Petri Net Implementations of Neural Network Elements

by

Andrew Seely

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science

The Graduate School of Computer and Information Sciences
Nova Southeastern University

2002

We hereby certify that this thesis, submitted by Andrew Seely, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the thesis requirements for the degree of Master of Science.

Cannady, James, Ph.D. Date
Chairperson of Thesis Committee

Simco, Greg E. , Ph.D. Date
Thesis Committee Member

Mukherjee, Sumitra, Ph.D. Date
Thesis Committee Member

Approved:

Lieblein, Edward, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

The Graduate School of Computer and Information Sciences
Nova Southeastern University

2002

An Abstract of a Thesis Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Master of Science

Petri Net Implementations of Neural Network Elements

by
Andrew Seely

September 2002

This thesis presents methods for organizing Petri Net structures into threshold learning units and radial basis function Perceptrons to create a Petri Net Threshold Learning Unit (PNTLU) and a Petri Net Radial Basis Function Perceptron (PNRBFP). Background for Petri Nets, neural networks, threshold learning units, and radial basis functions is presented. The functionality of the PNTLU and the PNRBFP are shown to be equivalent to conceptual threshold learning units and radial basis function Perceptrons, respectively. The PNTLU and PNRBFP are compared to similar Petri Net technologies like the Fuzzy Petri Net and Fuzzy Neural Petri Net in terms of functionality and implementation. PNTLU and PNRBFP complexity and practical bounds are explored. Pseudo-code implementations of both are described.

Table of Contents

Abstract iii
List of Tables v
List of Figures vi

Chapters

1. **Introduction** 1
    - 1.1. Introduction to Petri Nets 2
    - 1.2. Introduction to Threshold Learning Units 9
    - 1.3. Introduction to Perceptrons and Radial Basis Functions 13

2. **Review of the Literature** 19
    - 2.1. An Overview of Fuzzy Petri Nets 19
    - 2.2. An Overview of Fuzzy Neural Petri Nets 21

3. **Methodology** 23
    - 3.1. The Petri Net Threshold Learning Unit 23
    - 3.2. Training the PNTLU 27
    - 3.3. The Petri Net Radial Basis Function Perceptron 28
    - 3.4. Training the PNRBFP 34

4. **Results** 35
    - 4.1. PNTLU by Example 35
    - 4.2. PNTLU Complexity 37
    - 4.3. PNRBFP by Example 38
    - 4.4. PNRBFP Complexity 43

5. **Conclusions, Implications, Recommendations, and Summary** 46
    - 5.1. PNTLU Special Cases 46
    - 5.2. PNRBFP Special Cases 48
    - 5.3. A Comparison of the PNTLU to the FPN and FNPN 49
    - 5.4. Conclusions 50
    - 5.5. Summary 51

Reference List 53

List of Tables

Tables

1. Formal Definition and State of Figure Six 7
2. TLU Sample Values 12
3. RBF Perceptron Sample Values 17
4. PNTLU Values in TLU Terms 25
5. PNTLU ΣW or θ Modification Algorithm 27
6. PNTLU Sample Values 36
7. FPN, FNPN, PNTLU Comparison 50

List of Figures

Figures

1. Petri Net Place 3
2. Petri Net Arc 3
3. Petri Net Place with Token 4
4. Petri Net Transition 5
5. Petri Net Negation Arc 6
6. Petri Net Graph 6
7. Conflicting Transitions 8
8. Linear Separation 10
9. TLU Weight 11
10. TLU Concentrator 11
11. TLU Threshold 12
12. Simple TLU 12
13. Basis Function 15
14. Fully Connected Perceptron 15
15. Gaussian Formula 16
16. Behavior of Gaussian RBF 17
17. RBF Perceptron Output 18
18. Fuzzy Petri Net 20
19. Fuzzy Neural Petri Net 22
20. Petri Net Threshold Learning Unit 24

List of Figures Continued

21. Formula to Find the Number of Processing Layer Places 26
22. Combinations Formula 26
23. RBF Transition 29
24. Concentrator Transition 29
25. Threshold Transition 30
26. PNRBFP with Token Zones and RBF Settings 31
27. PNRBFP with Synchronization 33
28. PNTLU Before and After First Firing 35
29. PNTLU with Activated Processing Layer Transitions 36
30. Upper Bound for PNTLU Computations 38
31. PNRBF Example One 39
32. PNRBF Example Two 40
33. PNRBF Example Three 41
34. PNRBF Example Four 42
35. PNRBF Example Five 43

Chapter 1

Introduction

This thesis introduces methods to use Petri Nets for building the basic neural network components Threshold Learning Unit (TLU), also known as an Adaline (Adaptive Linear Element), Perceptron, or Mculloch-Pits Neuron (Nilsson, 1998), and the radial basis function (RBF) Perceptron. These methods, hereafter called the Petri Net Threshold Learning Unit (PNTLU) and the Petri Net Radial Basis Function Perceptron (PNRBFP), show how Petri Nets can be arranged or modified to achieve functional equivalency with standard neural networking methods while maintaining the benefits of analysis (Murata, 1989) (Pastor, 2001) (Sivaraman, 1999), state modeling (Jensen, 1996) (Murata, 1989) (Peterson, 1977) and simplicity that are commonly associated with Petri Nets. Discussions of practical uses of the PNTLU and PNRBFP within more complex systems are beyond the scope of this thesis.

The content of this thesis is arranged as follows: Section 1.1 describes the basic definition, function, and behavior of Petri Nets. Section 1.2 describes the basic definition, function, and behavior of threshold learning units. Section 1.3 describes the basic definition, function, and behavior of radial basis functions in neural networks. Chapter 2 presents a survey of the academic literature that discusses Petri Net concepts in neural networking contexts. Chapter 3 introduces the PNTLU and PNRBF. Section 3.1 details the PNTLU concept and algorithm, showing how the Petri Net and TLU ideas can work together. A method for adjusting PNTLU attributes is demonstrated in section 3.2, showing that PNTLU and TLU behaviors are equivalent within certain data domain

restrictions. Section 3.3 introduces the PNRBFP, and section 3.4 explores training concepts for the PNRBFP. Chapter 4 assesses the validity and complexity of both the PNTLU and the PNRBFP. A PNTLU example is given in section 4.1 and section 4.3 presents an example of an PNRBF. Chapter 5 discusses the relationship between the different technologies and presents conclusions. Section 5.1 describes three special cases of PNTLUs that may be basis for further research. Section 5.2 discusses special cases for the PNRBFP. Section 5.3 presents a comparison of the PNTLU and PNRBFP to the similar technologies of the Fuzzy Petri Net and the Fuzzy Neural Petri Net. Object-oriented pseudo-code and Java source code implementations of a PNTLU and a PNRBFP are provided in the appendixes.

1.1. Introduction to Petri Nets

A Petri Net is a graph-based representational system that allows simple modeling of potentially complex systems, especially systems that have concurrent and interdependent features. (Peterson, 1981) While Petri Nets are naturally suited to work with system modeling, perhaps the most interesting feature about them is that their context is arbitrary. The system modeling structures are clean, simple, easy to understand, and allow intuitive graphic representation and straightforward programming (Jensen, 1996), yet Petri Nets may be used outside of systems modeling situations by simply changing the type of job for which they are employed. Just because Petri Nets are used extensively for systems modeling does not mean that they are not useful for other things.

A Petri Net is made up of four primitive elements: Place, arc, token, and transition. The Petri Net *place*, shown in Figure 1, is typically drawn as a circle and may be thought

of as a container for tokens. In simple Petri Nets the place does not do any processing or calculation, but its inherent properties cause it to act within certain bounds. A place can only hold *tokens*, which can only be added or removed by way of *arcs* or by the external environment. A place may hold zero or more tokens, and the number of tokens may have an upper bound. Properties of a place are limited to the current token count, token count upper bound, and the place identification label. The number of places in a Petri Net is *n*, and the set of all places for a Petri Net is labeled $p_{1..n}$. A Petri Net that imposes an upper bound of *k* tokens for each place is called a *k-bounded* Petri Net. A special case of k-bounded Petri Nets is a *safe net*, where k = 1. (Peterson, 1977)

Figure 1. Petri Net place

The Petri Net *arc*, shown in Figure 2, is typically drawn as an arrow and is the basic communication pathway for a Petri Net. Arcs are directional, with the direction indicated by the arrow's point, and may lead from a place to a *transition* or from a transition to a place. An arc may not be used to connect two places to each other nor may it connect two transitions to each other. Conceptually, an arc is able to transfer a single token at a time from a place to a transition or from a transition to a place. An arc does not do any inherent processing, and no other data transfer between primitive structures is possible within the Petri Net. In a Petri Net definition, arcs are referenced in terms of the transitions to which they are connected. (Peterson, 1977)

Figure 2. Petri Net arc

The Petri Net *token*, shown in Figure 3, is the sole data element of a Petri Net. In simple Petri Nets, the token's very existence is the only data carried. This allows Petri Nets to represent Boolean or integer values which can in turn be interpreted by the environment to have arbitrary meanings dependent on the context in which the Petri Net is employed. For example, a Petri Net used to model some sort of request system may consider each token to be a different request for some particular service, while a Petri Net used to measure the presence of data may consider a token to represent the fact that the required data is present while any extra tokens are treated as redundant reports of the same fact. Tokens only exist within places, and are added or removed from places via arcs or by outside influence from the environment. Individual tokens are not labeled and are only accessible as a property of a place. A Petri Net that contains one or more tokens is called a *marked* Petri Net. (Peterson, 1977) Tokens are represented as filled circles or dots inside a place.

Figure 3. Petri Net place with token

The Petri Net *transition*, shown in Figure 4, is the communications element of the Petri Net. A transition is connected to one or more places by one or more arcs. The set of arcs connected to a transition can be divided into the set of input arcs, or arcs that are directed from a place to the transition, and the set of output arcs, or arcs that are directed from the transition to a place. There is no theoretical upper limit to the number of arcs in either direction that may be connected to a transition.

Figure 4. Petri Net transition

A transition is *activated* when all of its input arcs connect to places that have at least one token. When a transition is activated it *fires*, removing one token from every place in the input set and adding one new token to every place in the output set. If the place receiving a new token is already at its *k* limit, the new token is discarded by the transition. While it is intuitive to describe the transition as 'moving' tokens, in practice a transition deletes tokens from input places and creates new tokens in output places (Jensen, 1996). While more than one transition in a Petri Net may be activated, the firing of transitions is never simultaneous nor must activated transitions be fired in any particular order. For this reason Petri Nets are considered *non-deterministic*, meaning that it is theoretically not possible to determine which of any simultaneously activated transitions may fire first (Peterson, 1977). In practical software applications, however, firing order is likely to be dictated by environmental factors that may be very predictable, such as the software program inspecting every transition in order and firing the first one found to be activated. A transition may have zero arcs for input but this configuration would serve no purpose as the transition would never fire. A transition may have zero output arcs; a configuration that would serve only to remove tokens from the Petri Net every time the transition fires. Transitions are not prioritized and so can potentially disable each other. The number of transitions in a Petri Net is *m*, and the set of all transitions in a Petri Net is labeled $t_{1..m}$.

A special type of arc called a *negation arc* is shown in Figure 5. The negation arc does not add or remove tokens from places, but it does prevent a transition from firing unless the place to which the arc connects is empty. In place of the arrow, a small circle where the arc connects to a place shows the negation requirement. The negation arc is considered an extension to the basic Petri Net. (Peterson, 1981)

Figure 5. Petri Net negation arc

A simple Petri Net graph is shown in Figure 6. The configuration of the Petri Net combined with the location of tokens in the net at any particular time is called the Petri Net *state*.

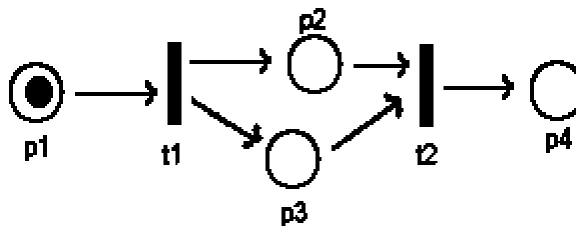


Figure 6. Petri Net graph

Petri Net structure is formally described by the five-tuple (P, T, I, O, M), where P is the set of places $\{p_1, .., p_n\}$, T is the set of transitions $\{t_1, .., t_m\}$, I is the set of places connected via arcs as inputs to transitions, O is the set of places connected via arcs as outputs from transitions, and M is the set of places that contain tokens (Ahson, 1995)

(Pastor, 2001) (Peterson, 1977). I, O, and M are *n*-tuples in their own right, and thus may be called *bags* instead of sets. (Peterson, 1981). I $(t_1, .. , t_m)$ is the set of input places for each transition. In Figure 6, for example, $I(t_1) = \{ p_1 \}$ and $I(t_2) = \{ p_2, p_3 \}$. $O(t_1, .., t_m)$ is the set of output places from each transition. In Figure 6, $O(t_1) = \{ p_2, p_3 \}$ and $O(t_2) = \{ p_4 \}$. $M(p_1, .., p_n)$ represents the *state* of the Petri Net and is the set of quantities of tokens per place, where the integer quantity of tokens *n* in a place is defined as $n \geq 0$. In Figure 5, $M(p_1) = 1$, $M(p_2) = 0$, $M(p_3) = 0$, $M(p_4) = 0$. M may also be written as $M = \{ 1, 0, 0, 0 \}$. The formal definition and state of the Petri Net in Figure 6 is shown in Table 1.

Table 1. Formal Definition and State of Figure Six

(P, T, I, O, M):	$P = \{ p_1, p_2, p_3, p_4 \}$
	$T = \{ t_1, t_2 \}$
	$I = \{ \{ p_1 \}, \{ p_2, p_3 \} \}$
	$O = \{ \{ p_2, p_3 \}, \{ p_4 \} \}$
	$M = \{ 1, 0, 0, 0 \}$

The concept of *environment* previously mentioned refers to the logical system outside of the Petri Net (Peterson, 1977) (Peterson, 1981). The environment can add and remove tokens from places but can not otherwise influence the Petri Net, i.e. it cannot cause an un-activated transition to fire. In practice, specific places may be arbitrarily designated as input places to be used as 'sensors' to the environment. Specific places may be arbitrarily designated as output places to be used by the environment to collect any results provided by the Petri Net. When used in this way, the Petri Net can be seen as a simple processing mechanism with discrete input and potentially unique output based on the internal configuration of places, transitions, and arcs. The environment, in this case,

could be a computer program or other complex system. It is important to note that the Petri Net is not inherently a processing system; any 'results' from a Petri Net are determined by subjective value assignment through outside observation.

Petri Nets can have configurations that cause three erroneous conditions. A transition that will never become activated given a particular marking M is considered *dead*. A place that will never receive a token given an initial marking M is considered *unreachable*. These two conditions are related: dead transitions may exist because of unreachable places on a marking's I(*t*) set, and places on a dead transition's O(*t*) set are automatically unreachable if they are not on the O(*t*) set of another transition. When two transitions are activated and the firing of one de-activates the other, they are said to be *conflicting*. (Jensen, 1996) (Peterson, 1977) (Sivaraman, 1999) Figure 7 shows a Petri Net with conflicting transitions.

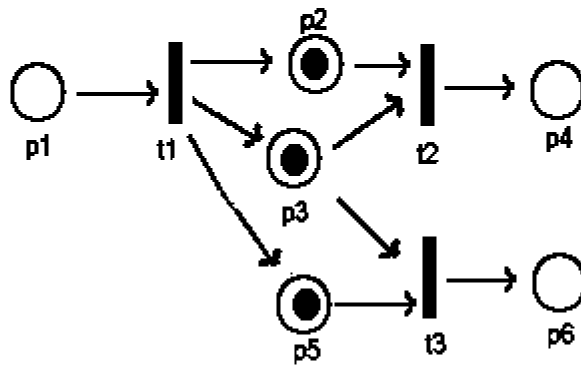


Figure 7. Conflicting transitions (t_2 and t_3)

In Figure 7, both t_1 and t_2 are activated. When this Petri Net allows a firing, either transition may fire and remove the tokens on its inputs. In either case the token at p_3 is removed, causing the transition that did not fire to lose its activation.

1.2. Introduction to Threshold Learning Units

A TLU is a basic building block for simple neural networks. While there are more advanced ways to build a neural network (Dean, 1995) (Gurney, 1997) (Nilsson, 1998) (Principe, 2000), no other method is as simple to implement or as easy to understand as the TLU. A TLU consists of a set of inputs, a set of adjustable weights that influence the inputs, a concentrator, a threshold, and an output (Principe, 2000) (Turban, 1998). In its basic form, a TLU is capable of accepting a set of real, integer, or Boolean values as input and creating a single Boolean value for output. This behavior limits a single TLU to a class of problems known as *linearly separable* problems, where a straight line may be drawn through a graph of the problem's domain to separate those values that would be designated false by the TLU from those values that would be designated true (Gurney, 1997). The threshold by which the TLU output is determined is designated θ. θ is a real number and may be adjusted to change the behavior of the TLU.

Linearly separable problems are problems of simple separation (Gurney, 1997). For example, selecting a male or a female from a group of people is linearly separable because the people can be arranged such that a straight rope separates men from women. Selecting all the men with children from the same group is not a linearly separable problem; the rope can either separate men from women or those people with children from those without, but in order to separate a group from a group it must have a bend. In numeric terms, all potential output values are plotted on an x-y coordinate graph. The problem represented by the potential output graph is linearly separable if and only if the set of answers that are true may be separated from the set of answers that are false by a

straight line that may be represented with the formula Y = mX + b. Figure 8 demonstrates linearly separable problems.

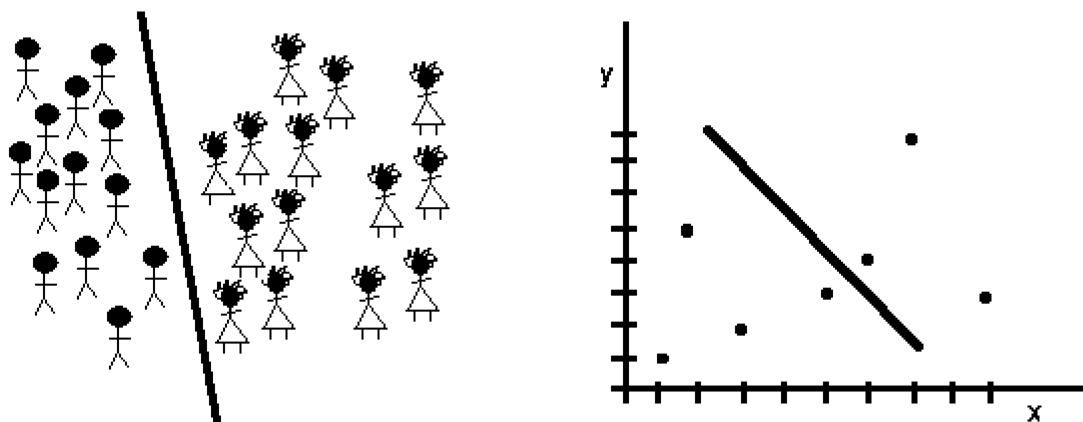


Figure 8a. Linear separation is possible when a graph of possible answers can be divided with a straight line. Choosing a man or a woman from a group of people or all points to one side of a line Y=-1¼X + 10 is a linearly separable problem.

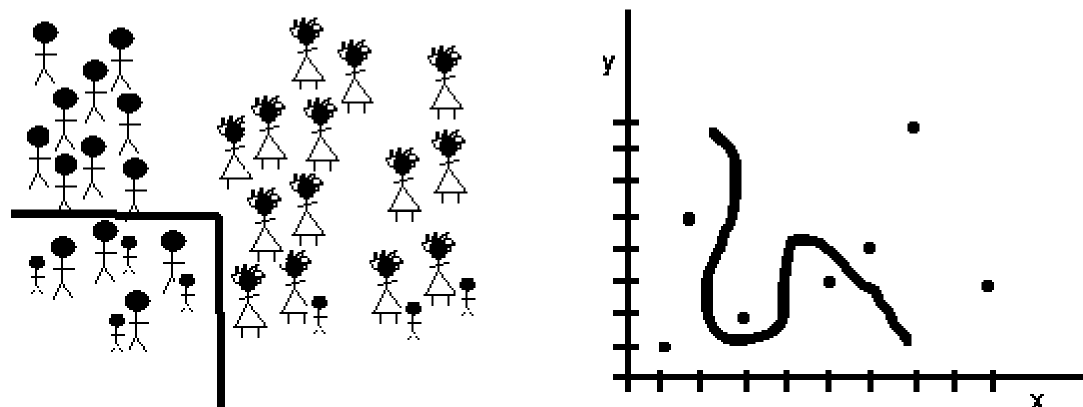


Figure 8a. Linear separation is not possible when making a division of a division. Choosing a man with a child from a group of men and women with and without children or all points to one side of a non-linear equation is not a linearly separable problem.

While inputs to a TLU may be any value from the set of real numbers, this paper will limit TLU inputs to Boolean values to allow a cleaner comparison with Petri Nets. A TLU will have n input values, where $n > 0$. There is no theoretical upper bound on n.

The set of inputs to a TLU is the ordered set $X = \{ x_1, .., x_n \}$. Input values can be derived from the external environment, from other areas of the neural network, or from other TLUs in the same system.

Each input in X has a corresponding numeric weight value in the set $W = \{ w_1, .., w_n \}$. Weights are individually adjustable and may be any value from the set of real numbers. The symbol used in this thesis for graphically representing a weight is shown in Figure 9. Weights are multiplied with input values to create the set of inputs presented to the concentrator: $X \bullet W = \{ x_1w_1, .., x_nw_n \}$.

Figure 9. TLU weight.

The concentrator, shown in Figure 10, serves to sum the values from the input/weight product set. The output of the concentrator is a single real number $\sum_{i=1}^{n} x_i w_i$, which is presented to the threshold.

Figure 10. TLU concentrator.

The TLU threshold, shown in Figure 11, accepts the concentrator value and performs a simple *greater-than/equal-to* calculation. If the concentrator's value is greater than or

equal to θ then the threshold will output a *true* value for the whole TLU, otherwise the threshold will output a *false* value.

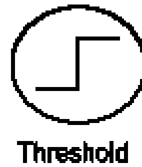


Figure 11. TLU threshold.

A simple TLU is shown in Figure 12. This TLU has an input set X = { x_1, x_2, x_3 } and a weight set W = { w_1, w_2, w_3 }. To illustrate the function of the TLU, let us assign arbitrary values to its attributes.

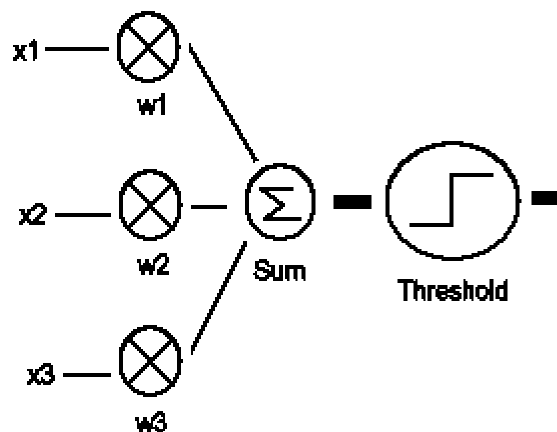


Figure 12. Simple TLU.

In the example in Table 2, the product X • W = { 3, 0, 2 }. The concentrator sums the values 3 + 0 + 2 = 5, which is less than θ = 6. The TLU output is *false*.

Table 2. TLU Sample Values

X = { 1, 0, 1 }
W = { 3, 1, 2 }
θ = 6

θ and W are adjustable to modify the behavior of the TLU. *Supervised learning* is the process of manual adjustment of θ and W to achieve expected output based on a known input (Gurney, 1997) (Turban, 1998). In the example above, say that the given X should have resulted in a TLU output of *true*. A simple reduction of θ by 1 would bring the TLU performance in line with expectations for this X. Or say that the output should have been *true* and that input x_1 is determined to be relatively more valuable than the other inputs. An increase of w_1 to 4 and no adjustment to θ would also correct the TLU's output by increasing the concentrator value. Specific algorithms and techniques to decide the magnitude of adjustment are beyond the scope of this thesis, but are discussed in detail in the references (Dean, 1995) (Gurney, 1997) (Principe, 2000).

1.3. Introduction to Perceptrons and Radial Basis Functions

The Perceptron is a neural network processing element that is significantly more complex and powerful than the TLU. While both the TLU and the Perceptron may act as classifiers of their input, the Perceptron adds extra processing elements to its structure and is able to classify input on a much more complicated decision space. The most significant difference between the structure of the Perceptron and the TLU is the addition of a *basis function* before each weight. (Gurney, 1997) (Orr, 1996)

The *decision space* in which a processing element like the Perceptron or TLU operates consists of the complete range of possible inputs and the designation of those inputs into values that are part of the classification and those that are not. In section 1.2, Figure 8 demonstrates a decision space in both linearly separable and non-linearly separable forms. In two dimensions, a linearly separable decision space is classified by a straight

line. In more than two dimensions, a *hyperplane* classifies the multi-dimensional decision space. (Nilsson, 1998) For example, a straight line may classify a decision space on a piece of paper, but a plane is required to classify a decision space shaped like a cube. Section 1.2 defined the TLU as a processing element capable of classification in a linearly separable decision space; classification in more than two dimensions merely requires the addition of separate TLUs for each dimension. This is the problem domain of the TLU.

The problem domain of the Perceptron includes that of the TLU, but also includes classification problems where the decision space is not definable as a hyperplane. The Perceptron can be used for two types of neural networking problems, function approximation and classification. (Orr, 1996) (Poggio, 1989) This thesis will introduce the Perceptron as used for classification problems where the decision space is non-linear.

The Perceptron accepts *n* inputs, each of which may be a real number. The Perceptron's internal basis function/weight node output may also be a real number, allowing the output of one Perceptron node to feed an input of another. A series of Perceptrons nodes connected in this way form a multilayer Perceptron (MLP), a processing unit that has been proven to be able to approximate any function. (Gurney, 1997) (Principe, 2000)

A Perceptron adds a basis function h(x) before each weight. Figure 13 shows the graphical representation of the basis function used in this thesis.

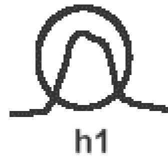


Figure 13. Basis function symbol

Another feature that differentiates a Perceptron from a TLU is the degree of interconnectivity between the inputs and the inner structures. While a TLU may directly connect inputs to weights, the Perceptron typically, but not as a rule, connects all inputs to all basis functions; each individual basis function produces its output based on the sum of all connected inputs. A fully connected Perceptron is shown in Figure 14.

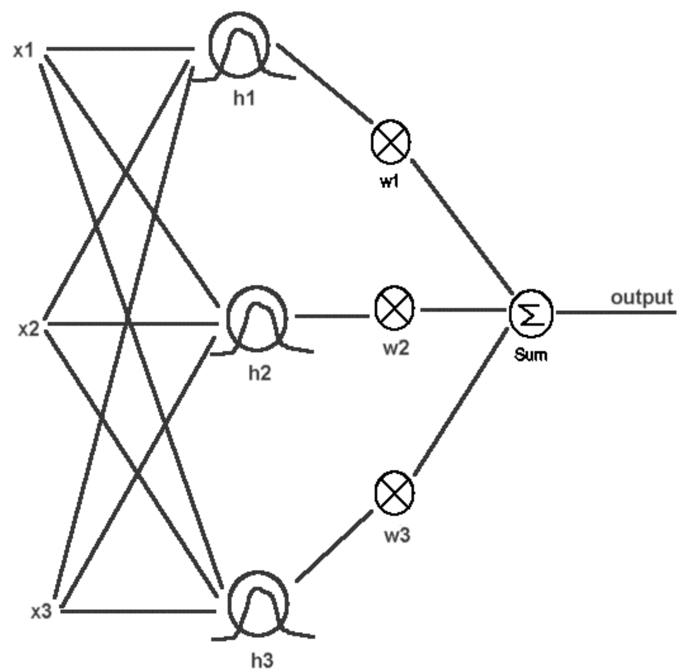


Figure 14. Fully connected Perceptron

The Perceptron in Figure 14 has an input set X = { x_1, x_2, x_3 }, a weight set W = { w_1, w_2, w_3 }, and basis functions H = { $h_1(x)$, $h_2(x)$, $h_3(x)$ }. The choice of basis function will

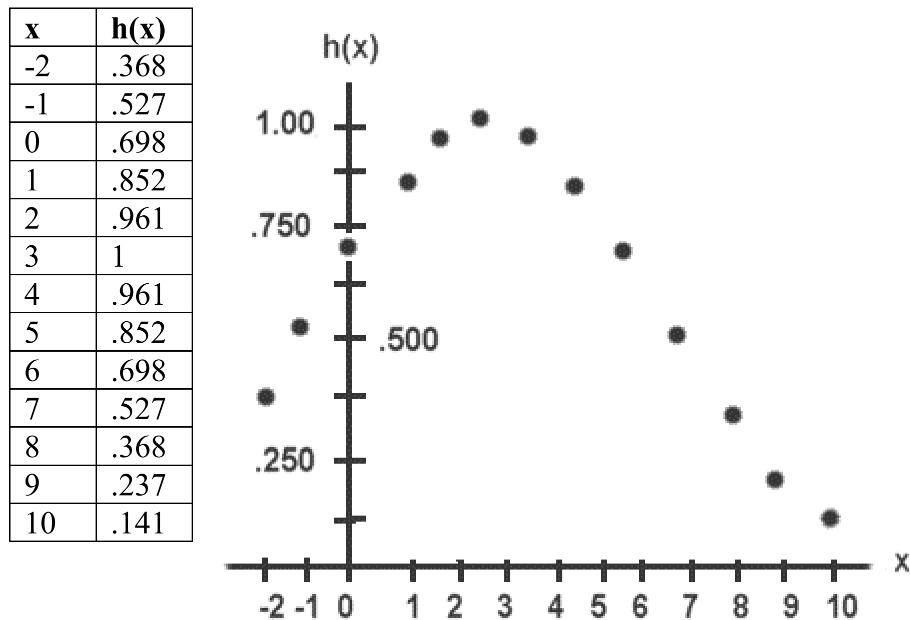
radically alter the behavior of the Perceptron. Assume a Perceptron with direct (x_n only connects to h_n) connections and a basis function f(x) = x. This Perceptron would behave exactly as the TLU from section 1.2; the basis functions would have no effect. Consider a directly connected Perceptron with a basis function f(x) = -x. This Perceptron would effectively invert the behavior, producing output opposite to that of the TLU.

The Perceptron used here will employ a class of basis functions known as Radial Basis Functions (RBF). An RBF produces continuous rather than linear output, and in particular an RBF's output changes with respect to distance from the RBF's *center*, thus the 'radial' aspect of the basis function. (Orr, 1996) One of the most common RBF functions used in neural networking is the Gaussian function shown in Figure 15. This function is well known in the study of statistics as the normal curve. (Walpole, 1993)

$$h(x) = \frac{1}{e^{\frac{(x-c)^2}{r^2}}}$$

Figure 15. Gaussian function

In Figure 15, *c* is the center of the function and *r* is the radius or effective width of the function. Whereas in the TLU, input is merely multiplied by weight to produce a linear output, when the Gaussian RBF is employed the output is greatest when the input is close to *c*, and quickly drops off as it approaches *r* distance from *c*. The example in Figure 16 demonstrates the behavior of the Gaussian RBF.



x	h(x)
-2	.368
-1	.527
0	.698
1	.852
2	.961
3	1
4	.961
5	.852
6	.698
7	.527
8	.368
9	.237
10	.141

Figure 16. Behavior of Gaussian RBF with r=5, c=3, x = { -2 .. 10 }

Notice that all inputs within *r* distance of *c* produce outputs above .5, with the strongest outputs being produced based on the inputs that are closest to c. The use of an RBF in the Perceptron adds two new points of adjustment. Each *w* in W can still be modified, but now too each *r* and *c* of each *h* in H may also be modified to adjust the position and sensitivity of each *h*'s contribution to the final output of the Perceptron. (Gurney, 1997) (Principe, 2000)

To illustrate the function of the RBF in the Perceptron context, let us assign values as we did for the TLU in section 1.2. All basis functions are Gaussian functions with attributes defined as (r, c).

Table 3. RBF Perceptron Sample Values

X = { 1, 0, 1 } W = { 3, 1, 2 } H = { (5,3), (7,2), (9,9) }

In the example in Table 3, all *x* in X are summed with the result provided as input to each *h* in H. Each *h* therefore receives $\Sigma X = 1 + 0 + 1 = 2$ as input. $h_1(2) = .960$, $h_2(2) = 1$, $h_3(2) = .237$. Each basis function result is then multiplied against its weight: $h_1(2) \times w_1 = 2.88$, $h_2(2) \times w_2 = 1$, $h_3(2) \times w_3 = .474$. The final output of the Perceptron is the summation of these products: 2.88 + 1 + .474 = 4.354. If this Perceptron were being used in a multilayer network, this value would then become the input to another basis function. If this Perceptron where the last or only processing element in the network, its value could then be compared with a threshold to produce a final Boolean answer. The steps shown in this equation are summarized in the formula show in Figure 17.

$$\sum_{1}^{n} \left[h_n \left(\sum_{1}^{m} x_m \right) w_n \right]$$

Figure 17. RBF Perceptron Output

Supervised learning for the RBF Perceptron is conceptually the same as for the TLU. Adjustments to W and to *i* and *c* for each individual *h* in H may be accomplished to change the behavior of the Perceptron based on expected output to known input. Methods for determining which attributes to adjust in what quantities are beyond the scope of this thesis, but are discussed in detail in the references. (Bors, 2001) (Broomhead, 1988) (Dean, 1995) (Orr, 1996) (Principe, 2000)

Chapter 2

Review of the Literature

This chapter presents the state of research into creating neural networking structures with Petri Nets. A related field of work, employing Petri Nets to process fuzzy logic problems, is also addressed. Despite the large body of work available with respect to both Petri Net and neural network theory, surprisingly little work has been documented using Petri Nets in neural networking contexts.

2.1. An Overview of Fuzzy Petri Nets

The Fuzzy Petri Net (FPN) is a Petri Net that has been modified to allow processing of "fuzzy" truth values. With arbitrary places designated as input and output interfaces to the environment, an FPN is able to accept tokens that represent truth values that are *true* (1), *false* (0), or some "fuzzy" measure n in between true and false ($0 \leq n \leq 1$). The output of a FPN is itself a fuzzy value that may in turn be fed to other FPNs. (Looney, 1988) In an essential deviation from the definition of simple Petri Nets, the token used in the FPN is actually a data carrier.

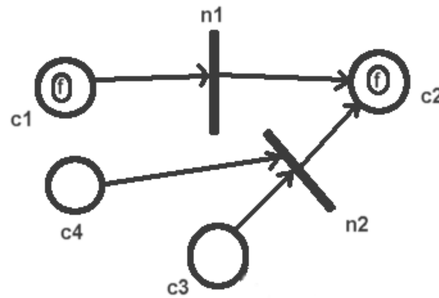


Figure 18. Fuzzy Petri Net, with $C_{1..4}$ places representing conditions, $N_{1..2}$ representing transitions that fire based on the relative *truth* in the attached C places, and tokens that carry the relative fuzzy truth values that are assessed by the N transitions. (Looney, 1988)

Fuzzy systems are made up of fuzzy truth-value propositions that represent the degree of surety or belief in the event or idea they represent combined with rules that determine a course of action based on the propositions present. For example, a proposition representing how hot a summer day feels could be combined with a rule describing how long you will stay at work today to make a decision to turn on the air conditioning. In the FPN, tokens are used to represent propositions to be evaluated. Rules are implemented as transitions in the FPN, which serve as limiters of the "strength of truth" that may pass through them. FPN transitions have a *valve setting* to limit the maximum value that may pass and they have a *threshold setting* that serves to limit the minimum value a fuzzy truth must have in order to pass. Thus, FPN transitions begin to fire when all their input places have tokens, but the transitions only complete the firing process after some amount of comparison processing has been accomplished. (Chen, 1990) (Cherniaev, 1995) (Looney, 1988)

The FPN is not intended to act as a learning tool, but rather it serves to process uncertain data and provide results between *true* and *false* that may be further interpreted by external elements of a system. The difference between the FPN and the TLU is in

what changes during the lifetime of the processing element. The TLU changes its internal settings to accommodate given inputs in order to provide desired outputs. The FPN expects the input to change in respect to the FPN's output. The problem domain of the FPN includes decision-making situations where input and output may be variable within a gradient yet the decision-making process remains relatively static. The input and output domain of the FPN is the set of real numbers n, where $0 \leq n \leq 1$.

2.1. An Overview of Fuzzy Neural Petri Nets

The Fuzzy Neural Petri Net (FNPN) extends the function of the Fuzzy Petri Net by adding additional properties to the Petri Net primitive structures. The most significant changes introduced by the FNPN are the TLU concepts of weights and thresholds. The FNPN restricts input places to each have a single arc to the input of a single transition, which then has a single output arc to a special common "concentrator place." This concentrator place has a single output arc which serves as a threshold. The threshold arc connects to a single transition that provides output to the environment via an arbitrary number of places on its output arcs. An FNPN may have multiple concentrator places that may be interconnected to various input weight transitions, and output threshold arcs may be interconnected to various output places. (Ahson, 1995)

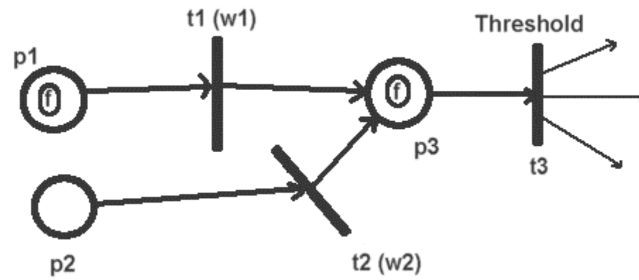


Figure 19. Fuzzy Neural Petri Net, with $t_{1..2}$ weight transitions, t_3 thresholding transition, $p_{1..3}$ condition places, and 'fuzzy' tokens (Ahson, 1995)

The FNPN allows the processing of fuzzy truth values presented as tokens in the same manner as the FPN, but the FNPN also allows a degree of TLU-style learning to be accomplished. The FNPN input weight transitions allow different inputs to be given ranked influence on the final outcome just as in a TLU. The output threshold acts to impose a binary order on the fuzzy truth values being processed: While the output may consist of a number of tokens each having a truth value $\{ n \mid 0 \leq n \leq 1 \}$, if there is not a sufficient quantity of these tokens at the concentrator place the FNPN will not deliver any value greater than 0 to the environment. (Ahson, 1995)

The FNPN functions as a TLU through supervised learning. Learning is accomplished through the comparison of the FNPN behavior to known input-output sets and the subsequent adjustment of the weight values for each input and the threshold value at the concentrator output. In this respect, the FNPN functions basically equivalent to the TLU, but with greater abstract representation power via the fuzzy truth representation and lessened neural network applicability through the limitations of the input domain. The problem domain of the FNPN is that of the FPN, but where the decision-making process itself must be adaptable to new situations.

Chapter 3

Methodology

This chapter introduces the Petri Net Threshold Learning Unit (PNTLU) and the Petri Net Radial Basis Function Perceptron (PNRBFP). Training concepts are discussed for both techniques and parallel examples are offered with the TLU of section 1.2 and the RBF Perceptron of section 1.3 to demonstrate the equivalent behaviors of the TLU to the PNTLU and the RBF Perceptron to the PNRBFP.

3.1. The Petri Net Threshold Learning Unit

As shown in section 1.2, a TLU is a relatively simple idea but is useful only in a neural network setting. Petri Nets have the advantage of being generalist tools; a Petri Net can be used to model a system, to do simple processing, to show relationships between concurrent systems, and to demonstrate parallel processing interactions (Murata, 1989) (Peterson, 1977). While TLU networks are parallel in nature, i.e. two TLUs in the same neural network can operate independently of each other (Gurney, 1997), Petri Net transitions are just as parallel but without the added complexity of W and θ adjustments. Both the Petri Net and the TLU can easily work with Boolean values, but the Petri Net is less able to handle real or integer numbers. This section presents a method for implementing the relatively special-purpose TLU with a relatively general-purpose Petri Net.

The PNTLU is a Petri Net that simulates the features of a TLU by varying the number of places and transitions and the degree of interconnectivity between them. Adjusting an

input weight or θ is done by reorganizing the Petri Net arcs to accommodate the changing quantities of places and transitions. For discussion purposes, it is useful to describe the PNTLU in terms of three layers: The Input Layer contains the 'input' places and 'weight' transitions; the Processing Layer contains 'weight' places and 'threshold' transitions; the Output Layer contains the 'output' place.

Figure 20 shows an example of a PNTLU. This PNTLU may be compared to the TLU in section 3.2: The input set $X \approx M(p_1, p_2, p_3)$; the weight set $W = \{ w_1, w_2, w_3 \} \approx O(t1, t2, t3) = \{ 3, 1, 2 \}$; θ is 3. These attributes are artificially assigned and only have meaning in this context; in all respects, Figure 20 is a pure Petri Net as defined in section 1.1. The following discussion demonstrates the functionality of a PNTLU using Figure 20 for reference.

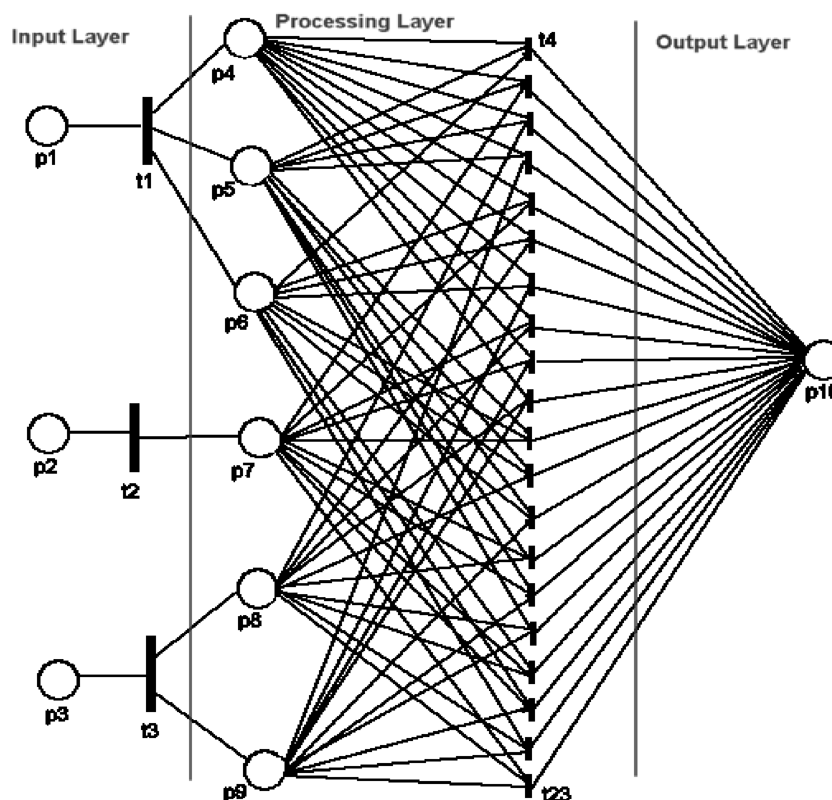


Figure 20. Petri Net Threshold Learning Unit with $W = \{ 3, 1, 2 \}$, $\theta = 3$.

The PNTLU logically functions in the exact same manner as the TLU from section 1.2. The TLU input set X = { 1, 0, 1 } is represented in the PNTLU by initial state markings $P_{1\text{-}10}$ = { 1, 0, 1, 0, 0, 0, 0, 0, 0, 0 }. In this example, a token present in $p_{1\text{-}3}$ is equivalent to a *true* input value in $X_{1\text{-}3}$. Places $p_{1\text{-}3}$ are designated as the input and p_{10} is the output and will be written p_{output} for clarity.

The set W of the TLU is represented by the relationship between $t_{1\text{-}3}$ and $p_{4\text{-}9}$. The presence of an input token on any of $p_{1\text{-}3}$ will activate the transition that satisfies the value $I(t_n) = p_n$. This transition in turn places tokens in its output places. The number of places connected to the output of each transition at this layer is analogous to the weight value for that input. Adjusting the weight associated with an input p_n is simply a matter of increasing or decreasing the number of places found in the set $O(t_n)$. Table 4 demonstrates the equivalency between the TLU and the PNTLU in Figure 20.

Table 4. PNTLU Values in TLU Terms, Using Figure 20 for Reference

$I(t_1) = \{ p_1 \}$ and $O(t_1) = \{ p_4, p_5, p_6 \}$. In TLU terms, $x_1 \approx M(p_1)$, $w_1 = 3$.
$I(t_2) = \{ p_2 \}$ and $O(t_2) = \{ p_7 \}$. In TLU terms, $x_2 \approx M(p_2)$, $w_2 = 1$.
$I(t_3) = \{ p_3 \}$ and $O(t_3) = \{ p_8, p_9 \}$. In TLU terms, $x_3 \approx M(p_3)$, $w_3 = 2$.

In Figure 20, the Processing Layer places $p_{4\text{-}9}$ are interconnected to the Processing Layer transitions $t_{4\text{-}23}$. The number of transitions at this layer and the degree of interconnectivity are determined by the threshold θ. In this example, θ = 3. Each Processing Layer transition *t* has θ = 3 places in its I(*t*) input set. The degree of interconnectivity is determined by the number of mathematical combinations of size θ that are possible from the number of places in the Processing Layer. The number of

Processing Layer places may be found by summing the number of output arcs from the Input Layer transitions, as shown in the formula in Figure 21.

$$\Sigma W = \sum_{i=1}^{3} O(t_i)$$

Figure 21. Finding number of Processing Layer places

The number of Processing Layer transitions *R* is defined by the combinations formula shown in Figure 22.

$$R = C(p, q) = \frac{p!}{q!(p-q)!} \quad \text{(Balakrishnan 1991)}$$

Figure 22. Finding the number of Processing Layer transitions, where p is the number of Input Layer transitions and q is θ

For this example, $R = C(\sum_{i=1}^{3} O(t_i), \theta) = C(6, 3) = 20$ Processing Layer transitions. For each transition in R there is one output arc: $O(p_{output})$.

The logical flow of the PNTLU is as follows: Boolean input values are received from the environment or from another PNTLU in the form of tokens (*true*) or no tokens (*false*). A particular input is given an integer weight of 1 for no change or greater than 1 to increase the input's influence on the PNTLU output in proportion to the other inputs; weights are represented by the number of places in the output O(*t*) set of the input's transition. Since every transition in R is connected to θ Processing Layer places and since R includes every possible combination of θ Processing Layer places, it is true that

any occurrence of θ tokens in the Processing Layer places will cause at least one Processing Layer transition to fire. Since every Processing Layer transition has only $O(p_{output})$, p_{output} will receive a token representing a PNTLU output of *true* whenever any transition in R fires.

3.2. Training the PNTLU

Section 1.2 discussed *training* the TLU by adjusting weights and/or θ. Conceptually, this process is exactly the same for the PNTLU. In the PNTLU, changing ΣW or θ in the governing formula C(ΣW, θ) will have the effect of increasing weights or the threshold for the overall PNTLU, respectively. Changing weights for a single input is a matter of changing the O(*t*) set for that input transition to properly reflect the input's proportion to the new ΣW value.

Modification of ΣW or θ requires a "re-wiring" of the PNTLU between the Processing Layer places and p_{output}. This may be accomplished by the algorithm shown in Table 6.

Table 5. PNTLU ΣW or θ Modification Algorithm

1. Modify θ, ΣW to bring the PNTLU behavior into tolerance
2. Recalculate O(*t*) for all transitions at the Input Layer
3. Remove all transitions at the Processing Layer
4. Remove all places at the Processing Layer
5. Create ΣW places at the Processing Layer and connect them to satisfy the new O(*t*)
6. Create R = C(ΣW, θ) transitions and set O(*t*) to be p_{output} for each transition *t*
7. Derive the set of combinations of size θ that can be made from the set W; for each combination, set the combination members to I(*t*) and move to the next t in the new Processing Layer transition set R.

3.3. The Petri Net Radial Basis Function Perceptron

The PNTLU adheres to a strict enforcement of Petri Net properties to take full advantage of the power of the inherent simplicity of Petri Nets. This enforcement limits the problem domain of the PNTLU to that of the TLU; a situation that is acceptable, but does not provide a very useful technology for practical neural networking work. This section introduces the Petri Net Radial Basis Function Perceptron (PNRBFP), a Petri Net-based Perceptron that mirrors the functionality of the Perceptron described in section 1.3. The PNRBFP breaks from the purist view taken by the PNTLU, following the lead of the FPN and FNPN reviewed in sections 2.1 and 2.2 to take advantage of modifications to the Petri Net primitives. These modifications allow the PNRBFP to maintain basic Petri Net structure while increasing the inherent processing power of the structure to that of the RBF Perceptron.

The transitions used in the PNRBFP are of four different types. Unmodified Petri Net transitions are used alongside three special-purpose transitions: The RBF transition, the concentrator transition, and the threshold transition. The RBF transition, shown in Figure 23, requires a single place connected on a single input arc. When activated, the RBF transition will fire until this place is empty. Internally, a 'token counter' will record the number of tokens removed from the input place. The pre-set RBF function built into the RBF transition will then accept this token count as its input. The output of the internal RBF function will be calculated; a new, non-primitive token called a *stage one token* will be created in the single output place with this new value. The RBF transition defined here uses the Gaussian function and has adjustable attributes for the Gaussian radius and center.

Figure 23. RBF transition

The concentrator transition requires a single input arc, just like the RBF transition, but each token in its input place is expected to be a stage one token. The concentrator place removes all stage one tokens from its input place and internally sums their values. A single *stage two token* with this value is created in the concentrator's single output place. Figure 24 shows a concentrator transition.

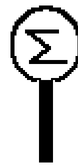


Figure 24. Concentrator transition

The threshold transition requires a single input place, which should only hold one stage two token. The threshold transition has a single attribute θ, equivalent to θ in the TLU. If the stage two token value is greater than or equal to θ, the threshold transition outputs a single primitive token to its output place or places to represent a *true* value result. The threshold transition in the PNRBFP behaves in a fashion similar to the FNPN thresholding transition described by Ahson (1995). Figure 25 shows a threshold transition.

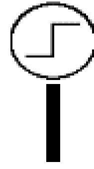


Figure 25. Threshold transition

Places in the PNRBFP are ordinary primitive Petri Net places. Tokens are of three varieties: Primitive, as described in section 1.1, stage one, and stage two. The difference between the three types of PNRBFP tokens is a matter of value. A primitive token carries no value other than its presence and is therefore limited to the domain of true (present) and false (absent). A stage one token can carry a real number value *n,* where $1 \geq n \geq 0$, in the same way that the tokens in both the FPN and FNPN carry fuzzy truth values. A stage two token can carry any real number *n*, where $n \geq 0$. Conceptually, the purposes of the two types of value-carrying tokens are distinct. The stage one token exists to transmit the output of the RBF transition to the concentrator. The stage two token serves to carry the concentrator's output value, which is the summation of all stage one tokens and thus could potentially be very large.

Figure 26 shows a fully connected PNRBFP. This PNRBFP is functionally equivalent to the RBF Perceptron described in section 1.3 and Table 3. The three X inputs are $p_{1..3}$, the weights W are represented by the number of arcs leading into p_{10}: $W = \{ w_1, w_2, w_2 \} =\sim \{ O(t_9), O(t_8), O(t_7) \} = \{ 3, 1, 2 \}$. For purposes of this discussion, RBF transitions $t_{4..6}$ will have their radius and center values set as in Table 3: $H = \{ (5,3), (7,2), (9,9) \}$.

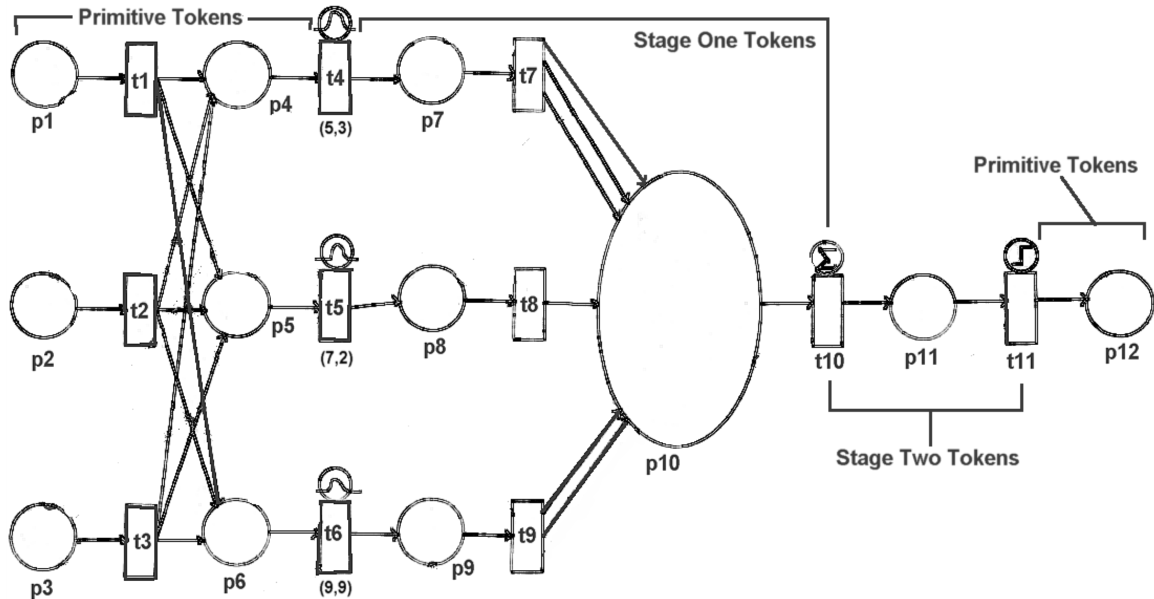


Figure 26. PNRBFP showing token zones and RBF settings

Logically, the PNRBFP behaves exactly like the RBF Perceptron. Boolean or integer inputs are entered in the input places as primitive tokens. In the case of Figure 26 the input places are $p_{1..3}$. Integer inputs are represented by multiple tokens in a place. The primitive transitions $t_{1..3}$ read the number of tokens in each input place and populate the next layer of places with these tokens. This first layer of transitions can be considered the point of interconnection, distributing the input across all basis functions as demonstrated in Figure 14. It follows that less-than-full interconnectivity is achieved by simply removing select output arcs from these transitions.

After all inputs have been distributed to the intermediate places, the RBF transitions read the number of tokens in their respective intermediate places, introduce this count to their individual basis functions, and create the stage one token with the RBF result in their respective output places. The stage one token is then removed by a primitive transition that acts as a weight by multiplying the stage one token's overall influence with

multiple output arcs. Weights are limited to integer values; a weight of three, for example, is achieved by connecting three output arcs from the transition as with t_7 in Figure 26.

All weight-enacting transitions connect their output arcs to a single, common place. It is important to remember that this place is expected to contain only stage one tokens. The concentrator transition reads the aggregate collection of stage one tokens and computes the sum of all their values. This sum is then used to create a stage two token, which is then deposited in a single output place. The stage two token is read by the threshold transition; if the stage two value is greater than or equal to the threshold transition attribute θ, a primitive token is placed in the outputs of the threshold transition to represent a *true* processing result.

PNRBFP processing requires synchronization between its stages in order to accurately process its data. For example, the non-deterministic nature of Petri Nets would allow t_4 in Figure 26 to fire and process input to its RBF before $t_{1..3}$ were finished transferring their input values to the place at $I(t_4)$. A very simple solution exists to control the PNRBFP synchronization problem. Addition of negation arcs, as defined in section 1.1, prevent transitions at each processing point from firing until their predecessor transitions complete their firing cycles. Figure 27 shows the PNRBFP from Figure 26 with synchronization control negation arcs in red.

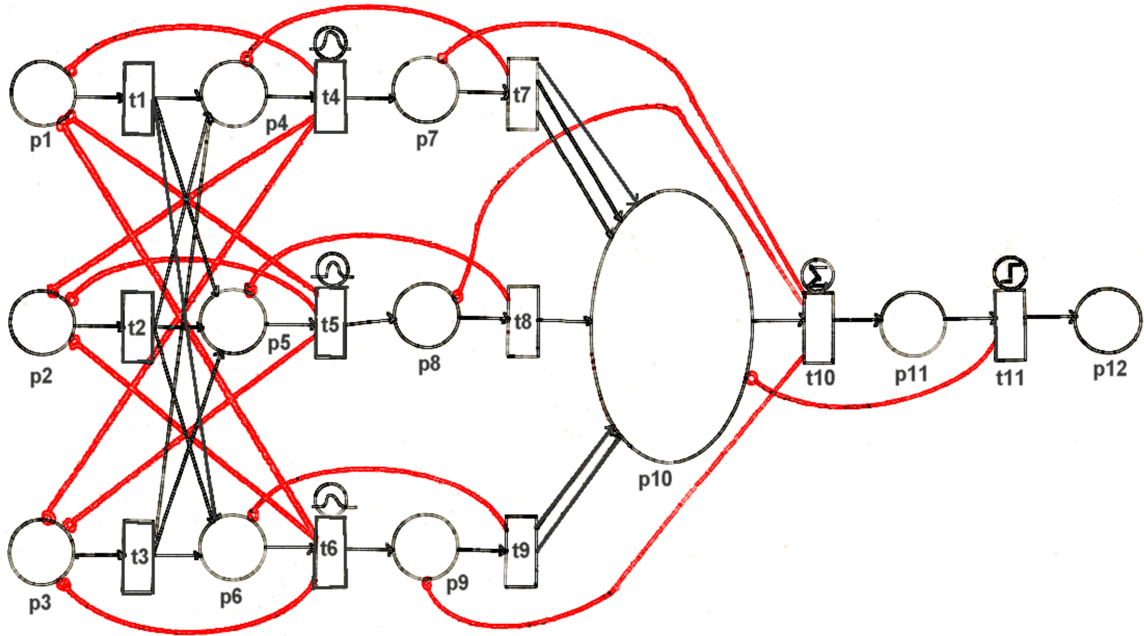


Figure 27. PNRBFP with synchronization

The negation arcs ensure synchronization by preventing premature transition firing. Since a transition is considered enabled whenever all places on the transitions I(t) set have tokens, the negation arcs ensure that the transition will not fire until the I(t) places of its predecessors have been fully processed. In context of the above discussion of PNRBFP flow, we can see that the RBF transitions $t_{4..6}$ will not fire until the input places $p_{1..3}$ have been completely emptied by transitions $t_{1..3}$. If $p_{1..3}$ are empty, it is assured that places $p_{4..6}$ have been populated since transition firing is logically a non-divisible event.

Similarly, transitions $t_{7..9}$ will not fire until $p_{4..6}$ are empty, respectively. This prevents the weight-enacting transitions from prematurely accepting RBF transition output. The concentrator transition at t_{10} will not fire until places $p_{7..9}$ are empty, assuring a full aggregation of stage one tokens in place p_{10}. Finally, the threshold transition at t_{11} will not fire until p_{10} is empty, enforcing a complete summary from the concentrator

transition. For purposes of this thesis, the PNRBFP with synchronization control shown in Figure 27 will be considered the standard PNRBFP.

3.4. Training the PNRBFP

While the PNRBFP is more complex in its construction than the PNTLU, adjusting the behavior of the PNRBFP is much easier. The PNTLU requires a nearly complete overhaul every time a weight or threshold is modified, but the PNRBFP is adjustable by modification of the attributes of the various non-primitive transitions. Change in the number of internal processing nodes would require significant labor, but this type of work is more in the realm of designing neural networks and is not always a requirement for neural network training.

Training the PNRBFP is a matter of adjusting the center and radius for each RBF transition, the number of output arcs for each weight-enabling transition, and/or θ for the threshold transition. Changing the number of weight arcs is a structural change to the underlying Petri Net, but simply entails changing the O(t) arc set for the weight transition. Changing the radius and center for each RBF transition and θ for the threshold transition are attribute changes rather than structure changes, and so are conceptually and programmatically very easy. Learning techniques for determining actual change values for these attributes are beyond the scope of this thesis, but are discussed in detail in the references. (Dean, 1995) (Gurney, 1997) (Nilsson, 1998) (Principe, 2000)

Chapter 4

Results

This chapter explores the practical implications of using the PNTLU and PNRBFP. Particular attention is given to processing complexity as each processing element's adjustable attributes grow large.

4.1. PNTLU by Example

Consider the example in Table 6 and Figure 28. In Petri Net terminology, X is equivalent to the marking set M = { 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, }. This X causes t_1 and t_3 to be activated. When t_1 and t_3 fire, the tokens from p1 and p3 are removed and new tokens are placed in all places found in O(t1, t3) = { p4, p5, p6, p8, p9 }. Five places in the Processing Layer have tokens, which is equivalent to the standard TLU having a value of 5 at the concentrator.

Figure 28. PNTLU before (left) and after first firing

Table 6. PNTLU Sample Values

X = { 1, 0, 1 }
W = { 3, 1, 2 }
$\theta = 3$

Since the interconnectivity of the Processing Layer transitions is designed such that any combination of three or more Processing Layer places with tokens will cause a Processing Layer activation, on the next firing at least one of the Processing Layer transitions $t_{(4,6,7,9,10,13,15,16,19,22)}$ will fire, as shown in Figure 29. A consequence of this firing is that other activated transitions at this layer may be disabled, but this does not have an effect on the overall truth-value of the PNTLU since only one firing is required to put a token in p_{output}. If θ in this example were to be increased to 6, none of the transitions in the Processing Layer would have sufficient tokens to allow activation.

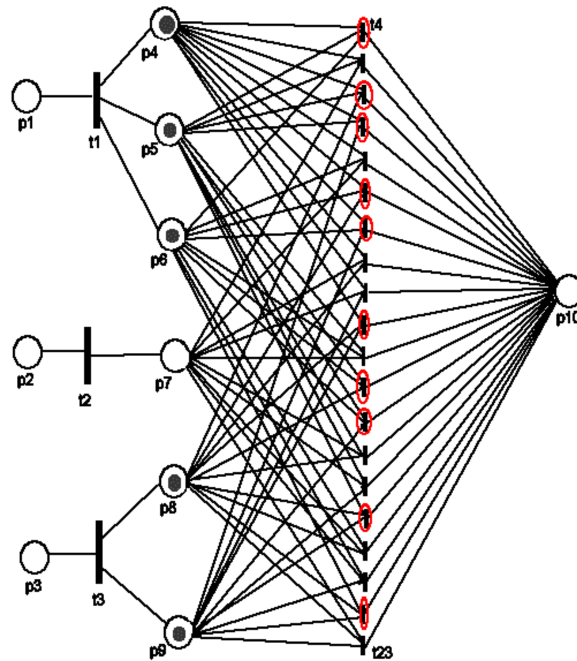


Figure 29. PNTLU with activated Processing Layer transitions $t_{(4,6,7,9,10,13,15,16,19,22)}$

Certain behaviors of the PNTLU θ, the quantity of R, and the quantity of weight places ΣW can be observed:

1. Normal PTNLU behavior requires the relationship $0 < \theta \leq \Sigma W$.
2. If $\theta \leq 0$, the PNTLU output place is *dead*. Conceptually, however, this state should imply that the PNTLU is always *true*.
3. If $\theta > \Sigma W$, the PNTLU output place is *unreachable*. Conceptually and practically, this state implies that the PNTLU is always *false*.
4. If $\Sigma W = \theta$, then R consists of a single transition with $I(t) = \{ W \}$.
5. If $\theta = 1$, then R consists of ΣW transitions, each with $I(t_i) = \{ p_i \}$.
6. When $1 < \theta < \Sigma W$, then the number of transitions in R is defined by the combinations formula $C(\Sigma W, \theta)$.

For purposes of this thesis, conditions 2 and 3 are undefined.

4.2. PNTLU Complexity

The complexity of a PNTLU is determined by the number of inputs, the threshold θ, and the number of weights ΣW for all inputs. In a software context, we can say that the 'processing' of a Petri Net is done at the transition; a transition does an *if* operation for each input arc in its $I(t)$ set. If every *if* is true for the transition, then the transition does a *remove token* operation for each arc in its $I(t)$ and a *create token* operation for each arc in its $O(t)$. In the best-case scenario a transition makes $\Sigma\ I(t)$ computations; in the worst-case, a transition makes $2(\Sigma\ I(t)) + \Sigma\ O(t)$ computations.

At the PNTLU Input Layer, the number of inputs = i transitions each with one input arc. The number of output arcs per transition is determined by $W(t)$ for that transition, but the total number of output arcs for all i transitions is ΣW. So at the Processing Layer of the PNTLU the number of computations ranges from i to $2i + \Sigma W$.

At the PNTLU Processing Layer, the number of transitions is determined by $C(\Sigma W, \theta)$, where W is the set of Input Layer transitions from the last paragraph. The number of input arcs per transition is θ, so the total number of input arcs at the Processing Layer is $C(\Sigma W, \theta) \times \theta$. Since every transition at the Processing Layer has a single output arc to p_{output}, the total output arcs at the Processing Layer is the number of transitions at the Processing Layer. The number of computations at this layer ranges from $C(\Sigma W, \theta) \times \theta$ to $2(C(\Sigma W, \theta) \times \theta) + C(\Sigma W, \theta)$.

Combining the processing requirements for the two layers gives an upper bound for the number of computations done by the PNTLU, as shown in Figure 28.

$$2i + \Sigma W + 2(C(\Sigma W, \theta) \times \theta) + C(\Sigma W, \theta)$$

Figure 30. Upper bound for PNTLU computations

The formula in Figure 30 is mostly influenced by ΣW and θ in the combinations portion. The increase of PNTLU complexity as ΣW and θ become larger is reflected by Pascal's Triangle, which governs the growth of mathematical combinations (Knuth, 1973).

4.3. PNRBFP by Example

Invoking the example from Table 3 and setting the θ attribute of t_{11} to $\theta = 3.000$, the processing flow of the PNRBFP may be easily demonstrated. Input $X = \{ 1, 0, 1 \}$, which equates to an initial state marking of $M = \{ 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \}$ for the

PNRBFP as shown in Figure 31. Notice that the standard definition tools from section 1.1 still apply to the PNRBFP; in practical implementation, care must be taken to differentiate between the various non-primitive structures, but for purposes of analysis and Petri Net-context definition, the PNRBFP remains a five-tuple as defined in Table 1.

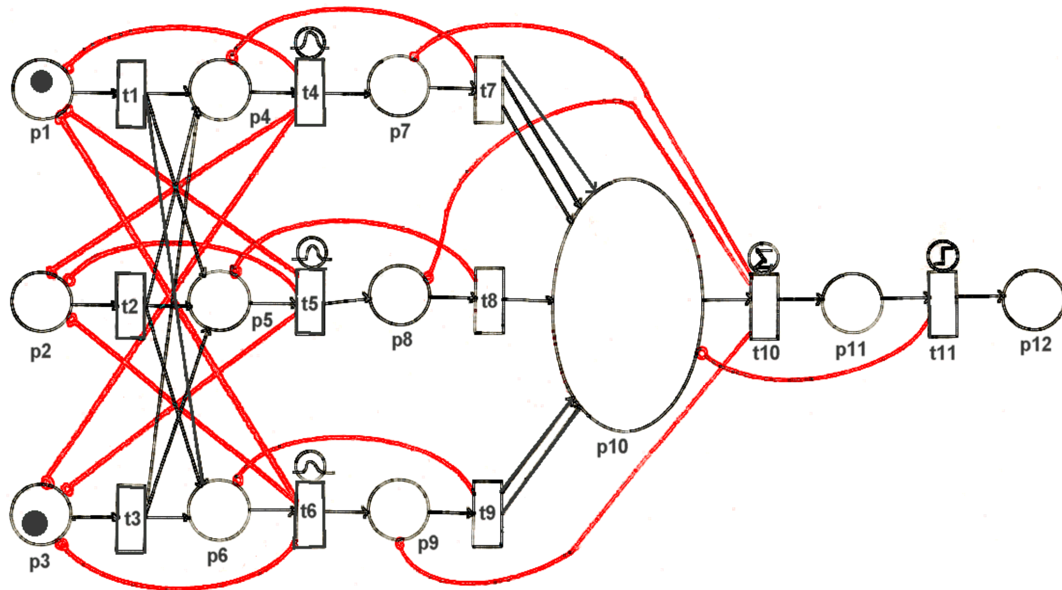


Figure 31. PNRBFP example one

After the initial tokens have been placed in $p_{1..3}$, the transitions $t_{1..3}$ will fire if they are enabled. In this example, Transitions t_1 and t_3 will each fire one time and t_2 will not fire at all. The firing of t_1 will place a single token in each place $p_{4..6}$, and the same will occur for the firing of t_3, causing $p_{4..6}$ to each have two tokens as shown in Figure 32. All three RBF transitions $t_{4..6}$ are enabled based on having tokens on their respective I(t) sets, and all three are synchronized by nature of their negation arcs connecting to the input places. Since all input places are now empty, the RBF transitions are able to fire.

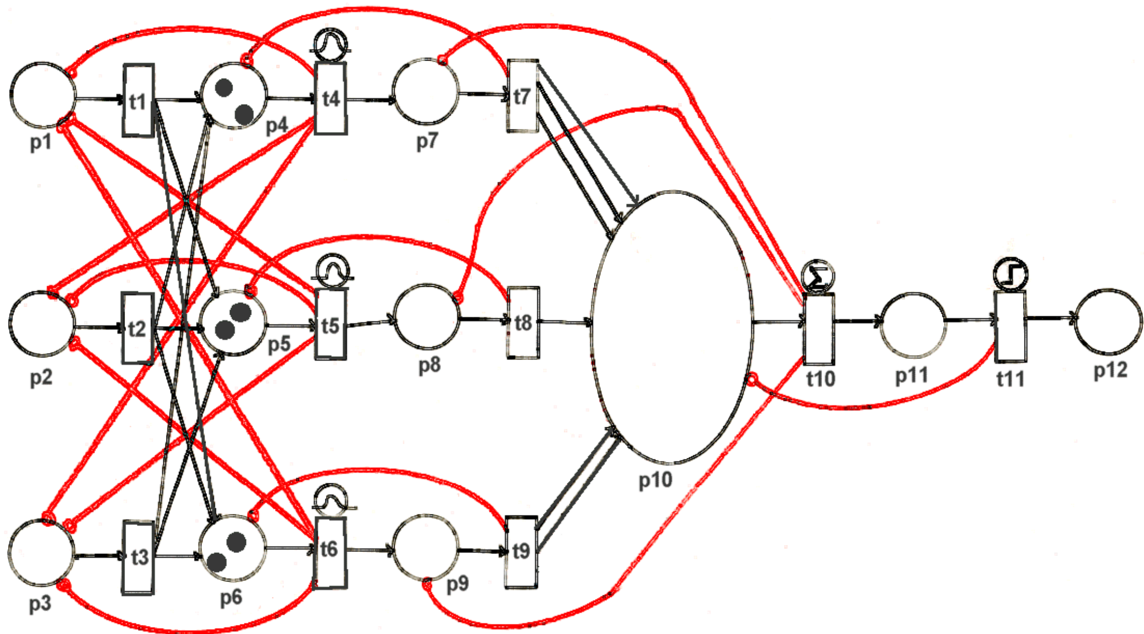


Figure 32. PNRBFP example two

Each RBF transition removes and counts the transitions from its respective I(t) place. Since each place $p_{4..6}$ has two tokens, the basis function for each RBF transition is given the input x = 2. The RBF transition t_4 now has x = 2, r = 5, c = 3. Invoking the formula from Figure 15, the stage one token deposited in p_7 has a value of 0.852. The RBF transition t_5 has values x = 2, r = 7, c = 2, creating a stage one token in p_8 with a value of 1.000. The RBF transition t_6 has values x = 2, r = 9, c = 9, creating a stage one token in p_9 with a value of 0.141, as shown in Figure 33.

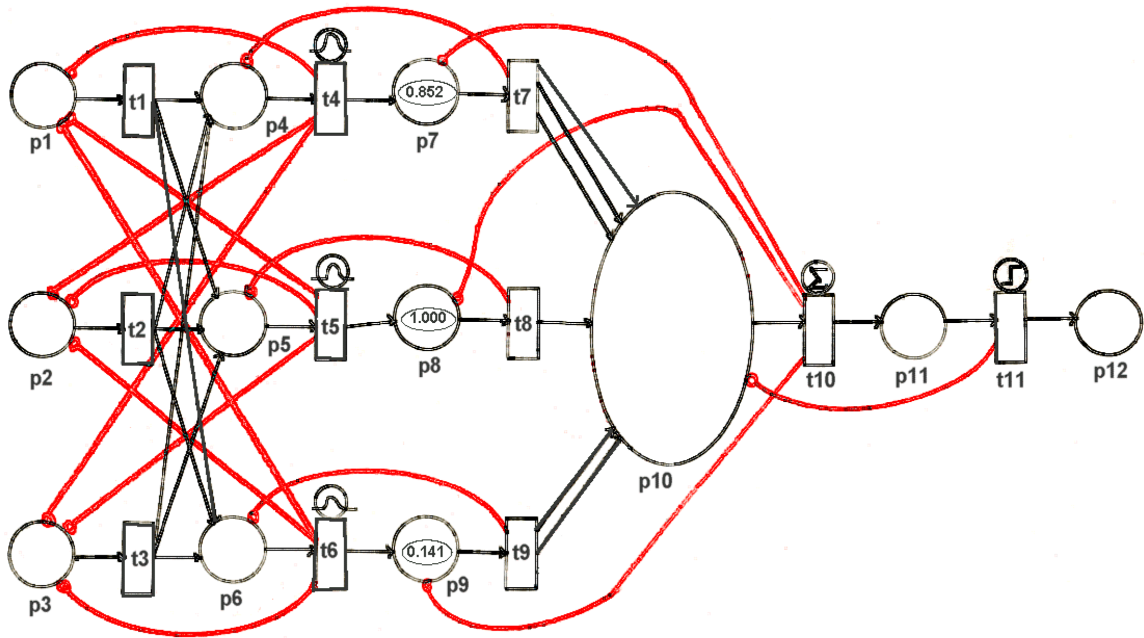


Figure 33. PNRBFP example three

The weight-enacting transitions at $t_{7..9}$ each read the stage one tokens from the places $p_{7..9}$, respectively, when their synchronizing negation arcs show that the input arcs of the RBF transitions have indeed been emptied. The transition t_7 removes the stage one token with value 0.852 from p_7 and deposits two copies of it in p_{10}, representing the weight w_1 of 2. The transition t_8 removes the stage one token with value of 1.000 from p_8 and deposits a single copy of it in p_{10}, representing the weight w_2 of 1. The transition t_9 removes the stage one token with value 0.141 from p_9 and deposits three copies of it in p_{10}, representing the weight w_3 of 3. The resulting marking with stage one tokens is shown in Figure 34.

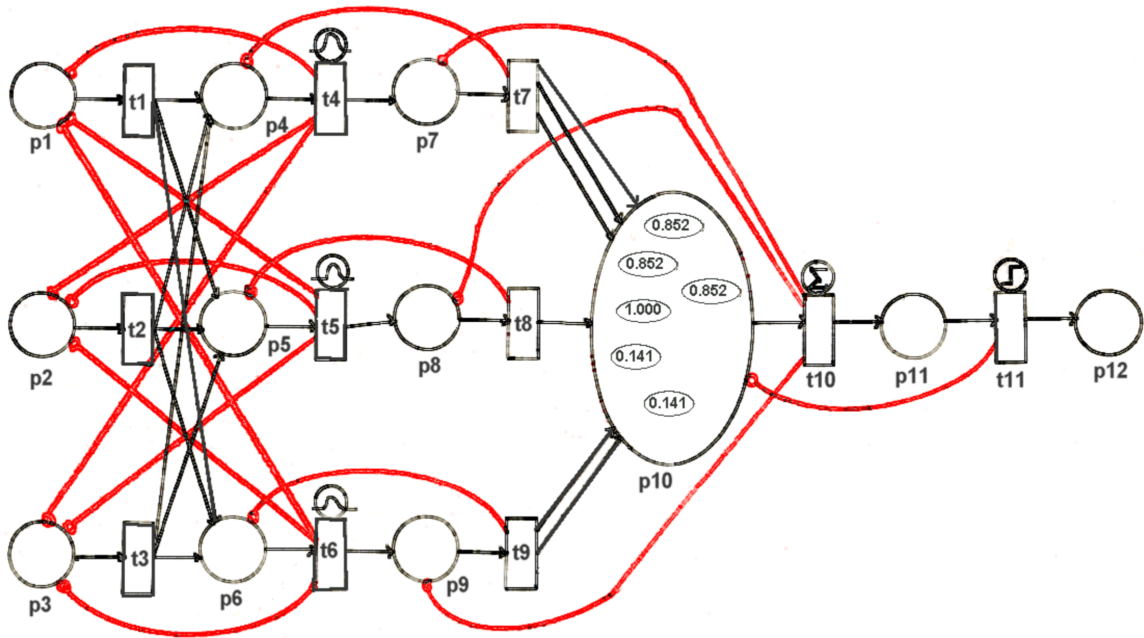


Figure 34. PNRBFP example four

When all RBF transition outputs have been read and transferred, with weights, to p_{10}, the concentrator transition is clear to fire. The concentrator transition t_{10} will repeatedly remove tokens from p_{10} until p_{10} is empty, adding the value of each stage one token to the value of a single stage two token. When p_{10} is empty, t_{10} deposits the stage two token with value (0.852 * 2) + (1.000 * 1) + (0.141 * 3) = 3.127 in p_{11} for reading by the threshold, as shown in Figure 35.

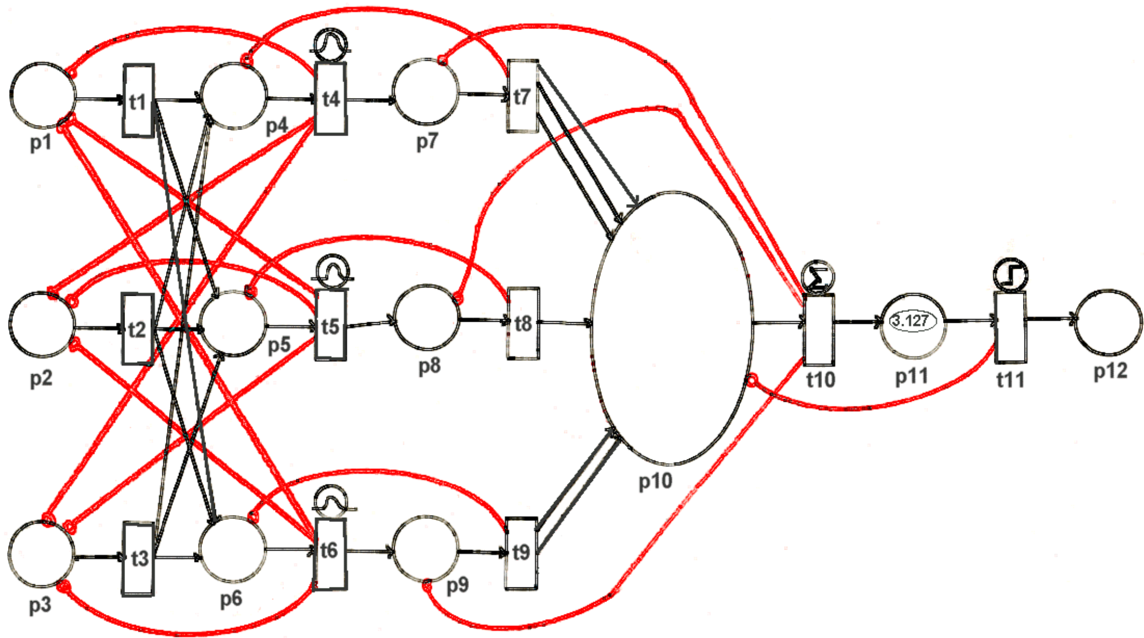


Figure 35. PNRBFP example five

The threshold transition t_{11} is free to fire when place p_{10} has been completely emptied by the concentrator transition. The threshold transition removes the stage two token from p_{11} and compares its value to its attribute θ. The stage two token value 3.127 is greater than θ = 3.000, so a primitive token is deposited in the output place p_{12} to represent a processing result of *true*.

4.4. PNRBFP Complexity

Section 4.2 demonstrated the processing requirements for the PNTLU as its attributes grow large. PNTLU growth is non-linear, as increases in weights add new internal structures that have additional processing requirements. In contrast, the processing requirements of the PNRBFP remain static with RBF attribute changes and mirror weight

changes in a one-for-one relationship. The processing done by the PNRBFP takes place inside augmented Petri Net elements rather than as a feature of Petri Net organization.

The number of computations a PNRBFP would take for a given input is directly related to the sum of all input values and can be demonstrated in steps. Returning to Figure 27 for reference and assuming a fully connected PNRBFP in general: Input transitions $t_{1..3}$ will all execute a remove token operation for each input token in their respective input places; the total number of remove token operations is thus equal to the total number of tokens placed in $p_{1..3}$ as input for step one.

Each transition $t_{1..3}$ will then deposit the same number of tokens it read into every place in its output O(t) set. Since in a fully connected Perceptron the same aggregate input quantity is presented to all basis functions, step two consists of the value of step one multiplied by the number of basis functions in the PNRBFP.

Step three consists of the RBF transitions each reading the individual tokens in their respective input places. Since each RBF input place contains the quantity from step one, step three consists of the number of computations from step one multiplied by the number of basis functions. Step three has the same value as step two.

Step four is in two parts. First, the input is given to the RBF function for computation. Second, the RBF output is used for a single create token operation. Step four consists of two operations multiplied by the number of basis functions.

Step five involves the weight transitions. Each weight transition executes a single remove token to read the stage one token created by the RBF transition. Then each weight transition executes as many create token operations as it has weight arcs. Thus, step five consists of the number of basis functions plus the total number of weights.

In step six, the concentrator transition executes a remove token operation for every token in its input place. One token is created for each weight arc from step five, so step six consists of the total number of weights plus a single create token operation.

Step seven removes the single token that is the output from step six, executes a comparison of the token value against the threshold value, and may or may not create a single token as output. Step seven thus consists of two or three operations.

Assigning M as the number of input tokens to the PNRBFP, N as the number of basis functions, and W as the total number of weights, the total number of computations for a PNRBFP can be stated: M (step one) + M * N (step two) + M * N (step three) + 2N (step four) + N + W (step five) + W + 1 (step six) + 3 (step seven) = M + MN + MN + 2N + N + W + W + 1 + 3 = 3N + 2MN + 2W + M + 4. Since N and W are constant after the PNRBFP has been built, it remains that the number of computations required by the PNRBFP grows in a linear fashion as the number of inputs increases.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

This chapter presents several special cases for both the PNTLU and the PNRBFP that may be basis for future research where Petri Net structures are desired in a neural network context. The implications of the PNTLU as a neural network processing element are explored in a functional comparison between the PNTLU and the FPN and FNPN reviewed in chapter two. Conclusions are drawn about the use and practicality of the PNTLU and PNRBFP as defined in this thesis, and a general summary is presented.

5.1. PNTLU Special Cases

In contrast to a TLU, the PNTLU as defined in section 3.1 is not able to handle negative input or negative weights, nor is it able to deal with real numbers. The domain of a simple Petri Net is the set of positive integers and the input/output domain of a PNTLU is the set of Boolean values { 0, 1 }. This makes the PNTLU less functional than a TLU in some neural network settings, although as demonstrated in section 4.1, both the PNTLU and the TLU function equivalently when dealing with a Boolean input/output domain. This section defines three special cases of the PNTLU that could be developed to overcome these limitations.

5.1.1. PNTLU Negative Values

The PNTLU could be modified to allow negative weights or negative input values, creating an input range of { -1, 0, 1 }, but this modification would increase the inter-

connective complexity of the PNTLU considerably. Allowing negative weights or inputs would require new 'subtraction' layers which would function in a manner similar to the Processing Layer, except that the Subtraction Layer output would be to a 'trash can' place (Ahson, 1995) (Murata, 1989) instead of to an output place. Negative-weight or negative-input Subtraction Layer transitions would serve to remove tokens from the Processing Layer as they were placed there from the Processing Layer.

5.1.2. PNTLU Integer Input

The PNTLU is limited to Boolean input and output, while the TLU allows real number input and Boolean output. Real numbers are outside of the domain of Petri Nets, but it may be possible to build a PNTLU that accepts positive integers for input. Integer input could be represented by the number of tokens placed in an input place. Since an input value is modified by multiplication with an associated weight, it follows that the PNTLU Processing Layer would need to be increased by a factor of the input quantity. Dynamic reconnecting, in a manner similar to that shown in Table 6, of the arcs in the PNTLU's first layer for every new set of inputs would accomplish this goal. Multiple-token input would be treated as a special case by essentially cloning the input's weight set by the quantity of input, but this would require addition of new Processing Layer places and reconnecting of Processing Layer transitions after input was presented to the PNTLU. For example, an input of 3 with a weight of 2 would be treated by the PNTLU, after dynamic reconnecting, as three inputs of *true*, each with a weight of 2. The influence at the Processing Layer would be the same: $3 \times 2 = 6$ compared to $\sum_{i=1}^{3} (1 \times 2) = 6$.

5.1.3. PNTLU Integer Output

The PNTLU could be further modified to give an integer value as output rather than just a simple Boolean. This behavior is not consistent with TLU function, but is an interesting side effect of the Petri Net architecture. If the conflict between Processing Layer transitions could be overcome, the number of tokens in p_{output} would represent the degree of satisfaction the PNTLU found from its input set. The non-removal of tokens from places simply implies that the motivation for their original placement remains true (Ahson, 1995). The PNTLU behavior could be modified to not remove tokens, which would prevent Processing Layer transitions from disabling each other but would add three new requirements to the PNTLU. First, removal of input tokens would become the responsibility of the environment; i.e. when the conditions change that caused the input to be created, the input is removed. Second, any change to the input places would require the PNTLU to re-process. Third, all activated PNTLU transitions would only fire one time per processing cycle to prevent transitions from firing an infinite number of times. These modifications would cause p_{output} to be filled with a number of tokens that represent the number of times θ was reached in the Processing Layer. For example, if $\theta = 3$ and there are 12 marked places in the Processing Layer then 4 Processing Layer transitions will fire, placing 4 tokens in p_{output}.

5.2. PNRBFP Special Cases

The input domain of the PNRBFP is positive integers and the output domain is Boolean. The PNRBFP could be easily modified to produce integer output for purposes of feeding the input of another PNRBFP or other structure with integer input. The

threshold transition of the PNRBFP could be replaced with a rounding function to translate the real number value of the stage two token into the required integer value, imposing no additional processing requirement on the overall structure.

The PNRBFP could also be adjusted to fit into a larger fuzzy system with a problem domain n, $0 \leq n \leq 1$, by replacing the threshold transition with another RBF transition that would normalize the stage two token's $0 \leq n$ real number into a proper fuzzy value. This type of modification could allow the PNRBFP some of the descriptive power of fuzzy systems, such as discussed in (Looney, 1996), while maintaining the learning power of the Perceptron.

5.3. A Comparison of the FPN and FNPN to the PNTLU

Both the FPN and the FNPN overlap with the problem domain and implementation of the PNTLU. The FPN and the FNPN are able to provide a unique output based on the effect their internal organizations have on a particular input, just as with the PNTLU. All three techniques require a stretch of the basic Petri Net concept, though the FPN and FNPN require physical changes while the PNTLU requires conceptual change. Each technique has a certain niche that suites it: FPNs are good for representing and comparing data that are non-binary, but FPNs are not natively able to be 'taught'. FPN output varies based on its inputs' proximities to an established belief; while modification of belief parameters in the form of valve and threshold settings may equate to learning, it is conceptually a different issue. FNPNs are able to represent and process non-binary data, but also incorporate the native learning ability of a TLU such that the FNPN may not only represent how a given input relates to known 'belief' data, but it can be taught to

only respond when that input is within certain parameters of quantity and input location. The PNTLU does not have any native fuzzy rule processing ability, but it is able to learn required output based on training for known inputs. The PNTLU expands the FNPN's learning ability by allowing weights from the set of positive integers vice the set { -1, 0, 1 }. Special cases of the PNTLU may also be implemented to allow input and output as positive integers rather than Boolean values. Table 7 shows a comparison between the FPN, FNPN, and PNTLU in terms of function and implementation.

Table 7. FPN, FNPN, PNTLU Comparison.

	FPN	**FNPN**	**PNTLU**
Type of input	0<=n<=1	0<=n<=1	Boolean (Special cases allow positive integer)
Type of output	0<=n<=1	Boolean value, where true contains a fuzzy value 0<=n<=1	Boolean (Specials case allow positive integer)
Supervised learning	No	Yes	Yes
'Fuzzy' data tokens	Yes	Yes	No
Requires modification to basic Petri Net structures	Yes	Yes	No
Output suitable for interfacing with others of its kind	Yes	Yes	Yes
May use any standard Petri Net software or analysis techniques	No	No	Yes

5.4. Conclusions

This thesis has demonstrated that Petri Nets may be arranged or modified to provide neural network processing power equivalent to special purpose implementations while retaining the strengths of Petri Net tools and simplicity. While it is shown to be possible,

it may prove true that the PNTLU and PNRBFP are too complex or unwieldy for general neural network use in situations where Petri Net structures are not required.

The PNTLU processes as a TLU, but requires a significantly more complex graph structure to implement the same or less functionality. As the PNTLU attributes grow larger, this complexity may make it less valuable as a processing element, especially when the TLU is very limited in processing power.

The true strength of the PNTLU is in its simplicity. Petri Net researchers or practitioners could easily use existing tools to add TLU functionality to Petri Net systems that have already been developed and put into service, adding a dimension of power to a system that is complex by organization but simple in its parts.

The PNRBFP does not suffer the same problems with graph complexity, but its specialized nature requires specialized implementation. The increased processing power and broader problem domain may make the PNRBFP practical as a neural networking tool for practitioners who have an existing base of Petri Net knowledge or infrastructure and need to expand into a neural network problem domain.

The largest drawback to the PNRBFP is its implementation. The PNRBFP requires specialized programming that may be even more difficult that simply creating a normal RBF-based Perceptron. The PNRBFP may only be useful in situations where Petri Net systems are established yet neural network processing is required.

5.5. Summary

This thesis has explained the functions of basic Petri Nets, threshold learning units, radial basis functions, and Perceptrons for the purpose of exploring the applicability of

joining Petri Net and neural network technologies. Similar research from the academic literature has been identified and reviewed.

Two different Petri Net implementations of neural network structures have been presented. The Petri Net Threshold Learning Unit has been demonstrated to be functionally equivalent to the threshold learning unit when both are used in an appropriate domain. The strength of the PNTLU as a pure Petri Net has been discussed and its complexity as its attributes grow has been addressed. The Petri Net Radial Basis Function Perceptron has been shown to be equivalent to a Perceptron with Gaussian radial basis functions. The complexity of the PNRBFP has been analyzed and the strengths and weaknesses of the modified Petri Net approach have been visited.

Step-by-step examples of both the PNTLU and the PNRBFP have been provided to illustrate their respective functionality. Special cases have been outlined to provide for the possibility of further research or practical use of both techniques. A comparison of the PNTLU and existing technologies has been provided to demonstrate the direction of this research in the context of other work. Finally, a realistic assessment of the practicality and applicability of both techniques has been presented.

Reference List

Ahson, S. I. (1995). Petri Net Models of Fuzzy Neural Networks. *IEEE Transactions on Systems, Man, and Cybernetics, 25* (6), 926-932.

Balakrishnan, V. K. (1991). *Introductory Discrete Mathematics*. New York: Dover Publications.

Bors, A. G. (2001). Introduction of the Radial Basis Function (RBF) Networks. *Online Symposium for Electronics Engineers,* issue 1, vol. 1, DSP Algorithms: Multimedia, http://www.osee.net/, pp. 1-7.

Broomhead, D.S., Lowe, D. (1988). *Multivariable Functional Interpolation and Adaptive Networks*. Complex Systems, vol.2, pp.321-355.

Chen, S., Ke, J., Chang, J. (1990). Knowledge Representation Using Fuzzy Petri Nets. *IEEE Transactions on Knowledge and Data Engineering*, 2 (3), 311-319.

Cherniaev, V. (1995). Intelligent Systems: Unified Approach to Knowledge Representation, Analysis and Implementation Based on Fuzzy Petri Nets. *Proceedings, Fourth Bar Ilan Symposium on Foundations of Artificial Intelligence*, 43-50.

Dean, T., Allen, J., Aloimonos, Y. (1995). *Artificial Intelligence: Theory and Practice.* Menlo Park, CA: Addison-Wesley Publishing Company.

Jensen, K. (1996). *Coloured Petri nets: Basic Concepts, Analysis Methods, and Practical Use (Vol. 1) (2nd Ed.).* Berlin: Springer-Verlag.

Knuth, D. E. (1973), *The Art of Computer Programming, Volume 1: Fundamental Algorithms 2nd ed.* Reading, MA: Addison-Wesley.

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

Looney, C. (1988). Fuzzy Petri Nets for Rule-Based Decision Making. *IEEE Transactions on Systems, Man, and Cybernetics* 18 (1), 178-183.

Looney, C. (1996). *Radial Basis Functional Link Nets as Learning Fuzzy Systems.* Technical Report, University of Nevada, Reno.

Murata, T. (1989). Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE* 77 (4), 541-580.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis.* San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Orr, M. J. L. (1996). *Introduction to Radial Basis Function Networks*. Technical Report, Centre for Cognitive Science, University of Edinburgh, Scotland.

Pastor, E., Cortadella, J, Oriol, R. (2001). Symbolic Analysis of Bounded Petri Nets. *IEEE Transactions on Computers*, 50 (5), 432-436.

Peterson, J. L. (1977). Petri Nets, *ACM Computing Surveys, 9* (3), 221-252.

Peterson, J. (1981). Petri Net Theory and the Modeling of Systems. Prentice-Hall, NJ.

Poggio, T. & Girosi, F. (1989), *A Theory of Networks for Approximation and Learning* , Technical Report AIM-1140, Artificial Intelligence Laboratory and Center for Biological Information Processing, Whitaker College, Massachusetts Institute of Technology.

Principe, J. C., Euliano, N. R., Lefebvre, W. C. (2000). *Neural and Adaptive Systems.* New York: John Wiley & Sons, Inc.

Sivaraman, E. (1999). *An Approach for Solving the General Petri Net Reachability Problem - Duality Theory and Applications*. Retrieved March 1, 2002, from http://www.okstate.edu/cocim/members/eswar/duality.pdf

Turban, E., Aronson, J. E. (1998). *Decision Support Systems and Intelligent Systems (5th ed.).* Upper Saddle River, NJ: Prentice-Hall.

Walpole, R. E., Myers, R. H. (1993). Probability and Statistics for Scientists and Engineers (5th). New York: Macmillian Publishing Company.