Information Processing using Liquid State Machines based on Spiking Neurons

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul Inginerie Electronica si Telecomunicatii
de către

ing. Radu Mirsu

Conducător ştiinţific:	prof.univ.dr.ing. Virgil Tiponut
Referenţi ştiinţifici:	prof.univ.dr.ing. Dorian Cojocaru
	prof.univ.dr.ing. Gavril Toderean
	prof.univ.dr.ing. Ivan Bogdanov

Ziua susţinerii tezei: 26.11.2011

BUPT

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica" din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

Copyright © Editura Politehnica – Timişoara, 2011

Această publicaţie este supusă prevederilor legii dreptului de autor. Multiplicarea acestei publicaţii, în mod integral sau în parte, traducerea, tipărirea, reutilizarea ilustraţiilor, expunerea, radiodifuzarea, reproducerea pe microfilme sau în orice altă formă este permisă numai cu respectarea prevederilor Legii române a dreptului de autor în vigoare şi permisiunea pentru utilizare obţinută în scris din partea Universităţii „Politehnica" din Timişoara. Toate încălcările acestor drepturi vor fi penalizate potrivit Legii române a drepturilor de autor.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

BUPT

Acknowledgements

This doctoral thesis was supported in part by POSDRU/6/1.5/S/13 strategic grant, ID6998, financed from European Social Fund "Investing in people" in the Human Resources Development Operational Programme 2007-2013.

This work was partially supported by the following grants:

"Noi metode de analiză şi recunoaştere a expresiei faciale", Program PNII, IDEI, Proiecte de cercetare exploratorie, cod 945/2008, finantat de Unitatea Executivă pentru Finanţarea Învăţământului Superior şi a Cercetării Stiinţifice Universitare (UEFISCSU), Nr. Contract: 599/19.01.2009.

"Research on Emotional Facial Expression recognition in Complicated Environment", Program PNII, CAPACITATI, Modul III, proiecte de cercetare bilaterale, România-China, 39-5/2008, finantat de Autoritatea Naţională pentru Cercetare Ştiinţifică (ANCS), Nr. Contract: 222/15.04.2009

I would like to thank my advisor, Prof. Dr. Eng. Virgil Tiponut, for the countless advices and work hours that he has put into this research. He has been a great colleague and friend showing optimism and high morale when needed most. I also would like to thank Prof. Dr. Eng. Catalin Caleanu with whom I have collaborated at GPU programming and Gabor filtering tasks.

I also owe my gratitude to the Applied Electronics Department Director, Prof. Dr. Eng Ivan Bogdanov, to Electronics and Telecommunications Faculty Dean, Prof. Dr. Eng. Marius Otesteanu, and to my fellow colleagues, PhD Candidates: Sebastian, George, Zoltan, Robert, Daniel and Mihai.

Special thanks to the PhD committee, Prof. Dr. Eng Dorian Cojocaru University of Craiova, Prof. Dr. Eng. Gavril Toderean, Technical University of Cluj-Napoca and Prof. Dr. Eng Ivan Bogdanov, Politehnica University of Timisoara for their evaluations and suggestions regarding the thesis.

I would like to thank my loving wife Betina, my parents and my sister Adina for their never ending understanding, patience and support.

Timişoara, November 2011 Radu Mirsu

BUPT

This doctoral thesis was supported in part by POSDRU/6/1.5/S/13 strategic grant, ID6998, financed from European Social Fund "Investing in people" in the Human Resources Development Operational Programme 2007-2013.

Mirsu, Radu

Information Processing using Liquid State Machine based on Spiking Neurons

Teze de doctorat ale UPT, Seria 7, Nr. 41, Editura Politehnica, 2011, 120 pagini, 64 figuri, 7 tabele.

ISSN: 1842-7014

ISBN: 978-606-554-376-8

Cuvinte cheie: spiking neural networks, Liquid State Machines, liquid computing, Gabor filtering, dynamic neurons, neural modelling, neurons on GPU, parallel processing

Rezumat,
Spiking neural networks are introduced as the third generation of neural models. They are dynamic models that potentially have much more processing power than classic neural networks. This thesis presents a novel approach to perform Gabor filtering using Liquid State Machines based on Spiking Neurons. The Liquid State Machine is a powerful architecture that is capable of performing universal computations without being trained on specific data. It is the job of special readout units to interpret the computation results and map them on specific target functions. In addition, the thesis presents tools that allow fast simulating of large neural networks by running the simulation in parallel on a GPU.

BUPT

CONTENTS

Acknoledgements.. 3
Abstract.. 4
Contents.. 5
List of Figures... 8

1. Introduction 11
1.1. Research Motivation 11
1.2. State of the art 13
1.3. Thesis Outline 16

2. Spiking Neural Networks 19
2.1 Synapses 21
2.2 Dynamic Synapses 21
2.2.1. Depressing Synapse 22
2.2.2. Synapse Delay 23
2.2.3. Facilitating Synapse 24
2.2.4. Recursive Model for Dynamic Synapse 25
2.3. Spiking Neuron Models 26
2.3.1. The Biological Neuron 26
2.3.1.1. Spike Generation 28
2.3.1.2. Refractory Period 28
2.3.2. Integrate and Fire Neuron 29
2.3.3. Integrate and Fire with Burst Neuron 31
2.3.4. Integrate and Fire with Adaptation Neuron 33
2.3.5. Resonate and Fire Neuron 34
2.4. Coding with Spikes 35
2.4.1. Spiking Neuron as Context Detector 36
2.4.2. Content Addressable Memory with Spiking Neural Networks 38
2.4.2.1. Choosing Spike Contexts 38

3. Modelling and Simulation 41
3.1. Model Objects 41
3.1.1. Network Object 41
3.1.2. Layer Object 42
3.1.3. Neuron Object 43
3.2. Model Functions 44
3.2.1. Simulation Functions 44
3.2.2. Vizualisation Functions 44
3.2.2.1. Vizualising Neural Time Traces 45
3.2.2.2. Vizualising Neural Spike Rates 45
3.2.2.3. Vizualising Neural Synchrony 45
3.3. Parallelizing the Model 47

BUPT

3.3.1. Choosing the Number of Slaves 48
3.3.2. Results 49

4. GPU Accelerated Model for Spiking Neural Networks 53
4.1. General Purpose GPU Computing 53
4.1.1. Early GPU Computing 54
4.1.2. NVIDIAs CUDA Architecture 55
4.1.3. Simple CUDA Example 55
4.2. Spiking Neural Network CUDA Model 57
4.2.1. Model Architecture 57
4.2.2 Delay Line Implementation 58
4.2.3. Moving Objects between Host Computer and Device GPU 61
4.2.4. Simulating the Model 63
4.2.5. Simulation Results 65
4.3. Improved CUDA Model 66
4.3.1. Minimizing the number of branches 66
4.3.2. Merging Kernels 67
4.3.3. Using Shared Memory 70
4.3.4. Overlapping Computation and Data Transfer 71
4.3.5. Using Constant Memory and Texture Memory 73
4.3.6. Simulation Results 74

5. Liquid State Machine And Liquid Computing 75
5.1. Introduction 75
5.2. Liquid State Machine Architecture 75
5.2.1. Recurrent Liquid Medium 76
5.2.2. The Read-Out Units 77
5.3. Liquid States. Separation Property 78
5.4. The Parallel Perceptron Readout Unit.
The p-Delta Learning Algorithm 79
5.4.1. The Parallel Perceptron 79
5.4.2. The Single Perceptron Delta Rule 80
5.4.3. The Parallel Perceptron p-Delta Rule 80
5.4.4. Adaptive Learning Rate 81
5.4.5. Greedy vs. Not Greedy 82
5.4.6. Adaptive Noise Margin Control 85
5.5. The Feedforward Readout Unit. Backpropagation 88

**6. Gabor Filtering using Liquid State Machines
and Spiking Neurons** 91
6.1. Introduction 91
6.2. Gabor Filtering 91
6.3. Filtering with Liquid State Machines 91
6.3.1. Input Signals 91
6.3.2. Spike Generator with Shifting Phase 97
6.3.3. Estimating Gabor Coefficients with Liquid State Machine 100
6.3.3.1. Approximation Accuracy 101
6.3.3.2 Approximation Speed 102
6.3.3.3. LSM Performance 103

7. Conclusions and Contributions 105
7.1. Conclusions 105
7.1.1. Spiking Neural Networks 105
7.1.2. MATALB simulation framework 105
7.1.3. Parallel MATLAB framework and GPU accelerated framework........ 105
7.1.4. Liquid State Machines and p-Delta Learning Rule........ 106
7.1.5 Extracting Gabor Coefficients from images using Liquid State Machines based on Spiking Neurons 106
7.2. Future Work 107
7.3. Theoretical Contributions........ 108
7.4. Practical Contributions 109
7.5. Publications List 110

Refferences 112

LIST OF FIGURES

Fig. 1.1. Parallel Implementations for simulating Spiking Neural Networks
Fig. 2.1. Second Generation Neural Networks
Fig. 2.2. Biological Neural Network
Fig. 2.3. Biological Synapse
Fig. 2.4. Synapse Dynamic functionality
Fig. 2.5. Hodgkin-Huxley Neuron Model
Fig. 2.6. Activation and inactivation variables
Fig. 2.7. Functioning of the Neuron. Generating spikes
Fig. 2.8. Integrate and fire neuron model
Fig. 2.9. Calcium Current Effect
Fig. 2.10. Adaptation Mechanism
Fig. 2.11. R&F Neuron Membrane Potential
Fig. 2.12. Spatial-temporal spike sequences. Complete and incomplete spike contexts
Fig. 2.13. Context size influence on performance.
Fig. 3.1. Model architecture
Fig. 3.2. Network and Layer Objects
Fig. 3.3. Updating the network state
Fig. 3.4. Neuron Object
Fig. 3.5. Network spiking activity object
Fig. 3.6. Visualization of spiking and potential traces
Fig. 3.7. Visualization of the average spike rate as image
Fig. 3.8. Visualizing Neural Synchrony
Fig. 3.9. Distributed Model on a network of Computers
Fig. 3.10. Master-Slave Interaction
Fig. 3.11. Non-Linear simulation time as a function of the number of neurons
Fig. 4.1. CPU vs. GPU evolution
Fig. 4.2. OpenGL graphics processing pipeline.
Fig. 4.3. Simple CUDA parallel program
Fig. 4.4. Kernel Launch
Fig. 4.5. MATLAB-CUDA C interfacing
Fig. 4.6. Spiking Neural Network Architecture
Fig. 4.7. Delay Line Functionality
Fig. 4.8. Incomplete Object transfer
Fig. 4.9. BasicObject Class
Fig. 4.10. Spiking Neural Network Simulation Flow
Fig. 4.11. GPU architecture
Fig. 4.12. Merging Kernels
Fig. 4.13. Two versions for implementing the delay line inside shared memory
Fig. 4.14. Overlapping computation and data transfer
Fig. 5.1. Liquid State Machine architecture
Fig. 5.2. Liquid State Machine implemented by a recurrent Spiking Neural Network
Fig. 5.3 Parallel Perceptron Readout Unit
Fig. 5.4. Multi-Layer Feedforward Readout Unit

Fig. 5.5. PDelta Convergence histogram
Fig. 5.6. Distribution of weight activity ("greedy" approach)
Fig. 5.7. Distribution of weight activity ("not greedy" approach)
Fig. 5.8. Convergence Rate
Fig. 5.9. K Control Rule
Fig. 5.10. Average margin during training
Fig. 5.11. Margin learning rate mlr during training
Fig. 6.1. 2D Gabor Filter Kernel
Fig. 6.2. Time-Multiplexing the Input Signals
Fig. 6.3. Pixel Multiplexing Circuit
Fig. 6.4. Membrane Potential of R&F Neuron
Fig. 6.5. Activation of R&F Neurons
Fig. 6.6. Optimization of m1 and m2 synaptic connections
Fig. 6.7. Linearized Rate-Code
Fig. 6.8. Shifting Phase Spike Trains
Fig. 6.9. Shifting Phase Circuit 1
Fig. 6.10. Selective Firing Window
Fig. 6.11. Shifting Phase Circuit 2
Fig. 6.12. Spiking Activity of Liquid Medium
Fig. 6.13. Estimating Gabor Coefficients
Fig. 6.14. Computing the Approximation Error

1. INTRODUCTION

1.1. Research Motivation

Networks based on spiking neurons are thought to be the third generation of neural networks. This classification is done by Maass in [1] as follows:

- Generation 1. Binary networks built from perceptrons that are able to perform simple classifications and compute digital functions
- Generation 2. Networks with real-numbered outputs that could be used as universal approximators to any degree of precision.
- Generation 3. Spiking neural networks (SNN).

In contrast with previous generations, spiking neural networks return to models that resemble the biological neuron, and capture its dynamic spiking functionality. Even though the mathematical model for a biological neuron is an old discovery [2], it was left aside for decades because its complexity led to computationally overwhelming problems. Recently, with the continuously growing processing power of computers, researchers are returning to spiking models in order to find solutions to problems that were not solved by previous generations of neural networks. Some of the disadvantages of the classical models are presented in the following paragraphs. These combined with the numerous advantages of the new spiking models represented a strong motivation for embracing this research track.

In case of older models it was assumed that the intricate details of the neuron behavior are irrelevant to information processing. If biological neurons used simple techniques to encode information, like rate coding, modeling by using an activation function would be sufficient. However, recent research shows that it is very likely that the dynamic behavior of the spiking neurons (tonic or phasic spiking, bursting, spike frequency adaptation, spike latency, sub-threshold oscillations, resonance, integration and coincidence, rebound spikes or bursts, bistability or threshold variations) plays an important part in information processing [81], [83]. These behaviors are well presented in [3]. Also, none of these behaviors were achievable by the older generations of neural networks.

Several mechanisms are present at synapse level. Observations made in [4] show that a synapse can adjust its strength depending on the timing between the presynaptic spike and the postsynaptic spike. This way, two neurons that are both firing within a short time interval can lock and fire in synchrony, thus creating a natural implementation of Hebbian learning. This property requires that the neuron model is spiking.

Research in the field of neuropsychology is continuously unveiling information regarding the technique used by the brain to code and process information. These discoveries, most often, are incompatible with the older neural models and cannot be applied directly because there is not a direct relationship between the biological observations and the elements of the model. In the case of spiking neural networks this discrepancy is eliminated.

Spiking neural networks have another property that the traditional models do not. Since the spiking models are dynamic time-based models they can oscillate and can exhibit resonant behavior. In [5], [6] a spiking neural network achieves signal multiplexing by using resonate and fire neurons that are able to lock and synchronize with specific spike frequencies. In [7], Izhihevich introduces another interesting property of spiking neurons called "polychrony" different from "synchrony". If synchrony refers to a group if neurons firing at the same time (or within a small time window), polychrony refers to a group of neurons that fire in the same order and with the same relative timing. This property is also referred to as temporal grouping in papers like [8], [9], [10], [11], [12], and seems to play a crucial part in how the central nervous system performs computations. Polychrony is possible due to the synapse propagation delays which cause the spiking network to generate spatial-temporal spike sequences when stimulated. The complex dynamics of an intricately connected network could lead to a potentially unlimited set of spike sequences. Therefore, a spiking neural network could implement a memory that has a storing capacity significantly bigger than its number of synapses.

Oscillation and synchrony are properties that are considered to play an important part in image segmentation and object binding [13], [14]. Csibra and Davis [13] observe a close relationship between object binding and 40Hz (gamma band) oscillations evident in the brain of an 8 months infant. This is the same age when the behavior of the child start to exhibit perception of spatially separated visual features. Oscillations can also play an important role in short term memory according to Jensen in [15] and [16]. Here it is demonstrated that oscillations in the alpha band (9 to 12Hz) have an increasing peak that related to the number of items retained by the short term memory.

Spiking neural networks are obviously an improvement compared to the more traditional models. Backing up this statement is the fact that spiking networks are able to perform all tasks and exhibit all behaviors present at the first generation and the second generation of networks. In addition, the above paragraph describes new behaviors that are new and are very promising from the information processing perspective. Also, these models are much more similar to the biological neurons. This fact is encouraging because nature has repeatedly proven to be the ultimate designer. Because all good things come with a price, spiking neural models have their own disadvantages:

- The complexities of the individual neuron and also the large size of the networks that need to be simulated require huge amounts of computational resources. This is even more critical when several simulations need to be performed during an optimization. The limited speed of the simulator was also the first big challenge of this thesis. A satisfactory solution to this problem was to use a GPU to accelerate the simulation in hardware. Of course, regardless of the simulation platform, there will always be a compromise between the sizes of the network, the complexity of the individual neuron, the duration of the simulation and the step size of the solver.
- Spiking neural networks are systems with extremely complex dynamic behavior. The major difficulty in understanding how this behavior can process information. Investigating the functionality of the network by using rigorous mathematical tools that could directly pinpoint the solution to the problem is improbable. More likely to be successful are evolutionary techniques that allow the system to evolve on its own.

BUPT

- Because these models are relatively new there is a shortage of knowledge, algorithms or tools that serve the utilization of such networks.
- The spiking neural networks are very successful in the real world. However, it should be always kept in mind that the brain has around 30 billion neurons and each neuron is served by several thousands of synapses. These figures that are beyond any present simulation capabilities. It is unclear if similar performances can be obtained using a reduced size spiking neural network even for significantly simpler applications.

1.2. State of the Art

There is a continuous struggle to find new architectures and solutions to designing a biologically inspired neural network that is computationally powerful. It is the research area where several adjacent study fields collide (i.e. computer science, neuroscience, biophysics, and machine learning). In [17] Jaeger and Maass present that there are currently two approaches when trying to design a biologically inspired neural network. The first method, the bottom-up way, investigates the structure and the dynamic functionality of the brain by observation. Then it uses mathematical tools to try and capture dynamical patterns and special behaviors that are believed to be significant to information processing. The method is called bottom-up because it deduces the method and flow of the computation by examining the functioning of the hardware. Alternately, we can use the top-down method. In this case we start from known computing techniques and design sub-modules of biologically inspired networks, each suitable for solving a part of the computational task. In contrast to the bottom-up method, the top-down approach fits the hardware to the method. Obviously, the top-down method has a higher chance of deviating from the actual biological architecture.

This thesis focuses on a new type of architecture called Liquid State Machine (LSM) introduced by Maass, Natschlaeger and Markram in [18]. It is a new design that adopts the principle of reservoir computing. In parallel, but independently, a similar architecture called Echo State Networks (ESN) was developed by Jaeger [19], [86]. The two models are similar in principle but were designed with different applications in mind. Liquid State Machines tend to be general and are formulated within the mathematical frameworks of dynamical systems theory and filtering theory. Because the model aims towards a biologically plausible implementation it is restricted to having spiking neurons as computational units that preserve the characteristics of the biological neuron and operate well under noisy conditions. On the other hand, the echo state networks are a particular case of LSM and so the theory applies. However, ESNs are designed to have a higher performance on particular engineering applications that use noiseless artificial neural networks as computing units.

There are several directions of research regarding LSMs and ESNs. The following paragraphs try to determine the most important research tracks and present some state of the art methods and results.

For example, Yamazaki and Tanaka [20] try to find a direct correspondence between the cerebellum and a liquid state machine. In fact they find evidence that the granular layer behaves as a liquid medium (reservoir), while the Purkinje cells which receive signals from the granular layer act as readout units. In [21], different forms of network plasticity are analyzed in an attempt to determine how they affect the dynamic behavior of recurrent spiking neural networks. The paper studies spike timing dependent plasticity (STDP) which is responsible for adapting synaptic

BUPT

strength, and intrinsic plasticity (IP) which is responsible for adapting the excitability of individual neurons. The interaction between the two forms of plasticity maintains the homeostasis of neural activity and stabilizes the LSM.

Joshi [22] demonstrates that a liquid state machine can also be used as a multi-tasking computing machine, where the computations related to all tasks are being performed in parallel by the same liquid medium. It is the duty of several readout units to map the liquid neural activity to the task specific output functions. It is very important to notice that unlike other parallel machines that dedicate separate resources to different tasks, the liquid state machine performs a unified computation of multiple tasks where the same neurons can participate at different tasks at the same time. In [22] it is also shown that the readout units can send feedback of their activity in order to tune up the activity of the liquid medium. The paper also presents an application with good results where a liquid state machine is used at driving electrical motors. In [23], Legenstein and Maass try to determine which properties of the recurrent circuits of spiking neurons are relevant for their computational performance. They also find to methods of analyzing the computational capabilities of spiking neural networks.

Echo state networks are also well used in speech recognition applications as demonstrated by Skowronski and Harris [24]. Here a speech classifier that can recognize words from a small vocabulary is designed by combining an ESN and a state machine. The new classifier outperforms the Hidden Markow Models in regimes of low signal to noise ratios. Article [25] also presents an application with speech recognition that uses spiking neurons arranged in an architecture called "mus-silicium". The system contained approximately one thousand spiking neurons and was able to recognize ten spoken words regardless of the speaking speed. The mus-silicium however, is not biologically inspired. Instead, it uses spiking neurons in an architecture that is built on more deterministic rules inspired from signal processing and computer science.

In [26], Verstraeten and Schrauwen use a liquid state machine that is trained to perform isolated word recognition. Several techniques for encoding the input signals are tested. The encoding represents a bridge between the actual recorded signals and the liquid state machine. Surprising and also encouraging is the fact that the liquid state machine performs best when the model of the inner ear is used at encoding the vocal signals. This result increases the confidence that the liquid state machine is indeed biologically plausible. Another speech/audio application if presented in [33].

Another field where liquid state machines and echo state networks are applied is grammar and language learning. Tong presents in [27] an application that uses an echo state network to learn grammatical structure. The results are comparable to those obtained by the existing Elman networks. The advantage over the Elman networks is simpler design rules and simpler training algorithms. The Elman networks, which are also recurrent, were able learn internal data representations that were sensitive to linguistic processes by adjusting the synapses of the recurrent connections during a training algorithm. ESNs, like liquid state machines, have a fixed recurrent medium that does not need to be trained. Training occurs only at readout level and is much simpler.

Currently liquid state machines and echo state networks have proven immense potential in several research fields targeting a wide range of applications. As expected some of these applications were more successful at using liquid state machines than others. Consequently, another research track arose that tries to determine what parameters of the liquid medium (reservoir) are most significant for

BUPT

performance and if these parameters are specific to particular applications. Haykin and Xue [28] present a modified version of echo state network that uses lateral inhibition to improve the richness of the liquid medium dynamics. In [29], the authors present a method for quantifying the richness of the liquid medium dynamics by evaluating the entropy of the echo states. In [30], Dedual and Ozturk present a modified version of the readout units called MACE that is able achieve higher specificity in pattern recognition applications.

The last part of this paragraph is dedicated to implementation and simulation techniques. Currently, there's no single simulation framework that can be considered standard for simulating spiking neural networks. Because the model is computationally demanding one of the biggest issues when choosing or designing a simulation framework is to accelerate it as much as possible. As it will be seen in the following chapters the model is very parallel and so the major trend in SNN simulation is to use a framework that runs on multi-core or GPU parallel architectures. Chapters 3 and 4 present two approaches to parallelize the model, one using a distributed architecture and another using a GPU. In [31] Bhuiyan, Pallipuram and Smith, compare the speed-ups obtained by four parallel architectures: IBM PS3, AMD Opteron, Intel Xeon and NVIDIA GPU [41]. The results of their study are presented in figure 1.1. The speedups obtained with the GPU architecture are very similar to those presented in chapter 4. The numbers, however, cannot be compared directly because the speedup was measured by benchmarking different processors.

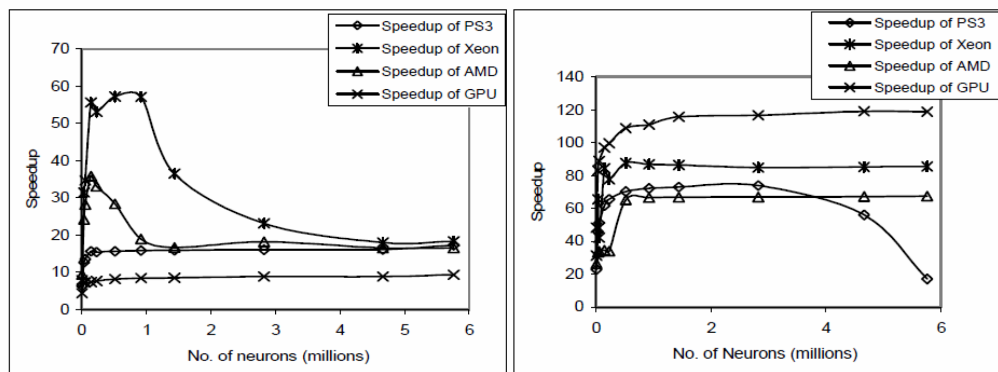


Fig. 1.1. Parallel Implementations for simulating Spiking Neural Networks

The left graph in figure 1.1 corresponds to simulations where each individual neuron uses the Izhikevich model, while the right graph contains results of simulations that use the Huxley-Hodgkin (HH) neuron model. It is very interesting to see that the speedup differs significantly between the two graphs and also the ranking of the four architectures is not the same (i.e. GPU is worst for the Izhikevich model and best for the HH model). The reason is that each architecture has a specific flop/byte (computation/transfer) ratio that makes it most efficient. A CPU core dedicates a lot of chip area to data caching and less to computation. On the other hand a GPU has a huge number of execution units and almost no data caching. The PS3, Opteron and Xeon are somewhere in between. In conclusion, the GPU outperforms the other architectures when the flop/byte ratio is high. This is the case of the HH neuron model that is very computationally demanding (flop/byte = 6.02 [31]). In the case of the Izhikevich model the flop/byte ratio is 0.65 [31] and

BUPT

so the GPU is outperformed by the other architectures due to lack of caches. Also worth noticing is the fact that when the size of the neural network is very high the speedups obtained by the Xeon and Opteron also drop down and almost equal the speedup of the GPU. This is because the size of the cache becomes insufficient for the size of the network and so the cache's miss rate increases dramatically.

In conclusion a parallel implementation of a spiking neural network is appealing and can achieve substantial improvement of the simulation speed. The simulation performed in this thesis use models that are similar to the Izhikevich model rather than the Hodgkin-Huxley model. Therefore, the GPU implementation is not necessarily the fastest. Nevertheless, it was chosen due to its accessibility and reduced cost. The speedup obtained by the GPU implementation was sufficient to serve all tasks involving this thesis.

1.3. Thesis Outline

The thesis is organized on seven chapters as follows:

- Chapter 2 introduces some basic knowledge about the structure of the biological neuron equipped with dynamic synapses. Several dynamic behaviors are studied and four mathematical models are chosen to be most significant: the integrate and fire neuron, the integrate and fire with adaptation neuron, the integrate and fire with burst neuron, and the resonate and fire neuron. The chapter concludes by presenting some spike coding techniques and the ability of a spiking neuron to act as a context detector and implement a content addressable memory.
- Chapter 3 presents a MATLAB simulation framework that was designed to easily implement and simulate a spiking neural network of desired architecture. The tool also contains functions for result analysis and visualization. The second part of the chapter presents a modification of the framework that allows parallel simulation on several computers connected by a network. The approach proves that the neural network model has a lot of parallelism to be exploited and also offers some improvement in terms of simulation speed. However, the improvement is not satisfactory because of the high communication time between computing units. Consequently, alternative parallel implementations are further searched.
- Chapter 4 presents a parallel implementation that uses a graphics processor to hardware accelerate the simulation. The GPU is designed and optimized to be very efficient on parallel processing and so the implementation of the neural network model is inherently promising. The simulation framework is redesigned such that it suits better the hardware design of the GPU. Throughout the chapter several difficulties are presented and also their solutions. In the second part of the chapter three additional improvements are presented that further accelerate the simulation. The chapter concludes by presenting a benchmark that compares the GPU model to the MATLAB model and also to a C++ model. A communication interface between CUDA C and MATLAB is also provided by means of MATLAB MEX files.
- Chapter 5 presents the Liquid State Machine computing architecture as a potential solution to information processing using spiking neurons. The chapter presents the design rules of the network and also its potential advantages and disadvantages. In the second part of the chapter two alternatives for implementing the readout units are presented (parallel perceptron and the multi-layer feedforward network), accompanied by appropriate training

algorithms (p-Delta learning rule and backpropagation). Additionally, three improvements are made to the p-Delta rule that increase the convergence rate and convergence speed.

- Chapter 6 presents an application that uses a Liquid State Machine based on spiking neurons to extract Gabor coefficients from fiducial points of an image. The pre-processing of the input signals is also done by means of spiking neural networks, yielding three variants for the input signal.
- Chapter 7 presents the theoretical and practical contributions brought by the thesis, the conclusions and future work.

BUPT

2. SPIKING NEURAL NETWORKS

In [1], spiking neural networks (SNN) are presented to be the third generation of neural models. The first generation of neural networks is based on McCulloch-Pitts neurons, also called perceptrons. Interconnecting the neurons in various ways yielded a wide variety of networks like: multi-layer perceptrons, Hopfield Networks, or Bolzmann machines. All networks using perceptrons had a digital output and were useful at classification or modeling digital functions.

The second generation of neural networks uses activation functions in order to obtain a continuously varying output. Some of the most significant networks of this type are: feed-forward sigmoid networks, recurrent networks, and radial basis function networks. This type of networks can model both digital functions and functions with analog input/output. The biological neurons generate spikes rather than analog outputs. However, the second generation of neural networks is biologically plausible assuming that the biological neurons code information using the frequency of the spike train rather than the inter-spike timing relationships. This is present in the higher cortical areas of the brain where neurons adjust their firing rate slowly and so are able to carry frequency information. Another advantage of the continuous activation functions is the possibility to use gradient based training algorithms. Figure 2.1 presents three types of the second generation neural networks.

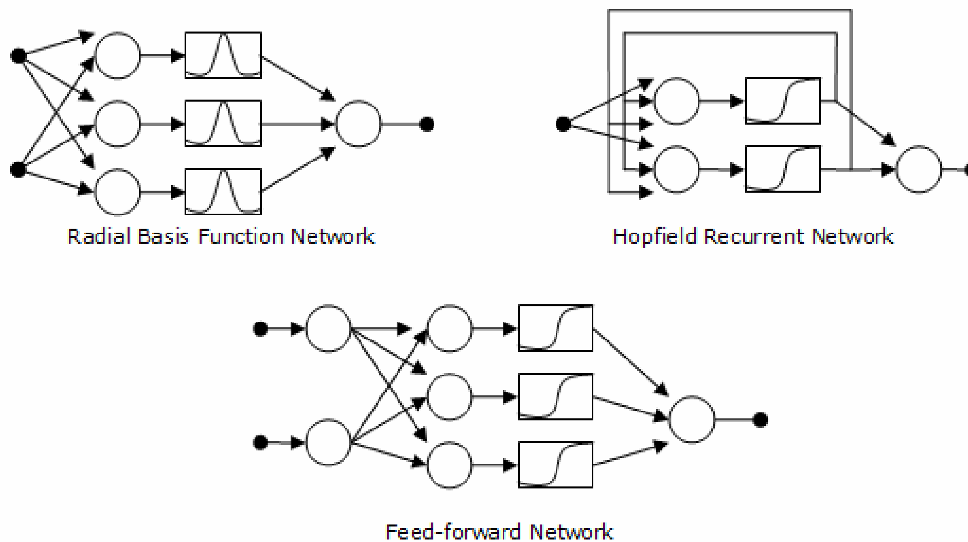


Fig. 2.1. Second Generation Neural Networks

However, some areas of the cortex perform tasks that require fast computations. In such cases frequency coding is questionable and it is more likely that inter-spike timing is used to encode information [45], [48], [51], [52]. The

third generation of neural networks tries to accommodate this new requirement. The new model has a "spiking" output that is very similar to the biological neurons. In addition, spiking networks still offer all the features present at the second generation of networks. Figure 2.2 presents the structure of a biological neural network.

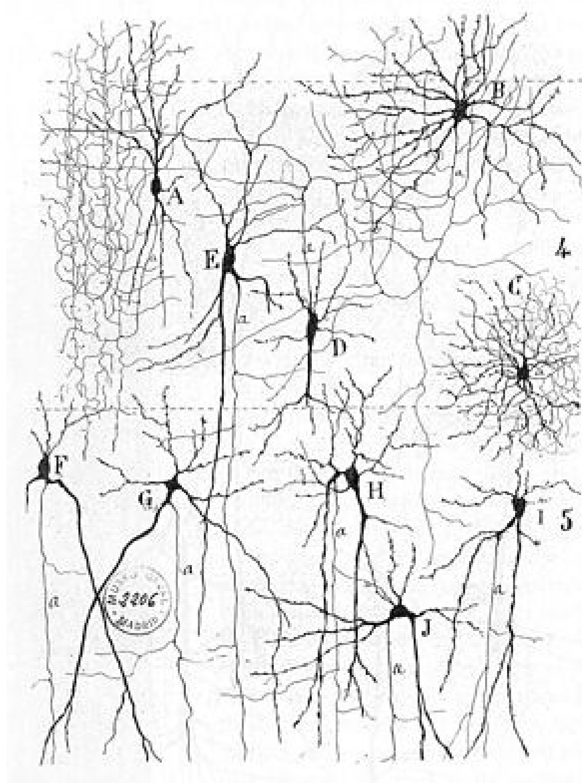


Fig. 2.2 Biological Neural Network

The most important parts of a biological neuron are: the neuron cell (the computational unit), the dendrites (the inputs of the neuron), and the axon (the output of the neuron). The bridge between an axon and a dendrite is called synapse. Spikes (also called action potentials) propagate along the axon of the source neuron, cross the synapse and reach the target neuron via its dendrite. When a spike crosses a synapse, the shape, the timing and also the amplitude of the spike is influenced by dynamic properties of the synapse. In reality spikes do not actually cross a synapse. They simply trigger a pre-synaptic mechanism that generates a new post-synaptic spike. Therefore, synapses are active elements. The rest of this chapter presents some physical and chemical mechanisms that are present in biological synapses and neuron cells. It does a mathematical description of the neuron functionality and introduces four simplified models for spiking neurons and dynamic synapses that are suitable for simulation. The models manage to remain simple enough and still capture the main characteristics of the biological neuron. Model simplicity is the key of this project in order to be able to simulate large networks

BUPT

2.1. Synapses

A synapse is an active functional unit that connects two neurons. Most often it lies at the junction of an axon and a dendrite, but other types of connections are also possible: axon-cell body, axon-axon or dendrite-dendrite. In older neural models synapses are static signal pathways that can only be used at weighing signals, hence the name "static synapses". This simplification comes from the assumption that neural processes are slow and that only firing rates are used at coding information rather than exact spike timing or network dynamics. In reality, the biological synapse, besides being a signal transducer, is a very powerful non-linear signal pre-processor that has a complex dynamic behavior [50], [53].

2.2. Dynamic Synapses

This sub-paragraph briefly presents the functioning of the biological synapse and introduces a simplified biologically inspired model called dynamic synapse. Figure 2.3 presents the structure of a biological synapse. In the left and right parts of the figure the pre-synaptic axon and the post-synaptic dendrite are depicted. Spikes arriving along the axon of the pre-synaptic neuron are also called action potentials. When an action potential reaches the terminal of the axon it raises the voltage locally causing calcium ion channels to open. This produces an influx of calcium ions that causes the vesicles of neurotransmitters to break and the release the neurotransmitters into the synaptic cleft. On the other side of the synapse, neurotransmitters bind with receptors triggering several ligand-gated ion channels.

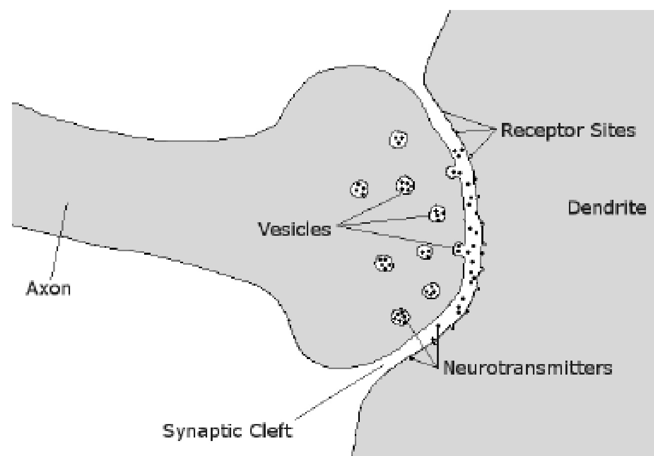


Fig 2.3. Biological Synapse

As a result, the membrane potential of the post-synaptic neuron is raised. Eventually, the neurotransmitters brake loose from the receptors causing the channels to close and stopping the increase of the membrane potential. Some of the free neurotransmitters are removed from the synaptic cleft by the reuptake pumps

BUPT

which send them back to the axon terminal. Here the neurotransmitters are repacked into vesicles and are ready to be used again.

The strength of the synapse is considered to be the impact of one pre-synaptic spike on the membrane potential of the post-synaptic neuron. This is analog to the weight of the static synapse. In contrast with the static synapse, the biological synapse does not have a constant strength, as it varies over time as an effect of several synapse internal mechanisms.

2.2.1. Depressing Synapse

Because neurotransmitters are the triggering factors for post-synaptic potentials they are considered to be the primary resources for this event. The amplitude of the post-synaptic potential depends on several factors like: quantity of released neurotransmitters, number of receptors, and the ability of each activated receptor to produce post-synaptic current. After a pre-synaptic action potential arrives a fraction of the available neurotransmitters are released. If another action potential arrives shortly after the first one, the quantity of available neurotransmitters (resources) might not be as high as for the first one. This is because the synapse did not have enough time to recover from the previous spike.

Equations (2.1), (2.2) and (2.3) model this process. X, Y and Z are functions of time that represent: resources available on the pre-synaptic side (waiting neuro-transmitters), active resources on the post-synaptic side (released neuro-transmitters), and recoverable resources (that can be pumped back to the axon terminal). All variables x, y and z are fractions where 1 represents the maximum level of resources the synapse can provide.

$$\frac{\partial x}{\partial t} = \frac{z}{\tau_{rec}} - x(t_{sp})U_{SE}\delta(t - t_{sp}) \tag{2.1}$$

$$\frac{\partial y}{\partial t} = -\frac{y}{\tau_{in}} + x(t_{sp})U_{SE}\delta(t - t_{sp}) \tag{2.2}$$

$$\frac{\partial z}{\partial t} = \frac{y}{\tau_{in}} - \frac{z}{\tau_{rec}} \tag{2.3}$$

Because the shape and energy of the pre-synaptic spike is not important and does not influence the functioning of the synapse or the shape of the post-synaptic potential, it is modeled by a Dirac pulse δ that occurs at the time of the spike t_{sp}. When a spike is received, a fraction U_{SE} (utilization of synapse efficacy) of the available resources is released (second term of eq. 2.1). At the same time, the same quantity of resources is received on the post-synaptic side (second term of eq. 2.2). As soon as the released resources bind with the receptors the post-synaptic membrane potential starts to increase at a rate proportional to the amount of active resources (eq. 2.4).

$$\frac{\partial V}{\partial t} = A * y(t) \Rightarrow V(t) = A\int_{0}^{t} y(t)dt \tag{2.4}$$

BUPT

Constant *A* is a scaling constant and represents the static efficacy (weight) of the synapse. The first term of eq. 2.2 represents the rate at which the resource-receptor binds break apart. The time constant τ_{in} controls how fast resources become inactive. All inactive resources are in fact recoverable resources and so Z increases with the same rate (first term of eq. 2.3). The first term of eq. 2.1 and second term of eq. 2.3 show the rate at which resources are recovered (pumped from post-synaptic to pre-synaptic side).

Figure 2.4 presents the dynamic functionality of a biologic synapse and the variation of resource quantities (neurotransmitters) over time. It is considered that the first pre-synaptic spike of the sequence occurs at a moment when all resources have had time to recover and are available for use (x = 1, y = 0, z = 0). It can be noticed that the first spike generates a maximum amount of active resources. This is controlled by the value of U_{SE}, which in this case is 0.5. The next three spikes have a smaller effect because there is not enough time for the resources to recover. Notice that the resources activated by the 2nd, 3rd and 4th spikes are similar in amount due to the fact that the period between these spikes is roughly the same. This leads to the conclusion that the efficacy of the dynamic synapse is frequency dependent. Literature shows that this mechanism is present and very useful in biological neural networks because it stabilizes and prevents saturation of the network.

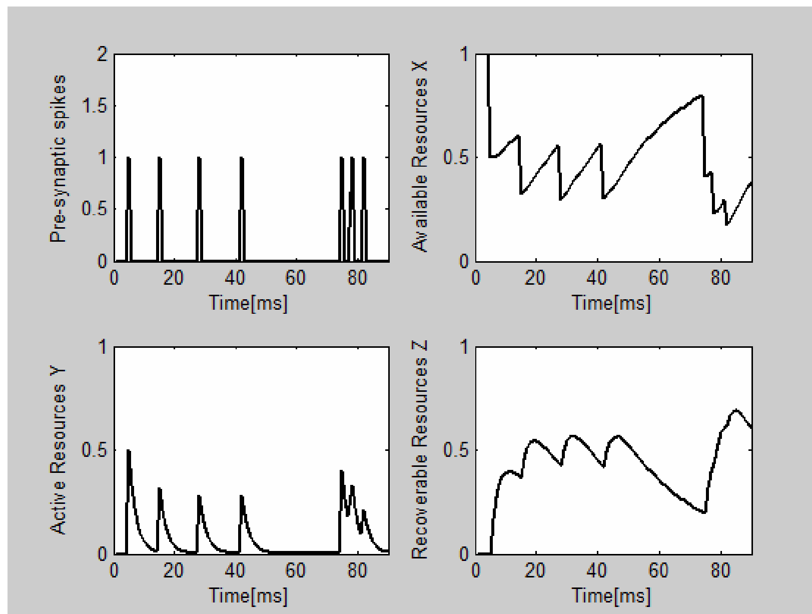


Fig. 2.4. Synapse Dynamic functionality

2.2.2. Synapse Delay

This sub-paragraph introduces a simplification to the synapse model in eq. 2.1, 2.2 and 2.3. As presented in 2.2.1 when a spike is received a certain amount of resources activate. Until these resources inactivate, they produce a current proportional to their amount causing the post-synaptic membrane potential to rise. We simplify the model by considering that resources activate and then inactivate

BUPT

instantaneously, reducing τ_{in} to zero. Doing so, they trigger their entire associated energy at once, causing the membrane potential to rise as a step function. The simplification causes the membrane potential to rise faster and can potentially lead to a post-synaptic neuron that fires a spike prematurely. In compensation, we introduce a delay between the time when the pre-synaptic spike arrives and the time when the resources activate. This delay is equal to the time the membrane potential would need to rise to the K fraction of its final value (eq. 2.5). D_0 is a delay that accounts for all other delays that could potentially appear in a biological neuron (flight time of pre-synaptic spike across the axon length). The model described by equations 2.1, 2.2, 2.3 changes to the model in 2.6, 2.7 and 2.8. N is the total number of pre-synaptic spikes.

$$D = -\tau_{in} \ln(1-K) + D_0 \tag{2.5}$$

$$\frac{\partial x}{\partial t} = \frac{z}{\tau_{rec}} - x(t_{sp})U_{SE}\delta(t - t_{sp}) \tag{2.6}$$

$$y(t) = U_{SE}\sum_{n=1}^{N} x(t_{spn})\delta(t - t_{spn} - D) \tag{2.7}$$

$$\frac{\partial z}{\partial t} = -\frac{z}{\tau_{rec}} + x(t_{sp})U_{SE}\delta(t - t_{sp} - D) \tag{2.8}$$

2.2.3. Facilitating Synapse

In the previous paragraphs U_{SE} is constant and controls the amount of resources that become active as the result of a pre-synaptic spike. In reality, U_{SE} is not constant and is affected by the past activity of the synapse. A model of this mechanism as presented in equation 2.9. When a pre-synaptic spike is received the increased voltage inside the axon terminal causes the calcium channels to open. The fraction of channels that open (out of the total number of available channels) is represented by U_{SE}. After opening, the channels close at a rate given by τ_{facil} (first term in eq.2.9). If a new pre-synaptic spike is received, a fraction U_{SE} of the closed channels (1 - $\boldsymbol{u_{SE}}$) open additionally to the channels that are already open ($\boldsymbol{u_{SE}}$). Therefore, the number of channels that open is higher than U_{SE} and is equivalent to the instantaneous value of $\boldsymbol{u_{SE}}$. This type of synapse is called to be facilitating. A spike will increase the efficacy of the synapse if the subsequent spikes arrive in a time window comparable to τ_{facil}. The depressing and facilitating mechanisms are contradictory but have different dynamics and time constants. Therefore, adjusting the τ_{rec} and τ_{facil} can change significantly the behavior of the dynamic synapse. Additional information about the processing power of synapses can be found in [46] and [49].

$$\frac{\partial u_{SE}}{\partial t} = -\frac{u_{SE}}{\tau_{facil}} + U_{SE}(1 - u_{SE})\delta(t - t_{sp}) \tag{2.9}$$

2.2.4. Recursive Model for Dynamic Synapse

The dynamic synapse can be modeled globally by a gain *G* and a delay *D*. The synapse is dynamic because the gain is not constant and depends on an internal synapse state [x u_{SE}]. At each moment the gain is controlled by the state variable as shown in equation 2.10. Superscript index *n* is a discrete time index associated with the nth pre-synaptic spike.

$$G^n = A * u_{SE}^n * x^n \tag{2.10}$$

Equations 2.6 and 2.9 can be transformed from differential into recursive expressions that allows us to compute the state variable at the time of the $(n+1)^{th}$ spike based on the state variable at the time of the nth spike. Equation 2.11 describes the computation of x^{n+1} based on x^n and u_{SE}^n. Δt_n^{n+1} is the time interval between the consecutive pre-synaptic spikes. In a similar way equation 2.9 can be rewritten as a recursive expression in 2.12, where U_{SE} is the utilization of synaptic efficacy in static conditions (low frequency spike trains compared to 1/ τ_{facil}).

$$x^{n+1} - x^n = -\underbrace{x^n u_{SE}^n}_{activated} + \underbrace{\left(1 - x^n + x^n u_{SE}^n\right)\left(1 - \exp\frac{-\Delta t_n^{n+1}}{\tau_{rec}}\right)}_{recovered} \Rightarrow$$

$$x^{n+1} = 1 + \left(x^n - x^n u_{SE}^n - 1\right)\exp\frac{-\Delta t_n^{n+1}}{\tau_{rec}} \tag{2.11}$$

$$u_{SE}^{n+1} = U_{SE} + u_{SE}^n\left(1 - U_{SE}\right)\exp\frac{-\Delta t_n^{n+1}}{\tau_{facil}} \tag{2.12}$$

A useful property of the above model is that whenever the pre-synaptic spike train has a steady frequency, the synapse state [x u_{SE}] stabilizes at a constant value that is dependent on the frequency of the spike train. This is a useful property that is used in the following chapters at building the spiking neural network simulation framework. Equations 2.13 and 2.14 compute the synapse state when the frequency of the spike train is steady and equal to f_{sp}.

$$x = \frac{1 - \exp\frac{-1}{f_{sp}\tau_{rec}}}{1 - \left(1 - u_{SE}\right)\exp\frac{-1}{f_{sp}\tau_{rec}}} \tag{2.13}$$

$$u_{SE} = \frac{U_{SE}}{1 - \left(1 - U_{SE}\right)\exp\frac{-1}{f_{sp}\tau_{facil}}} \tag{2.14}$$

BUPT

2.3. Spiking Neurons Models

Synapses transmit action potentials between pairs of neurons. When a neuron receives a post-synaptic spike its membrane potential is changed and in some cases the neuron generates an action potential. There are several models that simulate the mechanisms governing the functioning of a neuron cell. The most elaborate model is the Hodgkin-Huxley neuron which was developed after studying the giant squid neuron. Due to limited computing resources, several other simplified models have been later developed. These models try to preserve the aspects of neuron functionality that are believed to be significant to information processing. This thesis presents and uses four different models for spiking neurons. In order to justify the simplified models the next sub-paragraph introduces the functionality of the biological neuron as presented by Hodgkin-Huxley.

2.3. 1. The Biological Neuron

Like all cells the neuron has a cell membrane that separates it from the extra-cellular space. The interior of the cell is connected to the exterior by a series of ion pumps and voltage-controlled ion channels. Sodium, potassium and calcium ions are most important in the functioning of the neuron. The active pumps create a flux of ions between the interior and exterior of the neuron cell, thus creating a difference in ion concentration. Sodium is pumped out of the cell while potassium is pumped into the cell. The difference in concentration produces a voltage across the cell membrane called Nernst potential. This voltage opens the voltage-controlled ion channels and produces a flux of ions that is opposite to the pump flux, thus balancing the process. If only sodium was present the equilibrium is at about +50mV across the cell membrane. In the case of potassium equilibrium sets in at around -77mV. With both types of ions present, the equilibrium potential was experimentally determined around -65mV. At this voltage, sodium ions flow into the cell while potassium ion flow out of the cell (via the ion channels). On the other side, the ion pumps balance the process by pumping the ions back. The equilibrium voltage is called resting potential. This is the potential across the neuron membrane if the neuron is not disturbed by an external stimulus for a sufficiently long period of time. Figure 2.5 presents the Hodgkin-Huxley neuron model. The capacitor represents the neuron membrane that serves as an insulator between the inside and outside of the neuron cell. The two batteries represent the equilibrium potentials generated by the sodium and potassium ion flows. The ion channels are modeled by variable resistors that are controlled by the voltage across the membrane. The circuit is stimulated by an external current that represents the post-synaptic current.

Despite its apparent simplicity the model can have a very complex behavior. This is because the dependence of the channel conductance on the membrane potential is very dynamic and highly non-linear. Furthermore, combining two or more channels with different parameters can lead vast range of dynamic behaviors for the neuron model. Equation 2.15 describes the functionality of the model, where the sum is across all existing ion channels (Na and K in this case). Equation 2.16 sums the components of the sodium and potassium currents respectively. G_n is the maximum conductance of the channel. This conductance is modulated by the dynamic variables *m* and *h* which are called activating and inactivating variables. Parameters *p* and *q* are constants and are specific to the ion channel. In the case of

BUPT

the Hodgkin-Huxley neuron the parameters were experimentally determined (p_{Na}=3, q_{Na}=1, p_K=4, q_K=0). Because q_K is zero the potassium channel has no inactivation variable.

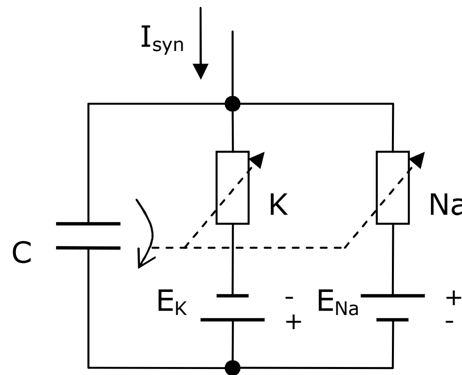


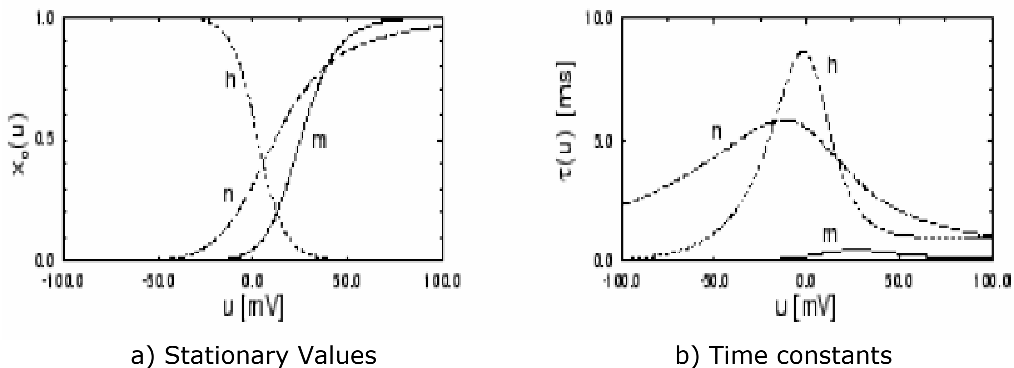
Fig. 2.5 Hodgkin-Huxley Neuron Model

$$C\frac{\partial u}{\partial t} = -\sum_n I_n(t) + I_{syn}(t) \tag{2.15}$$

$$I_{Na} + I_K = g_{Na} m_{Na}^{\ p_{Na}} h_{Na}^{\ q_{Na}} (u - E_{Na}) + g_K m_K^{\ p_K} h_K^{\ q_K} (u - E_K) \tag{2.16}$$

The activation and inactivation variables *m* and *h* are voltage dependent and also time dependent, therefore they have a dynamic behavior. Figure 2.6 shows an example of how these variables behave. The left picture represents the static value of the variable. The right picture shows the time constant of the variable.

$$\frac{\partial x}{\partial t} = -\frac{1}{\tau(u)}(x - x_0(u)) \tag{2.17}$$



a) Stationary Values b) Time constants

Fig. 2.6 Activation and inactivation variables

BUPT

During the transitory regime, variable x changes asymptotically towards the new stationary value $x_0(u)$ with time constant $\tau(u)$ (eq.2.17). Note that both the stationary value and the time constant are voltage dependent. It is important to notice that the m and h variables model biological mechanisms that activate and de-activate the ion channels independently. Therefore, the variables are also independent; they can change with different time constants; and their effect is multiplicative since any of the variables can annihilate the effect of the other.

2.3.1.1. Spike Generation

When no external stimulus is present the neuron settles at resting potential. This means that the activating and inactivating variables have just the right value to keep the Na and K ion channels open at the levels required for equilibrium. If an external synaptic current is injected the potential across the membrane potential rises. As an effect all variables m and h also rise for both Na and K. Notice that the activating variable m_{Na} of the Na channel has a time constant that is significantly smaller than the rest. This means that this variable will rise a lot faster that the other. It also means that the Na ion channel will open first allowing an influx of Na ions inside the cell. This causes an additional increase in membrane potential which in turn opens the Na channel even more. Therefore, a temporary positive feedback appears.

If the external stimulus is weak and initial increase in membrane potential is also small. This places the m_{Na} variable at a position on the graph where the sensitivity of the variable is small with respect to voltage variations. Consequently, the positive feedback created by the Na channel is small and is unable to de-stabilize the neuron. This corresponds to the situation presented in figure 2.7a. Having larger time constants it takes some time until the inactivating variable h_{Na} of the Na channel and the activating variable m_K of the K channel rise. However, when that happens two effects can be seen. First, the Na channel is closed by its inactivating variable. Second, the K channel is opened producing an ex-flux of K ions out of the cell. This produces a decrease in the membrane potential. Because of the high time constant of the K channel there is a certain delay until the channel manages to close. This creates a negative overshoot of membrane potential.

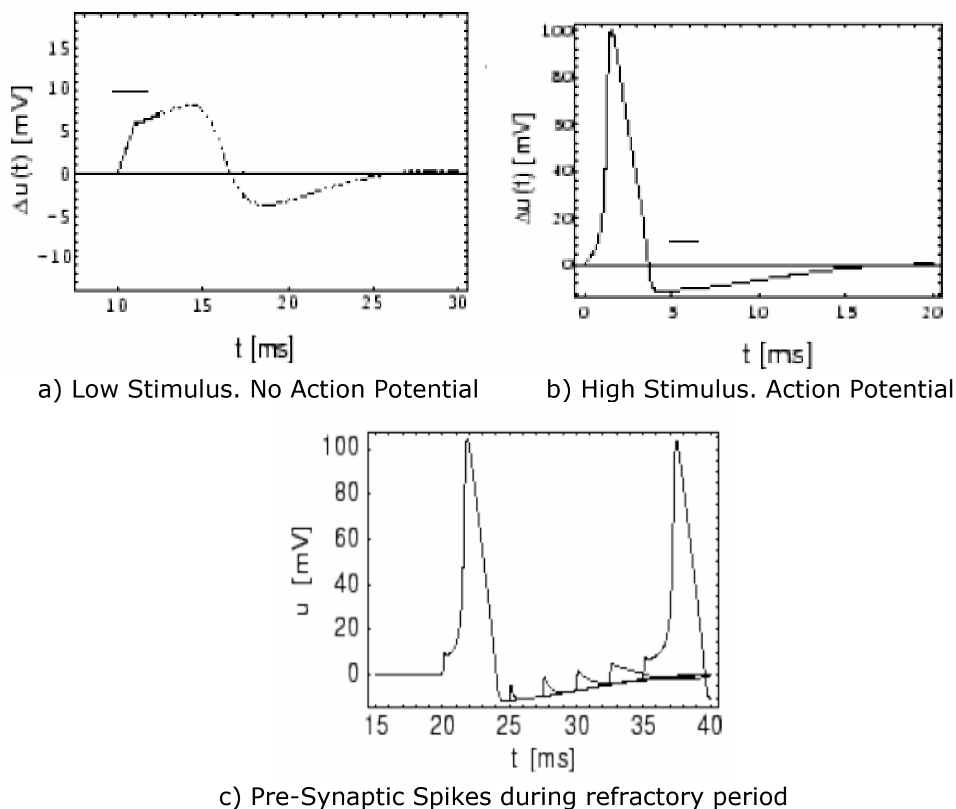
If the external stimulus is strong it will produce a higher membrane potential and will bring the Na channel in a state where it is more sensitive to potential variations (upper part of the graph). This creates a stronger positive feedback. If the membrane potential is sufficiently high (reaches a threshold), the positive feedback is sufficiently large to sustain itself. This opens the Na channel to maximum allowing a massive in-flux on Na ions inside the neuron and leading to the generation of an action potential. This is shown in figure 2.7b. At such high values of the membrane potential the time constants of the h_{Na} and m_K decrease significantly allowing quick inactivation on the Na channel and activation of the K channel. The high value of the membrane potential opens the K channel more than in the situation when an action potential is not generated. The result is a significantly stronger negative overshoot called refractory period.

2.3.1.2. Refractory Period

During the refractory period the neuron is less sensitive to incoming stimuli. This means that a stimulus that would generate an action potential if the neuron is in resting state might not generate an action potential if the neuron is in refractory state. The effect of a refractory period is not just a shift towards a more negative potential (an increased distance to the threshold). In addition, during the refractory

BUPT

period the overall conductance of the neuron membrane is reduced because very many ion channels are opened immediately after the action potential. This increased conductance allows the charge that is brought by the external stimulus to drain quickly and so diminishes the impact of the stimulus upon the membrane potential during the succeeding time window. This is depicted in figure 2.7c. There is a sequence of four spikes during the refractory period following the action potential. All spikes carry the same amount of energy and raise the membrane potential equally. However, the potential brought by the first spike diminishes a lot faster than the potential brought by the following spikes. This is because the refractory effect is more profound then. During the refractory period the chance that a group of spikes triggers an action potential is reduced. The same group of spikes might easily trigger an action potential if the neuron was at resting potential. In addition, if the group of spikes merged into a single spike that carries the cumulated energy it might trigger an action potential even if the neuron is in refractory period.



a) Low Stimulus. No Action Potential b) High Stimulus. Action Potential

c) Pre-Synaptic Spikes during refractory period

Fig. 2.7. Functioning of the Neuron. Generating spikes

2.3. 2. Integrate and Fire Neuron

The integrate-and-fire neuron model (I&F) is the simplest model inspired from the Hodgkin-Huxley neuron. The I&F neuron is shown in figure 2.8 and is modeled by equation 2.18 or 2.19 (differential form). The synapses are constructed

BUPT

from a gain block and a delay block. The gain block is either a simple constant gain or can be built as a state machine modeled by equation 2.10, 2.11 and 2.12. It can be seen that the neuron cell is represented by a lossy integrator which is the equivalent of the capacitor. The resting potential of the neuron is zero. All post-synaptic spikes are added and accumulated by the integrator as membrane potential. If the potential reaches threshold ϑ the integrator resets to its resting potential and an action potential is generated at the output. If the threshold is not reached and no new post-synaptic spikes arrive the integrator leaks to its resting potential with loss factor K. Two possibilities are proposed for modeling the refractory period. In figure 2.8 it is implemented by a switch controlled by a timing circuit. This prevents the neuron to accumulate any potential during the refractory period and so no new action potentials can be generated during this period.

Another method would be to reset the integrator at a value lower than its resting potential. This approach is similar to what the biological neuron does, and decreases the probability that a new action potential is generated during the refractory period. However, this model is linear while the biological neuron is highly non-linear and has a significantly higher probability to generate a new action potential during the refractory period. For this reason and also from experimental results the discontinuous switch controlled model (fig. 2.8) is preferred.

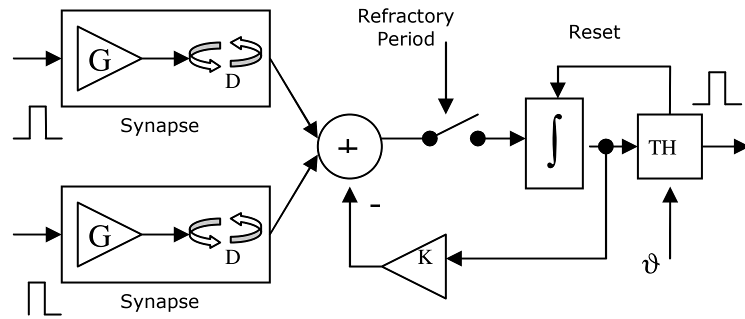


Fig. 2.8. Integrate and fire neuron model

$$p_i(t)=\int_{ti}^{t}\sum_{k=1}^{K}G_{ik}*u_i(t-D_{ik})-k_loss*p_i(t) \qquad p_i(t)\le Th \qquad (2.18)$$

$$p_i(t)=0,\ ti=t \qquad p_i(t)\ge Th$$

$$\frac{\partial p_i(t)}{\partial t}=I_{syn}-k_loss*p_i(t) \qquad (2.19)$$

The leakage current models the effect produced by the potassium currents which tend to bring the neuron at resting potential if the post-synaptic stimulus is not sufficient to trigger an action potential. The main advantage of this model is its simplicity, requiring a reduced number of floating-point operations for simulating one neuron. This advantage can be important if very large networks need to be simulated. As a tradeoff, the model does not have adaptation or bursting abilities.

BUPT

2.3.3. Integrate and Fire with Burst Neuron

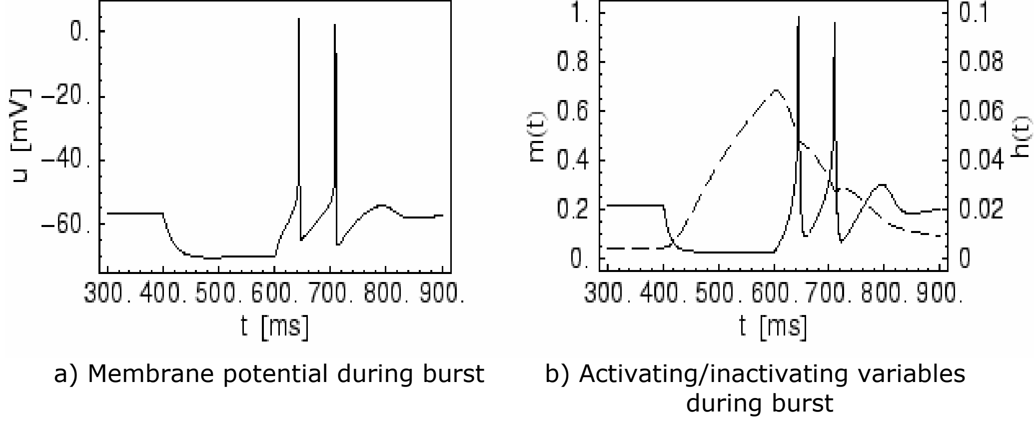
A burst is a sequence of action potentials generated at very short time intervals. The first action potential is triggered by the same mechanisms of a non-bursting neuron. The remaining spikes of the sequences are triggered by a self-sustained mechanism and do not need any synaptic stimulus. The biological explanation for such functionality stands in the existence of the calcium ion channel which was mentioned in paragraph 2.3.1. Like the sodium current, the calcium current is also from the outside towards the inside of the neuron cell, thus raises the membrane potential. The mathematical model of the calcium current is given by equation 2.16 and so is the same as that of sodium or potassium. Additionally, the shape of the activating and inactivating variables m and h is also similar. The distinctive characteristic of the calcium ion current is that the curves of the activating and inactivating variables are shifted left towards the negative area of the membrane potential. This is the reason why this current is also called "low-threshold calcium current". At resting potential the m_{Ca} variable is fully activated while the h_{Ca} variable is fully inactivated. Therefore, the calcium channel is closed as an effect of the h_{Ca} variable. The sodium channel is also closed at resting potential as an effect of the m_{Na} variable. Even if the sodium channel and calcium channel are in the same state, closed, each has a different reason for it: the sodium channel is not-activated, while the calcium channel is in-activated. If for any reason the membrane potential decreases below the resting potential the inactivating variable h_{Ca} will start to rise and the calcium channel will open. The membrane potential can go below the resting potential either as the result of an inhibitory synaptic current or an after-spike negative overshoot (due to potassium currents).

Figure 2.9 shows an example the neuron is stimulated with an inhibitory synaptic current. The inhibitory current is removed (at time 600ms) the membrane potential starts to increase, m_{Ca} starts to increase and h_{Ca} starts to decrease. Because h_{Ca} has a large time constant and does not decrease immediately the activating variable m_{Ca} opens the calcium channel. As a result the membrane potential is increases and might cause a sodium channel to open and generate an action potential. After the first action potential, if the h_{Ca} still has not had time to decrease, the calcium current is still on and causes a new action potential. Several such action potentials can be generated in a burst. The number of action potentials is given by the time window allowed by the time constant of the h_{Ca} variable. Once the calcium channel closes the mechanism cannot be triggered until the membrane potential is again sufficiently low.

The I&FB neuron is a simplified model that tries to capture the functionality described above. Equations 2.20 and 2.21 implement the model. The first two terms of eq.2.20 are the same as for the I&F neuron. Additionally, there is a third term related to the calcium current. The term p_{Ca} is smaller than the resting potential and sets the potential level where the calcium channel starts to open (m_{Ca} and h_{Ca} curves cross). Equation 2.21 shows that whenever the potential is below p_{Ca} the calcium current increases and when the potential is above p_{Ca} the calcium current decreases. In the case of the biological neuron, both the m and h variables influence the calcium channel at the same time. For simplicity, this model considers that the calcium current increases as effect of the m variable and that the current decreases as effect of the h variable. Consequently, the τ_m time constant controls the rate at

BUPT

which the calcium current increases while the τ_h controls the rate at which the current decreases.



a) Membrane potential during burst b) Activating/inactivating variables during burst

Fig. 2.9. Calcium Current Effect

Function *H* of eq. 2.20 is the Heaviside function. Its purpose is to disable the calcium current when the potential is below p_{Ca}. In the case of the biological neuron a membrane potential below p_{Ca} prepares the calcium channel to open by raising the level of h_{Ca}. However, the channel is closed because of the m_{Ca} variable and remains so until the membrane potential is above p_{Ca}. Because in a similar situation the model described in eq. 2.20 and 2.21 has an increasing current, the current needs to be shunted artificially until the membrane potential is above p_{Ca}. This is achieved by the *H* function.

The calcium current influences the neuron at small potentials above the resting potential. It generally helps at producing sodium current and does not trigger the action potential itself. For this purpose, the calcium current term in eq. 2.20 is modulated by a term that represents the distance between the membrane potential and the reference potential p_T.

$$\frac{\partial p}{\partial t} = I_{syn} - Kp + I_{Ca} H(p - p_{Ca})(p_T - p) \tag{2.20}$$

$$\frac{\partial I_{Ca}}{\partial t} = \begin{cases} -I_{Ca}^{\max} \dfrac{I_{Ca}}{\tau_h}, p > p_{Ca} \\ \dfrac{I_{Ca}^{\max} - I_{Ca}}{\tau_m}, p < p_{Ca} \end{cases} \tag{2.21}$$

If parameter p_{Ca} is set at a level that is reached by the membrane potential during the refractory period and also if parameters $I_{Ca}{}^{max}$ and τ_m are sufficiently large, any sodium generated action potential can trigger a burst. For this case the refractory period cannot be modeled by a switched as described in 2.3.2. The duration of the burst can be controlled by parameter τ_h.

BUPT

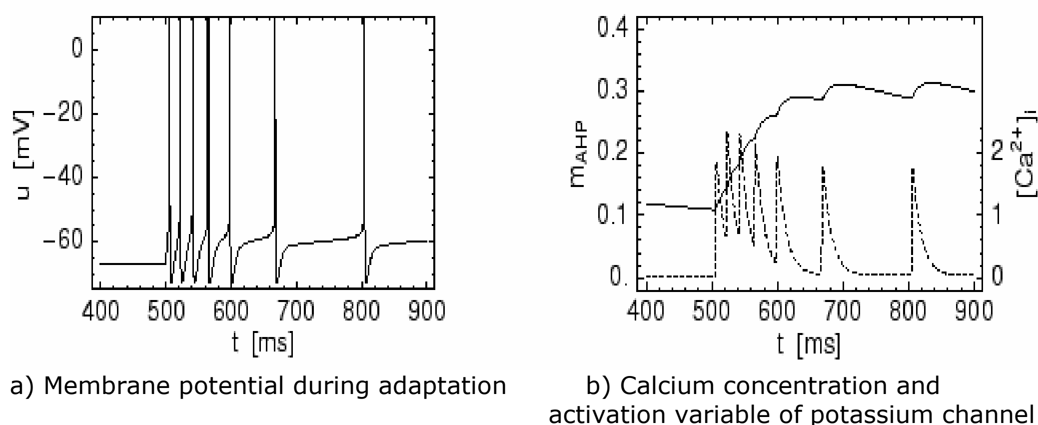
2.3. 4. Integrate and Fire with Adaptation Neuron

Adaptation is a neuron property to adjust its sensitivity to incoming stimulus in order to prevent over excitation and output saturation. The biological neuron is able to adapt by combining two special ion currents. The first current is called "high-threshold calcium current". This current is very similar to the low-threshold calcium current presented in the previous paragraph. The difference is that this current activates only at high values of the membrane potential, particularly during the generation of action potentials. The high threshold calcium current has two roles: firstly, it produces an additional boost to the membrane potential during the action potential; secondly, and more important for the adaptation mechanism, it temporarily increases the calcium concentration inside the neuron cell immediately after the action potential.

In addition, a new type of potassium current is present. It differs from the regular K currents by the fact that it flows through a calcium dependent-voltage independent potassium channel. Consequently, this channel opens when the calcium concentration is high regardless of the potential across the neuron membrane. This dependence law is shown in equation 2.22, where Ca^{2+} is the calcium concentration and c is a parameter.

$$\frac{\partial m}{\partial t} = \alpha m - \beta(1-m), \alpha = \min\left(c\left[Ca^{2+}\right], 0.01\right), \beta = 0.001 \tag{2.22}$$

If an isolated action potential occurs the calcium concentration rises but the calcium dependent potassium channel does not have sufficient time to open. On the other hand, figure 2.10b shows that whenever several action potentials occur during a short interval of time, the persistent high calcium concentration allow the potassium channels to open. As a result, the outward potassium current acts as an inhibition current. Figure 2.10a shows that the frequency of action potentials decreases even if the input stimulus is constant, exhibiting a mechanism of adaptation against over stimulation.



a) Membrane potential during adaptation

b) Calcium concentration and activation variable of potassium channel

Fig. 2.10. Adaptation Mechanism

BUPT

$$\frac{\partial p}{\partial t} = I_{syn} - Kp + I_K^{Ca}\left(p_K^{Ca} - p\right) \tag{2.23}$$

$$\frac{\partial I_K^{Ca}}{\partial t} = \frac{I_K^{\max}\delta\left(t_{spk}\right) - I_K^{Ca}}{\tau_K^{Ca}} \tag{2.24}$$

2.3. 5. Resonate and Fire Neuron

Resonate and fire (R&F) neurons exhibit sub-threshold oscillations when an action potential is not generated. This type of behavior makes the neuron sensitive to the exact timing of the pre-synaptic spikes and therefore requires an excitatory spike train that resonates with its internal frequency in order to generate an action potential. Many biological neurons also exhibit such behaviors. To explain such functionality we return to the biological model containing a sodium channel and a potassium channel. When the neuron is stimulated the sodium channel opens and the membrane potential rises. If the increase is not enough to trigger an action potential, the sodium channel inactivates and the potassium channel opens causing the membrane potential to decrease. However, the membrane potential does not decrease to the same extent as it would in the case of an action potential. The low potential inactivates the potassium channel and "de-inactivates" the sodium channel causing the membrane potential to rise again. After a few such damped oscillations the membrane comes to a stop at the resting potential. When an action potential is generated, the low membrane potential during the refractory period shuts down the sodium channel completely and so an oscillation does not occur.

The easiest way to model an R&F neuron is to represent the membrane potential as a complex number p, where the real part is the current component and the imaginary part is the voltage component. Equation 2.25 models the variations of membrane potential for an R&F neuron. Parameter b is negative and represents the attraction to the rest state. ω is the internal frequency of the neuron. Parameter a is introduced to modulate the amplitude and phase of the synaptic current. However, in most cases it is a real number and therefore cannot shift the phase of the membrane potential.

$$\frac{\partial p}{\partial t} = aI_{syn} + \left(b + j\omega\right)p \qquad \mathrm{Im}\left(p\right) < Th \tag{2.25}$$

$$p = jTh \text{ or } p = 0 \qquad \mathrm{Im}\left(p\right) \geq Th \tag{2.26}$$

Whenever the voltage component (Im(p)) exceeds the threshold the membrane potential is brought back to an initial reset state and the neuron output generates an action potential. The reset state is either zero, which is analog to the resting state of the previous models, or can be any complex number (i.e. *jTh*). In the latter case the neuron continues to oscillate starting from the amplitude and phase given by the reset state.

Figure 2.11 plots the membrane potential of a R&F neuron when it is stimulated with four different spike trains. In 2.11a the three spikes are grouped and arrive within a short interval of time. This boosts the membrane potential above the threshold and triggers an output action potential. The action potential is represented by the solid round marker. In this case the reset state is *j*TH.*

BUPT

In 2.11b the spikes are out of phase relative to the oscillation of the membrane potential. It can be seen that the third input spike actually lowers the amplitude of the oscillation and therefore diminishes the probability of an action potential. In 2.11c the third input spike has a timing that is not close together with the first and second spike. Even so, because the timing of the third spike has roughly the same phase relative to the phase of the oscillation an action potential is generated. In figure 2.11d the second spike is out of phase with the other spikes. However, because the spike is inhibitory, it is able to trigger an action potential.

Additional information about neuron models can be found in [47]

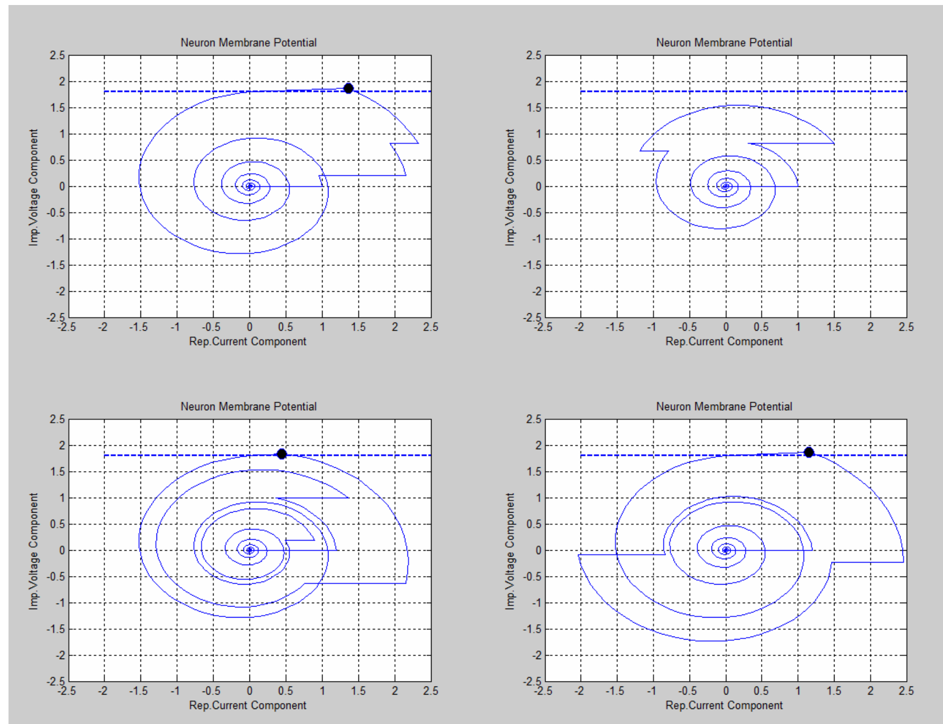


Fig. 2.11. R&F Neuron Membrane Potential

2.4. Coding with Spikes

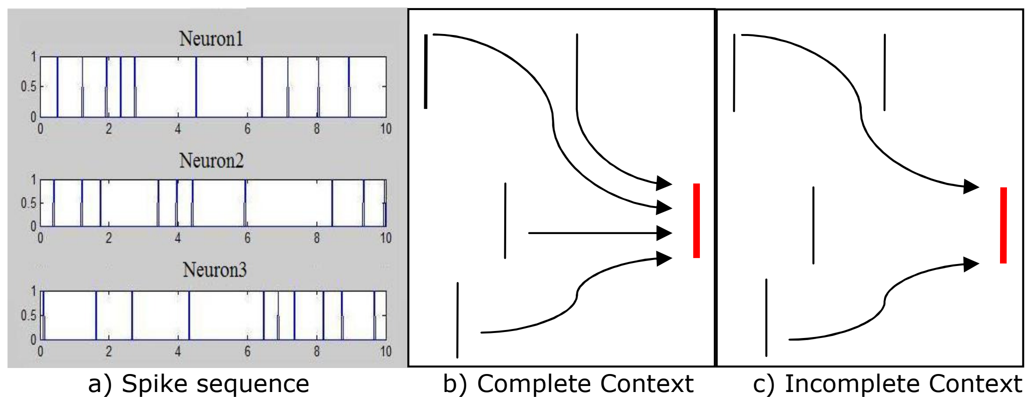
One of the most difficult problems involving processing with neural networks is finding an efficient method to represent and code information. Several approaches have been tried each of them having advantages and disadvantages [32], [34]. For example, some neural network models use a continuous variable as output even though the biological counterpart generates spikes. This approach is based on the assumption that the information is coded entirely in the rate of the spike train. Hence, time-averaging along the signal reduces it to a continuous variable.

Another method, which is presented in this paragraph, is coding by using spatial-temporal spike sequences. Such a sequence codes information in the relative timing between spikes and also in the identity of the neuron that generates the spike. Figure 2.12 presents an example of a spatial-temporal spike sequence generated by three neurons.

BUPT

The complete description of such a spike sequence is done by specifying the timing of each individual spike relative to the remaining spikes of the sequence. Such a set of time delays is called the "context" of the spike. There are two types of contexts as they include all the spikes of the sequence or just a partial subset. Figure 2.12 presents the two types of contexts and calls them complete contexts and incomplete contexts respectively. The number of spikes that comprises each context is referred to as "context size".

Consequently, a network that generates a spatial-temporal spike sequence needs to be comprised of several context detectors each of them being responsible for generating a spike if and only if the appropriate context has occurred. The next section presents a method of implementing a context detector by using an I&F spiking neuron. Further details concerning this coding scheme can be found in [25], [35], [36], [37] and [38].



a) Spike sequence b) Complete Context c) Incomplete Context

Fig. 2.12. Spatial-temporal spike sequences. Complete and incomplete spike contexts

2.4.1. Spiking Neuron as Context Detector

Every context contains a set of delays that correspond to the relative timings between spikes. If these delay values are programmed in the delay blocks of the neural synapses the occurrence of a certain context causes several spikes to be synchronized at the input of the neuron. This event can be easily detected if the neuron uses a large loss factor combined with appropriate synapse and threshold values. This is because the large loss factor does not allow the neuron potential to exceed the threshold unless all of the expected spikes arrive at the same time or within a very short time interval.

The value of the loss factor offers a mechanism that allows us to introduce a certain tolerance to the exactness with which the context needs to be reproduced in order to be detected. This means that if the context is similar but not identical to the context used during training the spikes will not be fully synchronized. However, if the loss factor is not too large a small de-synchronization is allowed and the context is still recognized. On the other hand if the loss factor is too small confusion between contexts can occur. The remaining of this section presents an analytical approach of how to compute a loss factor when a certain time tolerance is desired.

BUPT

The tolerance is denoted by symbol t_Δ and represents the time delay between the first and the last of the N received spikes comprising a context.

The charging of the lossy integrator is given by formula (2.27), where S is the gain of the synapse and k is the loss factor. After receiving N-1 spikes the potential accumulated by the integrator reaches level A_1 (2.28). The amplitude of each spike is 1 and the width is W. It is important that the neuron is not triggered after receiving the first N-1 spikes and therefore the condition $A_1 \leq Th$ must be satisfied. This leads to a restriction imposed on the synapse gain (2.29). Until the last spike is received at time t_Δ the integrator discharges from level A_1 to level A_2 (2.30). The last spike charges the integrator to level A_3 (2.31).

$$u(t) = \frac{NS}{k}\left(1 - e^{-kt}\right) \tag{2.27}$$

$$A_1 = \frac{(N-1)S}{k}\left(1 - e^{-kW}\right) \tag{2.28}$$

$$S_{\max} \leq \frac{kTh}{(N-1)\left(1 - e^{-kW}\right)} \tag{2.29}$$

$$A_2 = A_1 e^{-k(t_\Delta - W)} \tag{2.30}$$

$$A_3 = A_2 e^{-kW} + \frac{S}{k}\left(1 - e^{-kW}\right) \Rightarrow A_3 = \frac{S}{k}\left(1 - e^{-kW}\right)\left[1 + (N-1)e^{-kt_\Delta}\right] \tag{2.31}$$

In order for the context to be detected it is necessary that level A_3 exceeds threshold Th. Equation (2.31) shows that the highest value for k is achieved when $S=S_{max}$. Substitution of (2.29) in (2.31) and regrouping leads to condition (2.32). Equation (2.32) shows that when a high tolerance is desired the loss factor k needs to be decreased. When the size of the context N is large it is difficult to obtain a high time tolerance due to the excessively small loss factor. Theoretically, when N goes to infinitely the tolerance will need to be zero because $\ln\frac{N-2}{N-1} \to 0$.

Having defined how a single neuron can implement a context detector, the next section will present how a spiking neural network can implement a content addressable memory and also why this is useful.

$$Th < \frac{S}{k}\left(1 - e^{-kW}\right)\left[1 + (N-1)e^{-kt_\Delta}\right] \tag{2.32}$$

$$\frac{1 + (N-1)e^{-kt_\Delta}}{N-1} > 1 \Rightarrow e^{-kt_\Delta} > \frac{N-2}{N-1} \Rightarrow$$

$$k < -\frac{1}{t_\Delta}\ln\frac{N-2}{N-1} \tag{2.33}$$

BUPT

2.4.2. Content Addressable Memory with Spiking Neural Networks

Content addressable memories are different from conventional memories because of the addressing method they use. Rather than using an address as reference to a certain memory cell content addressable memories retrieve a memorized item by specifying an incomplete or noise affected version of the item itself. This approach is similar to the human brain functionality which remembers learned data when being stimulated with some similar or incomplete data.

Implementing a content addressable memory that stores spatial-temporal spike sequences can be useful in feature classification applications. Assuming that such a spike sequence is used for coding information, the sequence can be stored in a content addressable memory and then be used as the class prototype vector. Once the memory is addressed with a new spike sequence it will retrieve the prototype sequence that is most similar, hence classification is performed. Configuring a spiking neural network as a content addressable memory is done in three steps:

- A neuron is allocated for every spike in the sequence.
- For every spike (neuron) a context is chosen assuming that a context size has been pre-determined. A procedure to choose a context and also a context size is proposed in section 2.4.2.1.
- According to each context, synapse gains and delays can be programmed. If tolerance is desired an appropriate neural loss factor can be computed with formula (2.33).

The functionality is as follows: a new spike sequence is used to stimulate the network. If this sequence is similar to the sequence that is stored in the network some spikes might create appropriate contexts causing some of the missing spikes also to be generated. These new spikes will lead to new contexts and so after a few iterations all the missing spikes of the sequence will be added. A detailed description about the capacity of a content addressable memory implemented by a spiking neural network is found in [25]. The next section investigates how different parameters influence the ability of a network to successfully recall a stored spike sequence.

2.4.2.1. Choosing Spike Contexts

The previous paragraph defined the performance of a content addressable memory as the ability to completely reproduce a stored spike sequence from a partial sub-sequence. For evaluation purposes, we create a performance variable. The variable represents the ratio of the spikes that are correctly reproduced by the memory and the total number of spikes in the sequence. Ideally, after the initial sub-sequence is externally fed, the network adds all of the missing spikes. In practice however, due to several recursive dependencies some contexts are never completed and thus some spikes are not generated. This leads to other several incomplete contexts and so the network fails to self-lock on the entire spike sequence. Two factors that have significant influence on the network performance are the size of the contexts and the number of spikes used for initial excitation. When a large number of spikes are used for excitation the chance of a recursive dependency to appear is small. On the other hand a large value for context size increases this chance.

BUPT

When designing a network a compromise needs to be made with respect to the size of the context. This is because a context that is too large will most likely lead to the inability of the spike sequence to complete. A context that is too small will solve that problem but will reduce the robustness of the design because small contexts can falsely occur due to noise. The rest of the section will study the dependence of the network performance on the size of the context. The experiment generates a random sequence consisting of 100 spikes which is memorized in a neural network. The excitation of the network is done with partial sequences consisting of 50 spikes that are randomly picked out of the initial sequence. In this analysis every context is constructed from spikes picked out of the sequence according to a uniform spatial-temporal distribution. For every possible size of the context a statistical evaluation of performance is done by averaging the performance of 30 simulations.

Figure 2.13 presents the results of the analysis by plotting a family of curves parameterized by parameter C. The parameter models a degree of permissiveness that allows the algorithm to consider that a context is completed even if only a majority of the expected spikes have occurred. For example, if the size of the context is 10 and C is 2 than the context detector will activate if 8, 9 or 10 spikes of the context have occurred. The neuron context detector presented in section 2.4.1 can have this characteristic if the synapses are set to a value higher than S_{max} that is given in formula (2.29).

All the curves in the graph show that at some point when the context size is too large the performance drops as recursive dependencies start to appear. As expected, a larger value for C destroys some of these dependencies and so allows designs to perform well for larger contexts.

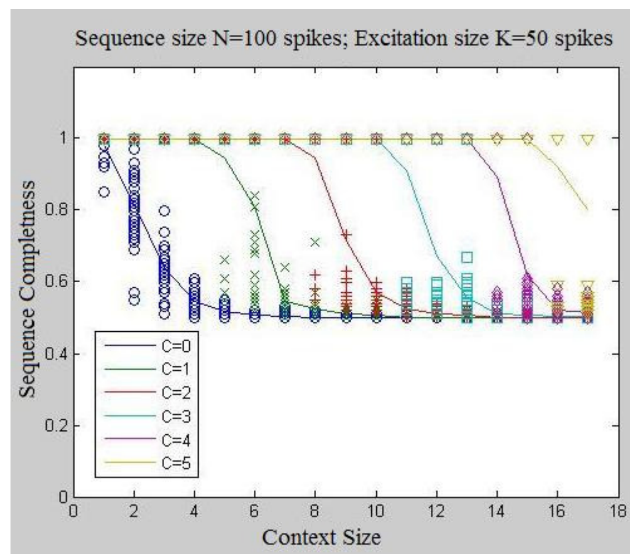


Fig. 2.13. Context size influence on performance.

An interesting observation can be made by examining individual performance markers rather than the average performance curve. It can be seen that curves having a larger value for C do not have markers distributed around the average as mostly expected. In this case the average is obtained from groups of

BUPT

points having either high performance or low performance, the average being determined by the number of points in each group. This type of behavior can be explained by examining the reason why performance drops. As stated before, performance decreases when some spikes do not occur due to incompletion of the contexts they depend on. This phenomenon is present when a recursive dependency occurs, meaning that a spike is never generated due to a linked dependency on itself. When C is small these dependencies can be present in small groups (even in pairs for C=0). Therefore, small groups of spikes fail to be generated causing the performance to be decreased only by a small percentage and so allowing markers to be distributed around the average. When C is large, the permissiveness of the context detector breaks the dependencies that appear in small groups allowing the performance to be high even for larger values for the context size. In this case, the performance decreases only as an effect of large group dependencies. This causes a sudden performance loss as several spikes fail to be generated together. Additional information on this topic can be found in [39].

Other ways of evaluating performance are presented in [25]. While our study presents the dependence of performance on the context size, [25] assumes a fixed context size and analyzes the ability of the network to store multiple spike sequences (capacity study) and to reject pattern noise.

BUPT

3. MODELING AND SIMULATION

In order to investigate the functionality of a spiking neural network, a simulation environment is required. Tools for processing and visualizing results are also needed. Such an environment is MATLAB [57], which preferred by most scientists due to its vast library of functions and toolboxes which are oriented towards scientific modeling and experimentation. Unfortunately, MATLAB does not include a toolbox that is suited for simulation of spiking neural networks. Therefore, one had to be developed. This chapter presents a MATLAB framework, designed by our research team, which can be used to simulate spiking neural networks [55]. It also describes some functions that are useful for processing and visualizing results.

3.1. Model Objects

The model is organized on objects of different hierarchical standings. This approach allows a direct association between software modules and actual parts of the network architecture, easing the task of extracting and interpreting simulation results. Figure 3.1 presents how the objects are organized.

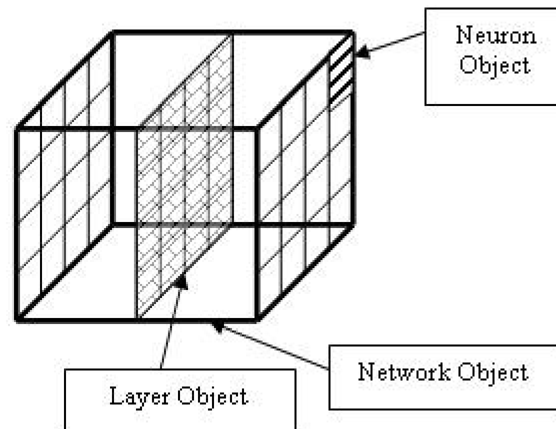


Fig.3.1. Model architecture

3.1.1. Network Object

The highest hierarchy object is the network object. It includes the entire network architecture, specifically all network parameters and all sub-objects. Figure 3.2a presents the contents of a network object. The network is volumetric, is organized on layers and each layer has a bi-dimensional topology. In this case the dimensions of the network are 3x5x5. An important variable of the network object is the layer array. The array has several entries, each being filled by one layer object. Other variables memorize network constants such as the membrane loss factor and neuron threshold. In order to create a network object function “createNetwork” is used.

3.1.2. Layer Object

The content of the layer object is presented in figure 3.2b. Some parameters like topology, loss factor and threshold are stored redundantly and are the same as found in the network object. The model is intended to be general and allows connections between any pair of layers. In practice, however, only some of these connections are valid and the rest are zero. For this reason every layer object contains a variable called connectivity vector. It is a Boolean map that shows which layers are connected to the current layer. This is useful because the simulator will examine this vector and will skip propagating spikes between layers with null connections, decreasing simulation time significantly. In this example the connectivity vector holds only zeros as the network has not been initialized.

The simulator, as it will be described later, is a time based solver. The output of the simulation is the spike activity of the network. Each spike is represented by a Boolean value coding the presence or the absence of the spike. Variable "state" of the layer object holds the spike activity of the layer at the current time instant. The variable is a matrix of the same size as the layer topology. Every entry in this matrix corresponds to the output of one individual neuron. This means that building the spike trace of one neuron is done by recording the value of one matrix entry at every time instant of the simulation.

```
>> network = create_network(3,[5 5],0.1,5)

network =

      nr_layers: 3
       topology: [5 5]
    loss_factor: [NaN 0.1000 0.1000]
      threshold: [NaN 5 5]
    layer_array: {[1x1 struct]  [1x1 struct]
                  [1x1 struct]}
```

a) Network object

```
>> network.layer_array(2)

ans =

       topology: [5 5]
       layer_nr: 2
    input_layer: 0
    loss_factor: 0.1000
      threshold: 5
          state: [5x5 double]
     next_state: [5x5 double]
    conectivity: [0 0 0]
   neuron_array: {5x5 cell}
```

b) Layer object

Fig. 3.2. Network and Layer Objects

The layer object contains a secondary state variable named "nextState". The reason why two state variables are needed comes from the functional difference that exists between a biological neural system and a computer system. All the neurons of the biological system operate in parallel while the computer code is executed sequentially. This means that any change in the output of one neuron will affect the simulation of the next neuron. This is incorrect since normally the two neurons are performing in parallel. In order to preserve the parallel functionality of the network two state variables are used. All neurons read signals from the "state" variable while their outputs write signals into the "nextState" variable. The framework considers time to remain unchanged until all the neurons are simulated. When the "nextState" variable is complete the simulator swaps the two variables and time is advanced. This technique is shown in figure 3.3. The "neuronArray"

BUPT

variable is an array with the same topology as the layer object. This array holds all the neuron objects.

3.1.3. Neuron Object

Each neuron object contains the mathematical model of an individual neuron together with the current neuron state and other parameters. Every individual neuron can use one of the four proposed models: I&F, I&FB, I&FA or R&F. Spikes coming from neuron k to neuron i cross a synapse which produces gain G_{ik} and delay D_{ik}. Both static and dynamic synapses can be used. Detailed information about the neuron and synapse models is found in chapter 2.

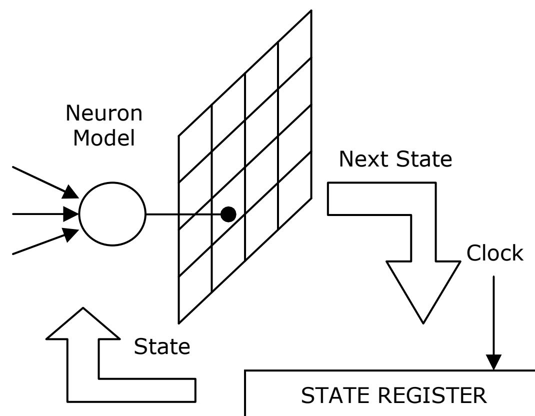


Fig.3.3. Updating the network state

The neuron object is depicted in figure 3.4. The "potential" variable holds the neuron potential that is computed according to the neuron model. The parameters and internal states of G_{ik} and D_{ik} are stored in the "synapseMatrix" and "delayMatrix" variables.

```
>> network.layer_array(2).neuron_array(1,1)

ans =

                out: 0
          potential: 0
        loss_factor: 0.1000
          threshold: 5
     synapse_matrix: [5x5x3 double]
       delay_matrix: [5x5x3 double]
    data_delay_line: 0
```

Fig.3.4. Neuron Object

Spikes propagate between neurons with different delays. This means that in order to compute the spike influence on current potentials a history of the spike activity of the neural network needs to be recorded. This history needs to be at least

BUPT

as long as the largest delay value. The simulation framework is organized such that each neuron keeps track of all the spikes that will affect its potential at some time in the future. This is done by placing a "dataDelayLine" vector in each neuron object. When a neuron object is simulated the framework solver computes all the post-synaptic spikes produced by all the synapses associated to this neuron. This is done by using the model of the synapse. The post-synaptic spikes are then placed into the dataDelayLine vector at a position given by the delay value of each synapse. At each simulation time step the delay line is shifted and the first entry is used for computing the new neuron potential.

The output of each neuron is Boolean and represents the spiking activity of the neuron. Every neuron output is mapped on the "state" variable of the layer object.

3.2. Model Functions

3.2.1. Simulation Functions

Simulating a network is done by calling function "simulateNetwork". The function has two input variables. The first input variable is the network object. The second variable is the simulation duration in seconds. The simulation is performed by calling repetitively subroutines like "advanceTime", updateNeuron" and "computeNextState". The function returns two objects as output. One output object is the post-simulation network. Therefore, a comparison between the internal state of the initial network object and the final network object can show the influence that the external stimuli has had on the network within the time span of the simulation. The second output object is an activity object. During the simulation, time traces of the neural outputs and membrane potentials are recorded. These traces are organized in multidimensional vectors that are stored inside the activity object. The activity object is very useful because it holds data that completely characterizes the behavior of the network during the simulation. An instance of such an object is shown in figure 3.5. The neural activity has the same topology as the network that generated it. Several other functions are offered for creating and initializing new objects and also for uploading stimuli and adapting synapses.

```
activity =

          topology: [10 10]
         nr_layers: 2
    time_instances: 100
       layer_state: {[]  [10x10x100 double]}
   layer_potential: {[]  [10x10x100 double]}
```

Fig.3.5. Network spiking activity object

3.2.2 Visualization Functions

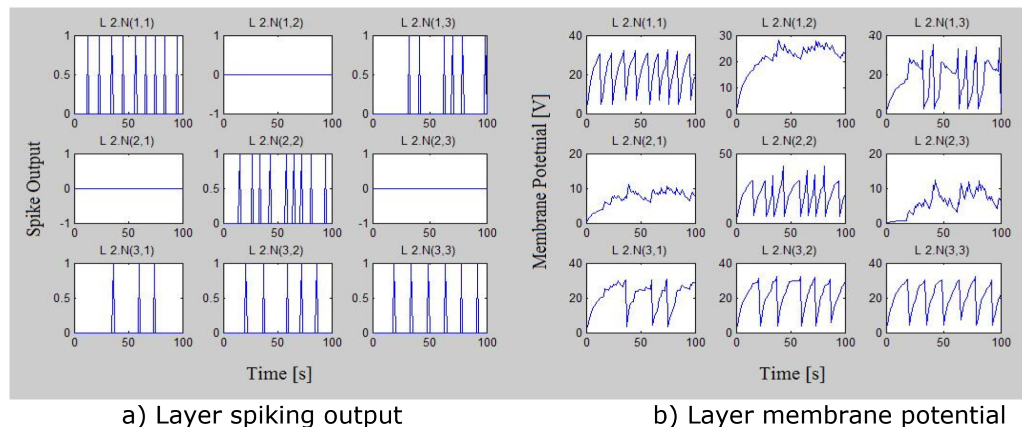
The activity object contains the time traces recorded from the membrane potentials and neural outputs. Most often, the easiest way to interpret this data is by visualization.

BUPT

3.2.2.1. Visualizing Neural Time Traces

The most straightforward way to visualize is by plotting the actual time traces. For this purpose function "displayActivity" can be used. The function accepts several variables which influence the display mode. The first variable for this function is the activity object that supplies the data. The second variable will specify the number of the neural layer that will be plotted. If this variable is omitted all layers will be plotted in several distinct windows. The third variable is a string that will select between displaying the neural spiking output activity or the membrane potentials. Figure 3.6 shows the output of this function for both cases. The fourth variable is optional and allows visualization of a sub-region of the neural array.

Visualization of the time traces is not very useful when it comes to the interpretation of the data because not very many neurons can be fitted clearly into a single window. However, the function described above can be very useful during the debugging period of a project. Subtle effects created by different network parameters can be spotted on the time traces and so several problems can be avoided or fixed.



a) Layer spiking output b) Layer membrane potential

Fig. 3.6. Visualization of spiking and potential traces

3.2.2.2. Visualizing Neural Spike Rates

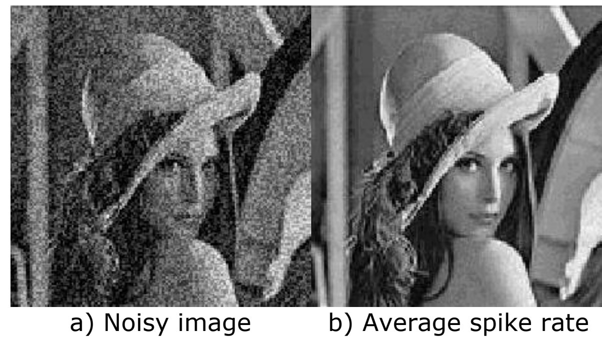
When the neural array is large visualizing by plotting time traces can be very difficult or even impossible. For this purpose the function "displayRate" was developed. This function computes the rate of the spike train for each neuron and then maps these rates onto a black-white image. This permits easy visualization of large arrays. The function's input variables allow selecting the time at which the average rate is computed and also the size of the averaging window. Figure 3.7 shows the output of this function. Figure 3.7a displays the image that is fed to the network as input stimuli. The image only presents a snapshot of the input stimuli which will continuously change during the simulation as an effect of the time-varying white noise. Figure 3.7b presents the rate-image of a spiking neural network that uses an I&F neuron model. The low-filtering effect of the neuron is seen in the fact that the noise is eliminated from the image.

3.2.2.3. Visualizing Neural Synchrony

Another important aspect in neural activity analysis is neural synchrony. For example, at image processing and shape recognition, neural synchrony can be used

BUPT

in the segmentation stage. Observations among biological systems have led to the idea that neurons processing pixels belonging to the same object tend to fire at the similar rates and also in synchrony.



a) Noisy image b) Average spike rate

Fig. 3.7. Visualization of the average spike rate as image

If neurons are stimulated by the pixels of an image they will fire spike trains that rate-code the pixel information. The neurons that are connected to pixels belonging to the same object will fire at the close rates. This is based on the assumption that the two pixels that are sourcing the neurons will have similar values. However, due to different initial conditions or system noise, these neurons will fall out of phase. Synchrony can still be achieved by using lateral connections in the proximity of each neuron. This way, spikes generated by one neuron can force neurons that are almost ready to fire to generate a spike ahead of time and thus inducing synchrony. Additional information on neural synchrony can be found in [43], [54] and [58]. For the purpose of visualizing neural synchrony function "displaySynchrony" was developed. The function expects three variables as input. The first one is the neural activity object that is supposed to be analyzed. The second one is the time value at which synchrony is evaluated. The third variable is a synchrony threshold.

The function uses a fraction variable to denote how well two neurons are synchronized with 0 meaning completely out of phase and 1 meaning fully synchronized. Assuming that the neurons are firing at the same rate full de-synchronization occurs when the time distance between spikes is half of the period. The function builds a map of synchrony between each neuron and its neighbors. Then, by comparing the synchrony levels with the synchrony threshold, decides whether the two neurons are sufficiently synchronous to be considered as belonging to the same object. This way segmentation is performed. Lastly, the function maps groups of synchronized neurons to different colors and plots the result. Figure 3.8a, 3.8b and 3.8c presents the output of this function at different times during the simulation.

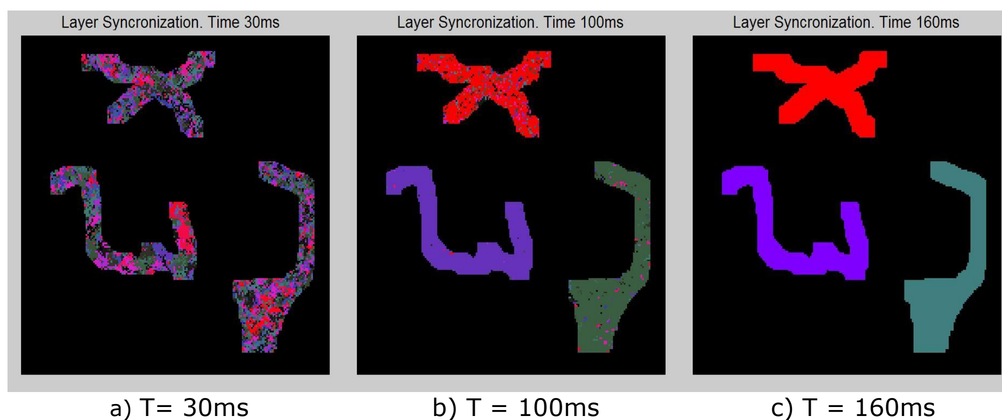
The activity object was obtained by simulating a network model similar to the one described above that was sourced with a grey scale image comprising of three objects, each at a different grey level. It can be observed that initially the neurons are unsynchronized and so the image is segmented into large number of objects. At 100ms large groups of neurons become synchronized. After 160ms all neurons of the same object are fully synchronized (above the synchrony threshold which in this case was 0.8).

BUPT

3.3. Parallelizing the Model

The major drawback for this approach is that the neural network is simulated by running the model of each neuron serially on the same processing unit. Because of this, the simulation time can become very large or even unacceptable in some cases. However, it is worth noticing that the model of the neural network is parallel and that the simulation of any neuron is fully independent of the results produced by the simulation of any other neuron during the same simulation time step. In order to simulate a neuron the following information needs to be known:

- The input stimuli
- The current network state
- The neuron's own internal state: membrane potential and content of the data delay line.



a) T= 30ms b) T = 100ms c) T = 160ms

Fig. 3.8. Visualizing Neural Synchrony

All this information is available at the beginning of every simulation time step and remains unchanged during the simulation time step. Consequently, the neuron can be simulated by any processing unit, in parallel with all the other neurons, provided that the above information is known at the beginning of each simulation time step. Therefore the simulation framework was broken apart and distributed in a computer network as shown in figure 3.9.

In the original framework, the neuron objects were stored locally inside the neuron arrays of the layer objects. Now, these arrays are extracted and distributed on several slave computers. Inside the layer objects, the neuron arrays are replaced with a neuron distribution map variable. This is necessary so the master computer can track the location of each neuron.

The master and slave computers communicate through files written on a shared hard drive space. At the beginning of each simulation time step the master computer writes a file that contains the current network state and the current input stimuli. The slave computers wait until the file is available, read it and start simulating the neurons. All the internal state variables of each neuron are stored by the neuron objects and therefore are available locally on the slave computers. When a slave computer finishes the simulation it writes a response file that contains the output of all neurons. The master computer reads the file and uses the neuron

BUPT

distribution map to build the next state of the neural network. When the next state is complete it replaces the current state and the simulation time is advanced to the next time step.

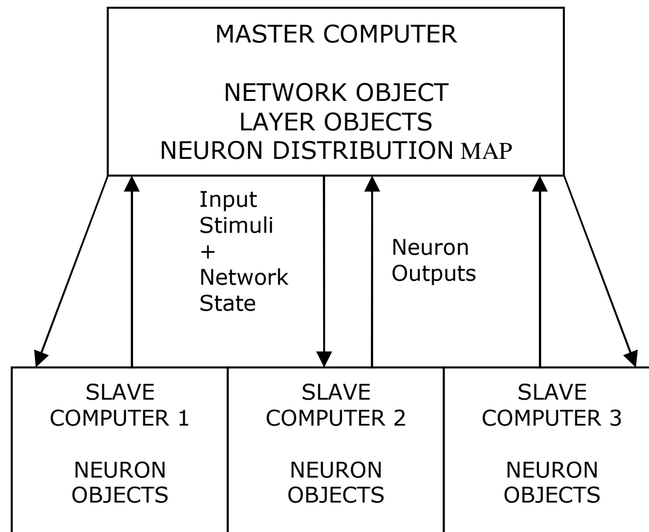


Fig. 3.9. Distributed Model on a network of Computers

3.3.1. Choosing number of slaves

By distributing the workload on several slaves which work in parallel the time required for simulating the models is reduced proportionally by the number of slaves. Therefore it is desired to have as many slaves as possible. However, some time is lost with communication. When more slaves are used, more communication time is needed. This paragraph studies what is the optimum number of slaves given that simulation time and communication time are known. Figure 3.10 presents all the stages that appear during one simulation time step. The master computer starts by sending the data to the slave computers. This operation takes time T_{comm11}. As soon as the data is sent, the slave computer starts to simulate (which takes time T_{S1}) and the master sends the data to the second slave (which takes time T_{comm21}). It is assumed that the master computer sends the data to all slave computers before the first slave finishes simulation. This is true if $T_1<T_2$. Otherwise, the master computer will not be ready to collect the data from the first slave as soon as it will be ready.

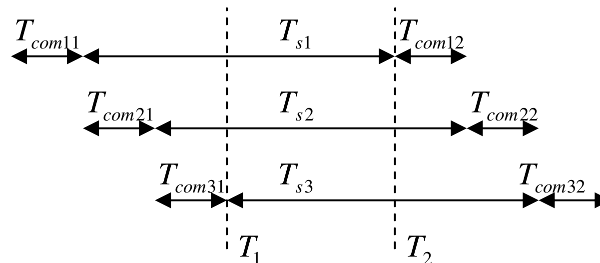


Fig. 3.10. Master-Slave Interaction

BUPT

Let T_{NP} be the time required for simulating the entire network model in a non-parallel approach. Considering that the slave computers are identical it can be accepted that the simulating times T_S are roughly the same and are equal to T_{NP}/N, where N is the number of slaves. Therefore, from figure 3.10 the time required for simulating one time step in a parallel approach T_P can be computed with (3.1).

$$T_p = N * T_{com} + \frac{T_{NP}}{N} + T_{com} = (N+1) * T_{com} + \frac{T_{NP}}{N} \quad (3.1)$$

$$\frac{T_p}{T_{NP}} = (N+1) * \frac{T_{com}}{T_{NP}} + \frac{1}{N} \quad (3.2)$$

In order to find the optimal number of slaves it is required to find the minimum for the ratio in (3.2). This is done in (3.3).

$$\frac{\partial}{\partial N}\left(\frac{T_P}{T_{NP}}\right) = \frac{T_{com}}{T_{NP}} - \frac{1}{N^2} = 0 \Rightarrow N = \sqrt{\frac{T_{NP}}{T_{com}}} \quad (3.3)$$

Equation (3.3) shows that if communication time is small, a high number of slaves are desired. Ideally, if communication time is zero, the more slaves are available the better. In a real situation any number of slaves higher than the one in (3.3) would increase simulation time. This happens because the communication time added by an additional slave is always constant while the advantage gained by distributing some of the workload to the additional slave is diminishing.

3.3.2. Results

The previous paragraph computes an optimal number of slave computers given some generic simulation time and communication time. In practice, the simulation time T_{NP} depends on the processing speed (CPU throughput) of the slave and master computer while the communication time depends on network speed and hard disk access speed. Some quantitative lab experimentation was performed on the following equipment:

- INTEL CORE2 QUAD Q6600 2x2.4Ghz
- RAM 4GB
- WINDOWS VISTA 32BIT SP2
- MATLAB 7.7.0 (R2008b)
- NET CARD INTEL 82566DM GIGABIT

First, several non-parallel simulations were performed in order to evaluate simulation time T_{NP}. Neural networks of different sizes were tested. The results are shown in Fig. 3.11 and numerically available in table (3.1). Due to the "multi-tasking" nature of the system, measurement during a single simulation is unreliable. Therefore, for each size of the neural network 100 simulations were performed and the simulation time was estimated to be the median value of those measurements.

It is seen that the simulation time does not increase linearly with the number of neurons. This happens because simulation time depends on the number

BUPT

of neurons and also on the time required for simulating each neuron. In the case of large neural networks each neuron receives signals from more sources and so the simulation time of each neuron also depends on the number of neurons. Consequently, the simulation time depends on the number of neurons in a quadratic manner. It is expected that the parallelization technique is more efficient for large neural networks, because in this case the serial simulation time increases drastically.

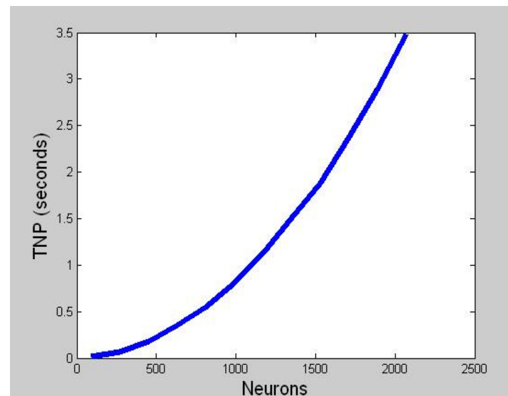


Fig. 3.11 Non-Linear simulation time as a function of the number of neurons

Table (3.1) also contains the size of the data packet. Please note that a larger network means a larger file due to the larger network state variable. However, a larger file does not mean a higher communication time. This is due to communication overhead which dominates the transmission (files are very small) and also due to the un-repeatability of the "multi-tasking" system.

Table 3.1. Simulation Results and optimal number of slaves

Number of neurons	T_{NP} (seconds)	Communication file size (BYTES)	Optimal Slave Nr. N
90	0.01	336	0
270	0.07	350	0
450	0.18	361	0
630	0.34	366	2
810	0.55	372	2
990	0.80	378	3
1170	1.13	426	3
1350	1.51	436	4
1530	1.88	443	4
1710	2.37	450	4
1890	2.90	454	5
2070	3.49	493	5

BUPT

It sometimes happens that large files require a smaller communication time than small files. Therefore in order to estimate communication time several measurements were performed. It resulted that T_{comm} is rather constant in the range 0.11 to 0.14 seconds. The median value of 0.12 seconds was chosen. Table (3.1) also shows the optimal number of slaves computed with (3.3). When the optimal number is zero it means that non-parallel computing is faster. This happens for small neural networks where simulation time is small and adding a single communication time would increase duration of the simulation. In order to quantify the improvement we choose the best case situation (2070 neurons).

If N=5 is substituted in (3.2) it yields T_P / T_{NP}= 0.4. This means that the parallel simulation is 2.5 time faster than the non-parallel simulation. Ideally, if there was no communication, the parallel simulation should be 5 times faster since there is 5 times more processing power. This shows that much more improvement can be achieved if communication between computing units is reduced. A possible option is to use a grid superscalar computer. In this case computing units communicate by shared RAM memory and so communication time can become insignificant compared to simulation time. A cheaper and more accessible method is to use a graphics processing unit GPU as a parallel general purpose processor. This hardware acceleration technique is presented in chapter 4. Further details of the parallel MATLAB implementation can be found in [56].

BUPT

4. GPU ACCELERATED MODEL FOR SPIKING NEURAL NETWORKS

The previous chapter presents a MATLAB framework useful for simulating spiking neural networks. It also introduces a set of functions useful for analyzing and displaying results. The major drawback is simulation speed, which can become a bottleneck when dealing with large neural networks or when trying to perform iterative simulation. Iterative simulation is a common situation especially when trying to perform an optimization of the network parameters. Therefore improving simulation speed is a critical issue. Chapter three proposes a possible solution to this problem by parallelizing the model and distributing it among a network of computers. This approach produces some improvement. However, due to the high communication time, the performance gain is small compared to the amount of additional computing power. This chapter presents an alternative solution by implementing the spiking neural network on a GPU.

4.1. General Purpose GPU Computing

Large scale simulations are often the task of massively parallel cluster computers. For a long time scientists dealing with complex modeling had to use such expensive machines in order to obtain simulation results in reasonable amounts of time. This made large scale parallel computing inaccessible and uncommon. Recently, CPU producers have started to incorporate several cores in the same chip growing the amount of computational parallelism. However, it is still not sufficient for some demanding tasks in modeling and simulation.

Initially, the GPU (Graphics Processing Unit) was introduced as a dedicated processing unit that dealt with image processing and display related tasks. Because its target applications differed in nature from the CPUs, the GPU evolved toward a different type of architecture. It mostly deals with image processing and 3D graphics rendering which are parallel applications (very often each pixel or vertex can be processed individually). Therefore, the GPU architecture has several simple processing units rather than a single complex core. Figure 4.1 shows the processing throughput of a GPU versus a CPU.

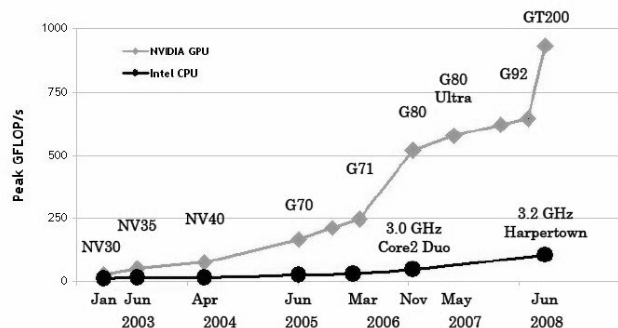


Fig.4.1. CPU vs. GPU evolution

The amazing throughput of the GPU is only available when the application is parallelizable. Otherwise it will only use a single GPU processing unit which is most likely to be outperformed by a CPU. The ability of a GPU to process in parallel drew attention to researchers interested in parallel simulations. The main attraction is the low price and availability of such a device.

4.1.1 Early GPU Computing

The usual GPU processing pipeline is presented in figure 4.2. This type of architecture is implemented under OpenGL[1] or DirectX[2] and is accessible via the regular display driver programs. The programmer is allowed to upload code fragments inside the VERTEX and PIXEL blocks. These fragments of code are called "shaders" and will be used at the processing of individual vertices or pixels. At the time when programmers observed the opportunity of a GPU to perform general purpose computing rather that graphics dedicated computing the hardware and software did not offer any explicit support for this matter. Therefore, in order to perform non-graphic computations the programmer had to "fool" the hardware and software by wrapping the application as if it were a graphics application. For example, the application's input data had to be organized as if it were graphics data: vertex coordinates, vertex colors, texture maps and many others. The application's processing code would be uploaded as vertex or pixel shaders even if the processing is not graphics related. In the end the frame buffer will contain processing results rather than images. This approach is successful but is indirect and requires 3D graphics knowledge. Therefore, even if the GPUs were extremely powerful and had the parallel computing capability, researchers were reserved towards using it as a general purpose computing machine.

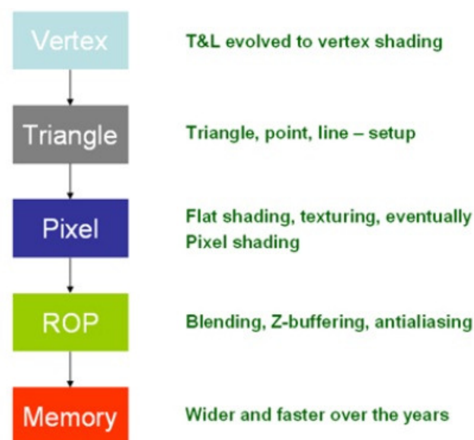


Fig.4.2. OpenGL graphics processing pipeline.

[1] OpenGL (Open Graphics Library) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics.
[2] Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms.

BUPT

4.1.2. NVIDIA's CUDA[3] Architecture

In late 2006 NVIDIA introduced a new computing architecture for its GPUs called CUDA [59]. The architecture is oriented towards general purpose computing (GPGPU[4]) and offers hardware and software support such that it can be used easily under Visual Studio C++ with only a small set of extensions to the basic programming language. The new architecture, driver and API allow the programmer to view the GPU as a parallel computing machine rather than a graphics processing pipeline [62], [63].

4.1.3. Simple CUDA Example

A simple example is presented in figure 4.3 for the purpose of describing the CUDA functionality [59].

```
int main{
        int vectorHost[N];                          //declares and allocates host vector
        InitalizeVector(vectorHost);                //initializes host vector
        int *vectorDevice;                          //declares pointer to device vector

        cudaMalloc(&vectorDevice, N*sizeof(int));   //allocates device vector

        cudaMemcpy(vectorDevice,  vectorHost,  N*sizeof(int),  cudaMemcpyHostToDevice);
                                                    //copies data from host to device
        VectorSquare<<<1, N>>>(vectorDevice);       //calls kernel

        cudaMemcpy(vectorHost,  vectorDevice,  N*sizeof(int),  cudaMemcpyDeviceToHost);
                                                    //copies data from device to host
        cudafree(vectorDevice);                     //frees device memory
}

__global__ void VectorSquare(int *vectorDevice){
        int index = threadIdx.x;                    //determine index inside vector
        vectorDevice[index] = vectorDevice[index]*vectorDevice[index];
}
```

Fig.4.3. Simple CUDA parallel program

As a general rule, all variables that have names terminating with "host" are located in the memory of the host PC, while variables terminating with "device" are

[3] CUDA or Compute Unified Device Architecture is a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages like C, C++ or FORTRAN. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs become as accessible for computation as CPUs.

[4] General-purpose computing on graphics processing units (GPGPU, also referred to as GPGP and less often GP^2U) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. It is made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines, which allows programmers to use stream processing on non-graphics data

BUPT

located in the memory of the GPU. Consequently, host variables are processed by the CPU and device variables are processed by the GPU.

The example in figure 4.3 has to square the elements of a vector without using a "for" loop. The first three statements declare, allocate and initialize the host vector (vectorHost) and also declare a pointer to the device vector (vectorDevice). The fourth statement allocates memory on the device and stores the reference to this memory space inside the pointer. It is very important to notice that dereferencing this pointer inside the CPU code leads to a run-time error or a false result. The pointer cannot be de-referenced because the CPU treats it as if it pointed to its own memory space and not the GPUs. Its purpose is to store a memory address that can be used later only by CUDA specific instructions (transfer functions and kernel launches).

The fifth statement transfers data from the host to the device. The function needs the pointers to the two memory spaces and also the size of the data block in bytes. The sixth statement is the "kernel" launch and represents the key to the CUDA optimized solution. A kernel is a piece of code that can be launched in parallel on multiple computing units inside the GPU. Each instance of the kernel code is called "thread". The number of launched threads is specified inside the brackets "<<>>"; in this case *N*. This is the same as the vector size meaning that each thread will operate on one entry inside the vector. The parameter to the function is the starting address to memory space where the vector has been stored on the device. Any function that is to be used as a kernel needs to have the "__global__" identifier and must respect the CUDA programming restrictions.

Figure 4.4 presents the launch of the kernel. Basically, the same code will be running inside every thread.

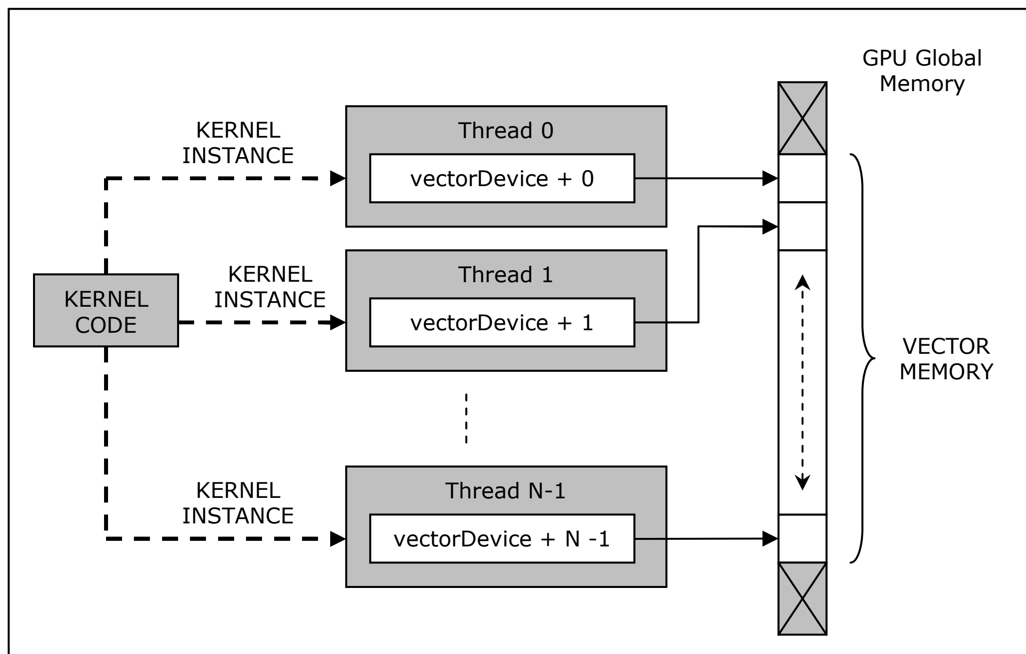


Fig.4.4. Kernel Launch

BUPT

However, CUDA offers hardware and software mechanisms that allows identification of the thread from inside the kernel code. This way, branches and accessing by reference allow different threads to have independent execution paths and operate on different memory addresses. This is achieved by the first line of the VectorSquare kernel in figure 4.3. Even though variable *threadIdx.x* is never declared it will be available at run-time and will differ in value from one thread to another, unveiling the identity of the thread (0 to N-1). The thread identity initializes an index that is used at referencing the vector memory. This way every thread will operate on a different entry inside the vector. The second statement of the kernel squares the element. The last two statements of the main function copy the results back to the host and free the device memory.

4.2. Spiking Neural Network CUDA Model

Implementing the spiking neural network model on the GPU can produce significant speedup by simulating the model in parallel. However, in order for this to be possible the model itself needs to be parallelizable. In chapter three we have proven the all neurons of a SNN can be simulated in parallel. However, the attempt to parallelize the model by distributing it on a network of computers was not satisfactory because of the significant communication times between computational units. In order to simulate the SNN model on the GPU, the MATLAB model had to be redesigned such that it suits the GPU architecture. In addition, the code had to be re-written in CUDA C. Figure 4.5 shows how the MATLAB code and the CUDA C code communicate by using the MEX interface. MEX files allow calling a pre-compiled C files as they were in-built MATLAB functions. It is important to be able to call the GPU simulator from MATLAB because this way we can reuse the design, analyze and display functions that were already written for the MATLAB simulator.

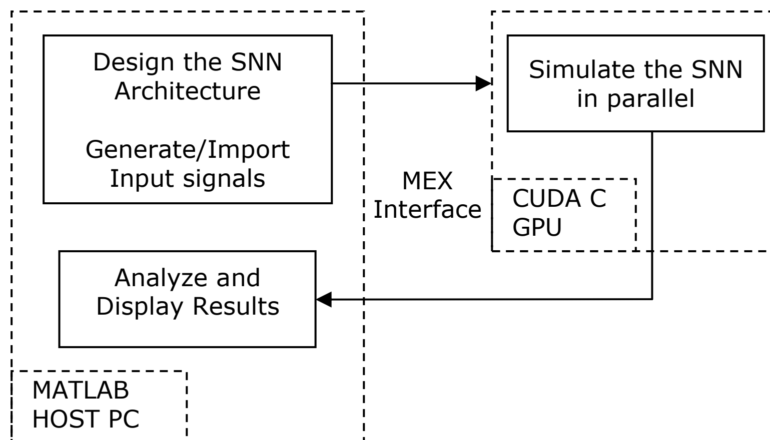


Fig. 4.5. MATLAB-CUDA C interfacing

4.2.1. Model Architecture

Our proposed GPU model is object oriented and contains 6 classes as follows: **SpikingNeuralNetwork**, **Neuron**, **DelayLine**, **Synapse**, **InputSource**, and **ActivityRecorder** (classes and objects are printed in bold). Each class contains

BUPT

all necessary host and device (GPU) methods. Host methods serve in: create/initialize the model; send/retrieve the model to/from the device; save results. Device methods perform the simulation. Figure 4.6 presents the CUDA C model. The arrows show the flow of information inside the model. The simulator sends input signals to the **InputSource** object, design parameters to the **SpikingNeuralNetwork** object and reads results from the **ActivityRecorder** object. The simulator also creates control variables and synchronizes the simulation. The dotted line marks the fact that the arrays of **Neuron** objects and **Synapse** objects are internal components of the **SpikingNeuralNetwork** object. The **DelayLine** object is responsible for keeping track of spikes while they propagate (with delay) between neurons. Consequently every **Neuron** object has a **DelayLine** pair-object.

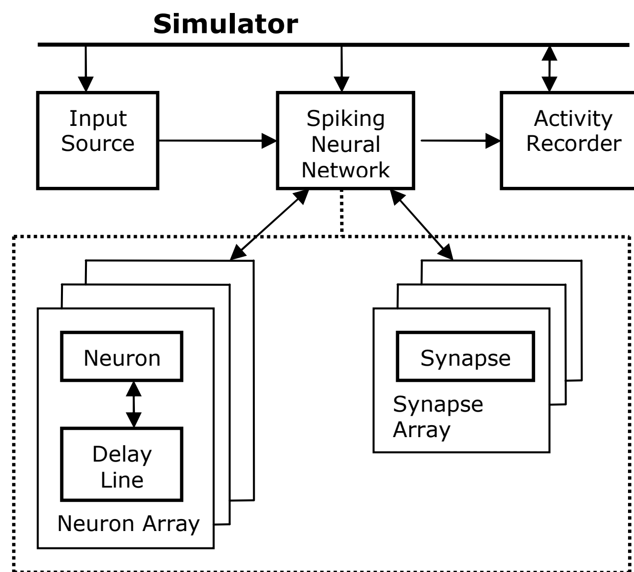


Fig.4.6. Spiking Neural Network Architecture

4.2.2 Delay Line Implementation

Paragraph 4.2.3 will reveal that the **DelayLine** object is the one that performs the most memory accesses during simulation. Therefore, for the purpose of accelerating the simulation, it is important to design the object carefully such that the number of memory accesses is minimized. Without going into the details of how the simulation works (presented in paragraph 4.2.3), it can be summarized that the **DelayLine** object performs three types of actions:

- inserts new spikes (when a neuron receives a new spike from a synapse the spike is inserted in the delay line)
- searches for active spikes (spikes that have time stamps equal to zero are called "active spike" in the sense that these spikes will contribute to the membrane potential of the neuron during the current simulation time step)
- updates the delay line (when the simulation time advances, the time stamp of all spikes needs to be modified; also, all active spikes are removed)

Two implementations for the **DelayLine** are proposed. Both implementations use two buffers of data: one for the spike amplitudes and one for the spike time stamps. Buffer entries that have the same index store information about the amplitude and time of the same spike. In the first implementation of the **DelayLine**, new spikes are stored at the end of the buffer. This way inserting a new spike is done easily because the position of the spike inside the buffer is pre-determined. This implies a reduced number of memory accesses. In the second implementation spikes are always stored in an ordered fashion, such that successive entries in the buffers store spikes with increasing delay times. Figure 4.7 shows the functionality of the **DelayLine** object for this case.

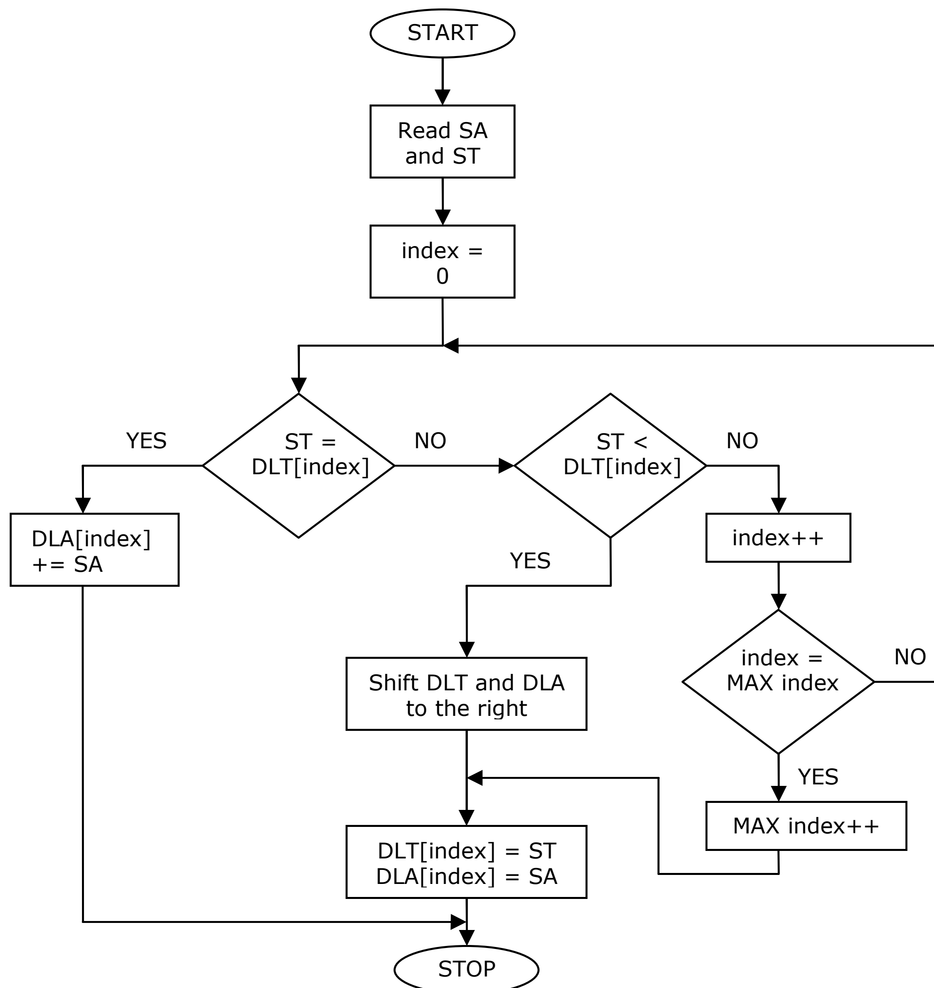


Fig.4.7. Delay Line Functionality

The abbreviations in the figure are as follows: SA and ST are spike amplitude and spike time stamp. DLA and DLT are buffers of data representing the delay line contents (amplitude and time). Pair entries to DLA and DLT store

BUPT

amplitude and time information of previously received spikes. When a new spike is received, its time ST is compared to the previously stored data in DLT. If the time of the new spike matches any of the entries in DLT then the same entry in DLA accumulates the amplitude of the new spike SA. Because the two spikes arrive at the neuron at the same time, there is no reason to store them separately since their effect on the neuron's membrane potential is cumulative. If the time of the new spike ST does not match any of the entries in DLT, a new entry needs to be allocated in DLT and DLA to store the new spike. Spikes are always stored in an ordered fashion, such that successive entries in DLT and DLA store spikes with increasing delay times. This is useful because the simulator only needs to check the first entry in DLT when computing the new value for the neuron membrane potential. Needless to say this approach requires significantly more memory accesses for inserting a spike. However, because the delay line is always ordered, it offers an advantage when searching for active spikes or updating the delay line.

In order to determine which of the two implementations is better we tried to quantify the average number of memory accesses for each of the two cases. This cannot be done by examining the compiled code alone, because the execution path is determined by the exact neural network architecture and value of the input signal. In order to estimate the average number of memory accesses for each of the two implementations 1000 simulations were performed for each implementation of the delay line by combining 20 neural networks and 50 versions for the input signals. A set of counters was used to determine the number of memory accesses for each simulation. Table 4.1 presents the average number of memory accesses for the three actions for each of the two implementations.

Table 4.1. Number of memory accesses

Action	Symbol	Implementation 1	Implementation 2
Insert new spike	IS	4 accesses	15 accesses
Search for active spike	SAS	14 accesses	2 accesses
Update delay line	UDL	119 accesses	22 accesses

In order to quantify the overall number of memory accesses for each of the three actions note that IS is performed for each spike propagating inside the neural network while SAS and UDL are performed for each neuron.

$$MEM_1 = 4 * N_S + (14 + 119) * N_N \qquad N_S = \text{nr of propagating spikes} \qquad (4.1)$$

$$MEM_2 = 15 * N_S + (2 + 22) * N_N \qquad N_N = \text{nr of neurons} \qquad (4.2)$$

The two implementations of the **DelayLine** have the same number of memory accesses if:

$$11 * N_S = 109 * N_N \Rightarrow \frac{N_S}{N_N} = 9.90 \qquad (4.3)$$

The average number of spikes that are propagating inside the neural network N_S is equal to the average number of neurons that are firing a spike multiplied by the average number of synapses branching from each neuron.

BUPT

$$\frac{N_{fire} * N_{syn}}{N_N} = \frac{N_N * K_{fire} * N_{syn}}{N_N} \Rightarrow K_{fire} * N_{syn} = 9.90 \qquad (4.4)$$

K_{fire} is the average fraction of neurons that are firing a spike during a single simulation time step. If for example 30% of all neurons are currently firing (which is a realistic situation), the average number of synapses branching from each neuron N_{syn} needs to be 33 in order for the two **DelayLine** implementations to have the same number of memory accesses. Chapter 5 presents introduces a computational architecture called Liquid State Machine that is based on a spiking neural network. In the case of the Liquid State Machine the average number of synapses branching from each neuron is N_{syn} is below 10 in most cases. Therefore, for the purpose of this project the second implementation (presented in figure 4.7) for the **DelayLine** is considered to be more efficient.

4.2.3. Moving Objects between Host Computer and Device GPU

The model is created and initialized by the host program. In order for the simulation to run on the device GPU a copy of the model needs to be transferred to the device. After the simulation is done the device model needs to be transferred back to the host in order to update the state of the host model and to retrieve the simulation results. Transferring the model between the host and device raises a problem which is discussed and solved by the next section.

CUDA C offers a single function for transferring data between the host and device (cudaMemcpy). The function transfers a block of data of given size between two specified memory addresses each belonging to the host and to the device respectively. In many situations, especially when memory is allocated dynamically, complex data structures (objects) are not a continuous block of data. The internal components of an object could be stored inside disjointed blocks of memory and be linked by pointers. In this case using cudaMemcpy directly on the object could lead to an incomplete transfer and also to false references inside the device memory. Figure 4.8 presents such a situation. This is the outcome of the following two lines:

```
cudaMalloc (&objectDevice, sizeof (objectHost));
cudaMemcpy (objectHost, objectDevice, sizeof (objectHost), cudaMemcpyDeviceToHost);
```

One problem is that the contents of the array are not transferred. The second problem is that the pointer to the array on the device contains the memory address of the host array. If the device code tries to dereference the pointer it will cause a runtime error.

In order to transfer all the internal data of the object and to keep the correct internal references the steps enumerated below need to be performed. Note the operations marked with *. Our design is build such that this restriction is not violated. However, there are situations when external factors (i.e. exceptions) can violate the restriction. Future improvements to the design will eliminate the problem.

- Allocate device memory for the main object
- Allocate device memory for all internal arrays
- Transfer internal arrays with cudaMemcpy
- Save original values for all host pointers

BUPT

- Replace host pointers with device pointers (Host object cannot be used because it temporarily contains false memory references) *
- Transfer main object with cudaMemcpy
- Restore values for all host pointers (Host object is ready to be used again)

The above steps assure that both the host and device objects will operate correctly inside the host and device code respectively. Similarly, the object transfer from device to host is performed as follows:

- Transfer internal arrays with cudaMemcpy
- Save original values for all host pointers
- Transfer main object with cudaMemcpy (Host Object pointers are falsely overwritten. Host object cannot be used until pointes are restored)*
- Restore values for all host pointers (Host object is ready to be used again).

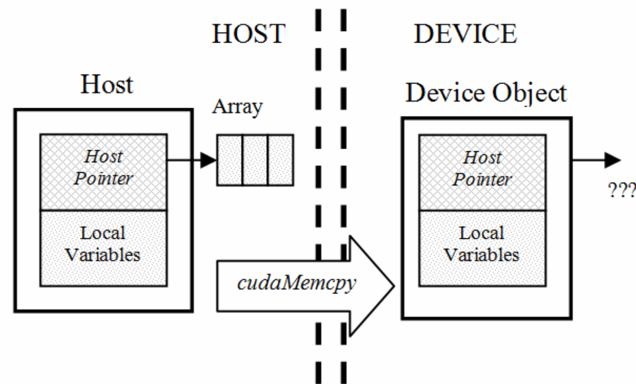


Fig.4.8. Incomplete Object transfer

The above paragraph describes how to transfer an object that contains an internal array. However, when a project uses a complex hierarchy of classes, where objects include sub-objects, the above steps cannot be applied directly. Therefore, it is desired to create a standardized mechanism that allows transfer of objects from host to device regardless of the number of class layers and the arrangement of child classes within parent classes. We propose the introduction of the **BasicObject** class for this purpose.

The **BasicObject** class contains all the necessary methods, memory address maps and control variables that facilitate the correct transfer of the object itself and all its internal sub-objects. All classes used in a CUDA project inherit the **BasicObject** class hence benefit from all the transfer methods. However, in order for the transfer methods to work, the new class needs a suitable constructor that correctly initializes the memory address maps and the control variables. Figure 4.9 presents the concept described above. It can be seen that the **BasicObject** class does not contain the memory maps and control variables explicitly. Instead, it contains a pointer to an external structure that contains them. This is useful because it avoids transferring this information to the device (where it is not needed; both transfers host-device/device-host are performed by the host) and so it saves device memory space. The **BasicObject** class is also described in [64].

BUPT

4.2.4. Simulating the Model

All actions are initiated by the host CPU. Figure 4.10 presents the flow of actions during the simulation of the Spiking Neural Network. For a clearer representation the figure uses the following abbreviations (**SpikingNeuralNetwork** = **SNN**, **InputSource** = **IS**, **ActivityRecorder** = **AR**). The host starts by creating / initializing the necessary objects and also by allocating the necessary device memory. Next, the objects are transferred from host to device by using the transfer methods inherited from the **BasicObject** class. The simulation starts with a sequence of kernel launches initiated by the host.

The host continues by launching the "*Propagate Synapse*" kernel. This kernel operates on the **SpikingNeuralNetwork** object. It has the task to read the synapse information and propagate (or not) a spike between a source **Neuron** and a target **Neuron** object. The **Synapse** object amplifies the spike with the specified gain and places the spike in the **DelayLine** of the **Neuron** object at a position given by the delay value.

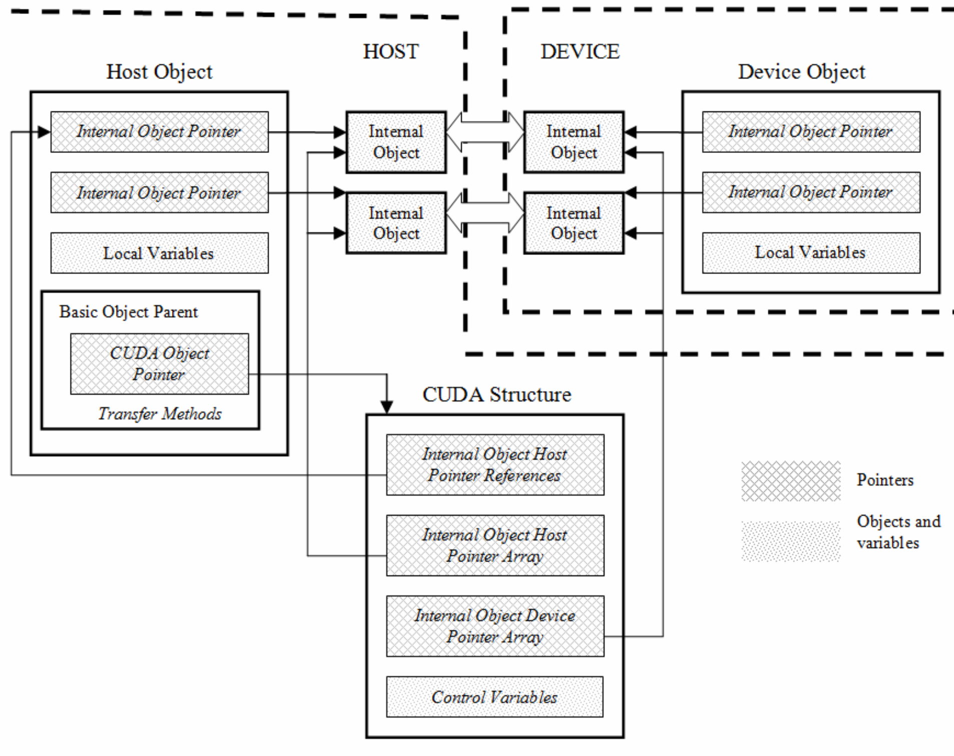


Fig.4.9. BasicObject Class

One possible problem with this approach is when two or more synapses targeting the same neuron try to execute propagation at the same time. Adding spikes to the delay line involves addition that needs to be performed atomically. One possible solution is to perform an atomic lock on the **DelayLine** object memory space before calling the propagation method. This way the hardware serializes the

BUPT

conflicting threads. Instead of using atomic operations which are slow, the thread conflict was solved by pre-processing the model on the host and by arranging it such that a thread conflict never occurs. The pre-processing to the model reads the array of synapses and re-arranges them such that adjacent synapses in the array can form large groups where the targeted neuron is not repeated. This way the host can launch "*Synapse Propagation*" kernels on one group at a time avoiding thread conflicts and thus performing software serialization between groups. Obviously this leads to performance decrease. On the other hand, if the problem had been solved by using atomic hardware locks the decrease in performance would have also been present. This is because even though the threads are launched at the same time the hardware serializes execution to assure atomicity.

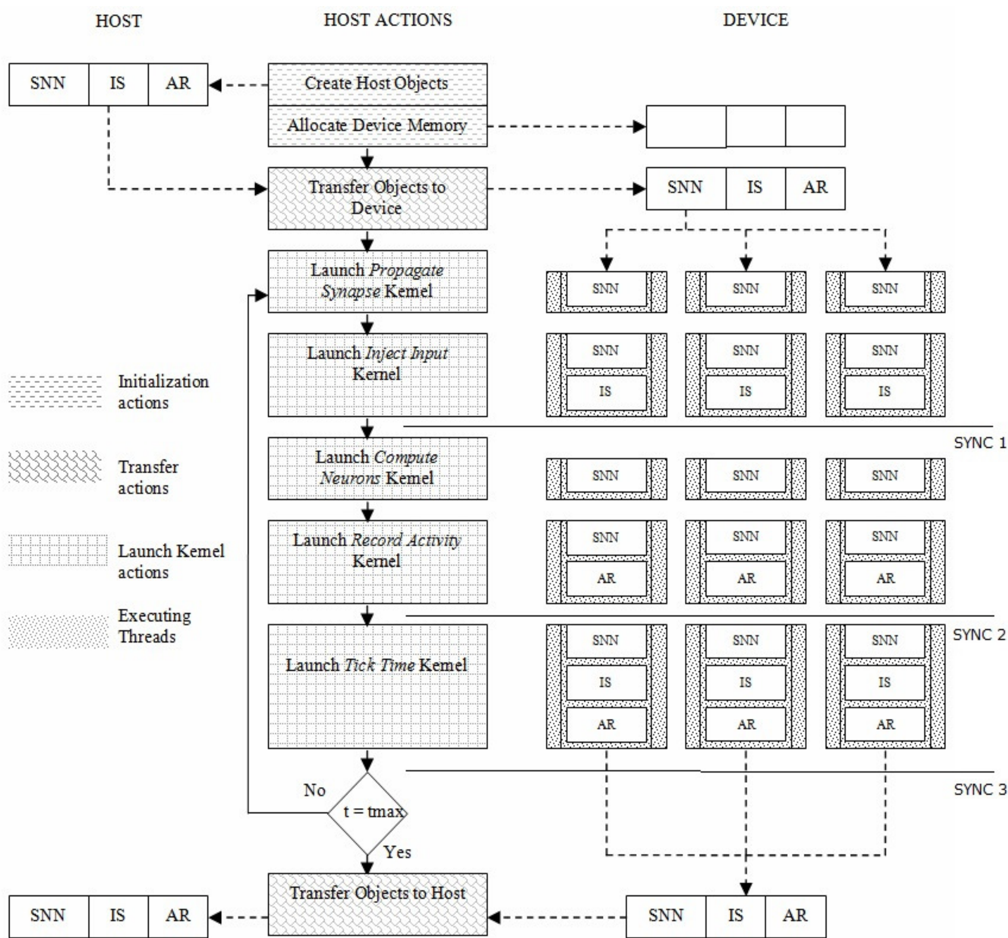


Fig.4.10. Spiking Neural Network Simulation Flow

Next, the host launches the "*Inject Input*" kernel. This reads input signals from the **InputSource** object and injects them as membrane potential at the appropriate locations inside the **SpikingNeuralNetwork** object. The number of threads is equal to the number of locations where input signals need to be injected.

BUPT

This is specified by the **InputSource** object. The "*Compute Neurons*" kernel computes the new membrane potential and output of all neurons. The number of threads is equal to the number of neurons. The "*Record Activity*" kernel reads the current spiking state (the output of all neurons) of the **SpikingNeuralNetwork** object and stores it as network spiking activity inside the **ActivityRecorder** object. The "*Tick*" kernel advances the simulation time. All the above kernel launches are performed iteratively for all simulation time steps.

4.2.5. Simulation Results

Before comparing simulation times of the MATLAB and CUDA models it is imperative to assure that the output of the two models is the same. This comparison is done independently for each simulation time step. Two output vectors are built containing the neuron membrane potentials of the two models. The error between the outputs is considered to be the norm of the difference vector. The maximum error ever found is $3.46*10^{-6}$. This small error is present because floating point operations are not associative due to rounding of intermediate results. This means that the order in which operations are performed matters. Obviously, this order cannot be guaranteed in a parallel thread-based system. Nevertheless, the output of the serial MATLAB model also depends on the order in which the network objects are picked for execution. Changing this order would also result in an error of the same magnitude. Anyway, in most applications where the system needs to be robust and insensitive to noise of significantly higher magnitude this error can be considered negligible.

Table 4.2 presents the simulation times for various network sizes. Shown times are an average from 100 simulations performed on different networks generated with the same parameters.

Table 4.2. Achieved Speedups

SNN Size	MATLAB Model Simulation Time Average (s)	MATLAB Model Simulation Time Relative Std (%)	CUDA Model Simulation Time Average (s)	CUDA Model Simulation Time Relative Std (%)	Speed-up
5x5x5	0.42	0.43	0.05	9.63	x8.3
6x6x6	1.18	0.24	0.09	8.45	x13.6
7x7x7	2.87	0.15	0.14	7.61	x21.0
8x8x8	6.29	0.18	0.21	6.29	x30.9
9x9x9	12.58	0.18	0.29	6.16	x42.4
10x10x10	23.47	0.18	0.43	7.91	x54.9
11x11x11	41.34	0.16	0.57	5.48	x72.9
12x12x12	69.48	0.19	0.75	6.21	x92.1
13x13x13	111.91	0.13	0.94	2.61	x119.5
14x14x14	173.71	0.18	1.15	2.51	x150.6

BUPT

It is worth noticing that the relative standard deviation is a lot larger in the case of the CUDA model (max 9.63%). The MATLAB model has a very small standard deviation (max. 0.43%). This is because the efficiency of the simulator is constant (one neuron at a time) regardless of the particular network synaptic connectivity. Still, a small deviation is present due to the multitasking nature of the operating system. On the other hand the CUDA simulator is always trying to exploit parallelism as well as possible. The efficiency of the simulator varies and depends on the exact network synaptic connectivity and amount of available parallelism. Therefore, the simulation time will vary significantly from one network to another. The standard deviation tends to decrease for large networks (down to 2.51%). This is because the synaptic diversity averages out for larger networks.

The speedups achieved by the CUDA model are great for all network sizes. However, it is the larger networks that allow the GPU to make use of its stunning computational power. When the network is large, the groups of synapses that can be processed in parallel are also large (see section 4.2.4) keeping all execution units busy. On the other hand, if the network is small, the groups of synapses that can be processed in parallel are small, leaving some of the execution units idle during the simulation of one group.

4.3. Improved CUDA Model

Explaining further improvements requires a deeper understanding of the GPU hardware [60], [61]. Figure 4.11 presents the GPU architecture. The GPU contains several Stream Multiprocessors (SMs). Each SM has eight Stream Processors cores (SPs)[5], one multi-threaded instruction unit and on-chip shared memory.

4.3.1. Minimizing the number of branches

Because there is only one instruction unit, each multiprocessor operates similarly to a SIMD[6] architecture. This means that all processors inside the same multiprocessor execute the same instruction on different data. Nevertheless, paragraph 4.1.3 states that each thread can branch independently based on conditions generated by its thread index. This property is particularly important for

---

[5] Stream processing is a computer programming paradigm, related to SIMD (single instruction, multiple data), that allows some applications to more easily exploit a limited form of parallel processing. Such applications can use multiple computational units, such as the FPUs on a GPU or FPGAs, without explicitly managing allocation, synchronization, or communication among those units. The stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed. Given a set of data (a stream), a series of operations (kernel functions) are applied to each element in the stream. Uniform streaming, where one kernel function is applied to all elements in the stream, is typical.

[6] SIMD is a type of multiprocessor architecture in which there is a single instruction cycle, but multiple sets of operands may be fetched to multiple processing units and may be operated upon simultaneously within a single instruction cycle. Acronym for single-instruction-stream, multiple-data-stream.

BUPT

this project when implementing the logic of the neuron **DelayLine**. In order for this to be possible the CUDA architecture deviates slightly from the SIMD architecture. Let's assume that a group of threads is running on the same multiprocessor. Whenever the code reaches a branch, the group is divided into two sub-groups each following a different execution path. Because the instruction unit can only dispatch a single instruction at a time, the two execution paths are serialized and so one sub-group will be pending while the other is executing. When the two execution paths have completed, the threads converge back to the same execution path.

This type of architecture allows the user to program freely without having restrictions regarding the control flow of a group of threads. However, when the code has a lot of diverging branches, the hardware serializes execution and resources are not used efficiently. With this thought in mind the application was programmed such that it avoids using any unnecessary branches.

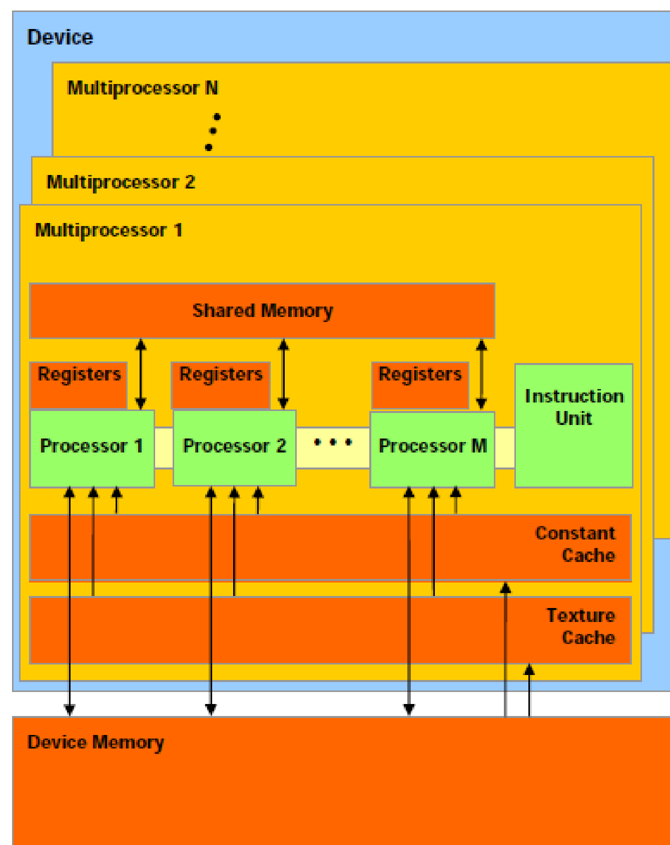


Fig.4.11. GPU architecture

4.3.2. Merging Kernels

Threads can be grouped into blocks with up to 512 threads per block and up to 64K blocks for each kernel launch. Having two ways of organizing parallelism (blocks and threads) creates control over the way the workload is distributed inside the GPU. When a kernel is launched it is assured that all threads in the same block

BUPT

will be executed inside the same SM. This allows the threads in the same block to: communicate (fast), synchronize and quickly switch register context when toggling the active thread.

Figure 4.12a presents a simplified version of the simulation flow shown in figure 4.10. The "*Propagate Synapse*" and "*Inject Input*" kernels are drawn is parallel. This symbolizes that the two kernels can be launched in any order. The three synchronization barriers are important because they assure that the different parts of the neural network remain synchronized during the simulation. For example, it is important that no neuron is simulated before all synapses have finished propagation. Otherwise, some data dependencies might be violated. Implementing the three synchronization barriers shown from figure 4.12a is done easily, as a consequence of the fact that the host launches the kernels in order and it synchronizes the launch of the new kernel with the termination of the previous one (synchronization is performed by the host). However, this approach has two major disadvantages. The first one is that very many kernels are launched during one simulation. Every kernel launch has an overhead time of around 3us. This is the time necessary for initiating the kernel execution and depends very little on the number of parameters passed to the kernel. Additionally, the overhead time does not depend on the amount of computation performed by the kernel.

If the kernel does not do a lot of computation the overhead time can represent a significant fraction out of the total simulation time. For example, if a 10s simulation is run with a 1ms time step the total amount of overhead time is 10s*10^3(time steps/s)*4(kernels/time step)*3us (overhead time/kernel) = 0.12s, which in this case can represent up to 30% of the entire simulation time. The second disadvantage comes from the inability to efficiently use shared memory. Because the GPU global memory is not cached, the shared memory is an alternative to accelerate data access. Each SM inside the GPU has 16KB of fast memory that is visible to and can be shared by all threads of the same block. The shared memory can be as fast as a processor register as long as a bank conflict is not present during the simultaneous accesses of different threads. Anyway, the intended purpose of shared memory is that at the beginning of a kernel each thread copies the necessary data from global memory to shared memory. During the execution of the kernel, all intermediary results are stored inside the shared memory. The final results are copied to global memory only at the end of the kernel. This way, the shared memory works like a software managed cache; it is the responsibility of the programmer to assure data coherency between threads of different blocks. The lifetime of a variable declared as shared is only as long as the duration of the kernel. This makes the model presented in figure 4.12a very inefficient because it is necessary to store intermediary results in global memory between kernels.

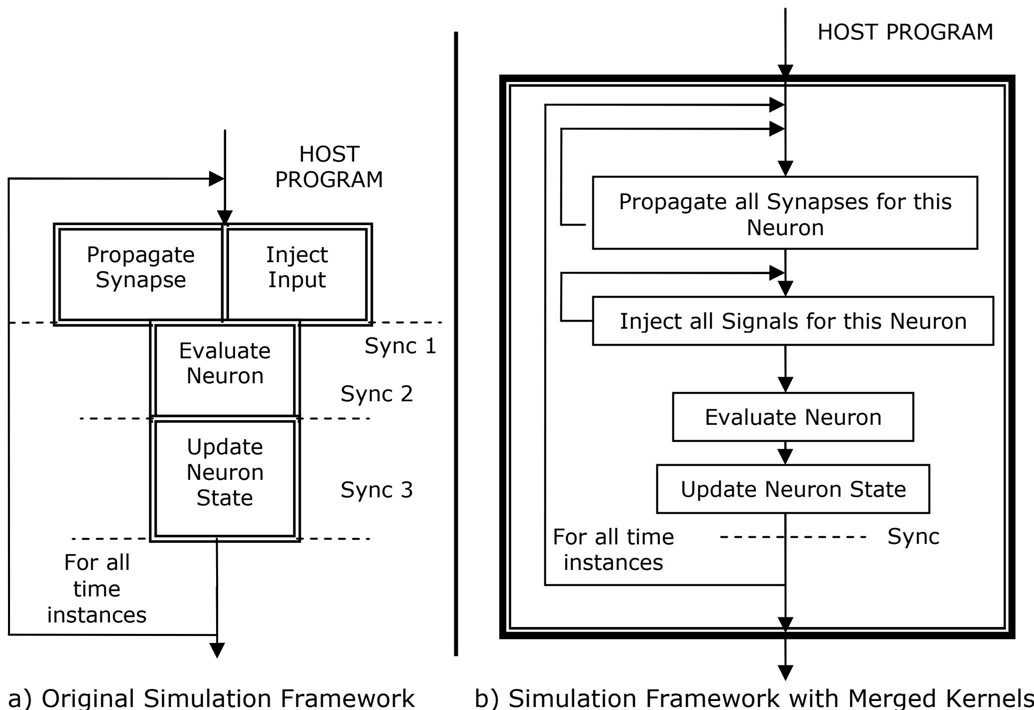
We propose a second approach to organizing the simulation depicted in figure 4.12b. In this case all kernels, and also the main time loop are merged together into a single kernel. This way, the kernel launch overhead time is eliminated. More importantly, shared memory can be exploited very efficiently. However, because the simulation is no longer divided into kernels, the host can not control and synchronize the simulation anymore. Once the kernel is launched, it is the job of the GPU to synchronize threads. First, the model was slightly reorganized such that it needs fewer synchronization barriers. In the original simulator (4.11a), the kernels dedicate individual threads for the processing of each synapse, input injection and neuron respectively. Alternately, in figure 4.12b, each thread processes everything that is related to the functioning of one neuron.

BUPT

The kernel has two internal loops that process serially all synapses and input signals targeting one neuron. Therefore, once the execution reaches the "Evaluate Neuron" stage it is assured that the previous stages have completed. It is not important that the processing of other synapses or other input signals in the neural network might not completed yet, since they do not influence the functioning of this neuron during the current time step. The introduction of two loops inside the kernel does not impact the efficiency of parallel processing if the number of neurons is large enough to fully load the resources of the GPU.

Unfortunately, the synchronization barrier at the end of the simulation time step cannot be avoided. The hardware offers a synchronization instruction, but it is only able to synchronize threads of the same block. This is because the threads of the same block will be executed on the same SM. The only way to use this hardware synchronization is to put all threads into the same block, but that would waste GPU resources since only one SM would be used. An alternative, which allows synchronizing all threads, is to use a software implemented barrier described by the following steps:

- Synchronize all threads in each block.
- First thread of each block atomically increments counter variable that resides in global memory.
- First thread of each block waits in loop until counter reaches the total number of blocks.
- Synchronize all threads in each block.



a) Original Simulation Framework b) Simulation Framework with Merged Kernels

Fig.4.12. Merging Kernels

The synchronization is done in two steps. First, all threads in each block are synchronized. Second, each block delegates a thread to communicate globally and synchronize blocks. Because the counter is a shared resource, it needs to be incremented atomically (serialized by hardware) in order to eliminate race conditions and miscounting. Also, because atomic operations are slow and because writing to global memory is slow in general, the two step approach presented above is much better than synchronizing all threads globally.

However, there is a trick to this approach. When a kernel is launched the blocks are enumerated and distributed evenly to the SMs of the GPU. After counting the necessary resources for the execution of each block, it is decided how many blocks can be launched in parallel on each SM. If the SM cannot accommodate all its designated blocks at the same time, the remaining blocks wait in a queue until a running block finishes execution and resources are freed. Note that a block does not suspend execution if it is not doing anything (waiting in a loop). This can lead to a dangerous situation where the counter condition is never achieved. This is because inactive blocks wait in a queue for the active blocks to complete execution. At the same time, the active blocks never complete execution because they wait for a condition that needs all blocks to be active. One way to solve the situation is to set the number of blocks equal to the number of SMs which in the case of the GT9800 GPU (employed in this paper) is 14. Afterwards, the number of threads in each block can be computed in order to have a total number of threads equal or higher to the number of neurons in the neural network.

4.3.3. Using Shared Memory

As presented in the previous paragraph, shared memory is a powerful tool to speed up memory access. Merging all kernels into a single one makes shared memory even more appealing because variables will be present in shared memory during the entire simulation without having to load/store them from/to global memory. Because the shared memory is a limited and precious resource it is important to see what to store in there and what not. A statistical analysis of how memory is accessed during a simulation reveals that almost 80% of all accesses are performed to the data delay line of the neurons. The number of accesses to this data structure is not constant and depends on the contents of the delay lines during the simulation. However, in all situations, they dominate the overall amount of accesses, making the data **DelayLine** objects the best candidates to occupy shared memory. The next best candidate is the neuron output variable which is responsible for about 14% of all accesses. This variable, however, is not suitable for shared memory because it needs to be broadcasted globally at the end of each simulation time step. This is because the model has no restrictions regarding the connectivity of the network, and so synapses can connect neurons that run on threads residing in different blocks on the GPU. Accesses to other variables are less significant in number. Therefore, in our implementation the data **DelayLine** are the only objects that are stored in shared memory.

In order to have a higher bandwidth, shared memory is divided into 16 banks that can be accessed simultaneously as long as there is no bank conflict (two threads accessing memory locations of the same bank). The shared memory space is organized such that consecutive addresses are found in consecutive memory banks rather than the same bank. This is because in most CUDA programs successive threads access consecutive memory addresses. With this argument in mind, we propose two ways to organize the data **DelayLine** class.

BUPT

Consider that TN is the number of threads running in the same block (and also on the same SM) and that DS is the size of each **DelayLine** buffer. Therefore, a total space of TNxDSx8 bytes needs to be allocated in shared memory for each block. 8 bytes are needed for each entry of the delay line because two 32bit words need to be stored (one float word for the spike amplitude; one unsigned integer for the spike timestamp). The size of the delay line DS is constant and is set at the beginning of the application. For example, if a network of 2000 neurons must be simulated, the computing grid will have 14 blocks x 143 threads. Given the maximum of 16KB of shared memory per block the maximum delay size DS is 14 (143x14x8B = 15.64KB). The limit of the delay line size would not be present when using global memory. However, it is a compromise worth taking because the speed improvement is significant and it is rarely the situation when more than 5 entries in the **DelayLine** buffer are needed. Anyway, if a neuron receives a spike and its delay line is already full, the spike that has the largest timestamp is eliminated.

Figure 4.13 shows two ways of storing the delay lines in memory. For simplicity, the figure only shows 4 neurons (threads) and each has a delay line of size 6. In the upper part of the figure the **DelayLine** entries of different neurons are interleaved. In the lower part of the figure all entries of the same neuron are grouped. In the upper part, banks conflicts are eliminated naturally because the threads access simultaneous consecutive addresses thus distinct banks. In the lower part, bank conflicts are eliminated if DS%16 is odd. For this implementation the first approach was preferred because no restrictions are imposed on DS.

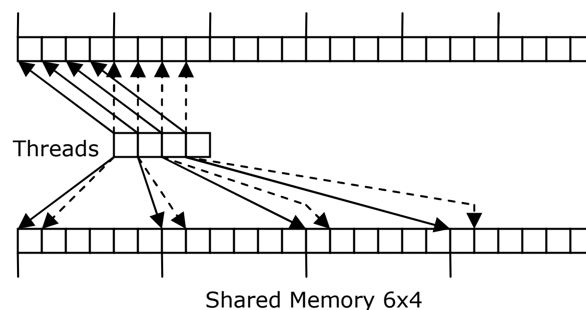


Fig.4.13. Two versions for implementing the delay line inside shared memory

4.3.4. Overlapping Computation and Data Transfer

The simulator records traces of the neural activity for all neurons. At the end of the simulation two sets of traces are available: one binary set for the neuron outputs and one floating point set for the neuron membrane potential. This data needs to be copied from the device memory to the host computer memory in order for it to be available to the host main application (unless the application continues on the GPU). If the neural network is large and/or if the simulation length is large the amount of data that needs to be transferred is significant and can add up to tens or even hundreds of MB. There are situations when the host application only requires the neuron binary spiking traces and not the neuron membrane potentials. If so, the amount of data that needs to be transferred is significantly reduced. Anyway, in order to reduce overall application time, NVIDIA offers hardware mechanisms that allow overlapping of computation and data transfer. This enables

BUPT

the user to transfer available simulation results while new results are being computed.

The NVIDIA architecture is organized as a stream processor. When the host computer expects the GPU to perform a certain task it places a request for this task in a queue called stream (usually a task is either a kernel launch or a memory transfer operation). The host continues to execute the host program until another GPU specific instruction is met. At this point, the host checks if the GPU has completed the previous action. If true, it sends a new request, else it waits for the GPU to finish. This is called synchronous operation mode, in the sense that the host is always synchronized to the GPU. It is worth mentioning that in this operation mode only one stream is used and that stream will only contain one action in its queue, since no new task can be sent to the GPU until the previous one is finished. Another type of operation mode is asynchronous. In this mode, the host does not need to synchronize to the GPU until a specific synchronization instruction is reached. This way the host application can send several tasks to the device without actually waiting for them to complete. Also, the host can place the tasks on distinct streams.

Figure 4.14 shows how this procedure allows overlapping of computation and data transfer. Consider that the simulator presented in paragraph 4.3.2 (figure 4.12b) breaks the simulation into N sequential parts and launches N consecutive kernels. This is useful because as soon as the first kernel finishes execution the hardware starts to transfer the results from the first kernel while the second kernel computes. The streams in figure 4.14 are conceptual and only exist from the programmer's point of view. In hardware, the tasks are sent to the actual engines in the same order they are inserted by the host into streams. Additionally, the hardware has an inter-engine mechanism which assures that all tasks coming from the same stream are performed in the same order as specified by the stream. For example, task CPY0 will start after task SIM0 is completed even though they are performed by different engines.

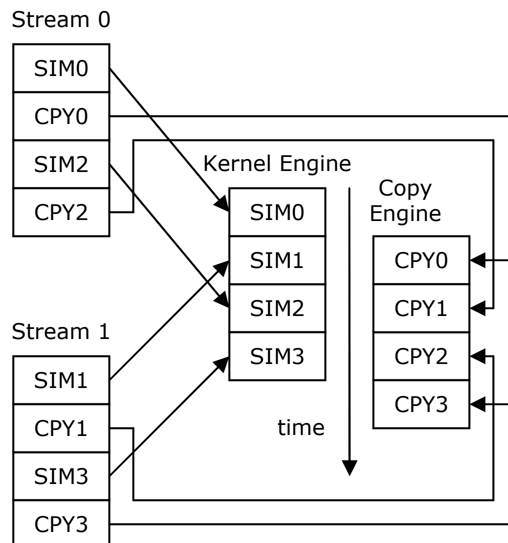


Fig.4.14. Overlapping computation and data transfer

BUPT

As presented in paragraph 4.3.2 launching a kernel produces about 3us of overhead time, regardless of the amount of computation inside the kernel. Therefore, splitting the simulation into N segments, and so launching N kernels instead of one, adds a penalty time equal to N*KLT, where KLT is the kernel launch time. At the same time, the transfer time required to send the results back to the host TRT reduces to its Nth fraction. Because the penalty increases linearly while the transfer time decreases non-linearly it is optimal to increase N as long as the rate of decrease for the transfer time is higher than KLT (4.5).

$$KLT \leq \frac{TRT}{N} - \frac{TRT}{N+1} = \frac{TRT}{N*(N+1)} \quad (4.5)$$

Because N is usually a large number (over 100) eq. (4.5) is approximated well by the next relation.

$$N < \sqrt{\frac{TRT}{KLT}} \quad (4.6)$$

4.3.5. Using Constant Memory and Texture Memory

As presented in figure 4.11 the GPU is equipped with two special types of memory spaces: the constant memory and the texture memory. Both types of memory are cached and can provide further improvement to memory access speed. As already mentioned, the GPU does not dedicate many transistors to complicated memory management and sophisticated data caching. Therefore, the constant memory and texture memory caches must be simple. The simplicity is assured by the fact that both memory spaces are restricted to being read-only. This way, the coherency of cached data is assured implicitly and significant hardware can be excluded from the cache. The only data in the Spiking Neural Network model that is constant is the synapse information and the general neural network parameters. The neural network parameters are few and it is easier to store them in shared memory which is faster than the caches. Because of its large size the synapse information cannot be stored in shared memory and so constant memory or texture memory can represent an alternative (Synapse data size = 16B x number of synapses; up to hundreds of MB).

The efficiency of caching depends drastically on the cache hit rate. Therefore, the address access pattern that results during the execution of an application is very important and determines the efficiency of the cache. In general, if some data is read and cached, it needs to be read again at least a few more times before it is evicted from the cache. Examining the memory access pattern for this application reveals that using the cache will be inefficient. During each simulation time step the data stored by a synapse is required only once. In addition, all synapse information is required during each simulation time step. Because the cache is only 64KB and cannot store the entire synapse data, each synapse is stored and then is evicted from the cache before it is needed again.

Consequently, for this application the cache cannot be exploited so the synapse information is stored in global memory. The improvements described in paragraph 4.3 are also found in [65].

BUPT

4.3.6. Simulation Results

Table 4.3 presents the achieved speedups for different neural network sizes. The GPU simulator is benchmarked against a simulator written in C++. Note that the C++ simulator is inherently faster than the original MATLAB simulator. The C++ model is run on a system with the following configuration: Intel Core i7 CPU at 2.67 GHz, 4GB DDR3 RAM, 64bit Windows 7. The CUDA C model is run on a NVIDIA GeForce GT9800 GPU with the following specifications:

- CUDA Cores 112 (14 Multiprocessors x 8 Stream Processors)
- Graphics Clock 600Mhz
- Processor Clock 1500Mhz
- Memory Size 1GB
- Memory Clock 900Mhz
- Memory Bandwidth 57.6 GB/sec

The speedup is calculated as the ratio of the C++ simulation time and the CUDA C simulation time. There are four versions of CUDA implementations: V1 is the CUDA model presented in figure 4.12a without any of the additional improvements; V2 is the CUDA model presented in figure 4.12b where the kernel overhead time eliminated; V3 is V2 with the delay lines implemented in shared memory; V4 is V3 with the overlap of computation and data transfer. The simulation times used at computing the speedups are estimated by averaging the results of 100 simulations performed on different networks of the same size and with the same number of synaptic connections.

Table 4.3. Achieved Speedups

Network Size	**Speedup**			
	V1	**V2**	**V3**	**V4**
8x8x8	x1.19	x1.42	x2.41	x2.44
9x9x9	x1.43	x1.66	x2.91	x2.98
10x10x10	x1.86	x2.14	x3.83	x4.01
11x11x11	x2.45	x2.60	x4.56	x4.96
12x12x12	x2.87	x3.25	x5.49	x6.14
13x13x13	x3.38	x3.76	x6.02	x6.84
14x14x14	x3.72	x4.08	x6.14	x7.12
15x15x15	x4.05	x4.39	x6.22	x7.23

It can be noticed that in general larger networks have greater speedups. This is because for larger networks, more computational workload allows the CUDA model to better exploit parallelism and more efficiently utilize its resources. On the other hand the simulation time of the C++ model scales linearly with the amount of computations.

BUPT

5. LIQUID STATE MACHINE AND LIQUID COMPUTING

5.1. Introduction

The concept of "Liquid State Machine" or "Liquid Computing" was introduced by Wolfgang Maass in [18], [73], and [74]. The idea behind the concept comes from a basic observation that a liquid medium (i.e. a pond of water) disturbed by a stimulus (i.e. a stone) will produce a sequence of unstable states that are stimulus specific. A trained independent observer should be able to extract all significant features of the stimulus by examining the trajectory of transitory states of the liquid medium. The approach offers an advantage, if and only if, the observer is able to extract more features by examining the liquid medium than by examining the stimulus directly. This is potentially possible when the liquid medium is complex enough to produce information analysis and feature decomposition.

The theory can be generalized from an actual "liquid medium" to any recurrent medium having any physical support [70], [78], [80], [84], [85]. This project uses a "liquid medium" implemented within a recurrent neural network of spiking neurons. The authors of [18] also suggest that an implementation with spiking neurons is very suitable and also biologically plausible. Even though it is strictly symbolic this thesis continues to refer to the recurrent medium as "liquid medium".

5.2. Liquid State Machine Architecture

The architecture of a Liquid State Machine is presented in figure 5.1. Two major sections can be depicted: the Recurrent Liquid Medium and the Read-out Units.

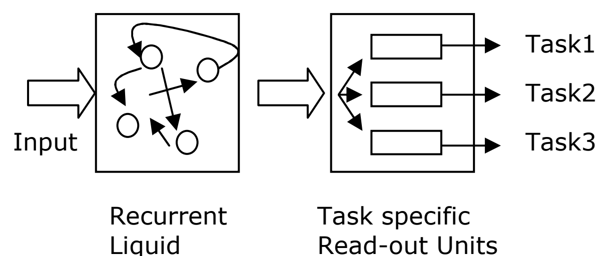


Fig.5.1. Liquid State Machine architecture

The liquid medium is task independent and has multiple random connections between units. The input stimulus is injected inside the liquid medium and produces a stimulus dependent sequence of unstable states. On the other hand the read-out

units are task dependent and need to be trained in order to read, interpret and convert the current liquid state into a desired output.

5.2.1. Recurrent Liquid Medium

The liquid medium behaves like a time invariant filter[7] than has fading memory[8]. It is implemented by a recurrent spiking neural network. Neurons are distributed along several bi-dimensional layers [82]. Each individual neuron can choose from the four neuron models presented in chapter two: "integrate and fire" (I&F), "integrate and fire with burst" (I&FB), "integrate and fire with adaptation" (I&FA) and "resonate and fire" (R&F) [72].

Neurons inside the liquid medium are connected via dynamic synapses having the model presented in 2.2.4. The probability of a synaptic connection to exist between neuron A and neuron B is given in eq. (5.1).

$$P(D) = C * e^{-\frac{D(A,B)}{\lambda}} \qquad (5.1)$$

- D is the Euclidian distance between neuron A and neuron B
- λ is the density of connections
- C is a scaling constant.

In this configuration neurons are connected to several other neurons that are placed in its immediate neighborhood. The chance of connection decreases as the neurons are further away. Some neurons are inhibitory. This means that their spikes will have a negative effect upon the membrane potential of the neuron that is receiving the spike. The amount of inhibitory neurons is determined by the

---

[7] A filter F is called *time invariant* if any temporal shift of the input function $u(\times)$ by some amount t_0 causes a temporal shift of the output function $y = Fu$ by the same amount t_0, i.e., $(Fu^{t0})(t) = (Fu)(t + t^0)$ for all t, $t_0 \in \mathbf{R}$, where $u^{t0}(t) = u(t + t_0)$. Note that if U is closed under temporal shifts, then a time invariant filter $F{:}U^n \rightarrow (\mathbf{R}^{\mathbf{R}})^k$ can be identified uniquely by the values $y(0) = (Fu)(0)$ of its output functions $y(\times)$ at time 0.

[8] *Fading memory* (Boyd at al., 1985) is a continuity property of filters F which demands that for any input function $u(\times) \in U^n$ the output $(Fu)(0)$ can be approximated by the outputs $(Fv)(0)$ for any other input functions $v(\times) \in U^n$ that approximate $u(\times)$ on a sufficiently long time interval $[-T,0]$. Formally one defines that $F{:}U^n \rightarrow (\mathbf{R}^{\mathbf{R}})^k$ has fading memory if for every $u \in U^n$ and every $\varepsilon > 0$ there exist $\delta > 0$ and $T > 0$ so that $||(Fv)(0) - (Fu)(0)|| < \varepsilon$ for all $v \in U^n$ with $||u(t) - v(t)|| < \delta$ for all $t \in [-T,0]$. Informally a filter F has fading memory if the most significant bits of its current output value $(Fu)(0)$ depend just on the most significant bits of the values of its input function $u(\times)$ from some finite time window $[-T,0]$ into the past. Thus, in order to compute the most significant bits of $(Fu)(0)$ it is not necessary to know the *precise* value of the input function $u(s)$ for any time s , and it is also not necessary to know anything about the values of $u(\times)$ for more than a finite time interval back into the past. Note that a filter that has fading memory is automatically causal.

BUPT

percentage parameter *INH*. Figure 5.2 presents a Liquid Medium created in MATLAB that has the following parameters:

- Network topology: 4x4x6 (96 neurons)
- Percentage of inhibitory neurons: 30% (drawn in red)
- Synapse probability parameters: C = 1, λ = 0.8

For clarity, the figure shows the synaptic connections of only one neuron. Blue connections are excitatory while red connections are inhibitory (coming from inhibitory neurons).

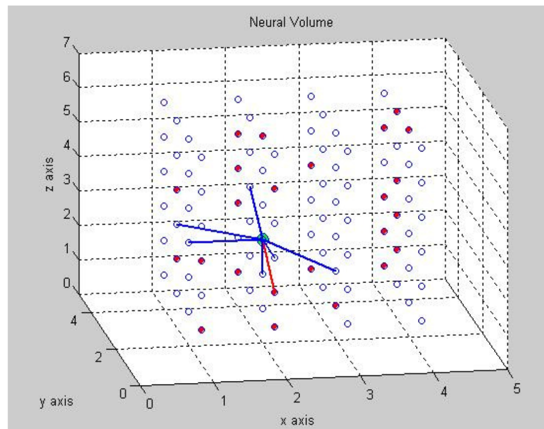


Fig.5.2. Liquid State Machine implemented by a recurrent Spiking Neural Network

5.2.2. The Read-Out Units

The Read-Out Units are implemented either by parallel perceptrons or by multilayer feedforward networks. A parallel perceptron is a group of *N* single McCulloch-Pitts perceptrons that are fed with the same input *P* (figure 5.3). The output of the parallel perceptron *Y* is produced by a squashing function *S* that counts the number of active neurons and maps this number onto a real number, as shown in equation (5.2), where *i* is the number of perceptrons and y_i is the output of each single perceptron.

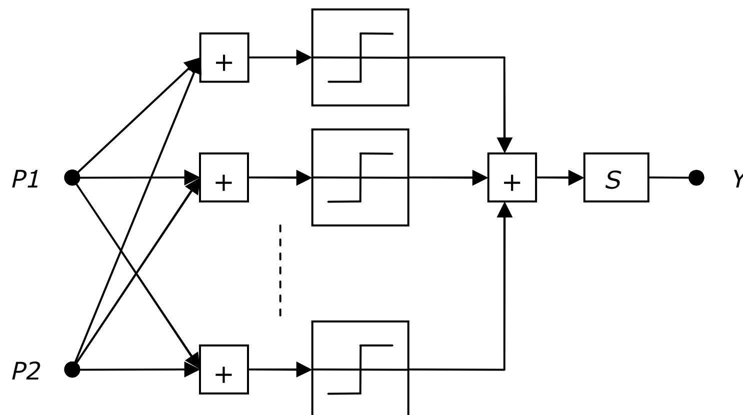


Fig. 5.3 Parallel Perceptron Readout Unit

BUPT

$$Y(p) = S\left(\sum_{i=1}^{N} y_i(p)\right) \quad (5.2)$$

Alternatively, the readout units can be implemented by a multi-layer feedforward neural network as presented in figure 5.4 [44]. In this case the activation function is the sigmoid function. Paragraphs 5.4 and 5.5 present the p-Delta training rule and the backpropagation algorithm, which are useful for training the parallel perceptron and the feedforward network. In addition three improvements of the p-Delta rule are also presented.

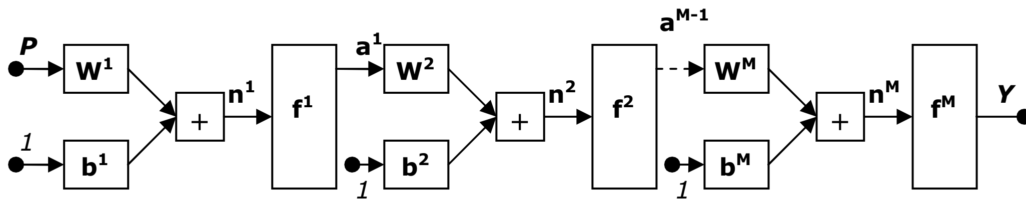


Fig. 5.4. Multi-Layer Feedforward Readout Unit

5.3. Liquid States. Separation Property

When the input signals are injected some neurons will start to fire spikes at different frequencies. These spikes propagate via the recurrent synapses and force other neurons to fire new spikes. The result is a complex spike activity propagating back and forth inside the liquid medium.

The spike pattern produced by an input stream should be input specific and should help at classifying the input. In order to quantify this ability [18] introduces a useful macroscopic property called separation property *SP*. This property allows comparison of how different are the spike patterns produced by the liquid medium when injected with two different input streams. A high separation property means an easy classification task for the read-out units. The following paragraphs present how the separation property is computed.

The spike pattern produced by the liquid medium when injected with an input stream is considered to be a sequence of states called liquid states (one liquid state for every time instance). Basically, a liquid state is a vector that has the dimension equal to the number of neurons inside the liquid medium, holding "1" if the neuron fires (at that time instance) or "0" if the neuron rests. The liquid states are produced by sampling the spike activity along time. In order to study if one spike pattern is different from another spike pattern both "non-temporal" and "temporal" differences/similarities need to be taken into account. Performing a "non-temporal" comparison can be done by simply computing the Euclidian distance between the two liquid states. However, computing the separation property as the time average of these distances is insufficient. This approach would loose any "temporal" relationships between spikes and also would result in a macroscopic property that is "time-noise" intolerant (if some spikes slightly change their time positions *SP* is high even though spike patterns are still very similar).

BUPT

In order that the *SP* makes both "non-temporal" and "temporal" comparisons each spike from the spike activity is replaced with a decaying exponential. This is done by performing a convolution operation between the spike activity and a decaying exponential. This way, every liquid state holds information about current spikes but also about past spikes (note that "time-noise" has little impact with this approach). After the convolution operation is performed the spike activity is sampled to produce the set of liquid states. *SP* is computed as the absolute difference between the liquid states. Further information about improving the separation property can be found in [71].

5.4. The Parallel Perceptron Readout Unit. The p-Delta Learning Algorithm.

The read-out units are trained using the "p-Delta" learning rule. The algorithm was introduced by Peter Auer in [66] as a direct solution to designing the readout units of a Liquid State Machine [18]. In general, the learning rule can be used for training any parallel perceptron regardless of the application. The original algorithm was developed assuming that it will be applied to a system implemented exclusively in hardware. For such a system, the communication between individual neurons and the control unit can be a major difficulty. Therefore, the algorithm had one important constraint, simplicity.

The following paragraphs present the original algorithm and also an improved version. It introduces a few modifications that make the algorithm faster, more stable and with a higher noise margin. However, the changes complicate the algorithm making it less suitable for a hardware implementation. Currently, our team is using a software model for the spiking neural network and is not aiming towards a software independent implementation. This allows a more complicated algorithm to be easily implemented.

As presented in 5.2.2 the read-out units can be implemented by parallel perceptrons. The following paragraphs present the concept more thoroughly.

5.4.1. The Parallel Perceptron

A single perceptron, as introduced by McCulloch-Pitts, is a gate that computes an averaged sum of all inputs. If the sum is greater than the threshold *TH* the perceptron outputs "1" otherwise "0". Mathematically, this is written as in equation (5.3).

$$y(p) = \begin{cases} 1, wp > TH \\ 0, wp < TH \end{cases} \tag{5.3}$$

where w is the synaptic weight vector and p is the input.

The perceptron model can be easily implemented by a spiking neuron [32], [34] if p is considered to be the rate of the spike train. The output is "1" when the neuron fires and "0" otherwise. As already mentioned a parallel perceptron is a group of N single perceptrons that are fed with the same input P. The output of the parallel perceptron Y is produced by a squashing function S that counts the number of active neurons and maps this number onto a real number. The squashing function

BUPT

S can be any monotonous continuous function. However, for the presentation of the training algorithm the linear function in equation 5.4 was used.

$$S(n) = \frac{n}{N} * (Y_{\max} - Y_{\min}) + Y_{\min} \tag{5.4}$$

where Y_{max}, Y_{min} are the boundaries of the output range and *n* is number of active neurons.

The "p-Delta" algorithm can be efficiently used for training a parallel perceptron to map a set of given input data *p* to a desired target output *t*.

5.4.2. The Single Perceptron Delta Rule

This is the simplest learning rule that can be applied to a single perceptron. Let *p, y* and *t* be the input, output and target data respectively. If the output *y* is '0' and the target *t* is '1' it means that the dot product *wp* is too small in comparison to the desired threshold *TH*. In order for the dot product to increase, the weight vector *w* needs to move toward the data vector *p*; hence the angle between the two vectors will decrease. If the output *y* is '1' and the target *t* is '0' it means that the dot product *wp* is too large and so the weight vector needs to move away from the data vector. If the output *y* matches the target *t* no change is done. The rule can be mathematically expressed by eq. (5.5).

$$w \Leftarrow \begin{cases} (1-\lambda) * w + \lambda * \begin{cases} +p, t > y \\ -p, t < y \end{cases} \\ w, t = y \end{cases} \tag{5.5}$$

where λ is the learning rate.

5.4.3. The Parallel Perceptron p-Delta Rule

In theory the approximation error of the parallel perceptron can be as small as half the size of the quantization step. Therefore, the algorithm could theoretically set the desired accuracy ε to the value given by eq. (5.4), where Y_{min} and Y_{max} are the same as in equation (5.4).

$$\varepsilon = \frac{Y_{\max} - Y_{\min}}{2 * N} \tag{5.4}$$

However, reaching this error level is not guaranteed. This is because the algorithm can get stuck in a local error minimum and so it will not find the global minimum that satisfies eq. (5.4). Therefore, from now on it is considered that the accuracy ε is set by the user application and that the number of neurons *N* is sufficient for the accuracy constraint to be met. Given the input data *p*, the output of the parallel perceptron *Y(p)* is computed with eq. (5.2). If the weights of the parallel perceptron are correct the output should be as close to the target *t* as

BUPT

constrained by ε. This is expressed in eq. (5.5).

$$|Y(p)-t|<\varepsilon \tag{5.5}$$

If the output is greater than the target it means that too many neurons are active and so the weights of "some" of the active neurons should move away from the data. If the output is too small compared to the target, too few neurons are active and so "some" of the inactive neurons should move their weights towards the data. The term "some" is flexible and represents the answer to the question: "how many and which neurons should be chosen for weight modification?" The authors of [66] suggest that all active neurons should be updated if the output is greater than the target and also that all inactive neurons should be updated if the output is smaller than the target. This approach does not offer a great convergence speed or stability. However, it minimizes communication between neuron units if a hardware implementation is preferred. In [66] it is also suggested that the stability and convergence speed could be improved if only a few neurons (or one [67]) are chosen for weight modification. Those neurons should be the ones that have a dot product *wp* that is closest to the threshold. This approach on the other hand increases communication as the neuron units would need to broadcast their dot product to the central unit.

Because we use a software implementation of the spiking neural network model, communication bandwidth is not a constraint. Therefore, during each training iteration, the weights of only one neuron are updated. This neuron is considered "winner". A neuron is declared winner if it has a dot product *wp* that is closest to the threshold and also if is on right side of the threshold. Therefore, the learning rule can be mathematically expressed by equation (5.6).

$$\begin{aligned} w_k \Leftarrow (1-\lambda)*w_k - \eta*\left(\|w_i\|^2-1\right)*w_i + \\ +\lambda*\begin{cases} -p, Y(p)-t>\varepsilon \\ +p, t-Y(p)>\varepsilon \end{cases} \end{aligned} \tag{5.6}$$

where:

- λ is the learning rate
- η is the normalization rate
- k is the winning neuron
- i =1….N

The middle term, containing the norm of the weight vector is a correction that is performed for each neuron on all iterations. This correction preserves the angle of the weight but brings the length of the vector to unit length. This is important because the dot product *wp* represents the angle between the two vectors only if the lengths of the vectors remains roughly the same.

5.4.4. Adaptive Learning Rate

The first modification to the original algorithm is the introduction of an adaptive learning rate. The learning rate is recomputed at each iteration as in eq. (5.7).

BUPT

$$\lambda \Leftarrow \lambda_{\max} * \frac{msq_error}{mean_msq_error} \quad (5.7)$$

The learning rate starts from a maximum value λ_{max} and then decreases as the parallel perceptron starts to approximate the data well.

5.4.5. Greedy vs. Not Greedy

The second modification to the algorithm is the implementation of a conscience mechanism. A statistical study was done to see how fast the algorithm converges. The algorithm is considered to have converged when the parallel perceptron approximates the target with an error smaller than ε for every data point in the training set. 10000 simulations were performed for every data point *p* and target *t*. The target *t* is the result of a randomly chosen linear function that takes *p* as input variable. Each simulation starts with different initial weights for the neurons, records the number of epochs that the algorithm needs to converge and places it in a convergence histogram. Such a histogram is presented in figure 5.5. It is seen that most trials converge in less than 400 epochs (aprox. 56.7%). Some trials converge in more than 400 epochs but it is most likely that their convergence is caused by chaotic effects and therefore is unreliable. Because the convergence percentage is not very high it was interesting to see what prevents some of the trainings from converging.

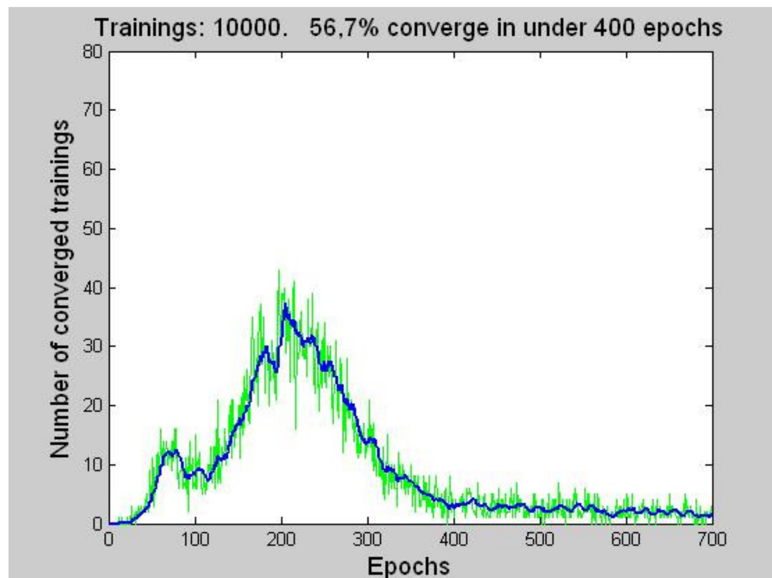


Fig. 5.5. PDelta Convergence histogram

An activity monitor variable was attached to each neuron forming the parallel perceptron. The activity variable counts the number of times the weight of a neuron is updated during the current epoch. Then, it divides the count to the total number of updates performed during the epoch for all of the neurons. After the epoch is finalized the activity variable reflects a percentage of how often was a

neuron declared winner. Figure 5.6 plots the activity traces for all the neurons during a trial that did not converge (each neuron is plotted in a different color). It is seen that initially several neurons have their weights updated. However, at some point, only one neuron is chosen exclusively for weight modification. This "greedy" behavior occurs when a neuron reaches a region that is densely populated with data and no other neuron is in the same region.

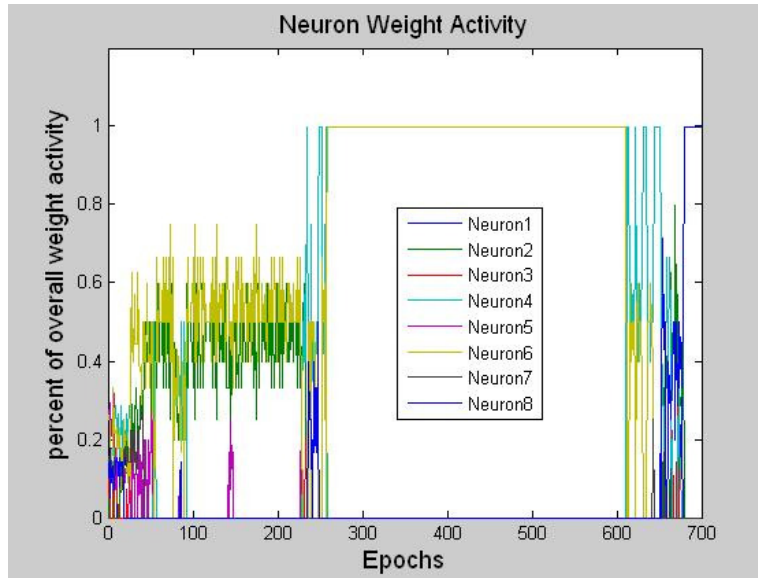


Fig. 5.6. Distribution of weight activity ("greedy" approach)

In order for the minimum error to be reached it is required that several neurons are present in this region such that the quantization is smoother. Unluckily, no other neuron is close enough to the data and so the single isolated neuron will always win the competition preventing other neurons to approach the region. In order to avoid this greedy behavior a conscience mechanism is inserted in the scoring function that is responsible for selecting the winner neuron. The scoring function calculates two scores: a proximity score *PS* and an activity score *AS*. Both scores are sub-unitary and reflect the probability of a neuron to be declared winner. The proximity score ranks the neurons based on dot product comparison. *PS* will be 1 for the neuron with a dot product that is closest to the threshold *TH* and 0 for the neuron that is furthest away. The activity score is computed by monitoring the activity trace of each neuron *i* inside a window of given size *WS*. The activity score *AS* is computed at any time *t* as shown in eq. (5.8).

$$AS_i(t) = 1 - \frac{1}{WS} * \sum_{k=1}^{WS} activity_trace_i(t-k) \tag{5.8}$$

The overall score is the product of the two scores *PS* and *AS*. The neuron with the highest overall score is declared winner. A neuron weight activity trace for the "not greedy" approach is presented in figure 5.7.

BUPT

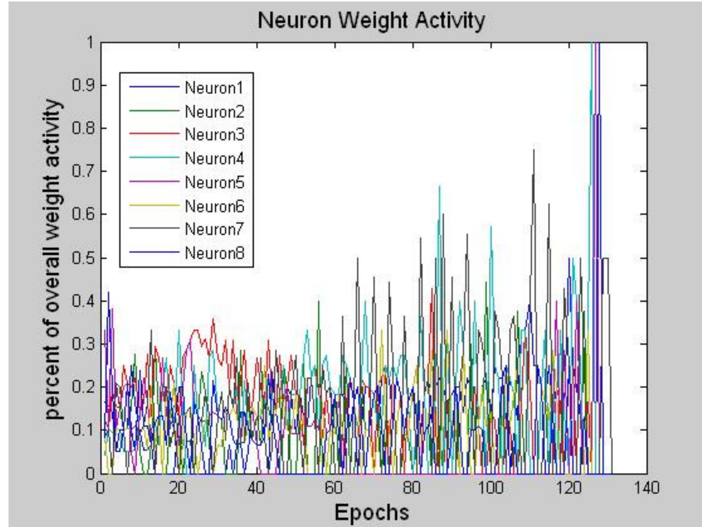


Fig.5.7. Distribution of weight activity ("not greedy" approach)

It is seen that in this case no neuron dominates as all neurons change weights throughout the epochs of the algorithm. It is seen that with this approach the algorithm converges a lot faster (140 epochs). In order to graphically compare the "greedy" and "not greedy" methods a similar histogram as the one in figure 5.5 was computed. Figure 5.8 plots the cumulated sums of several such histograms.

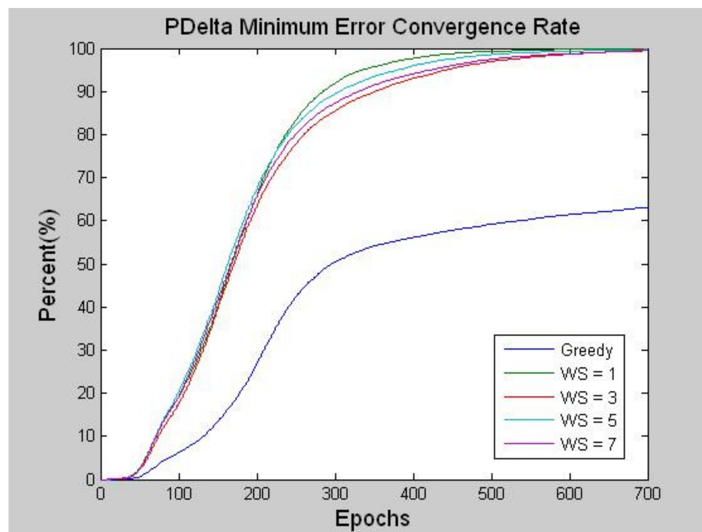


Fig.5.8. Convergence Rate

The blue trace represents the cumulated sum of the "greedy" histogram in figure 5.5. The other traces are cumulated sums of histograms obtained with the "not greedy" approach for several values of the window size *WS*. It is seen that the

BUPT

size of the averaging window WS does not significantly influence the convergence speed of the algorithm. However, it is also seen that the "not greedy" approach converges a lot faster than the "greedy" approach and also that the number of un-converged trials is significantly reduced. The same data is numerically available in table 5.1.

Table 5.1. Greedy vs. Not Greedy Convergence Statistics

Method	***Converge under x epochs***			***Never converge***
	100ep	200ep	400ep	
Greedy	6.21%	26.94%	56.13%	36.89%
Not Greedy	17.56%	63.38%	93.12%	0.53%

5.4.6. Adaptive Noise Margin Control

The algorithm stops when the error of the parallel perceptron is below the desired accuracy ε. This happens when for any data point in the training set all neurons will have a suitable dot product *wp* relative to the threshold *TH*. The problem with this approach is that some of the neurons could have a dot product *wp* that is very close to the threshold. In this case, any noise affecting the data can flip one of the neurons thus causing an undesired change at the output of the squashing function. More details on the necessity of a high noise margin can be found in [68], [69].

The original algorithm presents a solution to this problem by inserting a mechanism that produces a reasonable amount of noise margin between the thresholds and the *wp* products of all neurons for any data point. This is done by adding another term in the learning process as presented in equation (5.9). This applies only when the output is within the desired accuracy but the dot product *wp* is closer to the threshold *TH* than a specified margin *M*. In this case the weight vector *W* is moved towards or away from the data with the margin learning rate *mlr* such that the noise margin is increased.

$$w_k \Leftarrow (1-mlr) * w_k + mlr * \\ * \begin{cases} -p, -M < Wp - TH < 0 \\ +p, 0 < Wp - TH < M \end{cases} \quad \text{and} \quad \|Y(p)-t\| < \varepsilon \qquad (5.9)$$

The only problem with this idea is the constant learning rate *mlr* and the constant margin *M*. Choosing a margin beforehand can be tricky because the maximum obtainable margin is dependent on the distribution of the data within the input space. Also, a learning rate that is too big can lead to instability and also to the inability to reach the maximum margin even though this margin might has been guessed or computed beforehand. As a solution to this problem our approach introduces an adaptable learning rate *mlr* and an adaptable margin level *M*.

The margin level *M* is recomputed at each iteration as being the P% percentile of all margin levels for all neurons. Values for *P* between 5% and 20% have proven to work very well. This approach guarantees that the margin constraint *M* is not higher than what the p-perceptron can obtain considering the given data. It also assures that the algorithm adapts and increases the constraint *M* once the

BUPT

average margin increases. This leads the algorithm towards obtaining the highest possible margin even though the margin is not known beforehand.

Several attempts were made until an appropriate control rule for adapting the learning rate *mlr* was found. The first attempt was to increase the learning rate whenever the derivative of the average margin is high. This meant that the current average margin is still significantly small compared to the maximum margin so faster changes can be done. Whenever the derivative of the average margin is small or negative the learning rate *mlr* is decreased because the maximum margin is close or already reached. The problem with this method is that predicting the approach to the maximum margin by monitoring changes in the average margin can lead to a late prediction. If the learning rate is very high at this point the algorithm can become temporally unstable and loose whatever progress accumulated.

The second attempt tried to fix this problem by setting positive and negative boundaries for the learning rate. This assured that the algorithm will not get out of control. Unfortunately the boundaries were also data dependent and could only be set experimentally.

The third approach was more successful. At each step of the algorithm, the learning rate *mlr* is modified according to equation (5.10). *K* is a percentage with range -1 to 1 given by eq. (5.11), where Δm is the changes of the average margin during the last training epoch.

$$mlr \Leftarrow (1+K) * mlr \tag{5.10}$$

$$K = A * \Delta m + B \tag{5.11}$$

A normal learning regime is one where the margin increase Δm is equal to an estimated increase Δm_{est}. In this case mlr should be constant hence *K* should be zero. Δm_{est} is computed with eq. (5.12). In practice, Δm will not equal Δm_{est} but will randomly move inside a small interval around it. This will create small opposite changes in mlr that will average down to zero.

$$\Delta m_{est} = -\frac{B}{A} \tag{5.12}$$

Whenever Δm is constantly larger than Δm_{est} it is considered that the learning process allows a faster increase of the margin. Because in this case *K* is constantly positive the learning rate *mlr* will increase. In order to avoid an excessive increase of the learning rate the algorithm enters an adaptive regime where the estimated value Δm_{est} is reevaluated with eq. (5.13). This moves Δm_{est} towards the new average value of Δm. This is graphically shown in figure 5.9.

$$B = B + \xi * \left(\Delta m_{est} - \Delta m + \frac{dB}{dt} \right) \tag{5.13}$$

Whenever Δm is constantly smaller than Δm_{est} the algorithm adapts in the opposite direction. Figures 5.10 and 5.11 give the values of the average margin and of *mlr* respectively over the epochs. Please note that the margin enhancement

BUPT

mechanism is inhibited until the mean square error of the parallel perceptron reaches the desired accuracy level. This can be seen in the fact that until epoch 70 the average margin changes randomly as a result of the error minimizing learning.

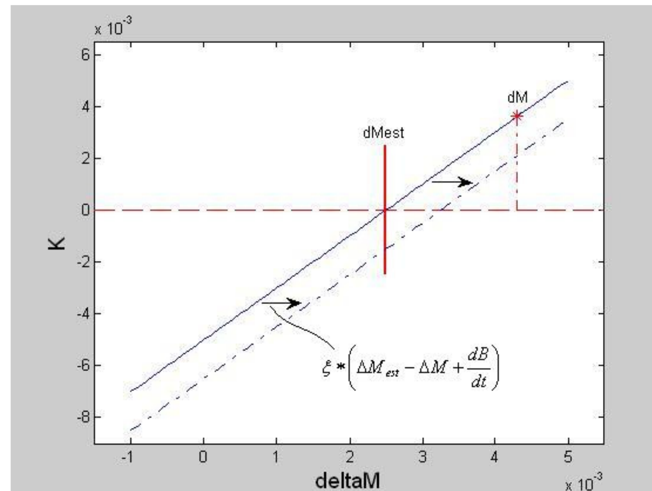


Fig.5.9. K Control Rule

It is seen that as the margin approaches its maximum value, the learning rate *mlr* decreases. The margin reaches its maximum value some time before epoch 150. It is seen that until this point the learning rate *mlr* is sufficiently small such that any additional changes do not make the learning process unstable or loose any of the gained progress.

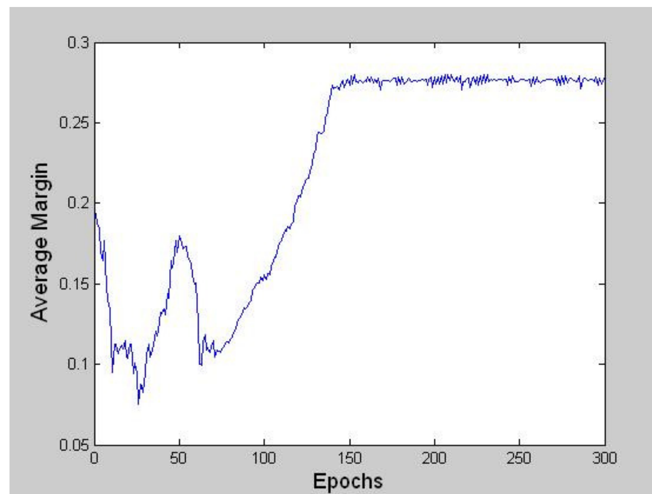


Fig.5.10. Average margin during training

The value for *A* was experimentally set to 2 and it reflects the sensitivity of *K* over *Δm*. The initial value for the learning rate *mlr* was also set experimentally. Anyway, it is preferred to have a very small value for *mlr* at the beginning of

training in order to avoid instability. The algorithm will quickly adapt *mlr* to a proper level.

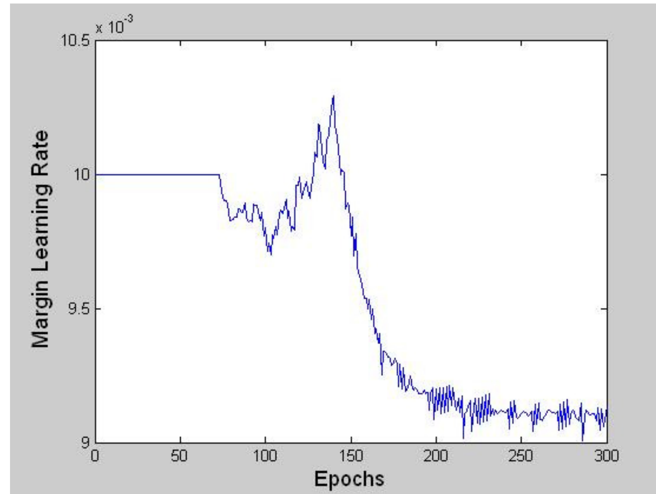


Fig.5.11. Margin learning rate ***mlr*** during training

5.5. The Feedforward Readout Unit. Backpropagation.

The structure of the multi-layer feedforward readout unit is presented in figure 5.4. The input of the readout unit is the state of the liquid medium. Because the spiking activity of the liquid medium is filtered, the resulting state will be a vector containing real numbers. Like all feedforward neural networks the readout unit is able to map the input vector to the desired output function. The backpropagation learning algorithm is used for training the readout unit. Equation 5.14 shows the recursive relation between the outputs of two successive neural layers. Index *m* is the number of the layer and has values between 1 and *M*. The equation is written by examining the network in figure 5.4

$$a^m = f^m\left(W^m a^{m-1} + b^m\right) \tag{5.14}$$

Consider that the following input-output training pairs are available (p^q, t^q) with *q*=1 to *Q*, where *Q* is the size of training set. For each input p^q the output of the network a^q can be calculated using eq. 5.14 and the error of the network is calculated as the mean square error *MSE* in eq. 5.15. It is considered that p^q, a^q and t^q are vectors. Index *k* in eq. 5.15 represents the kth training iteration.

$$F(k) = \left(t^q - a^q\right)^T \left(t^q - a^q\right) \tag{5.15}$$

The weights ***W*** and biases ***b*** of the neural network are adjusted such that the error function *F* is minimized. This is done by using the gradient decent technique presented in eq. 5.16. Indexes *i* and *j* denote a connection between neuron *i* of the mth layer and neuron *j* of the $(m-1)^{th}$ layer. It is seen that weight changes are inverse proportional to the derivative. Additionally, the changes are modulated by learning rate *lr* which controls the speed and stability of the

algorithm. This method guarantees that at each iteration *k* the error is decreased. However, it does not guarantee that error will ever reach its global minimum.

$$\Delta w_{i,j}^{m}(k) = -lr\frac{\partial F}{\partial w_{i,j}^{m}}, \ \Delta b_{i}^{m}(k) = -lr\frac{\partial F}{\partial b_{i}^{m}} \tag{5.16}$$

The biggest difficulty of the algorithm is to compute the partial derivatives in eq. 5.16. This is because the error function F is indirectly related to weight values of different layers. Equation 5.17 decomposes the derivative of 5.16 into a product where S_i^m represents the sensitivity of the error function *F* relative to the net output n_i^m of the mth layer.

$$\frac{\partial F}{\partial w_{i,j}^{m}} = \frac{\partial F}{\partial n_{i}^{m}} * \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}} = S_{i}^{m} a_{j}^{m-1}, \ \frac{\partial F}{\partial b_{i}^{m}} = S_{i}^{m} \tag{5.17}$$

Equation 5.18 groups all sensitivities of layer m into a single sensitivity vector $\mathbf{S^m}$. The sensitivity vector can also be expressed recursively as a function of the sensitivity vector of the next layer. This is shown in equation 5.19.

$$\mathbf{S}^{m} = \begin{bmatrix} \frac{\partial F}{\partial n_{1}^{m}} & \frac{\partial F}{\partial n_{2}^{m}} & \cdots & \frac{\partial F}{\partial n_{I^{m}}^{m}} \end{bmatrix}^{T} \tag{5.18}$$

$$\mathbf{S}^{m} = \frac{\partial F}{\partial \mathbf{n}^{m}} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}}\right)^{T} \frac{\partial F}{\partial \mathbf{n}^{m+1}} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}}\right)^{T} \mathbf{S}^{m+1} \tag{5.19}$$

It is seen that the sensitivity vector also depends on the Jacobian matrix shown in eq. 5.20

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}} = \begin{bmatrix} \frac{\partial n_{1}^{m+1}}{\partial n_{1}^{m}} & \frac{\partial n_{1}^{m+1}}{\partial n_{2}^{m}} & \cdots & \frac{\partial n_{1}^{m+1}}{\partial n_{I^{m}}^{m}} \\ \frac{\partial n_{2}^{m+1}}{\partial n_{1}^{m}} & \frac{\partial n_{2}^{m+1}}{\partial n_{2}^{m}} & \cdots & \frac{\partial n_{2}^{m+1}}{\partial n_{I^{m}}^{m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial n_{I^{m+1}}^{m+1}}{\partial n_{1}^{m}} & \frac{\partial n_{I^{m+1}}^{m+1}}{\partial n_{2}^{m}} & \cdots & \frac{\partial n_{I^{m+1}}^{m+1}}{\partial n_{I^{m}}^{m}} \end{bmatrix} \tag{5.20}$$

Each element of the Jacobian can be calculated by substituting in eq. 5.14. This transforms eq. 5.19 to eq. 5.21, where $\dot{\mathbf{F}}(\mathbf{n}^{m})$ is given by eq. 5.22. This relation allows computing the sensitivities on all layers starting from the last layer and advancing towards the first layer of the network (hence the name

BUPT

backpropagation). The sensitivities of the last layer can be easily computed because the error function depends directly on the net output of the last layer. Matrix function $\dot{\mathbf{F}}(\mathbf{n}^m)$ can also be computed by evaluating all derivatives of the activation functions for the values of $\mathbf{n^m}$ produced by the input data p^q.

$$\mathbf{S}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{S}^{m+1} \tag{5.21}$$

$$\dot{\mathbf{F}}(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^m(n_2^m) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^m(n_{I^m}^m) \end{bmatrix} \tag{5.22}$$

A few additional improvements can be made to backpropagation. For example, eq. 5.23 displays a training rule that uses momentum. At each training iteration the weights are updated as a combination of two terms. The first term is calculated according to eq. 5.16 of backpropagation. The second term equals the weight updates performed during the previous training iteration. This assures that the weights update has inertia (momentum) and has a tendency to preserve its dynamic trajectory. The advantage of having momentum is that the training is less likely to get stuck in a local error minimum. As a result, the algorithm allows a temporary increase of the error function. Another improvement is using an adaptive learning rate.

$$\Delta\mathbf{W}^m(k) = (1-\lambda)\Delta\mathbf{W}^m_{backprop}(k) + \lambda\Delta\mathbf{W}^m(k-1) \tag{5.23}$$

The readout units can also be implemented by spiking neurons instead of classic logsig neurons. In this case, an alternative to backpropagation in presented in [75], [76] and [77].

BUPT

6. GABOR FILTERING USING LIQUID STATE MACHINES AND SPIKING NEURONS

6.1. Introduction

Recent studies of the visual cortex have revealed the existence of specialized cells that are selective to particular frequencies and orientations. The discoveries of Hubel and Wiesel revealed significant information regarding the functioning of these cells. It is generally accepted that their functionality is successfully approximated by 2D Gabor wavelets, which are also tuned to specific orientations and frequencies.

Because face detection and recognition is a task that is easily performed by the brain, biologically inspired approaches like Gabor filtering are very appealing to researchers because of their potential. Presently, several face recognition systems using Gabor filtering have very good recognition rates proving that the approach is very promising [92], [93], [95] and [96].

6.2. Gabor Filtering

The Gabor filter was first introduced by Denis Gabor in [88] as joint entropy, minimizing frequency sensitive filter. It was later extended to two dimensions by Daugman [89], a modification that is also biologically motivating [90]. The kernel of the Gabor filter is shown in eq. 6.1. It is a 2D harmonic oscillation, composed of a sinusoidal plane wave of a particular frequency and orientation, restricted by a Gaussian envelope.

$$\Psi(k,x)=\frac{k^2}{\sigma^2}\exp\left(-\frac{k^2x^2}{2\sigma^2}\right)\left[\exp(ikx)-\exp\left(-\frac{\sigma^2}{2}\right)\right] \tag{6.1}$$

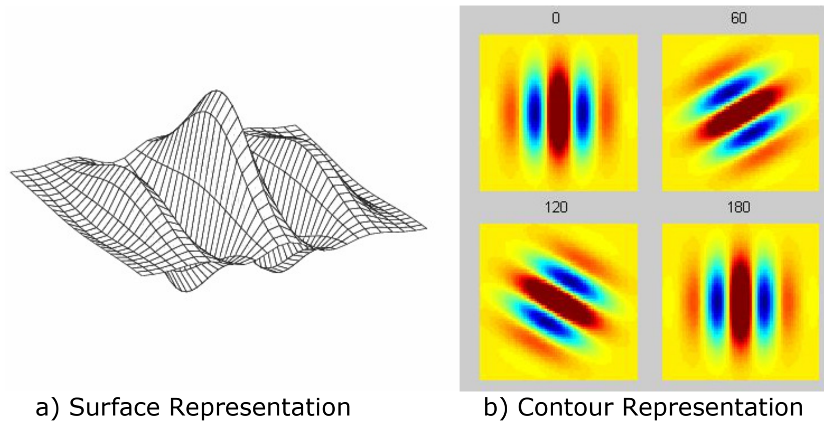
Figure 6.1a presents the surface representation for the absolute value of the Gabor filter. The contour of the Gabor surface can be depicted in 6.1b for 4 distinct orientations of the wave plane. Further information in Gabor filtering can be found in [91] and [94].

6.3. Filtering with Liquid State Machines

6.3.1. Input Signals

The input information is represented by raw pixel data. We have used two methods to feed the signals to the Liquid State Machine. The first method sends the pixel data directly to the liquid medium. Each pixel signal, represented as an analog current, can source a liquid neuron directly. Alternately, a layer of integrate and fire neurons can be interposed between the pixel signals and the liquid medium. In this

case the new layer rate-codes the pixel signals and sources the liquid medium with spike trains rather than analog signals.



a) Surface Representation b) Contour Representation

Fig. 6.1. 2D Gabor Filter Kernel

The disadvantage of this approach is the high number of input signals associated with larger input images. If we assume that one liquid neuron cannot be sourced by more than one input signal, the number of liquid neurons needs to be at least as large as the number of pixels in the input image. Moreover, if most of the liquid neurons are sourced by input data, the liquid medium might become saturated and might not have any free resources to perform any useful computation. This means that the size of the liquid medium must be at least a few times larger that the size of the input image. Another disadvantage is that in this case the input signals are static, fact that is more or less inconsistent with the dynamic behavior of the LSM.

The second method uses time-multiplexing in order to reduce the number of input signals. Figure 6.2 presents the architecture that was used. The left-side figure shows an image column circuit. It is constructed from a set of pixel circuits each connected to an individual pixel. All pixel circuits have a shared control signal called "phase shift signal". The control signal contains a pair of spike trains that have a constant phase shift relative to each other. The first spike train has a constant frequency and is considered to be the reference signal. The second spike train has a frequency that is either slightly smaller or higher, and so the phase of the second spike train shifts constantly relative to that of the first spike train. Each pixel circuit is sensitive to a unique phase difference. Whenever this phase difference occurs, the pixel circuit activates and generates its own spike train that is proportional to the input pixel. Otherwise the pixel circuit remains inactive. An I&F neuron combines all signals into a single output. The I&F neuron has a threshold that is sufficiently low such that the neuron generates a spike whenever it receives a spike. The speed of the phase shift determines the frequency of the time-multiplexing and also how much time is dedicated to each channel.

Several column circuits are combined into a set in order to cover the entire space of the input image. Figure 6.2 (right) shows this design. Each column circuit multiplexes the pixels from an image column. The width of a column circuit equals the number of rows in the image. Alternately, the image can be transposed in order

BUPT

to do a sweep along the rows. The number of signals that are sent to the LSM is reduced to the number of columns (or rows).

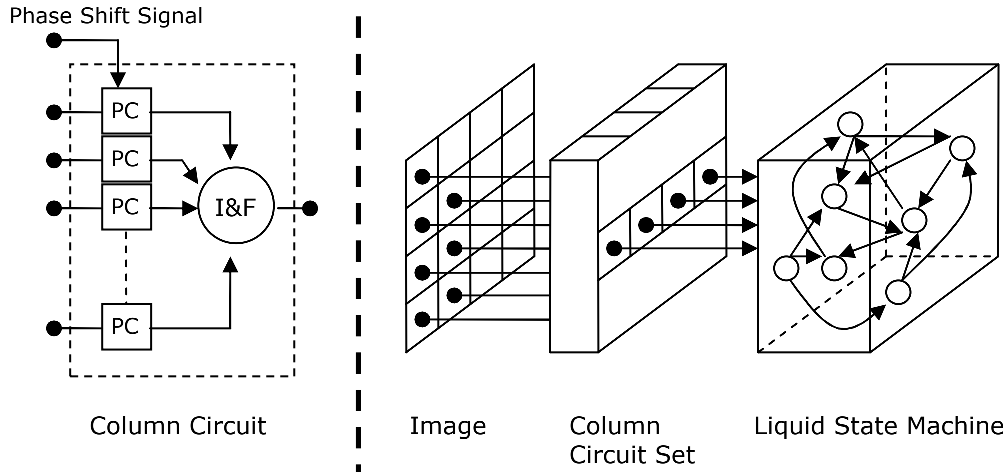


Fig. 6.2. Time-Multiplexing the Input Signals

One possible drawback of this method could be the fact that the LSM does not have access to the entire amount of information at the same time. However, the concept of liquid computing is based on the idea that the liquid medium is able to store past information in its high dimensional internal state. This means that if the frequency of the multiplexing is not too low, the LSM should be able to do computations on the entire image even though it has access to only part of the information at a time.

Figure 6.3 shows the circuit that performs the actual multiplexing (the pixel circuit PC). Central to this circuits is the resonate and fire neuron (R&F). Figure 6.4 (right) shows how the membrane potential of the R&F neuron is affected by the spikes of the phase shift signal. Initially, the membrane potential is in resting state which is zero. Consider that the reference spike occurs at time t_{sp1} and the variable phase spike occurs at t_{sp2}. The membrane potential is represented in the complex plane by *z(t)*. At t_{sp1} *z* changes from zero to *S*, where *S* is the strength of the input synapse of the R&F neuron. The membrane potential starts to oscillate according to (6.2), where ω_{RF} is the pulsation of the oscillation. Because the R&F neuron was resting before the first spike we can consider $t_{sp1} = 0$.

$$z(t) = S\exp(i\varphi) = S\exp(i\omega_{RF}t) \tag{6.2}$$

As presented in chapter 2, the timing of the second spike relative to the phase of the oscillation determines whether the membrane potential reaches the threshold or not. The neuron is most sensitive to spikes occurring at times when ωt is a multiple of 2pi. Therefore, as presented in figure 6.4 (left) and by equation 6.3, the pulsation ω_{RF} of the R&F neuron should be set such that the timing of the second spike t_{sp2} synchronizes with the phase of the oscillation at 2kpi.

BUPT

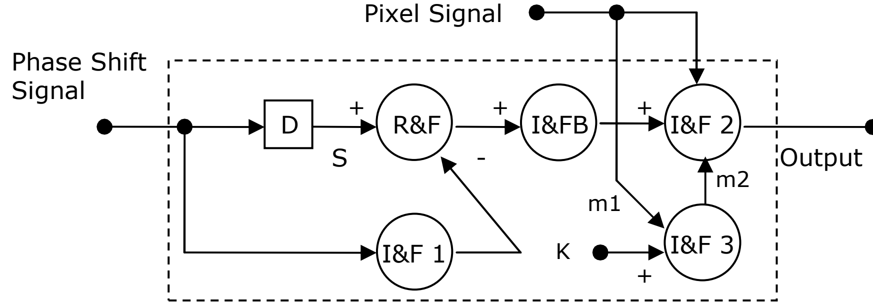


Fig. 6.3. Pixel Multiplexing Circuit

In addition, the threshold of the neuron should be set such that the R&F neuron is still sensitive to spikes that occur during a time window of Δt_{sp2} centered on t_{sp2}. Equation 6.4 can be written by examining the geometry of figure 6.4 (right). It shows how to calculate the threshold *TH* in order for the membrane potential *z* to exceed the threshold if the phase of the second spike is in range ±φ around 2pi.

$$\omega_{RF} = \frac{2k\pi}{t_{sp2}} \tag{6.3}$$

$$\cos\frac{\varphi}{2} = \frac{|z(t_{sp2})|}{2S}, |z(t_{sp2})| \geq TH \Rightarrow TH < 2S\cos\frac{\omega t_{sp2} - 2k\pi}{2} \tag{6.4}$$

$$TH_{RF} = 2S\cos\frac{\omega_{RF}\Delta t_{sp2}}{4} \tag{6.5}$$

Equations 6.3 and 6.5 design an R&F neuron that is sensitive to a pair of spikes that have a specific timing difference of $t_{sp2} \pm \Delta t_{sp2}/2$. The phase difference of the two control spike trains shifts constantly and so at same point the second spike will be inside the time window of the R&F neuron (fig. 6.5).

Note that it is assumed that the R&F neuron has a resting membrane potential before receiving the reference spike. It is the job of neuron I&F1 to assure that this happens. The threshold of this neuron is set such that it generates a spike for every three spikes that it receives. Its output is inhibitory and serves as a reset mechanism that brings the membrane potential of the R&F neuron to zero.

Note that the R&F does not fire a spike immediately after receiving the second control spike at $t_{sp2,}$ even if the timing of the spike is correct. It continues to oscillate for a short time (less than pi/2) until the membrane potential exceeds the threshold (also seen in fig. 6.5). If neuron I&F1 triggered the reset after two spikes instead of three it would risk bringing the R&F neuron to resting potential prematurely, preventing it from reaching the threshold. This is fixed by triggering the reset after three spikes and slightly delaying the control signal for the R&F neuron. This way, when the third control spike (a reference spike) reaches the R&F neuron it would have already caused neuron I&F1 to trigger the reset.

BUPT

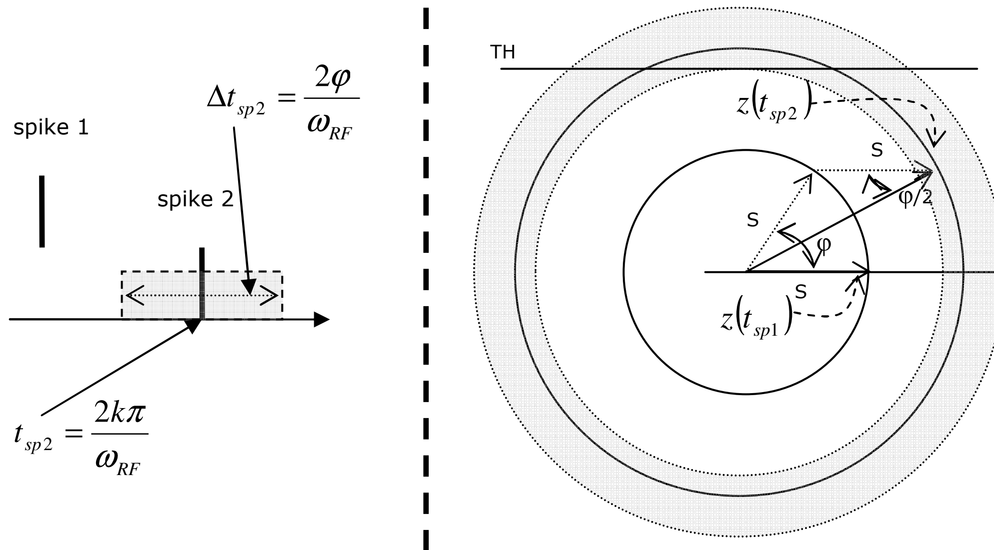


Fig. 6.4. Membrane Potential of R&F Neuron

Neuron I&F2 integrates the pixel signal and then generates an output spike train that rate-codes the pixel information. The threshold and loss factor of I&F2 are set high, and so the neuron is unable to reach the threshold with just the pixel signal at the input. When neuron I&F2 also receives spikes from neuron R&F its energy is boosted and the neuron starts to fire. This is equivalent with temporarily lowering the threshold. The period of the output spike train T_o is a function of the pixel P as shown in eq. 6.6 and 6.7. The value B is the average energy value produced by the boosting spike train. Note that the output of the resonate and fire neuron R&F is not used directly as a booster signal. Instead, an integrate and fire with burst I&FB is interposed between neurons R&F and I&F2. The reason is the following: in order for the boosting spike train to be well approximated by its average value B, its period needs to be much smaller than the operating time constants of neuron I&F2. Neuron I&FB has the duty to generate a burst of spikes for every spike generated by neuron R&F. This avoids a situation where the pixel circuit operates at a frequency a lot higher than that of the Liquid State Machine. The concept is also depicted in figure 6.5.

$$T_o(P) = -\frac{1}{k}\ln\left(1 - \frac{kTH}{P+B}\right) \tag{6.6}$$

$$\frac{P_{\max}}{k} \leq TH < \frac{P_{\min} + B}{k} \tag{6.7}$$

Equation 6.6 gives the period-pixel dependency and therefore defines the rate-code. It can be noticed that the code is not linear. This is not necessarily a problem but we would prefer to have a linear coding rather than a logarithmic one. Neuron I&F3 has the duty to linearize the transfer function as much as possible. The

BUPT

neuron is fed by the pixel signal *P* via a static synapse connection *m1* and also by a constant signal *K*.

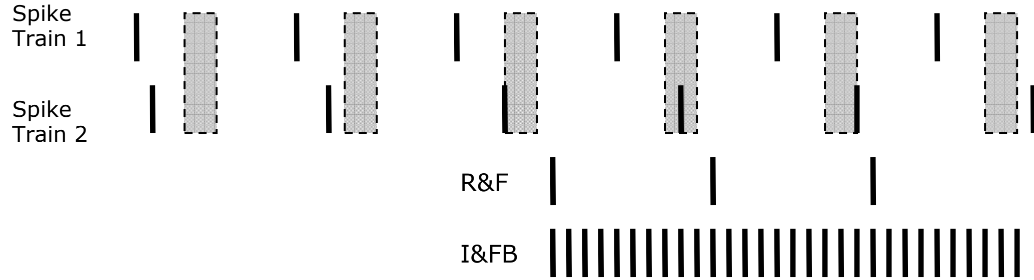


Fig. 6.5. Activation of R&F Neurons

The output of I&F3 is sent to neuron I&F2 via the synaptic connection *m2*. The expected functionality of the circuit is the following. For high values of *P* the rate code generated by neuron I&F2 is almost linear. Therefore, we want neuron I&F3 to have little or no influence in I&F2. This is achieved by making connection m1 inhibitory. For low values of *P* the inhibition of m1 is small and neuron I&F3 activates and starts firing. The average value of the spike train generated by I&F3 partially compensates the non-linearity of I&F2. The values of *m1* and *m2* are subject to optimization. Figure 6.6 shows the performance contour obtained for the scoring function in eq. 6.8. As expected, the best performance is obtained for a negative connection m1 and a positive connection m2.

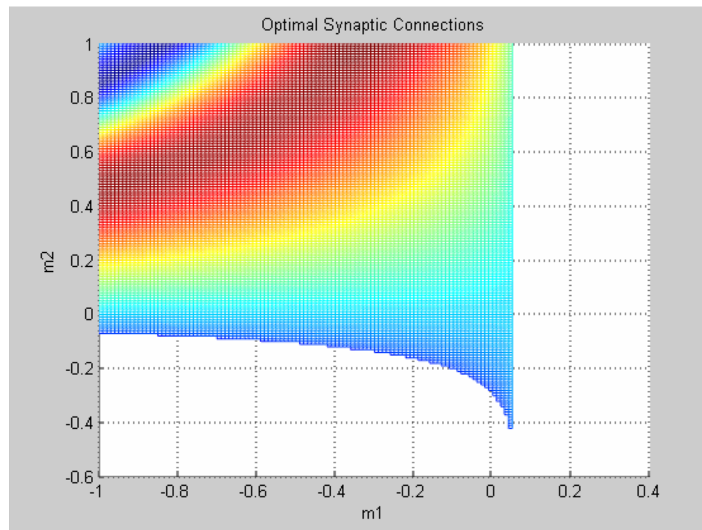


Fig.6.6. Optimization of m1 and m2 synaptic connections

$$Score = \frac{\max(T) - \min(T)}{\sqrt{\frac{1}{N}\sum_{k=1}^{N}\left[T(P_k) - \overline{T}(P_k)\right]}} \tag{6.8}$$

The scoring function in eq. 6.8 is a ratio of the range of the output period T and the mean square error between function T and its linear approximation $\overline{T}$ (linear regression). This way we seek a transfer function that is as linear as possible and that also preserves the output dynamic range. Figure 6.7 shows the results of the optimization. The blue trace represents the original transfer function (rate-code) as given by eq. 6.6. The green trace is its linear approximation obtained with linear regression. The dashed red trace is the corrected transfer function. This transfer function is obtained for the optimal pair (m1, m2) and is the most linear transfer function achievable with this architecture. Of course, adding additional correction neurons could improve the linearity.

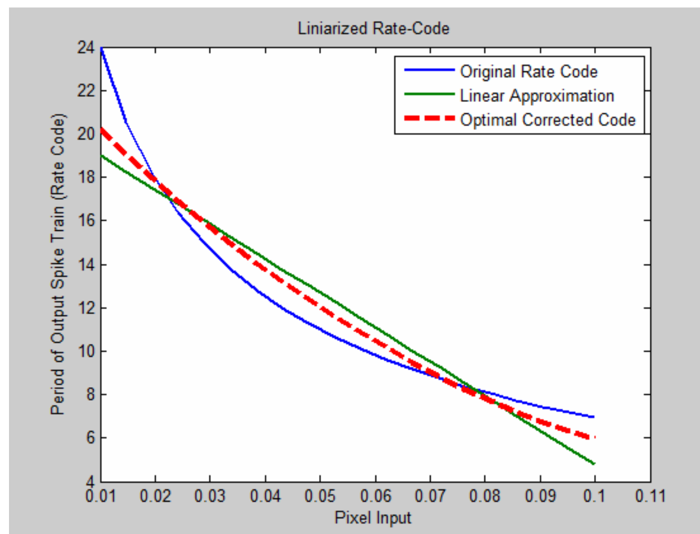


Fig.6.7. Linearized Rate-Code

6.3.2. Spike Generator with Shifting Phase

As presented in the previous paragraph the resonator circuit requires two control spike trains that have a shifting phase difference. Figure 6.8 depicts such signals. The first spike train has a steady period T. The second spike train has an initial phase difference and also a period that is roughly higher than T. Therefore the second spike train has a phase that will constantly shift away from the phase of the first spike train. The minimum and maximum phase differences are given by the phase shift control window. When the phase of the second spike train reaches the end of the phase shift window its period is decreased and so the process is reversed. The continuously changing time difference between the two spike trains can be used to activate selectively the resonate and fire neurons. Figures 6.9 and 6.11 show two I&F neural circuit implementations that generate the signals described above. The implementation shown in figure 6.9 has more neurons. However, the functionality of each neuron is simpler and more biologically plausible compared to figure 6.11.

BUPT

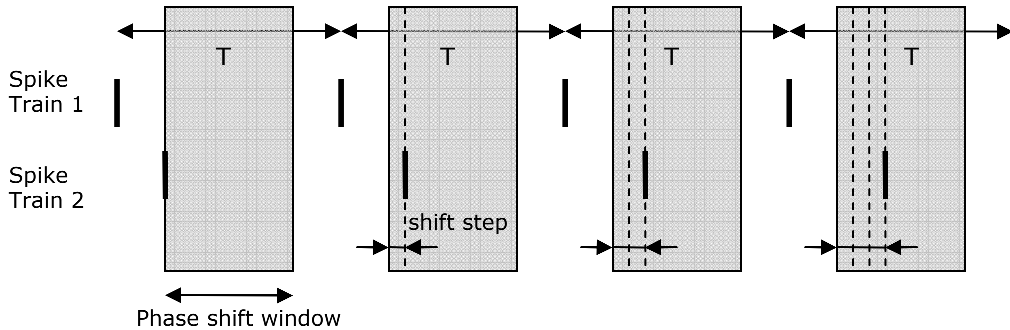


Fig. 6.8. Shifting Phase Spike Trains

Neurons 1, 2, 3 and 4 integrate the same analog stimulus and therefore generate spike trains that have the same frequency. The initial membrane potentials of the four neurons are evenly spread in their dynamic range and so there's a T/4 phase shift between spikes. Neurons 1 and 2 are responsible for generating the odd spikes of the first and second spike train respectively. Neurons 3 and 4 generate the even spikes. Neurons 5, 6, 7 and 8 are gate neurons. This means that they have a sufficiently low threshold that they fire a spike whenever they receive a spike, as long as there is no inhibition signal.

Neurons 9, 10, 11 and 12 are I&F neurons that have a very high loss factor. Their threshold is set such that two input spikes are able trigger the neuron as long as the two spikes both arrive within a given time window. The value of the loss factor controls the size of the time window. Figure 6.10 presents this concept (for simplicity the three spikes are abbreviated S1, S2, and S3).

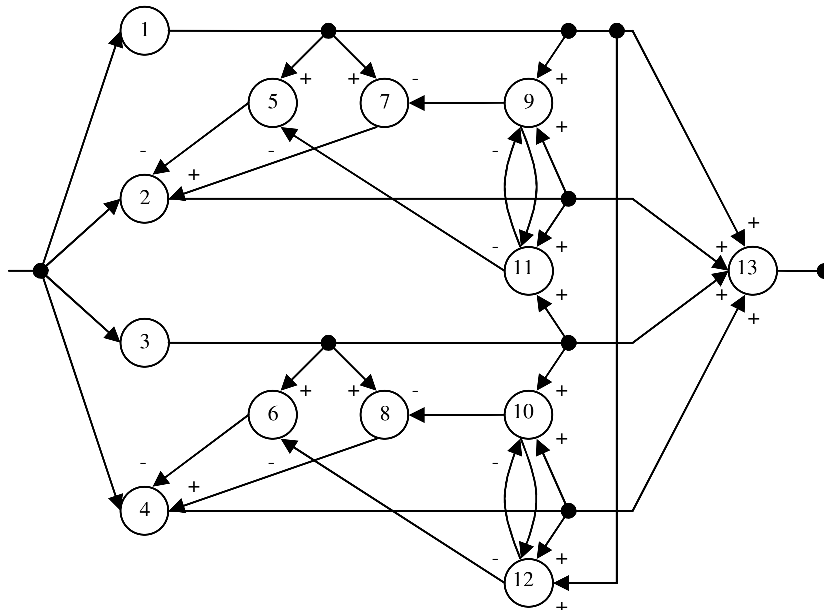


Fig. 6.9. Shifting Phase Circuit 1

BUPT

Initially, the timing of S2 is inside the time window of both neurons 9 and 11, and would potentially cause both of them to fire. However, because neuron 9 fires first (it receives S1 and S2; neuron 11 receives S2 and S3), it will send an inhibitory spike to neuron 11 forcing it not to fire. As a result, neuron 7 is inhibited and is deactivated while neuron 5 remains active. The purpose of neuron 5 is to retransmit the spikes of neuron 1 back to neuron 2 as an inhibitory feedback. This feedback delays spike S2 and creates the phase shift effect.

The speed of the phase shift is determined by the strength of the synapse that connects neuron 5 and neuron 2. When spike S2 exits the upper time window, neuron 9 will stop firing and neuron 11 will start firing (because there is no more inhibition). This deactivates neuron 5 and activates neuron 7. As a result the negative feedback of neuron 2 is replaced with a positive feedback that accelerates spike S2 and gradually decreases the phase difference. When spike S2 re-enters the time upper time window neuron 9 is ready to fire again. However, it does not fire because now it is inhibited by neuron 11. Finally, neuron 13 combines the four signals into a single spike train that is ready to stimulate the R&F neurons of layer 2.

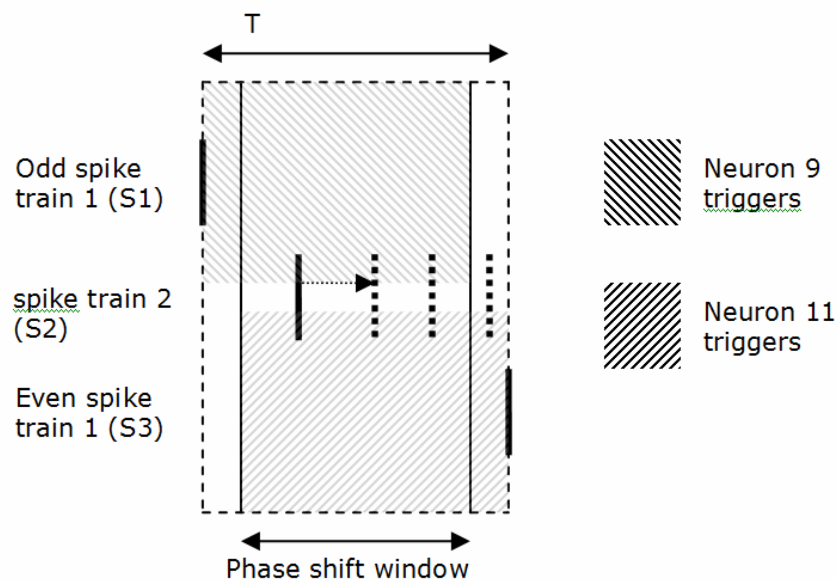


Fig. 6.10. Selective Firing Window

The circuit in figure 6.11 has the same functionality. The difference is that it uses a pair of I&F neurons as context detectors (CD). The context detectors are used for determining the moments when spike S2 crosses the borders of the phase shift window. Because the context detector is sensitive to the timing between spikes but also to the order of spikes (S1, S2 is not the same as S2, S1) the spike trains do not need to be separated on odd/even channels. Therefore, only 2 neurons are needed for generating the two spike trains (neurons 1 and 2).

When context detector CD1 fires a spike it triggers neuron 5. Because this neuron has a positive feedback connection it will keep firing until it is inhibited by CD2. Similarly, CD1 inhibits neuron 6. This mechanism toggles activation of gate neurons 3 and 4 and creates the phase shift effect as explained for circuit 6.2.

BUPT

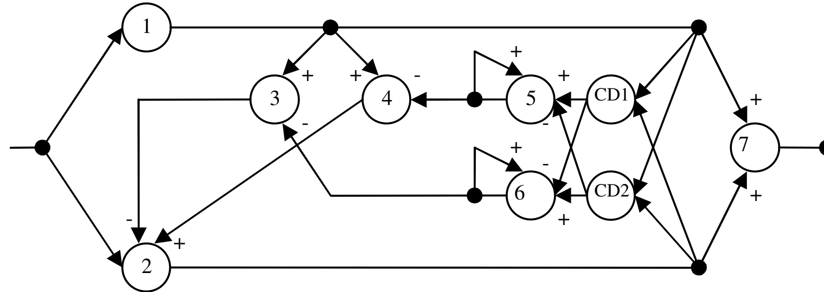


Fig. 6.11. Shifting Phase Circuit 2

6.3.3. Estimating Gabor Coefficients with Liquid State Machine

As presented in 6.3.1 there are two types of input signals. The first type, which we will call StaticInput, corresponds to the case when the entire image is sent to the liquid state machine. The second type, named DynamicInput, is the time-multiplexed signal. If the multiplexing is done along the columns the signal is called DynamicInputX; if the multiplexing is done along the rows the signal is called DynamicInputY. The period of the time-multiplexed signal is 1s. This means that it takes 1s to completely scan an input image and produce the input signal. After 1s the input signal is repeated. It takes at least one period until the entire information of the input image is available to the Liquid State Machine. Figure 6.5 presents some examples of spiking activity of the liquid medium when it is stimulated by input signals.

We set up 4 readout units that are connected to the liquid medium by static synapses. Each readout unit is responsible for estimating the Gabor coefficient corresponding to filters with 4 different orientations: 0°, 45°, $90\ ^o$ and $135\ ^o$. All filters have the same spatial frequency with wave number k = pi/2. The performance of the Liquid State Machine is tested with both types of implementations for the readout units: parallel perceptron (PP) and multilayer feed forward networks (MLFF).

In order to train the readout units and then test the Liquid State Machine, training and testing data had to be gathered. A set of 30 random images are selected for the experiment. For every image, 20 different points are chosen by generating their coordinates randomly. A window of 11x11 pixels is centered on each random coordinate, thus generating 600 smaller input images. For every input image the target Gabor coefficient t_G is calculated with eq. (6.1). Because filters with 4 different orientations are used, the target data will be 4x1 vectors.

The input images are converted to input signals according to 6.3.1 which are used at stimulating the liquid medium. The spiking activity of the liquid medium is recorded and converted to a set of liquid states as presented in 5.3. The liquid states are real Nx1 vectors, where N is the number of neurons of the liquid medium, and they represent the input of the readout units p_L. The (p_L t_G) data is divided into two sets of 400 pairs and 200 pairs used at training and testing accordingly. The training algorithm further divides the training set into training and validation. During training, the error of the approximation is a vector is considered to be the difference between the outputs of the 4 readout units and the target vector t_G. Each readout unit is trained independently from the others and tries to minimize one component of the error vector.

BUPT

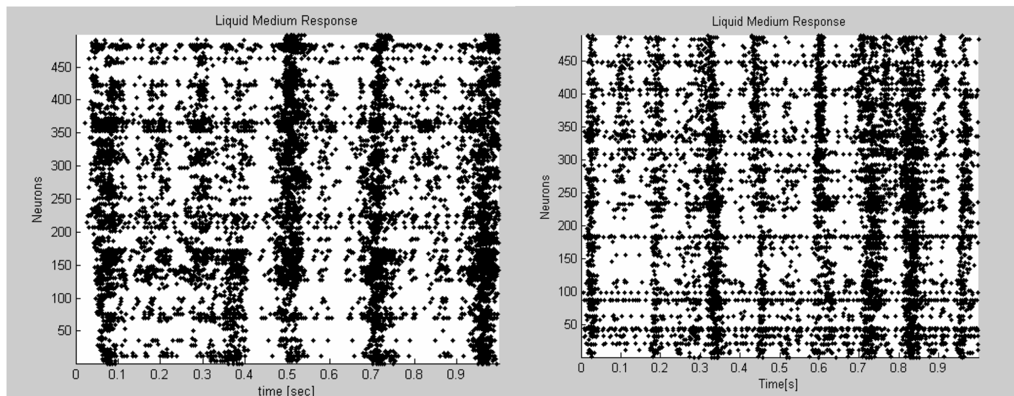


Fig. 6.12. Spiking Activity of Liquid Medium

The two types of implementation of the readout units (PP and MLFF) are tested separately. After the readout units are trained, the Liquid State Machine is tested with images from the testing set. Figure 6.6 presents an example of the functionality of a trained LSM. In this case the LSM is stimulated by 10 input images, the input being changed every 6 seconds. The red dashed line marks the value of the target Gabor coefficient (0° filter). The blue trace is the output of the LSM. It is important to notice that the LSM is not able to approximate the Gabor coefficient well immediately after the new input image is available.

A certain amount of time is needed until the LSM manages to process and accumulate enough useful information in its liquid states. Additionally, because the liquid medium behaves like a filter that retains information, it takes some time until the information of the old input image fades away.

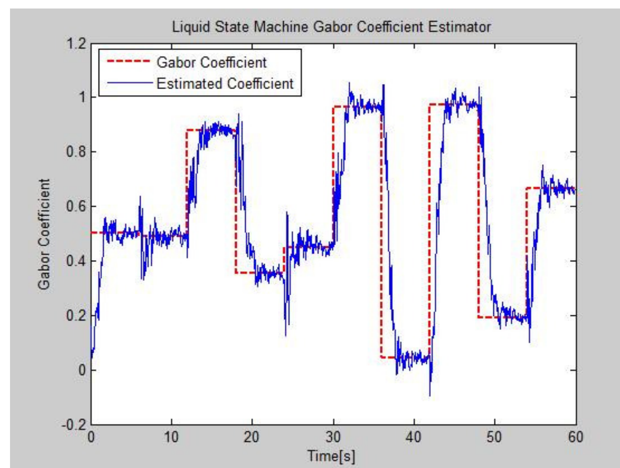


Fig. 6.13. Estimating Gabor Coefficients

6.3.3.1. Approximation Accuracy

The approximation error is computed as the absolute difference between the output of the readout unit and the actual Gabor coefficient from the target set. When stimulated by a new set of input signals, the Liquid State Machine will have a higher approximation error at the beginning. Afterwards, if the LSM is able to

BUPT

perform useful computations, the absolute error should have a tendency to decrease exhibited by $e(t)$. In order to quantify the approximation accuracy e_{min} of the LSM we determine the minimum value $e(t_m) = min(e(t))$ as long as for any moment of time $t > t_m$ no more than K percent of the error samples $e(t)$ are higher than $e(t_m)$. In this case t_m is considered to be the time required by the LSM to reach its best approximation accuracy. A small error can either be the outcome of a good approximation or the effect of random noise. The measurement technique presented above is insensitive to noise and determines the approximation performance of the LSM. An example is presented in figure 6.14. Note that the Gabor coefficients are scaled [0 to 1] and so the approximation accuracy e_{min} is expressed in percentages.

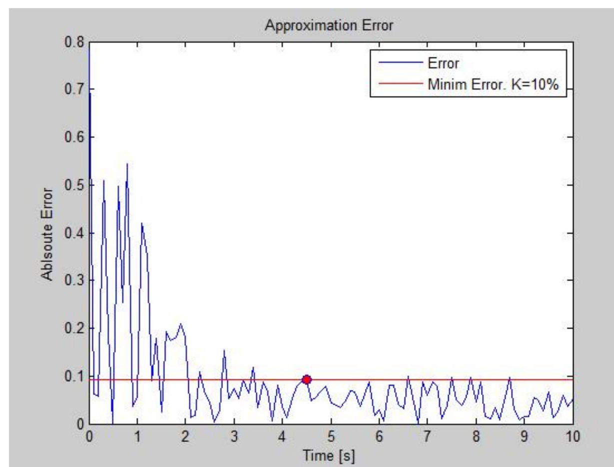


Fig. 6.14. Computing the Approximation Error

6.3.3.2 Approximation Speed

As explained in 6.3.3 the LSM requires some time until it can reach its best approximation accuracy. This time can also be considered a performance measure in addition to approximation accuracy. In order to quantify it, we introduce three additional measurements: $t_{10\%}$, $t_{20\%}$ and $t_{50\%}$. It is considered that for $t>t_{P\%}$, $e(t)$ verifies condition (6.9) for more than its K samples, where K is the same as for $e_{min.}$

$$e(t) < \left[\left(1 - \frac{P}{100}\right)e_{\min} + \frac{P}{100}\max(e(t))\right] \tag{6.9}$$

While $t_{10\%}$, $t_{20\%}$ are meant to measure how fast does the LSM reach its best accuracy, $t_{50\%}$ measures how fast is the LSM able the fade away the information of the previous input image.

6.3.3.3. LSM Performance

Table 6.1 presents the performance results for three different LSM networks equipped with MLFF readout units. All types of signals (Static, DynamicX and DynamicY) are tested. Every group of 3 rows in the table shows the performance results of one LSM network. Each LSM has different network parameters that are optimized to different performance criteria. The optimization is performed by using

BUPT

a genetic evolution method [87]. The vector of parameters that is subject to optimization is shown in eq. 6.10.

$$\overbrace{topX \quad topY \quad topZ}^{Network \quad Parameters} \quad \overbrace{TH \quad kLoss}^{Neuron \quad Parameters} \quad \overbrace{C \quad \lambda \quad INH \quad IPER}^{LSM \quad Paramters} \quad \overbrace{D_{max}}^{Synapse \quad Parameters} \qquad (6.10)$$

The performance criteria for the three optimizations are: e_{min} for LSM1, $0.5^* t_{10\%} + 0.5^* t_{20\%}$ for LSM 2 and $t_{50\%}$ for LSM3. The optimization criterion is also underlined by the curly border of the table cell. The numbers shown in table 6.1 are computed by averaging the results of 600 simulations performed on 30 different LSMs (20 different input stimuli for each LSM). In addition, the performances of the 4 readout units (one for each angle) are also averaged. All LSMs have the same parameter vector (6.10). Nevertheless, because some of the vector components are statistical parameters (the LSM parameters), the 30 LSMs are different.

Table 6.1. Feed-Forward Multilayer Readout Unit

	Signal Type	Best Aprox. e_{min} [%]	Time to e < $t_{10\%}$ [s]	Time to e < $t_{20\%}$ [s]	Time to e < $t_{50\%}$ [s]
LSM 1	Static	8.2%	2.85	2.04	1.06
	DynamicX	6.2%	2.91	1.98	1.10
	DynamicY	6.8%	2.97	2.03	1.07
LSM 2	Static	7.8%	2.75	1.94	1.05
	DynamicX	6.5%	2.69	1.89	1.03
	DynamicY	6.9%	2.65	1.84	1.01
LSM 3	Static	13.2%	2.92	1.88	0.97
	DynamicX	9.1%	2.88	1.92	0.93
	DynamicY	9.3%	2.93	1.89	0.89

The first general observation is that all LSMs perform better for the DynamicX and DynamicY input signals compared to the Static signal. Secondly, even though LSM1 and LSM2 are optimized for different performance criteria, they both perform well and similar for both criteria. For LSM3 the approximation accuracy e_{min} is up to 5% worst while the $t_{50\%}$ is decreased with up to 10%. It is interesting to notice that both LSM1 and LSM2 have neuron loss factors *kLoss* between 0.07 and 0.12. On the other hand LSM3 has a lost factor between 0.16 and 0.19. The higher loss factor gives the LSM the ability to “forget” the old information faster, and so it is able to deal better with transitory regimes.

Table 6.2 presents the approximation accuracy of LSM1 when the network is equipped with readout units implemented by a parallel perceptron.

Table 6.2 Parallel Perceptron Readout Unit

Signal Type	Best Aprox. e_{min} [%]
Static	11.5%
DynamicX	9.2%
DynamicY	8.9%

BUPT

The parallel perceptron uses enough neurons such that the approximation accuracy of the LSM is not limited by the quantization ability of the readout unit. Anyway, it can be noticed that the PP readout unit does a poorer approximation of the liquid states compared to the MLFF readout unit. However, the training of the PP readout unit is a lot quicker because the p-Delta algorithm does not need to compute the derivatives of backpropagation.

BUPT

7. CONCLUSIONS AND CONTRIBUTIONS

7.1. Conclusions

This thesis proposes a novel approach to information processing using neural networks. The subject is particularly interesting because spiking neural networks are used instead of traditional models. These are considered to be the third generation of neural networks and are the models most similar to the biological neurons. The new spiking models preserve all properties of previous models by using the frequencies of the spike trains to encode information.

7.1.1. Spiking Neural Networks

The first chapter of the thesis presents up-to-date studies regarding the potential applicability of spiking neural networks and the motivation to choose such models. The second chapter presents the relation between spiking models and the biological neuron. The dynamic functionality of the synapse is presented and a simplified mathematical model (with variable internal state) is proposed {1}[9]. An important observation is made about the fact that the internal state of the synapse converges towards a constant when the pre-synaptic spike train is of constant frequency. This observation creates the opportunity for an important simplification inside the simulation framework. The simulator divides the neural network into areas and monitors the average spike frequency of each area. Afterwards, it uses the average frequency to estimate the internal state of the synapses in each area. Consequently, the internal state of each synapse must not be recomputed at each simulation time step. It will only be updated periodically as a response to average frequency changes. The second chapter also studies the structure of the neuron cell. First, the functionality of a single ion channel is introduced. Then, several possible configurations of ion channels are presented, each leading to a different dynamic behavior for the neuron cell. The analysis concludes with a selection of four neuron models that are considered to be most useful for simulation purposes {1}.

The second chapter concludes by analyzing some information encoding techniques and also the ability of a spiking neural network to implement a content addressable memory with complete and incomplete spike contexts {2}.

7.1.2. MATALB simulation framework

Chapter 3 presents a MATLAB simulation framework for spiking neural networks together with all tools necessary for analyzing and visualizing results {7}.

7.1.3. Parallel MATLAB framework and GPU accelerated framework

The parallel implementations performed in the third and fourth chapters show that spiking neural networks are ideal candidates for parallel implementations.

[9] {*n*} associates this conclusion to contribution *n* in paragraphs 7.3 and 7.4

Neural networks in general are structures with an extremely high level of parallelism. The first attempt to parallelize the model uses a distributed MATLAB model that runs on several computers connected by a network. The attempt is successful and proves the potential of a parallel implementation. However, the improvement is not satisfactory considering the amount of available computational power and the difficulties encountered while setting up the simulator. The second attempt to parallelize the model uses a GT9800 NVIDIA GPU. This time the implementation is much more successful and achieves significant speed-up. In addition, further improvements are made to the model like: minimizing number of branches, eliminating unnecessary kernel overhead time by merging kernels, improving data access time by storing the data delay line in shared memory, reducing the transfer time of result by overlapping computation and data transfer. Chapter four concludes with a brief presentation of the MEX interface that connects the MATLAB application to the compiled CUDA C simulator.

7.1.4. Liquid State Machines and p-Delta Learning Rule

While the structure of the biological neurons is very well known, the exact architecture and computational models of the brain is still a mystery. Chapter five presents an architecture called Liquid State Machine that is able to perform dynamic data computation without previous knowledge of process or of the data itself. The structure of this computing machine is not deterministic and is generated by statistical rules. This aspect concludes that such architectures do not need to be designed and can easily be built using genetic evolution of their statistical parameters. When being stimulated by external signals the Liquid State Machine generates an un-stable sequence of states (neural activity) that is stimulus specific. A trained read-out unit is used to map the neural activity into a desired output function.

Chapter five also presents two possibilities of implementing the read-out units: the parallel perceptron, and the feed forwards sigmoid networks; accompanied by suitable training algorithms: the p-Delta rule, and backpropagation. In the case of the p-Delta rule chapter five also presents three improvements that lead to a better convergence rate and a higher convergence speed: adaptive learning rate, not-greedy neuron competition, and a mechanism for adapting the noise margin {2}. The improvements are the result of a thorough investigation of the dynamics of the training algorithm. Several scopes were attached to different variables that change during training. Analyzing the time traces of these variables uncovers the reasons why training fails to converge or does so very slowly.

7.1.5 Extracting Gabor Coefficients from images using Liquid State Machines based on Spiking Neurons

The concept of processing information using Liquid State Machines as the main computational core is proven by successfully extracting Gabor coefficients from images at different fiducial points. The Liquid State Machine manages to approximate the Gabor coefficients well without any pre-processing of the pixel data. Both types of readout units (parallel perceptrons and multi-layer networks) prove to have similar performances from the error approximation point if view. From the computational perspective the parallel perceptron and the p-Delta training algorithm is less demanding. The parameters of the Liquid Medium are optimized using a genetic optimization. This approach was appealing because it is very likely

that the biological networks have evolved in a similar way. In addition, the genetic optimization can deal well with the large parameter space and the highly dynamic performance function that would be impossible to optimize with gradient methods.

7.2. Future Work

The ultimate goal of this research is to design a system that can perform face recognition by using neural networks exclusively. So far, a good architecture (LSM) that can perform universal computation has been chosen. The architecture has been implemented to run on a GPU in order to accelerate simulation, and an application that estimates Gabor coefficients has been designed using LSM.

In the future we plan to design subsequent parts of the system that use the estimated Gabor coefficients to perform face recognition [109], [110]. Also, we desire to use a feedback technique that helps the Gabor estimator at selecting the most significant fiducial points of the image rather than using a uniformly spaced grid. This technique will use adaptive coordinates for the fiducial points and will probably be biologically unfaithful. This is because the brain is able to recognize a face in just fractions of a second and it is unlikely that there is enough time to use an adaptive method for selecting fiducial points. Alternately, it performs analysis of the entire image in parallel and selects the relevant information in subsequent processing layers. However, because of our limited computational resources, and because we are not necessarily aiming towards a real-time recognition system, an adaptive solution will be a good compromise. In fact the performances are predicted to be similar with only speed being altered. So far, our research has only restricted to face recognition. However, there are several other biometric recognition applications that can use the liquid state machine architecture. The remaining of this paragraph presents such applications that are possible future research tracks.

One of the research directions that we are aiming towards is affective computing. This differs from face recognition because we are interested in classifying the facial expression of a person rather than determining its identity. More information about affective computing can be found in [105], [106], [107] and [108]. An aspect that is very interesting and that deserves investigation is the potential ability of a liquid state machine to unify face recognition and affective computing by using a single liquid medium and multiple readout units. This is potentially possible because the two applications are similar in terms of input signals (Gabor coefficients). Therefore, it is the duty of the liquid medium to decompose the information and isolate aspects that are significant to either face recognition or affective computing.

Probably the oldest biometric identification technique is fingerprint identification. In [97] Elmir and Zakaria present an application that uses a liquid state machine for fingerprint classification. It is very appealing that they also use Gabor filtered images to obtain the stimulus for the LSM and so facilitate unification with the techniques presented above.

Another application that we are considering is human gait recognition, which is also a biometrical recognition problem [98], [111]. In contrast to the face recognition problem where the stimulus is static (or artificially dynamic as in the case of the swept input), the human gait analysis inherently comes with dynamic signals. This type of stimulus is better fitted to the functioning principle of the liquid state machine. The goal for this application is to use key points on the human body that form a set that is significant to human movement (i.e. feet, knees, elbows, shoulders, center of head). The trajectory of each point will be spike coded in terms

BUPT

of absolute or relative changes of coordinates. The goal of the liquid state machine is to analyze the trajectory of each key point and also combine the set of movements into a unified gait characteristic. Such an application together with face recognition and affective computing can have numerous applications starting from human-machine interfaces serving visually impaired people ([99], [100], [101], [102], [103], [104]), biometrical identification, automotive or surveillance.

Another question that remains unanswered is what parameters of the liquid state machine architecture are most significant; and also to what aspects of the information processing task. So far, parameters have only been improved by means of evolutionary methods without having a rigorous understanding of their influence.

Although simulation speed has been greatly improved by using a GPU there is always desire for more. Presently, the achievable network sizes that can be simulated in reasonable amounts of time are still by far smaller than the biological counterparts. A grid super-computer would be an improved, thus expensive solution. Another less expensive approach is to use multiple GPUs, a feature that is offered by NVIDIA's CUDA C. Global memory access time is a bottleneck for a single GPU and so combining several GPUs will raise additional communication issues. The best guess is that an application will have to be divided into several LSMs that do not directly interact; each LSM will run on an individual GPU. It will be the duty of a sub-sequent layer (a LSM, readout unit or another type of neural network) to combine the neural activities of the multiple LSMs and do a unified analysis. This is analog to the brain containing several clusters, each dedicated for a sub-task, without being interconnected to the other clusters (obviously a simplification).

7.3. Theoretical Contributions

The theoretical contributions are as follows:

1) Improved the p-Delta learning algorithm useful at training the parallel perceptron. This results in an algorithm that has a significantly higher convergence rate, convergence speed and stability. The improvements are:

- An adaptive learning rate
- A not-greedy competition between neurons. This prevents the algorithm from getting stuck due to situations when a single neuron wins the competition exclusively.
- A mechanism for adapting the targeted noise margin and the noise margin learning rate. A momentum effect is introduced in the adaptation method in order to make it insensitive to noise. This offers the advantage that the noise margin (which is data dependent) does not need to be pre-computed.

2) A 4 step thread synchronization procedure was defined. The procedure is useful for synchronizing all threads of the application regardless of the block/thread design. This contribution is very useful at merging all kernels of the application into a single one.

3) A study is made regarding the ability of the integrate and fire neuron to implement a context detector and a content addressable memory. In addition, an analysis is made to determine the relationship between the size of the spike context and the ability of a content addressable memory to correctly recall a spike sequence.

BUPT

4) An analysis was performed and it was determined that the spiking neural network simulator is parallelizable. Additionally, this conclusion is tested with the MATLAB parallel model.

5) The thesis proposes a novel approach to perform filtering by using a Liquid State Machine based on spiking neurons. Gabor coefficients of different orientations computed around fiducial points of an image can be accurately estimated using the proposed architecture.

7.4. Practical Contributions

6) Implementation of MATLAB simulation framework useful at simulating spiking neural networks. Individual neurons can use one of the following models: integrate and fire, integrate and fire with adaptation, integrate and fire with burst, and resonate and fire. The model uses dynamic synapses.

7) Implementation of tools useful at:

- Generating the network and synapse parameters required for creating a Liquid State Machine
- Visualizing the structure, connectivity, neural activity and separation property of a Liquid State Machine
- Visualizing membrane potential traces and spiking traces
- Visualizing the neural activity of large networks as images by time averaging the spike trains
- Displaying the synchrony between neighboring neurons.

8) Implementation of Parallel MATLAB simulation framework by distributing the workload on several computers connected by a network.

9) Implementation of parallel GPU CUDA model for Spiking Neural Networks. The implementation is written in CUDA C and accelerates the model dramatically making fast simulation and iterative simulation a reality.

10) The BasicObject Class was created. This class allows easy transferring of objects from host memory to GPU memory. When an object contains internal arrays or internal sub-objects a transfer cannot be done by directly using the transfer functions provided by CUDA C. The BasicObject Class contains lists of pointers and transfer methods necessary for a correct and complete object transfer regardless of the object structure.

Several improvements were made to the initial GPU simulation framework. Most significant are the following:

11) Minimizing number of branches. The code was developed such that it uses as few braches as possible. This prevents the threads from having a divergent execution path and the hardware from serializing execution.

12) Merging kernels. The reduced number of kernels eliminates the kernel launch overhead time. This time can add up to a significant percentage out of the total simulation time and therefore the improvement is significant. Merging the kernels was possible after rearranging the model such that several synchronization barriers were eliminated.

13) Using the shared memory for storing the Data Delay Lines. Using of shared memory was possible after merging all kernels into a single one. The result is a significantly reduced data access time. This is because shared memory is on

BUPT

chip and is as fast as register if a bank conflict does not occur. In addition, the DataLine objects were organized such that the data access generates as few bank conflicts as possible.

14) The time required for transferring the simulation results back to the host was drastically reduced by overlapping computation with data transfer. This is possible by using the hardware abilities of the GPU which is a stream processor. Also, a study is made to determine the optimal fraction for segmenting the computation. This is the fraction at which the time gained from overlapping the data transfer equals the overhead time cause by the additional kernel launches.

15) An analysis was made which determined that using the Constant Memory and Texture Memory caches will not improve data access time for the Spiking Neural simulator. This is because the address access pattern is uniform and causes a low cache hit rate.

16) A communication interface was build between MATLAB and the compiled CUDA C GPU model. The interface makes use of the MATLAB MEX files. This interface is particularly important because it allows the user to manipulate and visualize GPU simulation results in MATLAB.

17) A time-multiplexer has designed using I&F, I&FB and R&F spiking neurons. The multiplexer reduces the dimensionality of the input signal.

Other Contributions:

18) A study was performed regarding the dynamic behavior of different biological neurons. In addition, a few spiking neuron models were chosen to be most appropriate for simulation. It is believed that the chosen models make the best compromise between model complexity and computational demand.

7.5. Publications List

- R. Mirsu, T. Hentea and D. Gray, "Optimization for Hybrid Vehicles", Proc. of the 12th WSEAS International Conference on Computers, Heraklion, Greece, pp. 135-141, July 2008
- R. Mirsu, V. Tiponut and I. Gavrilut, "Storing Information with Spiking Neural Networks", Proc. of the 13th WSEAS International Conference on Computers, Rhodes, Greece, ISBN: 978-960-474-099-4, ISSN: 1790-5109, 23-25 pp. 318-322, July 2009.
- I. Bogdanov, R. Mirsu and V. Tiponut, "MATLAB Model For Spiking Neural Networks", Proc. of the 13th WSEAS International Conference on Systems, Rhodes, Greece, ISBN: 978-960-474-099-4, ISSN: 1790-5109, 23-25, pp. 533-537, July 2009.
- I. Bogdanov, V. Tiponut and R. Mirsu, "New Achievements in Assisted Movement of Visually Impaired in Outdoor Environments", WSEAS Transactions on Circuits and Systems Volume 8 Issue 9, September 2009.
- R. Mirsu, V. Tiponut, L. Petromanjanc and Z. Haraszy, "Improved p-Delta Learning Algorithm", Proc. of the 14th WSEAS International Conference on Systems, Corfu, Greece, ISBN: 978-960-474-199-1, ISSN: 1792-4235, 22-24, pp. 282-287, July 2010.
- R. Mirsu and V. Tiponut, "Parallel Model for Spiking Neural Networks using MATLAB", 9th International Symposium on Electronics and

BUPT

Telecommunications (ISETC), Timisoara, Romania, ISBN 978-1-4244-8457-7, pp 369-373, November 2010

- R. Mirsu, C. Caleanu and V. Tiponut, "GPU Accelerated Model for Liquid State Machine based on Spiking Neurons", 17th International Conference on Soft Computing (MENDEL 2011), Brno, June 2011.
- D. B. Mirsu, G. Prostean and R. Mirsu, "Genetic Optimization for Transportation Problem", Proc. of the 15th WSEAS International Conference on Systems, Corfu, ISBN: 978-1-61804-023-7, ISSN: 1792-4235 Greece, 14-16, pp. 459-462, July 2011.
- R. Mirsu, C. Caleanu and V. Tiponut, "Optimized Model for Spiking Neural Network using CUDA" – submitted to "Neural Networks", ISSN 0893-6080 (Elsevier)
- R. Mirsu, V. Tiponut, "Trajectory Analysis using Liquid State Machine" – unpublished

REFERENCES

[1]. Maass, W. (1997). "Networks of spiking neurons: the third generation of neural network models." Neural Networks 10(9): 1659-1671.

[2]. Hodgkin, A. L. and A. F. Huxley (1952). "A quantitative description of membrane current and its application to conduction and excitation." J. Physiol 117: 500–557.

[3]. Izhikevich, E. M. (2004). "Which model to use for cortical spiking neurons?" Neural Networks, IEEE Transactions on 15(5): 1063-1070.

[4]. Bi, G. and M. Poo (1998). "Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type." Journal of Neuroscience 18(24): 10464-10472.

[5]. Izhikevich, E. M. (1999). "Weakly Connected Quasi-Periodic Oscillators, FM Interactions, and Multiplexing in the Brain." SIAM Journal on Applied Mathematics 59(6): 2193-2223.

[6]. Izhikevich, E. M. (2001). "Resonate and Fire Neurons". Neural Networks 14 (2001) 883-894.

[7]. Izhikevich, E. M. (2005). "Polychronization: Computation with Spikes." Neural Computation 18(2): 245-282.

[8]. Abeles, M. (1991). Corticonics: Neural Circuits of the Cerebral Cortex, Cambridge University Press.

[9]. Abeles, M. (2002). Synfire Chains. The handbook of brain theory and neural networks, MIT Press.

[10]. Bienenstock, E. (1995). "A model of neocortex." Network: Computation in Neural Systems 6(2): 179-224.

[11]. Beggs, J. M. and D. Plenz (2003). "Neuronal Avalanches in Neocortical Circuits." Journal of Neuroscience 23(35): 11167-11177.

[12]. Ikegaya, Y., G. Aaron, et al. (2004). "Synfire Chains and Cortical Songs: Temporal Modules of Cortical Activity." Science 304(5670): 559-564.

[13]. Csibra, G., G. Davis, et al. (2000). "Gamma Oscillations and Object Processing in the Infant Brain." Science 290(5496): 1582-1585.

[14]. Engel, A. K., P. Fries, et al. (2001). "Dynamic Predictions: Oscillations and Synchrony in Top–Down Processing." Nature Reviews Neuroscience 2(10): 704-716.

[15]. Jensen, O., J. Gelfand, et al. (2002). "Oscillations in the Alpha Band (9–12 Hz) Increase with Memory Load during Retention in a Short-term Memory Task." Cerebral Cortex 12(8): 877-882.

[16]. Jensen, O., M. A. Idiart, et al. (1996). "Physiologically realistic formation of autoassociative memory in networks with theta/gamma oscillations: role of fast NMDA channels." Learn Mem 3(2-3): 243-56.

[17]. H. Jaeger, W. Maass, and J. Principe. Introduction to the special issue on echo state networks and liquid state machines. Neural Networks, 20(3):287-289, 2007.

[18]. W. Maass, T. Natschlaeger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural Computation, 14(11):2531-2560, 2002

[19]. H. Jaeger, The "echo state" approach to analyzing and training recurrent neural networks. GMD Report 148, German National Research Center for Information Technology, 2001.

[20]. Yamazaki T., Tanaka S., "The cerebellum as a liquid state machine", Neural Networks. 2007 Apr; 20(3):290-7. Epub 2007 Apr 29.

[21]. Lazar A., Pipa G., Triesch J., "Fading Memory and Time Series Prediction in Recurrent Networks with Different Forms of Plasticity", Neural Networks 20, 3, 312-322, 2007.

[22]. Joshi P., "From memory-based decisions to decision-based movements: A model of interval discrimination followed by action selection", Neural Networks 20 (2007) 298–311.

[23]. R. Legenstein, W. Maass, "Edge of Chaos and Prediction of Computational Performance for Neural Circuit Models", Neural Networks, 20(3):323-334, 2007.

[24]. Skowronski M., Harris J., "Noise-Robust Automatic Speech Recognition using a Predictive Echo State Network", IEEE Transactions on Audio, Speech and Language Processing (2007), Volume: 15, Issue: 5, Pages: 1724-1730 `

[25]. Sebastian W., "Computation with Spiking Neurons", PhD Thesis, University of Cambridge 2004

[26]. David Verstraeten, Benjamin Schrauwen, Dirk Stroobandt, "Isolated word recognition using a Liquid State Machine", In ESANN'05, European Symposium on Artificial Neural Network

[27]. Tong, M. H., Bickett, A. D., Christiansen, E. M., & Cottrell, G. W. (2007). Learning grammatical structure with Echo State Networks. Neural Networks, 20, 424–432.

[28]. Yanbo Xue, Le Yang, Simon Haykin, "Decoupled Echo State Networks With Lateral Inhibition", IEEE Workshop on Machine Learning for Signal Processing, 2008. MLSP 2008, 444-449

[29]. Ozturk M., Xu D., Principe H., "Analysis and design of echo state networks", Journal of Neural Computation, Volume 19 Issue 1, 2007

[30]. Nicolas J Dedual, Mustafa C Ozturk, Justin C Sanchez, Jose C Principe, "An Associative Memory Readout in ESN for Neural Action Potential Detection", 2007 International Joint Conference on Neural Networks (2007), 2295-2299.

[31]. Mohammad A. Bhuiyan, Vivek K. Pallipuram, Melissa C. Smith, "Acceleration of Spiking Neural Networks in Emerging Multi-core and GPU Architectures", IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010

[32]. Wulfram Gerstner and Werner M. Kistler, "Spiking Neuron Models: Single Neurons, Populations, Plasticity", Cambridge University Press, 2002.

[33] L. Pape, de Gruijl J., and M. Wiering, "Democratic liquid state machines for music classification", Speech, Audio, Image and Biomedical Signal Processing using Neural Networks, Bookseries: Studies in Computational Intelligence, 83, 2008.

[34] Wulfram Gerstner, "Coding Properties of Spiking Neurons: reverse and cross-correlations", Neural Networks, Vol.14, Lausanne, 2001, pp. 599-610.

[35] Thomas Natschlager, Berthold Ruf, "Spatial and Temporal Pattern Analysis via Spiking Neurons", Network: Computer Neural Systems, Vol.9, 1998, pp. 319-332.

[36] J. Hopfield, "Pattern Recognition Computation using Action Potential Timing for Stimulus Representation", NATURE, 376:33-36, 1995.

BUPT

[37] Y. Prut, E. Vaadia, H. Bergman, I. Haalman, H. Slovin, M. Abeles. "Spatiotemporal structure of cortical activity: properties and behavioral relevance", Journal of Neurophysiology, 1998, 79:2857-2874.
[38] M. Abeles, H. Bergman, E. Margalit, E. Vaadia. "Spatiotemporal firing patterns in the frontal cortex of behaving monkeys", Journal of Neurophysiology, 70(4), 1993, pp. 1629-1638.
[39] Mirsu, R., Tiponut, V., Gavrilut, I. Storing Information with Spiking Neural Networks, Proc. of the 13th WSEAS International Conference on Computers, Rhodes, Greece, ISBN: 978-960-474-099-4, ISSN: 1790-5109, 23-25 July 2009, pp. 318-322
[40] Jayavan H. B. Wijekoon, Piotr Dudek, Simple Analogue Circuit of a Cortical Neuron, 13th IEEE International Conference on Electronics Circuits and Systems, 2006, pp. 1344-1347.
[41] Fabrice Bernhard, Renaud Keriven, Spiking Neurons on GPU's, CERTIS-ENPC Research Report, 2005.
[42] Giacomo Indivieri, Neuromorphic bistable VLSI synapses with spike-timing-dependent plasticity, Advances in Neural Information Processing Systems, Vol.15, Cambridge MA, 2002.
[43] Misha Tsodyks, Asher Uziel, and Henry Markram. "Synchrony generation in recurrent networks with frequency-dependent synapses" J. Neurosci, 20:50, 2000.
[44] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators" Neural Networks, 2(5):359–366, 1989.
[45] DeLiang Wang, Temporal Pattern Processing, The Handbook of Brain Theory and Neural Networks, 2nd Edition, MIT Press Cambridge MA, 2003, pp. 1163-1167.
[46] L. F. Abbott, "Synaptic plastictiy - taming the beast", Nature Neurosci., 3:1178-1183, 2000
[47] L. F. Abbott, T. B. Kepler, "Model neurons: from Hodgkin-Huxley to Hopfield". In Garrido, L., editor, Statistical Mechanics of Neural Networks. Springer, Berlin, 1990
[48] Bell, C., Han, V., Sugawara, Y., and Grant, K., "Synaptic plasticity in a cerebellum-like structure depends on temporal order", Nature, 387:278-281, 1997
[49] Brunel, N., Chance, F., Fourcaud, N., and Abbott, L. F., "Effects of synaptic noise and filtering on the frequency response of spiking neurons", Phys. Rev. Lett., 86:2186-2189, 2001
[50] Chow, C. C. and Kopell, N., "Dynamics of spiking neurons with electrical coupling", Neural Comput., 12:1643-1678, 2000
[51] W. Maass and H. Markram, "On the computational power of recurrent circuits of spiking neurons", Journal of Computer and System Sciences. 69(4), 593–616, 2004.
[52] R. Legenstein, D. Pecevski, and W. Maass, "A learning theory for rewardmodulated spike-timing-dependent plasticity with application to biofeedback" PLoS Computational Biology. 4(10), 1–27, 2008.
[53] M. Rabinovich, R. Huerta, and G. Laurent, "Transient dynamics for neural Processing", Science. 321, 45–50, 2008.
[54] Talia Konkle, Image Segmentation Using Neural Oscillators.
[55] Bogdanov, I., Mirsu, R., Tiponut, V. MATLAB Model For Spiking Neural Networks, Proc. of the 13th WSEAS International Conference on Systems,

BUPT

Rhodes, Greece, ISBN: 978-960-474-099-4, ISSN: 1790-5109, 23-25 July 2009, pp. 533-537
[56] Mirsu, R., Tiponut, V. Parallel Model for Spiking Neural Networks using MATLAB, 2010 9th International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, ISBN 978-1-4244-8457-7, page 369
[57] Duane Hanselman, Bruce Littlefield, MATLAB The Language of Technical Computing, Prentice Hall, 2001.
[58] Burkitt, A. N. and Clark, G. M., "Analysis of integrate-and-fire neurons: synchronization of synaptic input and spike output", Neural Comput., 11:871-901, 1999
[59] Sanders, J., Kandrot, E. CUDA by Example. An Introduction to General-Purpose GPU Programming. Addison Wesley, Reading, MA, 2010.
[60] NVIDIA CUDATM. NVIDIA CUDA C Programming Guide 3.1.1. 2010.
[61] NVIDIA CUDATM. Parallel NSight 1.51 User Guide 2010.
[62] Kirk, D., Hwu, W. Programming Massively Parallel Processors. Morgan Kaufmann, Burlington, MA, 2010.
[63] Herlihy, M., Shavit, N. The Art of MultiProcessor Programming. Morgan Kaufmann, Burlington, MA, 2010.
[64] Mirsu R, Caleanu C., Tiponut V. GPU Accelerated Model for Liquid State Machine based on Spiking Neurons, 17th International Conference on Soft Computing (MENDEL 2011), Brno, Czech Republic
[65] Mirsu R., Caleanu C. and Tiponut V., "Optimized Model for Spiking Neural Network using CUDA", to be published
[66] Peter Auer, Harald M. Burgsteiner, Wolfgang Maass. The p-Delta Learning Rule for Parallel Perceptron, 2002.
[67] Wolfgang Maass. Neural Computation with Winner-Take-All as the only Nonlinear Operation, Advances in Neural Information Processing Systems, vol. 12, MIT Press, Cambridge, 2000, pp. 293-299.
[68] Mirsu, R., Tiponut, V., Petromanjanc, L., Haraszy, Z. Improved p-Delta Learning Algorithm, Proc. of the 14th WSEAS International Conference on Systems, Corfu, Greece, ISBN: 978-960-474-199-1, ISSN: 1792-4235, 22-24 July 2010, pp. 282-287
[69] Freund Y., Schapire R. E. Large margin classification using the Perceptron algorithm. Machine Learning, 37(3), 1999, pp. 277-296.
[70] G.M. Wojcik, W.A. Kaminski, "Liquid State Machine Built of HodgkinHuxley Neurons and Pattern Recognition," Neurocomputing, vol. 239, pp. 245-251, 2004.
[71] G.M. Wojcik, W.A. Kaminski, "Liquid State Machine and its separation ability as function of electrical parameters of cell," Neurocomputing, vol. 70, pp. 2593-2697, 2007.
[72] B.J. Grzyb, E. Chinellato, G.M. Wojcik, W.A. Kaminski, "Which Model to Use for the Liquid State Machine?", IntI. Joint Conf. on Neural Networks, Atlanta 2009.
[73] T. Natschlger, W. Maass, H. Markram, "The "Liquid Computer": A Novel Strategy for Real-Time Computing on Time Series," Special Issue on Foundations ofInformation Processing ofTELEMATIK, vol. 8, 2002, pp. 39-43, 2002.
[74] Bertschinger, N. & Natschläger, T. Real-Time Computation at the Edge of Chaos in Recurrent Neural Networks, Journal of Neural Computation, vol. 16(7), pp.1413-1436 (2004).

BUPT

[75] Bohte, S.M., Kok, J.N. & La Poutré, H. Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons, Neurocomputing, 2000.
[76] Sander M. Bohte, Joost N. Kok and Han La Poutre, "SpikeProp: Backpropagation for Networks of Spiking Neurons"
[77] S. McKennoch, D. Liu, and L. G. Bushnell, "Fast Modifications of the SpikeProp Algorithm"
[78] Fernando, C. & Sojakka, S. Pattern recognition in a bucket: a real liquid brain, ECAL, 2003.
[79] Goldenholz, D. Liquid Computing: A Real Effect, Technical report, Boston University Department of Biomedical Engineering, 2002.
[80] S. Haeusler and W. Maass, "A statistical analysis of information processing properties of lamina-specific cortical microcircuit models", Cerebral Cortex. 17(1), 149–162, 2007.
[81] R. Legenstein and W. Maass, "What makes a dynamical system computationally powerful?", New Directions in Statistical Signal Processing: From Systems to Brains, pp. 127–154. MIT Press, 2007.
[82] B. Schrauwen, M. D'Haene, D. Verstraeten, and D.Stroobandt, "Compact hardware liquid state machines on FPGA for real-time speech recognition", Neural Networks. 21, 511–523, 2008.
[83] D. Buonomano and W. Maass, "State-dependent computations: Spatiotemporal processing in cortical networks", Nature Reviews in Neuroscience. 10 (2), 113–125, 2009)
[84] B. Jones, D. Stekel, J. Rowe, and C. Fernando, "Is there a liquid state machine in the bacterium escherichia coli?", Artificial Life. ALIFE'07, IEEE Symposium, 187–191, 2007.
[85] A. Nugent, "Physical neural network liquid state machine utilizing nanotechnology." (US-Patent 7 392 230 32, June 2008).
[86] K. Bush and C. Anderson, "Modeling reward functions for incomplete state representations via echo state networks", In Proceedings of the International Joint Conference on Neural Networks, Montreal, Quebec, 2005.
[87] David Norton, "Improving liquid state machines through iterative refinement of the reservoir", Master's thesis, Brigham Young University, 2008.
[88]. Gabor, D., "Theory of communications". J. Int. Electr. Eng. 93, 427–457, 1946
[89]. Daugman, J.G.,"Uncertainty relation for resolution in space, spatial frequency and orientation optimized by 2D visual cortical filters." J. Opt. Soc. Am. 2 (7), 1160–1169, 1985.
[90]. Liu, C., Wechsler, H., "A Gabor feature classifier for face recognition" In: Proc. of Eighth IEEE Int. Conf. on Computer Vision 2, July 7–14, pp. 270–275, 2001.
[91] Kamarainen, J., Kyrki, V., Hamouz, 2002. Invariant Gabor features for face evidence extraction. In: Proceedings of the IAPR Workshop on Machine Vision Applications. Nara, Japan, pp. 228–231
[92] Ayinde, O., Yang, Y. Face recognition approach based on rank correlation of Gabor-filtered images. Pattern Recognition 35, 6 (2002), 1275_1289
[93] Kyrki, V., Kamarainen, J., Kalviainen, H., 2001. Content based image matching using Gabor filtering. In: Proceedings of the Int. Conf. on Advanced Concepts for Intelligent Vision Systems Theory and Applications. Baden-Baden, Germany, pp. 45–49.
[94] Kyrki, V., 2002. Local and global feature extraction for invariant object recognition. Ph.D. thesis. Lappeenranta University of Technology.
[95] Z. Zhang, M. Lyons, M. Schuster, S. Akamatsu, Comparison Between Geometry-Based and Gabor-Wavelets-Based Facial Expression Recognition

BUPT

Using Multi-Layer Perceptron, Third IEEE International Conference on Automatic Face and Gesture Recognition, pp. 454-459, 1998.
[96] C. Caleanu, S. Huang, V. Gui, V. Tiponut, V. Maranescu, "Interest Operator versus Gabor filtering for facial imagery classification", Pattern Recognition Letters 28 (2007) 950–956
[97] Youssef Elmir, Zakaria Elberrichi, Reda Adjoudj, "Liquid State Machine Based Fingerprint Identification", Australian Journal of Basic and Applied Sciences, 5(5): 857-865, 2011
[98] Mirsu, R., Tiponut, V. Trajectory Analysis using Liquid State Machine, to be published
[99] Bogdanov, I., Tiponut, V., Mirsu, R., New Achievements in Assisted Movement of Visually Impaired in Outdoor Environments, WSEAS Transactions on Circuits and Systems Volume 8 Issue 9, September 2009.
[100] A. Helal, S, Moore, B. Ramachandran-Drishti, An Integrated Navigation System for Visually Impaired and Disabled, International Symposium on Wearable Computers (ISWC), 2001, pp. 149-156.
[101] V. Kulyukin, C. Gharpure, J. Nicholson, S.Pavithran, RFID in Robot Assisted Indoor Navigation for the Visual Impaired, IEEE/RSJ Intern. Conference on Intelligent Robots and Systems, Sendai, Japan (IROS), 2004, pp. 353-357.
[102] I. Ulrich, J. Borenstein, The GuideCane – Applying Mobile Robot Technologies to Assist Visually Impaired, IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, vol. 31, no. 2, 2001, pp. 131-136.
[103] S. Soval, I. Ulrich, J. Borenstein, Robotics-based Obstacole Avoidance Systems for Blind and Visually Impaired, IEEE Robotics Magazine, vol. 10, no. 1, 2003, pp. 9-20.
[104] H. Shim, J. Lee, E. Lee, A Study on the Sound-Imaging Algorithm of Obstacles Information for the Visually Impaired, The 2002 Intern. Conf. on Circuits/Systems, Computers and Communications (ITC-CSCC), 2002, pp. 29-31.
[105] Beata J. Grzyb, Eris Chinellato, Grzegorz M. Wojcik, and Wieslaw, "A. Kaminski, Facial Expression Recognition based on Liquid State Machines built of Alternative Neuron Models", Proceedings of International Joint Conference on Neural Networks, Atlanta, Georgia, USA, June 14-19, 2009
[106] R. Adolphs, "Neural systems for recognizing emotion," Current Opionion in Neurobiology, vol. 12, pp. 169-177,2002.
[107] C.J. Harmer, K.V. Thilo, J.C. Rothwell, G.M. Goodwin, "Transcranial magnetic stimulation of medial-frontal cortex impairs the processing of angry facial expressions," Nature Neuroscience, volA, pp. 17-18, 2001.
[108] R.J. Blair, J.S. Morris, C.D. Frith, D.I. Perrett, R.J. Dolan, "Dissociable neural responses to facial expressions of sadness and anger," Brain, vol. 122, pp. 883-893, 1999.
[109] C. Caleanu, "Facial Recognition using Committee of Neural Networks". In: Proc. 5th Seminar on Neural Network Applications in electrical engineering, NEUREL 2000, Belgrade, Yugoslavia, pp. 97–100.
[110] C. Caleanu, "Face recognition using parallel neural processing and interest operator method", Ph.D. thesis. University POLITEHNICA Timisoara, 2001
[111] P. Joshi and W. Maass, "Movement generation with circuits of spiking neurons", Neural Computation, 17(8), 1715–1738, 2005.

BUPT