A Thesis

entitled

A Parallel and Distributed Computing Platform for Neural Networks Using Wireless Sensor Networks

by

Linqian Liu

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Master of Science Degree in Engineering

_________________________________________
Dr. Gursel Serpen, Committee Chair

_________________________________________
Dr. Mohsin M. Jamali, Committee Member

_________________________________________
Dr. Srinivasa Vemuru, Committee Member

_________________________________________
Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo

August 2012
Artificial neural network algorithms inherently possess fine-grain parallelism and offer the potential for fully distributed and local computation. A scalable hardware computing platform that can take advantage of such a massive parallelism and distributed computation attributes of artificial neural networks is considered to be well-poised to compute real-time solution of complex and large-scale problems. This thesis proposes a novel computing architecture for parallel and distributed computation where the hardware-software platform is the wireless sensor networks complete with its wireless protocol stack. More specifically, the proposed idea leverages the existing wireless sensor networks technology to serve as a hardware-software platform to implement and realize certain type of algorithms with fine-grain parallelism, such as those in the domain of artificial neural networks, in massively parallel and fully distributed mode. The research vision is to enable real time computation of solutions of large-scale and complex problems through the proposed parallel and distributed hardware realization of computational algorithms. The thesis defines the new parallel and distributed processing
(PDP) and computing architecture and its application for artificial neural network computations. The underlying architectural principles, and structure of the proposed parallel and distributed computing platform are formulated and established.

The proposed design is illustrated for feasibility through a simulation-based case study that leverages Kohonen’s self-organizing map or SOM neural network on a number of different problem domains or data sets. The research study demonstrates mapping Kohonen’s self-organizing map or SOM, configured for a set of domain specific problems, to the proposed PDP architecture. A comprehensive simulation study is conducted to assess the performance profile of and demonstrate the proposed computing architecture, with respect to feasibility. A wireless sensor network simulator (PROWLER) is employed for validation and performance assessment of the proposed computational framework. Three data sets, namely Alphanumeric or Text, Iris, and Wine, where each one differs in the number of attributes, instances, and clusters, are employed to profile the performance of the proposed computing platform. The simulation results are compared with those from the literature and through the MATLAB SOM toolbox. Comparative performance analysis suggests that the proposed computing platform is feasible and promising.

The proposed design has potentially much wider applicability for problems with inherent fine-grain parallelism in various domains where mathematics-based problem-solving methodology is not applicable due to lack of a closed-form model for the process or system. Solving complex and very large-scale problems in real time is likely to have radical and ground-breaking impact on the entire spectrum of scientific, technological, economic and industrial endeavors enabling many solutions that were simply not feasible.
Acknowledgements

This research thesis would not have been possible without the support of many people. I wish to express my gratitude to my supervisor, Dr. Gursel Serpen, who was abundantly helpful and offered invaluable assistance, support and guidance.

The research performed in this thesis was based on the research projects described in the following NSF proposals submitted by Dr. Gursel Serpen: Brain-In-A-Box- A Novel Parallel and Distributed Computer Architecture for Neurocomputing; AI-WSN: Adaptive and Intelligent Wireless Sensor Networks; Wireless Sensor Networks as Parallel and Distributed Hardware Realization of Artificial Neural Networks; and Autonomous, Real-Time, and Distributed Construction and Adaptation of Connected Dominating Sets for Hierarchical Infrastructure Realization in Wireless Sensor Networks.

Deepest gratitude is also due to the members of the supervisory committee, Dr. Mohsin Jamali and Dr. Srinivasa R. Vemuru, without their knowledge and assistance this study would not have been successful.

Special thanks also to all my friends, especially my group members: Chao Dou, Jiakai Li and Zhengning Gao for sharing the literature and invaluable assistance. Finally, I want to thank my family for their love and unconditional support that made me conquer all the difficulties throughout my graduate study.
# Table of Contents

Abstract iii  
Acknowledgements v  
Table of Contents vi  
List of Tables xi  
List of Figures xii  
1 Introduction 1  
1.1 VLSI for ANN Implementations .......................................................2  
1.2 Simulation of ANNs ........................................................................4  
1.3 Hybrid (software and hardware) Computing Systems for ANNs ..........6  
2 Background 9  
2.1 Artificial Neural Networks ..................................................................9  
2.1.1 Structure .....................................................................................10  
2.1.2 Application and Limitations .........................................................17  
2.2 Wireless Sensor Networks ..................................................................19  
2.2.1 Architecture ................................................................................20  
2.2.2 Network Protocols .........................................................................21  
3 Proposed Computer Architecture for Neurocomputing 26  
3.1 Wireless Processor Networks (WPN) as a PDP Computing System ..........28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Processing Node and Neural Computations</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>The Novel Contribution</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Scalability, Computational Cost, Messaging Complexity and Power</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Scalability of the Proposed System in terms of its Real-Time Computation</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>The Proposed Idea and Literature</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>Mapping ANN Algorithms and Domain Problems to PDP Computing Architecture</td>
<td>38</td>
</tr>
</tbody>
</table>

**4 Proposed Computer Architecture and Technology** 40

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Physical Packaging, Spacing and Layout</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>WSN Technology Today and Feasibility Assessment of Proposed PDP Computing Platform</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Realization through Custom Development</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>4.2.2 Realization through Integration of COTS Components</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>4.2.3 Example of the Smallest Processing Node</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>On the Market Today</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>4.2.4 Processing Power and Memory Storage Requirements</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>In Today’s Motes</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>4.2.5 Are the Wireless Networking Protocols, Architectures and Software Mature</td>
<td>44</td>
</tr>
</tbody>
</table>
5 Demonstration of Proposed Methodology: An Illustrated Case Study

5.1 Kohonen’s Self Organizing Map (SOM)

5.1.1 SOM Neural Network Fundamentals

5.1.2 Solution Quality Measures

5.1.3 Data Visualization

5.1.4 SOM Map Construction, Initialization and Training

5.2 Embedding Kohonen’s SOM into a Wireless Sensor Network

5.2.1 Mapping SOM to Wireless Sensor Network

5.2.1.1 Initialization Phase

5.2.1.2 Training Phase

5.2.1.3 Deployment Phase

5.2.1.4 Routing Protocol

6 Simulation Study

6.1 PROWLER – A MATLAB-BASED WSN Simulator

6.1.1 Radio Propagation Model

6.1.2 Reception Condition

6.1.3 MAC Protocol

6.1.4 Message Formatting and Routing Protocol

6.1.5 Radio, Topology and Application

6.1.6 PROWLER code for Kohonen’s SOM Implementation

6.2 Data Sets
6.2.1 Alphanumeric or Text Data Set ......................................................103
6.2.2 Iris Data Set ..............................................................................105
6.2.3 Wine Data Set ...........................................................................105

6.3 Simulation Study & Results .................................................................107
6.3.1 SOM and PROWLER Parameter Settings ....................................108
6.3.2 Dataset: Alphanumeric or Text Symbols .....................................109
   6.3.2.1 Solutions Reported in Literature ...........................................110
   6.3.2.2 Solutions with MATLAB SOM Toolbox ............................111
   6.3.2.3 Solutions with WSN-SOM ..................................................115
6.3.3 Dataset: Iris .................................................................................119
   6.3.3.1 Solutions Reported in Literature ...........................................119
   6.3.3.2 Solutions with MATLAB SOM Toolbox ............................119
   6.3.3.3 Solutions with WSN-SOM ..................................................122
6.3.4 Dataset: Wine .............................................................................125
   6.3.4.1 Solutions Reported in Literature ...........................................125
   6.3.4.2 Solutions with MATLAB SOM Toolbox ............................126
   6.3.4.3 Solutions with WSN-SOM ..................................................128
6.3.5 Summary and Conclusions of Simulation Study ..........................131

7 Conclusions and Recommendation for Further Study ..............................133
7.1 Research Study Highlights and Conclusions ....................................133
References 135

Appendix A: User’s Manual and Installation Guide 146

Appendix B: Program Code 148

som_application.m .................................................................................................................. 148

som_animation.m .................................................................................................................. 167

som_topology.m .................................................................................................................. 168
List of Tables

2.1 Analogy between Biological Neuron and its Computational Model ...................12
5.1 Map Dimensions and Neuron Counts in SOM Topology for Datasets ..............63
5.2 Training Iteration Counts for all Simulations ...............................................65
6.1 PROWLER MAC Protocol Parameter Settings ..............................................82
6.2 Message Types and Descriptions ..................................................................83
6.3 Supported Events and Actions in Application Layer in PROWLER ...............91
6.4 Datasets and Characteristics .........................................................................98
6.5 Encoding of Alphanumeric Symbol Set .......................................................100
6.6 Sample Pattern for Each of Three Classes in Iris Dataset ............................101
6.7 Wine Data Attributes List ............................................................................102
6.8 Sample Pattern for Each of the Three Classes in Wine Dataset ...................102
6.9 Parameter Settings for PROWLER Simulations ...........................................105
6.10 Simulation Study Results for Alphanumeric Dataset With 5% Message Loss Tolerance.................................................................111
6.11 Simulation Study Results for Iris Dataset with 5% Message Loss Tolerance ....119
6.12 Simulation Study Results for Iris Dataset With 5% Message Loss Tolerance ..124
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Structure of a Neuron and Connectivity between Two Neurons</td>
<td>11</td>
</tr>
<tr>
<td>2-2</td>
<td>Diagram of Neuron Computational Model</td>
<td>12</td>
</tr>
<tr>
<td>2-3</td>
<td>Activation Functions</td>
<td>13</td>
</tr>
<tr>
<td>2-4</td>
<td>Multilayer Perceptron Neural Network Diagram</td>
<td>15</td>
</tr>
<tr>
<td>2-5</td>
<td>Three-Layer Feed-Forward Perceptron Network</td>
<td>16</td>
</tr>
<tr>
<td>2-6</td>
<td>Single Node Architecture</td>
<td>19</td>
</tr>
<tr>
<td>2-7</td>
<td>Multi-Hop Network</td>
<td>21</td>
</tr>
<tr>
<td>5-1</td>
<td>Topology of a Two-Layer SOM Neural Network</td>
<td>46</td>
</tr>
<tr>
<td>5-2</td>
<td>Depiction of how the Neighborhood Shrinks with Time</td>
<td>48</td>
</tr>
<tr>
<td>5-3</td>
<td>Linear, Inv and Power Learning Functions with Initial Value 0.5 and Total Training Iterations of 100</td>
<td>49</td>
</tr>
<tr>
<td>5-4</td>
<td>Bubble, Gaussian, Cutgauss, Ep shrinking Function Plots</td>
<td>50</td>
</tr>
<tr>
<td>5-5</td>
<td>Pseudocode for Kohonen’s SOM Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>5-6</td>
<td>U-Matrix and Component Maps for Iris Dataset</td>
<td>59</td>
</tr>
<tr>
<td>5-7</td>
<td>Iris Data Map in 3-D</td>
<td>60</td>
</tr>
<tr>
<td>5-8</td>
<td>Message Transaction between Supervisory and Generic Motes</td>
<td>69</td>
</tr>
<tr>
<td>6-1</td>
<td>PROWLER Simulation Interface GUI</td>
<td>76</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6-2</td>
<td>PROWLER GUI for Parameter Settings</td>
<td>77</td>
</tr>
<tr>
<td>6-3</td>
<td>Message Structure Employed for the Simulation Study</td>
<td>82</td>
</tr>
<tr>
<td>6-4</td>
<td>Illustration of Routing Protocol Employed for the Simulation Study</td>
<td>85</td>
</tr>
<tr>
<td>6-5</td>
<td>PROWLER GUI for A WSN with 300 Motes Uniformly Distributed Across A</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>Two-Dimensional 10×10 Field</td>
<td></td>
</tr>
<tr>
<td>6-6</td>
<td>An Example Case Explaining Partitioning a Rectangular Deployment Field into Equal-Sized Grids</td>
<td>90</td>
</tr>
<tr>
<td>6-7</td>
<td>Event-Driven Execution of PROWLER Application Code</td>
<td>93</td>
</tr>
<tr>
<td>6-8</td>
<td>PROWLER Application Event-Driven Execution Diagram</td>
<td>94</td>
</tr>
<tr>
<td>6-9</td>
<td>Application Code Execution for Init-Memory Routine</td>
<td>95</td>
</tr>
<tr>
<td>6-10</td>
<td>Application Code Execution as Driven by Message Type</td>
<td>97</td>
</tr>
<tr>
<td>6-11</td>
<td>Calibrated Training Results After 10,000 Steps</td>
<td>106</td>
</tr>
<tr>
<td>6-12</td>
<td>Minimum Spanning Tree Representation of SOM Mapping of Alphanumeric Dataset</td>
<td>107</td>
</tr>
<tr>
<td>6-13</td>
<td>Simulation Results Using MATLAB SOM Toolbox for Alphanumeric Dataset in Cited Literature [22]</td>
<td>108</td>
</tr>
<tr>
<td>6-14</td>
<td>Component (left) and Histogram (right) Maps for Alphanumeric Dataset</td>
<td>109</td>
</tr>
<tr>
<td>6-15</td>
<td>Histogram Maps for Pattern J (top left), R (top right), Z (bottom left) and 6 (bottom right)</td>
<td>110</td>
</tr>
<tr>
<td>6-16</td>
<td>Simulation Results for Alphanumeric Dataset by PROWLER GUI</td>
<td>113</td>
</tr>
<tr>
<td>6-17</td>
<td>WSN-SOM Component Map with Label Information for Alphanumeric</td>
<td>113</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>6-18</td>
<td>Minimum Spanning Tree of WSN-SOM Results ........................................114</td>
<td></td>
</tr>
<tr>
<td>6-19</td>
<td>U-Matrix (top left) and Component Maps Using MATLAB SOM Toolbox for Iris Dataset .........................................................................................................................116</td>
<td></td>
</tr>
<tr>
<td>6-20</td>
<td>Component Map with Label Information for Iris Dataset ................................117</td>
<td></td>
</tr>
<tr>
<td>6-21</td>
<td>Iris Simulation Results Shown in Prowler GUI ........................................119</td>
<td></td>
</tr>
<tr>
<td>6-22</td>
<td>WSN-SOM Results Component Map ................................................................120</td>
<td></td>
</tr>
<tr>
<td>6-23</td>
<td>U-Matrix (top left) and Component Maps Using MATLAB SOM Toolbox for Wine Dataset ..........................................................................................................................123</td>
<td></td>
</tr>
<tr>
<td>6-24</td>
<td>Component Map with Label Information for Wine Dataset ............................123</td>
<td></td>
</tr>
<tr>
<td>6-25</td>
<td>Simulation Results for Wien Dataset as Shown by PROWLER GUI .................125</td>
<td></td>
</tr>
<tr>
<td>6-26</td>
<td>WSN-SOM Results Component Map ................................................................125</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A truly parallel and distributed hardware implementation of artificial neural network (ANN) algorithms has been a leading and on-going quest of researchers. Numerous attempts to realize the hardware implementation in silicon proved to be too challenging or nearly impossible particularly when the scale of the implementation increased to dimensions at par with the real life problems. Many attempts were made through VLSI-based technology with limited success. In some other cases hybrid systems with specially designed accelerator hardware boards were interfaced with von Neumann based systems (standalone supercomputers or even parallel computing machines) to achieve acceptable speed of training and execution. The results were mixed: although some really large-scale ANN algorithms could be trained and executed on these platforms, it is the lack of availability of such platforms for ready access for the typical user, which served as one main hindrance. Pure software implementation of artificial neural networks was also attempted mainly for smaller scale or non-real-time problems due to obvious time complexity of such realizations. Again, larger scale problems would require supercomputing-grade platforms for software-only realizations and are simply not practical. A massively parallel and distributed implementation of artificial neural networks algorithms in a form somewhat similar to how biological neural networks exist
in a brain could facilitate true real-time computation of solutions for very large scale real life problems. This project proposes a new computer architecture based on wireless sensor networks to serve as the true parallel and distributed processing platform for artificial neural networks to fully enable their computational potential for real-time context. In the following several subsections however, we will first present a survey of and more-detailed discussion for each of existing ANN implementation technologies.

1.1 VLSI for ANN Implementations

The very large scale integration (VLSI) technology has been unable to deliver a truly parallel and distributed realization of artificial neural networks algorithms at large scales. This section briefly presents the state of the art in hardware realization of artificial neural network (ANN) algorithms through the VLSI technology. One of a few larger scale VLSI technology based hardware realization of neural networks is discussed in [1] from purely an academic perspective. Authors report a neural network design with 6144 spiking neurons and 1.57 million synapses. In another recent study reported in [2], a neural network with up to 10K neurons has reportedly been realized and plans for implementing a neural network with 100K neurons and 100 million synapses was discussed as part of the future plans. A comprehensive review of commercial, and yet mainly experimental, VLSI ICs (analog, digital or hybrid) implementing ANN algorithms is presented with detailed characterization of number of neurons (nodes or processing elements), on-chip learning capability, type of neural network algorithm, and others in [3]. Their paper concludes “Moreover, there is no clear consensus on how to exploit the currently
available VLSI and even ultra-large-scale integration technological capabilities for massively parallel neural network hardware implementations.” Also other studies report essentially very small-scale VLSI hardware realizations of artificial neural networks [4, 5 and 6].

As to actual VLSI hardware that made into the marketplace, the news is relatively old and discouraging. Some of the well-known hardware implementations are Intel ETANN, CNAPS, Synaptics Silicon Retina, NeuraLogix NLX-420, HNC 100-NAP, Hitachi WSI, Nestor/Intel NI1000, Siemens MA-16, SAND/1, MCE MT19003, AT&T ANNA among others [6 and 34]. Many of these are reportedly not available for purchase anymore and there are no recent notable entries into the commercial domain either. Most of these hardware realizations are very small-scale and geared towards highly specific neural network algorithms. The most recent reported example of custom VLSI IC that implements a neuromorphic chip was reported in September 2011 by a team of researchers with IBM. This IC houses 256 neurons and 64 K binary synapses [87]. The authors claimed that the proposed system is scalable, and yet no reports of an actual prototype being produced to date materialized.

The realization of neural algorithms through VLSI on silicon hardware as integrated circuits (IC) has been essentially a mainly academic exploration and failed to score any notable real-life or commercial success. The VLSI hardware implementation of a neural network, where neurons are physically discrete entities within a given IC, is not able to deliver large-scale neural network realizations. The number of neurons, which is direct
indication of the computational power, in a VLSI-based neural network realization is limited to no more than 10,000 neurons under the best and most optimistic scenarios (and that is typically in an academic or research lab setting). The VLSI approach further suffers from inflexibility, and the curse of exploding parameter space (too many parameters to set, adjust, tune or adapt). The VLSI realizations of ANNs are in almost all cases closely aligned and optimized for a specific artificial neural network algorithm without much flexibility to accommodate another even slightly different neural algorithm. The size of networks that can be implemented is still small by any measure. It is perhaps fair to conclude that VLSI technology failed to deliver a hardware computing platform that is generic and possesses fine-grain parallelization to house an arbitrary selection of large scale neural network algorithms for massively parallel and distributed computation in real time.

1.2 Simulation of ANNs

Simulations on even parallel architectures (in traditional computer architecture sense) fail to scale with the size of the neural network since both time and space complexities quickly reach a level that is beyond what is affordable. Even if multiple processors of a parallel computing platform update a multiplicity of neurons in a given neural network and specialized concurrency techniques perform or facilitate certain operations in parallel (i.e. matrix algebra) the spatio-temporal cost of pure simulation is still insurmountable as extensive empirical evidence indicated. The following example exposes one such scenario. Simulating an artificial neural network, say the Hopfield network algorithm, in a purely software form or on an hybrid platform with neural processing accelerators for
large instances of (optimization) problems poses overwhelming challenges in terms of memory space that must be allocated for the weight matrix, which is the highest-cost data structure. In general, it is well known that the number of bytes (in real or virtual memory) required is on the order of $O(N^4)$ for $N$-vertex graph-theoretic problems mapped to a Hopfield neural network topology under the assumption that an $N\times N$ neuron array is employed. Searching a 1000-vertex graph would require approximately $10^{12}$ bytes of main memory storage to maintain weight matrix entries: two assumptions prevail for this computation, which are that each weight matrix entry is stored in a float type variable and a float type variable requires several bytes of storage space on a given computing platform. Reasoning along the same lines, a computing platform with on the order of a few Giga bytes of main memory could accommodate up to 200-vertex graph search problems. Given that a 1000-vertex graph would require on the order of Tera ($10^{12}$) bytes of storage for the weight matrix, memory space requirements for Hopfield neural network simulations quickly increase to levels of being too costly in terms of memory space. Perhaps lack of simulations for truly large-scale neural networks in either academic literature or in use anywhere (with one recent exception [8], which apparently has been subject to controversy as to validity of its findings [9],) is a testimony to the fact that the option of pure simulation is severely constrained for any practical utility.
1.3 Hybrid (Software & Hardware) Computing Systems for ANNs

There have been numerous attempts to build specialized or custom computing platforms based on a mix of hardware and software components. The resultant computing systems were byproducts of different techniques drawn from software or hardware domains to essentially speed up or accelerate computations.

One paradigm entails software models running on high-end supercomputer grade computing platforms like the Blue Brain [10] or Beowulf cluster [11]. Blue Brain project reportedly aims to simulate sections of the brain and uses the IBM Blue Gene/L supercomputer (360 Tera flops through 8192 PowerPC™ CPUs). The computing platform is claimed to be able to simulate 100K neurons with very complex biological models and 100 million neurons with simple biological models. The focus of this project is simulation of parts of brain through realistic and accurate models of biological neural system. In the case of Beowulf cluster which is a 27-processor machine, simulation of a thalamocortical model for one second of activity required $10^{11}$ neurons and $10^{15}$ synapses, and took nearly two months to complete. Such an approach projects high flexibility but requires hard-to-access and very expensive hardware.

Field programmable gate array (FPGA)-based approach forms the basis of a second paradigm where primary software routines are implemented in hardware for significantly accelerated computing. Although FPGA-based approach offers great flexibility,
practitioners often struggle to establish the correct system balance between processing and memory while also dealing with a harder programming aspect compared to software.

The third paradigm is the custom-built hardware which has been tried many times without notable success owing mainly to the fundamental problems which application specific integrated circuits (ASIC) possess. It has proven to be a major challenge, as evidenced by the lack of an operational system deployed in the field, to deal with the issue of deciding how much of the neural network functionality should be realized through hardware, which typically lead to the optimization of performance but the loss of flexibility.

There are some recent examples of projects that attempted to implement hybrid paradigms. The Synaptic Plasticity in Spiking Neural Networks (SP2INN) project [12] envisioned custom hardware design and prototyping for a neural network with one million neurons alongside of several million synaptic connections. The outcome is not a success and this project has been reportedly abandoned. The follow-up project, SEE, attempted to leverage FPGA-based approach with a certain level of success – it was claimed that about a half million neurons each with up to 1.5K synaptic connections could be modeled [13]. The SpiNNaker project [14] aimed at development of a massively parallel computing platform based on essentially a modified and highly-tuned system-on-a-chip (SOC) technology as to serve a neural network realization with up to a billion spiking neurons and intended to explore the potential of spiking neuron based
systems in engineered systems. However, a successful outcome apparently does not appear to have been achieved [14].

As this discussion exposed, even though there are claims suggesting that some really large number of neurons along with a very high count of synapses for each neuron can be simulated in real time, they are mostly published in academic and research literature. There is no product or system that has been deployed in the marketplace for real-life use or for the masses. This observation suggests that there is much work left to be done to carry these systems from research and development labs to practice as true products accessible to a large segment of potential user base.
Chapter 2

Background

2.1 Artificial Neural Networks

An artificial neural network (ANN), also called neural network (NN), is a mathematical model or computational model that is inspired by the structure and/or functional aspects of biological neural networks. They, in contrast to classical computers and computer programs, address problems whose solutions have not been explicitly formulated [16].

The first models of neural network go back to 1940s when two mathematicians, McCullocha and Pitts [16] suggested the description of a neuron as a logical threshold element with $L$ input channels, one output channel and two possible states. One input channel is either active or silent. The activity states of all input channels thus encode the input information as a binary sequence of $L$ bits. The state of the threshold element is
given by comparing the summation of all input signals and the threshold value, if it exceeds the threshold value, then the neuron is excited, otherwise it is in the quiescent state [17]. However, this theory failed in some aspects. The next major development occurred in 1949 when Hebb proposed a learning mechanism which is the starting point for artificial neural network learning algorithms. He postulated that when the axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing cell B, is increased [18].

2.1.1 Structure

A neural network can be defined as a (rather over-simplified) model of the most simplistic reasoning process in the animal brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called neurons [19]. Three main structures can be distinguished in a typical neuron: dendritic tree, cell body (soma) and axon as shown in Figure 2-1. The dendritic tree, a branched structure of thin cell extensions, forms the main input pathway of a neuron. It is spread out within a region of up to 400 $\mu m$ in radius around the neuron and sums the output signals of the surrounding neurons in the form of an electric potential, which it then sends to the cell body of the neuron. If the electric potential exceeds some threshold value, the cell body produces a nerve fiber conducted along the axon. The axon also branches out and passes the fiber to target neurons. The electric pulse of the axon causes secretion of a transmitter
substance and further leads to a change in the potential at the dendritic tree or cell body of the target neuron. The contacts of an axon also called synapses increase or decrease the pulses depending on its type [17].

![Figure 2-1: Structure of a Neuron and Connectivity between Two Neurons](image)

An artificial neural network consists of a number of simple processors, also called neurons, which are analogous to the biological soma in the brain, connected by weighted links passing signals from one neuron to another. The signal coming through the incoming connections, after being processed by the neuron, is transmitted through the neuron’s outgoing connection. Normally, the outgoing connection splits into a number of branches and terminate at the other neurons’ incoming connections in the network. The correspondence between the major parts of a biological neuron and its computational or mathematical model as employed in artificial neural networks is presented in Table 2.1.
Table 2.1: Analogy between Biological Neuron and its Computational Model

<table>
<thead>
<tr>
<th>Biological Neuron</th>
<th>Computational Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soma</td>
<td>Neuron</td>
</tr>
<tr>
<td>Dendrite</td>
<td>Input</td>
</tr>
<tr>
<td>Axon</td>
<td>Output</td>
</tr>
<tr>
<td>Synapse</td>
<td>Weight</td>
</tr>
</tbody>
</table>

Simplest mathematical or computation model of a neuron has weighted incoming connections that originate from the outputs of other neurons, a summation block that computes a scalar using the incoming input vector and the weight vector, and a nonlinear or activation function block that maps the scalar to a new scalar through a typically nonlinear mapping as shown in Figure 2-2.

![Figure 2-2: Diagram of Neuron Computational Model](image)
The neuron computes the sum of its weighted input signals and then transforms the value according to an activation function as shown in Equation 2.1. Step function, sign function and sigmoid functions are normally used as activation functions as shown in Figure 2-3:

\[ y = f(\sum_{i=1}^{n} x_i \times w_i), \quad (2.1) \]

where \( f \) is the activation function.

Figure 2-3: Activation Functions: (a) Sign Function (b) Step Function (c) Sigmoid Function (d) Linear Function
The neurons or processing elements connect together and compose the artificial neural network (ANN). The overall behavior is determined by the structure and the strength of these connections. There are different topologies for arrangement and connectivity of neurons, which also establish if there is any feedback for the signals flowing through the network. In one embodiment, ANN is called multilayer perceptron neural network and the associated network structure consists of the processing elements arranged in groups or layers [17]. Normally, two layers are fixed, namely input layer and output layer. Also, there is no data processing in the input layer; it only distributes the inputs to the neurons to the next layer. Unlike the input layer, the processing is done at the output layer. Therefore, in a two-layer network only one layer does the processing and this type of ANN can be used for linearly separable problems. Most real life problems are not linearly separable and hence a two-layer network cannot be used. In the literature, it is mentioned that three layer network which has a hidden layer between the input and output layer can handle most of the non-linearly-separable problems. The hidden layer provides networks with ability to perform non-linear mappings. For more complex problems, ANN developers may use trial-and-error method to decide the number of layers. An example topology for a multilayer perceptron neural network with four layers (one input, two hidden, and one output layers) is shown in Figure 2-4.
For a multi-layer network, every layer except the input layer that gets input directly takes the output from the previous layer as input; the last layer’s output is recognized as the output of the network. As denoted in Figure 2-5, $y^1$, $y^2$ and $y^3$ represent the output of each layer, $x$ is a $[n \times 1]$ input vector that is connected with all neurons in each layer by the weight matrix $W^i$ which is defined by Equation 2.2. Every layer has an accumulator, a transfer function $f$ and an output vector denoted as $y$.

$$W^i = \begin{bmatrix}
w_{i,1}^1 & w_{i,2}^1 & \cdots & w_{i,S_{i-1},1}^i \\
w_{i,1}^2 & w_{i,2}^2 & \cdots & w_{i,S_{i-1},2}^i \\
\vdots & \vdots & \ddots & \vdots \\
w_{i,1}^i & w_{i,2}^i & \cdots & w_{i,S_{i-1},i}^i
\end{bmatrix}^{(2.2)}$$

where $i$ represent the layer, $S^i$ means the total number of neurons in layer $i$, $S^0$ is the number of attributes of input data, the row number of the element in the matrix represents the connected target neuron, and the column number represents the connected input
neuron. For example $w_{1,1}^1$ means the weight that connects the first neuron in the input layer and the first neuron in the first hidden layer.

The connections between neurons in different layers propagate signals in two ways, namely feed-forward and feedback [21]. The feed-forward signals only propagate along connections in one direction, while feedback signals propagate in either direction and/or recursively. Multi-layer perceptron networks and radial basis function networks are prominent examples of feed-forward type networks, while time-delay neural networks (TDNN), Hopfield neural networks, Elman neural networks belong to the feedback or recurrent type.

Figure 2-5: Three-Layer Feed-Forward Perceptron Network
Learning in an ANN is defined as any progressive systematic change in the memory (weight matrix) and can be supervised or unsupervised [18]. Supervised learning usually employs some techniques such as error-correction learning, reinforcement learning etc.; error back-propagation and the Boltzman machine learning belong to this type. Unsupervised learning, on contrast, relies only upon local information and internal control strategies. Examples of this type include Adaptive Resonance Theory (ART) variants, Hopfield associative memory networks, bi-directional associative memory (BAM), learning vector quantization (LVQ) and self-organizing maps (SOM).

2.1.2 Application and Limitations

Neural networks can be applied to a wide variety of problems, from diagnosing breast cancer to classification of satellite imagery. According to [22], the applications are classified into four main venues as follows:

- Classifiers, Approximators and Autonomous Drivers: This type of network can respond instantaneously to the applied input.
- Simple memory and Restoration of Patterns: This kind of neural network responds to a presented pattern in time and in a very characteristic way through a gradual reconstruction of a stored pattern.
- Optimizing Networks: Optimizing problems are among the most important problems in engineering. The purpose of optimizing is to minimize certain cost functions, like cost of time, cost of money etc.

- Clustering and feature Detecting Networks: Clustering and feature detecting networks exhibit properties of self-organization. These properties are closely related to the information of knowledge representation in artificial intelligence and information theory.

While artificial neural networks are widely applied to solve many real life problems, for example game playing and decision making (poker and chess), data mining, medical diagnosis, pattern recognition (face and object identification) and etc., they also have some limitations, some inherent while others due to current implementation technologies, in many respects. Neural systems and artificial neural network algorithms are inherently parallel, while the simulations have to be on sequential machines. The processing time increases quickly as the size of the problem grows. The quality and the type of the preprocessing of the input data largely affect the performance of the network since there is strict dependence on the data. Performance is measured by statistical methods and it is not convenient for visualization in most cases. Many of the design decision required in the development are based on, at least in part, ad hoc procedures [18].
2.2 Wireless Sensor Networks

A wireless sensor network (WSN) can be described as a network of devices, denoted as node, that wirelessly communicate among themselves, cooperatively sense and may control the environment enabling interaction between persons or computers and the surrounding environment [23]. Since the start of the third millennium, wireless sensor networks generated an increasing interest from industrial and research perspectives. One of the most mentioned application types of WSN are disaster relief operation: sensors with detecting environmental parameters equipment collectively collect data of the environment and determine if a disaster is going to happen. Another noticeable application is intelligent buildings. Sensors deployed in the buildings can observe and intelligently adjust the light, temperature, airflow, humidity and other physical parameters to improve convenience and save energy. Beyond these two examples, the applications are as diverse as forest fire monitoring and detection, bio-chemical hazard detection in a volume of space, personal bio-networks for health monitoring, distributed process control, precise farming and agriculture, and more.
2.2.1 Architecture

*Single Node Architecture*

Building a wireless sensor network requires the constituting nodes, also commonly called motes, to be developed and available. An application’s requirements play an important part with regard mostly to size, costs, and energy consumption of the nodes – they may have to be small, cheap, or energy efficient; they have to be equipped with the right sensors, the necessary computation and memory resources; and they need adequate communication facilities [24]. Normally, five main components can be recognized for a basic sensor node as shown in Figure 2-6.

![Figure 2-6: Single Node Architecture](image)

Sensors/actuators are the physical interface to the world; they observe or control physical parameters of the environment. The memory is fairly straightforward; it is used to store
some immediate sensed data and relevant code. The (micro) controller is the central unit of the node, it can collect and receive data from sensors and other nodes, execute code stored in the memory, process the data and decide the next behavior-send response or keep silent. The communication device is used to exchange data between individual nodes; normally it uses radio frequencies, optical communication and ultrasound as transmission medium. Certainly, all of these components need power. Power supply is a critical system component storing energy and providing power when it is required.

**Network Architecture**

A number of nodes, which can interact with others through wireless radio, form a network. Interaction with others requires utilization of a transmission medium. The choices include radio frequencies, optical communication, ultrasound and other media like magnetic inductance. Because of the benefits of radio frequency such as relatively long range, high data rates, low error rates, reasonable energy consumption, and not requiring line-of-sight between sender and receiver, it is the most appropriate one. Usually, the frequencies between about 433 MHZ and 2.4 GHZ are used [24].

**2.2.2 Network Protocols**

The basics of radio communication and the inherent power limitation of radio communication lead to the limitation on the feasible distance between a sender and a
receiver [24]. Due to the limited range of radio transceivers, direct communication between two nodes is not always possible, so the transmission between nodes relay on others’ cooperation, which means data packets take multiple hops to travel from the source to the destination.

Figure 2-7: Multi-Hop Network (circle shows the radio range of a mote at its center)

In fact, the activity of sensing, processing, and communication under limited amount of energy, requires a cross layer design approach typically culminating in the joint consideration of distributed signal/data processing, medium access control and communication protocols [25].
**Media Access Control Protocols**

The medium access control (MAC) protocols regulate the access of a number of nodes to a shared medium in such a way that certain application-dependent performance requirements are satisfied. During last thirty years, a very large number of MAC protocols for wired, ad hoc mobile and sensor networks have been devised. These MAC protocols can be classified into three categories: fixed assignment protocols, demand assignment protocols and random access protocols [24]. The fixed assignment protocols, as the name suggests, divides the available resource among the nodes so that each node uses its resource without colliding with others. Typical protocols of this class are Time Division Multiple Access Protocol (TDMA), Frequency Division Multiple Access Protocol (FDMA), Code Division Multiple Access Protocol (CDMA), and Space Division Multiple Access Protocol (SDMA). Demand assignment protocols are similar to the fixed assignment protocols; the main difference is that the resource assignment is made on a short-term basis, especially for the duration of a data burst. Random access protocols are implemented using a random component or element, for example, randomly generating a wait time before sending a packet, and they are suitable for fully distributed computing. MAC protocols specifically developed for wireless sensor networks have a prominent and distinguishing attribute which is that these protocols must accommodate sleep-wakeup periods. Sensor MAC (S-MAC), Berkeley MAC (B-MAC), and X-MAC are duty cycled MAC protocols that use awake-sleep intervals to conserve energy.
**Routing Protocols**

In a multi-hop network, nodes rely on intermediate nodes to forward packets on its behalf, so the intermediate node has to decide who the packet should be passed to. This decision making process is embodied within routing protocols. The simplest routing protocol is flooding: send a packet to all neighbors; neighbors who get the message the very first time transmit it to all of their neighbors, and if it is not the first time then they ignore the message. Usually the message carries some kind of time expiration information to avoid looping. In this way, as long as the sender and receiver are in the same network, the message is guaranteed to reach its destination. One major problem for the flooding based protocols is the excessive energy use due to redundant messaging which is a serious impediment for their use in wireless sensor networks (WSN). More advanced protocols appropriate for WSNs are rumor routing, gossiping based routing, SPIN routing, and directed diffusion routing protocols [24].

**Time Synchronization Protocols**

Like in other distributed systems, time is an important aspect in wireless sensor networks. Because of random phase shifts and drift rates of oscillators, the local time of nodes may differ. Usually, there are two types of time synchronization protocols, one is based on sender/receiver and the other one is based on receiver/receiver. In the first kind of protocols, the sender sends a package containing time information so that the receiver can synchronize to its clock; TPSN is an appropriate example of such protocols and
developed for WSNs [24]. In contrast, the receiver/receiver synchronization allows multiple receivers of the same time-stamped packet to synchronize with each other, but not with the sender. Reference broadcast synchronization (RBS) and Hierarchy Referencing time synchronization (HRTS) belong to this type [24].

**Localization and Positioning Protocols**

Another important consideration for wireless sensor network motes at an individual level is related to localization and positioning, especially when the network is deployed for tracking or event-detection problems. In large-scale networks, manually configuring location information into each node during deployment is certainly not an option. Three main approaches exist to determine a node’s position: using neighborhood information, exploiting geometric properties of a given scenario, and comparing the characteristic properties of the position with premeasured properties. Usually, the localization consists of two important phases: distance (or angle) estimation and distance (angle) combining. Received Signal Strength Indicator (RSSI), two time-based methods Time-of-arrival (ToA) and Time-difference-of-arrival (TDoA), and Angle-of-Arrival (AoA) are main techniques used in the first phase. For the combining phase, multiteration and Maximum Likelihood (ML) estimation are the most popular choices. Determining positions in a wireless sensor network is burdened with considerable overhead and the danger of inaccuracies and imprecision [24]. More advanced material can be found in [24].
Chapter 3

Proposed Computer Architecture for Neurocomputing

We are proposing a new computer architecture for fine-grain and massively parallel and distributed hardware realization of artificial neural network algorithms towards true real-time computations of solutions for problems of very large-scale and complexity. The technology that has been emerging for the wireless sensor networks (WSN) will be leveraged to conceptualize, design and develop the new computer architecture that will serve as a true parallel and distributed processing (PDP) hardware platform for the real-time realization of artificial neural networks (ANN).

Wireless sensor networks (WSN) are topologically similar to artificial neural networks. A WSN is constituted from hundreds or thousands of sensor nodes or motes each of which typically has reasonable level of computational power (through the onboard and often basic microcontroller as the computing platform). An artificial neural network is composed of hundreds or thousands of (computational) nodes or neurons, each of which is assumed to possess only very limited computational processing capability, which is easily addressable by even the basic microcontroller onboard any given WSN mote. It is then unavoidable that the fusion of two highly parallel and distributed systems, the
artificial neural network and the wireless sensor network, is in order. In fact there is a one-to-one correspondence, in that, a sensor mote can represent and implement computations associated with a neural network neuron or node, while (typically multi-hop) wireless links among the motes are analogous to the connections among neurons.

Upon further consideration, it is also relevant to state that sensors and associated circuitry on motes are not needed for implementation of artificial neural network computations. Accordingly it is sufficient for the nodes or motes in the wireless network to possess the microcontroller (or similar) and wireless communication radio to be able to serve as a PDP hardware platform for ANN computations. Additionally since this modified version of motes and the associated wireless sensor network will not need to be deployed in the field, the batteries may be replaced with the grid or line power and hence eliminating the most significant disadvantage (e.g., limited power storage or capacity) associated with the operation of wireless sensor networks.
3.1 Wireless Processor Networks (WPN) as a PDP Computing System

A wireless processor network (WPN) is the same as a wireless sensor network (WSN) with one major exception – the processing nodes or motes in the WPN are supplied grid or line power rather than portable or battery power. The processing nodes in a WPN don’t need sensors compared to motes in a WSN. Accordingly, a wireless processor network is composed of discrete processing nodes each of which has an on-board microcontroller with ROM and RAM, a radio, and an antenna. Each processing node is ideally small in size (comparable to the size of a dime or smaller), with reasonable processing power, up to megabyte-size permanent and non-volatile storage, and a radio whose transmission parameters that are controllable at a reasonable precision and accuracy. A WPN may consist of tens or hundreds of thousands of (either homogeneous or heterogeneous) processing nodes, where any given node can communicate with other nodes through multi-hop wireless radio channels unless they are neighbors in which case one-hop communication is possible. The WSN protocols for medium access, time synchronization, positioning and localization, topology management and routing along with the middleware like the operating system (i.e. TinyOS) and application-layer tools like programming language (i.e. nesC) are mature and can be employed as appropriate in WPNs with minimal or no adaptation and modification in most cases [26,27].
Each mote in a WPN has substantial computational power due to the on-board microcontroller and can operate independent of other motes for asynchronous processing or in time synchronization with the rest of the network. Distributed code can be embedded within the local storage or memory of each mote either during the initial manufacturing phase or after deployment and through the off-the-air wireless channel of the on-board radio transreceiver. Motes can exchange their computations with other motes over the air through their radio transreceivers. Typically, reach of each antenna for uni/multi/broadcasting and reception will be constrained to a close geographic neighborhood of each mote for a number of reasons including, but not limited to, the need to reduce the interference and crowding in a given channel for the purposes of medium access control. However, contrary to what is a typical and important concern in WSNs, the power consumption is not an issue with WPNs due to availability of line or grid power. It is also highly relevant to state that a WPN mote does not need to go into sleep or power-down mode since unlimited power is available through the grid. Routing protocols would be implemented to facilitate exchange of data and information among the motes themselves and with the gateway mote, which would typically be interfaced to a powerful laptop-grade computer. It is a powerful combination when a WPN as described above with thousands or hundreds of thousands of motes is coupled with a distributed algorithm that implements a certain task that can be decomposed into a very large number of subtasks potentially with massive concurrency or parallelism for execution. Mapping such an algorithm to a WPN will result in truly and massively parallel and distributed computation, and hence is well-poised to facilitate real time solution of large-scale problems.
One prominent example of parallel and distributed family of algorithms is the artificial neural networks. A neural network is composed of a very large number of neurons (for a typical large-scale problem), each of which with identical (or similar in some cases) computational capabilities and is able to compute concurrently with the rest of the neurons in the network. The overall neural computation is composed of a very large number of similar and rather simple calculations which can be performed in massive parallelism and a fully distributed fashion. Associating a processing node in a WPN with a computational node or neuron in an ANN will naturally induce a completely and maximally parallel and fully distributed computation scheme. The wireless connectivity of the WPN results in some very desirable properties of the newly-proposed computing platform. For instance, the entire embedded neural network can be recast or redefined for its type, structure, topology, connections or parameters (weights) with minimal effort, cost and, perhaps more importantly, dynamically. This suggests that the WPN is a generic (rather than specialized or customized) hardware computing platform for neural networks, and other parallel and distributed computing tasks.

3.2 Processing Node and Neural Computations

Each processing node (PN) will host one (or more) artificial neural network neuron(s) while the one-neuron-per-mote configuration facilitates a maximally parallel and fully distributed computation. The communications among neurons, which will implement the
exchange of output values, will be over the air through the radio channels. Each PN will store the weight vector for its own embedded neuron, output values communicated from other neurons that are connected to this neuron, and any other parameters related to training or learning. The on-board microcontroller will execute the neuron computations while the both types of memory, read-write and read-only, will be utilized to store values of adaptable parameters and the constants, respectively.

Associating a processing node with a neuron in an ANN will naturally induce a maximally parallel and fully distributed computation scheme. The wireless connectivity among neurons or processing nodes results in some very desirable properties of the computing platform for artificial neural networks. For instance, the entire embedded neural network can be recast or redefined for its type, structure, topology, connections or parameters (weights) with minimal effort, cost and, best of all, on-the-fly and over-the-air.

3.3 The Novel Contribution

The novel aspect of the research project is that a fully-parallel and maximally-distributed generic hardware-based computing architecture for fine-grain computational tasks (as in for artificial neural network) is proposed. A wireless sensor or processing network (WSN/WPN) is transformed into a true parallel and distributed processing (PDP) hardware platform and configured for implementation of PDP tasks like ANNs. The proposed design, for the first time, offers a viable PDP hardware platform for ANNs (and
similar PDP tasks) and facilitates real-time computation of solutions for potentially very large-scale problems.

3.4 Scalability, Computational Cost, Messaging Complexity and Power

The time complexity of the proposed computing platform is affected by two main factors, namely the specific artificial neural network algorithm chosen and the communication requirements of the problem being addressed by the neural network. As to the first factor, there are typically two distinct phases: training that bears a substantial time cost and deployment whose time cost tends to be negligible compared to that of the training. As an example, for feed-forward neural networks, the training time is mainly dictated by the convergence properties of the specific problem being addressed, which also affects the topology of the neural network. The convergence properties of feed-forward networks vary dramatically from one problem domain to another. The empirically specified convergence criterion, i.e. one being cumulative error satisfying a user-defined upper bound, also plays a significant role in the time complexity. There will also be onboard processing time associated with implementing the neuron dynamics which is, in most cases, negligible compared to other cost elements, and hence will be ignored for the rest of the discussion. All of these costs are already inherent in the neural network algorithm regardless of its realization on a specific (hardware or software) platform.

There is, however, a new cost component due to distributed implementation of the neural network algorithm on a WPN, and this cost originates due to the need to exchange neuron
outputs among neurons each of which could be embedded in a different mote. Access to wireless communication channel will be restricted and hence delayed at times for a given neuron-mote duo per the design of the WPN and the choice of a specific medium access control (MAC) protocol. Additional delay may originate due to multi-hop routing requirements per the routing protocol of the network. However, all of these extra delay factors are offset by the fact that the motes don’t need to go to “sleep” to save energy at any time as is typical for a WSN, which would mean that the computation can be accelerated up to twenty times when compared to a 5%-95% wakeup-sleep schedule in WSNs.

The largest data structure for an artificial neural network algorithm is the weight matrix, which can quickly require on the order of Tera bytes of storage for certain neurocomputing applications. In the proposed architecture, however, the weight matrix is distributed across the WPN: each mote needs to store only one row of the matrix, namely the weight vector for the embedded neuron. All that is needed is the local (or distributed) storage of weight vectors. Another vector needs to be created locally (say within the read-write memory of microcontroller) on-board a mote to store output values of other neurons in the network. If the worst-case (or maximum) connectivity is $N$, which cannot be larger than the total number of neurons in the neural network, then the space or memory cost is $O(2 \times N)$ real numbers. This translates into $y \times 2 \times N$ bytes under the assumption that each real number requires $y$ (a small positive integer) bytes in some digital representation scheme. Typically, for many neural network algorithms, a neuron connects to a number of neurons which is much smaller than the total number of neurons.
in a given network. Accordingly, the memory space requirement or cost per mote is linear in the size of the neural network neuron count and therefore is not expected to be significant.

Another fundamental point of interest is the requirement for wireless communications that needs to be performed to exchange neuron output values among the neurons which are embedded within processing nodes or motes. Typically a given neuron on a mote will exchange messages with a small number of other neurons on (neighboring) motes (compared to the total number of motes in the WPN) which are possibly $k$-hop neighbors, where $k$ is a small positive integer, since multi-hop communications is preferable to direct mode (single-hop) for a number of reasons including, but not limited to, radio interference. The total number of messages to be exchanged will depend on a number of factors. For instance if the neural network is a multilayer perceptron with a back-propagation-type learning algorithm, the training mode would require a number of iterations for weight updates (until a convergence criterion is met). During training, first outputs of neurons in a previous layer are communicated (through the wireless channel) to the neurons in the next layer. Subsequently outputs of neurons from the next layer need to be communicated (again through wireless channel) back to neurons in the previous layer for weight updates. This feed-forward feed-backward signaling continues until a convergence criterion is satisfied, which is problem and neural network instance dependent among others.
It is relevant to note that computational complexity aspects of neural networks is a domain that is largely incomplete and fragmented although there have been noteworthy advances during the last decade [8, 29]. There are too many neural network paradigms that are substantially different and countless parameters to consider for a unified and coherent treatment of the subject. This fact led to limited number of computational complexity analyses for specific instances of neural network algorithms and associated learning processes. An illustrated message complexity analysis will be presented for a specific neural net configured for a specific class of problems in a later section. The main conclusion here, however, is the fact that message complexity is significant and must be carefully managed to be able to scale up the proposed WPN-ANN computing system.

The proposed PDP computing architecture will utilize power from the grid and not the stored and limited-capacity power (i.e. batteries) which typical WSNs utilize. Accordingly, processing nodes or motes of the WPN upon which the proposed PDP computing architecture is based will have access to continuous and uninterrupted power source. Accordingly, the proposed WPN-ANN design does not require lengthy sleep periods that typically lasts 95% or more of the time to conserve energy. Instead, this sleep period can be utilized for computation and communication by the mote, speeding up the overall distributed computation for the neural network algorithms.
3.5 Scalability of the Proposed System in terms of its Real-Time Computation

There are two main considerations that may affect the scalability of the proposed architecture. These are the computational cost and the communication cost. Due to massive parallel computation feature, the first cost element is addressed for the most part: the increase in the number of neurons is moderated by the increase in the capacity to perform more computations in parallel. As an example, considering the graph search problems, if the graph size increases, this directly translates into increasing the number of neurons in the ANN, and hence increasing the number of motes in the WPN. Since each mote (neuron) can compute in parallel with others, there is practically no increase in the amount of time needed to compute the solution of the larger size problem excluding the communication cost aspect. However the second consideration, namely the communication cost or the messaging complexity is the dominant factor and will establish an upper bound for the scale of the neural network and problem pair for all practical purposes. This is no surprise however since every physical system has operational boundaries. The more relevant issue is what these boundaries are, and an appropriate simulation study can, for the most part, provide an answer, at least from an empirical study perspective, to this question. In conclusion, it is relevant to note that the wireless networking protocol stack (and particularly the choice and settings of MAC and routing protocols), the type of neural network algorithm, and the specific domain of the application problem all matter in this respect.
3.6 The Proposed Idea and the Literature

One of the most recent surveys reports mainly on application of computational intelligence techniques for WSN and its protocols including design and deployment, localization, security, routing and clustering, scheduling and MAC, data aggregation and fusion, and QoS management, is presented in [30]. ANN applications for addressing security, routing, and MAC are reported. There are also a number of other attempts in the literature [31, 32, 33, 34, 35, and 36] that strive to bring together the WSN and ANN technologies. In some cases what has been done is to simply embed an entire neural network, say a Kohonen’s self-organizing map or multilayer perceptron network, within each and every mote. In other cases, a gateway node (often another name for a laptop-grade computing platform) calculates a centralized (non-distributed) solution through a neural network algorithm using global information and the solution is transmitted to the WSN motes afterwards [37]. None of the existing studies views the WSN as a hardware implementation platform for an ANN for massively parallel and fully distributed computation. On the other hand, the proposed design envisions down to one ANN neuron per each WPN mote, if and when maximum parallelism and distributed computing are desired, and can accommodate multiple neurons per WPN mote up to a number that is feasible through the processing power of the microcontroller onboard each mote as needed.
3.7 Mapping ANN Algorithms and Domain Problems to PDP Computing Architecture

A comprehensive set of ANN algorithms can be mapped to the proposed PDP computer architecture including feed-forward architectures like MLP and RBF, self-organization algorithms including Kohonen’s SOM, LVQ and similar, associative memory neural networks including Hopfield associative memory, BAM, ART and its many derivatives, and recurrent neural networks including Elman, Jordan, Hopfield, and simultaneous recurrent neural nets.

It is possible to map important classes of domain problems to artificial neural network algorithms embedded within a WPN hardware platform. VLSI realizations (within parallel and distributed computation framework) of many neural network-based algorithms configured for solutions of a very large spectrum of problems are detailed in [38 and 39]. This existing work can be readily leveraged towards achieving our objectives in showing the mapping of these problems to WPN-ANN architectures. The set of problems addressed in [38] is very diverse and comprehensive. For instance, a non-inclusive list covers problems from the domains of linear, quadratic, and linear complementarity problems; systems of linear equations; least squares problem; mini-max ($L_{\infty}$-norm) solution of over-determined system of (linear) equations, and least absolute deviation ($L_1$-norm) solution of systems of equations; discrete Fourier transform; matrix algebra problems including inversion, LU decomposition, QR factorization,
spectral factorization, SVD, Lyapunov’s equation for generalized matrices and PCA adaptive estimation; graph theoretic problems; and static optimization problems.
Chapter 4

Proposed Computer Architecture and Technology

4.1 Physical Packaging, Spacing and Layout

It will be necessary to package the collection of processing nodes (on the order of tens of thousands of nodes if not hundreds of thousands) in a geometric structure to address a number of important considerations. A possible realization of hardware architecture is likely to be a box with internal 3D grid-like structure to anchor the processing nodes while supplying them with the line power from the grid. The size of the box will be determined by a number of factors including, but not limited to, the number of processing nodes, the minimum spacing requirements of transreceiver antennas for wireless communications, and implications of congestion control aspects dictated by wireless sensor network (WSN) protocols. The physical radio communications aspect imposes a minimum distance limit between any two RF trans-receiver antennas. Another concern is dictated by traditional topology control issues related to wireless sensor networks. In simplified terms, if there are too many nodes within close proximity of each other, communications congestion control needs to be exercised. One option for that is to have controllable radio transmission power levels and to manipulate it to reduce the reach of the transmitter to only close neighbors around a given node. This will also help reduce
the congestion so that MAC protocols can facilitate an efficient and reliable operation for the network. It is then reasonable to assert that a volume-based geometry is appropriate since it does minimize the space needed to house the wireless network.

4.2 WSN Technology Today and Feasibility Assessment of Proposed PDP Computing Platform

We believe the fundamental building blocks for the enabling technologies, in terms of both hardware and software for implementation of the proposed parallel and distributed realization of artificial neural networks on the proposed computing platform based on WSNs, are in place [40, 9]. The analysis on the state of available commercial-off-the-shelf (COTS) components like WSN motes, embedded and distributed operating systems, and wireless networking and protocol stack demonstrates that technology infrastructure is mature and ready.

4.2.1 Realization through Custom Development

This option offers the most optimal framework for the realization of the proposed PDP computing platform since many aspects can be customized for the single intended computational task at hand, namely artificial neural network computations. The design of processing nodes can be simplified down to a level to be able to accommodate what is required by the most complex neural computing requirements. This will reduce
complexity, cost and power consumption. Radio and antenna design can further be tuned to the tasks at hand with improved communication efficiency and enhancements.

4.2.2 Realization through Integration of COTS Components

Particularly in the case of hardware, truly small processing nodes with radio which are specifically designed with very low power consumption in mind are currently on the market. For instance, the DN2510™ Mote-On-A-Chip by Dust Networks has both microcontroller (along with embedded software) and wireless/RF circuitry on a single IC. There are commercially available software and communications protocols (i.e. wirelessHART and ISA100 by ISA Standard Committee, 2009 [41]) that take into account scalability, reliable and timely routing and message delivery aspects, which testifies to the maturity of the software side of the technologies. In fact certain product literature claims 99.99% network reliability (SmartMesh™ IA-510 Intelligent Networking Platform by Dust Networks, Inc. [42]). An engineering effort to integrate all these technologies within the conceptual framework will suffice to create the computational system being proposed herein.
4.2.3 Example of the Smallest Processing Node on the Market Today

Wireless motes by Dust Networks LLC are ideal for the aims of this research since they are truly small in size (dimensions of 12mm×12mm×1mm), and have the microcontroller and radio integrated in a single IC design. The only external components that are needed to bring this IC alive are the power source and the radio antenna.

4.2.4 Processing Power and Memory Storage Requirements in Today’s Motes

The on-board memory storage capacity on each mote is a critical aspect of design. Within the context of an \( N \)-neuron neural network implementation, the maximum level of connectivity would require a single neuron to communicate with \( N-1 \) other neurons. Accordingly, every neuron and the processing node it resides on would have to store a weight vector with \( N-1 \) elements, each of which is a real number for adequate accuracy and precision in calculations. Assuming that each real element of the weight vector can be represented accurately using 4 bytes of storage (in 32-bit floating point format), then it would be necessary to allocate \( 4 \times (N-1) \) bytes for the weight vector. Additionally, each neuron or the processing node will need to store the value of outputs from \( N-1 \) other neurons. Thus there is a need for \( 2 \times 4 \times (N-1) \) bytes of local storage for the neural network output vector and the neuron weight vector combined. If the size of the wireless processing network is on the order of 1 million nodes (neurons), then each node will require \( 8 \times 1,024,000 \) bytes which is approximately 8 MB of storage. Of course this is the worst case or upper bound in the event the neural network is fully connected. The
technology for this level of memory requirements is already on the market. For instance, a commercial mote by Xbow Inc., namely Imote2™ based sensor node platform called IPR2400, has 32 MB of RAM and 32 MB of Flash memory on board. The on-board processing power (typically due to a microcontroller) for each node is expected to satisfy the requirements of calculations associated with updating even the most complex neuron dynamics models which are typically represented by discretized equivalents of first-order differential equations.

4.2.5 Are the Wireless Networking Protocols, Architectures and Software Mature?

The protocols for medium access control (MAC), positioning, localization, time synchronization, topology control, and routing developed for mobile ad hoc networks (MANET) and wireless sensor networks (WSN) are applicable for the proposed system in this research either directly or with minimal modification. In fact the existence of a number of successful commercial enterprises including the Dust Networks LLC with field-deployed WSN platforms is a testimony to the stability and maturity of software technologies to facilitate realization of proposed ideas in this proposal within rather short time frames.

In the next section, we will discuss and demonstrate, through a specific but relatively complex example, the feasibility of how to embed an ANN algorithm within a WPN for a fully parallel and massively distributed realization.
Chapter 5

Demonstration of Proposed Methodology: An Illustrated Case Study

This chapter will demonstrate the proposed parallel and distributed computer architecture for neurocomputing and the associated methodology through a illustrative case study. The Kohonen’s self-organizing (SOM) neural network will be embedded into a wireless sensor network for fully parallel and distributed computation of solutions for topological mapping problems.

5.1 Kohonen’s Self Organizing Map (SOM)

Kohonen’s self-organizing map (SOM) is one of the fundamental unsupervised neural network algorithms that was originally proposed by Tuevo Kohonen of the University of Helsinki in 1981 [16]. Its main feature is to facilitate mapping high-dimensional patterns to low dimensional representations while preserving the topological neighborhood relationships among the same patterns. This makes SOMs useful for visualizing low-dimensional representations of high-dimensional data. The SOM is also used for grouping similar objects in neighboring clusters and displaying clusters in a grid
(normally a 2-dimensional space). Numerous SOM algorithms have been applied to a
diverse set of problems in a multitude of fields from biology, engineering, economy to
astronomy [32, 38, 28, 51, and53].

SOM, as a clustering and pattern recognition technique that can group common
characteristic in large amounts of data and recognize patterns from massive data, is
widely applied in detecting, pre-diagnosis and classification problems. Some illustrative
examples include detection of forest fires and risk analysis [43], cancer diagnosis [44],
predicting the progression of medical data over time, for example prediction of cell
growth and disease dispersion [45], classification [46], soil analysis [47], remote sensing
[48], and classification of hand movements by recording EEGs during sequence of
periodic left or right hand movements [49]. Another prominent application is in analysis
of temporal or spatial data, for example as in image processing and speech recognition:
more applications of SOM for these two domains can be found in [50] and [51].

The SOM can be used to visualize and interpret large high-dimensional data sets.
Typically, applications are visualization of process states by representing the central
dependencies within the data on the map. Because of this property, SOM has been
progressively developed in numerous fields, such as medicine, agriculture, finance and so
on. In agriculture, SOM can be used to detect and analyze the risk of forest fire [52],
detect plant conditions [53] and [54], better understanding agriculture data [55]. In
finance, it can be used to predict financial distress [56], detect fraud, debt leasing [57],
understand the relationship between leverage, and performance and value and etc. It has also been used by meteorologist to study the climate change [58].

5.1.1 SOM Neural Network Fundamentals

Kohonen’s self-organizing map (SOM) is a two-layer architecture with one input and one output, which has the output-layer neurons arranged in a two-dimensional topology. Like most artificial neural networks, SOMs have two phases: training and mapping. The main idea behind the training is to cause different parts of the network to respond similarly to certain inputs. It begins with the neurons randomly generating weight vectors, which should have the same dimension with the training data. Next, the training data are input to the network, where neurons calculate their outputs according to the distance measure chosen. Euclidean distance and the dot product are the default choices in this respect. The neuron with the smallest distance, which translates into the largest output value, affect its neighbors within a range, and all these neurons update their weight vectors according to the weight update function described in Equation 5.10. The network repeats this process until it reaches a convergence point; normally it is a specified number of iterations over the training pattern set or until the topological neighborhood function shrinks to a small and predetermined value. The resulting mapping is also called calibration, since the network actually deals with unlabeled data; this process can be construed to provide a label for the category of objects.
To further illustrate, consider the SOM network presented in Figure 5-1 as an example. On the left of Figure 5-1, the input represents an $n$-dimensional data; the training data set is composed of several sets of this data $\{x_1, x_2, ..., x_L\}$ in which $L$ represents the number of data instances. The SOM network is two-layered with $n$ input nodes (which do not perform any computation but simply distribute the exact value of the corresponding component of the input pattern to neurons in the output layer) and $p$ output nodes with the weight value of the connection between the neuron $j$ and the input neuron $i$ given as $w_{i,j}$.

![Figure 5-1: Topology of a Two-Layer SOM Neural Network](image)
The weight matrix is formed as

$$W = \begin{bmatrix}
  w_1^i \\
  w_2^i \\
  \vdots \\
  w_p^i
\end{bmatrix}$$  \hspace{1cm} (5.1)

where $p$ is the number of nodes in the next layer, and

$$w_i = [w_{1,i}, w_{2,i} \ldots w_{n,i}] \text{ for } i = 1, 2, \ldots, p$$ \hspace{1cm} (5.2)

Initial values of $w_{i,j}$ are determined randomly and updated throughout the training or learning process. There are several distance metrics applicable to measure the closeness of the weight vector of a given neuron to the training pattern. Some more frequently used examples include Euclidean distance, direction cosines, inner product, and Tanimoto similarity etc. We will employ the inner product (of two vectors which are the neuron weight vector and the input pattern vector) to find the best matching unit. The output of the SOM neural network for a specific input pattern $x_i$ is calculated using Equation 5.3:

$$y = Wx_i.$$ \hspace{1cm} (5.3)

The neuron with the largest output value is the best matching unit (BMU) or the winning neuron and the neurons positioned in the affected area (which is the topological neighborhood of the winning neuron) update their weight vectors along with the BMU. Neurons in the same neighborhood of a winning neuron all respond to update or change their weight vectors to be more “like” the input training pattern being evaluated at the time. This results in a local relaxation or smoothing effect on the weight vectors of
neurons in the same neighborhood, which, through continued learning, leads to a global ordering. All this is achieved through a neighborhood function that is initialized to a value to facilitate each neuron to cover a good number of other neurons within their so-called “neighborhood”. As the training progresses, the neighborhood function is shrunk following a schedule, which, in one embodiment, may look like as depicted in Figure 5-2.

![Figure 5-2: Depiction of how the Neighborhood Shrinks with Time (t represent the time and Ne is the neighborhood node sets)](image)

Two factors that decide the time-varying behavior of neighborhood function are the learning factor and shrinking function. Three different choices for the learning function are shown in Figure 5-3.
Figure 5-3: Linear, Inv and Power Learning Functions With Initial Value 0.5 and Total Training Iterations of 100.

There are many neighborhood shrinking functions that are applicable, for example four types of shrinking functions are available in MATLAB SOM toolbox: bubble function, Gaussian function, cutgauss function and ep function.

\[
h_{ci}(t) = \begin{cases} 
 f_{step}(\sigma_t - d_{i,c}), & \text{bubble} \\
 e^{\frac{-d_{i,c}^2}{2\sigma_t^2}}, & \text{Gaussian} \\
 e^{\frac{-d_{i,c}^2}{2\sigma_t^2}} \times f_{step}(\sigma_t - d_{i,c}), & \text{cutgauss} \\
 \max\{0,1 - (\sigma_t - d_{i,c})^2\}, & \text{ep}
\end{cases} \tag{5.4}
\]
Figure 5-4: (a) Bubble, (b) Gaussian, (c) Cutgauss, (d) Ep shrinking Function Plots

In literature, two simple choices occur frequently [16]. Generally, the simpler choice of the two separates the neurons in different sets according to time $t$, and the learning factor is defined as $\alpha(t)$, where $0 < \alpha(t) < 1$. If the node $i$ belongs to the neighborhood node set $i \in N_c(t)$, then the neighborhood shrinking value is set as 1, and the shrinking performance only contains the learning factor. Or else, the neighborhood shrinking part is set to 0, meaning if the node is not a member of the neighborhood set it does not learn and its weight vector stays the same. In this case, if $h_{cl}(t)$ denotes the shrinking function, then
Another choice is the so-called “shrinking function,” which is often modeled after the Gaussian function as

\[ f(x) = ae^{-\frac{(x-b)^2}{2c^2}}, \quad (5.6) \]

where \( a, b \) and \( c \) are some real constants with \( a, b, c > 0 \) and \( e \approx 2.718281828 \). A widely applied Gaussian shrinking function is defined by [16]

\[ h_{ci}(t) = \alpha(t) \times e^{-\frac{d_{ic}^2}{2\sigma(t)^2}}, \quad (5.7) \]

where \( \alpha(t) \) is the learning factor defined by

\[ \alpha(t) = \alpha_0 \times e^{-\frac{t}{\lambda}}, \quad (5.8) \]

where \( \alpha_0 \) is the initial value of learning factor, \( t \) is the current training epoch, and \( \lambda \) is the total training iterations; \( d_{ic} \) is the distance between the node \( i \) and \( c \) (BMU), and \( \sigma(t) \) denotes the kernel width as defined by

\[ \sigma(t) = \sigma_0 \times e^{-\frac{t}{\lambda}}, \quad (5.9) \]

where \( \sigma_0 \) is the initial value of width and normally set to half of the smaller value between topology length and width, \( t \) is the current training epoch and \( \lambda \) represents the number of total iterations.
The weight update for the winning neuron and those other neurons in its affected neighborhood is described by

$$\mathbf{w}_i(t + 1) = \mathbf{w}_i(t) + h_{ci}(t) \times (\mathbf{x}(t) - \mathbf{w}_i(t)), \quad (5.10)$$

where $i$ denotes the current node, $t$ is the training iteration, and $h_{ci}(t)$ is defined by Equation 5.7. The pseudocode for the SOM algorithms is given in Figure 5-5.

---

**Step 0:** Initialize the SOM neural network  
a. Initialize weights $\mathbf{w}_i(t)$  
b. Initialize topological neighborhood parameters $a_0$.  
c. Initialize learning rate parameters $a_0$.  

**Step 1:** For each input vector $\mathbf{x}(t)$, do  
a. Compute the BMU: $\| \mathbf{x}(t) - \mathbf{w}_c(t) \| = \min \| \mathbf{x}(t) - \mathbf{w}_i(t) \|$  
b. Update the weights of BMU and its neighbors:  
$$\mathbf{w}_i(t + 1) = f(x) = \begin{cases}  
\mathbf{w}_i(t) + h_{ci}(t) \times [\mathbf{x}(t) - \mathbf{w}_i(t)], & i \in N_c(t) \\
\mathbf{w}_i(t), & i \notin N_c(t)  
\end{cases}$$

c. Go to the next unvisited input vector. If there are no unvisited input vectors left then go back to the very first one and go to Step 2.  

**Step 2:** Incrementally decrease the learning rate and the neighborhood size, and repeat Step 1.  

**Step 3:** Keep executing Steps 1 and 2 for a sufficient number of iterations.  

Figure 5-5: Pseudocode for Kohonen’s SOM Algorithm.

Since the learning is a stochastic process, the final statistical accuracy of the mapping depends on the convergence of shrinking function. It is suggested that the number of training iterations should at least be 500 times the number of network units [16] although this is more of an upper bound which is also computationally very expensive to achieve in practice.
5.1.2 Solution Quality Measures

Usually, several metrics are used to measure the performance of SOM: these may include quantization error, topographic product, topographic error and trustworthiness, neighborhood preservation, SOM distortion, and Huang’s accuracy among others [64].

Quantization error is a well-known property that is traditionally related to all forms of vector quantization and clustering algorithms. It is computed by determining the average distance of the sample vectors to the cluster centroids or prototype vectors by which they are represented. It completely disregards the map topology and alignment, and works with datasets containing missing values. The Equation 5.11 is used for the computation of this error quantity:

\[
\varepsilon_q = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{1}{d} \sum_{j=1}^{d} (x_j - w_j)^2 \right),
\]

where \( n \) is the instance number of the sample data, \( d \) represents the attribute number, \( x \) and \( w \) are the sample data and centroid data, respectively.

Topographic product is one of the oldest measures that qualify the topology preservation properties of the SOM. It is a measure for the preservation of neighborhood relations in maps between spaces of possible different dimensionality and used to check whether the
size of the map is appropriate to fit onto the dataset. If the topographic product is smaller than 0, the map is too small, or else the map is too big for the given dataset. However, this measure presents reliable results only for nearly linear datasets [61].

Topographic error is the most simple of the topology preservation measures. Unlike the topographic product, this measure requires the dataset information. It increases error when the respective best and second-best matching units are not adjacent. And the total error is normalized to a range from 0 to 1 where 0 means perfect topology preservation. The topographic error is calculated through Equation 5.12 which is given by

\[ \varepsilon_t = \frac{1}{n} \sum_{i=1}^{n} u(x_i), \]  

(5.12)

where \( n \) is the instance number of the sample data, function \( u(x_i) \) is 1 if for input \( x_i \) the first and second BMU are not adjacent, and is 0 otherwise [60].

Trustworthiness and neighborhood preservation measure, as with the topographic error, requires the dataset to compute the trustworthiness and it is similar to the computation of topographic error. For every rank order of \( k \), the set of the \( k \)-nearest neighbors in output space and input space are determined. Whenever one of the neighbors on the map is not a member of the closest neighbors in the original data space, the error counter is increased. The trustworthiness is one minus the average of these errors, where a value of 1.0 means a perfect preservation.
The neighborhood preservation is very similar to trustworthiness, only that the input and output space rankings are swapped. Both of these two measures can be applied to all vector projection algorithms and can be visualized on the map, while they cannot be used when the dataset contains missing values [62].

SOM distortion is a cost function the SOM optimizes if the neighborhood kernel radius is made constant. Accordingly, it can be decomposed into three parts as quantization error, neighborhood bias and neighborhood variance [63]. Distortion is only measurable if and when the learning factor is a constant, which is typically not the case for SOM learning algorithms.

Another measure that is used to assess and evaluate SOM solutions is Huang’s accuracy which is computed by

$$r = \frac{\sum_{i=1}^{k} n_i}{n},$$

(5.13)

where $k$ is the number of clusters detected by the map, $n$ is the total number of data points in data set, and $n_i$ is the number of data occurring in both the $i$th cluster and its corresponding true cluster. A higher value indicates better clustering results [64].
In this study, three of the measures listed above, which include the quantization error, the topographic error and Huang’s accuracy, are used as performance measures for the quality of SOM mapping.

### 5.1.3 Data Visualization

The goal of visualization is to present large amounts of detailed and raw information in summary form in order to offer a qualitative idea of the properties of the data [65]. Through processing and representation of data with charts, graphs, colors and other means, it is much easier and quicker to be able to interpret the meaning, observer patterns and discern meanings where they otherwise might be impossible to detect. Visualization techniques are divided into three groups: visualization of cluster structure and shape, visualization of components, and visualization of data on map [66].

Visualization through cluster structure and shape tries to give an idea of the relations of the clusters in the input space by coloring the similar clusters using the same color. Techniques are distance matrices, color maps and projections. Among them, distance matrix, which is a matrix of distances between neighboring map-units, is the most commonly used visualization technique to detect clusters embedded within SOM solutions. It can either hold all distances between map-units and their immediate neighbors, as in U-Matrix [67], or just a single value for each map unit, as shown in
Figure 5-6. Another related technique, color map, is introduced in [68]. The main idea of this technique is to use similar colors for similar map units. Not only it can show the clustering information, it can also be used to create color coding which takes the cluster structure into account. Projection approach uses projections of prototype vectors. Visualizing the SOM by projecting the prototype vectors to a low dimensional space and connecting each unit to its neighbors gives the map its characteristic net-like look and makes it easy to grasp its shape.

Component map technique is based on visualization of the component planes. The component plane is defined as a slice of the map, which makes it possible to visualize the spread of values of that component, and all these planes composed of the components map. Coupled with the clustering information, the component plane shows the values of the variables in each cluster. Component map of Iris dataset is shown in Figure 5-6.
Figure 5-6: U-Matrix and Component Maps for Iris Dataset: Left top is U-Matrix, the Rest Are Component Planes of Four Attributes and the Bottom Right Is Map Unit Labels.

Data on the map is another visualization technique to identify regions from the map based on familiar data samples, and to check how well the mapping has succeeded. Usually, similar data distribute nearly with each other and this leads to dense data clusters and color groupings. For example in Figure 5-7, which is the Iris data map in three dimensions, three groups can be easily observed.
In this study, the U-Matrix map and labeled component map are used to show the simulation results of MATLAB SOM toolbox, while only a variant of the component map is used to visualize the results of WSN with SOM embedded (or WSN-SOM for short). Prior to generating a U-Matrix, the node representation has to be decided; usually rectangle and hexagonal arrangement or layout are used, and then all the nodes represented by the unit size and shape will be distributed in rows and lines within a two-dimensional area, which eventually leads to the topology of the network to be defined. A node can have up to 8 neighbor cells for rectangular representation, and 6 neighbor cells for hexagonal representation. For WSN however, if mote can communicate with another mote, then the two motes are considered as neighbors. By this definition, for the WSN context, a mote can have more than 8 neighboring motes and this makes it difficult to choose mote representation for WSN-SOM problems. A variant of the labeled component
map, which labels the best matching unit (neuron) for each training pattern data, is used as shown in Figure 6-16.

MATLAB provides functions to draw three and four dimensional data maps, but most of the data sets used in this study are more than four dimensional, see Table 6.4, so data maps representation is not an option.

5.1.4 SOM Map Construction, Initialization and Training

Topology

A specific topology instantiation is specialized to the problem domain. In most cases, a two-dimensional topology is constructed. In many applications reported in the literature, the dimension choices typically induce a rectangular layout for neurons. The MATLAB SOM toolbox uses a formulation by which the number of patterns is the basis in determining the number of neurons in SOM as follows:

\[ N = 5 \times n^{0.54321}, \]  

(5.14)

where \( N \) represents the total number of neurons and \( n \) is the number of data instances. Large map option in the MATLAB SOM toolbox uses 4 times of value from Equation 5.14, while the small map option employs one fourth of the number computed through the same equation. In general, researchers may employ multiple maps and determine
through a simulation study the best one that results in a desirable value for a set of performance measures of interest.

We follow also a similar approach to determine the topology size for a given dataset. Prior studies reported on the same dataset in the literature, in-house simulations on the same dataset using the MATLAB SOM toolbox provide valuable reference points for a good choice for the topology and neuron count for a given dataset. We used the quantization error, topographic error and Huang’s accuracy to compare the solution quality among multiple solutions computed on different topologies. Table 5.1 presents the values reported in literature for all datasets used in this study and employed for the rectangular SOM map topologies in MATLAB SOM toolbox and WSN-SOM simulation studies.

Table 5.1: Map Dimensions and Neuron Counts in SOM Topology for Datasets Studied (For WSN-SOM simulations, 1 mote will be chosen as supervisory mote)

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Literature</th>
<th>MATLAB SOM Toolbox</th>
<th>WSN-SOM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Map Size</td>
<td>Number of Motes</td>
<td>Topology Field Size</td>
</tr>
<tr>
<td>Text</td>
<td>10×7</td>
<td>71</td>
<td>10×10</td>
</tr>
<tr>
<td>Iris</td>
<td>16×4</td>
<td>65</td>
<td>10×10</td>
</tr>
<tr>
<td>Wine</td>
<td>5×4</td>
<td>21</td>
<td>10×10</td>
</tr>
</tbody>
</table>

Initial Weight Values

Before generating initial values for all the weights, the data is normalized using the Min-Max normalization technique as explained by Equation 6.9 to a value in the range [0, 1]. The weights are initialized to random values between 0 and 1 [16]. This is implemented
by using the value that is generated by the MATLAB \texttt{rand()} function (without any arguments) which randomly generates a value in the range of (0, 1) with standard uniform distribution. It has been demonstrated that initially unordered vectors will be ordered in the long run [16].

**Neighborhood Function**

If the self-organizing process is started with a wide neighborhood function, it is reported that there is usually no risk for ending in metastable configurations of the map [16]. Erwin et. al. [16] stated that, if the neighborhood function is convex, there exist no stable states other than ordered ones. On the other hand, if the neighborhood function is concave, then there exists a metastable state that may slow down the ordering process by orders of magnitude [16].

In this study, a variant of Gaussian neighborhood shrinking function is employed, which uses message hop count instead of distance between motes to establish the neighborhood relationship, and is shown in Equation 5.15:

\[
h_{ci}(t) = \alpha(t) \times e^{-\frac{m_{hc}}{2\sigma(t)^2}}, \tag{5.15}
\]

where \(\alpha(t)\) is the learning function described in Equation 5.8, \(m_{hc}\) represents the message hop count from sender \(c\) to \(i\), where this information can be extracted from the message structure (see Figure 6-3), and \(\sigma(t)\) is defined by Equation 5.9.
***Training Iterations***

Determining the number of training iterations is mainly based on trial-and-error for any given dataset [86] through the MATLAB SOM toolbox. These empirically determined values reflect what should be pursued as the ideal cases and yet the PROWLER WSN simulation platform available for this research study imposes highly constraining limitations for both available memory space (which does not facilitate simulation of WSNs with more than 100 motes) and simulation time. The latter is mainly due to the PROWLER WSN simulator being implemented entirely within the MATLAB process environment, which is excruciatingly slow being an interpreted environment versus a compiled one.

In order to explore more practical lower bounds on the training iterations, we employed the MATLAB SOM toolbox on the datasets chosen for the simulation study. Additionally we run the MATLAB SOM toolbox using the same parameter setting (topology, topological neighborhood function etc.) if reported in literature so that WSN-SOM results could be compared and validated. The training is conducted in two phases. In the first phase, Kohonen’s SOM sequential training algorithm and the MATLAB SOM toolbox 2.0 simulation will be implemented using the same parameters reported in the literature. In the second phase, the simulations using MATLAB SOM toolbox and WSN-SOM will be executed applying the same parameters (due to the excessive simulation time and memory limitation of PROWLER, the parameter settings for the
second phase may be different from those for the first phase). For those datasets for which no parameter values were reported in the literature, the first phase can be eliminated.

In this study, 5 rough and 5 fine tuning iterations, which are also the default values of MATLAB SOM toolbox [79], are applied to all data sets with the exception of Alphanumeric or Text data. This way, through comparison of the MATLAB SOM toolbox results with those by the literature study, and comparison of the MATLAB SOM toolbox results with WSN-SOM results, indirect comparison between WSN-SOM results and literature results can be accomplished. Table 5.2 lists the training iterations for all datasets considered in the simulation study.

Table 5.2: Training Iteration Counts for all Simulations

<table>
<thead>
<tr>
<th>Data Set</th>
<th>First Phase</th>
<th>Second Phase</th>
<th>WSN-SOM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Literature</td>
<td>MATLAB SOM toolbox [rough, fine]</td>
<td>MATLAB SOM toolbox [rough, fine]</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>1000</td>
<td>1000</td>
<td>[900,100]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[400, 100]</td>
</tr>
<tr>
<td>Iris</td>
<td>Unknown</td>
<td>No Simulation</td>
<td>[5,5]</td>
</tr>
<tr>
<td>Wine</td>
<td>Unknown</td>
<td>No Simulation</td>
<td>[5,5]</td>
</tr>
</tbody>
</table>
Training Sequence

Two training sequences are available through the MATLAB SOM toolbox: sequential and batch [79]. In this study, sequential is applied to all MATLAB SOM toolbox simulations, and consequently, it was also used for WSN-SOM simulations. This is done to establish conformity among the simulations reported in the literature, through MATLAB SOM toolbox, and through WSN-SOM.

Convergence

Convergence happens and can be detected when the topological neighborhood function shrinks to include only the best matching unit (BMU) and no others in its neighborhood, and when the weight changes reduce to a very small value for an entire sweep or epoch over the training patterns. This is the procedure applied for both MATLAB SOM and WSN-SOM simulations.

5.2 Embedding Kohonen’s SOM into a Wireless Sensor Network

In this section, the methodology used to embed a SOM neural network into a wireless sensor network will be presented. The fundamental assumption is that WSN motes are constantly awake: no sleep schedule is needed since it is assumed that grid power is supplied to all the motes in the WSN.
5.2.1 Mapping SOM to Wireless Sensor Network

The basic principle of mapping a given SOM to a wireless sensor network (WSN) is to embed one output-layer neuron into a single mote. This is desirable for fully parallel and maximally distributed computation. Input layer neurons in a two-layer SOM network can also be embedded on a one-neuron-per-a-single-mote basis or a single mote can be designated to take over the role of the entire input layer in the SOM. If a single mote is designated to store and supply the entire set of training patterns, then it can also serve as a synchronizer for the phases of overall SOM computation, which includes deployment and initialization, training, and normal operation. Such a mote will be designated as the “Supervisory Mote” or SM for short. Each (generic) mote (GM) that is responsible for hosting a neuron will need to store the weight vector of the associated SOM neuron, will compute the output for the same neuron, and communicate the neuron output value to the supervisory mote, and update the neuron weight vector if it is the BMU or in the neighborhood of another BMU as determined by the neighborhood function.

It is worth further elaboration on the management of SOM computations on the WSN. More specifically the nature of management, if it should be completely distributed or centralized, is of interest. In a typical scenario, after every mote computes the neuron output, the next step is to choose the BMU and update weight vectors for the BMU and all other neurons in its topological neighborhood. In the completely distributed scheme,
each mote would have to broadcast its neuron output value to every other mote across the entire WSN. This method seems to be desirable from the perspective of distributed and local computing, and is relatively easy, but it requires that each mote can receive results from all the other motes. Timing this to occur in the absence of non-deterministic transmission-reception scenarios might be rather challenging since a WSN typically cannot establish a guaranteed packet delivery. Assuming this distributed approach is feasible, then upon receiving neuron output values from every other mote, a mote will, through a simple sort operation, determine the BMU. If it itself is the BMU, then it will notify others in its neighborhood for weight updates as well as updating its own weight vector and informing the SM that the weight update is completed. Note that the message complexity of this scheme is substantially increased due to every mote broadcasting its neuron output value to every other mote in the WSN.

In this study, a more efficient method (with respect to communication cost) is proposed, in which a single mote is designated as the so-called “Supervisory Mote (SM).” The SM has all the information of training data and distributes to all others, and which is also in charge of identifying and informing the BMU. When generic motes compute the output value of their corresponding neurons, they send them to the supervisory mote, which, in turn, selects only one BMU according to the values reported by the generic motes. In the event that there are ties for the winning place, then the SM randomly chooses one of them as the BMU, and communicates the outcome of this selection to the mote that houses the BMU neuron. The mote-neuron pairs in the neighborhood of the BMU mote-neuron pair are informed of the need to update their weight vectors also by means of Equation 5.10.
5.2.1.1 Initialization Phase

The entire computation starts with a signal from the supervisory mote (SM). Next, any mote in the WSN will need to be informed of the dimensionality of the training pattern by the SM to determine the dimensions of the weight vector for the SOM neuron that is embedded into it. Upon receiving this information, each mote will create a weight vector with appropriate dimensions and initialize its elements to random values as dictated by the problem context. Values of all the other parameters for the SOM neural network will need to be communicated by the SM to all motes in the WSN. Each mote will communicate with the SM to indicate the completion of the initialization phase for itself, and the SM, after being informed by (nearly) all the motes in the WSN, will broadcast to all the motes in the WSN the conclusion of the initialization phase for the SOM computation.
5.2.1.2 Training Phase

Once the initialization phase is over, the SM will start broadcasting the training patterns one at a time. After transmitting a particular training pattern, the SM will wait until most (say more than 95%) if not all of the motes respond back with the value of the corresponding neuron output. The mote-neuron pair with the maximum output value has the smallest distance between its weight vector and the current input pattern and has the potential to be the best matching unit (BMU). Once the BMU is identified by the SM, it will inform the mote with the BMU neuron through a message. Then the BMU will update its weight vector and flood a weight update request to its neighbors. Neighbors in the range as determined by the neighborhood function will also update their weights.

The neighborhood shrinking is implemented by means of message hop count, as shown in Equation 5.15. Once a message is transmitted its message hop count is increased by one by the receiving mote prior to retransmitting that message. When a neighbor receives the weight update request, it will check if the message hop count value is smaller than the shrinking hop-count value of $hop(t)$ as given by Equation 5.16.

$$hop(t) = hop_0 \times e^{-\lambda t},$$  \hspace{1cm} (5.16)

where $hop_0$ is the initial hop count value which is set to cover approximately half of the number of total motes, as defined in Equation 5.17,
\[ hop_0 = \frac{\sqrt{l_0^2 + l_1^2}}{2 \times \sqrt{(2 \times \frac{r}{t})^2 + (2 \times \frac{t_1}{r})^2}}, \quad (5.17) \]

where \( l_0 \) and \( l_1 \) are the width and length of the topology field, and \( r \) is the smallest integer that is larger than the square root of the number of motes; \( t \) represents the current training epoch; and \( \lambda \) is the total number of training iterations to be performed. More details on how to formulate Equation 5.17 had been presented in Figure 6-6.

If the condition is satisfied, then it means that the receiver mote is in the update neighborhood range of the BMU and should update the weight vector of its neuron. Otherwise, it should just retransmit the message. Once the weight updates are completed, the motes, which participated in the weight update, will inform the SM for the completion so that the SM can broadcast the next training pattern in the sequence. This process will repeat until convergence, which may be predetermined by a certain upper bound on the number of iterations on the training set or by some other criterion such as neighborhood function shrinking to a value that only includes the BMU but no others while also weight changes over the entire data set are negligible.
5.2.1.3 Deployment Phase

Upon achieving convergence or terminating the training phase for another criterion, the SOM embedded into the WSN is ready for either extracting the topological neighborhood relationships among the training patterns or process a previously unseen pattern for associating with one of the topological neighborhoods. The entry into and exit from this phase is also managed by the SM.

5.2.1.4 Routing Protocol

In a wireless sensor network packet transmissions between a sender and a receiver are realized by multi-hop communications since the radio range of a given mote is kept small due to a number of reasons including channel crowding for medium access, and energy savings requirements. A sender will typically transmit a packet to one of its so-called one-hop neighbors which will forward the packet to their own one-hop neighbors and so on until the packet reaches its destination. In such a network, the source and the intermediate mote at least need to know where the packet is destined to so that it may reach its intended destination, and this act of passing on is called forwarding or routing.

The simplest forwarding rule is flooding: The source sends packets to all its neighbors, and if it is the first time that the neighbor gets this message, it retransmits to its own
neighbors, or else it ignores it. This only-first-time transmit approach avoids packets circulating around the network endlessly. Additionally, a message might possess some expiration date information, which can be also employed to establish that the message will not be endlessly propagated in the network. Flooding algorithm may be applicable when destination is not a single mote but a large group of motes or the entire WSN. The simple flooding routing protocol is one of the most costly options in terms of the message complexity, which translates into extended computing times as well as excessive energy use for the overall WSN. In most cases of typical WSN deployments and applications, the flooding routing protocol might not be acceptable and accordingly other more efficient routing protocols must be utilized.

The routing protocol employed for this study is based on a hybrid that leverages a simple source-based routing and the flooding protocol; details are shown in Figure 6-4. This protocol performs the message transactions listed in Table 6.2. From Table 6.2, following observations can be made. Except for message type 4, which is for test use, and type 5, which is sent from the BMU to generic motes, all the other messages are either from the supervisory mote to all generic motes or from a generic mote to the supervisory mote. For the case where message type is 5 which is used by the BMU to send weight update notice to all generic motes in its topological neighborhood, one message needs to be transmitted to all the others in the network. Therefore, flooding is better suited. For the case, where a generic mote wants to send message to the supervisory mote, the source-based routing can perform the task by remembering the last transmitter and sender of all the messages.
that were received. Details of source-based routing protocol applied in this study can be viewed in Figure 6-4.
Chapter 6

Simulation Study

The simulation study aims to validate the proposed parallel and distributed processing (PDP) hardware design for artificial neural network (ANN) implementations. The focus is on demonstrating that a wireless sensor network (WSN) or wireless processing network (WPN) can, effectively serve as a PDP computing platform for ANN implementations.

The simulation study is intended to show feasibility of the proposed methodology. The aim is to simulate a wireless processing network (WPN) embedded with a specific neural network configured for a large-scale instance of a domain problem of interest in a fully-distributed and parallel-computation scenario. The focus of the simulation study will be limited to covering a representative sample of data sets employed for the Kohonen’s self-organizing map (SOM) neural network algorithm in the literature. Performance profile of the specific simulation scenarios including quality of solutions, computational complexity, messaging cost will be monitored and established. The scenarios will be created under the fundamental assumption that there is unlimited power available for the proposed PDP computing platform and no sleep periods are needed which is the most significant distinguishing attribute when compared to traditional WSNs. Although
scalability of the simulations is of importance, which is measured by the number of motes, or neurons in the case of WPN-ANN design, limitations due to the simulation platform PROWLER does not facilitate empirical exploration of the effect of this major parameter.

6.1 PROWLER – A MATLAB-Based WSN Simulator

Probabilistic wireless network simulator (PROWLER) is a wireless distributed system event-driven simulator running under MATLAB whose current target platform is the Berkeley MICA motes running the TinyOS [69]. It simulates WSN starting from the bottom physical communication layer to the top application layer supporting any arbitrary network topology. It is freely available and can be downloaded from http://www.isis.vanderbilt.edu/projects/nest/prowler. The simulation interface for the PROWLER is shown in Figure 6-1.

A survey of projects reported in the literature [79, 80, 81, 82, 83, and 84] indicates that in all cases researchers used WSNs with up to 100 or so motes, which is a major limitation of the PROWLER simulation platform. We believe memory space for the MATLAB environment and process is the source of this limitation. The impact of this limitation on the simulation study is that scalability assessment for the proposed parallel and
distributed computing platform is not possible since WSNs with mote counts larger than 100 is not feasible on PROWLER.

In PROWLER GUI, as presented in Figure 6-1, the motes are represented by small rectangles with a dot at the left top, and the size of the topology field, in other words the area where the motes are distributed, is fixed. User specified size of the topology field is scaled into default topology field size, which means the shape of the topology field (rectangle or square) cannot be shown by the PROWLER GUI. Though the topology field information cannot be seen from the GUI, it is important to calculate motes’ position and know motes’ neighbor information.
In order to simulate a wireless sensor network, a number of parameters must be initialized or set by the user. These parameters include radio definition, network topology, application description and specification of MAC layer, and the routing protocol. The parameters settings interface, which is shown in Figure 6-2, offers mainly two sets of parameters: those for the radio and signal related, and those for the MAC protocol.

Figure 6-1: PROWLER Simulation Interface GUI
6.1.1 Radio Propagation Model

The radio propagation model is used to estimate the strength of a transmitted signal at a particular point of the space, and based on this the reception condition can be defined and collisions can be detected.

The radio propagation model uses a deterministic function combined with a random fading function to evaluate the signal strength. The deterministic function is user specified, and the most frequently employed model is given by

\[ P_{\text{ideal}}(d) = P_{\text{transmit}} \frac{1}{1+d^\gamma} \]

(6.1)

where \( P_{\text{ideal}} \) represents the ideal received signal strength, \( P_{\text{transmit}} \) is the signal strength when the transmission begins, \( d \) is the distance between two nodes, and \( \gamma \) is a decay parameter with default values in the range of [2, 4] [70 and 71].
Many factors in a given environment affect the signal being communicated. These factors cause the transmitted signal to fade along a communication path. Weather, time of the day, the distance, obstacles along the most direct communication path, and interference in the frequency band are among those factors. In order to account for at least some of the
more significant factors, a fading function is defined and composed of two parts as shown below:

\[ f = [1 + \alpha(d)] \cdot [1 + \beta(t)], \]  

(6.2)

where \(d\) is the distance, the random variables \(\alpha\) and \(\beta\) have normal distribution \(N(0, \sigma_a)\) and \(N(0, \sigma_\beta)\). Only when distance changes, the \(\alpha\) part is recalculated and the \(\beta\) part reflects the time effect.

The actual signal strength is the product of the ideal received signal strength and the fading function:

\[ P_{\text{actual}}(i,j) = P_{\text{ideal}}(i,j) \cdot f(i,j), \]  

(6.3)

where \(i\) and \(j\) represents the receiver and transmitter, respectively.

In this study, the fading effect is not considered since signal fading is not relevant to and within the scope of this study, and it is assumed that all the motes broadcast or send messages with signal strength of 1.0 unit, \(P_{\text{transmit}} = 1.0\), and \(\gamma\) is set to a default value of 2.
6.1.2 Reception Condition

The PROWLER has two models defined; the simpler one determines the signal is received if the received signal strength is greater than the bottom line strength which can be specified through the GUI. If no signal has been received, the channel is set to idle, and if two transmissions overlap in time and both could be received, a collision is detected.

The second one is similar with the first one, only that a noise variance parameter has been taken into account and using the received signal strength after variance during the whole length of the transmission to compare the reception bottom line. The Signal to Interference and Noise Ratio (SINR) for receiver $i$ and transmitter $j$ is defined by

$$ SIN_{i,j} = \frac{P_{rec(i,j)}}{\sigma_n^2 + \sum_{k \neq j} P_{rec(i,k)}}, $$

(6.4)

and the total signal strength at $i$ is calculated by

$$ P_{total}(i) = \sum_k P_{rec}(i,k), $$

(6.5)

The channel is idle if the total signal strength is no larger or equal to the bottom line, and collision is detected if the SINR at the receiver is smaller than the bottom line at any time during the reception.
In this study, the motes are uniformly and randomly distributed across the two-dimensional topology, and without signal fading the minimum signal strength that can be received are set according to Equation 6.6.

\[ P_{\text{receive, min}} = \frac{1}{1 + (2 \times \sqrt{\frac{l_0}{r}})^2 + (2 \times \sqrt{\frac{l_1}{r}})^2}, \]  

(6.6)

where \( l_0 \) and \( l_1 \) are the width and length of the topology field, and \( r \) is the integer square root of the number of motes \( N \). More information on how this function is derived is presented through Figure 6-6.

### 6.1.3 MAC Protocol

A simple Carrier Sense Multiple Access (CSMA) protocol is implemented: when a node is ready to transmit, it first waits for some random time called wait time and tries to sense the channel. If the channel is occupied by some other transmission, it backs off for another random time and retries until the transmission can be performed. Both the wait time and back off time are generated as a sum of a fixed value and a random value which can be set through the GUI. There is no transmission drop off; nodes keep retrying until the transmission can be initiated; this could potentially create delay and backlogs of pending messages, so the wait time and back off time settings become very important.
There are two parameters in PROWLER where each has two fields for which the user can provide values. The two parameters are “Waiting Time” and “Back off Time”. Each has a constant base value plus a randomly determined value summed for its overall value.

The waiting time determines how long a mote should wait before trying to implement the sending action. The back off time determines, if the channel is occupied after attempting a send action, how long the mote should wait and try again. The back off time plays an important part in eliminating unnecessary trying and send-receive conflicts. If it is too small, the motes will try to check the channel too frequently, and for large and highly-connected networks it will cause too many collisions, which will also unavoidably lead to unnecessary memory use and long simulation times. If the value is too large, the messages will be delayed and computations and communications will slow down.

In this study, these four values for the two parameters are set as shown in Table 6.1 based on our own exploratory work and experience with the PROWLER simulator, the WSN protocol stack, the distributed SOM algorithm implementation, and the datasets considered.
Table 6.1: PROWLER MAC Protocol Parameter Settings (units in bit-time, 1 bit time = 1/40000 seconds)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Fix Part (in bit time)</th>
<th>Random Part (in bit time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait Time</td>
<td>200</td>
<td>128</td>
</tr>
<tr>
<td>Back Off Time</td>
<td>100</td>
<td>300</td>
</tr>
</tbody>
</table>

6.1.4 Message Formatting and Routing Protocol

PROWLER allows user-specified message structure. Normally, the message should at least contain the sender’s ID, destination ID, and the data. In this study, the message also contains the last_transmitter_ID, message_type, message_ID and transmit_hop information based on the requirements of the unique aspects of the WSN-SOM paradigm. The data structure that implements the message structure for this simulation study is presented in Figure 6-3.

```matlab
msg=struct(...
    'sender_ID',ID,...
    'destination_ID',destination_ID,...
    'last_transmitter_ID',ID,...
    'msg',infor,...
    'msg_ID',MSGN,...
    'msg_type',0,...
    'hop',0
)
```

Figure 6-3: Message Structure Employed for the Simulation Study
The last_transmitter_ID shows the last transmitter of this message and is updated until it reaches its destination. The hop field is used to maintain a count of the number of hops realized by a message in transmission. Its value is increased by one when a mote retransmits this message. The message_type field is used to identify the type of message by assigning a unique integer value that represents that type of message being carried. The message_type is set based on the contents of the message. For instance, a message that carries a training pattern and broadcast by a Supervisory Mote (SM) has message_type value of 0, while a message that is used by the SM to inform the BMU has a message_type value of 2. All message types employed in this study are presented in Table 6.2.

<table>
<thead>
<tr>
<th>MsgType</th>
<th>Sent From</th>
<th>Sent To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SM</td>
<td>GM</td>
<td>Distribute training pattern</td>
</tr>
<tr>
<td>1</td>
<td>GM</td>
<td>SM</td>
<td>Return computed neuron output to supervisory mote</td>
</tr>
<tr>
<td>2</td>
<td>SM</td>
<td>BMU</td>
<td>Inform the BMU</td>
</tr>
<tr>
<td>3</td>
<td>BMU</td>
<td>GM</td>
<td>Weight update request</td>
</tr>
<tr>
<td>4</td>
<td>BMU</td>
<td>SM</td>
<td>Test only</td>
</tr>
<tr>
<td>5</td>
<td>SM</td>
<td>BMU</td>
<td>Send label Information, used to label clusters with appropriate category names or numbers, also called calibration [22]</td>
</tr>
<tr>
<td>6</td>
<td>GM</td>
<td>SM</td>
<td>Confirm label finished, prepare next label</td>
</tr>
<tr>
<td>7</td>
<td>GM</td>
<td>SM</td>
<td>Weight Update confirmation</td>
</tr>
</tbody>
</table>
The routing protocol employed for this study is based on a hybrid that leverages a simple source-based routing and the flooding protocol. Upon receiving a message, if it is not the destination, a mote remembers the last transmitter from this source, change the last transmitter to itself and retransmit the message. Generally speaking, every mote maintains a routing table which records all the incoming message’s source ID and last transmitter ID. When the mote wants to send out a message, it first searches the destination in the source column of its routing table, if exists, then sends message out to the last transmitter who must be a neighbor of this mote. Otherwise, it floods the network with the message.

An illustrated case is presented in Figure 6.4 to better explain the routing process for the simulation study. For example, let mote 1 broadcast or flood a message to all the motes in its one-hop neighborhood or broadcast domain. After motes 2 and 3 receive this message, they add a new record with source_ID=1 and last_transmitter_ID=1 to their routing tables, and then change the last_transmitter_ID field value in the message to themselves and retransmit it. When the two neighbors of mote 2, which are motes 4 and 5 receive this message from mote 2, they also add a record with source_ID=1 and last_transmitter_ID=2 to their routing tables, and as motes 2 and 3 did, change the last_transmitter_ID to themselves and retransmit the updated message. Mote 6 does exactly the same thing as motes 3 and 4 did. Mote 6 receives the message from mote 3 so it adds source_ID=1 and last_transmitter_ID=3 to its routing table. Now assuming that motes 4 and 3 each wants to send a message to mote 1, then they search for destination_ID=1 under the source column in their respective routing tables, and
accordingly find that last transmitters are mote 2 from 4’s table and mote 1 from 3’s routing table. So mote 4 sends this message to mote 2 with destination _ID=1 and last_transmitter_ID=4. Mote 2 checks the destination and determines that is it not the destination and mote 1 is the destination. Then mote 2 searches for source_ID=1 under the source column of its own routing table and finds accordingly last_transmitter_ID=1 entry. Then mote 2 retransmits this message to mote 1. Mote 3 sends this message to mote 1 with destination_ID=1 and last_transmitter-ID=3. Since both motes 2 and 3 are mote 1’s one hop neighbors, mote 1 will receive both these messages upon transmissions by motes 2 and 3.

---

![Figure 6-4: Illustration of Routing Protocol Employed for the Simulation Study](image)

*(In routing tables, S and L denote source and destination, respectively; and in message notations, S, L and D represent sender, last transmitter, and destination, respectively.)*
As described earlier, the supervisory or control mote needs to distribute training data to all the other motes, and here the flooding algorithm becomes relevant. As in Figure 6-4, when motes send computed neuron output values back to the supervisory of control mote, source-based routing algorithm is employed.

6.1.5 Radio and Topology

Three radio definitions are included in the PROWLER simulator environment as MATLAB m-files, namely radio_channel.m radio_channel_Rayleigh_nd.m and radio_channel_sinr.m, which in turn correspond to the three propagation models described in Section 6.1.1. In addition to defining the radio propagation model, these files implement the MAC protocol explained in section 6.1.3 and invoke further events to take charge of message transmission process. For example, if a Send_Packet action is received from the application layer, then the radio definition tries to get access of the channel using the CSMA protocol, see section 6.1.3. Assuming that it gets the channel, then the radio propagates the message to other motes and at the same time, it causes a Transmission_Start event for the sender mote and a Receive_Start event for the nodes who can hear. Both the Transmission_Start and Receiver_Start events are MAC-layer events, which do not allow user access or manipulation. When the transmission is completed, for the sender, the Transmission_End event at the MAC layer invokes Packet_Sent event at the application layer to notify the sender the transmission has completed. For the receiver motes, the Receive_End event which is also a MAC layer
event compares the received signal strength with the user-specified signal strength that can be received. If the received signal strength is not smaller than the threshold or lower limit, it concludes the packet has been received and invokes Packet_Received event at the application layer and the application layer takes over from that point on.

In addition to the radio definition, two other files are necessary for a user to completely specify a simulation setup, namely topology file and application file. Topology file defines the placement of motes across the field of deployment: it is used to facilitate the distribution of all the motes in a topology field. PROWLER only supports two dimensional rectangle or square topology fields and there is no upper limit for the width or the length of the topology field. An example two-dimensional 10×10 topology with 300 motes uniformly randomly distributed is shown in Figure 6-5. The topology only affects the distance between nodes, and distances between nodes determine the lower limit of received signal. To keep a network connected, the radio range must be set to keep most of the motes connected to at least one neighbor. So whatever the topology is, changing the lower limit of the received signal strength can make the network connected.
In this study, only square and rectangle two-dimensional topologies are employed. The motes are uniformly distributed in the square or rectangular layout by breaking up the topology shape into many small rectangles and randomly placing motes in each such area. Assume that the width and length of the topology shape is \( L_w \) and \( L_l \), respectively, and the total number of neurons in the network is \( N \). The first scenario is as follows: if the square root of \( N \) is an integer \( (r = \sqrt{N}) \), then partition the topology into \( r \times r \) small squares with width and length for each partition given as, respectively,

\[
I_w = \frac{L_w}{r} \quad \text{and} \quad I_l = \frac{L_l}{r},
\]

(6.7)

As an example, consider a topology where 64 motes need to be uniformly distributed in an \( 8 \times 16 \) rectangle as illustrated in Figure 6-6. According to Equation 6.7, the width and
length of each small grid is 1 and 2, respectively. Now to place one mote in each small
grid of dimensions of 1×2, calculate the position of the motes as follows

\[ x = rand \times l_w + (i - 1) \times l_w, \]

and

\[ y = rand \times l_l + (j - 1) \times l_l, \]  \hspace{1cm} (6.8)

where \( rand \) is a uniform random number generator function with a range of (0, 1), and
\( i=1,2,... \) and \( j=1,2,... \) are the row and column indices of the partition in which the motes
need to be placed. The rows are numbered from bottom to up while the columns are
numbered from left to right.

One (or more) mote(s) is randomly placed within each grid. For example, to place a mote
within the grid located at 7\(^{th}\) row and 2\(^{nd}\) column, and assuming that the function \( rand \)
generates 0.3 and 0.5 for \( x \) and \( y \), the position of this mote is calculated as \( 0.3 \times 1 + (7-1) \times 1 = 6.3 \) and \( 0.5 \times 2 + (2-1) \times 2 = 3. \)

The topology is divided into equal-sized and rectangular grids, and at least one mote for
each grid can be positioned. However, if the number of motes does not have an integer
value for the square root of the number of nodes, then more than one mote may be placed
in one grid. A mote may have up to eight one-hop neighbors: two on sides, two at bottom
and top, and four in the grids, which are neighbors through the corners.
For neighbors at bottom and up, the maximum distance between two 1-hop neighbors can be thought as being equivalent to the maximum distance of any two points within a rectangle that has the width and the length as $2 \times l_t$ and $l_w$, respectively. So the maximum distance between a mote and its 1-hop neighbors located in the top or bottom grids is the diagonal distance given by $\sqrt{(2 \times l_t)^2 + l_w^2}$. Line 1 in Figure 6-6 exemplifies this distance.

The same is applicable for left, right, left-bottom, right-bottom, left-up, and right-up neighbors, only that the combined rectangles have different widths and lengths. For left and right neighbors the combined rectangle is $(l_t, 2 \times l_w)$, while for the rest it is $(2 \times l_w, 2 \times l_t)$. Accordingly, the maximum distance between two neighboring motes within these two rectangles are given by $\sqrt{l_t^2 + (2 \times l_w)^2}$ and $\sqrt{(2 \times l_w)^2 + (2 \times l_t)^2}$. Lines 2 and 3, respectively, exemplify these distances in Figure 6-6. It is easy to observe that the Line 3 has the largest value among the three maximum distances.

It is also likely that the square root of $N$ is not an integer. In that case, the computations will proceed as follows. First find the number $N'$ which is the largest integer smaller than $N$ and has integer square root. Then use the methodology explained above to randomly place $N'$ motes across the topology. Next, randomly assign locations for the rest of the motes. Even though some of the grids will receive more than one mote, the calculations
for the maximum distance between two-hop neighbors and consequent radio range settings remain the same.

Figure 6-6: An Example Case Explaining Partitioning a Rectangular Deployment Field into Equal-Sized Grids

The value of the maximum distance between two 1-hop neighbors is used to set the minimum signal strength that can be received for the wireless radio in the PROWLER environment. This way, we can guarantee that each mote is connected with at least one other mote and therefore keep the network connected. The minimum received signal strength is set according to Equation 6.6.

An additional use for the maximum distance between two one-hop neighbors is for initialization of message hop parameter. The diagonal distance for the entire network topology as given by Line 4 in Figure 6-6 is divided by the maximum distance between
any two one-hop neighbors. This value provides the maximum number of hops along the longest straight path between any two motes in the network. Initial value of the message hop parameter given Equation 5.17 is set to half of this value.

6.1.6 PROWLER Code for Kohonen’s SOM Implementation

A PROWLER application is a collection of event-driven service routines. Each routine is executed in response to an event generated by either interrupts (internal or external) or timers embedded within the software or computational model of the mote. Since the WSN-SOM PROWLER application is event-driven; actions can only be activated by events occurring which then may cause further events. Supported events and actions in application layer are listed in Table 6.3.

Table 6.3: Supported Events and Actions in Application Layer in PROWLER

<table>
<thead>
<tr>
<th>Events</th>
<th>Init_Application, Packet_Sent, Packet_Received, Collided_Packet_Received, Clock_Tick, Application_Finished, Application_Stopped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions</td>
<td>Set_Clock, Send_Packet</td>
</tr>
</tbody>
</table>
The overall event-driven software architecture of the PROWLER application is shown in Figure 6-7. Figures 6-8, 6-9 and 6-10 are the program flowcharts for the PROWLER application.

The execution of code in the application program is triggered by the arrival (on the time line) of an event which is passed from the simulation event queue. Depending on the type of event, a dedicated code block or routine executes. Upon arrival of an event at the application layer, the application first determines the event type and according to the type executes the matching routine as shown in Figure 6-8. For instance, if it is an initialization event and it is generated by the supervisory mote, the supervisory mote loads and normalizes the training data, and sends it to the generic mote. If, however, the event is generated by a generic mote, then the generic mote initializes the weight vector. If it is a Packet_Received or a Collided_Packet_Received event, the application first
checks to see if is the first time the message has reached this mote. If the answer is yes, then it processes the data according to Figure 6-10. Otherwise, it simply ignores it. When application runs to completion, it stores the final results to a file. The application will give an error for an unknown event.

Figure 6-8: PROWLER Application Event-Driven Execution Diagram

Another parameter which is called “Notification_Limit” is specified in the application layer code. This parameter is used when SM determines the BMU. Since message loss is an unavoidable aspect of the operation of a wireless sensor network, SM may not receive the computed neuron output values from all the motes in the network. Accordingly, this parameter is conceived to prevent the SM from waiting for too long and its value is set to 0.95 for all simulations. This value indicates that as soon as the SM receives responses
from 95% or more of the motes in the WSN, it will proceed to the next step of computations.

When a message arrives at the supervisory mote for the first time, and if the message type is 1, the results will be stored in supervisor’s memory. If 95% of motes with neurons responded, the supervisory mote chooses the BMU and then informs BMU to update its weight vector and send the label information. If the message type is 6, meaning the supervisory mote receives an update from a generic mote, the SM records this information in its local memory. The SM waits until no further update is received from
the BMU and motes in its topological neighborhood during a predetermined time period, then it sends out the next training pattern.

When a generic mote receives a message for the first time, it also first checks the message type. If it is type 0, then it calculates its neuron output using input data pattern and sends the computed neuron output back to the supervisory mote with a message type of 1. If it is type 2, meaning that the SM informs this generic mote that it is the BMU, the generic mote updates its weight vector and sends weight update request to other motes in its topological neighborhood. If it is type 3, meaning that the mote receives a weight update request from the BMU, the generic mote checks to see if it is in the current topological neighborhood of the BMU mote. If yes, then the generic mote updates its weight vector. If the message type is 5, meaning that the generic mote which is also the BMU receives label information from supervisory mote, then it attaches the label information and sending a message type 6 to supervisory mote, which informs supervisory mote label attached and ask supervisory to prepare next label. The WSN-SOM application program code is presented in Appendix B.
Figure 6-10: Application Code Execution as Driven by Message Type
6.2 Data Sets

Three datasets that differ for their number of attributes, number of clusters, number of patterns or instances, missing values, and domain are used for the simulation study. Table 6.4 lists all the dataset used for simulation and presents a brief introduction of each one. All three datasets can be retrieved from the UCI Irvine machine learning repository [72].

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Number of Attributes</th>
<th>Number of Clusters</th>
<th>Number of Instances</th>
<th>Missing Value</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>5</td>
<td>32</td>
<td>32</td>
<td>N</td>
<td>Word</td>
</tr>
<tr>
<td>Iris</td>
<td>4</td>
<td>3</td>
<td>150</td>
<td>N</td>
<td>Plant</td>
</tr>
<tr>
<td>Wine</td>
<td>13</td>
<td>3</td>
<td>177</td>
<td>N</td>
<td>Life</td>
</tr>
</tbody>
</table>

All datasets are normalized using the Min-Max normalization as defined in Equation 6.9. Min-Max normalization method is a linear transformation of the data, which maps the original value to the interval [0, 1]:

\[ x' = \frac{x - \text{min}A}{\text{max}A - \text{min}A}, \quad (6.9) \]

where \( x' \) is the normalized value, \( x \) is the actual value, and \( \text{min}A \) and \( \text{max}A \) are the minimum and maximum values of attribute \( A \), respectively.
6.2.1 Alphanumeric or Text Data Set

This data set is from [22] and entails five dimensional instance patterns representing uppercase letters in the English alphabet as well as Arabic numerals 1 through 6 as presented in Table 6.5.
Table 6.5: Encoding of Alphanumeric Symbol Set

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>x_1 1 x_2 0 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>B</td>
<td>x_1 2 x_2 0 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>C</td>
<td>x_1 3 x_2 0 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>D</td>
<td>x_1 4 x_2 0 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>E</td>
<td>x_1 5 x_2 0 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>F</td>
<td>x_1 3 x_2 1 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>G</td>
<td>x_1 3 x_2 2 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>H</td>
<td>x_1 3 x_2 3 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>I</td>
<td>x_1 3 x_2 4 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>J</td>
<td>x_1 3 x_2 5 x_3 0 x_4 0 x_5 0</td>
</tr>
<tr>
<td>K</td>
<td>x_1 3 x_2 3 x_3 1 x_4 0 x_5 0</td>
</tr>
<tr>
<td>L</td>
<td>x_1 3 x_2 3 x_3 2 x_4 0 x_5 0</td>
</tr>
<tr>
<td>M</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>N</td>
<td>x_1 3 x_2 3 x_3 4 x_4 0 x_5 0</td>
</tr>
<tr>
<td>O</td>
<td>x_1 3 x_2 3 x_3 5 x_4 0 x_5 0</td>
</tr>
<tr>
<td>P</td>
<td>x_1 3 x_2 3 x_3 6 x_4 0 x_5 0</td>
</tr>
<tr>
<td>Q</td>
<td>x_1 3 x_2 3 x_3 7 x_4 0 x_5 0</td>
</tr>
<tr>
<td>R</td>
<td>x_1 3 x_2 3 x_3 8 x_4 0 x_5 0</td>
</tr>
<tr>
<td>S</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>T</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>U</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>V</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>W</td>
<td>x_1 3 x_2 3 x_3 3 x_4 0 x_5 0</td>
</tr>
<tr>
<td>X</td>
<td>x_1 3 x_2 3 x_3 6 x_4 1 x_5 0</td>
</tr>
<tr>
<td>Y</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 0</td>
</tr>
<tr>
<td>Z</td>
<td>x_1 3 x_2 3 x_3 6 x_4 3 x_5 0</td>
</tr>
<tr>
<td>1</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 1</td>
</tr>
<tr>
<td>2</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 2</td>
</tr>
<tr>
<td>3</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 3</td>
</tr>
<tr>
<td>4</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 4</td>
</tr>
<tr>
<td>5</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 5</td>
</tr>
<tr>
<td>6</td>
<td>x_1 3 x_2 3 x_3 6 x_4 2 x_5 6</td>
</tr>
</tbody>
</table>
6.2.2 Iris Data Set

Iris data set [72] has four attributes as sepal length, sepal width, petal length and petal width. This dataset contains 3 classes of 50 instances each where each class represents a type of iris plant. One class is linearly separate from the other two, while the other two are not linearly separate from each other. A sample pattern from each class is shown in Table 6.6.

<table>
<thead>
<tr>
<th>Sepal length</th>
<th>Sepal Width</th>
<th>Petal length</th>
<th>Petal Width</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Setosa</td>
</tr>
<tr>
<td>7.0</td>
<td>3.2</td>
<td>4.7</td>
<td>1.4</td>
<td>Versicolor</td>
</tr>
<tr>
<td>6.3</td>
<td>3.3</td>
<td>6.0</td>
<td>2.5</td>
<td>Virginica</td>
</tr>
</tbody>
</table>

6.2.3 Wine Data Set

The Wine data are the result of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars [72]. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The 13 attributes which are all of continuous type are listed in Table 6.7, and sample patterns from each of the three classes in the data set can be found in Table 6.8.
Table 6.7: Wine Data Attributes List

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alcohol</td>
<td>11.03-14.83</td>
</tr>
<tr>
<td>Malic Acid</td>
<td>0.74-5.8</td>
</tr>
<tr>
<td>Ash</td>
<td>1.36-3.23</td>
</tr>
<tr>
<td>Alcalinity of Ash</td>
<td>10.6-30</td>
</tr>
<tr>
<td>Magnesium</td>
<td>70-162</td>
</tr>
<tr>
<td>Total Phenols</td>
<td>0.98-3.88</td>
</tr>
<tr>
<td>Flavanoids</td>
<td>0.34-5.08</td>
</tr>
<tr>
<td>Nonflavanoid phenols</td>
<td>0.13-0.66</td>
</tr>
<tr>
<td>Proanthocyanins</td>
<td>0.41-3.58</td>
</tr>
<tr>
<td>Color Intensity</td>
<td>1.28-13</td>
</tr>
<tr>
<td>Hue</td>
<td>0.48-1.71</td>
</tr>
<tr>
<td>OD280 / OD 315 of diluted wines</td>
<td>1.27-4</td>
</tr>
<tr>
<td>Proline</td>
<td>278-1680</td>
</tr>
</tbody>
</table>
Table 6.8: Sample Pattern for Each of the Three Classes in Wine Dataset

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alcohol</td>
<td>14.23</td>
<td>12.37</td>
<td>12.86</td>
</tr>
<tr>
<td>Malic Acid</td>
<td>1.71</td>
<td>0.94</td>
<td>1.35</td>
</tr>
<tr>
<td>Ash</td>
<td>2.43</td>
<td>1.36</td>
<td>2.32</td>
</tr>
<tr>
<td>Alcalinity of Ash</td>
<td>15.6</td>
<td>10.6</td>
<td>18</td>
</tr>
<tr>
<td>Magnesium</td>
<td>127</td>
<td>88</td>
<td>122</td>
</tr>
<tr>
<td>Total Phenols</td>
<td>2.8</td>
<td>1.98</td>
<td>1.51</td>
</tr>
<tr>
<td>Flavanoids</td>
<td>3.06</td>
<td>0.57</td>
<td>1.25</td>
</tr>
<tr>
<td>Nonflavanoid phenols</td>
<td>0.28</td>
<td>0.28</td>
<td>0.21</td>
</tr>
<tr>
<td>Proanthocyanins</td>
<td>2.29</td>
<td>0.42</td>
<td>0.94</td>
</tr>
<tr>
<td>Color Intensity</td>
<td>5.64</td>
<td>1.95</td>
<td>4.1</td>
</tr>
<tr>
<td>Hue</td>
<td>1.04</td>
<td>1.05</td>
<td>0.76</td>
</tr>
<tr>
<td>OD280 / OD 315 of diluted wines</td>
<td>3.92</td>
<td>1.82</td>
<td>1.29</td>
</tr>
<tr>
<td>Proline</td>
<td>1065</td>
<td>520</td>
<td>630</td>
</tr>
</tbody>
</table>

6.3 Simulation Study & Results

For each dataset, SOM solutions reported in the literature, those through in-house simulations using the MATLAB SOM toolbox, and finally those using the WSN-SOM platform are all utilized for a comparative performance analysis.
6.3.1 SOM and PROWLER Parameter Settings

This study uses the PROWLER integrated carrier sense MAC protocol. Two important parameters related to MAC protocol is the back-off time and waiting time. Both of these parameters are composed of a fixed part and a random part. In this simulation, the back-off time is set to [100, 30] and the waiting time is set to [200, 128]. The purpose of the random part of the waiting time not being set as an integer is to increase time difference and try to avoid the circumstances where a mote tries to send a message at the same time with other motes. These parameter settings are established through empirical study. Because of the uncertainty associated with wireless transmissions, 5% message lost is allowed for all simulations. This means that the supervisory mote proceeds as soon as it receives responses from 95% of motes in the WSN. Gaussian shrinking function and sequential training are employed for both PROWLER and MATLAB SOM toolbox simulations. Detailed parameter settings can be found in Table 6.9.
Table 6.9: Parameter Settings for PROWLER Simulations

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology Field</td>
<td>$10 \times 10$</td>
<td>See Table 5.1.</td>
</tr>
<tr>
<td>Number of Motes</td>
<td>Table 5.1</td>
<td>Values based on empirical work with MATLAB SOM Toolbox on the same datasets</td>
</tr>
<tr>
<td>Neighborhood Function</td>
<td>Mess-Gaussian</td>
<td>The same for all datasets as defined by Equation 5.15</td>
</tr>
<tr>
<td>Neighborhood Size</td>
<td>$\sigma(t)$</td>
<td>The same for all datasets as defined by Equation 5.9</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>$\alpha(t)$</td>
<td>The same for all datasets as defined by Equation 5.8</td>
</tr>
<tr>
<td>Initial Weights</td>
<td>(0,1)</td>
<td>All dataset are normalized using Equation 6.9</td>
</tr>
<tr>
<td>Order of Input Vectors</td>
<td>Sequential</td>
<td>The same for all datasets</td>
</tr>
<tr>
<td>Termination Condition</td>
<td>Iteration</td>
<td>See Table 5.2</td>
</tr>
</tbody>
</table>

6.3.2 Text or Alphanumeric Symbols Dataset

A solution using SOM for this dataset was reported in [22], where a 70-neuron network was employed to map the five-dimensional data vectors labeled through A to 6 to two dimensions. The data is shown in Table 6.5.
6.3.2.1 Solutions Reported in Literature

Figure 6-11 and 6-12 are results from [22] after 10,000 training steps. The network used in [22] consists of 70 neurons distributed in a rectangle as in Figure 6-11. Calibration was performed by labeling supervised labeling of array neurons in response to a specific known vector from the training set.

![Figure 6-11: Calibrated Training Results After 10,000 Steps](image)

As Figure 6-11 shows all the patterns are recognized and a unique BMU neuron is assigned to each symbol. Patterns with “similar” attributes have been grouped into a cluster. For patterns A, B, C, D and E (where two consecutive patterns differ in only one attribute value) are grouped at the top left corner of the map. The same happens to groups of patterns representing 1 through 6, W through Z, S and V, F and G, (Q and R), and others. The minimum spanning tree is derived from the original data set, by
connecting all the points with its closest neighbor. Comparison of Figures 6-11 and 6-12 shows that they have the same structure [22].

![Diagram of Minimum Spanning Tree Representation of SOM Mapping of Alphanumeric Dataset](image)

**Figure 6-12:** Minimum Spanning Tree Representation of SOM Mapping of Alphanumeric Dataset in Cited Literature Study [22]

### 6.3.2.2 Solutions with MATLAB SOM Toolbox

The data set has been processed through the SOM algorithm `som_read_data` and `som_normalize` functions in MATLAB SOM toolbox [79] using the same topology (7×10), but for two different iteration counts of 10,000 (9,000 for rough and 1,000 for fine tuning) and 500 (400 for rough and 100 for fine tuning). During rough training, the
neighborhood radius shrinks from 3 to 1, and stays at 1 during the fine tuning. The results for these two simulation cases are the same and presented in Figures 6-13 and 6-14, where the quantization and topographic errors are 0.321 and 0.094, respectively.

Figure 6-13: Simulation Results Using MATLAB SOM Toolbox for Alphanumeric Dataset

From the U-Matrix map in Figure 6-13, it is hard to see a clear boundary for the separation of clusters. The reason may be that each cluster only has a few nodes for the most part. In the component map in Figure 6-14, there are labels for 28 patterns: labels for J, R, 1 and 6 are not represented. Individually verifying the BMU for these patterns provides the pattern-specific component maps in Figure 6-14 (left). Comparison of Figures 6-14 and 6-15 indicates that certain neurons act as the BMU for more than one
pattern. This exclusively happens for those patterns, which are “similar” to each other: only value of one attribute differs in all cases. The BMUs for I and J, R and Q, 1 and X, and 5 and 6 are the same.

Figure 6-14: Component (left) and Histogram (right) Maps for Alphanumeric Dataset
Figure 6-15: Histogram Maps for Pattern J (top left), R (top right), 1 (bottom left) and 6 (bottom right)
6.3.2.3 Solutions with WSN-SOM

The network is designed as 70+1 neurons (1 as supervisory mote) distributed randomly in a 10×10 deployment field to facilitate comparison of solutions among those reported in the literature, and computed by the MATLAB SOM Toolbox and the WSN-SOM. Gaussian shrinking function is employed as the topological neighborhood function, and the sequential data sequence is used for training. Results of the simulation are presented in Table 6.10.

Table 6.10: Simulation Study Results for Alphanumeric Dataset With 5% Message Loss Tolerance

<table>
<thead>
<tr>
<th>Simulation Study Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>500</td>
</tr>
<tr>
<td>Quantization Error</td>
<td>0.278</td>
</tr>
<tr>
<td>Topographical Error</td>
<td>0.031</td>
</tr>
<tr>
<td>Incoming Message Count (average per mote)</td>
<td>61,537</td>
</tr>
<tr>
<td>Outgoing Message Count (average per mote)</td>
<td>12,912</td>
</tr>
<tr>
<td>PROWLER Simulation Time</td>
<td>52.89 hours</td>
</tr>
</tbody>
</table>

Figure 6-16 is the PROWLER GUI representing the solution computed by the WSN-SOM, where the rectangles represent the motes, and the motes IDs make it difficult to recognize the pattern or cluster information. Figure 6-17 presents a revised form of the component map to clarify the representation. In Figure 6-17, all patterns are assigned a neuron as the BMU in the SOM mapping, and the patterns X and 1 are represented by the
same neuron. A minimum spanning tree (MST) representation of the SOM mapping in Figure 6-17 is drawn by connecting all the motes (neurons) and their closest neighbors as shown in Figure 6-18. The MST suggests that the WSN-SOM solution is able to preserve the topological neighborhood of patterns in the two-dimensional mapping as they appear in the high dimensional attribute space.

Figure 6-16: Simulation Results for Alphanumeric Dataset by PROWLER GUI (a label starts with *; ‘X1’ means the node is selected as the BMU for both patterns X and 1; and other numbers above the mote represent the mote ID).
Figure 6-17: WSN-SOM Component Map with Label Information for Alphanumeric

Figure 6-18: Minimum Spanning Tree of WSN-SOM Results
WSN-SOM results have lower quantization error and lower topographic error compared with the MATLAB SOM toolbox solutions. This means that the WSN-SOM can better represent the original data, and the weights after training using the WSN-SOM are closer (in the Euclidean distance sense) to the original data patterns than those obtained through the MATLAB SOM toolbox solution. Using the MATLAB SOM toolbox, there are four patterns that are mapped to the same BMU with at least another different pattern, while there is only one such double-representation by a single BMU in the WSN-SOM solution. However, these observations do not reflect generalized observations or claims since either solution can be further improved through fine-tuning the training process.

Further comparison of Figures 6-12 and 6-18 shows that the two MSTs present “similar” topological structures. One minor exception is that patterns for X and 1 are represented by the same neuron or BMU, and 2 is grouped with the (W, Y, X, Z) branch instead of grouping with (3, 4, 5, 6). In conjunction with this, note from the data in Table 6.5 that pattern for 2 differs in only one attribute from the patterns for 1, 3 and X, so 2 being grouped with the (W, Y, X, Z) branch is not a surprise. Recognizing that solutions are affected by a multitude of factors including initial conditions and parameter settings, choice of topological neighborhood function and rate of shrinkage, training procedure and order of patterns, etc. the observed differences among the three solutions due to the literature study, in-house MATLAB SOM toolbox and the WSN-SOM are explainable rather easily.
Finally, it must be noted that Huang’s accuracy is not discussed for the text data in any literature reference identified and therefore is not considered for the comparative analysis for this dataset.

6.3.3 Iris Dataset

6.3.3.1 Solutions Reported in Literature

Figure 3.18 in [74] shows the results of a 2D 10×10 SOM map of the Iris data set, which uses Gaussian shrinking function and sequential training but does not provide or report any learning rate, number of iterations, quantization error or topographic error information. From the results, the Setosa is linearly separated from the other two, and there is clear separation line between the other two species.

6.3.3.2 Solutions with MATLAB SOM Toolbox

The same data set has also been processed through the MATLAB SOM toolbox using the `som_read_data` and `som_normalize` functions [79]. The same topology of 10×10 is also used for the MATLAB SOM toolbox solution, and the results after 5 rough training and 5 fine tuning are shown in Figures 6-19 and 6-20. The quantization and topographic errors, and Huang’s accuracy are 0.337, 0.007 and 0.887, respectively.
Figure 6-19: U-Matrix (top left) and Component Maps Using MATLAB SOM Toolbox for Iris Dataset

Figure 6-20: Component Map with Label Information for Iris Dataset
From the U-Matrix map in Figure 6-19, it is easy to see the top three rows form a well-defined cluster, which appears to represent the Setosa subspecies. The other two subspecies Versicolor and Virginica are mixed in the other cluster; the U-Matrix map does not show clear separation of them. But by looking at the Labels map in Figure 6-20, it seems they correspond to two different parts of the cluster. The component map of petal length and petal width as shown by the last two maps in Figure 6-19 are closely related to each other. Combining the four Component maps with the Labels map in Figure 6-20, it can be found that Setosa has small petals (small petal length seen from component map of petal width and small petal width seen from component map of petal width), and short and wide sepals (seen from component map of sepal length and sepal width). From the maps, no noteworthy difference can be observed as the distinguishing factor when comparing Versicolor with Virginica. In fact, the only distinguishing factor for these two subspecies is that Virginica has bigger leaves.

The [74] does not provide or report any quantization error or topographic error information. However from Figure 3.18 in [74] and the labeled component map in Figure 6-20, it can be concluded that the SOM solutions reported in the cited literature study and through the in-house work using the MATLAB SOM toolbox have similar structures. In other words, there are reasonably defined separation lines between Setosa and Versicolor, and between Versicolor and Virginica.
6.3.3.3 Solutions with WSN-SOM

The MATLAB SOM toolbox uses the complete dataset as the training data and also as the test data. To be able to compare the results with MATLAB SOM Toolbox implementation, no separation of training and test data is implemented. The WSN deployment area has the dimensions of 10×10 and motes are randomly positioned using a uniform distribution. The results are shown in Figure 6-21 and Table 6.11.

Table 6.11: Simulation Study Results for Iris Dataset with 5% Message Loss Tolerance

<table>
<thead>
<tr>
<th>Simulation Study Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>10</td>
</tr>
<tr>
<td>Number of Motes</td>
<td>101 (1 supervisory mote)</td>
</tr>
<tr>
<td>Quantization Error</td>
<td>0.690</td>
</tr>
<tr>
<td>Topographical Error</td>
<td>0.288</td>
</tr>
<tr>
<td>Huang’s Accuracy</td>
<td>0.973</td>
</tr>
<tr>
<td>Incoming Message Count (average per mote)</td>
<td>90,871</td>
</tr>
<tr>
<td>Outgoing Message Count (average per mote)</td>
<td>12,593</td>
</tr>
<tr>
<td>PROWLER Simulation Time</td>
<td>72.03 hours</td>
</tr>
</tbody>
</table>

Figure 6-22 is another representation of Figure 6-21 by removing the rectangles representing the motes, and uses square, five-pointed star and circle for Setosa, Versicolor and Virginica, respectively, and dot for the motes not chosen as the BMU for any data pattern.
Figure 6-21: Iris Simulation Results Shown in Prowler GUI (1: Setosa, 2: Versicolor, 3: Virginica, label information starts with *, ‘*233’ means the node has been chosen as BMU for 3 times; one time for Versicolor, and two times for Virginica)

Figure 6-22: WSN-SOM Results Component Map (square: Setosa, five-pointed star: Versicolor, circle: Virginica, dot: not chosen as the BMU for any training pattern)
In Figure 6-21, there are some motes chosen as the BMU for more than one pattern belonging to different classes. For example, in the middle of the map, there is a mote which was selected as the BMU for patterns from classes 2 and 3. For these situations, the following procedure is implemented to disambiguate the class this mote represents. First, the average distances from this mote’s weight vector to all the data instances in classes 2 and 3 are calculated. The mote is considered to represent the class with which it’s weight vector has the smaller average distance. If the mote is chosen as the BMU for more than one time for the same pattern, then only one circle with the color representing that pattern will be drawn.

Two lines are drawn in Figure 6-22 to separate the three species. The Setosa is linearly separated from the other two species, while for the two species, there is nonlinear separation boundary with most Virginica (circle) being above Versicolor (five-pointed star).

Comparing the results of WSN-SOM with those through the MATLAB SOM toolbox, for both of them, Setosa is clearly separated from the other two, and there is reasonable defined separation line between Versicolor and Virginica. Smaller quantization error of MATLAB SOM toolbox results indicates that the final weight are more closer to the original data, and the MATLAB SOM toolbox has smaller topographic error which means the topology preservation is better than WSN-SOM results. The WSN-SOM results have larger Huang’s accuracy meaning it has better clustering performance than
MATLAB SOM toolbox. And the MATLAB SOM results have similar structure with that from literature, so the WSN-SOM results have no worse cluster performance than those in the cited literature for Iris data.

6.3.4 Wine Dataset

6.3.4.1 Solutions Reported in Literature

The results using “The Kohonen Package for R” using a 4×5 SOM topology with Gaussian shrinking function and initial learning rate of 0.5 for the wine data are shown in Figures 1 in [78]. The same figure uses different colors to represent different attributes, and the area of the colored shape represents the value of that attribute the color represents. From Figure 1 in [78], the high hue and OD ratio instances are mainly distributed at the right side of the map, and the left side tends to have larger value of nonflavanoid phenols. The instances locate at the bottom two rows contains higher alcohol, and the left corner contains higher malic acid. It does not show any clustering information. Topographic error and Huang’s accuracy values are not reported.
6.3.4.2 Solutions with MATLAB SOM Toolbox

The same data set is processed by the Kohonen’s SOM algorithm as implemented in the MATLAB SOM toolbox for a 4×5 topology. The results after 5 rough training and 5 fine tuning iterations are shown in Figures 6-23 and 6-24. The quantization error is 2.280, the topographic error is 0.000, and the Huang’s accuracy is 0.972.

The U-Matrix map in Figure 6-23 shows three clusters, which are color coded as dark blue, light blue and red, which represent classes 2, 1 and 3, respectively. Combining all the component maps in Figure 6-23 and Figure 6-24, most patterns in class 2 contain lower alcohol, lower malic acid, low-mid Magnesium, low level proline and the color intensity is low. Figure 6-24 shows that patterns 1 and 3 are mainly distributed at the bottom right and top right, while the main feature of these two patterns is that they contain mid-to-high alcohol, ash and magnesium.
Figure 6-23: U-Matrix (top left) and Component Maps Using MATLAB SOM Toolbox for Wine Dataset

Figure 6-24: Component Map with Label Information for Wine Dataset
Comparing Figure 1 in [78] with component map for each attribute in Figure 6-23, it is reasonable to assert that they have similar structures. Unfortunately, there is no class information reported in [78], so it is not possible to compare the classification performances of the study reported in the literature and the MATLAB SOM toolbox. Larger quantization error of MATLAB SOM toolbox indicates the solution reported in literature represents smaller distances between the weight vectors of corresponding BMUs and the original data pattern.

6.3.4.3 Solutions with WSN-SOM

Simulation results for the same data set on the WSN-SOM platform along with parameter values are shown by Figures 6-25 and 6-26, and Table 6.12.

<table>
<thead>
<tr>
<th>Simulation Study Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>10</td>
</tr>
<tr>
<td>Number of Motes</td>
<td>21 (1 supervisory mote)</td>
</tr>
<tr>
<td>Quantization Error</td>
<td>0.507</td>
</tr>
<tr>
<td>Topographical Error</td>
<td>0.028</td>
</tr>
<tr>
<td>Huang’s Accuracy</td>
<td>0.678</td>
</tr>
<tr>
<td>Incoming Message Count (average per mote)</td>
<td>47,255</td>
</tr>
<tr>
<td>Outgoing Message Count (average per mote)</td>
<td>11,940</td>
</tr>
<tr>
<td>PROWLER Simulation Time</td>
<td>120.47 hours</td>
</tr>
</tbody>
</table>
Figure 6-25: Simulation Results for Wine Dataset as Shown by PROWLER GUI

Figure 6-26: WSN-SOM Results Component Map (square: class 1, five-pointed star: class 2, circle: class 3 and dot: does not represent any class)
Figure 6-26 is another representation of Figure 6-25 using the same method derived in Figure 6-22. Motes chosen as BMU for more than 1 pattern only represent the pattern that has smaller average distance from all data instances in that class. Two lines are drawn in Figure 6-26 to separate the three classes and minimize the fault, there are clear separating boundaries among the three clusters; the class 1 is mainly positioned at left bottom corner, while the class 2 is occupies the top half of the topology, and class 3 patterns are in the second quadrant for the most part.

Through further analysis, there are 49 patterns belonging to class 2 while being recognized as members of class 3, and 8 class 3 patterns are recognized as class 2 members. Consequently the Huang’s accuracy is calculated as

\[
1 - \frac{(49 + 8)}{177} = 0.678
\]

Comparing Figure 6-24 of MATLAB SOM toolbox results with Figure 6-26 of WSN-SOM results, it can be asserted that they have similar structures. One major difference is that in Figure 6-24, there are more motes or neurons representing class 1, while in Figure 6-26, only two motes or neurons represent class 1. The WSN-SOM results have smaller quantization error, meaning its results are more like and closer to the original data. The WSN-SOM results have smaller Huang’s accuracy than those using MATLAB SOM toolbox which means there are more misclassified instances using WSN-SOM. Larger topographic error for WSN-SOM means the topology preservation is not good as using MATLAB SOM toolbox.
6.3.6 Summary and Conclusions of Simulation Study

Simulation study entailed comparative performance assessment of an illustrative instance of the proposed methodology or design which embodies a Kohonen SOM neural network embedded within a wireless sensor network on three problems or datasets, namely text, Iris and Wind, each with different number of attributes, instances and from different domains. Simulation work was performed on PROWLER wireless sensor network simulator where a SOM algorithm that was embedded into a wireless sensor network was executed in fully parallel and distributed computation mode. Similar studies reported on the same datasets in the literature were leveraged for the comparative performance analysis. MATLAB SOM toolbox was utilized to validate solutions reported in the literature for these three datasets as was as to explore the parameter space in preparation for the SOM algorithm implementation on the wireless sensor network.

Simulation results indicate that the WSN-SOM design is able to compute competitive quality solutions for all three problem domains as assessed by three common performance metrics which include the topological error, quantization error and Huang’s accuracy. Based on the simulation study results, it can be concluded that the proposed WSN-ANN parallel and distributed computing architecture is feasible.

It is also relevant to note a number of limitations of the simulation platform PROWLER that posed a challenge to further progress with the research work. The analysis for
scalability and message complexity was not possible due to severe limitations imposed by the PROWLER simulator where WSNs with more than 100 motes either crashed with “out of memory” error or failed to terminate even after weeks of simulation time. Simulation time through PROWLER even for successfully runs turned out to be a notable obstacle in terms of a more comprehensive empirical analysis since the time it took for one episode of training the SOM was in the range of 52 hours to 168 hours depending on the datasets employed among the four studied. In conjunction with that lengthy simulation times for even small scale datasets, PROWLER simulations often crashed with an “out of memory” error if a larger instance count or attribute count dataset was employed. Many of these challenges are expected however given the emerging nature of the WSN field and the complexity of associated simulations. With the availability of scalable and stable simulators in the near future, these issues can be studied with relative ease and are therefore designated as future study recommendations.
Chapter 7

Conclusions and Recommendation for Further Study

7.1 Research Study Highlights and Conclusions

In this study, a parallel and distributed computing platform based on a wireless sensor network for neural networks is proposed and described. A case study based on simulation of a wireless sensor network, which employed Kohonen’s self-organizing map and three datasets from the literature was performed. The goal of the simulation study was to establish the feasibility of the proposed parallel and distributed computing architecture. A MATLAB-based wireless sensor network simulator, PROWLER, was employed for the simulation study.

A comparative performance assessment using the reported studies in the literature, in-house simulation using the MATLAB SOM toolbox, and the PROWLER for each dataset was conducted. In all cases, the proposed parallel and distributed neurocomputing platform, called WSN-SOM, based on the wireless sensor network was able to compute solutions to the problems associated with each dataset. Furthermore, these solutions
computed by the WSN-SOM design were comparable to those reported in the literature and obtained through in-house MATLAB SOM toolbox based on the three solution quality metrics, which included quantization error, topological mapping error, and Huang’s accuracy. In conclusion, the simulation study demonstrated the feasibility of the proposed parallel and distributed neurocomputing platform based on the wireless sensor networks.

7.2 Further Study Recommendations

There are many opportunities for further exploration. For instance, other neural network algorithms and problem domains from classification, regression, function approximation are among those. Specifically, fundamental neural network paradigms including multilayer perceptron, radial basis function, adaptive resonance theory, recurrent neural networks, and associative memory will need to be embedded into the proposed computing architecture for a comprehensive study. Data sets from different problem domains should be employed to comprehensively profile the performance of the proposed neuro-computing architecture. Scalability study with respect to time and message complexity is essential for a wireless network based computing platform. The effect of increasing mote counts on the real-time computation performance and the message complexity will need to be assessed to understand the limitations of the proposed parallel and distributed computing framework.
References


Internet: www.nsi.edu/users/izhikevich/interest/index.htm [2005].


Appendix A

User’s Manual and Installation Guide

System Requirements and Installation

The experiments in this study use PROWLER which is a MATLAB-based simulator to simulate wireless sensor networks, and hence MATLAB and PROWLER are required.

- Operating System
  
The experiments in this study are all under Microsoft Windows XP Professional version 2002 service pack 3 operating system.

- MATLAB
  
MATLAB 2011 b is used in this study. It can be purchased from the http://www.mathworks.com/products/matlab/.

- PROWLER
  
The PROWLER v1-25 is used in this study can be downloaded from http://www.isis.vanderbilt.edu/projects/nest/prowler/.

- System Requirements
To install MATLAB, at least 16 MB memory is required, and PROWLER requires at least 2 MB memory. But to execute the experiments in this study, 3 GB or larger memory is suggested.

- Installation
  First, install MATLAB according to the install guide. And then unzip the PROWLER.zip file and copy all the files to MATLAB root directory. Type `prowler` in the MATLAB command window, PROWLER GUI should pop up if PROWLER is successfully installed.

**User’s Manual**

Three files named `som_application.m`, `som_animation.m` and `som_topology.m` can be found in Appendix B. To execute the experiments in this study, first copy all these three files into MATLAB root folder. And then open `register_applications.m` file, two variables named `application_name` and `radio_name` are defined in this file, to register the application, add `som` to `application_name` variable.

Type `prowler` in MATLAB command window, the PROWLER GUI will pop up, select `som` in the drop down list under application name. Click the parameter button in the simulation group, and set the parameters according to Table 6.9. Click the start button on the GUI to run the simulation.
Appendix B

**som_application.m**

function application(S)

S; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S; persistent app_data
S; global ID t
S; [t, event, ID, data]=get_event(S);
S; [topology, mote_IDs]=prowler('GetTopologyInfo');
S; ix=find(mote_IDs==ID);
S; if ~strcmp(event, 'Init_Application')
S; try memory=app_data{ix}; catch memory=[]; end,
S; end
S;

clear global Control;
clear global Pattern;
clear global InData;
clear global EndIteration;
clear global Dimension;

global Control;
global Pattern;
global MoteNum;
global InData;
global EndIteration;
persistent MSGN;
global Dimension;
global Percent;
global MaxHops;
global InitialRate;
global Pattern_Name;

InitialRate=0.9;
MaxHops=3;
Percent=0.95;
Dimension=13;
Pattern=177;
Control=MoteNum;
EndIteration=Pattern*10;

switch event
  case 'Init_Application'
    signal_strength=1;
    MSGN=0;
    input=zeros(1,Dimension);
    weight=zeros(1,Dimension);
    result_ID=0;
    result=100;
    alpha=InitialRate;
    msgID=[];
    msgSender=[];
    amount_flag=0;
    receive=[];
    path=[];
    label_r='*';
    memory=struct('send',1, 'signal_strength', signal_strength, 'input', input, 'weight', weight, 'result_ID', result_ID, 'current', 0, 'result', result, 'alpha', alpha, 'msgID', msgID, 'msgSender', msgSender, 'amount_flag', amount_flag, 'BMU', 0, 'flag', 0, 'receive', receive, 'label_r', label_r, 'control_flag', 0, 'flag1', 0, 'income', 0, 'outcome', 0);
    if ID==Control
      memory.result_flag=zeros(1,MoteNum-1);
      MSGN=0;
      InData=load('C:\lyn\prowler\wine_d.data');
      [Pattern,Dimension]=size(InData);
      Pattern_Name=zeros(1,Pattern);
      for i=1:1:Pattern
        Pattern_Name(i)=int2str(InData(i,Dimension));
      end
      InData=InData(:,1:Dimension-1);
      Dimension=Dimension-1;

      for i=1:1:Dimension
        minIn=0;
        maxIn=0;
        minIn=min(InData(:,i));
        maxIn=max(InData(:,i));

      end

  end

for j=1:1:Pattern
    InData(j,i)=(InData(j,i)-minIn)/(maxIn-minIn);
end
end
fprintf('Data Normalization Finished!\n');
memory.input=InData(:,1:Dimension);
Set_Clock(1000);

Else
    % Initialization Weight
    memory.weight=rand(1,Dimension);
end

case 'Packet_Sent'
    memory.send=0;

case 'Packet_Received'
    msg=data.data;
    msg_new=msg;
    % Generic mote
    if ID~=Control
        % If it is first time receive
        if ~isFirstTime(msg,memory.msgID,memory.msgSender)
            % Remember the message
            memory.income=memory.income+1;
            memory.msgID=[memory.msgID,msg.msg_ID];
            memory.msgSender=[memory.msgSender,msg.sender_ID];
            % release memory
            if length(memory.msgID)==800
                tmp=memory.msgID(1,300:800);
                tmp1=memory.msgSender(1,300:800);
                clear memory.msgID memory.msgSender;
                memory.msgID=tmp;
                memory.msgSender=tmp1;
                clear tmp tmp1;
            end
            msg.hop=msg.hop+1;
            if msg.F_destination_ID==0
                % Message Type 0, calculate results and send back
                if msg.msg_type==0
                    memory.current=msg.msg.current;
                    memory.path=[];
                    % remember the last transmitter from sender
                    memory.path=[memory.path;msg.last_transmitter_ID];
                    memory.update_flag=0;
                    % calculate results
        }
if memory.current==msg.msg.current
memory.input=msg.msg.dt;
result=memory.input-memory.weight;
tmpresult=0;
memory.flag=0;
memory.flag1=0;
for i=1:1:length(result)
    tmpresult=tmpresult+result(i)^2;
end
tmpresult=double(sqrt(tmpresult));
result=tmpresult;
msg_new.msg.dt=[result,0,0,0,0];
msg_new.msg_type=1;
msg_new.sender_ID=ID;
msg_new.msg.current=memory.current;
msg_new.last_transmitter_ID=ID;
msg_new.F_destination_ID=MoteNum;
msg_new.destination_ID=memory.path(1);
msg_new.hop=0;
if rand<1
    MSGN=MSGN+1;
    msg_new.msg_ID=MSGN;
    % send results to supervisory
    Send_Packet(radiostream(msg_new,memory.signal_strength));

    memory.msgID=[memory.msgID,msg_new.msg_ID];

    memory.msgSender=[memory.msgSender,msg_new.sender_ID];
    memory.outcome=memory.outcome+1;
end
if memory.current<=EndIteration &&
    mod(memory.current,Pattern*2)==0
    % Record results
    fid_step=fopen('C:\lyn\prowler\step_log.txt','a+');
    fprintf(fid_step,'%d %d
',
    memory.current,ID);
    for i=1:1:Dimension
        fprintf(fid_step,'%5.3f ',
            memory.weight(i));
    end
    fprintf(fid_step,'%d %d
',
        memory.current,ID);
memory.income, memory.outcome);
end
end

% Message type 3, check if should update weight
elseif msg.msg_type==3
if msg.hop<=Neighbor_Hop(memory.current)
    memory.alpha=Get_Rate
    (memory.current, msg.hop);
    memory.weight=memory.weight+memory.alpha*(memory.input-
    memory.weight);
    msg_new.F_destination_ID=MoteNum;
    msg_new.destination_ID=memory.path(1);
    msg_new.last_transmitter_ID=ID;
    msg_new.sender_ID=ID;
    msg_new.msg_type=7;
    msg_new.msg.current=memory.current;
    msg_new.msg.dt=zeros(1,Dimension);
    MSGN=MSGN+1;
    msg_new.msg_ID=MSGN;
    msg_new.hop=0;
    % Inform BMU Weight update finished
    Send_Packet(radiostream(msg_new, memory.signal_strength));
    memory.outcome=memory.outcome+1;
end

% Message type 5, label message from supervisory
elseif msg.msg_type==5
if ID==msg.msg.dt(1)
    % Attach label
    memory.label_r=[memory.label_r, Pattern_Name
    (memory.current-EndIteration)];
    msg_new.F_destination_ID=MoteNum;
    msg_new.destination_ID=memory.path(1);
    msg_new.sender_ID=ID;
    msg_new.msg_type=6;
    msg_new.hop=0;
    msg_new.last_transmitter_ID=ID;
    MSGN=MSGN+1;
    msg_new.msg_ID=MSGN;
    msg_new.msg.current=memory.current;
    msg_new.msg.dt=zeros(1,Dimension);
    % Inform BMU label finished
    Send_Packet(radiostream(msg_new, memory.signal_strength));
    memory.outcome=memory.outcome+1;
% Record Results to file
fid_r=fopen('C:\lyn\prowler\result_log.txt', 'a+');

for i=1:1:Dimension
    fprintf(fid_r,' %5.3f ', memory.weight(i));
end
fprintf(fid_r,'
');
fclose(fid_r);
end

% Message type 2, BMU of this iteration
elseif msg.msg_type==2
    if memory.current==msg.msg.current && ID==msg.msg.dt(1)
        memory.alpha=Get_Rate(memory.current,msg.hop);
        % Update Weight
        memory.weight=memory.weight+memory.alpha*(memory.input-memory.weight);
        msg_new.msg.dt=zeros(1,Dimension);
        msg_new.msg_type=3;
        msg_new.sender_ID=ID;
        msg_new.F_destination_ID=0;
        msg_new.last_transmitter_ID=ID;
        msg_new.destination_ID=0;
        msg_new.msg.current=memory.current;
        msg_new.hop=0;
        MSGN=MSGN+1;
        msg_new.msg_ID=MSGN;
        % Send Weight Update Request to others
        Send_Packet(radiostream(msg_new,memory.signal_strength));
        memory.msgID=[memory.msgID,msg_new.msg_ID];
        memory.msgSender=[memory.msgSender,ID];
        memory.outcome=memory.outcome+1;
        msg_new.F_destination_ID=MoteNum;
        msg_new.destination_ID=memory.path(1);
        msg_new.last_transmitter_ID=ID;
        msg_new.sender_ID=ID;
        msg_new.msg_type=7;
        msg_new.msg.current=memory.current;
        msg_new.msg.dt=zeros(1,Dimension);
        MSGN=MSGN+1;
        msg_new.msg_ID=MSGN;

    end

    msg_new.msg_ID=MSGN+1;
msg_new.hop=0;
Send_Packet(radiostream(msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
end
end
if msg.sender_ID~=0
% Retransmit Message
msg.last_transmitter_ID=ID;
if msg.F_destination_ID==0 &&
ID~=msg.msg.dt(1);
Send_Packet(radiostream(
msg,memory.signal_strength));
memory.outcome=memory.outcome+1;
else
if msg.F_destination_ID~=ID
if ID==msg.destination_ID
msg.destination_ID=memory.path(1);
Send_Packet(radiostream(
msg,memory.signal_strength));
memory.outcome=memory.outcome+1;
else
if msg.destination_ID==0 &&
msg.sender_ID==MoteNum
Send_Packet(radiostream(
msg,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
end
end
% Supervisory Mote
elseif ID==Control
% First Time Receive
% if ~isFirstTime(msg,memory.msgID,memory.msgSender) &
% msg.sender_ID~=0
memory.income=memory.income+1;
memory.msgID(length(memory.msgID)+1)=msg.msg_ID;
memory.msgSender(length(memory.msgSender)+1)=msg.sender_ID;
% Release Memory
if length(memory.msgID)==800
  tmp=memory.msgID(1,300:800);
end
elseif ID==Control
...
tmp1=memory.msgSender(1,300:800);
clear memory.msgID memory.msgSender;
memory.msgID=tmp;
memory.msgSender=tmp1;
clear tmp tmp1;
end
msg.hop=msg.hop+1;
% Retransmit Message
if ID~=msg.F_destination_ID && ID~=msg.sender_ID
msg.last_transmitter_ID=ID;
Send_Packet(radiostream(
    msg,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
% Message type 1, results from generic motes
if msg.msg_type==1
    if ~ismember(msg.sender_ID,memory.receive) &&
        memory.control_flag==0
        memory.amount_flag=memory.amount_flag+1;
        memory.receive(length(memory.receive)+1) =msg.sender_ID;
        tmp=memory.result;
        tmp_ID=memory.result_ID;
        if tmp>msg.msg.dt(1);
            memory.result=msg.msg.dt(1);
            memory.result_ID=msg.sender_ID;
        end
        % Receive Enough Results then Choose BMU
        if memory.amount_flag>=(MoteNum-2)*Percent;
            memory.control_flag=1;
            if memory.current<=EndIteration
                % Choose BMU
                Choose_BMU(t);
            else
                % Send Label
                msg_new.sender_ID=ID;
                msg_new.last_transmitter_ID=ID;
                msg_new.F_destination_ID=0;
                msg_new.destination_ID=0;
                msg_new.msg_type=5;
                msg_new.hop=0;
                MSGN=MSGN+1;
                msg_new.msg_ID=MSGN;
                msg_new.msg.d"t=zeros(1,Dimension);
                msg_new.msg.dt(1)=memory.result_ID;
                msg_new.msg.current=memory.current;
            end
        end
    end
end
Send_Packet(radiostream(
    msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
end
end
% Message Type 6, label finished, prepare next label
elseif msg.msg_type==6
    Set_Clock(t);
% Message Type 7
elseif msg.msg_type==7
    if memory.current==msg.msg.current
        if ~ismember(msg.sender_ID,memory.receive)
            memory.receive=
                [memory.receive,msg.sender_ID];
        end
        if length(memory.receive)==1 &&
            memory.current<=EndIteration
            memory.next_amount=length(memory.receive);
            Check_Next(t+20*40000);
        end
    end
end
case 'Collided_Packet_Received'
    msg=data.data;
    msg_new=msg;
% Generic Motes
    if ID~=Control && ID~=0
        % First Time Receive
        if ~isFirstTime(msg,memory.msgID,memory.msgSender) &&
            msg.sender_ID~=0
            memory.income=memory.income+1;
            memory.msgID=[memory.msgID,msg.msg_ID];
            memory.msgSender=[memory.msgSender,msg.sender_ID];
            % Release Memory
            if length(memory.msgID)==800
                tmp=memory.msgID(1,300:800);
                tmp1=memory.msgSender(1,300:800);
                clear memory.msgID memory.msgSender;
                memory.msgID=tmp;
                memory.msgSender=tmp1;
                clear tmp tmp1;
end
156
end
msg.hop=msg.hop+1;
if msg.F_destination_ID==0
  % Message Type 1, Input from Supervisory
  if msg.msg_type==0
    memory.path=[];
    memory.current=msg.msg.current;
    memory.path=
    [memory.path;msg.last_transmitter_ID];
    memory.update_flag=0;
    % Calculate Results
    if memory.current==msg.msg.current
      memory.resend_update=0;
      memory.input=msg.msg.dt;
      result=memory.input-memory.weight;
      tmpresult=0;
      memory.flag=0;
      memory.flag1=0;
      for i=1:1:length(result)
        tmpresult=tmpresult+result(i)^2;
      end
      tmpresult=double(sqrt(tmpresult));
      result=tmpresult;
      msg_new.msg.dt=[result,0,0,0,0];
      msg_new.msg_type=1;
      msg_new.sender_ID=ID;
      msg_new.msg.current=memory.current;
      msg_new.last_transmitter_ID=ID;
      msg_new.F_destination_ID=MoteNum;
      msg_new.destination_ID=memory.path(1);
      msg_new.hop=0;
      MSGN=MSGN+1;
      msg_new.msg_ID=MSGN;
      % Send Results to Supervisory
      Send_Packet(radiostream
      msg_new,memory.signal_strength));
  end
  tmpresult=double(sqrt(tmpresult));
  result=tmpresult;
  msg_new.msg.dt=[result,0,0,0,0];
  msg_new.msg_type=1;
  msg_new.sender_ID=ID;
  msg_new.msg.current=memory.current;
  msg_new.last_transmitter_ID=ID;
  msg_new.F_destination_ID=MoteNum;
  msg_new.destination_ID=memory.path(1);
  msg_new.hop=0;
  MSGN=MSGN+1;
  msg_new.msg_ID=MSGN;
end
if memory.current<=EndIteration &&
  mod(memory.current,Pattern*2)==0

end
% Write Results to File
fid_step=fopen('C:\lyn\prowler\step_log.txt','a+');
fprintf(fid_step,'%d %d',memory.current,ID);
for i=1:1:Dimension
    fprintf(fid_step,'%5.3f ',memory.weight(i));
end
fprintf(fid_step,'%d %d
',memory.income,memory.outcome);
end

% Message Type 3, Receive Weight Update from BMU
elseif msg.msg_type==3
    if msg.hop<=Neighbor_Hop(memory.current)
        % Update Weight
        memory.alpha=Get_Rate(memory.current,msg.hop);
        memory.weight=memory.weight+
                        memory.alpha*(memory.input-
                                      memory.weight);
        msg_new.F_destination_ID=MoteNum;
        msg_new.destination_ID=memory.path(1);
        msg_new.last_transmitter_ID=ID;
        msg_new.sender_ID=ID;
        msg_new.msg_type=7;
        msg_new.msg.current=memory.current;
        msg_new.msg.dt=zeros(1,Dimension);
        MSGN=MSGN+1;
        msg_new.msg_ID=MSGN;
        msg_new.hop=0;
        % Inform BMU Weight Update
        Send_Packet(radiostream(msg_new,memory.signal_strength));
        memory.outcome=memory.outcome+1;
    end
end

% Message Type 5, label from Supervisory
elseif msg.msg_type==5
    if ID==msg.msg.dt(1)
        % Attach Label
        memory.label_r=[memory.label_r,
                        Pattern_Name(memory.current-
EndIteration));
PrintMessage(memory.label_r);
msg_new.F_destination_ID=MoteNum;
msg_new.destination_ID=memory.path(1);
msg_new.sender_ID=ID;
msg_new.msg_type=6;
msg_new.hop=0;
msg_new.last_transmitter_ID=ID;
MSGN=MSGN+1;
msg_new.msg_ID=MSGN;
msg_new.msg.current=memory.current;
msg_new.msg.dt=zeros(1,Dimension);
% Inform BMU label finished
Send_Packet(radiostrem(
msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
fid_r=fopen('C:\\lyprowler\result_log.txt','a+');
fprintf(fid_r,'%d ',ID);
for i=1:1:Dimension
    fprintf(fid_r,'%5.3f ',
            memory.weight(i));
end
fprintf(fid_r,'
');
close(fid_r);
end
% Message Type 2, BMU of this iteration
elseif msg.msg_type==2
    if memory.current==msg.msg.current &&
        ID==msg.msg.dt(1)
        memory.alpha=Get_Rate(msg.current,msg.hop);
        % Update Weight
        memory.weight=memory.weight+
        memory.alpha*(memory.input-
        memory.weight);
        msg_new.msg.dt=zeros(1,Dimension);
        msg_new.msg_type=3;
        msg_new.sender_ID=ID;
        msg_new.F_destination_ID=0;
        msg_new.destination_ID=0;
        msg_new.last_transmitter_ID=ID;
        msg_new.msg.current=memory.current;
        msg_new.hop=0;
        MSGN=MSGN+1;
        % Send Weight Update Request to others
Send_Packet(radiostream(
  msg_new,memory.signal_strength));
memory.msgID=[memory.msgID,
  msg_new.msg_ID];
memory.msgSender=[memory.msgSender,ID];
msg_new.F_destination_ID=MoteNum;
msg_new.destination_ID=memory.path(1);
msg_new.last_transmitter_ID=ID;
msg_new.sender_ID=ID;
msg_new.msg_type=7;
msg_new.msg.current=memory.current;
msg_new.msg.dt=zeros(1,Dimension);
MSGN=MSGN+1;
msg_new.msg_ID=MSGN;
msg_new.hop=0;
Send_Packet(radiostream(
  msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
end
end
if  msg.sender_ID~=0
  msg.last_transmitter_ID=ID;
  % Retransmit Message
  if msg.F_destination_ID==0
    Send_Packet(radiostream(
      msg,memory.signal_strength));
    memory.outcome=memory.outcome+1;
  else
    if msg.F_destination_ID==ID
      if ID==msg.destination_ID
        msg.destination_ID=memory.path(1);
        Send_Packet(radiostream(
          msg,memory.signal_strength));
        memory.outcome=memory.outcome+1;
      else
        if msg.destination_ID==0 &&
          msg.sender_ID~=MoteNum
          Send_Packet(radiostream(
            msg,memory.signal_strength));
          memory.outcome=memory.outcome+1;
        end
      end
    end
  end
end
end
end
% Supervisory Mote
elseif ID==Control
  % First Time Receive
  if ~isFirstTime(msg,memory.msgID,memory.msgSender) &
    msg.sender_ID==0
    memory.income=memory.income+1;
    memory.msgID(length(memory.msgID)+1)=msg.msg_ID;
    memory.msgSender(length(memory.msgSender)+1)=msg.sender_ID;
  % Release Memory
  if length(memory.msgID)==800
    tmp=memory.msgID(1,300:800);
    tmp1=memory.msgSender(1,300:800);
    clear memory.msgID memory.msgSender;
    memory.msgID=tmp;
    memory.msgSenser=tmp1;
    clear tmp tmp1;
  end
  msg.hop=msg.hop+1;
  if ID~=msg.F_destination_ID && ID~=msg.sender_ID
    Send_Packet(radiostream( 
      msg,memory.signal_strength));
    memory.outcome=memory.outcome+1;
  end
  % Message Type 1, results from generic motes
  if msg.msg_type==1
    if ~ismember(msg.sender_ID,memory.receive) &
      memory.control_flag==0
      memory.amount_flag=memory.amount_flag+1;
      memory.receive(length(memory.receive)+1)=msg.sender_ID;
      if memory.result>msg.msg.dt(1);
        memory.result=msg.msg.dt(1);
        memory.result_ID=msg.sender_ID;
      end
    % Enough Results Received
    if memory.amount_flag>=
      (MoteNum-2)*Percent;
        memory.control_flag=1;
        if memory.current<=EndIteration
          % Choose BMU
          memory.next_amount=0;
          memory.receive=[];
          Choose_BMU(t);
        else
          end
        else
          else
          end
        end
      end
    end
    end
  end
end
fprintf('@@ Label ID=%d\n', memory.result_ID);
msg_new.sender_ID=ID;
msg_new.last_transmitter_ID=ID;
msg_new.F_destination_ID=0;
msg_new.destination_ID=0;
msg_new.msg_type=5;
msg_new.hop=0;
MSGN=MSGN+1;
msg_new.msg_ID=MSGN;
msg_new.msg.dt=zeros(1,Dimension);
msg_new.msg.dt(1)=memory.result_ID;
msg_new.msg.current=memory.current;
% Inform Supervisory label finished
Send_Packet(radiostream(
    msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
end
end

% Message Type 6, label finished, send next label
elseif msg.msg_type==6
    Set_Clock(t);
% Message Type 7, No Update, Send next input
elseif msg.msg_type==7
    if memory.current==msg.msg.current
        if ~ismember(msg.sender_ID,memory.receive)
            memory.receive=[memory.receive,msg.sender_ID];
        end
        if length(memory.receive)==1 &&
            memory.current<=EndIteration
            memory.next_amount=length(memory.receive);
        end
    end
end

case 'Clock_Tick'
    % Prepare Message Data
    try type=data.type;
    catch type='none';end
    % Type BMU, choose BMU and Send Weight Update To It
    if strcmp(type,'BMU')
msg_new.sender_ID=ID;
msg_new.F_destination_ID=0;
msg_new.destination_ID=0;
msg_new.last_transmitter_ID=ID;
msg_new.msg.dt=zeros(1,Dimension);
msg_new.msg.dt(1)=memory.result_ID;
msg_new.msg.current=memory.current;
MSGN=MSGN+1;
msg_new.msg_ID=MSGN;
msg_new.msg_type=2;
msg_new.hop=0;
memory.msg_ID=[memory.msgID,msg_new.msg_ID];
memory.msgSender=[memory.msgSender,ID];
Send_Packet(radiostream(msg_new,memory.signal_strength));
memory.outcome=memory.outcome+1;
memory.amount_flag=0;
memory.receive=[];
memory.next_amount=0;
% Type Next, Check Update
elseif strcmp(type,'next')
  if memory.next_amount==length(memory.receive) &&
    memory.current<=EndIteration &&
    memory.control_flag==1 && memory.flag==0
    memory.flag=1;
    Set_Clock(t);
  else
    memory.next_amount=length(memory.receive);
    Check_Next(t+20*40000);
  end
% Type Result, If No Update, Send Next Training Pattern
elseif strcmp(type,'result')
  if memory.next_amount==length(memory.receive)
    if memory.amount_flag~=floor((MoteNum-2)*Percent) &&
      memory.next_amount~=0 && memory.control_flag==0
      memory.flag1=1;
      Set_Clock(t);
    else
    memory.next_amount=length(memory.receive);
    Check_Result(t+4*40000);
  end
else
  if memory.control_flag==0
    memory.next_amount=length(memory.receive);
    Check_Result(t+4*40000);
  end
% Preparing Next Training Data And Send It
else

memory.current=memory.current+1;
if memory.current<=EndIteration+Pattern
    fprintf('%d\r\n',memory.current-EndIteration);
    memory.flag=1;
    memory.flag1=1;
    memory.next_amount=0;
    memory.amount_flag=0;
    memory.result=100;
    memory.result_ID=0;
    p=mod(memory.current,Pattern);
    memory.receive=[];
    memory.control_flag=0;
    if p==0
        p=Pattern;
    end
    dt=memory.input(p,:);
end
% Prepare Message Data
infor=struct('current',memory.current,'dt',dt);
MSGN=MSGN+1;
msg=struct(...
    'sender_ID',ID,...
    'destination_ID',0,...
    'last_transmitter_ID',ID,...
    'msg',infor,...
    'msg_ID',MSGN,...
    'msg_type',0,...
    'hop',0,...
    'F_destination_ID',0);
memory.msgID=[memory.msgID,msg.msg_ID];
memory.msgSender=[memory.msgSender,ID];
end

case 'GuiInfoRequest'
    if ~isempty(memory)
        disp(sprintf('Memory Dump of mote ID# %d:\n',ID));
        disp(memory)
    else
        disp(sprintf('No memory dump available for node %d.\n',ID));
    end

case 'Application_Stopped'
    % this event is called when simulation is stopped/suspended
    % When application stops, write results to file
    if ID==Control
fid_log=fopen('C:\Documents and Settings\lliu\Desktop\prowler\winner2_log.txt','a+');
fprintf(fid_log,'%d ',ID);
for i=1:1:Dimension
    fprintf(fid_log,' %5.3f ',memory.weight(i));
end
fprintf(fid_log,'%d %d 
',memory.income,memory.outcome);
fclose(fid_log);
else
    fprintf('Total Iteration=%d
',memory.iteration);
end

case 'Application_Finished'
    % this event is called when simulation is finished
    if ID~=Control
        fid_log=fopen('C:\Documents and Settings\lliu\Desktop\prowler\winner2_log.txt','a+');
        fprintf(fid_log,'%d ',ID);
        for i=1:1:Dimension
            fprintf(fid_log,' %5.3f ',memory.weight(i));
        end
        fprintf(fid_log,'%d %d 
',memory.income,memory.outcome);
        fclose(fid_log);
    else
        fprintf('Total Iteration=%d
',memory.iteration);
    end
otherwise
    error(['Bad event name for application: ' event])
end

% Application Ends

S; % housekeeping
S; app_data{ix}=memory;
S;

% Check if it is first time receive
function r=isFirstTime(msg,msgID,msgSender)
r=0;
if ~isempty(msg) && ~isempty(msgID) && ~isempty(msgSender)
    if ismember(msg.msg_ID,msgID)
        r=1;
    else
        r=0;
    end
end
% Send Packet Function
function b=Send_Packet(data)
global ID t
radio=prowler('GetRadioName');
b=feval(radio, 'Send_Packet', ID, data, t);

function b=Set_Clock(alarm_time)
global ID
prowler('InsertEvents2Q', make_event(alarm_time, 'Clock_Tick', ID));

function Choose_BMU(alarm_time)
global ID
clock.type='BMU';
prowler('InsertEvents2Q',make_event(alarm_time,'Clock_Tick',ID,clock));

function Check_Next(alarm_time)
global ID
clock.type='next';
prowler('InsertEvents2Q',make_event(alarm_time,'Clock_Tick',ID,clock));

function Check_Result(alarm_time)
global ID
clock.type='result';
prowler('InsertEvents2Q',make_event(alarm_time,'Clock_Tick',ID,clock));
function x=animation_data
% Animation definition for application som

small=5; medium=20; large=50;

% Event_name    Animated   Color/{on/off/toggle}    Size
anim_def={...
    {'Init_Application',         0,        [0 0 0],     small}, ...
    {'Packet_Sent',              1,        [0 1 0],     small}, ...
    {'Packet_Received',          1,        [0 1 0],     small}, ...
    {'Collided_Packet_Received', 0,        [1 0 0],     small}, ...
    {'Clock_Tick',               0,        [0 0 0],     small}, ...
    {'Channel_Request',          0,        [0 0 0],     small}, ...
    {'Channel_Idle_Check',       1,        [1 0 0],     small}, ...
    {'Packet_Receive_Start',     0,        [0 1 0],     small}, ...
    {'Packet_Receive_End',       0,        [0 0 0],     small}, ...
    {'Packet_Transmit_Start',    1,        [1 0 0],     medium},...
    {'Packet_Transmit_End',      0,        [0 1 0],     small}};

for i=1:length(anim_def)
    a=anim_def{i};
    x(i)=struct('event', a{1}, 'animated', a{2}, 'color', a{3}, 'size', a{4});
end
function [topology,mote_IDs]=topology(varargin);
% Topology information for application som

% Uniformly Distribute the motes
for i=1:sq
    for j=1:sq
        x=rand*(MAXX/sq)+(j-1)*(MAXX/sq);
        y=rand*(MAXX/sq)+(i-1)*(MAXX/sq);
        tmp=[tmp;x,y];
    end
end

% Set the minimum received signal strength
P_rec=1/(1+4*(MAXX/sq)^2+4*(MAXX/sq)^2);

% Assign the IDs to all the motes
mote_IDs=1:MoteNum;