Master Thesis

on

First-Order rule learning
through a
Pulsed Neural Network

by
Ashish Darbari
e-mail: `ashish@pikas.inf.tu-dresden.de`

Supervisor: Prof. Dr. rer. nat. habil. Steffen Hölldobler

Department of Computer Science
Dresden University of Technology
Dresden, Germany

To my late grandfather
D S Darbari

Statement of Academic Honesty

I hereby declare that I have not used any auxiliary means for my thesis work other than that what has been cited in my thesis.

Dresden, Winter '00

Ashish Darbari
Matr. No: 2668822

Acknowledgement

Its my pleasure to thank Prof Steffen Hölldobler for providing me an opportunity to work in this exciting field of research. I thank him for giving me total freedom to think and pursue the work in the way I wanted, and at the same time always chipped in with useful suggestions and critics.

This work would not have been possible, had I not got valuable and timely contributions from the following researchers:

- Prof Wulfram Gerstner
- Prof Krysia Broda
- Dr Lokendra Shastri
- Dr Bernd Fritzke
- Mr Artur Garcez

Prof Gerstner at Centre for Neuro-Mimetic Systems, Lausanne, Switzerland, was always available for discussions through email. He not only provided significant contribution through his research paper on learning, but also helped me with very useful comments during my initial stages of work. I thank him also for providing me some of the figures, related to his experimental work with the Barn Owl Auditory System.

Prof Krysia Broda and Mr Artur Garcez at Imperial College, London provided me an opportunity to work with them for a short time. It was through discussions on the existing work on structured connectionist systems, that we realized the problems in some of the existing models.

Dr Shastri at ICSI, Berkley, provided useful starting point for the whole work. His ideas of using temporal synchrony for dynamic variable binding were quite useful. He gave useful insights into the concept of modelling reflexive reasoning in connectionist systems through the SHRUTI system.

Dr Fritzke at our university provided useful critics and showed some possibilities for future work.

My special thanks to my friends — Andreas Henschel and Alwen F. Tiu, for going through my manuscript, and pointing out discrepancies.

Thanks are due, to Dr Alessio Guglielmi and Mr Axel Grossmann for tipping me with timely advice and suggestions.

Last but not the least, I thank my parents for making it all happen.

Contents

List of Figures

Preface

The area of Rule Extraction has recently drawn much attention in the machine learning community. This can be attributed mainly because of its successful application in the area of computational molecular biology. Rule extraction process is a part of a broader paradigm of hybrid machine learning. Hybrid machine learning combines the nice properties of inductive and analytical learning. The extraction of refined rules from a trained neural network serves to refine the domain theory.

The extraction of propositional *if-then* rules from neural networks trained by standard backpropagation learning techniques has found many successful applications not only in protein prediction and molecular biology but also in power system control. However, the size of extracted rule-set is usually too large to be comprehended well. If one uses the expressive power of first-order logic, it not only helps to reduce the size of rule-set, but also allows us much richer expressivity. This leads us to the point where we start our work in the area of first-order rule extraction.

We make an analogy to the standard way of implementing rule-extraction at the level of propositional logic, by starting with the construction of a knowledge based neural network. However, the problem here is not only to encode the structure of the first-order logic program, but also to allow unification — between *terms* (and *variables*) of first-order logic. The idea to perform unification is borrowed from Lokendra Shastri and Venkat Ajjanagadde's SHRUTI system [111], where temporal synchrony was used to infer a dynamic variable binding. By encoding the structure of the first-order logic theory in a neural network, we produce a knowledge based neural network. The next step is to train this network on a set of examples, by using a learning algorithm. After training the network we reach the point where we can extract rules from the trained network.

I propose a transformation algorithm to encode a small subset of first-order logic program to a neural network. This network is not an ordinary sigmoidal unit feed-forward network. In this network all the neurons or units exhibit temporal behaviour. Any neuron in this network receives a spiking input at a certain time point and the neuron responds by producing a spike at a certain time point. We see plenty of neuro-biological motivations for doing such an encoding. Moreover, by working within the realm of spiking networks, we are in a much better position to make the knowledge based neural networks learn. We use a variant of Hebbian type learning for training the network. The algorithm that we use is the one experimented for learning in Barn Owl Auditory System by Wulfram Gerstner and his group in [123].

By means of examples I illustrate the novel concept of first order rule-learning through a spiking neural network.

As a useful side effect of this approach, we get a new model which gives us useful insights as to how we humans could possibly map structure in the human brain to learn and perform day-to-day common-sense reflexive reasoning.

This work should be viewed as the first steps rather than an end, towards understanding low-level cognition within the integrated framework of symbolic reasoning and neural networks. Moreover, the examples show that it is possible to do first-order rule learning and extract refined rules from the trained networks, thus leading way for more exciting research to be done in the field of rule-extraction.

Dresden, Autumn '00 *Ashish Darbari*

Chapter 1

Neural Symbolic Integration

In this chapter we have a look at some of the most compelling reasons for integrating neural and symbolic paradigms. We discuss the reasons of integration in the light of machine learning. It is believed that learning could be basically classified into two different kinds of techniques inductive and analytical. Each has its own set of advantages and disadvantages. Theoretical and practical motivations are presented in this chapter and the general framework for hybrid learning is explained. In the subsequent section an informal introduction to first–order logic is presented. Lastly in this chapter, we study the general problems of embedding first-order logic in connectionist systems and derive motivations for the work that follows in the next chapters.

1.1 Introduction

Since the computers were first invented we have wondered if ever it would be possible for them to learn. We know that by programming them to learn - to improve automatically with experience - the impact would be dramatic just imagine a similar case of an infant who learns and learns over a span of few years to perform tasks of high complexity with tremendous ease. Imagine having a computer learn from medical records which treatments are most effective for new diseases, homes learning from experience to optimize energy costs based on the particular usage pattern of their occupants, or personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the on-line morning newspaper. We do not yet know how to make computers learn nearly as well as people learn, however with the emergence of algorithms which are effective for certain types of learning tasks a theoretical understanding of learning is beginning to emerge. How do we then define the task of learning ?

Definition 1 A computer program is said to *learn from experience* E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself. In general, to have a well defined learning problem we should identify these three features: *the class of tasks*, *the measure of performance to be improved*, and *the source of experience*. Designing a machine learning approach involves a number of design choices, including choosing the type of training experience, the *target function* to be learned, a representation for this target function, and an algorithm for learning the target function from training examples. Throughout the machine learning paradigm it is useful to reduce the problem of improving performance P at a task T to the problem of learning a function which is called a target function. Intuitively, a target function maps the intended problem solution as a concrete mathematical function which the system is expected to learn. As an example, in the checkers learning problem a possible target function V could map any legal board state from a set of legal states B to some real value in a set of real numbers denoted by $\mathcal{R}$. Learning involves search: searching through a space of possible hypothesis, to find the hypothesis that best fits the available training examples and other prior constraints or knowledge. The different machine learning approaches can be basically classified into *Inductive Learning* and *Analytical Learning*. The subsequent paragraphs investigate the details of these methods and then proceeds to justify the combination of these two methods in order to have a sufficiently general learning scheme which is called the *hybrid learning* framework.

1.2 Inductive Learning

The phrase Inductive Learning is used for the class of methods that generalize from observed training examples by identifying features that empirically distinguish positive from negative training examples. Decision tree learning

[94], neural network learning such as *backpropagation* [10, 130, 89, 24, 25], inductive logic programming [78], and genetic algorithms are all examples of inductive methods that operate in this fashion. Pure inductive learning methods formulate general hypothesis by finding empirical regularities over the training examples. The key practical limit on these inductive learners is that they perform poorly when insufficient data is available. In fact there are theoretical analysis tools to show that there are fundamental bounds on the accuracy that can be achieved when learning inductively from a given number of training examples. In inductive learning, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D=\{\langle x_1,f(x_1)\rangle,\ldots\ \langle x_n,f(x_n)\rangle\}$ where $f(x_i)$ is the target value for the instance x_i. The desired output of the learner is a hypothesis h from H that is consistent with these training examples.

1.3 Analytical Learning

Analytical learning uses prior knowledge and deductive reasoning to augment the information provided by the training examples, so that it is not subject to these same bounds. In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input - a *domain theory*. A domain theory B consists of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B. As an example consider a chess playing scenario where each instance x_i would describe a particular chess position, and $f(x_i)$ would be *True* when x_i is a position for which black will lose its queen within two moves, and *False* otherwise. A hypothesis space H might be defined to consist of sets of Horn clauses (if-then rules). The domain theory B would consist of a formalization of the rules of chess, describing the legal moves, the fact that players must take turns, and the fact that the game is won when one player captures her opponent's king. It is important in analytical learning that the learner must output a hypothesis that is consistent with *both* the training data and the domain theory. The notion of consistency is a usual one where a hypothesis h is consistent with domain theory B provided $B \cup h \nvdash \perp$. In other words the hypothesis together with domain theory should not generate a contradiction. This additional constraint that the output hypothesis must be consistent with B reduces the ambiguity faced by the learner when the data alone cannot resolve among all hypothesis in H. The net effect, provided the domain theory is correct, is to increase the accuracy of the output hypothesis. A very special form of analytical learning is Explanation-Based Learning (EBL). In explanation based learning, prior knowledge is used to analyze, or explain how each observed training example satisfies the target concept. This explanation is then used to distinguish the relevant features of the training example from the irrelevant, so that examples can be generalized based on logical rather than statistical reasoning.

1.4 The Need for Synergy

Two distinctly different machine learning paradigms have been examined above and the way they work gives us enough intuition to delve into the possibility of combining them. These two learning paradigms are based on fundamentally different justifications for learned hypothesis and offer complementary advantages and disadvantages. Can a combination of these two give us a powerful learning tool ? The subsequent section discusses this in detail.

1.4.1 Theoretical Motivation

Purely analytical learning methods offer the advantage of generalizing more accurately from less data by using prior knowledge to guide learning. However, they can be misled when given incorrect or insufficient prior knowledge. It is rarely the case that the real time domain theories are perfect, on the contrary most of the time the domain theory is imperfect. Purely inductive methods offer the advantage that they require no explicit prior knowledge and learn regularities based solely on the training data. However, they can fail when given insufficient training data, and can be misled by the implicit inductive *bias* they must adopt in order to generalize beyond the observed data. Bias can be understood as an apriori assumption regarding the identity of the target concept in order to have a rational basis for classifying any unseen instances of data. The inductive bias can be formally defined under the following setting: Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let D_c $= \{\langle x_1, c(x_1)\rangle, \ \ldots, \langle x_i, c(x_i)\rangle\}$ be an arbitrary set of training examples of

c. Let $L(x_i, D_c)$ denote the classification assigned to the instance x_i by L after training on the data D_c. The inductive bias of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c

$$(\forall x_i \in X)(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c) \tag{1.1}$$

The difference between inductive and analytical learning methods can be seen in the nature of the *justifications* that can be given for their learned hypothesis [77]. Hypothesis output by purely analytical learning methods carry a *logical* justification; the output hypothesis follows *deductively* from the domain theory and the training examples. Hypothesis output by purely inductive learning methods such as backpropagation carry a *statistical* justification; the output hypothesis follows from statistical arguments that the training sample is sufficiently large that it is probably representative of the underlying distribution of examples. Given that analytical methods provide logically justified hypothesis and inductive methods provide statistically justified hypotheses, it is easy to see why combining them could be useful. Logical justifications are only as compelling as the assumptions, or prior knowledge, on which they are built. They are suspect or powerless if prior knowledge is incorrect or unavailable. Statistical justifications are only as compelling as the data and statistical assumptions on which they rest. They are suspect or powerless when assumptions about the underlying distributions cannot be trusted or when the data is scarce.

1.4.2 Practical Motivation

Artificial Neural Networks (ANN) have been successfully applied in the area of speech generation [110], speech recognition [129], vision and robotics [90, 42], handwritten character recognition [58], medical diagnostics [50] and game playing [115] . However their inability to produce plausible explanations render them useless for safety-critical applications like Power System Protection where remedial actions have to be taken based upon the explanation generated for a fault. On the other hand symbolic learning systems have been used successfully to solve problems such as learning which chemical bonds fragment in a mass spectrometer [32, 55], learning which chemical substructures produce mutagenic activity (a property related to carcinogenicity) [5], and learning to design finite element meshes to analyze stresses in physical structures [8]. Systems like Prolog-EBG [104] construct useful intermediate features as a side effect of analyzing individual training examples. This analytical approach to feature generation compliments the statistically based generation of intermediate features (e.g., hidden unit features) in inductive methods such as backpropagation. One important class of problems for which a correct and complete domain theory can be found is the class of large state-space search problems. In such a scenario, we have a very strong motivation to guide the integration of these two forms of learning. An issue of foremost importance at this point is to emphasize that much of the research is focussed on combining *neural network* learning as the inductive form of learning and *rule based* learning for symbolic learning. In the subsequent paragraphs a general framework for this integration has been studied.

1.5 The General Hybrid Learning Framework

There is a significant gap between the knowledge-intensive, learning-by-being-told approach of hand built classifiers and the virtually knowledge-free approach of inductive learning. Some of this gap is filled by *hybrid* learning methods, which use both hand-constructed rules and classified examples during learning. Several trends have made the development of such systems an active area in machine learning. Perhaps the most important of these trends is the realization that knowledge intensive and knowledge-free learning are just two ends of a spectrum along which an intelligent system may operate. Staying at one end or the other of the spectrum of possible learning systems simplifies the learning problem by allowing strong assumptions to be made about the nature of what needs to be learned. However, the middle ground is appealing; it offers the possibility that synergistic combinations of theory and data will result in powerful learning systems.

Another trend spurring the development of hybrid systems is the growth of a body of psychological evidence that people rarely, if ever, learn purely from theory or examples. For instance, Murphy and Medin suggest that "feature correlations are partly supplied by people's theories and that the causal mechanisms contained in theories are the means by which correlational structure is represented." That is theory and examples interact closely during human learning. While it is clear that people learn from both theory and examples, the way in which the interaction occurs is yet to be determined. Jude and Shavlik in [113] have defined the scope of hybrid learning by the following three items:

- Knowledge Insertion into ANN (Rules to ANN)
- Refinement of this prior knowledge in its neural representation (Training the ANN)
- Refined Rule Extraction from trained ANN (Rule Extraction Problem)

Knowledge when expressed in the form of propositional logic programs can be encoded into an ANN by a well detailed algorithm [113]. Refinement of such knowledge within the ANN implies training the ANN with the data sets, by using some learning algorithms. One of the typical algorithm found very useful for training a network obtained from a propositional logic program is the backpropagation algorithm. Several groups have exploited the standard backpropagation or one of its variants to train such kind of Knowledge Based Networks [113], [121], [21], [7]. After these networks are trained on a rather noisy data set, one focuses on the rule extraction part. At the level of propositional logic, there are several solutions proposed to the problem of rule extraction. The solutions are robust in the sense that they can also deal with noisy data and also fulfill some of the nice properties like expressivity, translucency etc [20]. However, the issue of *comprehensibility* is not so well addressed. Even the very best and the latest work by [21] fails to reduce the size of the rule set. This in turn has serious implications because if the neural network was criticized for being abstruse, then the solution coming from the hybrid learning schemes is at least so huge in size that even though they are logical if-then propositional rules, one can't really expect a person to read and understand them, in order to know what the system is doing. In-spite of several post processing optimizations being employed there is a bottleneck beyond which the size of the rule set can't be reduced [121], [122].

1.6 The First Order Logic Framework

The language of plain propositional logic is too weak to address the "generality" feature of a rule set and capture "relations". This goads us to think of a more powerful language, the expressive power of first order logic. The language of first order logic provides a richer syntax consisting of a set of *variables*, set of n - *ary function symbols*, set of *terms* constructed over the set of variables and the set of *function symbols* and a set of *relation symbols*.

Apart from this vocabulary we can make use of universal quantifier ($\forall$) and existential quantifier ($\exists$) to represent general sentences or rules e.g. "All men who are above 40 years old are prone to cardiac ailments", or "If John has a friend having a 100 dollar note, John will get it".

By using the first order logic we can use variables to denote a placeholder. This placeholder could be instantiated by terms (or constants). Constants are nullary function symbols used to denote objects of our domain with whom we want to reason. Consider for example the following rule set *Transaction*:

$$\begin{aligned}\mathcal{P} = \{ & \mathit{gives(andreas,\ book,\ muna)} \rightarrow \mathit{gets(muna,\ book)}, \\ & \mathit{gives(janis,\ pen,\ alwen)} \rightarrow \mathit{gets(alwen,\ pen)}, \\ & \mathit{gives(prakash,\ pizza,\ thomas)} \rightarrow \mathit{gets(thomas,\ pizza)}\}\end{aligned}$$

The set of constants $\mathcal{C}$ and the set of relation symbols $\mathcal{R}$ are given below:
$\mathcal{C} = \{\mathit{andreas,\ alwen,\ janis,\ muna,\ prakash,\ thomas,\ book,\ pen,\ pizza}\}$
$\mathcal{R} = \{\mathit{gives,\ gets}\}$

Instead of writing 3 rules above as in $\mathcal{P}$, we can write one rule:
$\forall XYZ.\, gives(X, Y, Z) \rightarrow gets(Z, Y)$

From the above it is quite clear how easily relations are expressed, and the size of rule set is decreased, when we use the expressiveness of first order logic. Semantics of a first order logic program can be understood by constructing *models* or sets of interpretations where grounded instances of relations are a member of this set if the relation has a boolean value of true, and not if otherwise. First order logic employs what is called as *unification*, a process of instantiating variables by terms, and possibly generating grounded instances of relations. This process of unification is a way to become more specific from a general context, although in some cases unification can just produce a renaming of a variable. However as we will shortly see, it is this unification mechanism which is the most crucial factor in determining how a first order logic rule could be encoded in a connectionist system. There are some ways suggested to encode the relation symbols in a connectionist network[1] and also to explain how unification could take place dynamically during a process of inference. But as we will see in later sections, there are problems with the present approaches when it comes to learning such kind of connectionist systems.

[1]throughout our discussion the phrase connectionist network would mean a neural network

1.7 Encoding First Order Logic in a Connectionist Network

The first steps towards encoding the first order clause were quite impressive in a way that they addressed two important aspects of representation, the depiction of relation symbols, and their corresponding arguments and a way to explain unification. The motivation for representing a predicate or relation symbol and other syntactic constructs of first order logic came from the understanding of how humans could do reasoning and perception. Insights from researchers studying oscillations in neural systems [27] suggested that neurons in different parts of the brain oscillate in synchrony in response to certain stimuli. One hypothesized function for such synchronous activity was to solve the binding problem in human perception.

However it was Fahlman in [31], who showed that by using *temporal synchrony* in a connectionist model one can support inference. Temporal synchrony was represented as the notion of marker passing. Each node in the network stores a small number of discrete markers. These markers propagate between nodes under the supervision of the *network controller* to achieve dynamic bindings and symbolic inference. Thus, nodes in his system were required to have high computational abilities, in order to store, match and selectively propagate marker bits. Later Clossman in [12] used synchronous activity to represent argument-filler bindings for a connectionist model of categorization and learning, but without an efficient encoding mechanism for rules and facts. It was not until Shastri and Ajjanagadde (S & A) in [6] and in [111] proposed a connectionist model for reflexive reasoning called SHRUTI, that we have an efficient connectionist mechanism to represent dynamic bindings. It was based on the idea of temporal synchrony and other mechanisms to encode rules and facts involving n - *ary* predicates. Shastri and Ajjanagadde in their 1993 paper explained the view of information processing as implied by SHRUTI. Infact their view was widely accepted as a defacto, and since then several researchers like in [88], [102] have borrowed the key idea of dynamic variable binding in their models. The key features of the view proposed by S & A is elucidated below:

- reasoning is the transient but systematic propagation of a rhythmic pattern of activity,
- each entity in the dynamic memory is a phase in this rhythmic activity,

- dynamic bindings are represented as the synchronous firing of appropriate nodes,
- long-term facts are subnetworks that act as temporal pattern matchers, and
- rules are interconnection patterns that cause the propagation and transformation of rhythmic patterns of activity

It is implausible to represent dynamic bindings by using structural changes, pre-wired interconnection networks, or by communicating names/pointers of arguments and fillers. Instead, what is required is a neurally plausible mechanism for rapidly and temporarily labeling the representations of fillers and predicate arguments to encode dynamically argument-filler bindings. Also required are mechanisms for systematically propagating such transient labels and allowing them to interact with long-term structures. It is true that SHRUTI addresses all the above issues, however as pointed out by Park et.al in [88], SHRUTI represents variable bindings by inactivity of argument nodes during inference. It does not therefore differentiate one variable binding from others, which according to Park et.al is a basic and fundamentally important feature of symbolic inference. Second point mentioned by Park et.al is that SHRUTI's fact-encoding mechanisms do not consider all the conditions that symbolic rules and facts may impose. In particular he mentions that a sub-mechanism for binding interaction between arguments is not considered. Park et.al in [88] illustrate that their model can represent groups of unifying arguments even when it is not known to which individual the variable in those arguments is bound. The proposed fact-encoding mechanism by Park et.al [88] does not need different network configurations for yes-no and wh-type query processing as it was the case of SHRUTI. In a separate motivation Hölldobler et.al addressed the representational aspect of encoding a first order horn clause program and they observed that SHRUTI's structure is quite complex, because there are many different types of units with a complex connection structure. Hölldobler et.al propose a new calculus called Bottom Up Reduction Calculus (BUR) where they propose a mechanism to construct connectionist networks from a restricted form of first order logic, and the connectionist network computes a least fixed point $\mathcal{T}_P$ of a first order logic program with constants and multi-place relations. Their calculus is sound and complete for unary relation symbols, and complete but not necessarily sound for multi-place relation symbols.

In all the above approaches however the algorithms proposed for constructing connectionist networks from first order logic programs even with some restrictions is *local* in nature. All of them use the special ρ, π, τ units as in [111], [88] which have temporal behaviour and serves to selectively switch on the units and parts of network and leads to efficient unification mechanisms and consistency checking [88]. The whole network however is so complex that the task of training such a network by using a learning algorithm from neural network paradigm is very difficult. Even developing an algorithm for such networks which may not be a typical neural network learning algorithm is extremely difficult because the networks are so local in representations, and are not even hierarchically layered. Infact its fair that these algorithms don't have such capabilities, because their design was not motivated from a Rule Extraction Task, which is our goal in the last step of the neural-symbolic integration.

Hence in a later chapter I propose another method of representing symbolic-connectionist networks. The idea is to have an architecture which is distributed and at the same time simple and layered and can be trained by a neural learning algorithm, thereby taking advantage of the robust learning algorithms we have in neural networks.

Chapter 2

Encoding Information in Neuronal Activity

The majority of artificial neural networks are based on a computational paradigm involving the propagation of continuous variables from one processing unit to the next. In recent years, however, data from neuro-biological experiments have made it increasingly clear that biological neural networks, which communicate through pulses, use the timing of these pulses to transmit information and to perform computation. Even though a lot of detailed information about neurons and their connections is available by now, one of the fundamental questions of neuroscience — what is the neural code? is unsolved. This chapter discusses some of the most recent research in this area with an aim to try to find answers to the following questions: What is the code used by the neurons? Do neurons communicate by a 'rate code' or a 'pulse code'?

2.1 The Problem of Neural Coding

The elementary processing units in the brain are *neurons* which are connected to each other in an intricate pattern. A typical neuron has three parts, called *dendritic tree*, *soma* and *axon*. Loosely speaking, signals from other neurons arrive onto the dendritic tree and are transmitted to the soma and the axon. The transition zone between the soma and the axon is of special interest. In this area the essential non-linear processing step occurs. If the total excitation caused by the input is sufficient, an output signal is emitted which is propagated along the axon and its branches to other neurons. The junction between an axonal branch and the dendrite (or the soma) of a receiving neuron is called a *synapse*. It is common to refer to the sending neuron as the *pre-synaptic* neuron and to the receiving neuron as the *post-synaptic neuron*. A neuron in the cortex often makes connections to more than 10^4 postsynaptic neurons. Many of its axonal branches end in the direct neighborhood of the neuron, but the axon can also stretch over several millimeters and connect to neurons in the other areas of the brain. Neurons transmit signals along the axon to thousands of other neurons, but what do these signals look like? The neuronal signal can be observed by placing a fine electrode close to the soma or axon of a neuron. The voltage trace in a typical recording shows a sequence of short pulses, called action potentials or *spikes*. A chain of pulses emitted by a single neuron is usually called a spike train - a sequence of stereotyped events which occur at regular or irregular intervals. The duration of an action potential is typically in the range of 1-2 ms. *Since all spikes of a given neuron look alike, the form of the action potential does not carry any information. Rather it is the number and the timing of spikes which matter.*

Definition 1 Firing time: The firing time of a neuron is defined as the moment when a given neuron emits an action potential. The firing time of neuron i will be denoted by ${t_i}^f$. The spike train of a neuron i is fully characterized by the set of firing times

$$\mathcal{F} = \{{t_i}^1, \dots, {t_i}^n\} \tag{2.1}$$

where ${t_i}^n$ is the most recent spike of neuron i.

A spike train may be described as a sequence of ones and zeroes for "spike" and "no spike" at times $\Delta t, 2\Delta t, \dots$, respectively. We divide the global clock of continous time into short discrete intervals of time known as Δt. The choice of ones and zeroes is of course arbitrary. We may just as well take the number $1/\Delta t$ instead of unity to denote the occurrence of a spike. With this definition, the spike train of a neuron i corresponds to a sequence of numbers $S_i(\Delta t), S_i(2\Delta t), \dots$, with

$$S_i(n\Delta t) = \begin{cases} 1/\Delta t & \text{if } n\Delta t \leq {t_i}^f \leq (n+1)\Delta t \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

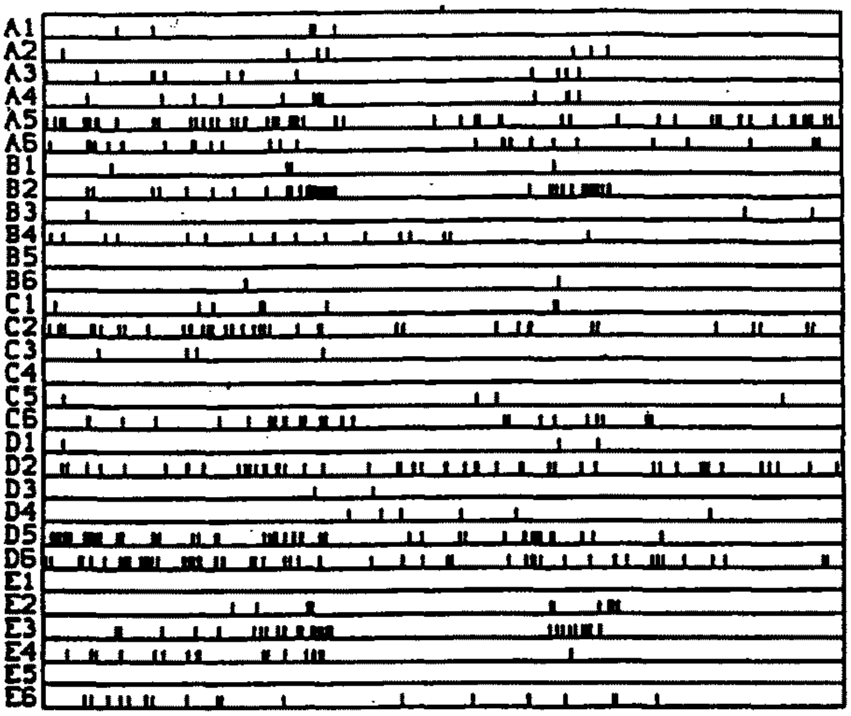


Figure 2.1: Spatio-temporal pulse pattern. The spikes of 30 neurons (A1-E6, plotted along the vertical axes) are shown as a function of time (horizontal axis, total time is 4000 ms). The firing times are marked by short vertical bars. Taken from [73]

On taking the limit $\Delta t \to 0$, we can write the spike train as a sequence of $\delta-$functions

$$S_i(t) = \sum_{t_i{}^f \in \mathcal{F}_i}^{n} \delta(t - t_i{}^f) \tag{2.3}$$

where $\delta(.)$ denotes the Dirac δ function with $\delta(s) = 0$ for $s \neq 0$ and

$\int_{-\infty}^{\infty} \delta(s)ds = 1$
Since there are so many neurons in the brain, thousands of spike trains are emitted constantly by different neurons (see Figure 2.1) What is the information contained in such a spatio-temporal pattern of pulses? What is the code used by the neurons to transmit that information? How might other neurons decode the signal? As external observers, can we read the code, and understand the message of the neuronal activity pattern? At present, a definite answer to these questions is not known. In the subsequent sections we present some of the potential ways in which some of the first solutions were proposed.

2.2 Potential Coding Schemes

Traditionally it has been thought that most, if not all, of the relevant information was contained in the mean firing rate of the neuron. The firing rate is usually defined by a temporal average. The experimentalist sets a time window of lets say $T = 100\,ms$ or $T = 500\,ms$ and counts the number of spikes $n_{sp}(T)$ that occur in this time window. Dividing these number of spikes by the length of time window gives the mean firing rate

$$\nu = \frac{n_{sp}(T)}{T} \tag{2.4}$$

usually reported in units of s^{-1} or Hz. The concept of mean firing rates has been successfully applied during the last 80 years. It dates back to the pioneering work of Adrian [13] who showed that the firing rate of stretch receptor neurons in the muscles is related to the force applied to the muscle. In the following decade, measurement of firing rates became a standard tool for describing the properties of all types of sensory or cortical neurons, partly due to the relative ease of measuring rates experimentally. It is clear, however, that an approach based on temporal average neglects all the information possibly contained in the exact timing of spikes. It is therefore no surprise that the firing rate has been repeatedly criticized and is subject of an ongoing debate [61, 120, 44, 79, 98, 30]. During recent years, more and more experimental evidence has accumulated which suggests that a straightforward firing rate concept based on temporal averaging may be too simple for describing brain activity. One of the main arguments is that reaction times in behavioral experiments are often too short to allow slow temporal averaging [109]. More-over, in experiments on a visual neuron in the fly, it was possible to 'read the neural code' and reconstruct the time dependent stimulus based on the neurons firing times [120]. There is evidence of precise temporal correlations between pulses of different neurons [61, 93] and stimulus dependent synchronization of the activity in populations of neurons [92, 65, 64, 83, 51, 127]. Most of these data is inconsistent with a naive concept of coding by mean firing rates where the exact timing of spikes should play no role.

2.2.1 Rate Coding

A quick glance at the experimental literature reveals that there is no unique and well defined concept of 'mean firing rate'. In fact, there are at least three different notions of rate which are often confused and used simultaneously. The three definitions refer to three different averaging procedures: either an average over time, or an average over several repetitions of the experiment, or an average over a population of neurons. For an excellent discussion of rate coding one can refer [30].

2.2.1.1 Rate as a Spike Count (Average over Time)

The first and most commonly used definition of a firing rate refers to a temporal average. As discussed in the preceding section, this is essentially the spike count in an interval T divided by T. The length of the time window is set by the experimenter and depends on the type of neuron recorded from and the stimulus. In practice, to get sensible averages, several spikes should occur within the time window. Values of $T = 100\,ms$ or $T = 500\,ms$ are typical, but the duration may also be longer or shorter.

This definition of rate has been successfully used in many preparations, particularly in experiments on sensory or motor systems. A classical example is the stretch receptor in a muscle spindle [13]. The number of spikes emitted by the receptor neuron increases with the force applied to the muscle. Another example is the touch receptor in the leech [11]. The stronger the touch stimulus, the more spikes occur during a stimulation period of 500 ms. These classical results show that the experimenter as an external observer can evaluate and classify neuronal firing by a spike count measure – but is this really the code used by neurons in the brain? In other words, is a neuron which receives signals from a sensory neuron only looking at and reacting to the numbers of spikes it receives in a time window of, say 500 ms?

From behavioral experiments it is known that reaction times are often rather short. A fly can react to new stimulus and change the direction of flight within 30-40 ms; see the discussion in [30]. This is not long enough for counting spikes and averaging over some long time window. It follows that the fly has to react to single spikes. Humans can recognize visual scenes in just a few hundred milliseconds [109], even though recognition is believed to involve several processing steps. Again, this leaves not enough time to perform temporal averages on each level.

Temporal averaging can work well where the stimulus is constant or slowly moving and does not require a fast reaction of the organism and this is the situation usually encountered in experimental protocols. Real world input, however, is hardly stationary, but often changing on a fast time scale. For example, even when viewing a static image, we perform saccades, rapid changes of the direction of gaze. The retinal photo receptors receive therefore every few hundred milliseconds a new input.

Despite its shortcomings, the concept of a firing rate code is widely used not only in experiments, but also in models of neural networks. It has led to the idea that a neuron transforms information about a single input variable (the stimulus strength) into a single continuous output variable (the firing rate). In this view, spikes are just a convenient way to transmit the analog output over long distances. In fact, the best coding scheme to transmit the value of the rate ν would be by a regular spike train with intervals $1/\nu$. In this case, the rate could be reliably measured after only two spikes. From the point of view of rate coding, the irregularities encountered in real spike trains of neurons in the cortex must therefore be considered as noise. In order to get rid of the noise and arrive at a reliable estimate of the rate, the experimenter (or the postsynaptic neuron) has to average over a larger number of spikes. A critical discussion of the temporal averaging concept can be found in [79, 98, 30].

2.2.1.2 Rate as a Spike Density (Average over Several Runs)

There is a second definition of rate which works for stationary as well as for time–dependent stimuli. The experimenter records from a neuron while stimulating with some input sequence. The same stimulation sequence is repeated many times and the results are reported in a Peri Stimulus Time

Histogram (PSTH). For each short interval of time $[t, t+\Delta t]$, before, during, and after the stimulation sequence, the experimenter counts the number of times that a spike has occurred and sums them over all repetitions of the experiment. The time t is measured with respect to the start of the stimulation sequence and Δt is typically in the range of one or a few milliseconds. The number of occurrences of spikes $n(t; t + \Delta t)$ divided by the number K of repetitions is a measure of the typical activity of the neuron between time t and $t + \Delta t$. A further division by the interval length Δt yields the spike density of the PSTH

$$\rho(t) = \frac{1}{\Delta t} \frac{n(t; t + \Delta t)}{K} \tag{2.5}$$

Sometimes the result is smoothed to get a continuous 'rate' variable. The spike density of the PSTH is usually reported in units of Hz and often called the (time dependent) firing rate of the neuron. As an experimental procedure, the spike density measure is a useful method to evaluate neuronal activity, in particular in the case of time dependent stimuli. The obvious problem with this approach is that it can not be the decoding scheme used by neurons in the brain. Consider for example a frog which wants to catch a fly. It can not wait for the insect to fly repeatedly along exactly the same trajectory. The frog has to base its decision on a single 'run' – each fly and each trajectory is different.

Nevertheless, the experimental spike density measure can make sense, if there are large populations of neurons which are independent of each other and sensitive to the same stimulus. Instead of recording from a population of N neurons in a single run, it is experimentally easier to record from a single neuron and average over N repeated runs. Thus, the spike density coding relies on the implicit assumption that there are always populations of neurons and therefore leads to the third notion of a firing rate, viz., a rate defined as a population average.

2.2.1.3 Rate as Population Activity (Average over Several Neurons)

The number of neurons in the brain is huge. Often many neurons have similar properties and respond to the same stimuli. For example, neurons in the primary visual cortex of cats and monkeys are arranged in columns of cells with similar properties [37, 38, 36]. Let us idealize the situation and consider a population of neurons with identical properties. In particular, all neurons in the population should have the same pattern of input and output connections. The spikes of the neurons in a population j are sent off to another population k. In our idealized picture, each neuron in population k receives input from all neurons in population j. The relevant quantity, from the point of view of the receiving neuron, is the proportion of active neurons in the presynaptic population j. Formally, we define the population activity

$$A(t) = \frac{1}{\Delta t} \frac{n_{act}(t; t + \Delta t)}{N} \tag{2.6}$$

where N is the size of the population, Δt a small time interval, and $n_{act}(t; t+\Delta t)$ the number of spikes (summed over all neurons in the population) that occur between t and $t + \Delta t$. If the population is large, we can consider the limit $N \to \infty$ and then take $\Delta t \to 0$. This yields again a continuous quantity with units s^{-1} – in other words, a rate.

The population activity may vary rapidly and can reflect changes in the stimulus conditions nearly instantaneously [117]. Thus the population activity does not suffer the disadvantages of a firing rate defined by temporal averaging at the single unit level. The problem with the above definition is that we have formally required a homogeneous population of neurons with identical connections which is hardly realistic. Real populations will always have a certain degree of heterogeneity both in their internal parameters and in their connectivity pattern. Nevertheless, rate as a population activity (of suitably defined pools of neurons) may be a useful coding principle in many areas of the brain. For inhomogeneous populations, the definition (2.6) may be replaced by a weighted average over the population. A related scheme has been used successfully for an interpretation of neuronal activity in primate motor cortex [84].

Since its clear by now that the rate coding ignores the temporal information, and the biological experiments contradict the assumptions of rate coding, we digress for now from this coding scheme to a different one in the next sub-section where by the temporal information from spikes could closely relate to neuro-biological findings. However, one of the most classic rate coding schemes, the formal neuron model that is most widely used for the analysis of learning and memory in artificial neural networks — the sigmoidal unit would be discussed in a later section.

2.2.2 Pulse Coding

Several coding schemes based on spike timing, are good candidates for understanding the relation between the neuro-biological experiments and reasoning and learning as understood by people in artificial intelligence.

2.2.2.1 Time-to-first-spike

Consider a neuron which abruptly receives a new input at time t_0. This could be the case when a neuron might be driven by an external stimulus which is suddenly switched on at time t_0. On the first glance this seems to be somewhat academic, but even in realistic situations abrupt changes in the input are quite common. When we look at a picture, our gaze jumps from one point to another. After each saccade, there is a new visual input at the photo receptors in the retina. Information about the time t_0 of a saccade would easily be available in the brain. We can then imagine a code where for each neuron the timing of the first spike to follow t_0 contains all information about the new stimulus. A neuron which fires shortly after t_0 could signal a strong stimulus, firing somewhat later would signal a weaker stimulation. In a pure version of this coding scheme, only the first spike of each neuron counts. All following spikes would be irrelevant. Alternatively, we can also assume that each neuron emits exactly one spike per saccade and is shut off by inhibitory input afterwards. It is clear that in such a scenario only the timing conveys information and not the number of spikes. A coding scheme based on the time-to-first-spike is certainly an idealization. Wolfgang Maass in [73] has rigorously formalized the scheme. In section 2.3.2 I would present a synopsis of this discussion.

In a slightly different context, coding by first spikes has also been discussed by S. Thorpe [109]. Thorpe argues that the brain does not have time to evaluate more than one spike from each neuron per processing step. Therefore the first spike should contain most of the relevant information. *Using information-theoretic measures on their experimental data, several groups have shown that most of the information about a new stimulus is indeed conveyed during the first 20 or 50 milliseconds after the onset of the neuronal response* [69, 126, 49, 48].

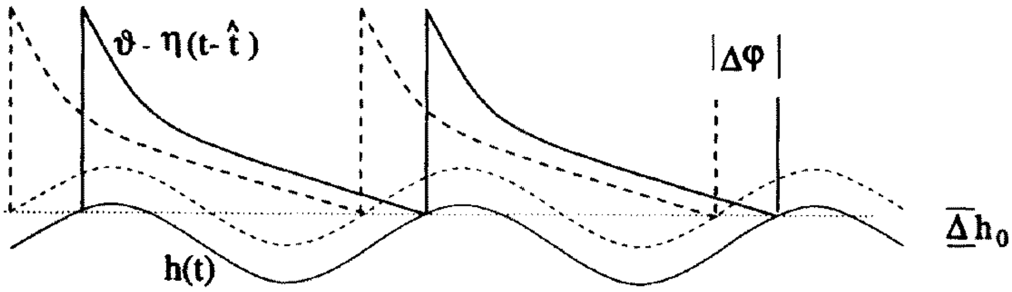


Figure 2.2: Phase coding. Firing occurs whenever the total input potential $h(t) = h_0 + h_1\,cos(2\pi t/T)$ hits the dynamic threshold $\theta - \eta(t- \hat{t})$ where $\hat{t}$ is the most recent firing time. In the presence of a periodic modulation $h_1 \neq 0$ a change Δh_0 in the level of (constant) stimulation results in a change $\Delta\,\varphi$ in the phase of firing. Taken from [73].

2.2.2.2 Phase

The idea of time-to-first-spike coding was extended to situations where the reference signal was not a single event, but a periodic signal. In the hippocampus, in the olfactory system, and also in other areas of the brain, oscillations of some global variable (for example the population activity) are quite common see Figure 2.2. These oscillations could serve as an internal reference signal. Neuronal spike trains could then encode information in the phase of a pulse with respect to the background oscillation. If the input does not change between one cycle and the next, then the same pattern of phases repeats periodically. The concept of phase coding has been studied by several different groups, not only in the area of model studies [44, 81, 71] but also experimentally [46]. There is also evidence that the phase of a spike during an oscillation in the hippo-campus of the rat conveys information on the spatial location of the animal which is not accounted for by the firing rate of the neuron alone [46].

2.2.2.3 Correlations and Synchrony

Using spikes from other neurons as the reference signal for a pulse code leads to the idea of coding by correlations and synchrony. As an example, consider a pair or a group of neurons, that could signify special events and convey information which is not contained in the firing rate of the neurons. One famous idea is that synchrony could mean 'belonging together' [68, 119]. Consider for example a complex scene consisting of several objects. It is represented in the brain by the activity of a large number of neurons. Neurons which represent the same object could be labelled by the fact that they fire synchronously [119, 92, 64]. Coding by synchrony has been studied extensively both experimentally [92, 64, 65, 127, 83, 51, 53] and in models [19, 91, 1, 85, 118, 96, 17, 18]. For a review of potential mechanism, refer [95].

More generally, *not only synchrony but any precise spatio-temporal* pulse pattern could be a meaningful event. For example, a spike pattern of three neurons, where neuron 1 fires at some arbitrary time t_1 followed by neuron at time $t_1 + \delta_{12}$ and by neuron 3 at $t_1 + \delta_{13}$, might represent a certain stimulus condition. The same three neurons firing with different relative delays might signify a different stimulus. The relevance of precise spatio-temporal spike patterns has been studied intensively by [60, 62, 61].

2.3 Neuron Models

Neural activity may be described at several levels of abstraction. On a microscopic level, there are a large number of ion channels, pores in the cell membrane which open and close depending on the voltage and the presence (or absence) of various chemical messenger molecules. Compartmental models [34], where each small segment of a neuron is described by a set of ionic equations, aim at a description of these processes.

On a higher level of abstraction, we do not worry about the spatial structure of a neuron nor about the exact ionic mechanisms. We consider the neuron as a homogeneous unit which generates spikes if the total excitation is sufficiently large. This is the level of the so called integrate-and-fire models [34].

Rate models neglect the pulse structure of the neuronal output, and are therefore higher up in the level of abstraction. On a yet coarser level would be models which describe the activity in and interaction between whole brain areas. Here with, we review some of the proposed spiking neuron models, which are studied extensively by neuro-biologists in [34].

2.3.1 Rate Model

In standard neural network theory, neural activity is described in terms of rates. The rate ν_i of neuron i is an analog variable which depends non-linearly upon the excitation u_i of the neuron,

$$\nu_i = g(u_i) \tag{2.7}$$

where $g(.)$ is usually taken as a sigmoidal function with $g(u) \to 0$ for $u \to -\infty$ and $g(u) \to 1$ for $u \to \infty$. The excitation is given by a linear sum over all input connections

$$u_i = \sum_{j \in \Gamma_i} w_{ij} \nu_j \tag{2.8}$$

where ν_j is the output rate of a presynaptic neuron j. The sum runs over all neurons which send signals to neuron i. The parameter w_{ij} which is called synaptic efficacy, is the weight attributed to the connection from j to i. The above two equations could be combined into one

$$\nu_i = g\left(\textstyle\sum_{j \in \Gamma_i} w_{ij}\, \nu_j \right) \tag{2.9}$$

The above equation is the starting point of standard neural network theory [43, 97, 41, 63]. However the above equation is a *static* equation. It applies to situations where a stationary input (a set of firing rates ν_j) is mapped to a stationary output (the rate ν_i). Is it possible to make the equation time-dependent? A straightforward way to introduce dynamics into the rate equation is to replace (2.9) by a differential equation as described in [14]

$$\tau \frac{d\nu_i}{dt} = -\nu_i + g\left(\textstyle\sum_{j \in \Gamma_i} w_{ij}\, \nu_j \right) \tag{2.10}$$

with some time constant τ. For stationary input and output the left hand side of the above equation vanishes and it reduces to equation (2.9). In other words the fix-point solutions of equation (2.10) are given by equation (2.9). *Equation* (2.10) *provides a convenient way to introduce some time dependence* *in the rate model equation* (2.9)*, but can it be considered a realistic description of neuronal activity? As we have discussed previously in Section 2.2.1, an analog variable defined by a spike count measure requires a long temporal averaging window. It can therefore be used only if the input and the output change on a slow time scale. Considering the fact that, for example, the visual input changes due to saccades every 200-500 ms, a slowly changing input can not always be assumed.* It has therefore been argued that the rate equation (2.10) refers to a *population average* rather than to a *temporal average.*

2.3.2 Spiking Neuron Model

The state of neuron i is described by a state variable u_i. The neuron is said to fire, if u_i reaches a threshold θ. The moment of threshold crossing defines the firing time t_i^f. The set of all firing times of neuron i is denoted by

$$\mathcal{F}_i = \{{t_i}^f; 1 \leq f \leq n\} = \{t \mid u_i(t) = \theta\} \tag{2.11}$$

For the most recent spike ${t_i}^f \leq t$ of neuron i we write either ${t_i}^n$ or shorter $\hat{\mathrm{t}}$. Two different processes contribute to the value of the state variable u_i.

First, immediately after firing an output spike at t_i^f, the variable u_i is lowered or 'reset'. Mathematically, this is done by adding a negative contribution $\eta_i(t - {t_i}^f)$ to the state variable u_i. The kernel $\eta_i(s)$ vanishes for $s \leq 0$ and decays to zero for $s \to \infty$.

Second, the model neuron may receive input from presynaptic neurons $j \in \Gamma_i$ where

$$\Gamma_i = \{j \mid j \textit{ presynaptic to } i\} \tag{2.12}$$

A presynaptic spike at time ${t_i}^f$ increases (or decreases) the state u_i of neuron i for $t > t_j^f$ by an amount $w_{ij}\epsilon_{ij}(t - t_j^f)$. The weight w_{ij} is a factor which accounts for the strength of the connection. The effect of a presynaptic spike may be positive (excitatory) or negative (inhibitory). Because of causality, the kernel $\epsilon_{ij}(s)$ must vanish for $s \leq 0$. A transmission delay may be included in the definition of ϵ_{ij}. The state $u_i(t)$ of model neuron i at time t is given by the linear superposition of all contributions

$$u_i(t) = \sum_{{t_i}^f \in \mathcal{F}_i} \eta_i(t - t_i^f) + \sum_{j \in \Gamma_i} \sum_{t_j^f \in \mathcal{F}_j} w_{ij}\epsilon_{ij}(t - t_j^f) \tag{2.13}$$

An interpretation of the terms on the righthand side of the above equation reveals two key contributions. The η_i contributions describe the response of neuron i to its own spikes. The ϵ_{ij} kernels model the neurons response to presynaptic spikes. Equations (2.11) – (2.13) describe what is known as the Spike Response Model (SRM). *In a biological context, the state variable u_i may be interpreted as the electrical membrane potential.* The kernels ϵ_{ij} are the postsynaptic potentials and η_i accounts for neuronal refractoriness.

As a specific example of these functions consider

$$\eta_i(s) = -\theta\, exp(-\frac{s}{\tau})\, \mathcal{H}(s) \tag{2.14}$$

where τ is a time constant and $\mathcal{H}(s)$ is the Heaviside step function which vanishes for $s \leq 0$ and takes a value of 1 for $s > 0$. At the moment of firing $u_i(t) = \theta$. The effect of the above equation is that after each firing the state variable u_i is reset to zero. If the factor θ on the right hand side of the above equation is replaced by a parameter $\eta_0 \neq \theta$, then the state variable would be reset to a value $\theta - \eta_0 \neq 0$. For a further discussion of the η-kernel, refer to [34].

The kernel ϵ_{ij} describes the response to presynaptic spikes. For excitatory synapses ϵ_{ij} is positive and is called the excitatory postsynaptic potential (EPSP). For inhibitory synapses, the kernel takes negative values and is called the inhibitory postsynaptic potential (IPSP) (see Figure 2.3). As an example of this kernel

$$\epsilon_{ij}(s) = \left[\ exp(-\tfrac{s-\Delta^{ax}}{\tau_m}) - exp(-\tfrac{s-\Delta^{ax}}{\tau_s})\ \right] \mathcal{H}(s - \Delta^{ax}) \tag{2.15}$$

where τ_s, τ_m are time constants and Δ^{ax} is the axonal transmission delay. The amplitude of the response is scaled via the factor w_{ij} in equation (2.13). For inhibitory synapses, the kernel ϵ_{ij} would have a negative sign in front of the expression on the right-hand side. Alternatively, we can put the sign in the synaptic efficacy and use $w_{ij} > 0$ and $w_{ij} < 0$ for inhibitory synapses.

Equation (2.11) - (2.13) give us a fairly general framework for describing neuron models. It was shown in [34] that the spike response model (2.11) - (2.13) is equivalent to the *integrate–and–fire–model*. Additionally, with a different choice of kernel, the spike response model also approximates the *Hodgkin-Huxley* equations with time dependent input as in [73].

The literature in neuro-biology is abound with other neuron models, however its not possible to discuss them all here. Also because they are not

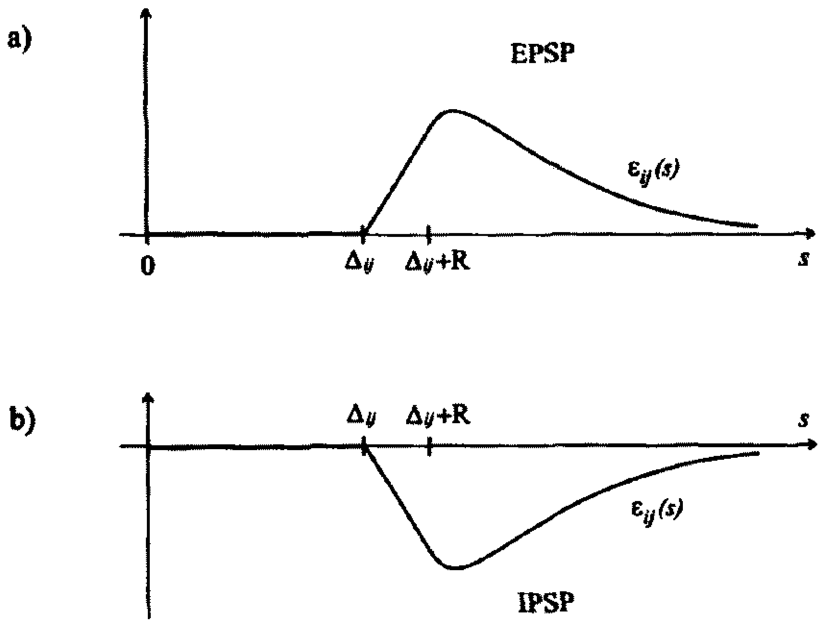


Figure 2.3: (a) Typical shape of an excitatory postsynaptic potential (EPSP). (b) Typical shape of an inhibitory postsynaptic potential. Taken from [73]

directly useful as a foundation for our integrated models of logical reasoning and neuro - biology (to be presented later in the next chapter) we don't discuss them here, for a detailed description of these models refer to [34].

2.4 Spiking Neurons and Other Neuron Models

In this section we relate the spiking neuron model to some of the well known more classical models in neural networks and analyze the computational power of the spiking neuron model. The simplest traditional neural network model is a *McCulloch–Pitts* neuron also referred to as threshold gate.

A McCulloch–Pitts neuron i with real valued weights α_{ij} and threshold θ receives as input n binary or real valued numbers $x_1, \ldots, x_n$. Its output has the value

$$\begin{cases} 1 & \text{if } \sum_{j=1}^{n} \alpha_{ij} \cdot x_j \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

In multilayer networks one usually considers a variation of the threshold gate to which we refer as a *sigmoidal gate*. The output of a sigmoidal gate is defined with the help of some nondecreasing continuous *activation function* $g : \mathbf{R} \to \mathbf{R}$ with bounded range as

$$g(\sum_{j=1}^{n} \alpha_{ij} \cdot x_j - \theta) \tag{2.17}$$

In the subsequent paragraphs, a comparison of the spiking neuron model and the sigmoidal and McCulloch–Pitts neuron is presented.

As an immediate observation from the discussion so far, its easy to see that a spiking neuron i can in principle simulate any given threshold gate equation (2.16) with positive threshold θ for binary input. For that one can assume that the response functions $\epsilon_{ij}(x)$ are all identical except for their sign (which we choose to be positive if $\alpha_{ij} > 0$ and negative if $\alpha_{ij} \leq 0$), and that all presynaptic neurons j which fire, fire at the same time $t_j = T_{input}$. In this case the spiking neuron i fires if and only if

$$\sum_{j\, fires\, at\, time\, T_{input}} w_{ij} \cdot \epsilon_{ij} \geq \theta \tag{2.18}$$

where ϵ_{ij} is the extremal value of $\epsilon_{ij}(s)$ (i.e., $e_{ij} = max_s\, e_{ij}(s)$ if $e_{ij}(s)$ represents an EPSP, $e_{ij} = min_s\, e_{ij}(s)$ if $e_{ij}(s)$ represents an IPSP), $w_{ij} \geq 0$ is the synaptic weight, and $\theta > 0$ is the firing threshold of neuron i, see Figure 2.4.

For $w_{ij} = \frac{\alpha_{ij}}{e_{ij}}$ the spiking neuron i can simulate any given threshold gate defined by equation (2.16), if the input bits $x_1, \ldots, x_n$ are encoded by the firing or non-firing of presynaptic neurons $j = 1, \ldots, n$ at a common time T_{input}, and if the output bit of the threshold gate is encoded by the firing or non-firing of the spiking neuron i during the relevant time window.

In the case of multi-layer network of threshold gates (i.e., a threshold circuit) a network of spiking neurons is substantially more difficult to simulate.

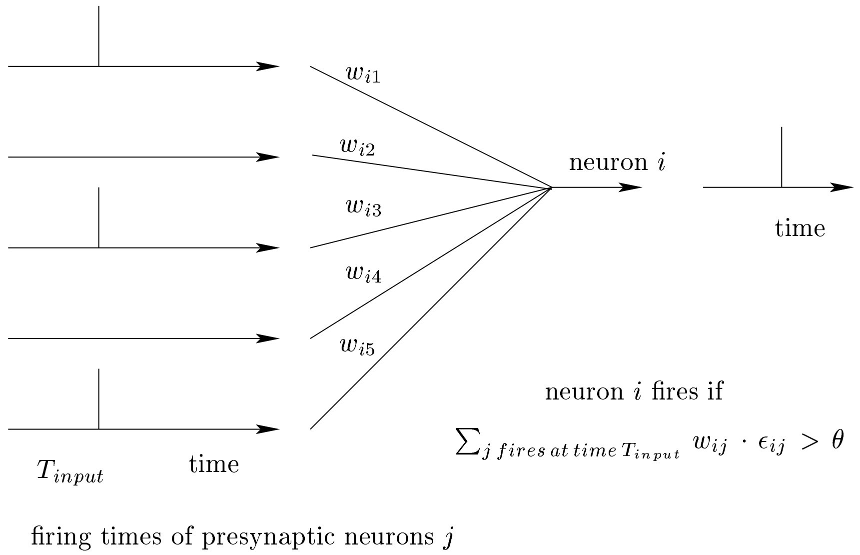


Figure 2.4: Typical input for a biological spiking neuron i, where its output cannot be easily described in terms of conventional computational units. Taken from [73]

The exact firing time of the previously discussed spiking neuron i depends on its concrete input $x_1, \ldots, x_n$. If $\sum_{j\, fires\, at\, time\, T_{input}} w_{ij} \cdot e_{ij} - \theta$ has a value well above 0, then the state variable $u_i(t) = \sum_{j\, fires\, at\, time\, T_{input}} w_{ij} \cdot e_{ij}(t - T_{input}$ will cross the firing threshold θ earlier, yielding an earlier firing time of neuron i, compared with an input where $\sum_{j\, fires\, at\, time\, T_{input}} w_{ij} \cdot e_{ij} - \theta$ is positive but close to 0. Therefore, if one employs several spiking neurons to simulate the threshold gates on the first layer 1 of a threshold circuit, those neurons on the first layer which do fire (corresponding to threshold gates with output 1) will in general fire at slightly different time points. This will occur even if all input neurons j of the network fired at the same time T_{input} . Therefore the timing of such straightforward simulation of a multilayer threshold circuit is unstable: even if all firings in one layer occur synchronously, this synchronicity will in general get lost at the next layer. Similar problems arise in a simulation of other types of multilayer boolean circuits by networks of spiking neurons. Consequently one needs a separate synchronization mechanism in order to simulate a multilayer boolean circuit – or any other common model for digital computation – by a network of spiking neurons with bits 0, 1 encoded by firing and non-firing. One can give a mathematically rigorous proof that such synchronization mechanism can in principle be provided by using some auxiliary spiking neurons [71]. This construction exploits the simple fact that the double-negation $\neg\neg b$ of a bit b has the same value as b. Therefore instead of making a spiking neuron i fire in the direct way described by equation (2.18), one can make sure that if $\sum_j \alpha_{ij}, \cdot x_j \geq \theta$, then the spiking neuron i is not prevented from firing by auxiliary inhibitory neurons. These auxiliary inhibitory neurons are connected directly to the input neurons whose firing/non-firing encodes the input bits $x_1, \ldots, x_n$, whereas the driving force for the firing of neuron i comes from input-independent excitatory neurons. With this method one can simulate any given boolean circuit, and in the absence of noise even any Turing machine, by a finite network of spiking neurons [71], please refer to [114] for earlier work in this direction.

It may be less advantageous for a network of spiking neurons to employ a forced synchronization. The small temporal differences in the firing times of the neurons i that simulate the first layer of a threshold circuit according to equation (2.18) contain valuable additional information that is destroyed by a forced synchronization: these temporal differences contain information about how much larger the weighted sum $\sum_j \alpha_{ij} \cdot x_j$ is compared with θ. Thus it appears to be advantageous for a network of spiking neurons to employ instead of a synchronized digital mode an asynchronous or loosely synchronized analog mode where subtle differences in firing times convey additional analog information. In the next section we discuss computational operations which spiking neurons can execute in this quite different computational mode, where analog values are encoded in temporal patterns of firing times.

2.5 Computing with Temporal Patterns

In the preceding section a mathematical model for spiking neurons was introduced. Spiking neurons differ in essential aspect from the familiar computational units of common neural network models, such as McCullochPitts neurons or sigmoidal gates. Therefore the question arises how one can compute with spiking neurons, or with related computational units in electronic hardware whose input and output consists of trains of pulses.

Let I be a set of neurons, and assume that one has specified for each neuron $i \in I$ a set Γ_i of immediate predecessors ("presynaptic neurons") in the network. The firing times ${t_i}^f \in \mathcal{F}_i$ for all neurons $i \in I$ are defined recursively by simultaneous recursion along the time axis. According to equation (2.13) the neuron i fires whenever the state variable $u_i(t)$ reaches the firing threshold θ of neuron i. If one focuses on the case where the neuron i has not fired for a while, then the first summand with refractory terms $\eta_i(t - t_i{}^f)$ can be ignored. In order to complete the definition of a network of spiking neurons as a formal computational model one has to specify its network input and output. We assume that subsets of neurons $I_{input} \subset I$ and $I_{output} \subset I$ have been fixed, and that the firing times $\mathcal{F}_i$ for the neurons $i \in I_{input}$ constitute the network input. Thus for a while we assume that these firing times are determined through some external mechanism, rather than computed according to the previously described rules. The firing times $\mathcal{F}_i$ of the neurons $i \in I_{output}$ constitute the network output. These firing times are computed in the previously described way (like for all neurons $i \in I - I_{input}$) with the help of state variable $u_i(t)$ given by equation (2.13).

Thus from a mathematical point of view, a network of spiking neurons computes a function which maps a vector of several time series $\langle \mathcal{F}_i \rangle_{i \in I_{input}}$ on a vector of several other time series $\langle \mathcal{F}_i \rangle_{i \in I_{output}}$. It has often been argued that the fact that network input and output are vectors of time series, rather than vectors of numbers as in conventional neural network models, is essential for understanding computation in biological neural systems. After all, a biological organism has to be able to respond very fast in an online fashion to a constantly changing environment. Since artificial versions of pulsed neural nets are in a similar manner well-suited for computing in the time series domain, this application area is of particular interest for the design of artificial pulsed neural nets.

It has been shown in [73] that in an asynchronous mode, with analog values encoded by a temporal pattern of firing times, a spiking neuron has in principle not only more computational power than a McCullochPitts neuron, but also more computational power than a sigmoidal gate.

2.5.1 Coincidence Detection

One new feature of a spiking neuron — which has no analog in the computational units of traditional neural network models is that it can act as coincidence detector for incoming pulses [59]. Hence if the arrival times of the incoming pulses encode numbers, a spiking neuron can detect whether some of these numbers have almost equal value. However, from the view of traditional neural network models, this computational operation is a rather expensive one.

Consider a spiking neuron i connected to n immediate predecessors, given by

$$\Gamma_i = \{1, \ldots, n\} \tag{2.19}$$

assume that the connections to neuron i all have the same transmission delay Δ_{ij} and that $w_{ij} = 1$ for all $j \in \Gamma_i$. Furthermore also assume that the response functions $\epsilon_{ij}(s)$ are defined as in equation (2.15) by

$$\epsilon_{ij}(s) = \frac{1}{1-(\tau_s - \tau_m)} \left[\ exp(-\tfrac{s-\Delta^{ax}}{\tau_m}) - exp(-\tfrac{s-\Delta^{ax}}{\tau_s})\ \right] \mathcal{H}(s-\Delta^{ax}) \tag{2.20}$$

with time constants $0 < \tau_s < \tau_m$. The plot of an EPSP of such shape is shown in Figure 2.3.

It consists of an almost linearly rising phase for small s, exponential decay for large s, and a smooth transition between both phases when it reaches its maximal value in between. For every given values of the time constants τ_s, τ_m with $\tau_s < \tau_m$ one can find values $0 < c_1 < c_2$ and θ so that $u_i(t) < \theta$ for any input consisting of an arbitrary number of EPSP's with distance $\geq c_2$, whereas $u_i(t)$ reaches a value $> \theta$ for two EPSP's in distance $\leq c_1$. Then the spiking neuron i does not fire if the neurons $j \in \Gamma_i$ fire (each at most once) in temporal distance $\geq c_2$ (see Figure 2.5a)

but it fires whenever two presynaptic neurons $j \in \Gamma_i$ fire in temporal distance $\leq c_1$, (see Figure 2.5b). Consequently, if for example one encodes n real numbers $x_1, \ldots, x_n$ through the firing times of the n neurons in Γ_i, and decodes the output of neuron i as "1" if it fires and "0" if it does not fire, the neuron i computes the following function $ED_n : \mathbf{R}^n \rightarrow \{0,1\}$:

$$ED_n(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if there are } j \neq j' \text{ so that } \mid x_j - x_{j'} \mid \leq c_1 \\ 0 & \text{if } \mid x_j - x_{j'} \mid \geq c_2 \text{ for all } j \neq j' \end{cases} \tag{2.21}$$

This function $ED_n(x_1, \ldots, x_n)$ (where ED stands for "element distinctness") is in-fact a partial function, which may output arbitrary values in case that $c_1 < min\{\mid x_j - x_{j'} \mid : j \neq j'\, and\, j, j' \in \Gamma_i\} < c_2$. Therefore hair-trigger situations can be avoided, and a single spiking neuron can compute this function ED_n even if there is a small amount of noise on its state variable $u_i(t)$.

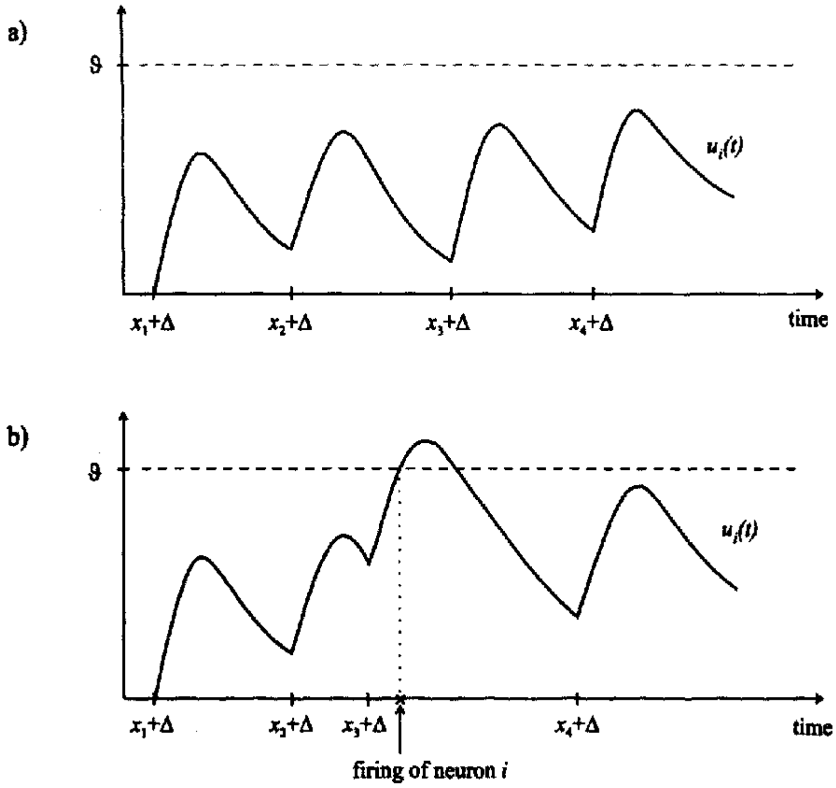


Figure 2.5: (a) Typical time course of the state variable $u_i(t)$ of $ED_4(x_1, x_2, x_3, x_4) = 0$. (b) Time course of $u_i(t)$ in the case where $ED_4(x_1, x_2, x_3, x_4) = 1$ because $|x_3 - x_2| \le c_1$. Taken from [73]

Interestingly the following result shows that the same partial function ED_n requires a substantial number of neurons if computed by neural networks consisting of McCulloch-Pitts neurons (threshold gates) or sigmoidal gates. These lower bounds hold for arbitrary feed-forward architectures of the neural net, and any values of the weights and thresholds of the neurons. The inputs $x_1, \ldots, x_n$ are given to these neural nets in the usual manner as analog input variables.

Theorem 1 Any layered threshold circuit that computes ED_n needs to have at least $log(n!) \geq \frac{n}{2} \cdot log\, n$ threshold gates on its first layer.
For the proof see [72].
Theorem 2 Any feed-forward neural net consisting of arbitrary sigmoidal gates needs to have at least $\frac{n-4}{2}$ gates in order to compute ED_n.

For proof see [72].

2.5.2 RBF - Units in the Temporal Domain

We have demonstrated in the preceding subsection that for some computational tasks a single spiking neuron has more computational power than a fairly large neural network of the conventional type. We will show in this subsection that the preceding construction of a spiking neuron that detects coincidences among incoming pulses can be expanded to yield detectors for more complex temporal patterns.

Instead of a common delay Δ between presynaptic neurons $j \in \Gamma_i$ and neuron i (which appears in our formal model as the length of the initial flat part of the response function $\epsilon_{ij}(x)$ one can employ for different j different delays Δ_{ij} between neurons j and i. These delays Δ_{ij} represent a new set of parameters that have no counterpart in traditional neural network models. There exists evidence that in some biological neural systems these delays Δ_{ij} can be tuned by "learning algorithms". In addition one can tune the firing threshold θ and/or the weights w_{ij} of a spiking neuron to increase its ability to detect specific temporal patterns in the input. In the extreme case one can raise the firing threshold θ so high that all pulses from presynaptic neurons have to arrive nearly simultaneously at the soma of i to make it fire. In this case the spiking neuron can act in the temporal domain like an RBF–unit (i.e., radial basis function unit) in traditional neural network models: it will fire only if all presynaptic neurons $j \in \Gamma_i$ fire at times t_j so that for some constant T_{input} one has $t_j \approx T_{input} - \Delta_{ij}$ for all $j \in \Gamma_i$, where the vector $(\Delta_{ij})_{j\in\Gamma_i}$ of transmission delays plays now the role of the center of an RBF-unit. This possibility of using spiking neurons as RBF-like computational units in the temporal domain was first observed by Hopfield [Hopfield, 1995]. In the same article Hopfield demonstrates an advantageous consequence of employing a logarithmic encoding $x_j = logy_j$ of external sensory input variables y_j through firing times $t_j = T_{input} - x_j$. Since spiking neurons have the ability to detect temporal patterns irrespective of a common additive constant in their arrival times, they can with the help of logarithmic encoding ignore constant factors λ in sensory input variables $\langle \lambda \cdot y_j \rangle_{j \in \Gamma_i}$. It has been argued that this useful mechanism may be related to the amazing ability of biological organisms to classify patterns over a very large scale of intensities, such as for example visual patterns under drastically different lighting conditions.

2.5.3 Computing a Weighted Sum in Temporal Coding

A characteristic feature of the previously discussed computation of the function ED_n and the simulation of an RBF-unit is the asymmetry between coding schemes used for input and output. Whereas the input consisted of a vector of analog numbers, encoded through temporal delays, the output of the spiking neuron was just binary, encoded through firing or non-firing of that neuron. Obviously for multilayer or recurrent computations with spiking neurons it is desirable to have mechanisms that enable a layer of spiking neurons to output a vector of analog numbers encoded in the same way as the input. For that purpose one needs mechanisms for shifting the firing time of a spiking neuron i in dependence of the firing times t_j of presynaptic neurons, in a manner that can be controlled through the internal parameters w_{ij} and Δ_{ij}. As an example for that we will now describe a simple mechanism for computing for arbitrary parameters $\alpha_{ij} \in \mathbf{R}$ and inputs $x_j \in [0, 1]$ the weighted sum $\sum_j \alpha_{ij} \cdot x_j$ through the firing time of neuron i. We assume that each response function is described by the property that $\epsilon_{ij}(s)$ has a value 0 for $s \leq \Delta_{ij}$ and then rises approximately linearly (in the case of EPSP) with slope $\lambda_{ij} \in \mathbf{R}$ for an interval of length at least $R > 0$ (see [73] for figures and more detailed explanation). Assume that the presynaptic neurons $j \in \Gamma_i$, fire at times $t_j = T_{input} - x_j$. If the state variable $u_i(t) = \sum_{j \in \Gamma_i} w_{ij} \cdot \epsilon_{ij}(t - t_j)$ of neuron i reaches the threshold θ at a time t_i, when the response functions $\epsilon_{ij}(t - t_j)$ are all in the their initial linear phase of length $\geq R$ then t_i is determined by the equation

$$\sum_{j \in \Gamma_i} w_{ij} \cdot \epsilon_{ij}(t_i - t_j) = \sum_{j \in \Gamma_i} w_{ij} \cdot \lambda_{ij} \cdot (t_i - t_j - \Delta_{ij}) = \theta \tag{2.22}$$

Obviously the above equation implies that

$$t_i = \frac{\theta}{\sum_{j \in \Gamma_i} w_{ij} \cdot \lambda_{ij}} + \frac{\sum_{j \in \Gamma_i} w_{ij} \cdot \lambda_{ij}(t_j + \Delta_{ij})}{\sum_{j \in \Gamma_i} w_{ij} \cdot \lambda_{ij}} \tag{2.23}$$

Then by writing λ for $\sum_{j\in\Gamma_i} w_{ij} \cdot \lambda_{ij}$ and expressing t_j as $T_{input} - x_j$ we get

$$t_i = \frac{\theta}{\lambda} + \sum_{j\in\Gamma_i} \frac{w_{ij} \cdot \lambda_{ij}}{\lambda} (T_{input} - x_j + \Delta_{ij}) \tag{2.24}$$

or equivalently

$$t_i = T_{ouput} - \sum_{j\in\Gamma_i} \alpha_{ij} \cdot x_j \tag{2.25}$$

for some input independent constant $T_{output} := \frac{\theta}{\lambda} + \sum_{j\in\Gamma_i} \frac{w_{ij} \cdot \lambda_{ij}}{\lambda} (T_{input} + \Delta_{ij})$, and formal "weights" $\alpha_{ij} := \frac{w_{ij} \cdot \lambda_{ij}}{\lambda}$. These "weights" α_{ij} are automatically normalized: by definition of λ they satisfy $\sum_{j\in\Gamma_i} \alpha_{ij} = 1$. Such automatic weight normalization may be desirable in some situations. One can circumvent it by employing an auxiliary input neuron. In this way one can compute an arbitrary given weighted sum $\sum_{j\in\Gamma_i} \alpha_{ij} \cdot x_j$ in temporal coding by a spiking neuron. Note that in this construction the analog output $\sum_{j\in\Gamma_i} \alpha_{ij} \cdot x_j$ is encoded in exactly the same way as the analog inputs x_j.

2.5.4 Computing with Firing Rates

There exists ample biological evidence that information about a stimulus is in many biological neural systems encoded in the firing rates of neurons. However recent empirical results from neurophysiology have raised doubts whether the firing rate of a biological neuron i does in fact depend on the firing rates x_j of presynaptic neurons $j \in \Gamma_i$ in a way that can be described by an expression of the form $g(\sum_{j\in\Gamma_i} w_{ij}\, x_j)$. Results of [28] and others about the dynamic behavior of biological synapses show that for some neural systems above a "limiting frequency" of about 10 Hz the amplitudes of postsynaptic potentials are inversely proportional to the firing rate x_j of the presynaptic neuron $j \in \Gamma_i$. These results suggest that instead of a fixed parameter w_{ij} one has to model the "strength" of a biological synapse for rate coding by a quantity $w_{ij}(x_j)$ that depends on the firing rate x_j of the presynaptic neuron, and that this quantity $w_{ij}(x_j)$ is proportional to $\frac{1}{x_j}$. But then the weighted sum $\sum_{j\in\Gamma_i} w_{ij}(x_j) \cdot x_j$, which models the average membrane potential at the soma of a spiking neuron i, does no longer depend on the firing rates x_j of those presynaptic neurons j that fire above the limiting frequency.

A very interesting dual interpretation of the computational role of spiking neurons in the context of rate coding is due to [116]. They have pointed out that *if a spiking neuron is working in the coincidence detector mode (i.e., with short integration time relative to the number and firing rates of presynaptic neurons), and the input spike trains are generated by stochastically independent Poisson processes, then its output firing rate is in a certain parameter range proportional to the product of the firing rates of the presynaptic neurons.* If for example the spike train from 3 presynaptic neurons j are generated by stochastically independent Poisson processes, then the probability that all of these three presynaptic neurons j fire within the same short time window A is proportional to the product of the firing probability of the individual neurons j. Hence if one assumes that a spiking neuron i fires if and only if these 3 presynaptic neurons j fire within a short time window A, its firing rate encodes (approximately) the product of the firing rates of the presynaptic neurons j.

2.5.5 Computing with Temporal Correlations

This section discusses computations that employ a quite different type of "temporal coding". Communication by spike trains offers a direct way to encode transient relations between different neurons: through coincidences (or near coincidences) in their firing times. Hence computations with spiking neurons may in principle also involve complex operations on relations between computational objects, *a computational mode which has no parallel in traditional neural network models or any other common computational model*. This type of temporal coding need not necessarily take place on the microscopic level of coding by single spikes, but can also take place on a macroscopic level of statistical correlations between firing times of different neurons. Milner had conjectured already in 1974 that visual input might be encoded in the visual cortex in a way where "cells fired by the same figure fire together but not in synchrony with cells fired by other figures"([68]). This conjecture has been supported more recently by experimental data from several labs (see for example [92, 64, 54, 26]). A variety of models have been proposed in order to shed light on the possible organization of computing with firing correlations in networks of spiking neurons. We have already shown in the preceding sections that spiking neurons are well-suited for detecting firing correlations among preceding neurons. They also can induce firing correlations in other neurons k by sending the same output spike train to several other neurons k. But the question remains what exactly can be computed with firing correlations in a network of spiking neurons.

In [91] a computational model was introduced whose computational units are modifications of integrate-and-fire neurons that receive two types of input: feeding input and linking input. Both types of inputs are processed in this model by leaky integrators with different time constants, and are then multiplied to produce the potential $u_i(t)$ of an integrate-and-fire neuron i. In networks of such computational units the feeding input typically is provided by feed-forward connections, starting from the stimulus, whereas the linking input comes through feedback connections from higher layers of the network. These higher layers may represent information stored in an associative memory like in a Hopfield net. Computer simulations have shown that this model is quite successful in reproducing firing patterns in response to specific stimuli that match quite well firing patterns that have been experimentally observed in the visual cortex of cat and monkey. So far no theoretical results have been derived for this model.

Other models aim directly at simulating effects of computing with firing correlations on an even more abstract conceptual level, see for example [119, 111]. In [111] a formal calculus is developed for exploiting the possibility to compute with relations encoded by temporal coincidences.

2.5.6 The Idea of Temporal Binding

Synchronized activity in populations of neurons in cortical and sub-cortical regions, and in species as diverse as turtles [47], pigeons [107], rats [2], cats [65], and monkeys [106, 54] is often found to coincide with oscillations in the gamma frequency range (20 - 70 Hz). This synchronization can occur with zero phase delay across multiple visual association areas, between visual and motor areas, and between cortical and sub-cortical regions. The presence of the zero phase shift synchronized firing has been shown to exist between particular subsets of neurons within an area and to occur in relation to specific behavioral events. In the cat visual system synchronized activity was found in cells in separate cerebral hemispheres [51]. The evidence for synchronized activity and the significance of this phenomenon has been recently thoroughly reviewed [128, 52].

It has been suggested that this synchrony provides a means to bind together in time the features that represent a particular stimulus [68, 119]. There is substantial evidence to suggest that the individual features of a perceived object are encoded in distributed brain regions and fire collectively as a cell assembly [80]. The central idea is that there is some temporal process during which information about an object or group of objects is processed. In order to keep the attributes of these objects from interfering, they are separated in time, as illustrated in Figure 2.6. In this figure there are two

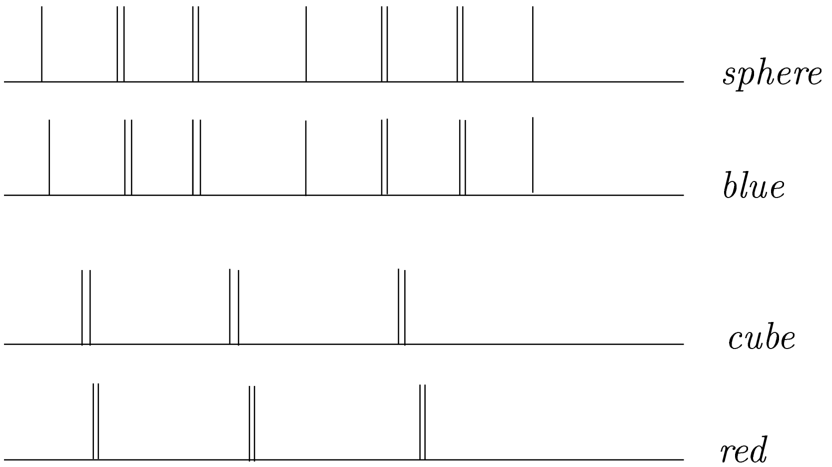


Figure 2.6: Temporal binding of features in cell assemblies. In this example both a blue sphere and a red cube are present in a scene. If sets of neurons code for the colors red and blue, and for the shapes sphere and cube, then the correct association of color and shape might be represented in the brain by synchronizing the activity of the neurons within each of the two cell assemblies. In this way neurons coding for blue and sphere are active at a different time from those coding for red and cube, and the two objects are processed simultaneously.

cell assemblies, one that codes for the features of a blue sphere, and one that codes for the feature of a red cube. Within a cell assembly the neurons fire in a synchronized pattern with zero phase shift. There is a time shift between the two assemblies so that each remains coherent, and the scene is not confused with one containing a red sphere and a blue cube. With firing rate coding it is not possible to construct this type of neuronal system in which multiple simultaneous use is made of a set of feature encoding neurons.

For over forty years it has been known that a firing rate code is not the most efficient way for a neuron to transmit information with action potentials [15]. Some of the most convincing evidence that precise timing of spikes provides additional information, not present in firing rate alone has come from experiments carried out by van Steveninch, Bialek and coworkers. The timing sequence of action potentials in the H1 neuron of the fly have been shown to contain the information needed to reconstruct the stimulus pattern. This reconstruction implies that there is a precision on the order of a millisecond in the firing of each spike. Furthermore they have shown that the H1 neuron only produces one or two spikes in the 30 milliseconds that a fly samples before it can make a response, suggesting that the information encoded in the inter-spike intervals must be used by the down stream neurons.

2.6 Continuous Functions and Spiking Neurons

We will show in this subsection that on the basis of the computational mechanism described in the preceding subsection one can build networks of spiking neurons that can approximate arbitrary given bounded continuous functions in the temporal domain. We first observe that one can expand the previously described mechanism for computing a weighted sum $\sum_{j \in \Gamma_i} \alpha_{ij} \cdot x_j$ in the temporal domain to yield for temporal coding also a simulation of an arbitrary given sigmoidal neuron with the piecewise linear gain function

$$sat(x) = \begin{cases} x, & \text{if } 0 \leq x \leq 1 \\ 0, & \text{if } x \leq 0 \\ 1, & \text{if } x \geq 1 \end{cases} \tag{2.26}$$

In this case we want that neuron i responds to firing of its presynaptic neurons at times $t_j = T_{input} - x_j$ by firing at time

$$t_i = T_{output} - sat(\sum_{j \in \Gamma_i} \alpha_{ij} \cdot x_j) \tag{2.27}$$

For that purpose one just needs auxiliary mechanisms that support an approximation of the given sigmoidal neuron in the saturated regions of its gain function sat, i.e., for $x \leq 0$ and $x \geq 1$. Translated into the temporal domain this requires that the spiking neuron i does not fire before some fixed time T (simulating $sat(x) = 1$ for $x \geq 1$) and by the latest at some fixed time $T_{output} > T$ (simulating $sat(x) = 0$ for $x \leq 0$). This can be easily achieved with the help of auxiliary spiking neurons. Computer simulations suggest that in a practical situation such auxiliary neurons may not even be necessary [74].

According to the preceding construction one can simulate any sigmoidal neuron with the piecewise linear gain function *sat* by spiking neurons with analog inputs and outputs encoded by temporal delays of spikes. Since inputs and outputs employ the same coding scheme, the outputs from a first layer of spiking neurons (that simulate a first layer of sigmoidal gates) can be used as inputs for another layer of spiking neurons, simulating another layer of sigmoidal gates. Hence on the basis of the assumption that the initial segments of response functions $\epsilon_{ij}(s)$ are linear one can show with a rigorous mathematical proof [72]:

Theorem 3 Any feed-forward or recurrent analog neural net (for example any multilayer perceptron), consisting of s sigmoidal neurons that employ the gain function *sat*, can be simulated arbitrarily closely by a network of $s + c$ spiking neurons (where c is a small constant) with analog inputs and outputs encoded by temporal delays of spikes. This holds even if the spiking neurons are subject to noise.

Theorem 2 and 3 together exhibit an interesting asymmetry regarding the computational power of standard sigmoidal neural nets (multilayer perceptron) and networks of spiking neurons: Whereas any sigmoidal neural net can be simulated by an insignificantly larger network of spiking neurons (with temporal coding), certain networks of spiking neurons can only be simulated by substantially larger sigmoidal neural nets.

It is well known that feed-forward sigmoidal neural nets with gain function *sat* can approximate any given continuous function $F : [0, 1]^n \to [0, 1]^m$ with any desired degree of precision. Hence Theorem 3 implies:

Corollary 4 Any given continuous function $F : [0, 1]^n \to [0, 1]^m$ can be approximated arbitrarily closely by a network of spiking neurons with inputs and outputs encoded by temporal delays.

Remarks:

1. The construction that yields the proof of Theorem 3 shows that a network of spiking neurons can change the function: $F : [0, 1]^n \to [0, 1]^m$ that it computes in the same way as a traditional neural net: by changing the synaptic "weights" w_{ij} that scale the slopes of the initial seg-

ments of postsynaptic pulses. The delays Δ_{ij} between neurons need not be changed for that purpose (but they could be used to modulate the effective "weights" of the simulated sigmoidal neural net by additive constants). From that point of view this construction is complementary to the simulation of RBF-units by spiking neurons described in Subsection 2.5.2 there the "program" of the encoded function was encoded exclusively in the delays Δ_{ij}.

2. It turns out that the network of spiking neurons constructed for the proof of Theorem 3 computes approximately the same function in rate coding and in temporal coding.

2.7 Learning with Spiking Neurons

The neural models which have been described in neuro-biology and some of them which have been reviewed in Section 2.3 all assume that synapses are "static", i.e., that they change their "weight" only on the slow time scale of learning. It will be clear from the discussion in the subsequent sections that this assumption is not justified for biological neural systems.

(i) The inherent temporal dynamics of synaptic weights may not just be a curse, but also a blessing as dynamic synapses provide novel computational units for neural computation in the time series domain, i.e., in an environment where the inputs (and possibly also the outputs) of the networks are functions of time.

(ii) One has to revise the foundations of learning in neural nets. In fact, it is not even clear anymore which are the parameters of networks of spiking neurons in which their "program" is stored. In particular all classical learning algorithms for neural nets that provide rules for tuning synaptic "weights" become dubious in this context. Furthermore, in view of point (i) it is not even clear what the goal of a learning algorithm for pulsed neural nets should be: the goal to learn a function (from input-vectors to output-vectors of numbers) or a function (from input-functions to output-functions)?

2.7.1 Implications for Learning in Pulsed Neural Nets

In most models for networks of spiking neurons one assumes that the weight (or "efficacy") of a synapse is a parameter that changes only on the slow time scale of learning. It has however been known for quite some time (see for example [9, 56, 101] that synaptic efficacy changes with activity. In [75] there is a plenty of experimental data and associated models that show that it is quite problematic to view a biological synapse as a trivial computational unit that simply multiplies its input with a fixed scalar w, its synaptic "weight". Instead, a biological synapse should be viewed as a rather complex nonlinear dynamical system. It is known that the "hidden parameters" that regulate the dynamics of a biological synapse vary from synapse to synapse [57], and that at least some of these hidden parameters can be changed through long–term potentiation (LTP) (i.e., through "learning"). This has drastic consequences for our view of learning in biological neural systems. Since synapses are history-dependent it does not suffice to consider training examples that consist of input and output vectors of numbers. Instead a neural system learns to map certain input functions of time to given output functions of time. Hence a training example consists of a pair of functions or time series. Furthermore the learning algorithm itself has to specify not only rules for changing the scaling parameter w of a synapse, but also for changing the internal synaptic parameters that regulate the dynamic behavior of the synapse. Experimental evidence that Hebbian learning ("pairing") in biological neural system changes the dynamic behavior of a synapse, rather than just the average amplitude of its responses, was provided by [40]. Their data (see [75]) suggest that learning may redistribute the strength of synaptic responses among the spikes in a train, rather than changing its average response. The particular change in synaptic dynamics makes the postsynaptic neuron more sensitive to transients, i.e., to rapid changes in the firing rate of the presynaptic neuron. In this way synaptic plasticity may change the sensitivity of a neuron to different neural codes. In this example it increases the sensitivity for temporal coding, while at the same time decreasing its limiting frequency above which it is unable to distinguish between different presynaptic firing rates. Liaw and Berger in [105] have shown that interesting changes of the dynamic properties of a neural circuit with dynamic synapses can already be achieved by just changing the scaling factors of excitatory and inhibitory connections, without changing the hidden parameters that control the dynamics of the synapses themselves. Preliminary results from [70] show that by applying gradient descent learning also to the hidden parameters of a synapse, a neural circuit can in principle "learn" to realize quite general given operators that map input time series to given output time series.

In the subsequent section we discuss an unsupervised learning scheme originally proposed by Donald Hebb in 1949. This is of particular interest to us as this would be a foundational tool for learning in our integrated model discussed in the next chapter.

2.7.2 Hebbian Learning of Pulse Timing

Hebb's postulate of learning is the oldest and most famous of all learning rules, it is named in the honor of neuropsychologist Hebb (1949). Quoting from Hebb's book, *The Organization of Behavior* (1949, p62):

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Hebb proposed this change as a basis of associative learning (at the cellular level), which would result in an enduring modification in the activity pattern of a spatially distributed "assembly of nerve cells". We may expand and rephrase it as a two - part rule as follows [108, 82, 41].

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

Such a synapse is called a Hebbian synapse. (The original Hebb rule did not contain part 2). More precisely, we define a Hebbian synapse as a synapse that uses a *time dependent, highly local, and strongly interactive mechanism to increase synaptic efficiency as a function of the correlation between the presynaptic and postsynaptic activities.* From this definition we may deduce the following four key mechanisms (properties) that characterize a Hebbian synapse [35]:

1. *Time-dependent mechanism* This mechanism refers to the fact that the modifications in a Hebbian synapse depend on the exact time of occurrence of the presynaptic and postsynaptic activities.

2. *Local mechanism* By its very nature, a synapse is the transmission site where information-bearing signals (representing ongoing activity in the presynaptic and postsynaptic units) are in *spatiotemporal* contiguity. This locally available information is used by a Hebbian synapse to produce a local synaptic modification that is input-specific. It is this local mechanism that enables a neural network made up of Hebbian synapses to perform unsupervised learning.

3. *Interactive mechanism* Here we note that the occurrence of a change in a Hebbian synapse depends on activity levels on both sides of the synapse. That is, a Hebbian form of learning depends on a "true interaction" between presynaptic and postsynaptic activities in the sense that we cannot make a prediction from either one of these two activities by itself. *Note:* This dependence or interaction may be *deterministic* or *statistical* in nature.

4. *Conjunctional or correlational mechanism* One interpretation of Hebb's postulate of learning is that the condition for a change in synaptic efficiency is the conjunction of presynaptic and postsynaptic activities. Thus, according to this interpretation, the co-occurrence of presynaptic and postsynaptic activities (within a short interval of time) is sufficient to produce the synaptic modification. It is for this reason that a Hebbian synapse is sometimes referred to as a *conjunctional synapse*. For another interpretation of Hebb's postulate of learning, we may think of the interactive mechanism characterizing a Hebbian synapse in statistical terms. In particular, the correlation over time between presynaptic and postsynaptic activities is viewed as being responsible for a synaptic change. Accordingly, a Hebbian synapse is also referred to as a *correlational synapse*.

The relevance of precise temporal spike timing in the cortex or neural systems in general is a fundamental, yet unsolved question [61, 120, 45, 46, 29, 79, 98, 30]. There are, however, a few specialized subsystems for which the relevance of temporal information has been clearly shown. Prominent examples are the electrosensory system of electric fish and the auditory system of barn owls [23, 22, 125, 66, 67].

To explore the effect of learning on pulse coding, we consider the example of auditory processing. The auditory system of the barn owl operates with a temporal precision in the microsecond range. After delay tuning, obtained by applying the learning rule to an integrate and fire model of an auditory neuron, the temporal precision of spike firing is in the range of 20 - 30 microseconds. This level of precision, typical for the early stages in auditory processing, is necessary for sound source localization. The problem of spike-based learning arises whenever a neural system uses a strategy of temporal coding by action potentials [123].

Later I will be show in the next chapter how this idea of learning spike timing in an unsupervised way is useful for learning dynamic variable binding in the integrated neuro-symbolic model.

2.8 Conclusion

In this chapter we surveyed some of the best known results in the area of Pulsed Neural Networks. We started with the idea to find a potential solution to the problem of neural coding. It is clear by now that the functioning of the brain can be better understood if we take into account the timing of spikes. Recent neuro-biological findings have substantiated this claim. We can no longer understand the activities in the cortical region of the brain simply by the rate models of neural networks.

Hebbian type learning was studied in the neuro-biological side, with an aim to answer questions like: How can the timing of spikes be learned during early development of an owl? How can the process of pulse generation and signal transmission be fine-tuned so as to achieve the required temporal precision? For us the relevance of Hebbian type learning is prime, because it is based on the idea of spike timing. In the next chapter I will show how we can use Hebbian type learning of pulse timing to do rule learning.

Chapter 3

Integrating Neuro-biological Models and Logical Reasoning

The foundations for integration of the neuro-biological models with the logical reasoning systems have been laid already in the first two chapters. In this chapter I present a novel and simple model of integrating the above two paradigms. The model is theoretically formulated and then examples and case studies are demonstrated in order to illustrate the novelties of the model.

3.1 Classes of Reasoning

The confluence of studies in psychology, computational neuro-sciences, deductive reasoning and cognitive sciences lead to an emerging consensus – that *reasoning* is central to the understanding of low-level cognition. Here low-level cognition means deductive reasoning or common-sense reasoning. Issues of high-level reasoning involving concepts of intuition, introspection and metaphysical reasoning are not within the scope of computational models for reasoning, that we are interested in. Even though common sense reasoning is trivial for humans in day-to-day life, making computers do common sense reasoning is extremely difficult. This is because, in common sense reasoning humans use a lot of implicit knowledge, and in some cases perform on-the-fly learning in order to make correct deductions. Using logic as a formal tool to model explicit knowledge and use the deductive inferencing strategies e.g., resolution [3, 4] has been very much in practice. However, it is clear from the perspective of neuro-sciences that there is currently no evidence to show how humans could do common-sense reasoning from a large network of neurons. This sets up a challenge for all the people involved in Artificial Intelligence – to develop computational models of human reasoning, which should now explain the reasoning process in human brain in the light of emerging results from neuro-biology. Considering the fact that the research in brain sciences is pretty much in infancy, and the problem of neural coding is far from being solved at this point of time, emphasis should be on using as much as possible inputs from the neuro-biologists, and integrating them with the deductive models of human reasoning in a sufficiently general and extensible framework.

However, before we embark on such an integration of neuro-biology and deductive reasoning, it is imperative to reflect on the class of reasoning problems we intend to solve by such an integration.

3.1.1 Reflexive and Reflective reasoning

Consider a scenario where a person is told that "Alex walked into the room". This person will soon infer out that "Alex is in the room". Shastri and Ajjanagadde in his [111] paper describes such a reasoning as being *reflexive* if one can draw a variety of inferences rapidly, spontaneously, and without conscious effort - as though they were a reflexive response of the cognitive apparatus.

In contrast to such a reflexive reasoning there is another class of low-level reasoning which could be appropriately called as reflective reasoning. This is a reasoning which requires reflection, conscious deliberation and often an overt consideration of alternatives and weighing of possibilities. Reflective reasoning takes longer and often requires the use of external props such as a paper and pencil. Examples of such reasoning are solving logic puzzles, doing crypto-arithmetic, or planning a vacation!

Reflexive reasoning occurs with reference to a large body of *long-term* knowledge. This knowledge forms an integral part of an agent's conceptual representation and is retained for a considerable period of time once it is acquired. An agent's long term knowledge base (LTKB) encodes several kinds of knowledge. Shastri in [111] suggests that long term knowledge should be distinguished from short-term as well as medium-term knowledge. Medium-term knowledge is referred to as the knowledge that persists longer than short-term knowledge and may be remembered for days or even weeks. Such medium-term knowledge, however *may* be forgotten without being integrated into the agent's long term conceptual representation. The distinction between medium- and long-term knowledge is not arbitrary and has a neurological basis. These include specific knowledge about particular entities, relations, events and situations, and general systematic knowledge about the regularities and dependencies in the agent's environment. There are basically two kinds of long-term knowledge – specific knowledge as *facts* (in the first-order logic program) and instantiation - independent knowledge as *rules* (the horn clause rules of the first-order logic).

3.1.2 Reasoning and the Dynamic Binding Problem

Consider a following knowledge base scenario: If X and Y are siblings, and X is male, then X is a brother of Y. If X and Y are siblings, and X is female, then X is a sister of Y. We can formalize these notions in the form of first-order logic rules.

Example: *Family Relation*
Rule 1: $\forall XY. brother(X, Y) \leftarrow sibling(X, Y) \wedge male(X)$
Rule 2: $\forall XY. sister(X, Y) \leftarrow sibling(X, Y) \wedge female(X)$

If we have more knowledge besides the above rules, in the form of facts that $john$ and $conni$ are siblings ($sibling(john, conni)$ and $sibling(conni, john)$), and $john$ is $male$ ($male(john)$) while $conni$ is $female$ ($female(conni)$), then the information that $john$ is a brother of $conni$ and $conni$ is a sister of $john$ can be easily inferred. We cannot infer that $john$ is a sister of $conni$, and $conni$ is a brother of $john$. A little introspection as to how this chain of reasoning takes place would reveal that, there is a certain correspondence from the arguments (X, Y) of the relation *sibling* to the arguments X and Y of the relation $brother$. The correspondence is that the first argument of *sibling* predicate is mapped to the first argument of $brother$ predicate and the second argument of the *sibling* predicate is mapped to the second argument of the $brother$ predicate. Moreover, during the chain of reasoning, the argument X of *male* predicate is mapped to the first argument X of the $brother$ predicate and the argument X of the $female$ predicate is mapped to the first argument X of the *sister* predicate. This mapping of variables from the body of the clause to the head of the clause can be understood as the process of *binding*. This notion of binding is different from the notion of binding as understood in logic programming. Here the variables X and Y of the predicate *sibling* in the body of the clause are bound to the arguments $john$ and *conni* respectively, for the fact $sibling(john, conni)$. The argument X of the *male* predicate in the body of *Rule 1* gets bound to argument of the fact $male(john)$. As a consequence the first argument of $brother$ predicate got bound to *john*, and the second argument of the *brother* predicate got bound to *conni* in *Rule 1*. The concept that *john* is the brother of *conni* is generated in the head of *Rule 1*. Similarly the concept that *conni* is the sister of $john$ evolves in the head of *Rule 2*.

A reflexive reasoning system should be capable of representing facts in a rapid and dynamic fashion. Observe that the reasoning process generates inferred facts dynamically and the reasoning system should be capable of representing these inferred facts. Furthermore, the reasoning system must interact with other processes that communicate facts and pose queries to it, and the system should be capable of dynamically representing such facts and queries. Representing a dynamic fact amounts to representing, dynamically the appropriate bindings between predicate arguments and objects in our domain ($john$ and *conni* in the above example).

3.1.3 Static versus Dynamic Bindings

Bindings could be essentially of two kinds: *static* or *dynamic*. Static binding usually refers to the static encoding using physically interconnected neurons and links to represent bindings between the arguments and the objects (usually the constants in a given logic program). The connection between the argument neurons and the neurons representing constants is *hard-wired*. It is assumed that the existence of these facts have been learned over a long period of time, and the facts are stable in the sense that they represent information that is going to be true for a much longer time and would be stable. However, encoding rules which describe causality present in our domain, requires a dynamic encoding, an encoding which should have the property to change. When presented with examples from a given domain, the dynamic bindings between the argument neurons of the predicates in head and the argument neurons for the predicates in the body should be learnt. So the way in which we encode the rules should allow us to change the links between the neurons, and not have them hard-wired as in the case of facts.

An interesting feature of these dynamic bindings is that during the chain of inferences dynamic bindings propagate quite rapidly within a few hundred milliseconds. This suggests that it is unlikely that there exists mechanisms that can support widespread structural changes and growth of new links within such time scales. As an alternative we can assume that interconnections exist between all possible pairs of arguments and objects (usually constants). These links can normally remain "dormant" and whether or not the links should remain active can be known by making the network learn on example data sets. This concept is central to the solution I present in the next sections.

3.2 The Philosophy of Integration

The idea is to have a neural network which should encode all the knowledge present in the logic program and possibly have the scope of encapsulating more, knowledge that in the beginning may not be known to be precisely correct. Often it happens, when a knowledge engineer models the knowledge about a particular domain it is not possible to know all the possible rules described by *relations or predicates*. Even if it may be possible to describe fully all the rules that explain the characteristic features of the given domain, it may be difficult to know if all the rules are *correct*. To take care of this bottle-neck, these hybrid neural-symbolic models were designed, whereby one can gain *confidence* about an existing rule by going to the domain, and testing the rule on a data set. This is precisely the goal of machine learning, to learn correct relations and rules. In Inductive Logic Programming (ILP), one tries to learn the first order logic rules that correctly model the domain. There are certain statistical functions to evaluate the *confidence* one can have in a given rule. We test the confidence by instantiating the first order clause with ground terms and checking whether or not the rule is correct. In this way the model that I'm going to present is comparable to a machine learning task as the goals that we have are the same, namely to learn the correct rule set, identify discrepancies in the existing rule set, and finally to extract the modified rule set from our connectionist system. However the statistical technique that we employ to evaluate *confidence* in our rule set is the one that comes to us through the learning dynamics of a spiking neural network, rather than any of the other conventional statistical tools like *entropy, relative frequency, m-estimate of accuracy, gain ratio* etc. [77].

3.3 From Neuro-biology to Logic-programming

In the last chapter we reviewed the concepts of pulsed neural networks and examined the possibilities of coding through such networks. In due course it was clear that there is a substantial evidence in neuro-biological literature that the working of pulsed neural network cannot be explained by just rate models. Hence the need of pulse coding was emphasized and some of the potential coding schemes within the realms of neuro-biology were reviewed. The purpose was to find out if spiking networks and pulse coding could be of any direct relevance to the proposed integration of neuro-biological models and logical reasoning systems. As already explained in the first chapter, some of the earlier hybrid models [111, 87, 102], had special temporal units which were a kind of *electronic switches* and they chose to selectively switch on and off a part of the connections in the network. The basic problem with these networks was, that the knowledge that they encoded was pretty much hard wired into the network and though the network could do effective reasoning as in [111, 86] and compute least models as in [103] it could not *learn* anything new. Although Shastri and Ajjanagadde in his [112] outlines a method of learning his SHRUTI like systems by a learning algorithm which is sensitive to the contextual or type information, the core learning algorithm was not technically accessible to me for scrutiny.

In the following sequel, I present a solution to learning a special class of first order logic programs, where the learning algorithm learns to *bind* the variables in the antecedent of the clause to the consequent. This kind of learning solution although applicable to a small subset of first order logic programs as of now, is quite effective to identify the correct rules, given a sample of training data for training the knowledge based spiking neural network. The approach fundamentally remains the same to the one employed by Jude Shavlik and his group in the sense that my approach is also a three tier solution, the first step being transformation of the logic programs to a neural network, the second being training the network on a data set, the third being the extraction of refined rules. However it needs to be stated here that apart from being fundamentally same, my approach is different to any of the previously investigated techniques in the area of rule extraction. In this sense, my approach is novel in the following ways:

- The solution that is being proposed in the following sections will work with the first-order logic programs, although a restricted class at the moment.

- The idea to use a *spiking* or a *pulsed neural network* for knowledge representation and reasoning is new. I haven't found any reference in the literature so far which uses the pulsed neural network for reasoning.

- To use Hebbian type learning for training the knowledge based spiking neural network is, also new in its concept. Of course there has been applications of Hebbian learning to spiking networks as it was the case with [123] however using Hebbian type learning in an unsupervised way for knowledge based neural network is a new approach.

In the next section I formalize a new transformation algorithm for mapping logic programs to connectionist networks (Knowledge Based Spiking Networks). After obtaining a neural network which already encodes some knowledge, the next step is to train the network. However, in my approach since the idea is to do rule-learning, identifying the correct rules and discovering unknown rules, we need not train the whole network but instead the part of the network where the rule encodings are present. This is technically called a *kernel*. Identification of this kernel from the obtained network is the next step after the transformation algorithm. This is explained in the section after the transformation algorithm. Then subsequently I present a mathematical model of Hebbian type learning for these spiking connectionist networks. Thereafter, I would illustrate with some examples how the transformation algorithm together with the learning algorithm helps to solve the subset of the dynamic variable binding problem.

However, there are some important restrictions on the class of first order logic programs which would be dealt here. The first one is that repeated occurrence of an argument within a predicate is not allowed. The second restriction is that "*all variables which occur in the body of a given clause must occur in the head of the same clause, if the body contains more than one predicate, and all variables that occur in the head of the clause must occur in the body of the same clause.*" The third one is on the multiple existence of ground facts, the restriction being that there can't be more than one grounded instance of a fact in a given logic program which is built over the same predicate name and same constants. The fourth restriction is that only nullary function symbols are allowed in the program. The fifth restriction is that we shall not deal with negation. These restrictions would be formally stated in the next section.

As an immediate consequence of the Transformation Algorithm and the Hebbian type learning, rule extraction can be performed on the trained network. However I'll not be able to present a formal model of rule extraction, but instead give the intuitive ideas of how it could be accomplished. Lastly, I present an overview of my proposed system and mention some open questions and probable future research directions.

3.4 Transformation Algorithm

Before we formalize the algorithm for transforming the logic programs to connectionist network, let us define some basic functions which would be handy in formalizing some other concepts later.

3.4.1 Basic Notions

- Let Var be a function which returns the *set* of variables of a given predicate, formally defined as:
  $Var :: Predicate \rightarrow Variables$
  $Var(p(x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\}$ where $\{x_1, \ldots, x_n\}$ is the set of "variables" of the predicate $p(x_1, \ldots, x_n)$.

- Let $Card$ be a function which returns the cardinality of a given set, formally defined as:
  $Card :: Set \rightarrow \mathbb{N}$
  $Card(\{e_1, \ldots, e_n\}) = n$ where n is the number of elements contained in the set $\{e_1, \ldots, e_n\}$.

- Let $Const$ be a function which returns the *set* of constants occurring in the predicate, formally defined as:
  $Const :: Predicate \rightarrow Set\,of\,Constants$
  $Const(p(x_1, \ldots, x_i, c_1, \ldots, c_m, x_{i+1}, \ldots, x_n)) = \{c_1, \ldots, c_m\}$. If there are repeated occurrences of constants, within a predicate, only one is written in the resulting set.

- Let $Pred$ be a function which takes a singleton set of first order clause and returns the *multi set* of the predicate names occurring in the body of the clause, formally defined as:
  $Pred :: Set\,of\,first\,order\,clause \rightarrow Multi\,Set\,of\,Predicates$

$$Pred(\{p(\dots) \leftarrow p_1(\dots) \wedge \dots \wedge p_n(\dots)\}) = \{p_1, \dots, p_n\}$$

- Let Mul be a function which returns the multiplicity of occurrence of a predicate symbol in the body of a given clause, it takes the multi set of the predicate names, and for a given predicate name it returns the number of times the predicate name occurs in the multi set. It is formally defined as:
  $Mul :: Multi\,Set\,of\,Predicates \times Predicate \rightarrow \mathbb{N}$
  $Mul(\{p_1, \dots, p_n\}, p_i) = m_i$, where m_i is the multiplicity of occurrence of the predicate symbol p_i in the given multi set of predicate symbols.

3.4.2 Special Class of Logic Programs

The class of logic programs that would be covered with this algorithm are denoted by $\mathcal{P}$, and the programs are first order logic programs with the following restrictions:

1. Any argument (variable or a constant) within a predicate name is allowed to occur only once. Repeated occurrences of a variable or a constant within a predicate not allowed, e.g., predicates like $p(x, y, x)$, $p(a, a, x)$ are not allowed.

2. For each clause of the form $p(\dots) \leftarrow p_1(\dots) \wedge \dots \wedge p_n(\dots)$ in $\mathcal{P}$, for n variables occuring in this clause, for any variable x_i, where $1 \leq i \leq n$, $x_i \in Var(p)$ *if* $x_i \in \bigcup Var(p_i)$. Informally it means that all variables that occur in the head of a clause must occur in the body of the same clause eventually. Secondly, for all variables $x \in \bigcup_{i=1}^{n} Var(p_i)$, $\exists j$ such that $i \neq j$ and $x \in Var(p_i) \wedge x \in Var(p_j)$ then $x \in Var(p)$. Informally it means that we can deal with the rules where a variable in the body does not occur in the head, but this variable in the body that does not occur in the head occurs only in *one* predicate in the body. If it occurs in *more than one* predicate in the body, then this variable must occur in the head of the same clause.

3. If there are two ground instances of a predicate p_i in the head of more than one rule, and let p_i^1 and p_i^2 denote the first and second occurrence of the ground instance of this predicate. If $\exists c.\ c \in Cons(p_i^1)$ and

$c \in Cons(p_i^2)$, then this is not allowed in programs that are covered by the algorithm. The problem relates to that of *cross-talk* and shall be discussed later in the discussion section. Informally this restriction means that one can't have two facts like $p(a, d, e)$ and $p(e, b, c)$ together in the same program, because the predicates have common name and they share a common constant.

4. Only nullary function symbols are allowed in the program. This means that we only deal with constants and not unary or other higher order functions.

5. We shall not deal with negation. This means literals of the kind $\neg p_i(\dots)$ are not allowed in the program.

The connectionist network corresponding to a first order logic program $\mathcal{P}$ is a *two layered* network with an *input*, and one *output layer*. This connectionist network is a *pulsed neural network*, a network where all the neurons or units are sensitive to time varying inputs, in-fact to trains of pulses or spikes. Now we define the translation function *trans*, which takes a logic program $\mathcal{P}$, and produces an equivalent connectionist network denoted by $\mathcal{N}$.

3.4.3 Translation Function: $trans(\mathcal{P}, \mathcal{N})$

1. For each clause of the form $p(\dots) \leftarrow p_1(\dots) \wedge \dots \wedge p_n(\dots)$ in $\mathcal{P}$ **do**:

   (A) For a clause of the form $p(\dots) \leftarrow p_1(\dots) \wedge \dots \wedge p_n(\dots)$:
   $Pred(\{p(\dots) \leftarrow p_1(\dots) \wedge \dots \wedge p_n(\dots)\}) = \{p_1, \dots, p_n\}$.

   (a) For each predicate p_i in the body of the clause
   let $Mul(p_i, \{p_1, \dots, p_n\}) = m_i$ and if $Var(p_i) = \{x_1, \dots, x_{\theta_i}\}$
   and if $Card(\{x_1, \dots, x_{\theta_i}\}) = \theta_i$, where $1 \leq i \leq n$,
   if there is no input layer **then**:

   (i) draw $m_i \cdot \theta_i$ neurons in a new layer. Name this layer as input layer.

   (ii) name the neurons in input layer from $p_i[1]$ to $p_i[m_i \cdot \theta_i]$.

   else if there is an input layer, but there is **no** neuron named $p_i[\theta_i]$, **then**

   (i) draw $m_i \cdot \theta_i$ neurons in the input layer.

   (ii) name the neurons in the input layer from $p_i[1]$ to $p_i[m_i \cdot \theta_i]$.

else if there is an input layer with neurons named $p_i[1], \ldots, p_i[\theta_j]$ where $\theta_j < m_i \cdot \theta_i$, **then**

(i) draw $m_i \cdot \theta_i - \theta_j$ neurons in the input layer.

(ii) name the neurons in the input layer from $p_i[\theta_j + 1]$ to $p_i[m_i \cdot \theta_i]$.

else if there is an input layer with neurons named $p_i[1], \ldots, p_i[\theta_j]$ where $\theta_j \geq \theta_i$, **then**

(i) do nothing.

(b) For each predicate p_i in the body of the clause, if $Const(p_i) = \{c_1, \ldots, c_{t_i}\}$, where $Card(\{c_1, \ldots, c_{t_i}\}) = t_i$, where $1 \leq i \leq n$

for$(j = 1; j \leq t_i; j++)$

{

For a constant $c_j \in \{c_1, \ldots, c_{t_i}\}$,

if the neuron $p[c_j]$ is **not** occuring in the input layer, **then**

(i) draw a neuron in the input layer, and name it as $p[c_j]$.

}

(B) For each predicate p in the head of the clause:

(a) For Variables in the predicate: with $Card(Var(p)) = \theta$

if there is not an output layer **then**

(i) draw a new layer, name the layer as "output layer".

(ii) draw $p[1], \ldots, p[\theta]$ neurons in the output layer.

else if there is an output layer but there is no neuron named as $p[\ldots]$ **then**

(i) draw θ neurons in the output layer

(ii) name them as $p[1], \ldots, p[\theta]$.

(b) For Constants in the predicate: with $Const(p) = \{c_1, \ldots, c_t\}$, where $Card(\{c_1, \ldots, c_t\}) = t$

for$(j = 1; j \leq t; j++)$

{

For a constant $c_j \in \{c_1, \ldots, c_t\}$,

if the neuron $p[c_j]$ is **not** occuring in the output layer, **then**

(i) draw a neuron in the output layer, and name it as $p[c_j]$.

}

(C) After the above steps, let the total number of neurons (representing variables[1]) corresponding to the predicates in the body be m and the total number of neurons (representing variables) corresponding to the predicate in the head be n then
for$(counter_m = 1;\ counter_m \leq m;\ counter_m + +)$
{
for$(counter_n = 1;\ counter_n \leq n;\ counter_n + +)$
{
connect neuron no. $counter_m$ *in the input layer to* $counter_n$ *in the output layer by arrows pointing outwards from the input layer.*
}
}

(D) For each constant c_j, occuring in the clause, if c_j occurs in the head in the predicate p and in the body in the predicate p_i, then connect each such $p_i[c_j]$ in the input layer to the neuron $p[c_j]$ in the output layer, by an arrow (projecting outwards from the input layer).

2. For each clause of the form $p_i(\dots)$.,

   (a) With variables occuring in the fact, the number given by
   $Card(Var(p_i)) = \theta_i$
   if the predicate p_i is **not** occuring in the body of any clause and
   if there is an output layer having θ_j neurons named $p_i[1], \dots, p_i[\theta_j]$
   and $\theta_j < \theta_i$**then**

      (i) draw θ_i neurons in the input layer.
      (ii) name them as $1, \dots, \theta_i$.
      (iii) draw $\theta_i - \theta_j$ neurons in the output layer.
      (iv) name them from $p_i[\theta_j + 1]$ to $p_i[\theta_i]$.

[1]neurons corresponding to variables are named as $p_i[1], \dots, p_i[m_i]$ for a predicate p_i, where the numbers $1, \dots, m_i$ are used to denote the variable occurrences for an argument in the predicate p_i

(v) connect $1, \ldots, \theta_i$ neurons in the input layer to the corresponding p_i neuron in the output layer, by arrows pointing outwards from the input layer.

else if there is an output layer but **no** p_i neuron **then**

(i) draw θ_i neurons in the input layer, name them from $1, \ldots, \theta_i$.

(ii) draw θ_i neurons in the output layer, and name them as $p_i[1] \ldots, p_i[\theta_i]$

(iii) connect each θ_i neuron in the input layer to the corresponding $p_i[\theta_i]$ neuron in the output layer, by arrows pointing outwards from the input layer.

(b) With constants occuring in the fact, with
$Const(p_i) = \{c_1, \ldots, c_t\}$, and
$Card(\{c_1, \ldots, c_t\}) = t$
for$(j = 1; j \leq t; j++)$
{
For a constant $c_j \in \{c_1, \ldots, c_t\}$,
if the neuron $p_i[c_j]$ is **not** occuring in the output layer **and** the neuron c_j is not occuring in the input layer **then**

(i) draw a neuron in the output layer, name it as $p_i[c_j]$.

(ii) draw a neuron in the input layer and name it as c_j.

(iii) connect the neuron c_j in the input layer to the neuron $p_i[c_j]$ by an arrow (projecting outwards from the input layer).

else if the neuron $p_i[c_j]$ is **not** occuring in the output layer, but the neuron c_j is occuring in the input layer **then**

(i) draw a neuron in the output layer, and name it as $p_i[c_j]$.

(ii) connect the neuron c_j in the input layer by an arrow (projecting outwards from the input layer) to the neuron $p_i[c_j]$ in the output layer.

else if the neuron $p_i[c_j]$ is occuring in the output layer, but the neuron c_j is not occuring in the input layer **then**

(i) draw a neuron c_j in the input layer, and

(ii) connect the neuron c_j in the input layer to the neuron $p_i[c_j]$ in the output layer by an arrow (projecting outwards from the input layer).

}

3. For each clause of the form $p_i(\dots)$.,

   (a) With variables occuring in the fact the number given by $Card(Var(p_i)) = \theta_i$,
   if the predicate p_i occurs in the body of some clause and
   if there is an output layer but **no** p_i neuron **then**

      (i) draw θ_i neurons in the output layer, and name them from $p_i[1]$ to $p_i[\theta_i]$.
      (ii) connect each $p_i[\theta_i]$ neuron in the input layer to the corresponding $p_i[\theta_i]$ neuron in the output layer, by arrows pointing outwards from the input layer.

   else if there is an output layer having $p_i[\dots]$ neurons **then**

      (i) do nothing

   (b) With constants occuring in the fact, with
   $Const(p_i) = \{c_1, \dots, c_t\}$, and
   $Card(\{c_1, \dots, c_t\}) = t$
   for$(j = 1;\ j \leq t;\ j++)$
   {
   For a constant $c_j \in \{c_1, \dots, c_t\}$,
   if the neuron $p_i[c_j]$ is **not** occuring in the output layer **and** a neuron $c_j$is not occuring in the input layer **then**

      (i) draw a neuron in the output layer, name it as $p_i[c_j]$.
      (ii) draw a neuron in the input layer and name it as c_j.
      (iii) connect the neuron c_j in the input layer to the neuron $p_i[c_j]$ by an arrow (projecting outwards from the input layer).

   else if the neuron $p_i[c_j]$ is **not** occuring in the output layer, but a neuron c_j is occuring in the input layer **then**

      (i) draw a neuron in the output layer, and name it as $p_i[c_j]$.
      (ii) connect the neuron c_j in the input layer by an arrow (projecting outwards from the input layer) to the neuron $p_i[c_j]$ in the output layer.

   else if the neuron $p_i[c_j]$ is occuring in the output layer, but the neuron c_j is not occuring in the input layer **then**

      (i) draw a neuron c_j in the input layer, and

(ii) connect the neuron c_j in the input layer to the neuron $p_i[c_j]$ in the output layer by an arrow (projecting outwards from the input layer).

}

4. Recursive links: For drawing recursive links, one has to exercise some caution and has to carefully examine the clauses in the given program, and depending upon some of the special cases, one needs to handle them differently. This is an important step, as computing the correct models from *long term memory* (facts) is tricky, more so if there are *rules with some constants in the predicates of the body of the rule* or *if there are multiple occurrences of variables across several different predicates* or *if there are multiple occurrences of variables across several multiple occurrences of the predicate within the body of the same clause.* We segregate these cases now and classify them as special cases for our programs. For each case we outline the procedure to preclude to draw wrong connections between the output and the input layer, by the recursive link.

   (a) *Rules with multiple occurrence of variables in different predicates within the body of the same rule:* Consider a rule of the form:
   Rule 1: $p(\ldots, x_{\theta_k}, \ldots) \leftarrow p_i(\ldots, x_{\theta_i}, \ldots) \wedge p_j(\ldots, x_{\theta_j}, \ldots)$, where a variable x occurs at position θ_i in predicate p_i, at position θ_j in predicate p_j and at position θ_k in the predicate p. Given such a rule and there are other rules of the form:
   Rule 2: $p_i(\ldots, a_{\theta_i}, \ldots) \leftarrow p_k(\ldots)$. and
   Rule 3: $p_j(\ldots, a_{\theta_j}, \ldots) \leftarrow p_l(\ldots)$, where an argument a, occurs in the head (as a variable or a constant) of the above rules at positions θ_i and θ_j, and $0 \leq k \leq n$, $0 \leq l \leq m$. There could be n predicates in the body of *Rule 2* and m predicates in the body of *Rule 3*. If the arguments at position θ_i in p_i and at position θ_j in p_j in *Rule 2* and *Rule 3* resp. are not identical then we don't draw any recursive links from the neurons corresponding to predicate assemblies p_i and p_j in *Rule 2* and *Rule 3* resp. in the output layer to the predicate assemblies p_i and p_j in the input layer.

   (b) *Rules with multiple occurrence of variables in multiple occurrence of a predicate within the body of the same rule:* Consider a rule of the form:

Rule 1: $p(\ldots, x_{\theta_k}, \ldots) \leftarrow p_i(\ldots, x_{\theta_i}, \ldots) \wedge p_i(\ldots, x_{\theta_j}, \ldots)$, where a variable x occurs at position θ_i in the first occurrence of the predicate p_i, at position θ_j in the second occurrence of the predicate p_j and at position θ_k in the predicate p. If there are other rules of the form:
Rule 2 $p_i(\ldots, a_{\theta_i}, \ldots, a_{\theta_j}, \ldots) \leftarrow p_k(\ldots)$. If the arguments at position θ_i and θ_j are not identical, then we don't draw any recursive links from the neurons corresponding to the predicate assembly p_i in *Rule 2*, to the neurons corresponding to the predicate assemblies p_i in *Rule 1* in the input layer.

(c) *Rules with some constants in the predicate of the body of a rule:* Consider a rule of the form:
Rule 1 $p(\ldots, c_{\theta_k}''', \ldots) \leftarrow p_i(\ldots, c_{\theta_i}', \ldots) \wedge p_j(\ldots, c_{\theta_j}'', \ldots)$, where a constant c' occurs at position θ_i in predicate p_i, a constant c'' at position θ_j in predicate p_j and a constant c''' at position θ_k in the predicate p. If there are other rules of the form:
Rule 2: $p_i(\ldots, d_{\theta_i}', \ldots) \leftarrow p_k(\ldots)$. **or**
Rule 3: $p_j(\ldots, d_{\theta_j}'', \ldots) \leftarrow p_l(\ldots)$., where d', and d'' are constants occuring in p_i in *Rule 2* and p_j in *Rule 3* at position θ_i and θ_j respectively, and $0 \leq k \leq n$, $0 \leq l \leq m$. There could be n predicates in the body of *Rule 2* and m predicates in the body of *Rule 3*. If c' is not identical to d' **or** c'' is not identical to d'' then we don't draw any recursive links from the neurons corresponding to the predicate assemblies p_i and p_j in *Rule 2* and *Rule 3* resp. to the neurons corresponding to the predicate assembly p_i and p_j occuring in the body of *Rule 1*.

It should be noted that if the given program doesn't have any of the above special cases, then one can easily draw the recursive connections as explained below.

For each predicate p occuring in the head of the clause, if p occurs in the body of any clause, and the multiplicity of occurrence of this predicate p be m in same or different clause, then for m occurrences of the predicate p in the body, where there are $Card(Var(p)) + Card(Const(p)) = n$ neurons in the head of the clause, there would be $m \times n$ recursive links from the output layer to the input layer. The connections are made as follows:

Case I *If a predicate p in the head of a clause is occurring in the body of the same clause and the predicate occurrence in head and in the body contains only variables:* In such a case, if variables at position i in the predicate p in the head is identical to the variable at position j in the predicate p in the body, then connect the neuron (in the output layer) corresponding to the variable in predicate p in the head ($p[i]$), to the neuron (in the input layer) corresponding to the variable in the predicate p in the body ($p[j]$ or j), by an arrow directed into the input layer.

Case II *If a predicate p in the head of a clause is occuring in the body of a different clause, and the predicate occurrence in head and in the body contains only variables:* In this case, connect each $p[\theta_i]$ neuron corresponding to the predicate p in the head to each $p[\theta_i]$ or θ_i neuron corresponding to the predicate p in the body.

Case III *If a predicate p in the head and in the body of the same or different clause contains identical constants:* In such a case the corresponding neurons $p[c_j]$ in the output and the input layer are connected by an arrow directed into the input layer.

Case IV *If a predicate p in the head of a clause contains a constant and the predicate p in the body of same or different clause does not contain that constant:* When the constant c at position i is only occuring in the head predicate p and nowhere in the body predicate p of any clause, then it is connected to the neuron corresponding to the argument occurrence at position i in the body predicate p.

Case V *If a predicate p in the body of a clause contains a constant and the predicate p in the head of same or different clause does not contain that constant:* When the constant c at position i is only occuring in the body predicate p and nowhere in the head predicate p of an any clause, then it is connected to the neuron corresponding to the argument occurrence at position i in the head predicate p.

The above steps make the recursive part of the network. In Appendix B, we show several examples to illustrate the cases above.

5. Initialize the weights on all the arrows connecting input layer to output layer, in this network to 1.0.

6. Add arrows entering into the neurons in the input layer in case there is not yet an arrow drawn into a neuron in the input layer. Draw arrows going outwards from the neurons in the output layer again in case there is not yet an arrow going out of a neuron in the output layer.

We shall see an example now to see how this transformation algorithm works. As an example program consider the logic program $\mathcal{P}$:

Example I
Rule 1: $q(c, X, Y) \leftarrow r(c, Y, X)$
Rule 2: $r(c, b, a)$.

It should be noted that this program fulfills all the restrictions that we have discussed before, and hence we could safely apply the function *trans*. The effect of transformation function is explained step by step below:

Step 1: Rule 1 is of the form $p(\ldots) \leftarrow p_1(\ldots) \wedge \ldots \wedge p_n(\ldots)$, so we execute point 1 (A). of the *trans* function.
$Pred(q(c, X, Y) \leftarrow r(c, Y, X)) = \{r\}$. There is only a single predicate[2] in the body of the clause, so $i = 1$. $Mul(r, \{r\}) = 1$. $Var(r) = \{X, Y\}, Card(\{X, Y\}) = 2$. By getting the values for $Pred$, Mul, Var and $Card$, we have executed part (a) of the point 1 (A). Now we check the **if** condition – the condition holds and there is not yet an input layer, so we execute point (i) and (ii) within part 1 (A)-(a). We draw 2 neurons in the input layer and name them $r[1]$, $r[2]$. These neurons act like place holders for the variables Y and X in the body of *Rule 1*.

Step 2: Now for the same rule, we now execute point 1 (A) – (b) of the *trans* function. $Const(r) = \{c\}$ and $Card(\{c\}) = 1$. The **for** loop is now executed, and the **if** condition is checked. The neuron $r[c]$ does not exist in the input layer yet, so we draw a neuron in the input layer and name it as $r[c]$. After executing the steps so far we get the network shown in the Figure 3.1.

Step 3: We are now at point 1 (B) of our *trans* function. So for the predicate q occuring in the head of rule 1, $Card(Var(p)) = 2$. Since there is

[2]here we use capital letters to write variables, in order to make them distinct from constants, this is also in line with the standard notion used in Prolog compilers

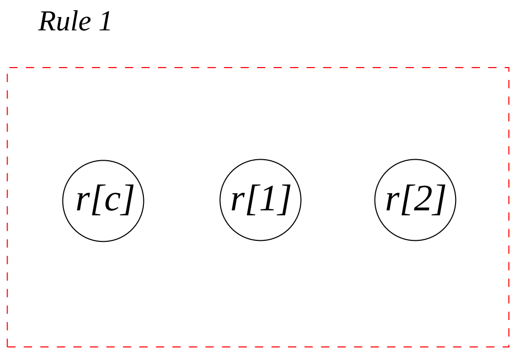


Figure 3.1: Applying the transformation algorithm

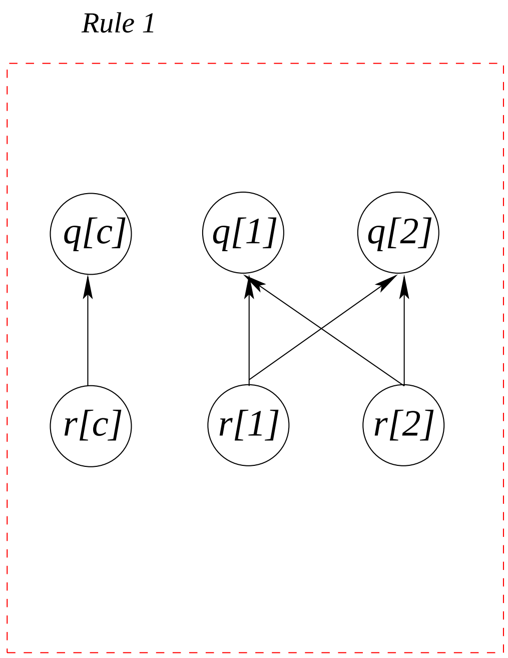


Figure 3.2: Applying the transformation algorithm

not an output layer yet, we draw a neuron named $q[1]$ and $q[2]$ in a new layer above the input layer and name the layer as output layer. Now we are at part 1 (B)–(b). $Const(q) = \{c\}$ and $Card\{c\} = 1$. We execute the **for** loop and draw a neuron $q[c]$ in the output layer. After this we are at part 1 (C). The number of neurons representing variables in the input layer corresponding to the predicate r in the body is 2, and the number of neurons representing variables corresponding to the predicate q in the head is also 2. Now we execute the nested **for** loop. This means that we connect each neuron $r[1]$ and $r[2]$ to both $q[1]$ and $q[2]$ by arrows as explained in the algorithm. This ensures the full connection between the input and the output layers in the part where the neurons representing the variables are present (see Figure 3.2). It should be noted here, that *this is this part of the network that will undergo later, a learning and it is by training this part that the rule learning is expected to take place.* After the above steps we reach the part 1(D) of the *trans* function. Here we connect the neuron $r[c]$ in the input layer to the neuron $q[c]$ in the output layer (by an arrow projecting outwards from the input layer).

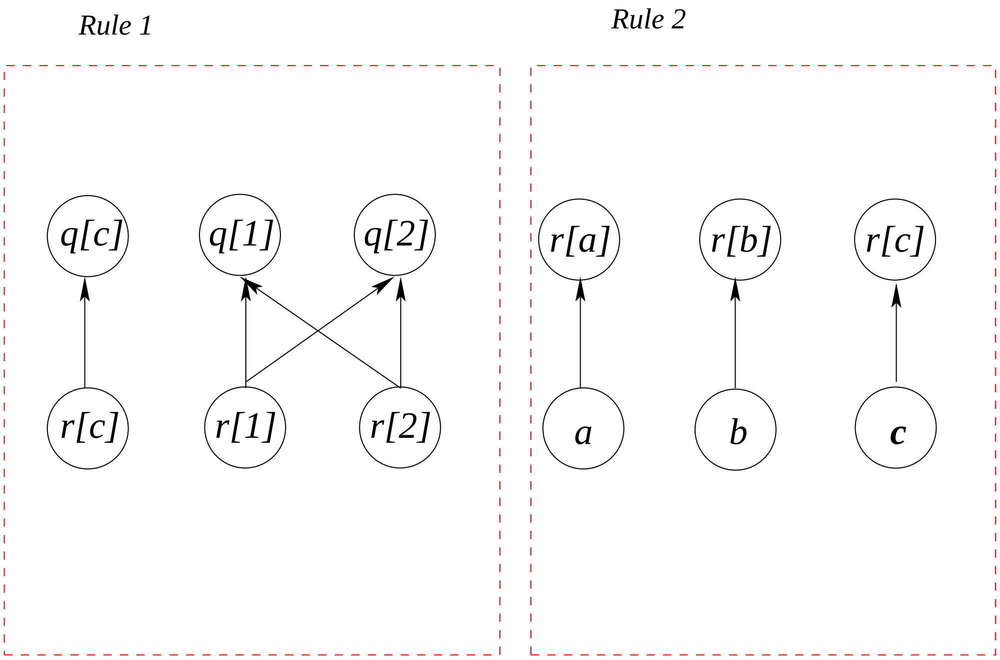


Figure 3.3: Applying the transformation algorithm

Step 4: Now we switch to *Rule 2*. This is of the form $p_i(\ldots)$. . We are at part 2 of the *trans* function. $Card(Var(r)) = 0$. Now since the predicate r occurs in the body of *Rule 1* we can't execute anymore the part 2 as the **if** condition is not satisfied. We reach part 3 of the algorithm. Since $Card(Var(r)) = 0$, we don't draw any neurons corresponding to

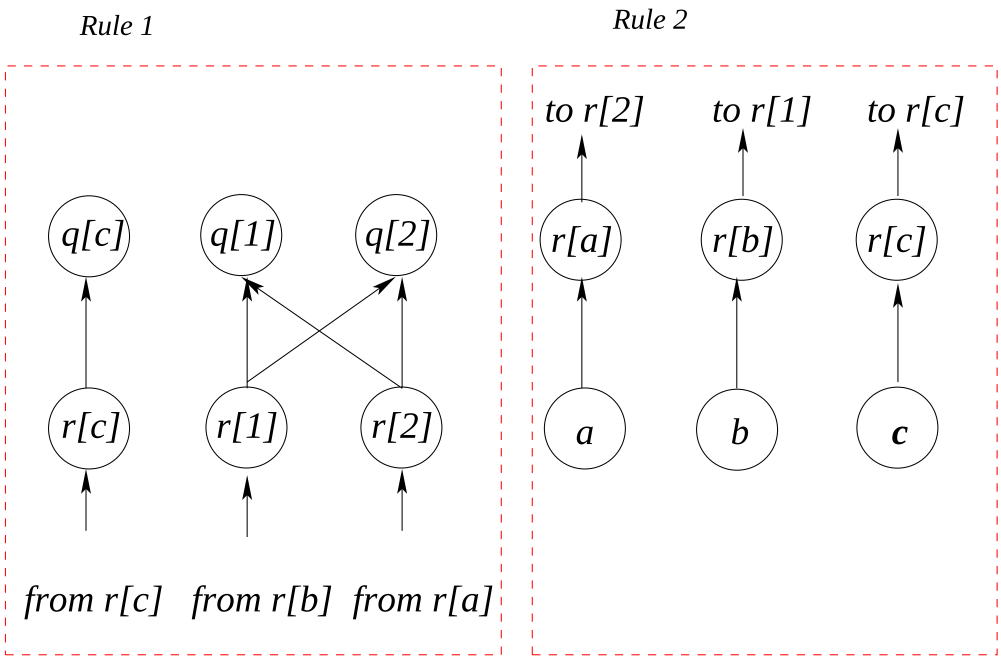


Figure 3.4: Applying the transformation algorithm

the part 3 (a). We check the part 3 (b). $Const(r) = \{c, b, a\}$. The **for** loop is executed next. The **if** condition is satisfied, the neuron $r[a]$, $r[b]$ and $r[c]$ are not present in the output layer and neurons a, b, and c are not present in the input layer, so part 3 (b)– (i), (ii) and (iii) are executed. Neurons $r[a]$, $r[b]$ and $r[c]$ are drawn in the output layer. Neurons a, b and c are drawn in the input layer. The neurons a, b and c are connected by upward directed arrows to neurons $r[a]$, $r[b]$ and $r[c]$ respectively. The network structure is shown in Figure 3.3.

Step 5: We have the feed-forward part of the network ready. We now execute the point 4 of the *trans* function. Here recursive links are made from r[c] in the output layer to $r[c]$ in the input layer, and from $r[b]$ and $r[a]$ to $r[1]$ and $r[2]$ respectively. Since the multiplicity of occurrence of the predicate r in the body of any rule is 1, and we have 3 neurons corresponding to the predicate r in the output layer, we have 3 recursive links drawn (see Figure 3.4).

Step 6: In the last step, add arrows entering into neurons a, b and c in the input layer and arrows going out of the neurons $q[c]$, $q[1]$ and $q[2]$. The final network is in Figure 3.5.The weights on all the arrows connecting the input layer and the output layer is set to 1.0.

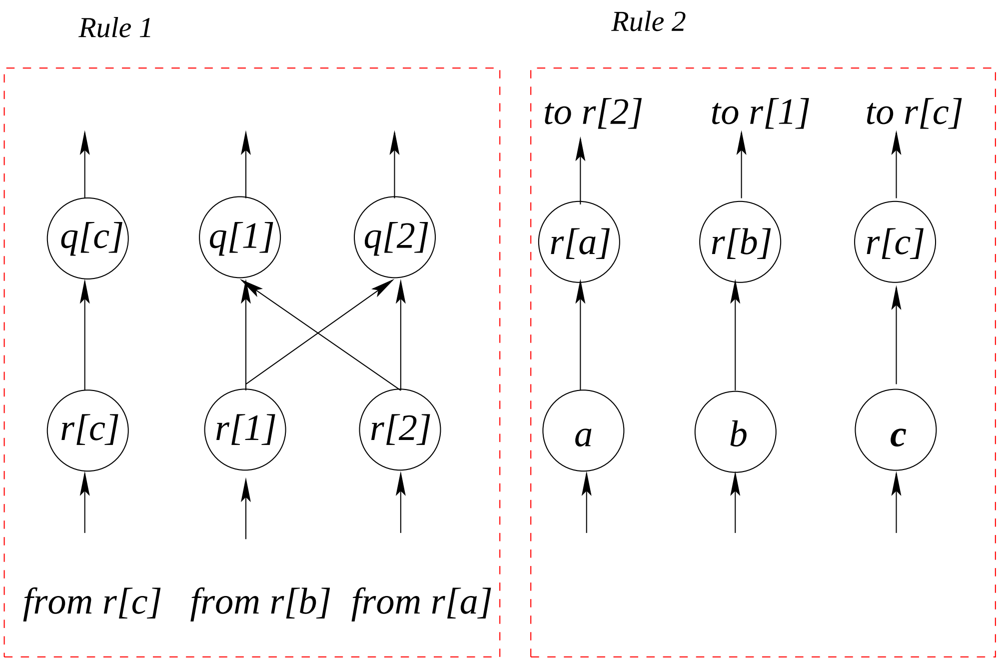


Figure 3.5: After applying the transformation algorithm

In the above program while connecting the neurons $r[c]$, $r[b]$ and $r[a]$ in the output layer to the $r[\ldots]$ neurons in the input layer, it is clear that in the input layer there is only $r[c]$, but not $r[b]$ and $r[a]$. In this case, one connects the $r[b]$ and $r[c]$ neurons in the output layer to $r[1]$ and $r[2]$ neurons (which actually correspond to variables in *Rule 1*) in the input layer, by arrows projecting outwards from the output layer. Altogether there would be 3×1 recursive links, for the above program.

3.5 Properties of the Knowledge Based Spiking Network

The *trans* function produces a neural network from a given logic program. In the subsequent paragraphs we examine some of the properties of this network.

1. The network $\mathcal{N}$ produced by the *trans* function is a pulsed neural network. Hence the network $\mathcal{N}$ enjoys the properties of a pulsed neural network. We outline these properties in the paragraphs below.

2. Each neuron i in this network is a spiking neuron, characterized by a set of firing times. There are fundamentally two different kinds of dynamics taking place in the network. One operates at a slower time scale and the other takes place at a faster time scale. The first is called the *neuronal* dynamics and the second is understood as the *learning* dynamics. For purposes of reflexive reasoning it is understood that the neuronal dynamics would be too slow to explain a possible chain of inference. However, learning dynamics which takes place in the order of few hundred milliseconds, can explain how the humans could possibly learn to do reflexive reasoning to arrive at correct deductions. This has an impact on the foundation for our model. Neuronal dynamics deals with the threshold of a neuron and the neuron i is suppose to fire when this threshold or electrical membrane potential (increasing from the minimum rather incrementally) is exceeded by the weighted inputs coming from j presynaptic inputs. Now since this is too slow a process, would it make sense to predict the firing time of a neuron involved in reflexive reasoning, by waiting for the threshold to be exceeded ? Here the answer is clear, it will not make a good model of reasoning, where by a group of neurons which are suppose to fire at the same time (because they receive inputs from a feature at the same time) don't fire because they await the threshold to be exceeded by the weighted inputs! Hence it is believed that reflexive reasoning which involves synchronous firing of appropriate nodes, cannot take place if the neurons in the network are characterized by slow neuronal dynamics. It is for this reason that we make an abstraction from the spiking neuron model presented in Chapter 2. As a first abstraction we make a simplification of the model, by rejecting the equation (2.13). This is done because of the following two reasons.

   - First we don't include the η expression in our model. This is because we want to use Hebbian type learning for which it is important that there is a conjunctional or correlational mechanism between a pair of neuron. As it was explained in Chapter 2, the η

contributions describe the response of a neuron i to its own spikes, so we can safely neglect the η part from the model.

- The second part of the equation is rejected because the ϵ function models the neuronal dynamics (Excitatory Post Synaptic Potential) after the postsynaptic firing has taken place. Since we are not interested in the slow neuronal dynamics we don't include the second half of the equation as well. This leads us to a stage whereby we can formulate a new spiking neuron model for our reasoning system.

Each neuron in our network $\mathcal{N}$ is characterized by the set of firing times given by

$$\mathcal{F}_i = \{t_i{}^f; 1 \leq f \leq n\} \tag{3.1}$$

and that each neuron may receive input from j presynaptic neurons

$$\Gamma_i = \{j \mid j \; presynaptic \; to \; i\} \tag{3.2}$$

We denote the presynaptic spike train by

$$S_j(t) \;=\; \sum_{t_j{}^f \in \mathcal{F}_j} \delta(t - t_j{}^f) \tag{3.3}$$

where $t_j{}^f$ denotes the arrival time at the synapse j. The spike train at the output is $S_i(t) = \sum_{t_i{}^f \in \mathcal{F}_i} \delta(t - t_i{}^f)$ where $t_i{}^f$ are the firing times and $\mathcal{F}_i$ is the set of all output spikes. Together the equations (3.1) - (3.3) constitute the simplified spiking neuron model for the network $\mathcal{N}$.

3. The neurons in the network $\mathcal{N}$ are connected by weighted links. When we train the network, we actually change the weights on these links. If a neuron i is receiving input from a neuron j, and the pair is connected by a link with weight W_{ij}, then if the neuron i fires shortly after or at the same time as the neuron j, then this weight is strengthened. If the neuron i fires before the neuron j fired the weight is decreased. The weight change is sensitive only to the spike timing differences between the input and the output neuron, very much in contrast to conventional sigmoidal feed-forward neural networks. Later in the section on learning, we formalize the weight update function.

4. A part of the network is fully connected. This is the part of the network which encodes the first order clauses with variables. This is the part of the network which is called a *kernel*. It is this kernel that we train on a sample data, and based on the quality of sample data, we establish either confidence in the clause or reject the clause. The full connection is made to ensure the possibility of learning rules that are not present in the program. If there is sufficient data in the domain, favoring the existence of a rule (that actually did not exist in our program), we would confirm whether or not to keep the rule, after the training is over. This is done by reading the weight values after the training, and pruning away weights with values less than a threshold.

5. This network $\mathcal{N}$ can compute the least model of the given logic program. More over since the network can learn the correct rule set, it can even learn to compute the correct models of the given program. Infact this network can compute not only the least model of a given logic program but also predict models — which can have possibly more information than what is given in the program. Of course whether or not all the information contained in this large model is correct depends on identifying the correct rule set. This task as explained above is handled by the learning algorithm.

6. Since we can compute the least models of the given logic programs, we can also do reflexive reasoning, not only within the least model but in a larger model.

3.6 Identifying the Kernel

After the *trans* function produces a spiking network, we identify the kernel from this network and feed this kernel to the learning algorithm. As explained above kernel is that part of the network which is fully connected between the input and the output layer. Full connection means that all the neurons in the output layer (representing variables named as $p[1], \ldots, p[\theta]$) for the predicate p appearing in the head of a clause, are connected to all the neurons (representing variables named as $p_i[1], \ldots, p_i[\theta_i]$ for a predicate p_i occuring in the body of a rule) in the input layer.

We extract that part of the network as a kernel, where all neurons in the output layer are named as $p[1], \ldots, p[\theta]$ where $1 \leq \theta \leq m$ for m vari-

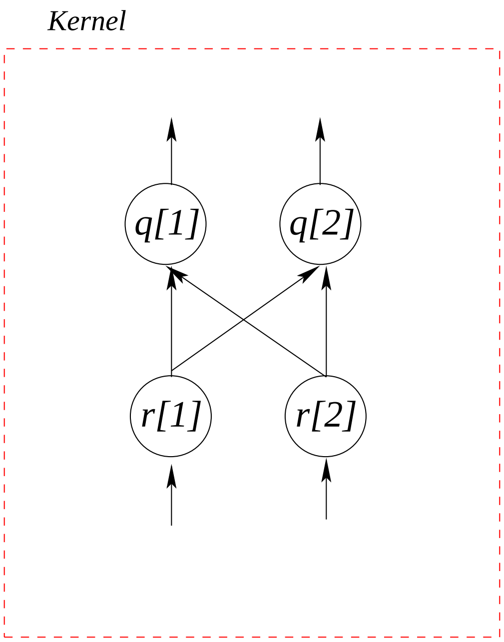


Figure 3.6: An example of a kernel

ables in the head of a rule and all neurons in the input layer are named as $p_i[1], \ldots, p_i[\theta_i]$, for $1 \leq i \leq n$, where there are θ_1 variables in the predicate p_1, θ_2 variables in predicate p_2 and so on. For the network shown in Figure 3.5 for the program $\mathcal{P}$, the equivalent kernel is shown in Figure 3.6.

3.7 Computing the Model

It will be shown in this section how one can compute the models from the temporal dynamics of the neural network. Once we obtain the kernel from the network, we train it by using a learning algorithm. The learning algorithm shall be explained in the next section, however here let us assume that we feed this network to a learning algorithm, and the learning algorithm decides certain weights to be pruned out, and certain others to be retained. This means that after the training we will have a different network than the original one, mainly because the kernel part after undergoing the training shall see some modifications. As an example let us consider the following program $\mathcal{P}$:

Example II

Rule 1: $q(c, X, Y) \leftarrow r(c, Y, X) \wedge s(X, Y, c)$

Rule 2: $r(c, b, a)$.

Rule 3: $r(d, e, f)$.

Rule 4: $s(a, b, c)$.

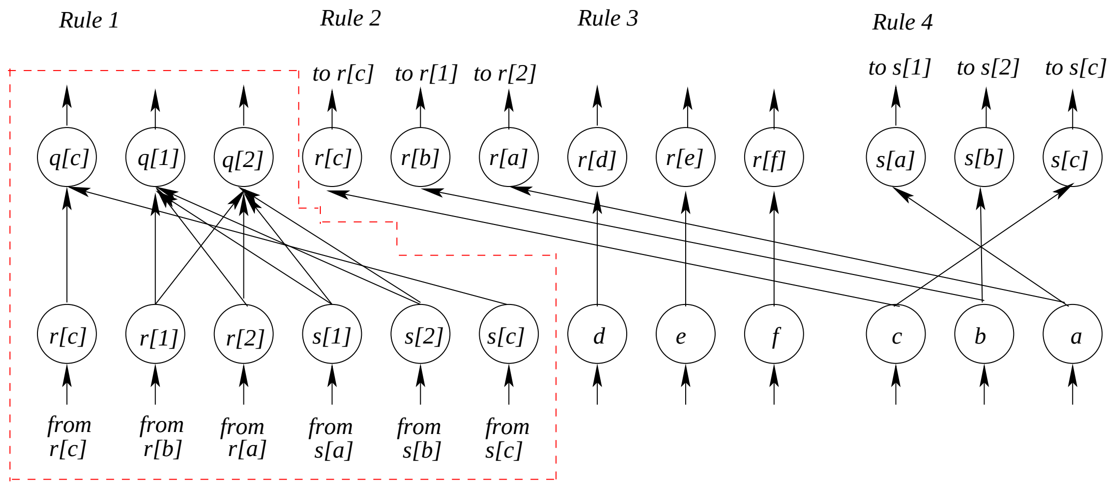


Figure 3.7: After applying the transformation algorithm

Following the transformation algorithm we can construct a network as shown in the Figure 3.7. After undergoing the training let us assume the weight update function in the learning algorithm decides to prune the weights connecting the following output-input pairs: $\langle q[1], r[1]\rangle$, $\langle q[2], r[2]\rangle$, $\langle q[2], s[1]\rangle$, $\langle q[1], s[2]\rangle$. This means that we can no longer make the bindings from the first variable argument of r in the body of *Rule 1*, (Y in the above case) to the first variable argument of q (X in the above case) in the head of *Rule 1*, similarly we cannot bind the second variable argument of r (X in the above case) to the second variable argument of q (Y in the above case). Moreover we cannot make bindings from the first variable argument of s to the second variable argument of q, and from the second variable argument of s to the first variable argument of q. This essentially means we prohibit the possibility to have rules where we could have had the above bindings. Hence when we shall compute the model from the timing diagram, we can com-

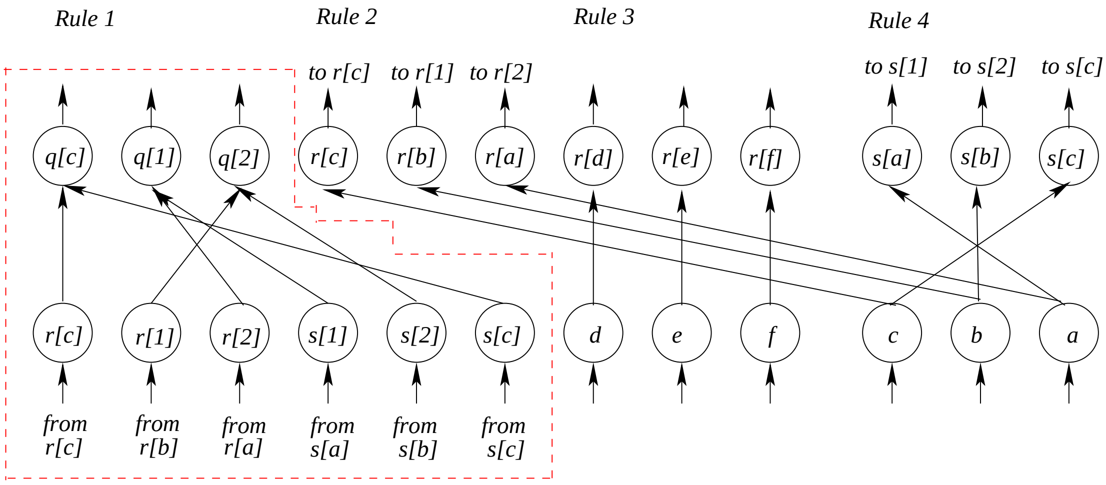


Figure 3.8: After training the network

pute only the model corresponding to the correct rules that we have learnt. Hence after the training we shall get the network shown in the figure 3.8. Let us understand how we compute the model from the spike timing of the neurons, with respect to the above program. We activate the network by an external mechanism, whereby we present facts to the neural network. This means we activate the units representing a, b, c and d, e and f. By activating these neurons, we initiate the cycle of inference. This will in turn activate the predicate assembly $r(c, b, a)$, $s(a, b, c)$, $r(d, e, f)$. This is reflected by the activation of the neurons $r[c]$, $r[b]$, $r[a]$, $s[a]$, $s[b]$, $s[c]$, $r[d]$, $r[e]$, and $r[f]$ in the input layer.

In the next cycle of inference, we read the activations as they pass from the facts to the rule assembly via the recursion links from the output layer to the input layer (see figure 3.9). This means that the following neurons become active in the output layer in the second inference cycle:

- $q[c]$ and $r[c]$ together in the same phase as $s[c]$ and c. This implies that the first argument of r and third argument of s in the body of *Rule 1* gets bound to the constant c. The first argument of q also gets bound to c.

- $q[1]$ and $r[2]$ together with $r[a]$, $s[a]$, $s[1]$ and the constant a. This implies that the third argument of r, and the first argument of s in the

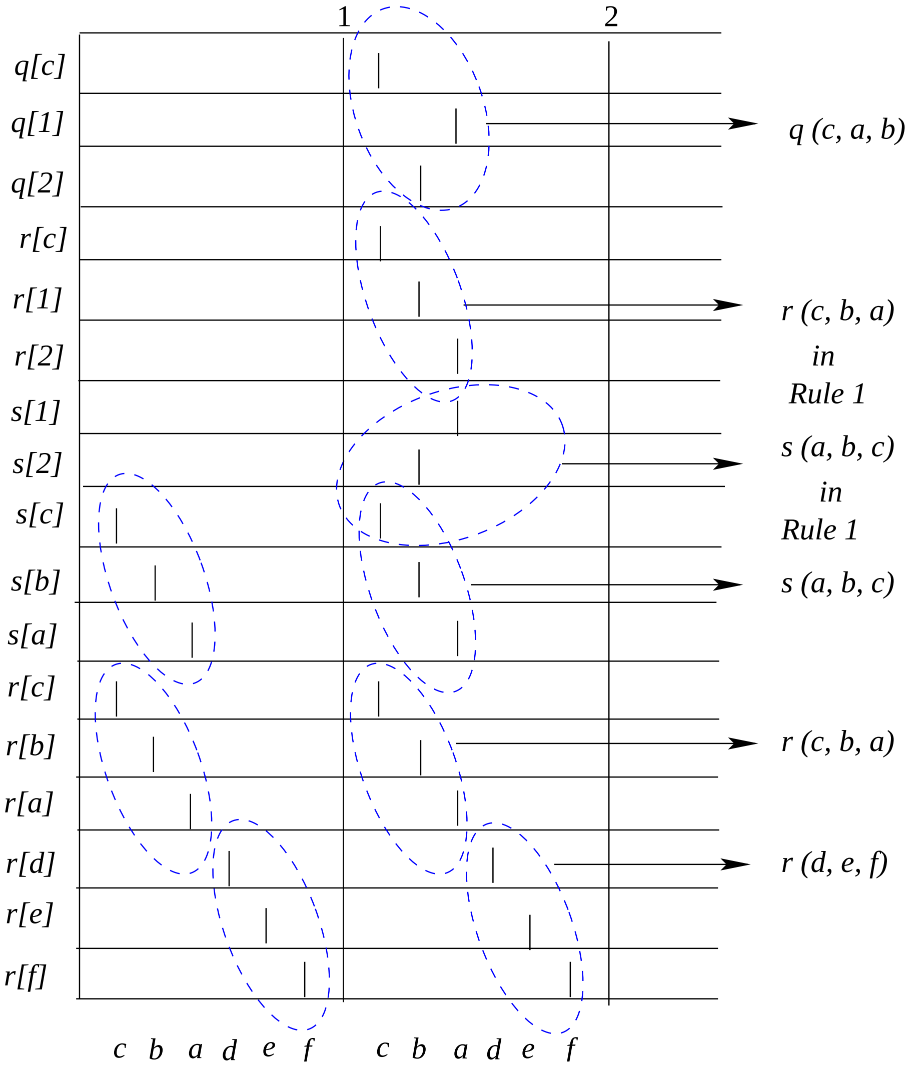


Figure 3.9: Reading the model

body of *Rule 1* (i.e., variable X) get bound to a. Moreover the second argument of q in the head of *Rule 1* gets bound to a.

- $q[2]$ and $r[1]$ together with $r[b]$, $s[b]$, $s[2]$ and the constant b. This implies that the second argument of r, and the second argument of s in the body of *Rule 1* (i.e., variable Y) get bound to b. Moreover the third argument of q in the head of *Rule 1* gets bound to b.

- the neurons activating the fact assemblies in the first inference cycle remain active in the second one as well.

- Hence at the end of the second cycle, the model of the program can be read out from the timing diagram. It consists of all the predicate assemblies active at the end of that cycle, namely $r(c, b, a)$, $r(d, e, f)$, $s(a, b, c)$, and $q(c, a, b)$.

3.8 Learning Algorithm

The *kernel* obtained from the knowledge based spiking neural network is trained by a learning algorithm. The learning algorithm for training such a kernel should have the feature of training the weights on the links of these networks in response to the spike arrival at the presynaptic and postsynaptic neuron. The learning algorithm that we use is a variant of Hebbian learning introduced in the last chapter. Before we formalize the learning algorithm I would like to state that there exists one interesting structural difference between this interpretation of Hebbian learning in temporal coding and its traditional interpretation: *here the time difference between a presynaptic firing and a postsynaptic firing provides a synapse with the full information about the correlation between the pre- and postsynaptic neuron in temporal coding, no matter whether both neurons give high or both give low output values.* However in the traditional interpretation of Hebbian learning in terms of firing rates the efficacy of a synapse is only increased if both neurons give high output values (in frequency coding).

Hebbian learning is often described as an adaptation rule where a synapse is strengthened if pre- and postsynaptic neuron are 'simultaneously' active. In the context of pulse coding the meaning of simultaneously must be specified. In the experiments that were carried out with the barn owl auditory system by Wulfram Gerstner and his group [123], the pre- and postsynaptic neurons were considered to fire simultaneously if the presynaptic neuron fired 0.05 milliseconds before the postsynaptic neuron. For the experimental set up for reasoning system I had taken the simultaneity constraint to be valid if the presynaptic firing took place 0.15 milliseconds before the postsynaptic firing. Whether or not this value has any relation to the processing time in the brain, is not the main concern at the moment, since the idea was to verify whether such a learning scheme could be useful for reasoning or not. So infact the choice to consider the firing of two neurons as coincidental when the time gap is 0.15 milliseconds was quite arbitrary in my experimental setup.

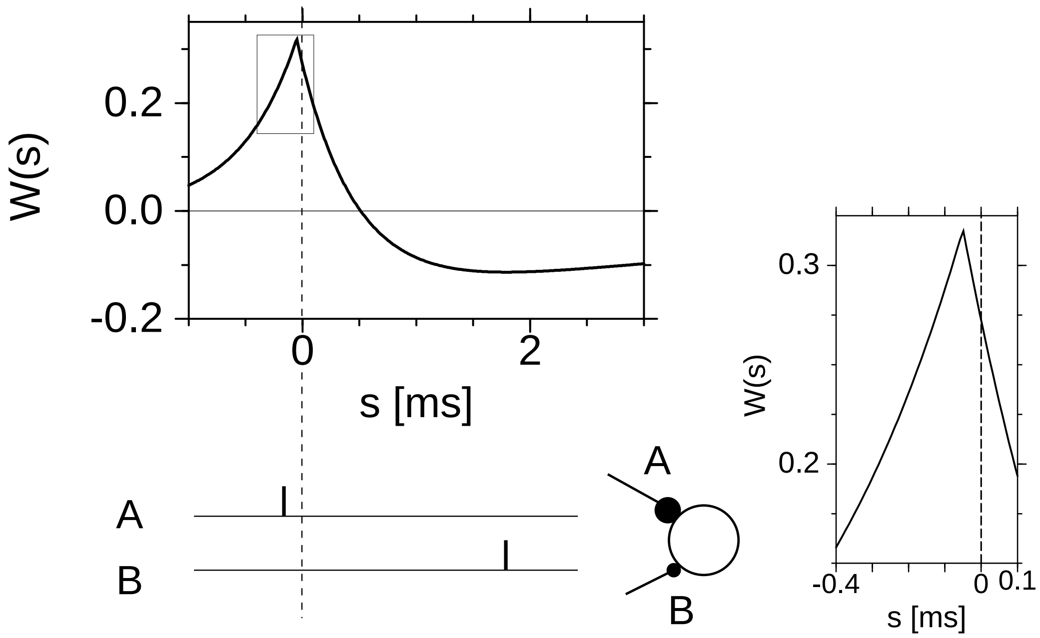


Figure 3.10: Learning Window W as a function of the delay s between postsynaptic firing and presynaptic spike arrival. The graph on the righthand side shows the boxed region around the maximum on an expanded scale. If $W(s)$ is positive (negative) for some s, the synaptic efficacy is increased (decreased). The postsynaptic firing occurs at $s = 0$ (vertical dashed line). Learning is most efficient if presynaptic spikes arrive shortly before the postsynaptic neuron starts firing, as in synapse A. Another synapse B, which fires after the postsynaptic spike, is decreased. Taken from [123]

The weight update function $W(s)$ that we have used is the one studied first by Wulfram Gerstner in [123]. Here s is the time difference between the arrival time of a presynaptic spike and the postsynaptic firing. The function is defined as below

$$W(s) = \begin{cases} (A_+ - A_-)\, exp(-(s^* - s)/\tau_{syn}) & \text{for } s < s^* \\ (A_+\, exp(-(s - s^*)/\tau_+) - A_-\, exp(-(s - s*)/\tau_-) & \text{for } s > s^* \end{cases} \tag{3.4}$$

where $s = t_j^f - t_i^f$ is the delay between presynaptic spike arrival and postsynaptic firing. The parameters that were used in the experimental setup for the barn owl auditory system were $A_+ = 0.5$, $A_- = 0.2$, $\tau_{syn} = 0.5ms$, $\tau_{plus} = 0.5ms$, $\tau_- = 5ms$, and $s^* = -0.05ms$. The learning window is plotted in figure 3.10, and describes the following effects. If a presynaptic spike arrives shortly before the postsynaptic firing, the synaptic efficacy w_{ij} between a pair of neuron i and j is increased ($W(s) > 0$ for $s < 0$). If a presynaptic spike arrives a few ms after the output spike, the synapse is weakened. ($W(s) < 0$). It was reported in [124] that qualitatively the form of the learning window was in accordance with the experimental results on cortical neurons [39, 76]. Time constants in the cortex are however, much longer than those chosen in Figure 3.10, which was intended for neurons in the auditory pathway. In the cortex or the hippo-campus, the learning window probably has a width of $50 - 200$ ms. However, for my experimental setup, I decided to keep the same values as the one used in the case of barn owl auditory system. One should observe from the Figure 3.10 that the maximum of the learning window $W(s)$ occurs for a value $s = s^* < 0$. Thus the synaptic increase is maximal if the input spike precedes the output spike by an amount $|s^*|$. Stable learning requires that s^* be negative and of the order of the rise time of an EPSP [123]. In accordance with Hebb's idea, the stability requirement ensures that those synapses which are already strong and contribute most to the triggering a spike receive the maximum reinforcement. The output spike occurs slightly delayed with respect to presynaptic spike arrival and hence $s^* < 0$ must hold. For an in-depth coverage of the spike based Hebbian learning one should refer to [124]. Here I would present a short review of the exposition given by Wulfram Gerstner et al. in [124], for the choice of the weight update function $W(s)$.

On a microscopic basis, Hebbian learning is thought to involve (at least) two components. Rather than speculating on the chemical nature of the components, we just call them a and b. We assume that the first component is generated by a chemical reaction chain triggered by the arrival of spike at the synapse. As a result of this processing chain the concentration $[a]$ of component a is after a delay Δ increased. In the absence of further input, the concentration decays with a time constant τ_a back to its resting level

$[a] = 0$. A simple way to describe this process is

$$\frac{d}{dt}[a] = -\frac{[a]}{\tau_a} + \sum_{t_j{}^f \in \mathcal{F}_j} \delta(t - t_j{}^f - \Delta) \tag{3.5}$$

A high level of $[a]$ sets the synapse in a state 'ready to learn'. To generate the synaptic change and perform learning, another substance b is needed as well. The production of b is generated by a second process triggered by a postsynaptic spike.

$$\frac{d}{dt}[b] = -\frac{[b]}{\tau_b} + \sum_{t_i{}^f \in \mathcal{F}_i} \delta(t - t_i{}^f) \tag{3.6}$$

where τ_b is another time constant. In analogy to (3.5) we could allow for a delay Δ_b in the dynamics of (3.6), but we will not do so.

Hebbian learning needs both substances to be present at the same time, thus

$$\frac{d}{dt} w_{ij}^{corr} = \gamma\,[a(t)][b(t)] \tag{3.7}$$

with some learning rate γ. The upper index $corr$ is intended to remind the reader that we are concerned with the Hebbian correlation term only. Consider the synaptic change caused by a single presynaptic spike at t_j^f and a postsynaptic spike at $t_i^f = t_j^f - s$. Integration of (3.5) and (3.6) yields

$$[a] = exp(-(t - t_j^f - \Delta)/\tau_a)\,\mathcal{H}(t - t_j^f - \Delta) \tag{3.8}$$

$$[b] = exp(-(t - t_i^f)/\tau_a)\,\mathcal{H}(t - t_i^f) \tag{3.9}$$

where $\mathcal{H}(.)$ denotes the usual Heaviside step function. The change caused by the pair of pulses $(t_j^f,\, t_i^f)$ and measured after a time $\mathcal{T}$ is

$$\Delta w_{ij}^{corr}(t_i^f, t_j^f) = \int_0^{\mathcal{T}} \left(\frac{d}{dt} w_{ij}^{corr} \right) dt \tag{3.10}$$

$$\Delta w_{ij}^{corr}(t_i^f, t_j^f) = \gamma \int_{t_0}^{\mathcal{T}} exp\left[-\frac{t - t_j^f - \Delta}{\tau_a} - \frac{t - t_j^f + s}{\tau_b} \right] dt \tag{3.11}$$

where $t_0 = max(t_j^f + \Delta, t_j^f - s)$ and $s = t_j^f - t_i^f$. The integral in (3.11) is solved and we introduce $s* = -\Delta$ and find for $\mathcal{T} \gg \tau_a, \tau_b, \Delta$

$$\Delta w_{ij}^{corr}(t_i^f, t_j^f) = \begin{cases} A\,exp(-(s^* - s)/\tau_a) & \text{for } s < s^* \\ A\,exp(-(s - s^*)/\tau_b) & \text{for } s > s^* \end{cases} \tag{3.12}$$

with $A = \gamma(\tau_a\,\tau_b/\tau_a + \tau_b)$. Equation (3.12) shows that the change of the synaptic efficacy Δ_{ij}^{corr} caused by a pair of pulses (t_i^f, t_j^f) depends only on the time difference $s = t_j^f - t_i^f$. We write

$$\Delta w_{ij}^{corr}(t_j^f, t_i^f) = W(t_j^f - t_i^f) \tag{3.13}$$

Equation (3.13) gives the change caused by a single pair of spikes. The full weight update function given in equation (3.4) can be constructed by linear superposition of two elementary windows of the form given in (3.12).

3.9 Experimental Setup

We have reached a point where by the stage for testing the ideas is set. I carried out some experiments to implement the ideas proposed in the above sections. But before I discuss the experimental set up, I recapitulate what has been discussed so far and where we stand. We had multi-fold motives to accomplish. We are working at the cross-section of many inter-related concepts. We wanted to accomplish rule-extraction at the level of first-order logic. In trying to understand how it could be done, we had to borrow insights and concepts from diverse areas like machine learning, neuro-biology, logic programming, mathematics and brain and behavioral sciences.

Since neural networks are extremely successful for inductive learning (learning from examples) but are very difficult to understand (since all the information that they encode is in their weights which are real numbers), it is clear that for the purposes of diagnostics and for purposes of a logical characterization of our domain of data set, we need some kind of explicit knowledge representation schema. Some of the well understood knowledge representation schemes are propositional logic and first order predicate logic. Expressing explicit knowledge by using propositional and/or predicate logic seems to be a rational step, however when we are not sure about the existence of some knowledge, we cannot possibly express it explicitly. This means that we can suspect to have more knowledge but we have no evidence to write it explicitly. Moreover, even if we can express explicit knowledge by writing out propositional or predicate logic rules, sometimes it may be very difficult to make sure whether or not it is correct.

It has been widely studied within the realms of psychology that humans do learning by two methods, from rules and from examples. Confidence in the rules is generated by checking out the instances where that rule could be supported by some examples. This goads us to rethink the way we should do the task of machine learning. This means that we should integrate the best of both the learning schemes (inductive and analytical look e.g in [77]). Integration of propositional knowledge and standard sigmoidal feed-forward neural networks has been found to be very successful [20, 7]. However, the bottle neck arising from the propositional knowledge representation is clear. We can't reason about general rules, and so we have to use the expressive power of first-order logic. This leads us to the point where we would like to do rule-extraction at the level of first-order logic. The expressive power of first order logic doesn't come to us for free, the price that we have to pay for integrating it with learning models of neural networks, is substantial. We need to somehow model the binding process of first order logic in the neural networks. It was demonstrated by Shastri and Ajjanagadde in [111] that one could use the idea of time varying input and output to encode dynamic representation of rules. His claim was supported by recent neuro-biological findings where it was asserted that certain regions of the brain become active at the same time on receiving an input from the senses. This was especially the case with the human visual cortex. This implied that the synchronous or near synchronous firing of some neurons could denote the existence of some structure, while the firing of these same neurons at some other time points could signal the presence of some other event. This lead to the idea of implementing dynamic variable binding in neural networks by neurons which are sensitive to time. However, this idea of having temporal units in neural network was not fully implemented within the mathematical framework of spiking neural networks. Obviously it had an impact on learning scheme. The networks presented in SHRUTI or in BUR calculus were indeed encoding a small subset of first order logic, but it was not clear how we could do learning in those networks.

I proposed a new transformation algorithm in the beginning of this chapter where we can encode a small subset of first order logic programs into neural networks. We encode not only the knowledge we know, but we also keep the possibility open to learn some more rules than we know in the beginning. This reflects in the full connection in the part of the knowledge based spiking network. The neural networks that encode this knowledge are not the conventional sigmoidal unit, feed-forward networks but are the spiking neural networks. This meant that we will use the learning dynamics of the spiking networks to do rule learning. This was done by the *trans* function that was explained in the beginning of this chapter. Taking a positive clue from the recent neuro-biological experiments carried out by Wulfram Gerstner and his group [123], I propose to use a similar model (Hebbian type) for learning in our knowledge based spiking networks. We use the same weight update function that Gerstner and his group used.

One of the problems that were encountered during experimental testing was the existence of the data set. As first steps to test the ideas, I created some sample data sets and built in some noise. The amount of correct examples and the wrong ones were modulated by hand.

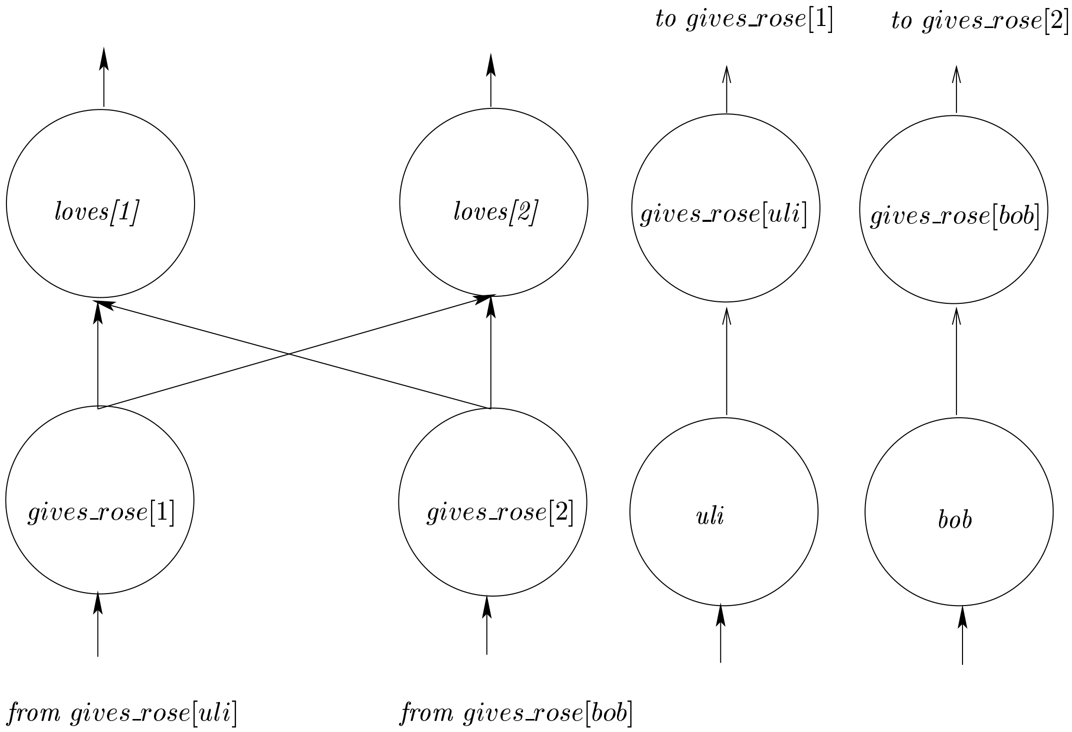


Figure 3.11: Applying the Transformation Algorithm

Consider the following logic program *Love*:

Example III
Rule 1: $loves(X, Y) \leftarrow gives\_rose(X, Y)$
Rule 2: $loves(Y, X) \leftarrow gives\_rose(X, Y)$
Rule 3: $gives\_rose(uli, bob)$.

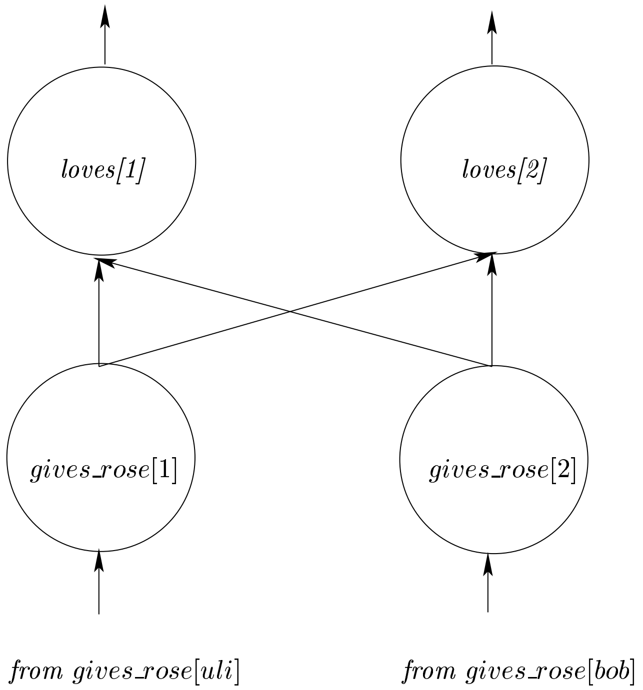


Figure 3.12: The kernel

If we apply the *trans* function to this program, it will produce the network shown in Figure 3.11 . The kernel is easily identifiable and is shown in Figure 3.12. If we compute the model of the above program by using the least-fix point operator $\mathcal{T}_P$, we will get
$\mathcal{T}_P = \{gives\_rose(uli, bob),\ loves(uli, bob),\ loves(bob, uli)\}$.

The same model can be computed from the timing diagram as shown in Figure 3.13. Intuitively we all understand since it appeals to our commonsense, that if X gives rose to Y, then X loves Y. It is clear that it would be incorrect to conclude that if X gives rose to Y then Y loves X. But consider for a while the above logic program, and we have *Rule 2* and *Rule 3* above. This example has been chosen to show that one can find out the correctness of the rules by learning only the correct ones and not the wrong

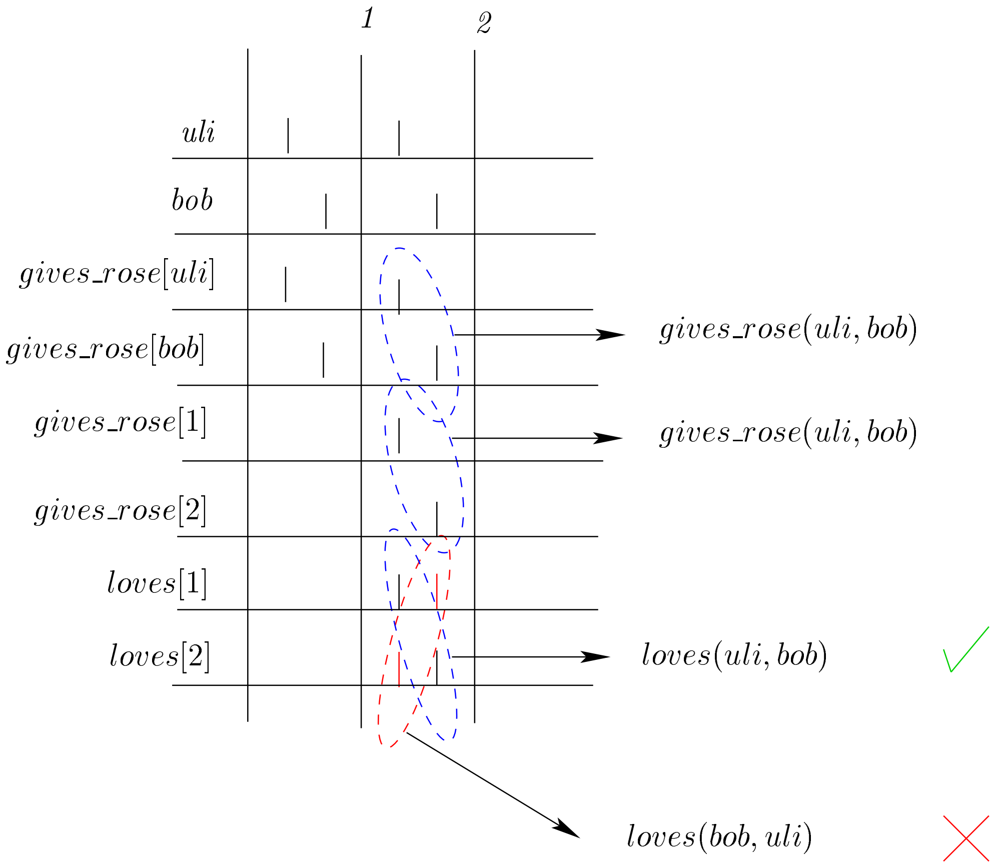


Figure 3.13: Reading the incorrect model. $loves(uli, bob)$ present but also $loves(bob, uli)$ present

ones, where wrong in the context of our program above is a rule that goes against the common-sense of almost all of us. Now, given such a program, if we have examples in the data file which corroborate the existence of a rule, we will at the end conclude the presence of that rule. This means that if the data set contains enough wrong representative examples then at the end of the training the statistical bias would tilt in the favour of the wrong rule. However given a scenario where we have a mixed representation of examples both in the favour of wrong ones and in the favour of right ones, then depending upon the amount of right statistical bias, we can discover the correct rule. In the experiments that I carried out for such a program, I hand crafted a small data set with 21 explicitly represented patterns. Let us

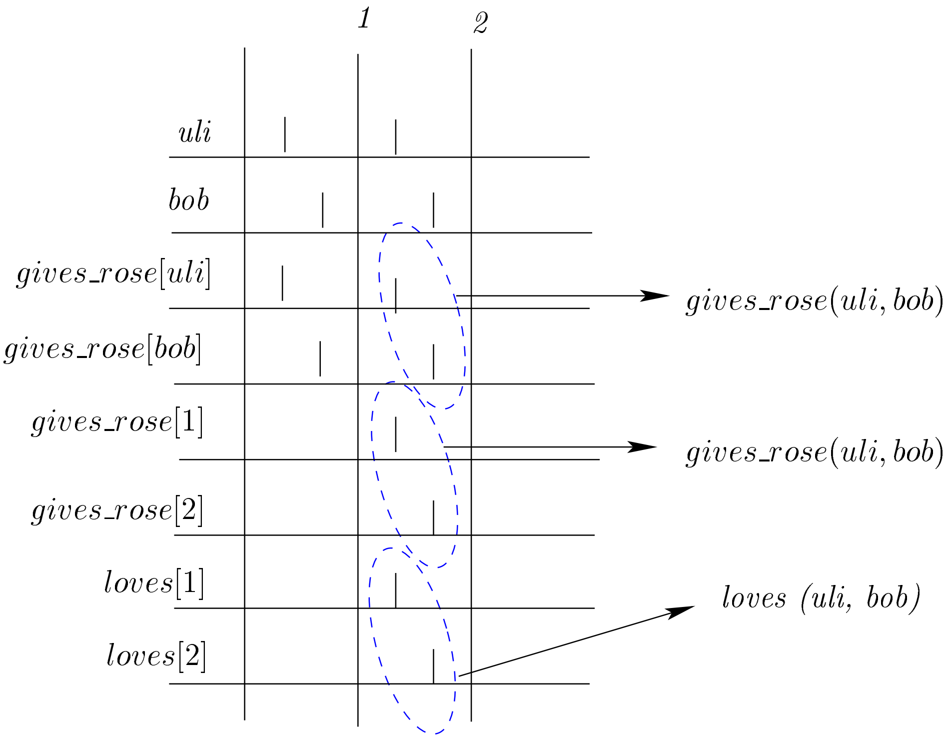


Figure 3.14: Reading the correct model

see what training in the sense of logic programming means.

Let us consider a scenario where we have enough distinct constants in our program, and hence we have the possibility to ground the rules in the given program by each of these constants. By instantiating the rules with constants we can create ground instances for the predicates *gives_rose* and *loves*. We can then build a set of examples where by we can relate the grounded instances of the predicates *loves* and *gives_rose*. Let us look at such an example data set:

Sample Data Patterns for the logic program

$\langle loves(adam, yve),\ gives\_rose(adam, yve)\rangle$
$\langle loves(romeo, juliet),\ gives\_rose(romeo, juliet)\rangle$
$\langle loves(badal, rekha),\ gives\_rose(badal, rekha)\rangle$
$\langle loves(mary, alex),\ gives\_rose(alex, mary)\rangle$
$\langle loves(janet, sebastian),\ gives\_rose(sebastian, janet)\rangle$
$\langle loves(rama, yve),\ gives\_rose(rama, yve)\rangle$
and so on...

Spiking Neural Network consists of neurons sensitive to spikes arriving at distinct time points. Let us allot each constant in our program a distinct time-point e.g., the mapping below:

Mapping constants to distinct time-points
$adam \mapsto 1.0$
$yve \mapsto 2.0$
$romeo \mapsto 3.0$
$juliet \mapsto 4.0$
and so on...

More precisely this mapping ensures that neurons representing the arguments of the predicate *gives_rose* namely *gives_rose*[1] and *gives_rose*[2] in figure 3.11, would be instantiated by adam and yve at times 1.0 and 2.0. This means that once our cycle of training starts, then each syntactically distinct constant in the logic program gets a distinct phase of time, and this constant when presented to a neuron will activate the neuron at that phase of time. Presenting a training example of the kind: $\langle loves(adam, yve)$, $gives\_rose(adam, yve)$ means that the neuron corresponding to *gives_rose*[1] and *gives_rose*[2] become active, at time 1.0 and 2.0 respectively. This in turn will activate the neurons *loves*[1] and *loves*[2] at time points close enough to 1.0 and 2.0 so that the activation of the concept $loves(adam, yve)$ could be assumed to have followed as a consequence of adam giving rose to yve. As we had observed that there is a certain time delay between the time a pre-synaptic firing occurs and the time the post-synaptic firing takes place. Hence we have to incorporate a small time delay between the firing of a pair of neurons in the input and the output layer. The limit that we have set in order to consider that the two time points are close enough to be considered as firing at the same time is 0.1-0.2. If for example *loves*[1] and *loves*[2] fire at 1.1 and 2.1 then they could be considered to be representing identical objects – *gives_rose*[1] and *gives_rose*[2] respectively. This in turn would mean that the conclusion of adam loving yve would follow immediately from the fact that adam gave a rose to yve. Any two neurons if they are firing at a delay of more than the above specified limit would be considered as signalling syntactically distinct entities of our reasoning domain. So for example if *gives_rose*[1] fired at 1.0 and *loves*[1] fired at 1.5 or 2.1 then it means that the firing of the output neuron did not take place soon after (i.e., within the next 0.1-0.2 ms) and hence the output neuron represents some other time point (constant) and not the one that activated the input neuron. The Appendix A shows the full data sets, training results and weight values for this program.

It could also be the case that the output neuron may fire shortly before the input, in this case this would correspond to the presence of a wrong example as well, since the output neuron should be active *shortly after* the input one and not the other way around. We have seen in the learning algorithm (weight update function) above that the weight on the link connecting an input-output pair would be weakened if the output neuron spiked before the input. The output when spikes after the input but not *shortly after* to be considered as being representing the same entity, still strengthens the weight positively, however the change is small enough to make any significant contribution to the weight increase. This is happening because of the way Hebbian learning takes place. Ideally speaking if the output neuron fired at 1.7 and the input had fired at 1.0 then even such a pair is an exemplary of a wrong pattern, and so the weight should not be increased, but Hebbian learning has the feature whereby it allows for such an increase, and infact this is because of the stability of the learning algorithm. However as we can see in Appendix A, the increase is almost close to zero, because of the *exponential* nature of the weight update function.

The notion of time here does not imply a presence of a global clock, instead it means that given a cycle of training inference, the neurons in the input layer — $gives\_rose[1]$ and $gives\_rose[2]$ became active at time 1.0 and 2.0 when presented with an example $gives\_rose(adam, yve)$, and then the same $gives\_rose[1]$ and $gives\_rose[2]$ neurons became active at 3.0 and 4.0 when the network was presented with the next training example $gives\_rose(romeo, juliet)$. Depending upon whether or not the first argument of *gives_rose* predicate was the same as the first argument of *loves* predicate and the second argument of *gives_rose* was the same as the second argument of *loves*, the idea of an example being correct or not is generated.

3.10 Discussion

The first experiments that we carried out gave us an impression that it is possible to perform first-order rule learning through the dynamics of the neural networks. The experimental results provide us an evidence that even with a small subset of first-order logic it is possible to understand how we could do rule-learning through connectionist systems. The results have multifold consequences in the context of rule-extraction, machine learning and brain and behavioural sciences.

3.10.1 Rule Extraction

In the context of rule-extraction the above experimental results would imply that we can extract first-order logic rules from a neural network. It is clear that this is possible if the first-order logic program has the restrictions mentioned in the Section 3.4.2, and that the neural network should have been constructed from the given domain theory or a first-order logic program.

3.10.2 Machine Learning

In the context of machine learning our results show that we can learn first-order rules simply by the dynamics of spiking neural network. Our experiments are based at present on hand crafted data sets and are not tested exhaustively on any real-domain data. It would be too early to predict any far reaching consequences of our solution in the light of neural symbolic integration. However given more time, we can test our setup on data from molecular biology or any similar domain. Assuming we get promising results, it would imply a very powerful way of doing first-order rule learning. It would be interesting then to compare our solution to several successful inductive logic programming algorithms.

Further in the context of machine learning we have to understand an important feature of our solution which is discovering rules. By fully connecting the network between the input and the output layers we leave open possiblities to discover rules for which there is no evidence in beginning. Later during the learning cycle, based on the amount of supporting data, we can discover a new rule. This has a direct impact on the area of computational molecular biology, where initially we may not know precisely the exact rule describing certain properties of for example *genes* or *chromosomes*, but given the right *quantity* of statistical bias supporting our rule we could at the end of the training discover a new rule characterizing the relation between certain known properties of genes or chromosomes. However due to lack of time, we could not test empirically our solution on data sets from the aforesaid domains. To illustrate the point however, let us consider an example domain theory *Gene*:

Example
Rule 1: $property_1(X, Y, Z) \leftarrow property_2(X, Z) \wedge property_3(Y, Z)$

Initially we only know that X, Y and Z are related to each other via *property*1 if X and Z are related by *property*2 and Y and Z are related by *property*3. Given this rule, by using our algorithm of transforming the program to a network, we will in the beginning make connections from neurons in the output layer representing X, Y and Z connections to each of the neurons representing X, Y and Z in the input layer. During training if we have good amount of examples favouring not only the above rule but also the following kind of rules:

Set of examples seen during training:
Rule 2: $property_1(c, a, b) \leftarrow property_2(a, c) \wedge property_3(b, c)$
Rule 3: $property_1(f, d, e) \leftarrow property_2(e, d) \wedge property_3(f, d)$

then we will at the end of learning and thresholding, extract the rules of the kind:

Rule 2: $property_1(Z, X, Y) \leftarrow property_2(X, Z) \wedge property_3(Y, Z)$
Rule 3: $property_1(Y, Z, X) \leftarrow property_2(X, Z) \wedge property_3(Y, Z)$

An expert in the domain of molecular biology may not be so sure about the existence of some of the above rules, but by using our solution we can discover initially unknown rules.

3.10.3 Brain and Behavioural Sciences

In the context of brain and behavioural sciences, our solution gives an insight as to how humans could possibly encode and reason with structured knowledge. Our model is too simple to explain in a reasonably complete way how humans could do reasoning. However our model provides useful insights as to how simple common-sense reasoning and learning could take place in our brains. We have not taken the rate models into account in our simplified model and we have instead used the idea of precise spike timing, taking clue from some recent neuro-biological findings where it was believed that the *neurons have full information from the pre- and postsynaptic firing.* However another recent study shows that depending upon the frequency of signal, humans could learn by precise spike timing or by rate models. If the frequency of signal is below 100Hz, then spike timing conveys more information and when the frequency of signal is greater than 100Hz the rate of firing could convey more information. In this light, if we want a model which could explain several aspects of reasoning, then we need to design several inter-related concepts into one, and we should investigate that model. It is however so hard to corroborate evidence from the neuro-biological experiments in order to fully explain how humans could do reasoning, because of practical limitations. There are results as we have seen in Chapter 2 that show that learning in human visual cortex and cortical activity in general could be related to the spike timing, than the rate of firing. In my opinion timing of spike will play a genuine role during fast common-sense reasoning and the rate of firing may also play a part but may not be so much in fast reasoning or reflexive reasoning.

3.10.4 Hebbian Learning in Reasoning Systems

It has been suggested that medium-term memories are encoded via long-term potentiation (LTP) [33] and some of them subsequently converted into long-term memories and encoded via essentially permanent structural changes (see e.g [16, 99, 100]). This has a relevance to the kind of learning algorithm we have used in our reasoning system — namely Hebbian type, see ection 2.7.1. There it was explained that the “hidden parameters” that regulate the dynamics of a biological synapse vary from synapse to synapse and that at least some of the hidden parameters can be changed through LTP (long-term potentiation). Wolfgang Maass and Anthony M. Zador in [75] present some interesting biological data on different kinds of synaptic plasticities. One of them — the LTP, they suggest depends on some conjunction of pre- and postsynaptic activity, and it obeys Hebb’s rule in that its induction requires simultaneous pre- and postsynaptic activation. In the wake of these evidences coming from these two groups, it seems that selecting a Hebbian type learning algorithm for purposes of learning in reasoning systems, is a good step.

3.10.5 Why Spiking Neural Networks?

The idea of using spiking neural networks rather than the conventional feed-forward sigmoidal unit networks should be seen as a useful concept towards integration of symbolic knowledge into neural networks. It is clear from the theory of spiking neural networks that they not only provide a Turing equivalent computational model but also provide a convenient framework for handling temporal signals. Infact it is clear from Section 2.5.5 that computations with spiking neurons may in priciple also involve complex operations on relations between computational objects, a computational mode which has no parallel in traditional neural network models or any other common computational model. It would be incorrect to say that the conventional sigmoidal unit neural network cannot handle time. In Section 2.3.1 we had seen how time dependence could be induced into the otherwise static activation function. However, the problem of learning precise spike timing in order to achieve dynamic variable binding finds a much easier solution within the framework of spiking networks than sigmoidal unit neural network. Also since any sigmoidal unit neural network can in principle be simulated by a network of spiking neurons, we can in principle map the whole solution of learning in spiking networks to a solution of learning in sigmoidal unit feed-forward neural network. Theorem 3 in Section 2.6 corroborates this. However as we had seen in Section 2.3.1 that an analog variable defined by spike count measure requires a long temporal averaging window. It can therefore be used only if the input and the output change on a slow time scale. But common-sense reflexive reasoning takes place in the order of few hundered milli-seconds so a slowly changing input cannot be assumed.

3.10.6 Who provides the activations?

While computing the least fix point for a logic program, one starts with facts in the program and then instantiates the body of the clauses, to see what can be inferred from the rules in the program, and this way finally the least model of the program is obtained. In our system we have adopted a similar concept. We start the cycle of inference by clamping the fact neurons. This is done by activating the fact arguments, the neurons representing the arguments of the fact in the input layer, and this is turn activates the neurons representing fact predicates in the output layer. So an external activation by the user is given to start the process of forward reasoning.

The activations are passed from the output layer to the input layer by the recursive links, and the process of inferencing starts. The transient ryhthmic activity can be read from the timing diagram. The timing diagram also computes the model for the knowledge base.

3.11 Open Questions and Future Work

Inspite of our successful attempt to design a model which could learn first-order rules within the neuro-symbolic paradigm, there are a couple of issues that need to be resolved and some open questions to be answered. It is imperative to introspect some of the restrictions that we imposed on the class of logic programs covered or the class of first-order rules that can be learnt by our model. Do we really need to have all the restrictions mentioned in Section 3.4, or can we waive some of them? The following paragraphs tend to answer this.

3.11.1 Restriction on the Arguments

It may not be necessary that all variables that occur in the head must occur in the body. By softening this constraint, we can allow variables in the head of the clause, that don't occur in the body. When we transform such a clause, we shall have a neuron in the output layer representing the new variable that is not existing in the body. Now since the body does not have this variable we can't have a neuron for such variable in the input layer. With which neuron in the input layer should we connect the neuron representing this new variable in the output layer? May be we should leave this neuron unconnected, signifying that it has no relation to any neuron representing variable or constant in the body of the clause. This creates a difficult situation from the rule-extraction point of view. Then how should a rule extraction module know where this new neuron comes from, and even if it knows that there is a new neuron representing a new variable present only in the head, at which position should this variable be in the output predicate assembly. It was not very clear how we should handle such cases.

Let us see the other restriction: "all variables that occur in the body must occur in the head if there is more than one predicate in the body". If the number of predicates in the body is only 1 (clause with only one antecedent) or 0 (i.e. fact) this restriction is no more needed. If there is more than one predicate in the body and the variable that occurs in more than one predicate in the body is not in the head, then we cannot deal with this case in our solution. If the variable that does not occur in the head of the clause, occurs only once in a predicate in the body then we can handle this case. Let us see why we need this restriction. Consider the axiom of transitivity where this restriction gets violated.

Axiom of Transitivity: $reach(X, Z) \leftarrow reach(X, Y) \wedge reach(Y, Z)$

We can create a dummy neuron in the output layer for the variable Y, however how should we create the links from the input to the output side. May be we can still make the links from the Y neurons in the input layer to the Y in the output, but we don't interpret the Y to be active at the output. Again, this solution could be provisional, but it makes the task of rule extractiom more complicated. Moreover, how should the rule extraction system know that a new variable that appears in the body in two predicates is the same.

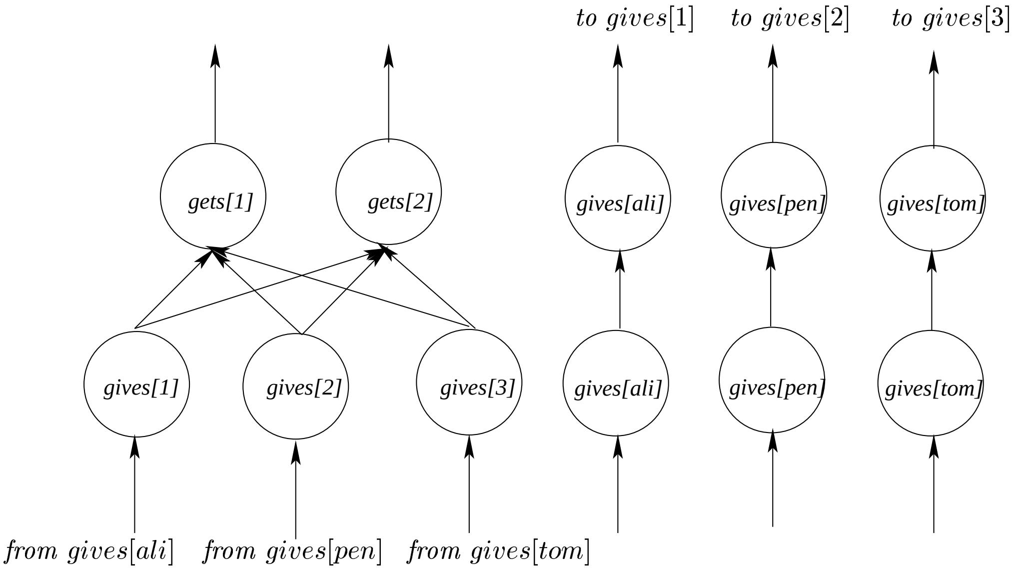


Figure 3.15: Before training the network

A better solution to handle the restriction on transitivity above could be to have a hidden layer in between the input and the output layer, and place a neuron Y there, and the output neurons X and Z should not only learn to fire in exact correspondence to X and Z in the input layer, but should also relate to the presence of Y at the input and hidden layer. This means that we can have the classical spiking neuron model with temporal weight coding as explained in Chapter 2, Section 5.3. The output neurons X and Z could sum up the weighted input coming from Y in the hidden layer and X and Z in the output layer should fire as soon as the corresponding X and Z fire in the

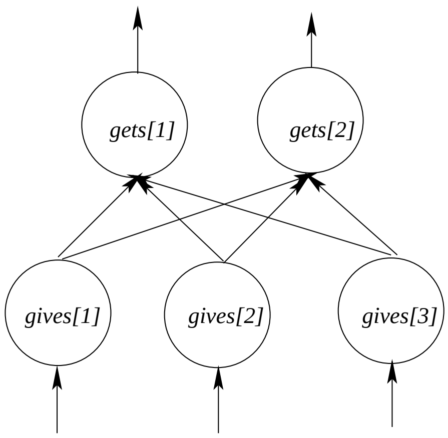


Figure 3.16: The kernel

input layer, here the additional information in the form of firing threshold comes from the presence of Y in the hidden layer. Here learning takes into account the slow neuronal dynamics and the fast correlation type behaviour of neuronal firing. It needs to be mentioned that these solutions are far from being tested, and they are only some of the first intuitive insights.

So why does our solution works for the case when a variable occurring only once in one predicate in the body does not occur in the head. Let us consider the program *Transaction* we first saw in the first chapter, which illustrates the fact that the variable that was absent from the head occurred only once in one predicate.

Example IV
Rule 1: $gets(Z, Y) \leftarrow gives(X, Y, Z)$
Rule 2: $gets(X, Y) \leftarrow gives(X, Y, Z)$
Rule 3: $gives(ali, pen, tom)$.

By transforming the above program, we will have the network shown in Figure 3.15. The kernel is shown in Figure 3.16. During the training if we have enough examples of the kind:
$\langle gets(muna, book), gives(andreas, book, muna)\rangle$
$\langle gets(thomas, pizza), gives(prakash, pizza, thomas)\rangle$
then at the end, this would imply that we will learn *Rule 1.* However if we have examples of the kind:
$\langle gets(andreas, book), gives(andreas, book, muna)\rangle$

$\langle gets(prakash, pizza), gives(prakash, pizza, thomas)\rangle$
then we will end up learning *Rule 2*.

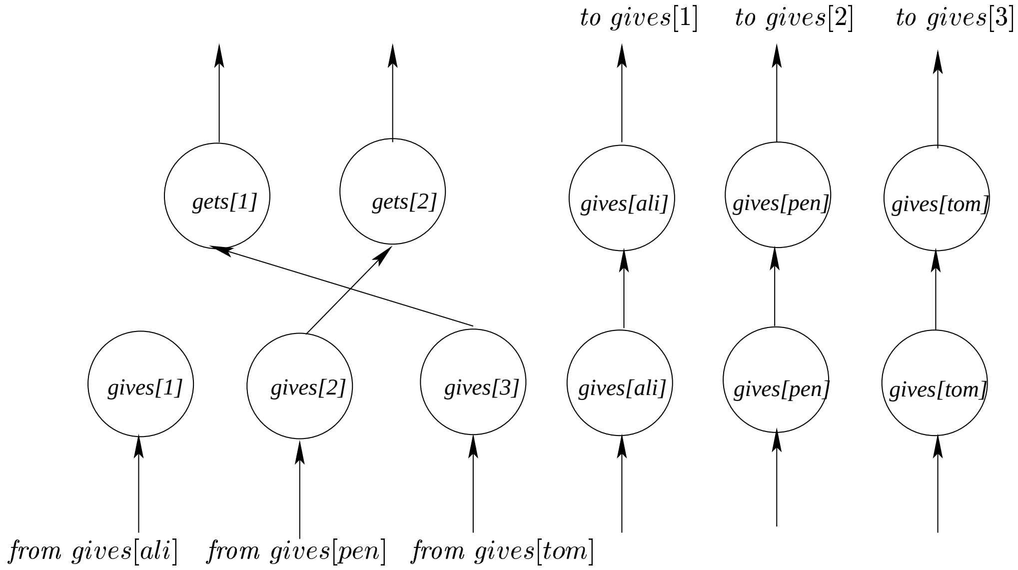


Figure 3.17: After training the network

Assuming we have enough examples of the first kind so that the final statistical bias is tilted in the favour of *Rule 1*, we will get the final trained network shown in 3.17 and we can read the correct models from the timing diagram shown in Figure 3.18.

3.11.2 Bound on the Number of facts

The next restriction that we will not deal with more than one instantiation of a predicate in the head of a clause built over identical predicate name and overlapping set of constants, is very much needed. If we don't have such a restriction, we will get the famous cross-talk prblem mentioned in [102]. As an example let us consider the following program *Cross-Talk*:

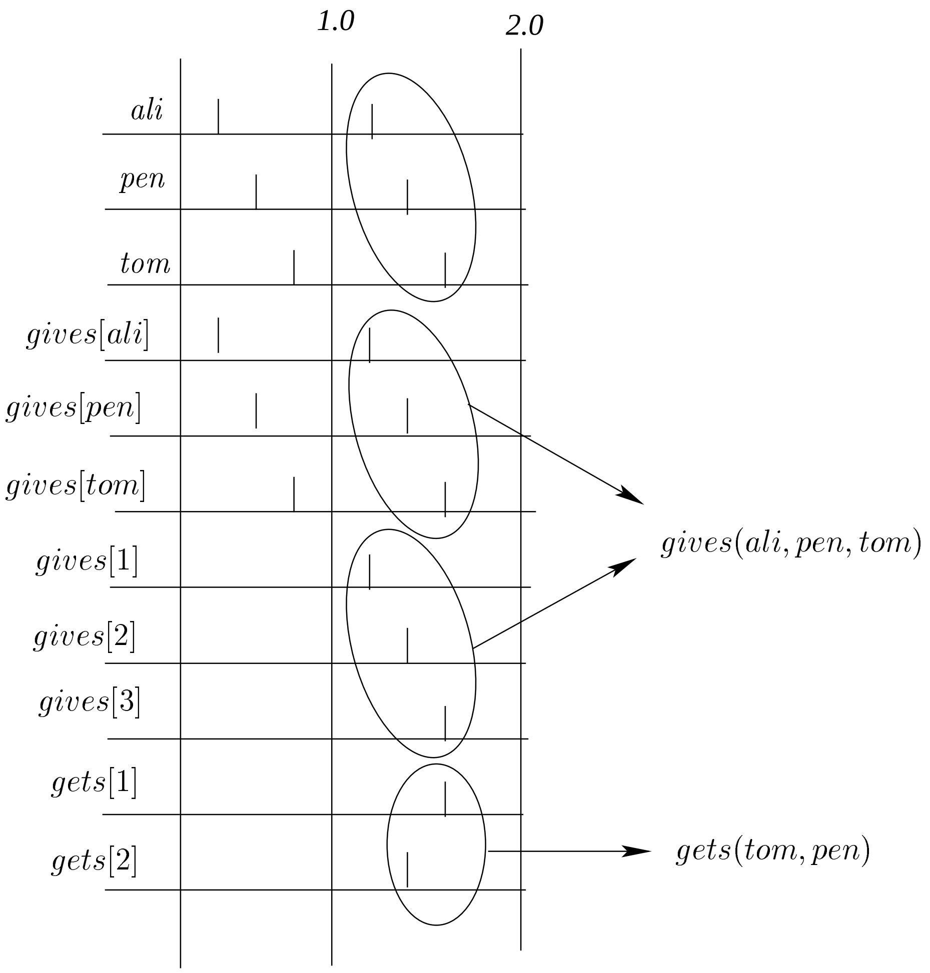
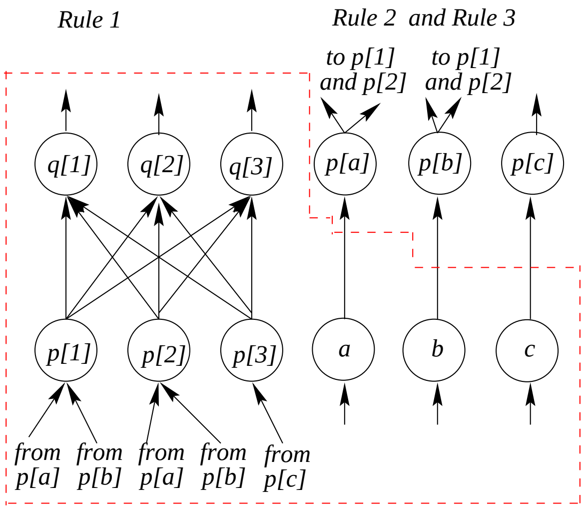


Figure 3.18: Reading the model

Example V:
Rule 1: $q(X, Y, Z) \leftarrow p(X, Y, Z)$
Rule 2: $p(a, b, c)$.
Rule 3: $p(b, a, c)$.

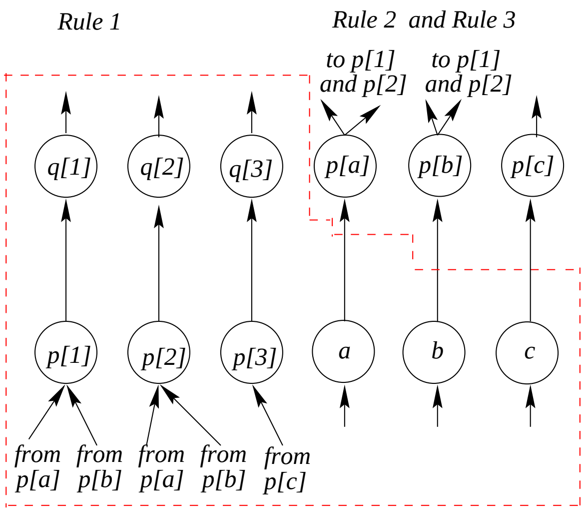
The above program contains two facts over the same name p and the predicate assembly is built over the overlapping set of constants $\{a, b, c\}$. If we construct an equivalent spiking network that encodes this program, we



Case: When the program does not fulfill the restriction no. 2

Figure 3.19: Before training the network

will get the one shown in Figure 3.19. If we train the kernel on a set of training data, then let us assume after the training the network looks like in Figure 3.20.



Case: When the program does not fulfill the restriction no. 2

Figure 3.20: After training the network

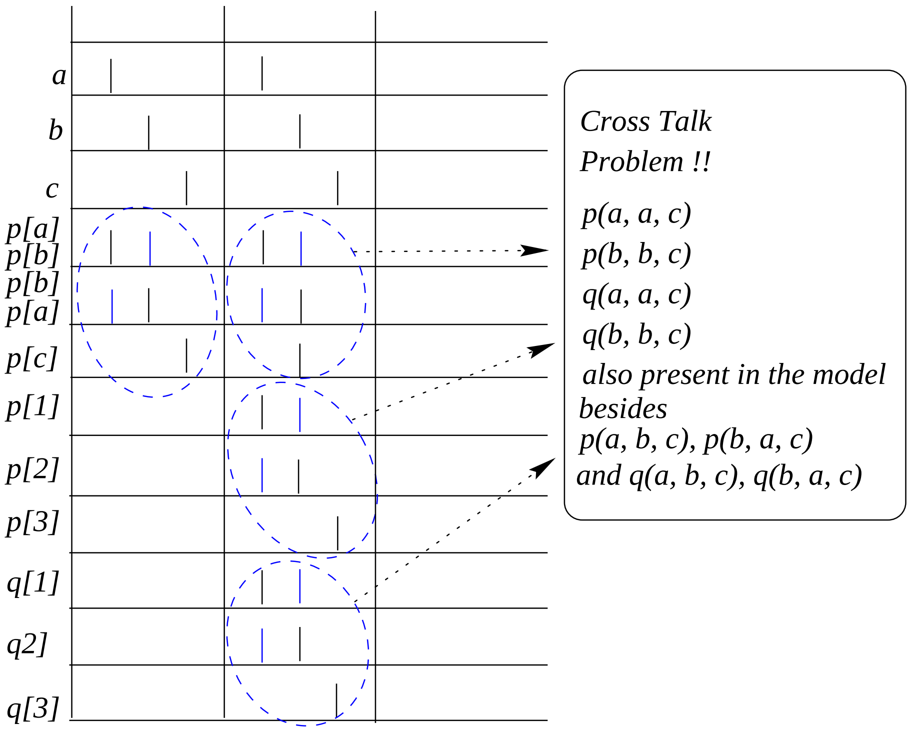


Figure 3.21: Reading the model

If we now read the model from the timing diagram (see Figure 3.21) for such a network, we will have not only predicates $p(a, b, c)$, $p(b, a, c)$, $q(a, b, c)$ and $q(b, a, c)$ but also many more like $p(a, a, c)$, $p(b, b, c)$, $q(a, a, c)$, $q(b, b, c)$. There is no way that we cannot misread the redundant predicates from such a timing diagram. This is the problem of cross-talk. In order to avoid this we need to have the limit on the number of grounded instances that we can have over the identical predicate name and overlapping set of constants in the head of a clause, to one.

3.11.3 Why multiple occurrence of an argument within a predicate prohibitive?

It is still not clear how we can check the binding propagation when there are repeated variables or constants in the head or in the body predicates, and so we decided to have the restriction that there should not be more than one occurrence of a variable or a constant within the same predicate name. This problem relates to checking consistency of bindings as they propagate from the body to the head and sometimes from the fact to a body and then to the head. Nam Seog Park in [87] have proposed some solutions to this problem. However, his connectionist systems are quite complex in structure and there was no clue how we could learn such networks. Since our focus from the beginning was to investigate learning in structured connectionist networks, we maintained this restriction. In any case the future work should also focus on this aspect of representing multiple occurrence of arguments within the same predicate name.

3.11.4 Why only constants?

We will deal with constants in our program, and not with unary or higher arity function symbols. This restriction is also needed as currently there are no solutions known where we can handle the representations of such functions in the network.

3.11.5 Why no negation?

We did not dealt with negation in the programs. In the next steps towards improving the proposed solution, we should think about how to handle the negation. Considering that one of the recent propositional logic rule-extraction algorithms in [21] have dealt with negation, we should take some clues from them.

3.11.6 Recursive part is hard-wired

The reader might have noticed that while constructing the recursive part of the network, we have been very cautious and have segregated several cases, and mentioned recipes to avoid certain prohibitive links. This was done in order to keep the scope of the problem tractable within the time frame I had. Ideally, even the recursive part should be fully connected, and the network should learn to find the correct links from the training data.

3.11.7 Relevance of full connection

Although the idea to have all neurons in the output layer (encoding rules) fully connected to the neurons in the input layer (encoding rules) seems to suggest tremendous increase in space requirements [111], we have clinching evidence from the learning results obtained by Wulfram Gerstner and his group [123] on the barn owl auditory system which motivate us to consider this as a step in our solution. In his experimental work [123], Gerstner and his group report that before learning there were 600 synapses (300 from each ear) with different delays, and after learning, only about 150 synapses (≈ 75 from each ear) survive (see Figure 3.22).

3.11.8 Time Delay between presynaptic and postsynaptic firing

Why there is a time delay between the input spike firing and the output spike firing. To keep our proposed model closer to the neuro-biological side, every attempt was made to retain the relevant features of the neuro-biological models we discussed in Chapter 2. This meant that we had to also retain the feature of axonal transmission delay in firing. This is reflected in the data set, where firing time points of the input neurons are temporally shifted from the firing time of the output neuron in most of the cases (see Appendix A) . Also because we wanted to use the same model of Hebbian type learning that Gerstner and his group used, we planned to retain the time delay property also, although from the point of view of the learning window we will still get good results even if we make the output neuron fire at the same time as the input neuron in case they are representing the same object. Another philosophical insight that we have now, is that we can design learning functions better suited to this reasoning application. Having the learning window we used is not at all a problem even for reasoning systems, but of course tuning the statistical parameters like A_+, A_-, τ_+, τ_- and τ_{syn} to several values other than what has been given in [124] should also be done as a next step.

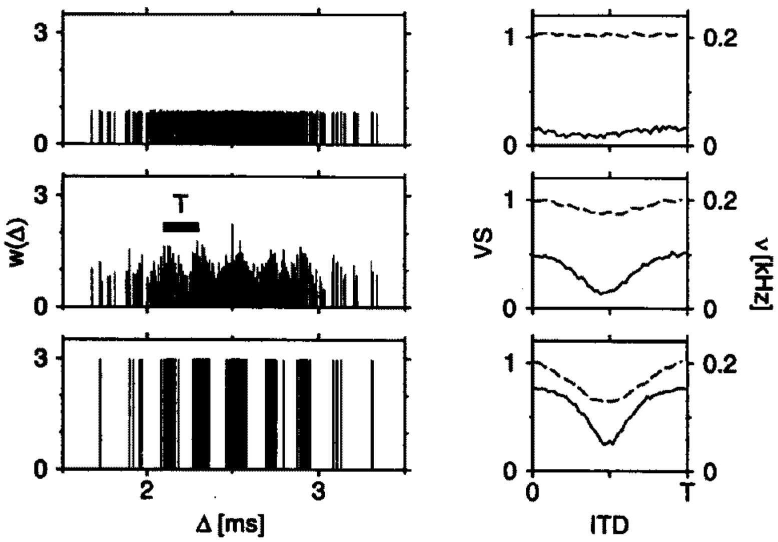


Figure 3.22: Development of tuning to a 5kHz tone. The left column shows the strength of synaptic efficacies w_{ij} of all synapses. Synapses are indexed according to the delay Δ_j of the corresponding transmission line and are plotted as $w_{ij} = w(\Delta)$. On the right, we show the vector strength (vs, solid line) and the output firing rate (ν, dashed) as a function of the inter-aural time delay (ITD). **Top.** Before learning, there are 600 synapses (300 from each ear) with different delays, chosen randomly from a Gaussian distribution with mean 2.5 ms and variance 0.3 ms. All weights have unit value. The output is not phaselocked ($vs \approx 0.1$) and shows no dependence upon the ITD. **Middle**. During learning, some synapses are strengthened others decreased. Those synapses which increase have delays that are similar or that differ by multiples of the period $T = 0.2\,ms$ of the stimulating tone. The vector strength of the output increases and starts to depend on the ITD. **Bottom.** After learning, only about 150 synapses ($\approx$ 75 from each ear) survive. Both the output firing rate ν and the vector strength vs show the characteristic dependence upon the ITD as seen in experiments with adult owls [23]. The neuron has the maximal response (ν = 200 Hz) for ITD= 0, the stimulus used during the learning session of the model neuron. The vector strength at ITD = 0 is vs $\approx$ 0.8 which corresponds to a temporal precision of 25 μs. Taken from [124].

Appendix A

A.1 Sample Results for the program *Love*

Example: *Love*
Rule 1: $loves(X, Y) \leftarrow gives\_rose(X, Y)$
Rule 2: $loves(Y, X) \leftarrow gives\_rose(X, Y)$
Rule 3: $gives\_rose(uli, bob)$.

In this section will see some sample log of the experimental results for the above program.

After obtaining the kernel for the *Love* program, we train the kernel on the training set shown in the sample pattern file. The pattern file `mpattern1` contains 21 entries. As explained in the experimental setup (section 3.9), the training set is an exemplary of the ground instances of the logic programming examples.

We assume during training that we have a large set of constants and a large set of real numbers such that each constant can be mapped to a distinct real number. However, as explained before the output neuron gets activated after a short time, and this time delay is set to 10-20 milli-seconds in our experimental setup. Hence if a spike at the output occurs at 1.1, and the presynaptic spike had occurred at 1.0, then we consider the number 1.0 and 1.1 as representatives (in our training data set) for a constant which in our program could be "adam".

A `C` program was written and compiled on the Sun Ultra Sparc workstations using the `GNU C` complier. The source code can be obtained from the following web site: http://pikas.inf.tu-dresden.de/∼ashish/thesis/source/.

The input for the program was a pattern file like `mpattern1`, and the output was stored in the files named as `mresult1`. Here 'm' means the input file contains mixed patterns (both good and bad). The result file shows us a log of individual iterations of weight update computations for the 4 links connecting the output layer and the input layer. At each step of the weight update computation, the difference between the presynaptic input and the postsynaptic output is calculated and this difference (s) is fed into the $W(s)$ function, where the value of s is compared to s^* (written as `s_star` in the result file). The data sets that were used for testing some other cases, can be found at the web site: http://pikas.inf.tu-dresden.de/∼ashish/thesis/data/.

If s is greater than s^*,the weight is decreased. If s is smaller than s^* the weight is increased.

The weight change shown in the result file in individual iterations is only printed upto the first place of decimal. It should be observed from the log of the result that even when the difference in pre-and postsynaptic spike was -1.1, even though the weight update function increases the weight (since the presynaptic spike precedes the postsynaptic spike), the increase is almost zero, printed to the first place of decimal as 0.0. The other values like A_+, A_-, τ_+, τ_-, τ_{syn} are also written as `A_plus`, `A_minus`, `t_plus`, `t_minus`, `t_syn`, respectively.

After the weight update is over, for all the 4 links, the final changes in the weight values are written in the form of `deltaW_ij`. This denotes the weight change on the link connecting the i^{th} neuron in the output layer to the j^{th} neuron in the input layer. For the sake of representing the data in experiments the names of the neurons $loves[1]$, $loves[2]$ were changed to 1 and 2 in the output layer, and similarly $gives\_rose[1]$ and $gives\_rose[2]$ were renamed as 1 and 2 in the input layer. Hence the final weight changes were written as: `deltaW_11`, `deltaW_12`, `deltaW_21`, `deltaW_22`. The total weight on all the links were obtained by adding the initial weight values which were all 1.0.

A.2 Rule Extraction

After the final weight values are obtained, one needs to extract the refined rules. For this the first step is to perform thresholding. Thresholding is the process in which a real number value is set to a threshold, and all numbers below this value will be considered as 0, and all numbers above this would be considered as 1. Thresholding is a mechanism for going from the real-valued continuous domain to a discrete valued boolean domain. This is done in order to get back refined rules for our predicate logic. Examining the weight values shown in the file `mresult1`, gives an impression that a value of 3.6 could be a good candidate for a threshold. This implies that the weights `W_11` and `W_22` would be changed to 1 each and `W_12` and `W_21` would be changed to 0.

Thresholding is an important feature of rule extraction mechanism, because in order to extract binary truth valued rules of predicate logic we need to segregate the real valued numbers into two ranges, *high* and *low*.

There is currently no universal way to set this threshold in our system. Any rule extraction algorithm should define such a threshold function. It should be stable enough in order to segregate the real valued weights correctly.

Once we have set up the threshold, we are now in a position to modify the kernel. The procedure is quite trivial, prune away the links with weight values less than the threshold and retain the links with weight values greater than the threshold. This would mean in our example, pruning away links connecting the neuron 1 in the output layer and the neuron 2 in the input layer, and the link between neuron 2 in the output layer and the neuron 1 in the input layer. This means that the links between the following pairs are removed: $\langle loves[1], gives\_rose[2]\rangle$ and $\langle loves[2], gives\_rose[1]\rangle$. The modified kernel is shown in Figure A.1 and the final modified neural network is shown in Figure A.2.

Looking at this spiking network reveals that the network will exhibit the following behaviour. Whenever the neuron $gives\_rose[1]$ fires, $loves[1]$ would fire either at the same time, or soon after (say within next 20 milliseconds). Similarly whenever the neuron $gives\_rose[2]$ fires, $loves[2]$ would fire either at the same time, or soon after (say within next 20 milliseconds). The notion of temporal binding suggests that neurons $gives\_rose[1]$ and $loves[1]$ represent identical object, and $gives\_rose[2]$ fires and $loves[2]$ represent identical object, and the features of first object is distinct from the feature of the second object. This means that if a feature say X appears at the $gives\_rose[1]$ neuron it should appear at the $loves[1]$ neuron as well. Similarly another distinct feature say Y when appears at $gives\_rose[2]$, appears at $loves[2]$ neuron. Hence we are now in a position to construct the first order rule from the network, by formalizing the above expressed notion Hence we can write: $loves(X,Y) \leftarrow gives\_rose(X,Y)$ or we can also write a rule like: $loves(Y,X) \leftarrow gives\_rose(Y,X)$

The important thing to remember is to bind the first argument in $gives\_rose$

predicate to the first argument in the *loves* predicate and the second argument in *gives_rose* predicate to the second argument of the *loves* predicate. We cannot construct *Rule 2* shown in the initial program, as there is no link between the following pairs:
$\langle loves[1], gives\_rose[2]\rangle$ and $\langle loves[2], gives\_rose[1]\rangle$.

Another feature during rule extraction which is important but is not illustrated by the above program is the idea of *distinct neurons.* A neuron i is distinct from a neuron j in the given layer (input or output) if they represent syntactically different variables in the given clause. This is an important notion. This tells us that in those cases where there are two neurons in the input layer for example, representing the same variable (the case where a variable in the body occurs in more than one predicate and also in the head predicate of the same clause) each of these two would be connected to the corresponding neuron in the output layer to which it has to be bound. This is because these neurons are not *distinct* from one another.

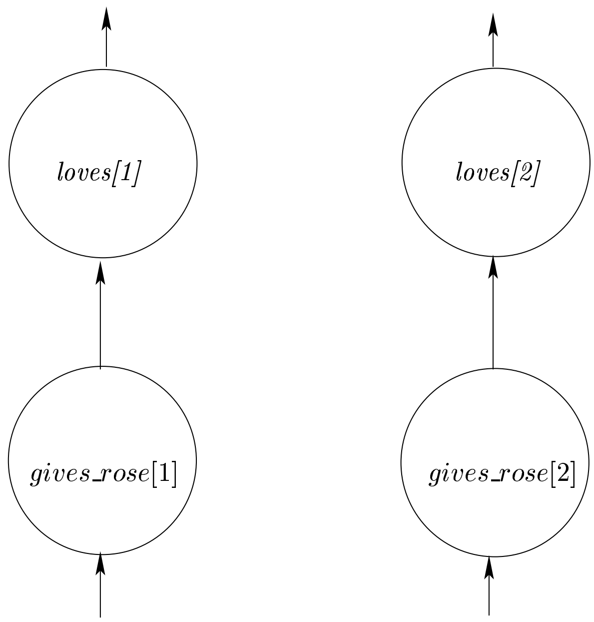


Figure A.1: The refined kernel

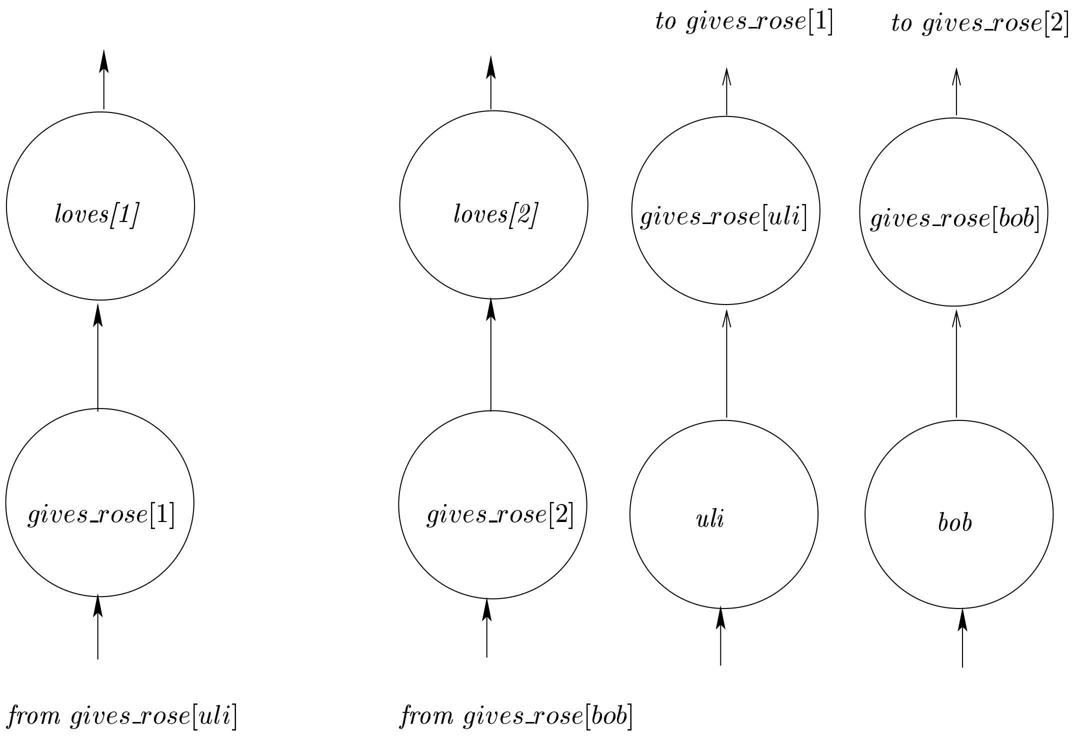


Figure A.2: The refined neural network

Sample Pattern File: `mpattern1`

$[1.1, 2.1]$ $[1, 2]$
$[3.2, 4.2]$ $[3, 4]$
$[5.1, 6.1]$ $[5, 6]$
$[7.1, 8.1]$ $[7, 8]$
$[9.1, 10.1]$ $[9, 10]$
$[11.1, 12.1]$ $[11, 12]$
$[13.1, 14.1]$ $[13, 14]$
$[15.1, 16.1]$ $[15, 16]$
$[17.1, 18.1]$ $[17, 18]$
$[19.1, 20.1]$ $[19, 20]$
$[21.1, 22.1]$ $[21, 22]$
$[23.1, 24.1]$ $[23, 24]$
$[25.1, 26.1]$ $[25, 26]$
$[27.1, 28.1]$ $[27, 28]$
$[30.1, 29.1]$ $[29, 30]$
$[20.1, 19.1]$ $[19, 20]$
$[22.1, 21.1]$ $[21, 22]$
$[24.1, 23.1]$ $[23, 24]$
$[26.1, 25.1]$ $[25, 26]$
$[28.1, 27.1]$ $[27, 28]$
$[30.1, 29.1]$ $[29, 30]$

Sample Pattern File: `mresult1`

$[1.1, 2.1]$ $[1, 2]$
$[3.2, 4.2]$ $[3, 4]$
$[5.1, 6.1]$ $[5, 6]$
$[7.1, 8.1]$ $[7, 8]$
$[9.1, 10.1]$ $[9, 10]$
$[11.1, 12.1]$ $[11, 12]$
$[13.1, 14.1]$ $[13, 14]$
$[15.1, 16.1]$ $[15, 16]$
$[17.1, 18.1]$ $[17, 18]$
$[19.1, 20.1]$ $[19, 20]$
$[21.1, 22.1]$ $[21, 22]$
$[23.1, 24.1]$ $[23, 24]$
$[25.1, 26.1]$ $[25, 26]$
$[27.1, 28.1]$ $[27, 28]$
$[30.1, 29.1]$ $[29, 30]$
$[20.1, 19.1]$ $[19, 20]$
$[22.1, 21.1]$ $[21, 22]$
$[24.1, 23.1]$ $[23, 24]$
$[26.1, 25.1]$ $[25, 26]$
$[28.1, 27.1]$ $[27, 28]$
$[30.1, 29.1]$ $[29, 30]$

```
No of input neurons is 2

No of output neurons is 2

s_star is -0.150

A_plus is 0.500

A_minus is 0.200

t_plus is 0.500

t_minus is 5.000

t_syn is 0.500
```

```
The weight change corresponding to the time difference
-0.1 is 0.3 for the link between output neuron 1 and
input neuron 1

The total weight change is 0.3 for the link between
output neuron 1 and input neuron 1

The weight change corresponding to the time difference
0.9 is -0.1 for the link between output neuron 1 and
input neuron 2

The total weight change is -0.1 for the link between
output neuron 1 and input neuron 2

The weight change corresponding to the time difference
-1.1 is 0.0 for the link between output neuron 2 and
input neuron 1

The total weight change is 0.0 for the link between
output neuron 2 and input neuron 1

The weight change corresponding to the time difference
-0.1 is 0.3 for the link between output neuron 2 and
input neuron 2

The total weight change is 0.3 for the link between
output neuron 2 and input neuron 2

The weight change corresponding to the time difference
-0.2 is 0.3 for the link between output neuron 1 and
input neuron 1
```

$\vdots$

```
The weight change corresponding to the time difference
-0.1 is 0.3 for the link between output neuron 1 and
input neuron 2

The total weight change is 0.4 for the link between
output neuron 1 and input neuron 2

The weight change corresponding to the time difference
-0.1 is 0.3 for the link between output neuron 2 and
input neuron 1
```

The total weight change is 2.4 for the link between output neuron 2 and input neuron 1

The weight change corresponding to the time difference 0.9 is -0.1 for the link between output neuron 2 and input neuron 2

The total weight change is 2.9 for the link between output neuron 2 and input neuron 2
deltaW_11 is = 3.9

W_11 is = 4.9

deltaW_12 is = 0.4

W_12 is = 1.4

deltaW_21 is = 2.4

W_21 is = 3.4

deltaW_22 is = 2.9

W_22 is = 3.9

Appendix B

Here will see how we construct the networks when there are special cases met in the transformation algorithm described in Section 3.4.3.

B.1 Illustrating the Transformation Algorithm

Case I: If a predicate p in the head of a clause is occuring in the body of the same clause and the predicate occurrence in head and in the body contains only variables

Example I
Rule 1: $p(X, Y) \leftarrow p(Y, X)$
Rule 2: $p(a, b)$

Given the above program, the predicate p that occurs in the head of *Rule 1*, occurs in the body of *Rule 1* as well. Hence the conditions are met for the above case. When the transformation algorithm is applied to the above program one can obtain the network shown in Figure B.1. In this network the links from $p[1]$ and $p[2]$ in the output layer denoting the variable X and Y of the predicate p in *Rule 1*, are connected to $p[2]$ and $p[1]$ in the input layer respectively. Since $p[1]$ and $p[2]$ in the input layer denote Y and X respectively, we exercise caution here while connecting this recursive part. In this sense, the network is a bit hard-wired here, and it can't really learn anything wrong in this part. Figure B.2 shows the kernel for this network and Figure B.3 shows the network after the training is over, here we assume that we had only learnt the rule of the kind given as *Rule 1*. Figure B.4 shows how to read model from the timing diagram for the trained network. The reader should observe that there is a fact also in the program, and hence

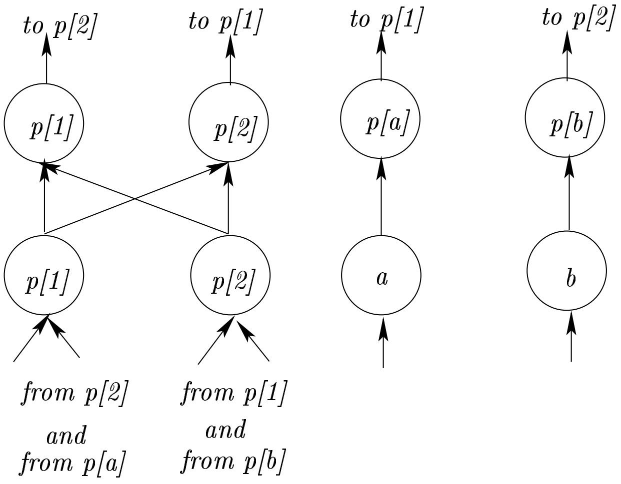


Figure B.1: Before training the network

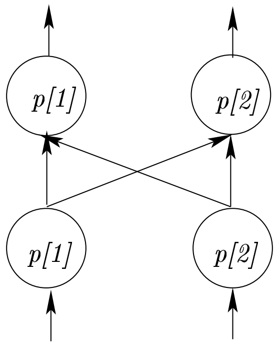


Figure B.2: The kernel

recursive links should be drawn from the fact arguments $p[a]$ and $p[b]$ to $p[1]$ and $p[2]$ respectively in the input layer.

Additional observation of timing diagram for reading the model (Figure B.4) would reveal an instance of cross talk problem. After one actuates $p(a, b)$ one gets $p(b, a)$. However reading from the timing diagram one can also read $p(a, a)$ and $p(b, b)$. This is something that is still present in our system.

Case II: If a predicate p in the head of a clause is occuring in the body of a different clause, and the predicate occurrence in head and in the body contains only variables

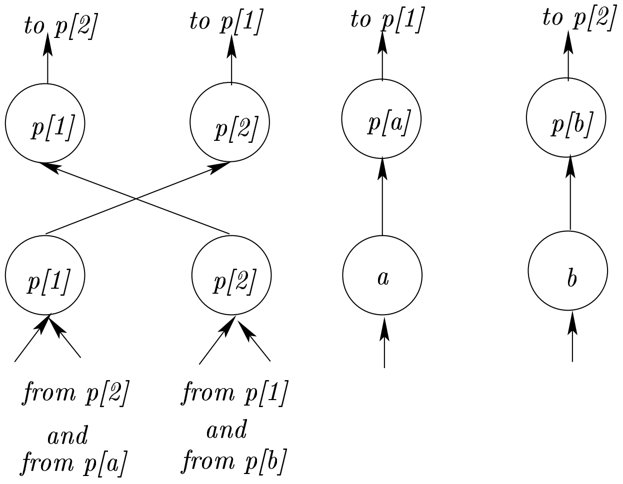


Figure B.3: After training assuming the rules were correct and supporting bias obtained from training data

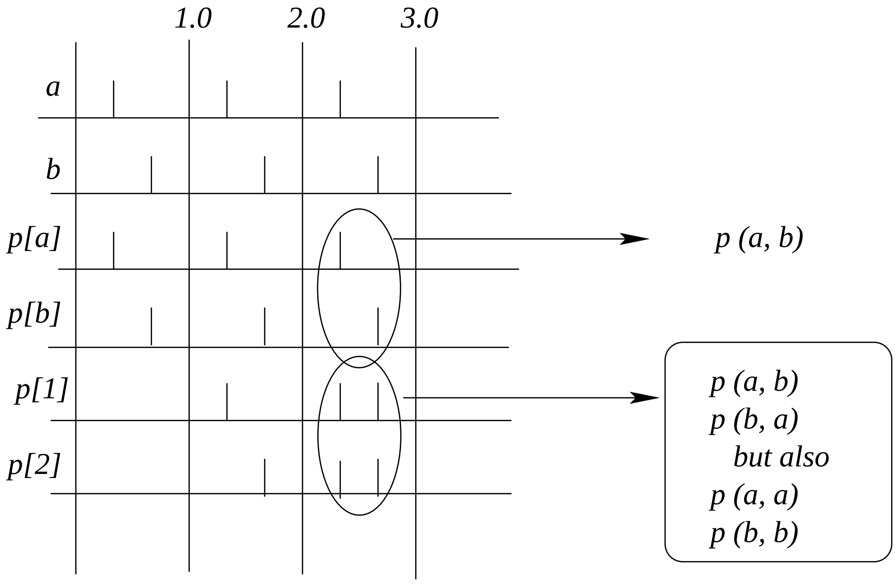


Figure B.4: Reading the model

Example II

Rule 1: $p(X,Y) \leftarrow q(Y,X)$
Rule 2: $q(X,Y) \leftarrow p(X,Y)$
Rule 3: $p(a,b)$

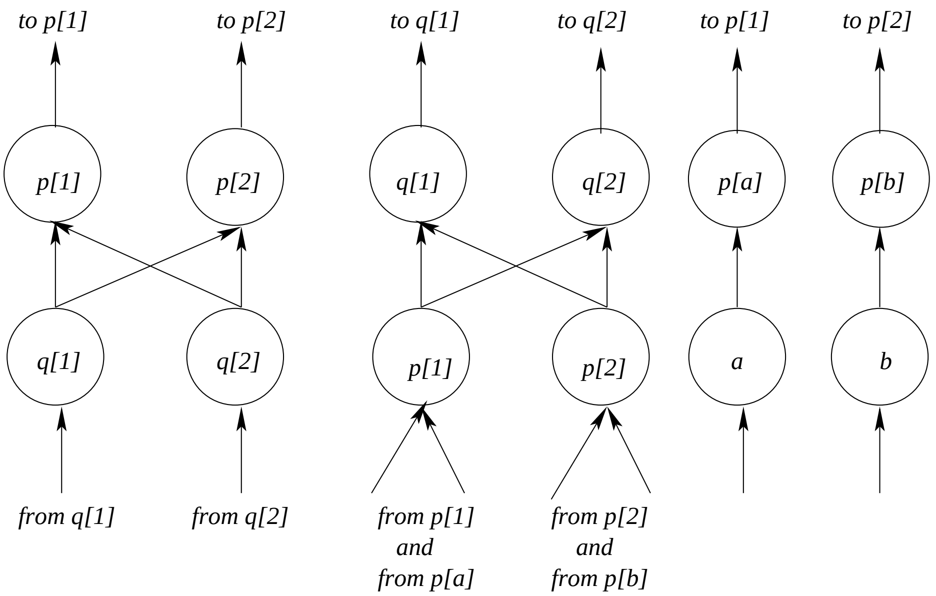


Figure B.5: Before training the network

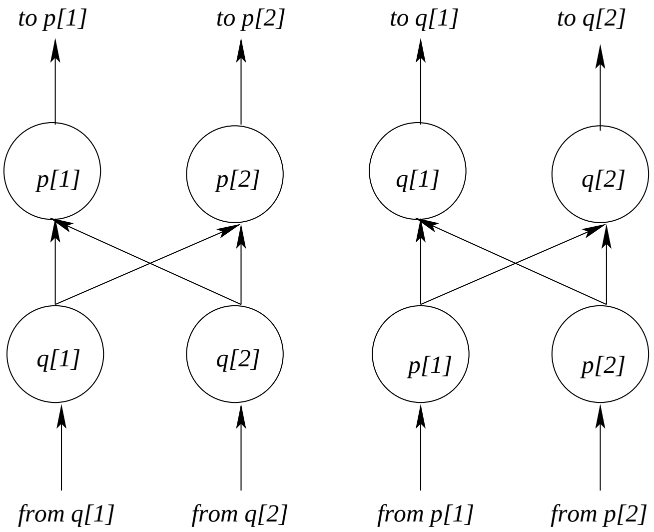


Figure B.6: The kernel

The above program illustrates the case, where a predicate p in *Rule 1* occurs in the body of different clause *Rule 2*, and in each occurrence of this predicate there are only variables in the argument. Here we make links

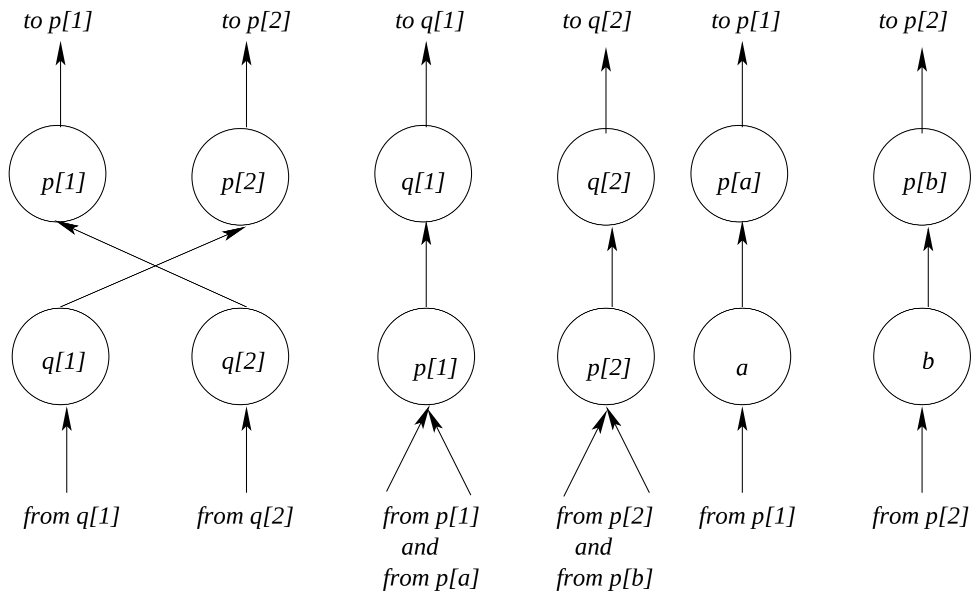


Figure B.7: After training assuming the rules were correct and supporting bias obtained from training data

from $p[1]$ and $p[2]$ in the output layer to $p[1]$ and $p[2]$ in the input layer respectively (see Figure B.5 - B.7). In this case we have to be cautious but in a slightly different way. We have to make sure that we map the corresponding argument positions, and not the variables, as it was in the first case. This means that it is only important to connect the argument at position 1 in predicate p –corresponding neuron $p[1]$ in input layer (*Rule 2*) to the argument at position 1 in predicate p –corresponding neuron $p[1]$ in the output layer (*Rule 1*) and similarly for position 2. Moreover while making recursive links from the q neurons of the output layer to the q neurons in the input layer, we may not exercise the caution that we did in *Case 1.* This is because the predicate q is occuring in the head of one rule and in the body of another. Hence the bindings would not be transferred wrong if we connect the $q[1]$ and $q[2]$ neuron in the output layer to the $q[1]$ and $q[2]$ neuron in the input layer respectively. The presence of the fact $p(a, b)$ ensures further recursive connections from neurons $p[a]$ and $p[b]$ to neurons $p[1]$ and $p[2]$ in the input layer.

Case III: If a predicate p in the head of a clause contains a constant and the predicate p in the body of same or different clause contain identical con-

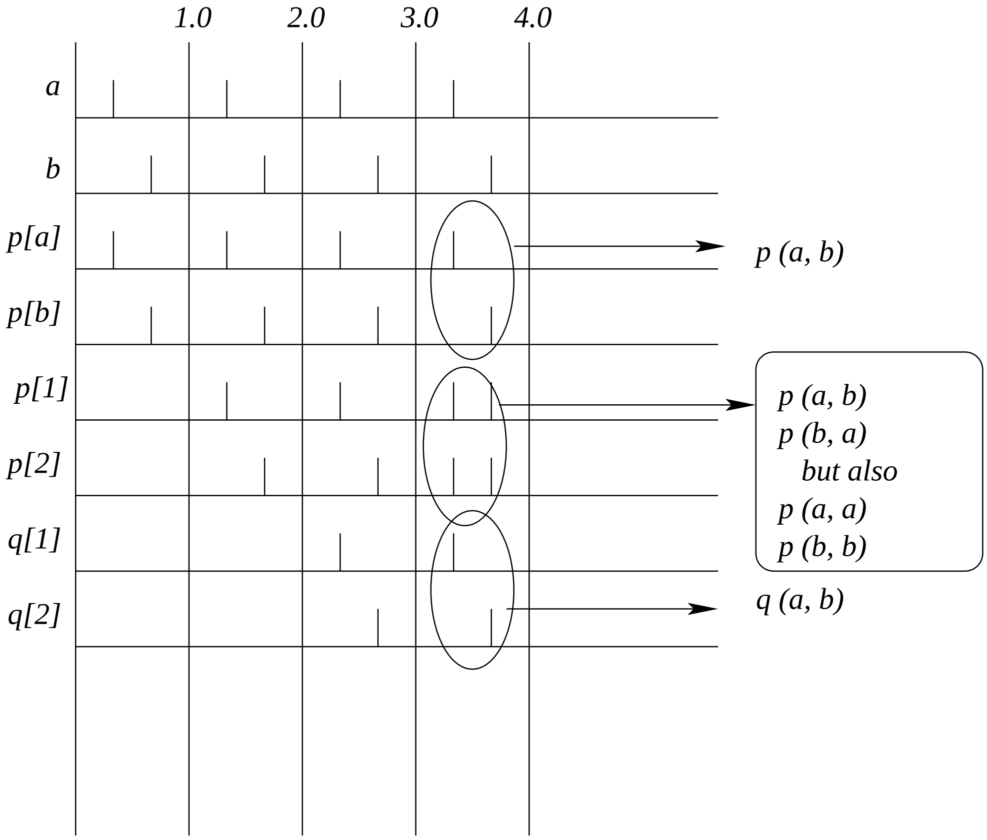


Figure B.8: Reading the model

stants

Example III
Rule 1: $q(c, X, Y) \leftarrow r(f, Y, X) \wedge s(X, Y, f)$
Rule 2: $r(c, b, a)$.
Rule 3: $r(f, e, d)$.
Rule 4: $s(d, e, f)$.

This program above shows two interesting points. Firstly, we should observe that the predicate r and s in the head of *Rule 3* and *Rule 4* resp. contain a constant f, and the predicate r and s in the body of *Rule 1* also contain the identical constant, and hence we satisfy the above case. Hence we have to connect the $r[f]$ and $s[f]$ neurons in the output layer to the

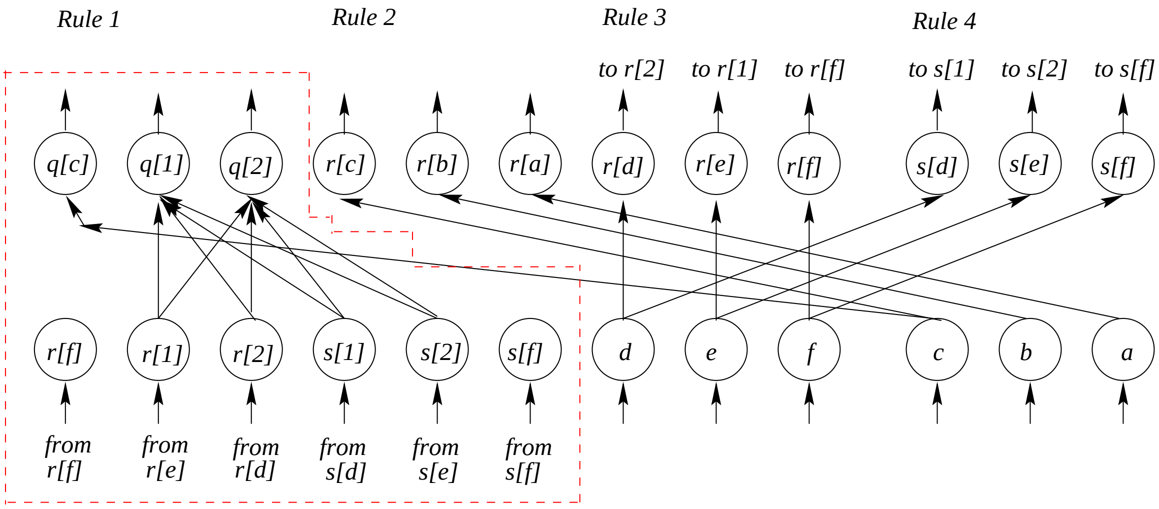


Figure B.9: Before training the network

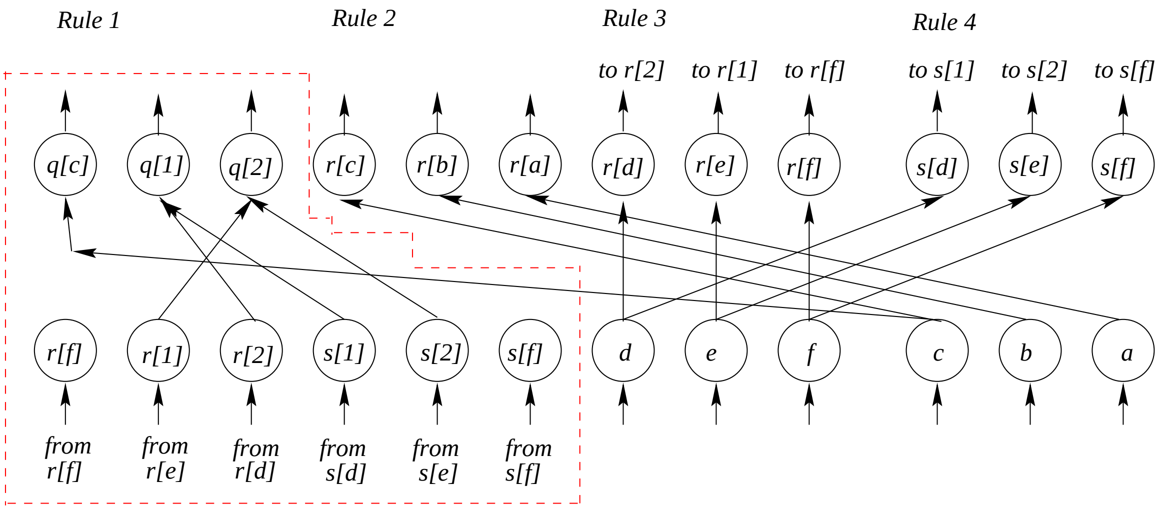


Figure B.10: After training the network

corresponding $r[f]$ and $s[f]$ in the input layer (see Figure B.9 - B.11). We should note that we have not drawn any recursive connections from the output neurons $r[c]$, $r[b]$ and $r[a]$ to the predicate r in the input layer. This is because the constant at position 1 in the predicate r in *Rule 1* is not identical to the constant at position 1 in the predicate r in *Rule 2*.

Secondly, the constant c that appears in the head of *Rule 1* does not

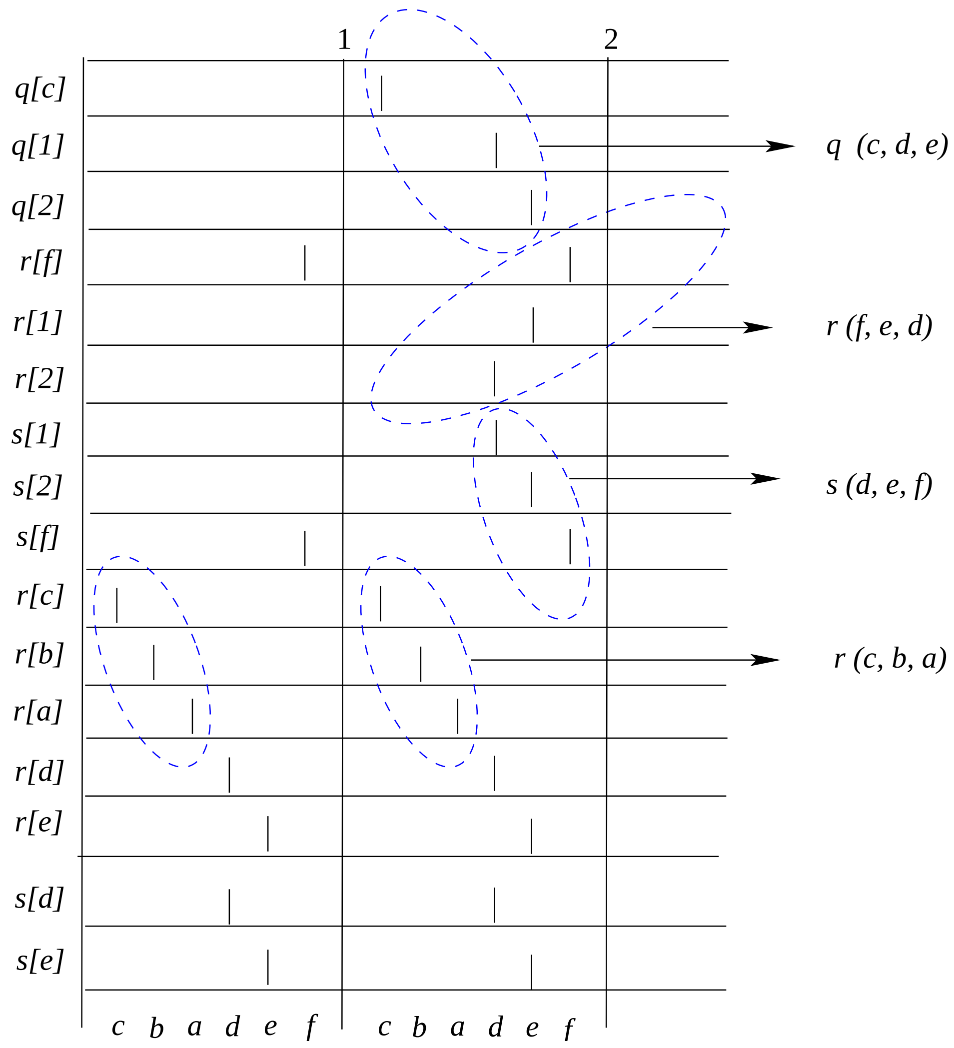


Figure B.11: Reading the model

appear in the body of *rule 1*, and hence the constant c in the input layer is joined to $q[c]$ and $r[c]$ both in the output layer.

Case IV: If a predicate p in the head of a clause contains a constant and the predicate p in the body of same or different clause does not contain that constant

We can illustrate the next case also with respect to Example III. The predicates $r(f, e, d)$ and $s(d, e, f)$ in the head of *Rule 3* and *Rule 4* contains constants d and e, but r and s in the body of *Rule 1* do not contain any constants d and e. The constants e and d occur at position 2 and 3 resp. in the predicate r in *Rule 3*; and at position 2 and 1 in predicate s in *Rule 4*. The argument neurons $r[e]$ and $r[d]$ in the output layer are connected to $r[1]$ and $r[2]$ in the input layer respectively. This is because at position 2 ($r[1]$) and 3 ($r[2]$) in the predicate r in *Rule 1*, we don't have constants but variables. Moreover $s[d]$ and $s[e]$ in the output layer are connected to $s[1]$ and $s[2]$ in the input layer, for the same reason (see Figure B.9 - B.11)

Case V: If a predicate p in the body of a clause contains a constant and the predicate p in the head of same or different clause does not contain that constant

Example IV
Rule 1: $q(c, X) \leftarrow r(c, b, X)$
Rule 2: $r(c, Y, Z)$.

The above program is chosen to illustrate the point that we can have variables in the facts as well and more importantly the body predicate may have some constant at position i which may not be present in the fact predicate (which is identical to the body predicate) at position i. With respect to the above program, this position is 2 in the predicate r. In *Rule 1* the constant b is there at position 2 in the predicate r in the body, and this constant is not present in the fact predicate r at position 2. This implies that during the chain of forward reasoning, the variable Y should get bound to this constant b. This is precisely what our solution provides us. By connecting the neuron $r[1]$ in the output layer to the neuron $r[b]$ in the input layer (Figure B.12 and B.14) we ensure that these bindings between b and Y take place. The kernel is shown in Figure B.13.

Additionally we have connected the $r[2]$ neuron in the output layer to the $r[1]$ in the input layer, to ensure that the variable X in the body of *Rule 1* may get bound to the variable Z appearing in the fact (shown in Figure B.15). From the logic programming point of view, this ensures that the *variable renaming* takes place during this binding. Why we have decided to ensure that X got bound to Z and not Z to X is just a matter of choice. We can also read the unification result as Z getting bound to X, however this would just lead to shifting of time points on the timing diagram.

The other notable observation, is that the original network does not change after the training. We have assumed here as in the previous cases that the network was shown enough examples in order to preserve the original *Rule 1*. Since in this case there were no connections from $q[c]$ to $r[1]$ because there was a constant in the head, rather than being a variable, we had less choices to make. The connection from $r[1]$ to $q[1]$ would have faded away had there not been sufficient examples where the third argument of r in the body was bound to second argument of q in the head of *Rule 1*.

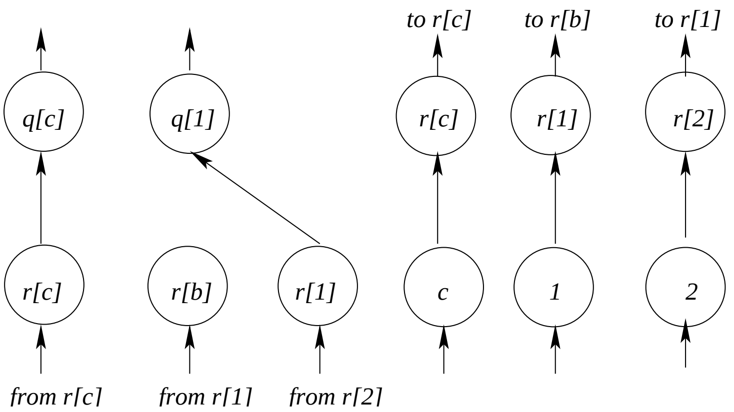


Figure B.12: Before training the network

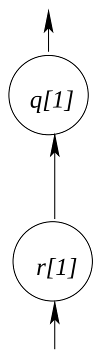


Figure B.13: The kernel

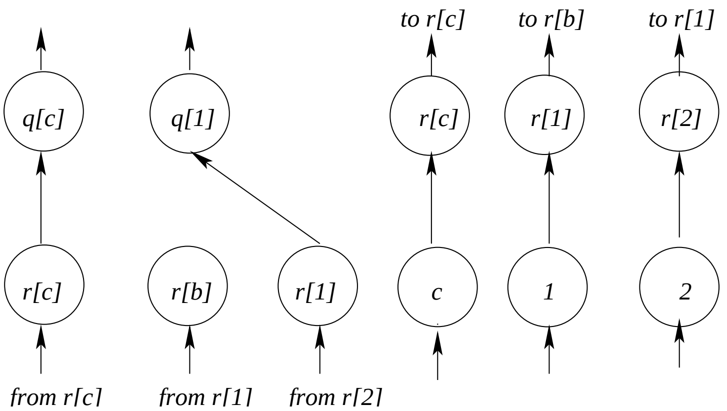


Figure B.14: After training, the network does not change

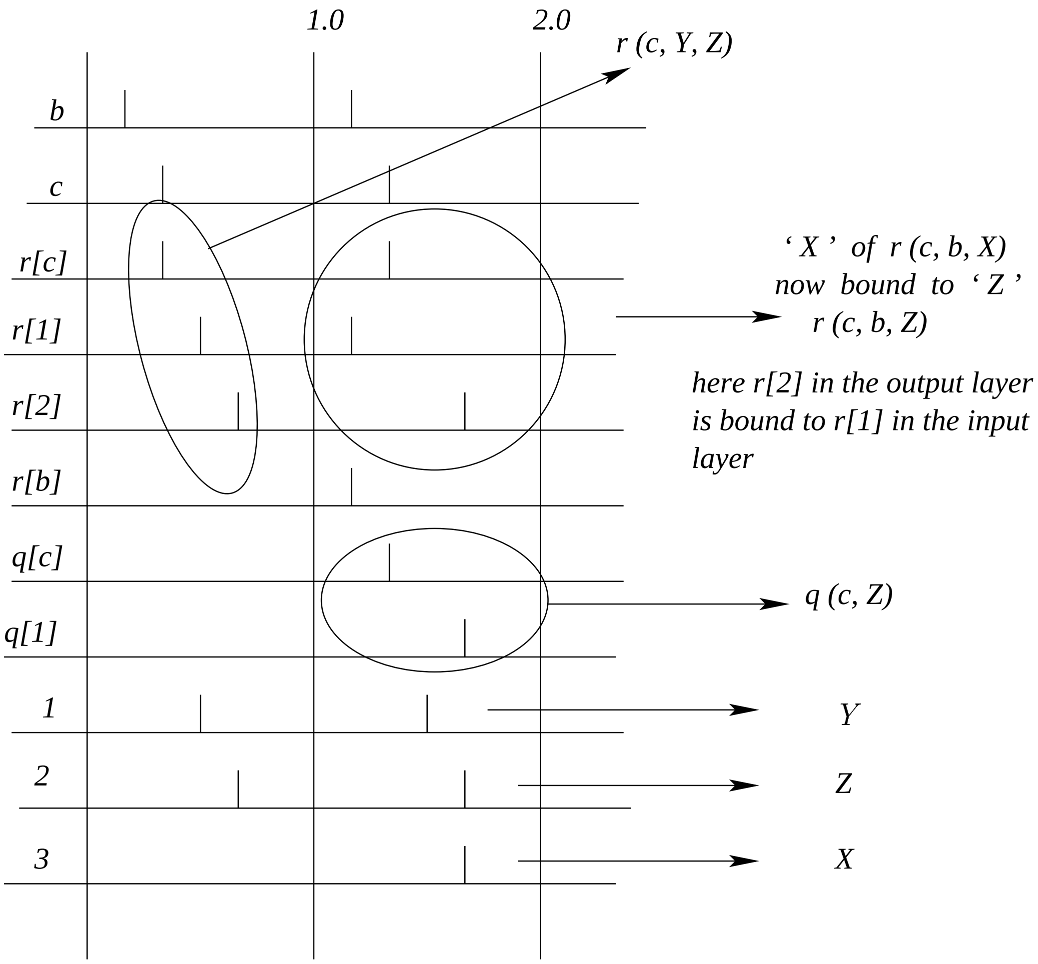


Figure B.15: Reading the model

Bibliography

[1] Aersten A. and Arndt M. Response synchronization in the visual cortex. *Current Opinion in Neurobiology*, 3:586-594, 1993.

[2] Bragin A., Jandó G., Nádasdy Z., Hetke J. K., Wise K., and Buzáski G. Gamma (40-100 hz). *J. Neurosciences*, 15:47-60, 1995.

[3] Robinson J. A. A machine oriented logic based on the resolution principle. *J. ACM*, 12:23-41, 1965a.

[4] Robinson J. A. Automatic deduction with hyper-resolution. *Internat. J. Comput. Math*, 1:227-234, 1965b.

[5] Srinivasan A., Muggleton S., King R. D, and Sternberg M. J. E. Mutagenesis: ILP experiments in a non-determinate biological domain. In *Proceedings of the Fourth Inductive Logic Programming Workshop*, 1994.

[6] V. Ajjanagadde and L. Shastri. Rules and variables in neural nets. *Neural Computation*, 3:121–134, 1991.

[7] R. Andrews, J. Diederich, and A. B. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge Based Systems*, 8(6), 1995.

[8] Dolsak B. and Muggleton S. The application of inductive logic programming to finite element mesh design. In *In S. Muggleton (Ed.), Inductive Logic Programming*. London:Academic Press, 1992.

[9] Katz B. Nerve, muscle and synapse. McGraw-Hill, New York, 1966.

[10] A. E. Bryson and Y. C. Ho. Applied optimal control. *Blaisdell*, 1969.

[11] Kandel E. C. and Schwartz J. H. *Principles of Neural Science*. Elsevier New York, 3rd edition, 1991.

[12] G. Clossman. *A model of categorisation and learning in a connectionist broadcast system*. PhD thesis, Department of Computer Science, Indiana University, 1988.

[13] Adrian E. D. The impulses produced by sensory nerve endings. *J. Physiology(London)*, 61:49-72, 1926.

[14] Cowan J. D. Statistical mechanics of nervous nets. In *NATO Conference on Neural Networks*. Springer, Berlin, 1968.

[15] MacKay D. and McCulloch W. S. The limiting information capacity of a neuronal link. *Bull. Math. Biophys.*, 75:2467-2485, 1952.

[16] Marr D. Simple memory: A theory of archicortex. *Philosophical Tran. Royal Soc. B*, 262:23-81, 1971.

[17] Terman D. and Wang D. Global competition and local cooperation in a network of neural oscillators. *Physica D.*, 81:148-176, 1995.

[18] Wang D. Emergent synchrony in locally coupled neural oscillators. *IEEE Transactions on Neural Networks*, 6:941-948, 1995.

[19] Wang D., Buhmann J., and von der Malsburg. C. Pattern segmentation in associative memory. *Neural Computation*, 2:94-106, 1990.

[20] Ashish Darbari. *Rule Extraction from Trained* ANN*: A Survey*. Technical report, Institute of Artificial Intelligence, Dept. of Computer Science, TU Dresden, Germany, 2000.

[21] A. S. d'Avila Garcez, K. Broda, and D. Gabbay. *Symbolic Knowledge Extraction from Trained Neural Networks: A New Approach*. Technical Report TR-98-014, Department of Computing, Imperial College, London, 1998.

[22] Carr C. E. Processing of temporal information in the brain. *Ann. Review Neurosci.*, 16:223-243, 1993.

[23] Carr C. E. and Konishi M. A circuit for detection of interaural time differences in the brain stem of the barn owl. *J. Neurosciences*, 10:3227-3246, 1990.

[24] Rumelhart D. E., G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 1986a. Reprinted in Anderson and Rosenberg[1988].

[25] Rumelhart D. E., G. E. Hinton, and R. J. Williams. Learning internal representations by error backpropagation. In *In Parallel Distributed Processing*, chapter 8. 1986b.

[26] Vaadia E., Aertsen A., and Nelken I. Dynamics of neuronal interactions cannot be explained by neuronal transients. *Proc. Royal Soc. of London*, 261:407-410, 1995.

[27] A. K. Engel, P. Koenig, C. M. Kreiter A. K. Gray, and W. Singer. Temporal coding by coherent oscillations as a potential solution to the binding problem: physiological evidence. In *H. G. Schuster and W. Singer (Eds.) Nonlinear Dynamics and Neural Networks*. Weinheim, Bethesda, MD, 1993.

[28] Abbott L. F., Sen K., Varela J. A., and Nelson S. B. Synaptic depression and cortical gain control. *Science*, 275:220-222, 1997.

[29] Mainen Z. F. and Sejnowski T. J. Reliability of spike timing in neocortical neurons. *Science*, 268:1503-1506, 1995.

[30] Rieke F., Warland D., de Ruyter van Steveninck R. R., and Bialek W. *Spikes - Exploring the neural code*. MIT Press, Cambridge M.A., 1996.

[31] S. E. Fahlman. Netl: A system for representing real-world knowledge. MIT Press, 1979.

[32] Buchanan B. G., Smith D. H.and White W. C., Gritter R., Feigenbaum E. A., Lederberg J., and Djerassi C. Applications of artificial intelligence for chemical inference, XXII: Automatic rule formation in mass spectrometry by means of the meta-dendral program. *Journal of the American Chemical Society*, 98, 1976. 6168.

[33] Lynch G. *Synapses, circuits and the beginnings of memory.* MIT Press, 1986.

[34] Wulfram Gerstner. Spiking neurons. In W. Maass and C. M. Bishop, editors, *Pulsed Neural Networks.* MIT Press, 1998. pp 3 - 54.

[35] Brown T. H., Kairiss E. W., and Keenan C. L. "hebbian synapses: Biophysical mechanisms and algorithms. *Annual Review of Neuroscience*, 13:475-511, 1990.

[36] Hubel D. H. Eye, brain and vision. W. H.Freeman, New York.

[37] Hubel D. H. and Wiesel T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J. Physiology(London)*, 160:106-154, 1962.

[38] Hubel D. H. and Wiesel T. N. Functional architecture of macaque monkey visual cortex. *Proc. Royal Soc. B.*, 198:1-59, 1977.

[39] Markram H., Lübke J., Frotscher M., and Sakmann B. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275:213-215, 1997.

[40] Markram H. and Tsodyks M. Redistribution of synaptic efficacy between neocortical pyramidal neurons. *Nature*, 382:807-10, 1996.

[41] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Maxwell Macmillan International, 1994.

[42] Goang-Tay Hsu and Reid Simmons. Learning footfall evaluation for a walking robot. In *Proccedings of the Eighth International Workshop on Machine Learning*, pages 303–307, Evanston,IL, June 1991.

[43] Herz J., Krogh A., and Palmer R. G. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City CA., 1991.

[44] Hopfield J. J. Pattern recognition computation using action potential timing for stimulus representation. *Nature*, 1995. 376:33-36.

[45] Hopfield J. J. and Herz A. V. M. Rapid local synchronization of action potentials: towards computation with coupled integrate-and-fire networks. *Proc. NAt. Acad. Sci. USA*, 92:6655, 1995.

[46] O' Keefe J. and Recce M. Phase relationship between hippocampal place units and the hippocampal theta rhythm. *Hippocampus*, 3:317-330, 1993.

[47] Prechtl J., Cohen L. B., Pesaran B., Mitra P. P., and Kleinfield D. Visual stimuli induce waves of electrical activity in turtle cortex. *Proc. Nat. Acad. Sci*, 94:7621-7626, 1997.

[48] Tovee M. J. and Rolls E. T. Information encoding in short firing rate epochs by single neurons in the primate temporal visual cortex. *Visual Cognition*, 2(1):35-58, 1995.

[49] Tovee M. J., Rolls E. T., Treves A., and Belles R. P. Information encoding and the responses of single neurons in the primate visual cortex. *J. Neurophysiology*, 70:640-654, 1993.

[50] M. Jabri, S. Pickard, and P. Leong. ANN based classification for heart defibrillators. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems*, pages 637–644, San Mateo, CA, 1992.

[51] Engel A. K., König P., Kreiter A. K., and Singer W. Interhemispheric synchronization of oscillatory neural responses in cat visual cortex. *Science*, 252:1177-1179, 1991.

[52] Engel A. K., Roelfsema P. R., Fries P., Brecht M., and Singer W. Role of the temporal domains for response selection and perceptual binding. *Cerebral Cortex*, 7:571-582, 1997.

[53] Kreiter A. K. and Singer W. Oscillatory neuronal response in the visual cortex of the awake macaque monkey. *Eur. J. Neurosciences*, 4:369-375, 1992.

[54] Kreiter A. K. and Singer W. Stimulus dependent synchronization of neuronal responses in the visual cortex of the awake macaque monkey. *The Journal of Neuroscience*, 16(7):2381-2396, 1996.

[55] Lindsay R. K., Buchanan B. G., Feigenbaum E. A., and Lederberg J. *Applications of Artificial Intelligence for organic chemistry.* New York:McGraw-Hill, 1980.

[56] Magleby K. Short term synaptic plasticity. In Edelman G. M., Gall W. E., and Cowan W. M., editors, *Synaptic Function*. Wiley, New York, 1987.

[57] Dobrunz L. and Stevens C. Hetrogeneity of release probability, facilitation and depletion at central synapses., 1997. 18:995-1008.

[58] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 1989.

[59] Abeles M. Role of the cortical neuron: integrator or coincidence detector? *Israel J. Medical Sciences*, 18:83-92, 1982.

[60] Abeles M. *Corticonics*. Cambridge University Press, 1991.

[61] Abeles M. Firing rates and well-timed events. In K. Schulten E. Domany and J. L. van Hemmen, editors, *Models of Neural Networks 2*, chapter 3, pages 121–140. Springer, New York, 1994.

[62] Abeles M., Bergman H., Margalit E., and Vaadia E. Spatiotemporal firing patterns in the frontal cortex of behaving monkeys. *J. Neurophysiology*, 70:1629-1638, 1993.

[63] Bishop C. M. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[64] Gray C. M., König P., Engel A. K., and Singer W. Oscillatory reponses in cat visual cortex exhibit intercolumnar synchronization which reflects global stimulus properties. *Nature*, 338:334-337, 1989.

[65] Gray C. M. and Singer W. Stimulus specific neuronal oscillations in orientation columns of cat visual cortex. In *Proc. Natl. Acad. Sci. USA*, volume 86:1698-1702, 1989.

[66] Konishi M. Centrally synthesized maps of sensory space. *Trends Neurosci*, 9:163-168, 1986.

[67] Konishi M. Listening with two ears. *Scientific American*, 34-41, 1993.

[68] Milner P. M. A model for visual shape recognition. *Psychol. Rev*, 81:521-535, 1974.

[69] Optican L. M. and Richmond B. J. Temporal encoding of two-dimensional patterns by single units in primate inferior temporal cortex. 3. information theoretic analysis. *J. Neurophysiology*, 57:162-178, 1987.

[70] Zador A. M., Maass W., and Natschläger T. Learning in neural networks with dynamic synapses.

[71] W. Maass. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8(1):1–40, 1996.

[72] W. Maass. Fast sigmoidal networks via spiking neurons. *Neural Computation*, 9:279–304, 1997.

[73] W. Maass. Computing with spiking neurons. In W. Maass and C. M. Bishop, editors, *Pulsed Neural Networks*, pages 55–85. MIT Press (Cambridge), 1999.

[74] W. Maass and T. Natschläger. Networks of spiking neurons can emulate arbitrary hopfield nets in temporal coding. *Network: Computation in Neural Systems*, 8(4):355–372, 1997.

[75] W. Maass and A. Zador. Computing and learning with dynamic synapses. In W. Maass and C. Bishop, editors, *Pulsed Neural Networks*, pages 321–336. MIT-Press (Cambridge), 1998.

[76] *The information content of action potential trains: a synaptic basis.* Springer Verlag, 1997. Artificial Neural Networks - ICANN '97, Lecture Notes in Computer Science, 1327.

[77] Tom M. Mitchell. *Machine Learning.* WCB/McGrawHill, 1997.

[78] Lavrăc N. and Džeroski S. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[79] Shadlen M. N. and Newsome W.T. Noise, neural codes and cortical organization. In *Current Opinion in Neurobiology*, volume 4:569-579. 1994.

[80] Hebb D. O. *The organization of Behavior.* Wiley, New York, 1949.

[81] Jensen O. and Lisman J. E. Hippocampal ca3 region predicts memory sequences: accounting for the phase precession pf place cells. *Learning and Memory*, 3:279-287, 1996.

[82] Changeux J. P. and A. Danchin. Selective stabilization of developing synapses as a mechanism for the specification of neural networks. *Nature(London)*, 264:705-712, 1976.

[83] Engel A. K. König P. and Singer W. Direct physiological evidence for scene segmentation by temporal coding. In *Proc. Natl. Acad. Sci. USA*, volume 88:9136-9140, 1991.

[84] Georgopoulos A. P., Schwartz A., and Kettner R. E. Neuronal population coding of movement direction. *Science*, 233:1416-1419, 1986.

[85] König P. and Schillen T. B. Stimulus dependent assembly formation of oscillatory responses: I. synchronization. *Neural Computation*, 3:155-166, 1991.

[86] Nam Seog Park. *A Connectionist Representation of First-Order Formulae with Dynamic Variable Binding.* PhD thesis, University of Edinburgh, 1997.

[87] Nam Seog Park. Addressing knowledge-representation issues in connectionist symbolic rule encoding for general inference. *LNAI 1778*, 2000.

[88] Nam Seog Park and David Robertson. A localist network architecture for logical inference. Department of Artificial Intelligence, University of Edinburgh.

[89] D. B. Parker. Learning logic. Technical report, Center for Compuatational Research in Economics and Management Science, MIT,MA, 1985.

[90] D. A. Pomerlau. ALVINN: an autonomous land vehicle in a neural network. Technical report, Computer Science Dept., Carnegie Mellon University, Pittsburgh PA, 1989.

[91] Eckhorn R., Reitboeck H. J, Arndt M., and Dicke P. Feature linking via synchronization among distributed assemblies: Simulations of results from cat visual cortex. *Neural Computation*, 2:293-307, 1990.

[92] Eckhorn R., Bauer R., Jordan W., Brosch M, Kruse W., Munk M., and Reitboeck H. J. Coherent oscillations: A mechanism of feature linking in the visual cortex? *Biol. Cybernetics*, 60:121-130, 1988.

[93] Lestienne R. Determination of the precision of spike timing in the visual cortex of anaesthetised cats. *Biol. Cybernetics*, 74:55-61, 1996.

[94] Quinlan J. R. Induction of decision trees. *Machine Learning*, 1986.

[95] Ritz R. and Sejnowski T. J. Synchronous oscillatory activity in sensory systems: new vistas on mechanisms. *Current Opinion in Neurobiology*, 1997.

[96] Ritz R., Gerstner W., and van Hemmen J. L. A biologically motivated and analytically soluble model of collective oscillations in the cortex: Ii. application to binding and pattern segmentation. *Biol. Cybernetics*, 71:349-358, 1994.

[97] Rojas R. *Neural Networks: A Systematic Introduction*. Springer, Berlin Heidelberg, 1996.

[98] Softky W. R. Simple codes versus efficient codes. In *Current Opinion in Neurobiology*, volume 5:239-247. 1995.

[99] Squire L. R. *Memory and brain*. Oxford University Press, 1987.

[100] Squire L. R. and Zola-Morgan S. The medical temporal lobe memory system. *Science*, 253:1380-86, 1991.

[101] Zucker R. Short-term synaptic plasticity. *Annual Review of Neuroscience*, 12:13-31, 1989.

[102] Hölldobler S., Y. Kalinke, and J. Wunderlich. A recursive neural network for reflexive reasoning. In Stefan Wermter and Ron Sun (eds.), Lecture Notes in Aritificial Intelligence 1778, January 2000.

[103] Hölldobler S., Kalinke Y., and H. P. Störr. Recurrent neural networks to approximate the semantics of acceptable logic programs. In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence*, volume 1502:LNAI, Berlin, Heidelberg. Springer-Verlag, 1998. Procceedings of the 11th Australian Joint Conference on Artificial Intelligence.

[104] Kedar-Cabelli S. and McCarty T. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389, San Francisco, 1987. Morgan Kaufmann.

[105] Liaw J. S. and Berger T. Dynamic synapse: A new concept of neural representation and computation. *Hippocampus*, 6:591-600, 1996.

[106] Livingstone M. S. Oscillatory firing and interneuronal correlations in squirrel monkey striate cortex. *J. Neurophysiology*, 75:2467-2485, 1996.

[107] Neuenschwander S., Engel A. K., König P., Singer W., and Varela F. J. Synchronization of neuronal responses in the optic tectum of awake pigeons. *Vis. Neuroscience*, 13:575-584, 1996.

[108] Stent G. S. A physiological mechanism for hebb's postulate of learning. *Proceedings of Nat. Acad. Sci. USA*, 70: 997-1001, 1973.

[109] Thorpe S., Fize D., and Marlot C. Speed of processing in the human visual system. *Nature*, 381:520-522, 1996.

[110] T. J. Sejnowski and C. R. Rosenberg. Nettalk: A parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, John Hopkins University, 1986.

[111] L. Shastri and V. Ajjanagadde. From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioral and Brain Sciences*, 16(3):417–451, 1991.

[112] Lokendra Shastri. Exploiting temporal binding to learn relational rules within a connectionist network. Technical report, ICSI, TR-97-003., 1997.

[113] Jude W. Shavlik and Geoffrey G. Towell. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1,2):119–165, 1994.

[114] Judd K. T. and Aihara K. Pulse propagation networks: A neural network model that uses temporal coding by action potentials. Neural Networks, 1993. 6:203-215.

[115] Gerald J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8, 1992.

[116] Srinivasan M. V. and Bernard G. D. A proposed mechanism for multiplication of neural signals. *Biol. Cybernetics*, 21:227-236, 1976.

[117] Tsodyks M. V. and Sejnowski T. Rapid rate switching in balanced cortical networks. *Network*, 6:111-124, 1995.

[118] Gerstner W. van Hemmen J. L. and Cowan J. D. What matters in neuronal locking. *Neural Computation*, 8:1689-1712, 1996.

[119] Christoph von der Malsburg. The correlation theory of brain function. Technical report, Max-Planck-Institute for Biophysical Chemistry, July 1981.

[120] Bialek W., Rieke F., de Ruyter van Stevenick R. R., and Warland D. Reading a neural code. *Science*, 1991. 252:1854-1857.

[121] Craven M. W. *Extracting Comprehensible Models from Trained Neural Networks*. PhD thesis, Computer Sciences Department, University of Wisconsin Madison, WI, 1996.

[122] Craven M. W and Shavlik J. W. Using sampling and queries to extract rules from trained neural networks. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 37–45, New Brunswick, N.J, 1994. Morgan Kaufmann.

[123] Gerstner W., Kempter R., van Hemmen J. L., and Wagner H. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 383:76-78, 1996.

[124] Gerstner W., Kempter R., van Hemmen J. L., and Wagner H. *Hebbian learning of pulse timing in the barn owl auditory system*, chapter 14. MIT press, 1998.

[125] Heiligenberg W. *Neural Nets in Electric Fish*. MIT Press, Cambridge, 1991.

[126] Kajer T. W., hertz J. A., and Richmond B. J. Decoding cortical neuronal signals: network models, information estimation and spatial tuning. *J. Computational Neuroscience*, 1:109-139, 1994.

[127] Singer W. *Models of Neural Networks 2.* Springer, Berlin, Heidelberg, NewYork, 1994. 141-173.

[128] Singer W. and Gray C. M. Visual feature integration and the temporal correlation hypothesis. *Ann. Rev. Neurosci.*, 18:555-586, 1995.

[129] A. H. Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1:39-46, 1989.

[130] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

Index