpretable elements are patterns of activation. The states of neural networks are expressed in numerical form, such as a landscape, in a multi-dimensional space. However, the states of the neural networks in our models are represented as the locations in the product automaton, which is automatically generated by the model checker. The locations have the characteristics of symbol systems. Model checking is based on symbolic manipulation of the product automaton, which maps to the landscape in the multi-dimensional space. We also argue that most of the properties that we have verified are hard to justify by simulation. This is because simulations can only test that something occurs but are unable to test that something can never occur. Simulations are also not able to test if something is (in)valid forever. Therefore, simulations are insufficient to justify safety and liveness properties without explicit mathematical analyses. Our verification approach on the other hand, explicitly and formally describes the system and properties, so safety and liveness properties can be verified.

Based on the verifications in our case studies, we observe that none of the learning algorithms considered can perform all kinds of learning in a stable fashion. This seems to suggest that animals might use different learning mechanisms or combinations of different algorithms and other neural mechanisms such as inhibition. We have seen in one of our case studies, that kTWA inhibition can play a key role in task learning under certain circumstances.

However, there are a number of limitations of the Uppaal based approach used here.

- Firstly, the model checking approach does not scale very well when there are more connections. For example, the verification of about 4000 initial weights will run for more than 7 hours for more difficult tasks, e.g. the XOR problem and for about one or two hours for relatively easy tasks, such as the “Easy” patterns in Figure 12(a).

- Secondly, unbounded liveness properties, such as the stability property, can not be model checked using Uppaal. However, it could be checked using other model checkers, e.g. a LOTOS based approach via the CADP model checker [Garavel et al, 2002].

- Thirdly, both the communicating automata model and the Uppaal timed automata models have only one level of decomposition. However, it is stated in earlier sections that decompositional structure is required for cognitive modelling. So, one could believe that the process algebra approach, such as LOTOS, could support top-down hierarchical specification of more complicated models.

- Fourthly, decompositional structure could be validated compositionally enabling the model checking approach to scale better.

- Lastly, other verification methods, such as the CADP tool that verifies LOTOS models, could provide real numbers. This enables more accurate modelling of neural networks. It also improves gradient based learning compared with integer weights. (Integer weights are considered as one of the reasons why our gradient based learning, such as our delta rule and BP learning, have very low success rates compared with the rates obtained by other researchers, e.g. [O’Reilly, 1996].)

Whether this architecture is a suitable representation of a mental system and whether such an approach can resolve any of the limitations discussed in Part 1 is still under investigation. A lot of work needs to be carried out in the future in order to discover solutions to the limitations of neural networks and therefore provide new architectures to model complex mental systems. We are also interested in investigating the potential of this approach in integrating symbolic and sub-symbolic computations, enforcing hierarchical compositional structure over neural networks, building or justifying brain-level models and model checking realistic neural network based controllers.

Part 7 Bibliography

[Alur & Dill, 1994] Rajeev Alur and David L. Dill, A theory of timed automata, Theoretical Computer Science, 126(2): 183-235, 1995

[Anderson, 1993] Anderson, J. R., Rules of the Mind, Hillsdale, NJ, Erlbaum, 1993

[Artho et al., 2003] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, W. Visser, Experiments with Test Case Generation and Runtime Analysis, Abstract State Machines (ASM'03) LNCS Vol:2589, 87-107, 2003

[Barnard & Bowman, 2000] Barnard, P.J. and Bowman, H., Rendering Information Processing Models of Cognition and Affect Computationally Explicit: Distributed Executive Control and the Deployment of Attention, Cognitive Science Quarterly, 2000

[Barringer et al, 2004] H.Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04), 2004

[Bakker et al, 2000] R. Bakker, J.C. Schouten, C.L. Giles, F. Takens, and C.M. van den Bleek. Learning Chaotic Attractors by Neural Networks. Neural Computation, 12:2355-2383, 2000

[Behrmann et al, 2004] Gerd Behrmann, Alexandre David and Kim G. Larsen. A Tutorial on Uppaal, In Proceeding of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), LNCS 3185, 2004.

[Bliss & Lomo, 1973] Bliss, T. V. P., and Lomo, T., Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path, Journal of Physiology, 232, 331-356, 1973

[Bolognesi & Brinksma, 1988] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language LOTOS. Computer Network and ISDN Systems, 14910: 25-29, 1988

[Bowman, 2001] Bowman, H., Time and action lock freedom properties of timed automata. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, Formal Techniques for Networked and Distributed Systems, pages 119-134. Kluwer Academic Publishers, August 2001

[Bowman & Gomez, 2006] H. Bowman and R. Gomez. Concurrency Theory - Calculi and Automata for Modelling Untimed and Timed Concurrent Systems. Springer, 2006

[Bowman & Derrick, 2001] H. Bowman, J. Derrick (editors), Formal Methods for Distributed Processing: a Survey of Object-oriented Approaches, Cambridge University Press, 2001

[Bozga et al, 1998] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool or real-time systems. In Proceedings of the 10th International Conference on Computer Aided Verification, Springer-Verlag 546–550, 1998

[Bratko, 1986] Bratko, I., Prolog Programming for Artificial Intelligence, Addison-Wesley International Computer Science Series, 1986

[Callan, 1999] Callan, R., The Essence of Neural Networks, Chapter 7, 8, Prentice Hall, 1999

[Cousot, 2001] Patrick Cousot, Abstract Interpretation Based Formal Methods and Future Challenges. In Informatics, 10 Years Back - 10 Years Ahead, R. Wilhelm (Ed.), Lecture Notes in Computer Science 2000, 138-156, 2001

[Damm, 2001] Werner Damm, Verification of a Radio-Based Signaling System Using the STATEMATE Verification Environment, Formal Methods in System Design, Volume 19, Issue 2, Pages: 121-141, 2001

[Fodor & Pylyshyn, 1988] Fodor, J.A. and Pylyshyn, Z.W., Connectionism and Cognitive Architecture: A Critical Analysis, Cognition: International Journal of Cognitive Science, Vol. 28, 1988

[Fernandez et al, 1996] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, Mihaela Sighireanu, CADP A Protocol Validation and Verification Toolbox, Proceedings of the Eighth International Conference on Computer Aided Verification CAV, 1996

[Garavel et al, 2002] Hubert Garavel, Frédéric Lang, Radu Mateescu, An overview of CADP 2001, European Association for Software Science and Technology (EASST) Newsletter volume 4, pages 13-24, 2002

[Gibson, 1993] Gibson, J.P., A LOTOS-Based Approach to Neural Network Specification, Technical Report CSM-112, Department of Computing Science and Mathematics, University of Stirling, UK, 1993

[Garcez, Broda & Gabbay, 2002] Artur S. d'Avila Garcez, Krysia B. Broda and Dov M. Gabbay, Neural-Symbolic Learning Systems: Foundations and Applications, Springer-Verlag, 2002

[Giannakopoulou et al., 2002] D.Giannakopoulou, C. Pasareanu, H. Barringer, Assumption generation for software component verification, Proceeding of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), 2002

[Grossberg, 1976] Grossberg S., Adaptive pattern classification and universal recoding, I: Parallel development and coding of neural feature detectors, Biological Cybernetics 23, 121-134, 1976

[Hasselmo et al., 2002] Hasselmo M.E., Bodelon C., and Wyble B.P., A proposed function for hippocampal theta rhythm: separate phases of encoding and retrieval enhance reversal of prior learning, Neural Computation 14(4): 793-817, 2002

[Havelund et al., 2000] K. Havelund, M. Lowry, S.Park, C. Pecheur, J. Penix, W. Visser, J. L. White, Formal analysis of the remote agent before and after flight, Proceedings of 5th NASA Langley Formal Methods Workshop, 2000

[Ibrahim et al, 1990] R.L.Ibrahim, J.A.Ogden, S.A.Williams, Should Concurrency be Specified?, Specification and Verification of Concurrent Systems, Springer-Verlag, 1990

[Koch, 1999] Christof Koch, Biophysic of Computation: Information Processing in Single Neurons, Oxford Uni. Press, 1999

[Lewis, 1999] Lewis, R.L, Cognitive Modelling, Symbolic. In Wilson, R. and Keil, F. (eds.), The MIT Encyclopedia of the cognitive Science. Cambridge, MA: MIT Press, 1999

[McClelland & Rumelhart, 1987] James L. McClelland, David E. Rumelhart and the PDP Research Group, Parallel Distributed Processing, The MIT Press, 1987

[Meyer & Kieras, 1997] Meyer, D. E. and Kieras, D. E., A computational theory of executive cognitive processes and multiple task performance: Part 1. Basic mechanisms, Psychological Review 104: 3-65, 1997

[Napolitano et al., 1998] M. Napolitano, G. Molinaro, M. Innocenti, and D. Martinelli, A complete hardware package for a fault tolerant flight control system using on-line learning neural networks, IEEE Control Systems Technology, 1998

[Newell, 1990] Newell, A., Unified Theories of Cognition, Cambridge, Massachusetts, Harvard University Press, 1990

[Newell & Simon, 1976] Newell, A., and Simon, H.A., Computer science as empirical inquiry: Symbols and search. Communications of the Association for Computing Machinery, 19(3), 113-126, 1976

[Norvig, 1992] Peter Norvig, Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, Morgan Kaufmann, 1992

[O'Reilly & Munakata, 2000] O'Reilly, R.C. and Munakata, Y., Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain, the MIT Press, 2000.

[O'Reilly, 1996] O'Reilly, R.C., Biologically Plausible Error-Driven Learning Using Local Activation Differences: The Generalized Recirculation Algorithm, Neural Computation 8, (895-938), MIT, 1996.

[Plonsey & Fleming, 1969] Robert Plonsey and David G. Fleming, Bioelectric Phenomena, McGraw-Hill Book Company, 1969

[Pettersson, 2000] Pettersson, P., Larsen, K.G.: UPPAAL2K: Small Tutorial. Bulletin of the European Association for Theoretical Computer Science 70:40–44, 2000

[Pollack, 1990] Pollack, J. B., Recursive distributed representations, Artificial Intelligence, 46:77-105, 1990

[Rodrigues et al, 2001] P. Rodrigues, J. F. Costa and H. T. Siegelmann. Verifying Properties of Neural Networks. Artificial and Natural Neural Networks, LNCS 2084, Springer-Verlag, 158-165, 2001

[Rumelhart et al, 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., Learning internal representations by error propagation. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, Paralled Distributed Processing. Explorations in the Microstructure of Cognition. Volume 1: Foundations, pages 318-362. The MIT Press, Cambridge, MA, 1986

[Sampaio, Woodcock & Cavalcanti, 2002] Sampaio, A., Woodcock, J. and Cavalcanti, A.: Refinement in Circus, FME 2002: 451-470, 2002

[Schneider & Treharne, 2003] Schneider, S. and Treharne, H., CSP Theorems for Communicating B Machines, Royal Holloway Department of Computer Science Technical Report CSD-TR-02-12, 2003

[Schumann et al, 2003] J. Schumann, P. Gupta and S. Nelson. On Verification & Validation of Neural Network Based Controllers, EANN'03, 2003

[Silver & Brzozowski, 2003] S.J. Silver; J.A. Brzozowski, True Concurrency in Models of Asynchronous Circuit Behavior, Formal Methods in System Design, 22(3):183-203, 2003

[Smith, 1992] L. S. Smith. A Framework for Neural Net Specification, IEEE Transactions on Software Engineering, 18(7): 601-612, 1992

[Sprinkhuizen-Kuyper & Boers, 1996] Ida G. Sprinkhuizen-Kuyper and Egbert J. W. Boers, The Error Surface of the Simplest XOR Network Has Only Global Minima, Neural Computation, 8, 1301-1320, MIT, 1996

[Sutherland & Rudy, 1989] Sutherland, R. J., and Rudy, J. W., Configural association theory: The role of the hippocampal formation in learning, memory, and amnesia. Psychobiology, 17(2), 129-144

[Wu, Chanson & Gao, 1999] Jianping Wu, Samuel T. Chanson, and Quiang Gao, editors, Proceeding of Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX), London, UK, Kluwer Academic Publishers, 1999

[Wyble et al., 2005] Wyble B, Sharma D, and Bowman H., Modelling the Slow Emotional Stroop Effect: Suppression of Cognitive Control. In Angelo Cangelosi, Guido Bugmann, and Roman Borisyuk, editors, Proceedings of the Neural Computation and Psychology Workshop, January 2005

[Zeidenberg, 1990] Matthew Zeidenberg, Neural Network Models in Artificial Intelligence, Ellis Horwood, 1990

Appendix A

The most realistic rule that determines the action potential is based on the bioelectric phenomena that were discovered by Hodgkin and Huxley. They wrote down a set of equations to describe action potential generation. In [Koch, 1999] the single equation for all the currents flowing across a patch of axonal membrane is as follows:

$$C_m \frac{dV}{dt} = \overline{G}_{Na} m^3 h(E_{Na} - V) + \overline{G}_K n^4 (E_K - V) + G_m (V_{rest} - V) + I_{inj}(t) . \qquad \text{(A.1)}$$

There are two voltage dependent ionic conductances, a sodium conductance, G_{Na}, and a potassium conductance, G_K. They are independent from each other. A leak conductance, G_m, is independent of the membrane potential. The sodium and potassium conductances are expressed as maximum conductances, $\overline{G}_{Na}$ and $\overline{G}_K$, multiplied by numerical coefficients, $m^3 h$ and n^4, which represent the fraction of the maximum conductance actually open. However, we will not explain these coefficients in this paper and full details of them can be found in [Koch, 1999] and [Plonsey & Fleming, 1969]. The ionic reversal potentials E_{Na} and E_K are given by Nernst's equation for the appropriate ionic species. They depend on the concentration difference of the ions and the electrical field across the cell membrane. C_m is the membrane capacitance, V is the membrane potential relative to the resting potential V_{rest} and I_{inj} is the current that is injected via an intracellular electrode. The injected current is not a function of V, it is determined externally.

Having Equation A.1 we can describe the change of membrane potential without considering the underlying ionic channels. Therefore many computational models of nerve systems are directly based on such an approach. For example the membrane potential updating rule of a prominent current connectionist modelling system PDP++ can be found in [O'Reilly & Munakata, 2000]. It is as follows:

$$V_m(t+1) = V_m(t) - dt_{vm}[g_e(t)\overline{g}_e(V_m(t) - E_e) + g_i(t)\overline{g}_i(V_m(t) - E_i) + g_l(t)\overline{g}_l(V_m(t) - E_l)] \qquad \text{(A.2)}$$

this is a discrete simplification of the Hodgkin-Huxley equations. In order to justify that Equation A.2 is a discrete simplification of Equation A.1, we transform it into a continuous equation and compare it with Equation A.1. Note that $\overline{g}_e$, $\overline{g}_i$ and $\overline{g}_l$ are set to 1 in the model, thus they are ignored in the other sections. This is consistent with the simulations in [O'Reilly & Munakata, 2000].

If we rearrange Equation A.2, we get the following:

$$V_m(t+1) - V_m(t) = -dt_{vm}[g_e(t)\overline{g}_e(V_m(t) - E_e) + g_i(t)\overline{g}_i(V_m(t) - E_i) + g_l(t)\overline{g}_l(V_m(t) - E_l)] \qquad \text{(A.3)}$$

and it can be shown that $V_m(t+1) - V_m(t)$ is equal to $\Delta V_m\big|_{\Delta t=1}$, which denotes the amount of change of V_m relative to an interval of 1 unit of time. We then replace the left side of the equation with $\Delta V_m\big|_{\Delta t=1}$ and distribute the negation on the right side of the equation into the brackets. Finally we divide both sides of the above equation by dt_{vm} and swap all the $g_e(t)$ and $\overline{g}_e$ ($g_i(t)$ $\overline{g}_i$, and $g_l(t)$ $\overline{g}_l$) around, resulting in the following:

$$\frac{\Delta V_m\big|_{\Delta t=1}}{dt_{vm}} = \overline{g}_e g_e(t)(E_e - V_m(t)) + \overline{g}_i g_i(t)(E_i - V_m(t)) + \overline{g}_l g_l(t)(E_l - V_m(t)) \qquad \text{(A.4)}$$

We then evaluate the limit of both sides of Equation A.4 with Δt tending towards zero. Taking the limit of a discrete equation converts the equation into a continuous equation. The limit of the right side of Equation A.4 is the original value of the right side. We show the limit of the left side as follows:

$$\lim_{\Delta t \to 0} \frac{\Delta V_m\big|_{\Delta t=1}}{dt_{vm}} = \frac{\lim_{\Delta t \to 0} \Delta V_m\big|_{\Delta t=1}}{dt_{vm}} = \frac{dV_m}{dt_{vm}} = \frac{dt}{dt_{vm}} \times \frac{dV_m}{dt},$$

where dt_{vm} is the time constant in [O'Reilly & Munakata, 2000]. The limit of a constant dt_{vm} is the value of the constant itself. The continuous form of Equation A.2 is as follows:

$$\frac{dt}{dt_{vm}} \times \frac{dV_m}{dt} = \overline{g}_e g_e(t)(E_e - V_m(t)) + \overline{g}_i g_i(t)(E_i - V_m(t)) + \overline{g}_l g_l(t)(E_l - V_m(t)) \qquad \text{(A.5)}$$

Now we compare Equation A.5 with Equation A.1. It should be noticed that the right side of both equations are equivalent if the injected current I_{inj} is zero, $g_e(t)$ and $g_i(t)$ are set to the appropriate coefficients and $\overline{g}_l$ is set to 1 (which is its typical value). The left side of both equations will be equal if the following holds.

$$\frac{dt}{dt_{vm}} = C_m \quad \text{or} \quad dt_{vm} = \frac{dt}{C_m} \qquad \text{(A.6)}$$

As it is defined in [O'Reilly & Munakata, 2000], the time constant $0 < dt_{vm} < 1$ slows the potential change, capturing the corresponding slowing of this change in a neuron primarily as a result of the capacitance of the cell membrane. Hence we believe that Equations A.6 and A.7 hold according to the definition of dt_{vm}. Hence, we have justified that Equation A.2 is a discrete simplification of Equation A.1.

Appendix B

Comparing the Generalized Recirculation and the Backpropagation Algorithms

It is interesting that the learning trajectories of GeneRec are generally much jumpier than error backpropagation in RNNs. This may suggest that the two algorithms sometimes compute different error gradients. It has been mathematically verified by [O'Reilly, 1996] and [O'Reilly & Munakata, 2000] that the difference of the two stage activations $-(a_j^+ - a_j^-)$ is a justifiable approximation of the error signal $\delta_j = -(\eta_j^+ - \eta_j^-)\sigma'(\eta_j^-)$ used in the BP algorithm. The reason is illustrated in Figure B1. Note that it is a perfect approximation if the activation function is linear. The steeper the slope, the worse the approximation. Finally, the larger $\eta_j^+ - \eta_j^-$, the worse the approximation.

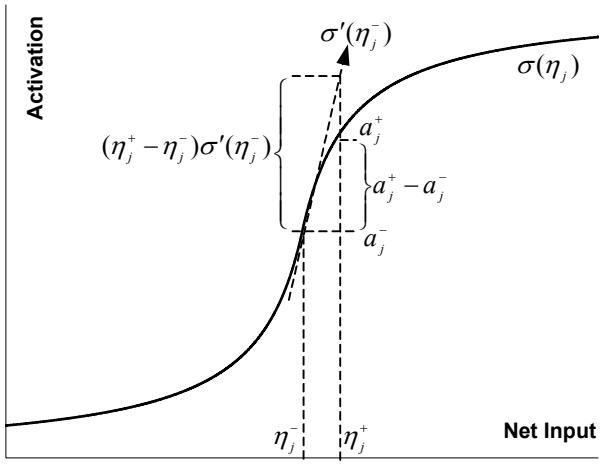


Figure B1 The difference in activation approximates the difference in net input times the derivative of the sigmoid activation function

It is important to note that the fundamental aspect of error driven learning is the credit assignment problem, i.e. to what extend a neuron is in error relative to other neurons in the same layer. This means that the absolute error is not essential to the credit assignment problem because there is also a learning rate parameter. The importance of the problem is the relation between errors among neurons. Failure to solve the credit assignment problem correctly may result in jumpy learning trajectories, to our knowledge, there is no formal proof of the credit assignment strategy in GeneRec. This is why we need to analyse it here. Assuming there are two neurons *i* and *j* in the hidden layer. We illustrate the credit assignment problem in Figure B2. Assume neuron *i* is less active than neuron *j* and the following holds:

- $\eta_j^+ - \eta_j^- = \eta_i^+ - \eta_i^-$.
- $\frac{\eta_j^+ - \eta_j^-}{2}$ and $\frac{\eta_i^+ - \eta_i^-}{2}$ are equally distant from the centre of the sigmoid.

We know that:

$$a_j^+ - a_j^- = a_i^+ - a_i^-$$
$$\sigma'(\eta_j^-) > \sigma'(\eta_i^-) \text{ and}$$
$$(\eta_j^+ - \eta_j^-)\sigma'(\eta_j^-) > (\eta_i^+ - \eta_i^-)\sigma'(\eta_i^-)$$

Hence, the following may not always hold.

$$\frac{a_j^+ - a_j^-}{a_i^+ - a_i^-} = \frac{(\eta_j^+ - \eta_j^-)\sigma'(\eta_j^-)}{(\eta_i^+ - \eta_i^-)\sigma'(\eta_i^-)}$$

This equation only holds if $\sigma'(\eta_j^-) = \sigma'(\eta_i^-)$. However, it is more interesting to analyse the general situation without making the assumptions, such as $\eta_j^+ - \eta_j^- = \eta_i^+ - \eta_i^-$. It is argued by [O'Reilly, 1996] that GeneRec will compute a good approximation of error derivatives of backpropagetion if the following holds:

- $\delta_j = (\eta_j^+ - \eta_j^-)\sigma'(\eta_i^-) \approx a_i^+ - a_i^-$
- The reciprocal weights are symmetric.

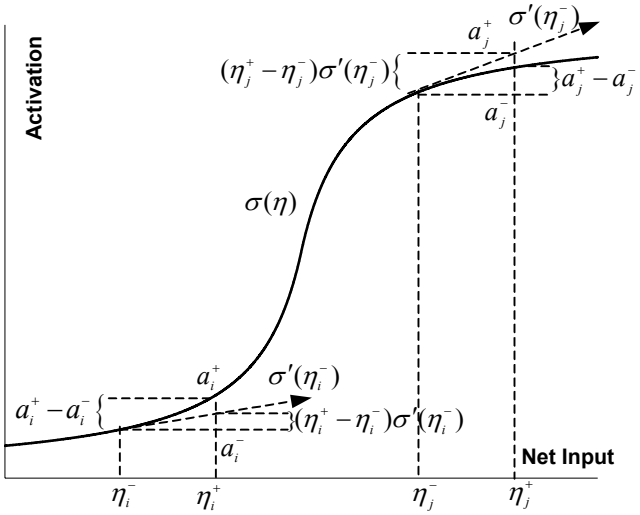


Figure B2 The difference in activation approximates the difference in net input times the derivative of the sigmoid

From Figure B1 we can see that $(\eta_j^+ - \eta_j^-)\sigma'(\eta_j^-) \approx (a_j^+ - a_j^-)$ only if $a_j^+ - a_j^-$ is small enough and both $\eta_i^+ - \eta_i^-$ and $\eta_j^+ - \eta_j^-$ are tending towards zero. As a result, the following holds.

$$\eta_j^+ - \eta_j^- \approx \eta_i^+ - \eta_i^- \text{ and } a_j^+ - a_j^- \approx a_i^+ - a_i^-$$

Then,

$$\frac{a_j^+ - a_j^-}{a_i^+ - a_i^-} \approx 1 \text{ and } \frac{\eta_j^+ - \eta_j^-}{\eta_i^+ - \eta_i^-} \approx 1$$

Hence,

$$\frac{a_j^+ - a_j^-}{a_i^+ - a_i^-} \approx \frac{(\eta_j^+ - \eta_j^-)\sigma'(\eta_j^-)}{(\eta_i^+ - \eta_i^-)\sigma'(\eta_i^-)} \text{ iff } \sigma'(\eta_j^-) \approx \sigma'(\eta_i^-)$$

This means that GeneRec and BP learning may use different credit assignment strategies. These two strategies are good approximations of each other only if all neurons in the same layer have similar derivatives of the sigmoid $\sigma'(\eta^-)$ at the minus phase. Unfortunately, this condition is not always valid during learning. In order to fully understand this difference we want to perform model checking to compare these two algorithms in respect of the credit assignment.

Appendix C

Consequences of Varying the Gain Parameter

In this section, we investigate the same network as in Figure 13 and try to obtain a set of suitable parameters, which allow the network to satisfy the stability property within the given deadline. Every trail of the experiment, the network is trained with a sequence of patterns. Between trials, we vary the gain parameter Φ. If the learning is successful, we can obtain the training epoch from the trace, which is generated by the model checker. We only consider the training epoch but not real-time, because we assume that the training data is pre-collected, i.e. it is always available. In this case, counting how many epochs have been trained is easier than verifying a real-time property. In a realistic situation, where the training data is generated on-line and outside the neural network, a real-time property may be considered.

Figure C1 shows the changes of learning speed, in terms of the epochs required, against different gain parameters. The network is not able to learn the XOR when $\Phi \leq 0.01$, i.e. verification found no satisfactory state. When $\Phi = 0.02$, it takes 281 epochs to learn the XOR problem. As we increase the gain parameter to around 0.04, the learning speed increases significantly. It reaches a lower bound, 26 epochs, when the gain parameter is around 0.1. The learning speed remains steady until the gain parameter is around 0.25, then it slows down gradually as the gain parameter increases. The learning is unsuccessful when $\Phi \geq 0.44$.

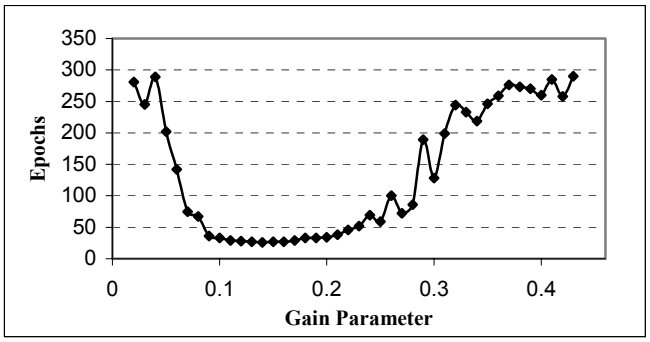


Figure C1 Learning speed vs. Gain parameter

The reason why the learning is unsuccessful when $\Phi \leq 0.01$ and $\Phi \geq 0.44$ is the following. In the standard BP algorithm, as the error is propagated through the network, we multiply the error by the derivative of the sigmoid function $\sigma'(a_i) = a_i - a_i^2$. This derivative is maximal when the activation of the neuron is around 0.5. It is minimal when the activation of the neuron is at the two extremes, i.e. 0 or 1. In this case, the weight change could be tiny even if the actual error is maximal. Moreover, the weight change could be zero when integer weights are used.

Consequences of Varying both the Learning Rate and the Gain Parameter

In this section we replicate the verification in the previous section, but we vary the learning rate and the gain parameter at the same time. Figure C2 (a) shows the result of training epochs with respect to different gain parameters and learning rates. Based on the original data, we plot how the covariance between training epoch and the reciprocal of the learning rate changes with different gain parameters in Figure C2 (b). Covariance measures the extent to which two sets of variables vary together. We use the following to examine whether greater learning rate accompanies greater learning speed.

$$Cov(Epoch, \varepsilon^{-1}) = \mu((Epoch - \mu(Epoch)) \cdot (\varepsilon^{-1} - \mu(\varepsilon^{-1}))),$$

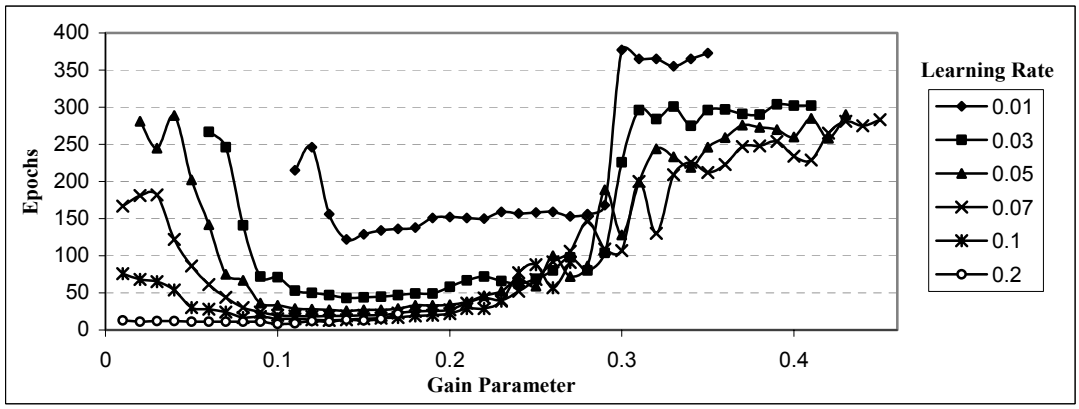
where $\mu()$ denotes the mean of the data set. A positive covariance indicates that higher learning speed tends to be paired with higher learning rate and greater covariance indicates a stronger relationship. In contrast, a negative covariance indicates that higher learning speed tends to be paired with lower learning rate and smaller covariance indicates stronger relationship. However, zero covariance does not necessarily indicate the two data sets are independent. It only indicates that there is a weak relationship.

Furthermore, it is difficult to compare covariances among data sets with different scales. We are in this situation. In order to overcome this limitation, we normalise the covariance to the product of the standard deviations of the data sets. The result is called the correlation coefficient. We also plot the correlation coefficient between training epoch and reciprocal of learning rate with different gain parameters in Figure C2 (c). The correlation coefficient determines the relationship between two sets of data. It is and is only a measure of linear association between variables, but not a measure of the strength of causality. We use the following to examine whether the learning rate and the learning speed are linearly related.

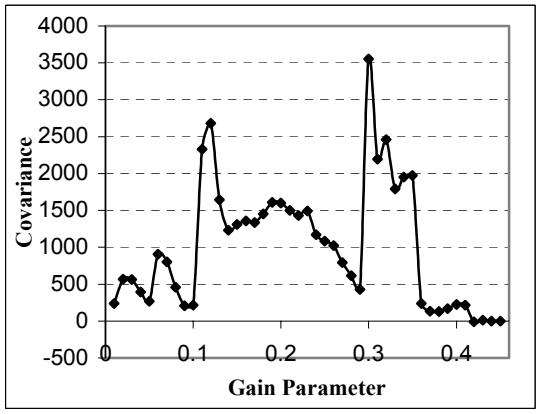
$$\rho(Epoch, \varepsilon^{-1}) = \frac{Cov(Epoch, \varepsilon^{-1})}{var(Epoch) \cdot var(\varepsilon^{-1})},$$

where *var*() denotes the standard deviation of the data set. The correlation coefficient ranges between -1 and 1. When $\rho(Epoch, \varepsilon^{-1}) = 1$, there is a perfect positive linear correlation between learning speed and learning rate. When $\rho(Epoch, \varepsilon^{-1}) = -1$, there is a perfect negative linear correlation between learning speed and learning rate. A zero correlation coefficient means there is no linear correlation, i.e. it is not always true that increase of learning rate leads to increase of learning speed. It is shown that learning speed and learning rate tend to linearly correlate in most of the cases except when the gain parameter is around 0.29.

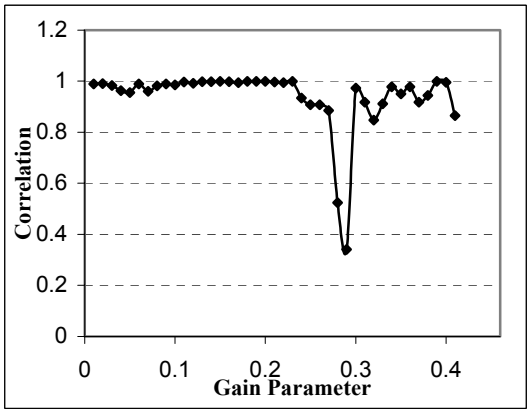
This investigation points out that the parameters are important factors causing the change of learning speed when the BP learning is used to solve the XOR problem. The learning quality, e.g. speed, accuracy and generalization, are highly dependent on a set of parameters, i.e. initial state, learning rate, threshold/gain parameter, number of hidden layers and hidden neurons. Verification can justify and help the selection of correct parameters, especially when there are both neural and non-neural components in the same controller.



(a)



(b)



(c)

Figure C2 (a) Training Epochs with Different Gain Parameters and Learning Rates, (b) Covariance between Training Epoch and Reciprocal of Learning Rate with Different Gain Parameters and (c) Correlation Coefficient between Training Epoch and Reciprocal of Learning Rate with Different Gain Parameters.

Appendix D

Range 1 is denoted R_1, where $R_1 = [0, Max] \subset \mathcal{N}$ and R_1 operations are $+, -, \times, \div$, and relations are $<, \leq, =$ and operator $[]_0^{Max}$, where

$$[y]_0^{Max} = \begin{array}{l} if\ \ y > Max\ then\ Max \\ else\ if\ \ y < 0\ then\ 0 \\ else\ y \end{array}$$

Range 2 is denoted R_2 and is the Uppaal arithmetic, where $R_2 = [-Max_u, Max_u] \subset \mathcal{Z}$ and Max_u is the Uppaal maximum integer value and $Max < Max_u$. R_2 operations are $+', -', \times', \div'$, and R_2 relations are $<', \leq', ='$, where $\div'$ is integer division, i.e. $a \div' b = \max\{c \in \mathcal{N} \mid c \leq a \div b\}$. Thus, $\div'$ rounds down, e.g. $3 \div' 4 = 0$ not 1.

Note, Uppaal checks a value is in range at the point at which on assignment is made, e.g.

$$d := Max_u + 1$$

is invalid and will generate a run time error, but

$$d := (Max_u + 1) - 1$$

is valid. Similarly,

$$\text{for } a, b \in R_2,\ d := a \times' b$$

is potentially invalid, but

$$d := (a \times' b) \div' Max$$

is certainly valid.

By contraction,

$$a + b, a - b, a \times b, a \div b \in R_1,$$

R_1 operations:

$$\begin{array}{c} \forall a, b \in R_1, \\ a + b \mathrel{\hat{=}} [a +' b]_0^{Max} \\ \wedge\ a - b \mathrel{\hat{=}} [a -' b]_0^{Max} \\ \wedge\ a \times b \mathrel{\hat{=}} (a \times' b) \div' Max \\ \wedge\ a \div b \mathrel{\hat{=}} [(a \times' Max) \div' b]_0^{Max} \end{array}$$

In order to ensure maximum discrimination we ensure that the truncation of $\div'$ and the ceiling and flooring of $[]_0^{Max}$ is applied at top-level of expressions. In addition,

$$\begin{array}{c} \forall a, b \in R_1 \\ a = b \mathrel{\hat{=}} a =' b \end{array}$$

$$\wedge\, a < b \mathrel{\hat{=}} a <' b$$
$$\wedge\, a < b \mathrel{\hat{=}} a \leq' b$$

The approach is also generalised as required to a $[-Max, Max] \subset \mathbb{Z}$ range in the obvious way.

Mapping between these two ranges:

$$f : R_1 \rightarrow R_2 \text{ defined as } f(a) = \frac{a}{Max}$$