View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

TRAINING ALGORITHMS FOR NETWORKS OF SPIKING NEURONS

A Thesis

by

NITYENDRA SINGH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Peng Li
Committee Members,	Daniel A Jimenez
	Aydin Karsilayan
	Raffaella Righetti
Head of Department,	Chanan Singh

December 2014

Major Subject: Electrical Engineering

Copyright 2014 Nityendra Singh

ABSTRACT

Neural networks represent a type of computing that is based on the way that the brain performs computations. Neural networks are good at fitting non-linear functions and recognizing patterns. It is believed that biological neurons work similar to spiking neurons that process temporal information. In 2002, Bohte derived a back-propagation training algorithm (dubbbed as SpikeProp) for spiking neural networks (SNNs) containing temporal information as firing time of first spike. SpikeProp algorithm and its different variations were subject of many publications in the last decade.

SpikeProp algorithm works for continuous weight SNNs. Implementing continuous parameters on hardware is a difficult task. On the other hand implementing digital logic on hardware is more straightforward because of many available tools. Training SNN with discrete weights is tricky because smallest change allowed in weights is a discrete step. And this discrete step might affect the accuracy of the network by huge amount. Previous works have been done for Artificial Neural Networks (ANNs) with discrete weights but there is no research in the area of training SNNs with discrete weights. New algorithms have been proposed as part of this thesis work. These algorithms work well for training discrete weights in a spiking neural network. These new algorithms use SpikeProp algorithm for choosing weights that are to be updated. Several standard classification datasets have been used to demonstrate the efficacy of proposed algorithms. It is shown that one of the proposed algorithms (Multiple Weights Multiple Steps) takes less execution time to train and the results are comparable to continuous weight SNNs in terms of accuracy.

DEDICATION

Dedicated to my parents, brothers and sisters.

ACKNOWLEDGEMENTS

I express my gratitude to Prof. Peng Li, for his continuous guidance, support and motivation. He wants his students to achieve only the very best. I admire his enthusiasm about each and everything he does, including research and teaching. Also, I admire his humility and openness. I have learned immensely in last couple of years under his supervision and I wish to match him up in skills, knowledge, and dedication.

I would like to thank my committee members, Daniel A Jimenez, Prof. Aydin Karsilayan and Prof. Raffaella Righetti for their support during the course of this research.

I am also grateful to friends, colleagues, department faculty and staff for making my time at Texas A&M University a great experience.

Finally, I thank to my parents, who have always shown confidence in me and supported me in my endeavors. Everything that I have achieved in life was not possible without their unconditional love and support.

NOMENCLATURE

ANN	Artificial Neural Network
PSP	Post Synaptic Potential
SNN	Spiking Neural Network
SNNs	Spiking Neural Networks

TABLE OF CONTENTS

Page

ABSTRACT . . . ii

DEDICATION . . . iii

ACKNOWLEDGEMENTS . . . iv

NOMENCLATURE . . . v

TABLE OF CONTENTS . . . vi

LIST OF FIGURES . . . viii

LIST OF TABLES . . . ix

1. INTRODUCTION . . . 1
   - 1.1 Problem Statement . . . 2
   - 1.2 Literature Review . . . 3
   - 1.3 Applications . . . 5

2. INTRODUCTION TO NEURAL NETWORKS . . . 6
   - 2.1 Biological Neurons . . . 6
   - 2.2 First Generation Neural Networks . . . 7
   - 2.3 Second Generation Neural Networks . . . 8
   - 2.4 Third Generation Neural Networks . . . 11

3. SPIKING NEURAL NETWORK ARCHITECTURE . . . 14

4. BACKPROPAGATION ALGORITHM FOR SNNS . . . 18
   - 4.1 SpikeProp Algorithm . . . 18
   - 4.2 Fast Learning based on Momentum [35] . . . 22
   - 4.3 Fast Learning based on RProp [20] . . . 22
   - 4.4 Fast Learning based on QuickProp [20] . . . 23

5. PROPOSED ALGORITHMS FOR TRAINING DISCRETE WEIGHTS . . 25

5.1 Motivation 25
5.2 Basic Idea 26
5.3 Single Weight Single Step Algorithm 28
5.4 Single Weight Multiple Steps Algorithm 28
5.5 Multiple Weights Single Step Algorithm 29
5.6 Multiple Weights Multiple Steps Algorithm 29

6. RESULTS AND CONCLUSION 31

6.1 XOR Problem 31
6.1.1 Input Encoding 31
6.1.2 Network Setup 31
6.1.3 Output Encoding 32
6.1.4 Convergence Criteria 34
6.1.5 Comparison of Discrete Weight Algorithms in terms of Execution Time 34
6.2 Other Standard Benchmarks 34
6.3 Fisher Iris Dataset 35
6.3.1 Input Encoding 36
6.3.2 Network Setup 37
6.3.3 Output Encoding 38
6.3.4 Convergence Criteria 38
6.3.5 Results 38
6.4 Wine Recognition Dataset 40
6.5 Speed Up Ideas 40
6.6 Conclusion 41

REFERENCES 42

LIST OF FIGURES

FIGURE Page

1.1 The schematic model of a biological neuron 2

2.1 Symbolic illustration of a linear threshold gate 8

2.2 Building blocks of sigmoidal neuron 8

2.3 Activation function used in sigmoidal neural network 10

2.4 Spiking neural network architecture; Input neurons fire spikes; Output spikes are measured 11

3.1 Synaptic connections between two neurons of neighboring layers . . . 15

3.2 Spike response function; it reaches maximum at τ 15

3.3 Demonstrating calculation of firing time 16

5.1 Discrete SNN system 26

5.2 Figure showing a big change in firing time of a neuron because of a small change in synaptic weight, the circle indicates the firing instant 27

6.1 One training sample for XOR problem 33

6.2 Comparison of discrete weight training algorithms 35

6.3 Population encoding scheme 37

6.4 Comparison of discrete weight training algorithms for Fisher Iris classification problem 39

LIST OF TABLES

TABLE Page

6.1 Input and output patterns for XOR problem 33

6.2 Comparison of different training algorithm on XOR problem 34

6.3 Comparison of different training algorithm on Fisher Iris Problem . . 39

6.4 Comparison of different training algorithm on Wine Recognition Problem 40

1. INTRODUCTION

We deal with computers every day. Sometimes computers become inefficient or incapable of processing desirable tasks. What we would wish is to make our computers smarter and more user friendly. It would be incredible if computers can communicate like us. That means it would involve thinking and understanding emotions just like humans do. To do this, we need to put ourselves in understanding how humans think and how our brain works. First let us consider, how computers work. They take some input, perform some calculation and produce some output. On the other hand, human brain is extremely complicated. Our brain is considered as one of the most intricate system available in the universe [13]. However, scientists from all over the world are still struggling to find out how exactly our brain works. One thing they know is our brain is a network of tiny cells that we call neurons. Our brain has billions of neurons and each of these neurons is connected to about 10000 other neurons via synapses [11]. This gigantic network of neurons and synapses builds up the hardware (or often called wetware) which carries out the computations underlying the human behavior [22]. Scientists found out that our brain processes information through neurons using electric impulses. These impulses travel through the axon (output wire of a neurons that connects to other neurons) and discharge through synapses (shown in figure 1.1).

The artificially made network of neurons is named as neural network. Over time, our perception of neural network has been changing [12]. Spiking neural networks (SNNs), which belong to the third generation model of neural networks, are considered to be the type of neural networks which closely match with networks in our brain. These three generation of neural networks would be discussed in the next

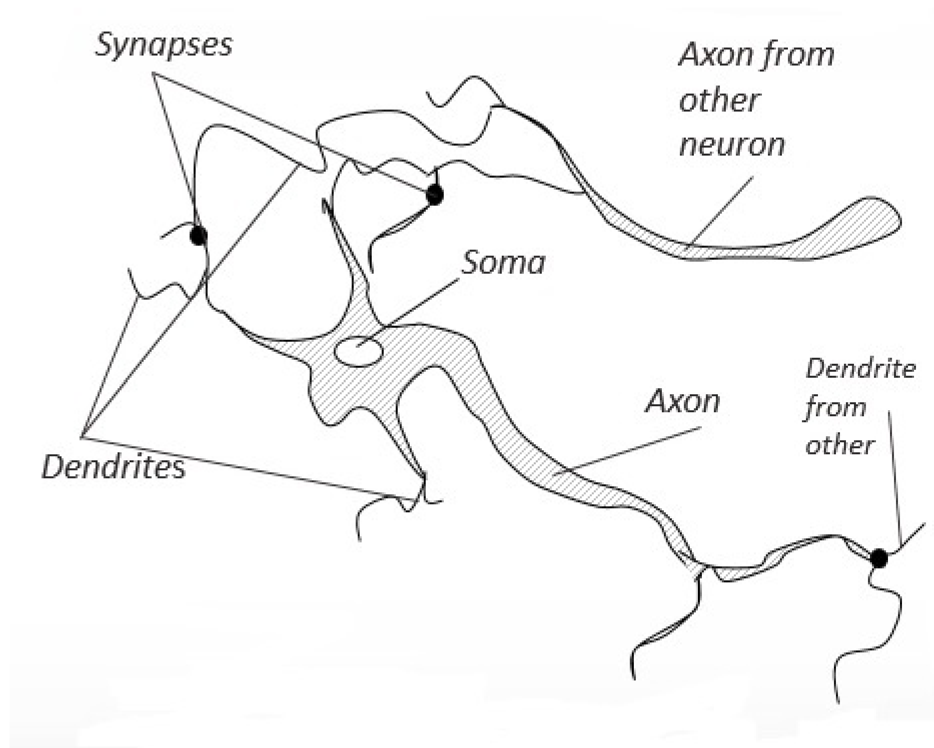

Figure 1.1: The schematic model of a biological neuron

chapter.

1.1 Problem Statement

Once we know the functioning of SNNs, the next step would be to implement it on hardware. There are two ways in which we can implement a function on hardware: analog or digital. Nowadays, implementing a function in digital is way more straightforward than implementing in analog. This is because there are many easy to use softwares available which do all the nitty-gritty work for us. Therefore, it would be better to analyze behaviors of SNNs with digital components. Training algorithms for SNNs with continuous parameters already exist. In [3], Bohte developed a training method, dubbed SpikeProp, similar to traditional error back-propagation al-

gorithms [10]. SpikeProp algorithm cannot be directly applied to SNNs with discrete parameters, because discrete parameters do not allow continuous-valued incremental changes. It only allows changes in steps (also known as quantization step). To overcome this problem, there is a need of more sophisticated algorithms to address this problem. In this work, new training algorithms are proposed that works for SNNs with discrete weights.

1.2 Literature Review

In recent years, the field of spiking neural networks (SNNs), a machine learning technique similar to traditional artificial neural networks (ANNs), has experienced an increasingly large amount of research attention [20]. It has been shown that networks of spiking neurons with temporal coding may have a larger computational power than sigmoidal neural nets with the same number of units [16]. Also, these SNNs model the precise timing of the spikes fired by the neuron, as opposed to conventional neural networks, which model only the average firing rate of the neurons [18]. Similar to the first generation neurons (Section 2.2), spiking neurons act as an integrate-and-fire unit and have an all-or-none response. The spiking neurons, however, have an inherent dynamic nature characterized by an internal state which changes with time. As a result, each postsynaptic neuron fires an action potential or spike at the time instance its internal state exceeds the neuron threshold [9].

SpikeProp algorithm works on the assumption that the value of the internal state of the neuron increases linearly in the infinitesimal time around the instant when neuron fires. Unlike feed forward sigmoidal neural networks where two neurons are connected by a single synapse only, SNNs contain several synapses between two neurons. Each synapse has weight and delay associated with it. In [3], the delays associated with the synapses are kept fixed and weights are trained. In [27, 26], it

was shown that even the delays can be trained and that way we need fewer synapses between neurons for a particular problem. The classification capabilities of SpikeProp training algorithm was investigated on several benchmark problems including the well-known XOR problem. Subsequently, various training algorithms such as back-propagation with momentum [35], resilient propagation (RProp) [20], QuickProp [20] and Levenberg-Marquardt BP [28] have shown better network training performances as compared to the original SpikeProp.

SpikeProp back-propagation algorithm considers weights as real numbers. It would be an important step to consider weights as discrete numbers so that SNNs is easily implementable on hardware. Pertinently, there has been some research in the area of implementing SNNs on hardware [23, 24]. In [23], authors implemented Gaussian Receptive Fields (GRF) on FPGA to encode real data into few spike trains. In [24], authors implemented over 1000 biologically plausible neurons on FPGA. There has not been any study on training SNNs with discrete weights. There have been some studies on training discrete weights of sigmoidal neural networks [33, 15]. However, both of these discrete weight training algorithms [33, 15] use traditional back propagation algorithms and choose weights which are to be trained. Proposed algorithms in this thesis are inspired from the works given in [33, 15]. In this thesis, new algorithms based on SpikeProp are shown to work well on many standard classification datasets. In supervised training, a set of input and output vectors are used to train the SNNs to behave in a desired way. First we analyze our algorithm on XOR problem, which is considered the most basic non-linear classification problem. XOR problem is usually used for demonstrating effectiveness of algorithms. We also analyzed these proposed algorithms on two more classification problems; 1. Fisher Iris Dataset (1988) [2, 5] 2. Wine Recognition data (1998) [2].

1.3 Applications

The most mysterious and little-understood organ in the entire body is the brain [13]. Our thoughts, emotions, and memories reside somewhere inside clusters of biological neurons. Our everyday decisions are taken somewhere in the brain. Because of the importance to study our brain, scientists are putting in their best effort to decode this complex organ. Well, there is a lot of research that has to happen in the effort to decode our brain. Currently, we are at a primitive stage of brain research.

The proposed algorithms in this thesis can be used to train a set of discrete weights of SNNs effectively. These proposed algorithms can be implemented on software to provide trained weights to SNN hardware. There have been many implementations of spiking neural network on hardware [34, 23, 24]. All these implementations assume that size of the weights is either continuous or the bit length is large. The proposed algorithms in this research would encourage the use of smaller number of bits for implementing SNNs on digital ASIC or FPGA platform.

2. INTRODUCTION TO NEURAL NETWORKS

Each living creature has to act in a suitable way in response to the various situations it encounters in its environment. Nowadays it is accepted that the nervous system, including the spinal cord as well as the neocortex, controls our behavior [31]. In a simplified picture one could view the nervous system as a device which receives input from various senses like auditory inputs from the ears and produces (computes) some output in the form of movement or speech. To understand how these computations are performed is one of the most interesting and challenging task in science. Thus it is not surprising that the field of neuroscience attracts a lot of researchers not only from biology but also from physics and computer science [22].

2.1 Biological Neurons

A neuron (also known as a neurone or nerve cell) is an electrically excitable cell that processes and transmits information through electrical and chemical signals. These signals between neurons occur via synapses which are specialized connections of a particular neuron with other neurons [31]. The nervous system consists of around 100 billion neurons and around 100 trillion synapses [11, 31]. All these neurons and synapses make a gigantic network which is responsible for our behavior and thought generation.

If one wants to understand how the nervous system computes a function one has to think about how information about the environment or internal states is represented and transmitted. It has been found that the shape of spike signal is always the same. One can defy the possibility that the voltage trajectory of an action potential (which is a short-lasting event in which the electrical membrane potential of a cell rapidly rises and falls and calculated by spike signals) carries relevant information. Thus a

central question in the field of neuroscience is how neurons encode information in the sequence of action potential they emit. There has been extensive research in this quest. The neural networks can be classified in three different generations, which would be discussed in next few sections.

2.2 First Generation Neural Networks

In 1943, McCulloch and Pitts proposed the first neuron model: the threshold gate [19]. The characteristics of their model was that they treated a neuron as a binary device i.e. they distinguished only between the occurrence and absence of a spike. Figure 2.1 shows a symbolic illustration of a linear threshold gate. The threshold gate is used as building block for various network types including multilayer perceptrons [1]. It turned out that the threshold gate is a computationally powerful device i.e. one can compute complex functions with rather small networks made up of threshold gates. From a theoretical point of view the threshold gate is a very interesting model but it is unlikely that real biological systems use such a binary encoding scheme. A prerequisite for such a binary coding scheme is a kind of global clocking mechanism but it is very unlikely that such a mechanism exists in biological systems [22].

The McCulloch-Pitts model of a neuron is simple yet has substantial computing potential. It also has a precise mathematical definition. However, this model is so simplistic that it only generates a binary output and also the weight and threshold values are fixed. The neural computing algorithm must encompass diverse features for various applications. Thus, we need to obtain the neural model with more flexible computational features [14]. Since there was no automatic algorithm to calculate weights, these weights were difficult to calculate by hand for bigger networks.

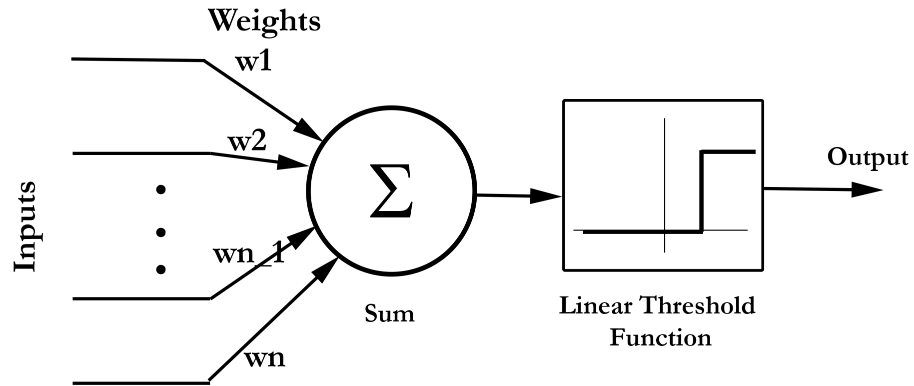

Figure 2.1: Symbolic illustration of a linear threshold gate

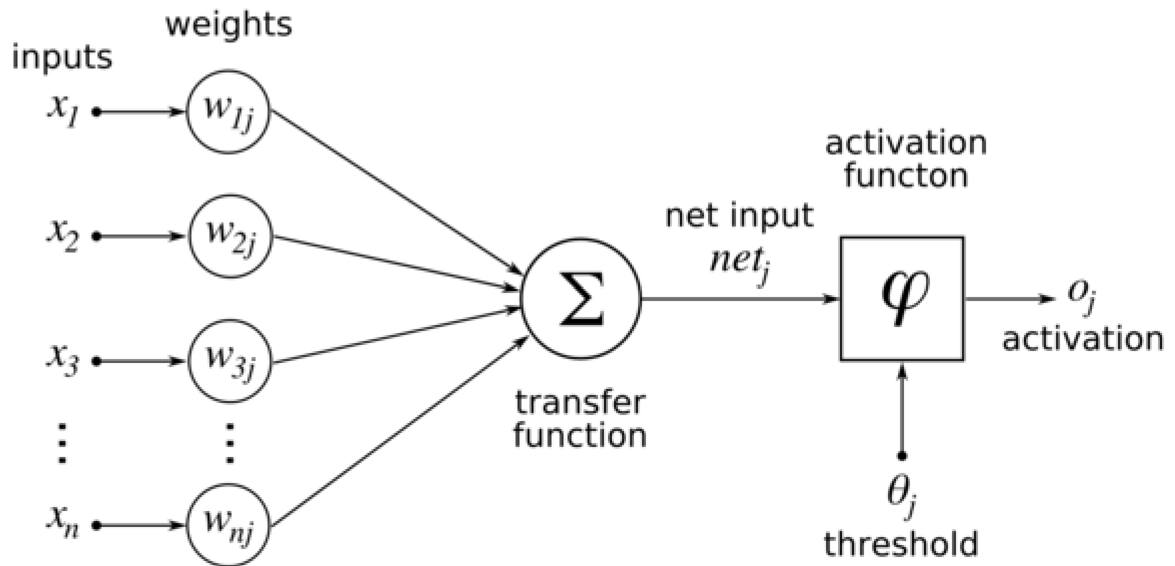

Figure 2.2: Building blocks of sigmoidal neuron

2.3 Second Generation Neural Networks

Around 1980s, back-propagation algorithm, for training neural networks weights, was invented and hand coded neural networks became outdated [10]. These neu-

ral networks were named as sigmoidal neural network. Figure 2.2 shows the basic functionality of a single neuron in sigmoidal neural network. In this, inputs to the neuron are weighted and then summed up. Output of the sum goes through an activation function (shown in 2.3) which is a monotonically increasing function [14]. The backpropagation learning algorithm can be divided into two phases: propagation and weight update [25]. Each propagation phase involves the following steps: 1. Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations. 2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas (backpropagation parameters) of all output and hidden neurons. Each weight update phase follows the following steps: 1. Multiply its output delta (calculated in propagation phase) and input activation to get the gradient of the weight. 2. Subtract a ratio (percentage) of the gradient from the weight.

These types of neural networks remain popular until a little different algorithm Support Vector Machine (SVM) came into picture. In many classification problems, SVM works better than Sigmoidal Neural Networks [12].

Neural Networks which work on spikes can be considered as Spiking Neural Networks. Figure 2.4 depicts how spiking neural networks work on spikes. Since inputs and outputs are also spikes, there must be a sophisticated encoding scheme to convert input features to spikes. Various coding methods exist for interpreting the outgoing spike train as a real-valued number, either relying on the frequency of spikes, or the timing between spikes, to encode information.

Another possibility is that the number of spikes per second (called the firing rate) encodes relevant information. This idea leads to a model neuron known as sigmoidal gate. The output of a sigmoidal gate is a number which is supposed to represent the

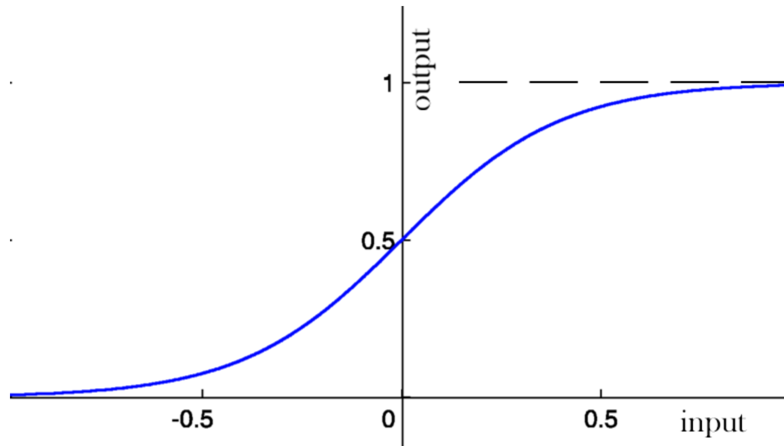

Figure 2.3: Activation function used in sigmoidal neural network

firing rate of the neuron. There exists a huge amount of literature which discusses in detail all aspects of this kind of neural network model. We just want to note that networks of sigmoidal gates can in principle compute any analog function and that along with this type of model the question of learning in neural networks was intensively investigated for the first time [32, 22].

It is well known that firing rates play an important role in the nervous system especially in the primary sensory areas where low level information processing takes place. However, recent experiments revealed that the human nervous system is able to perform complex visual tasks in time intervals as short as 150 ms. The pathway from the retina to higher areas in the neocortex along which a visual stimuli is processed consists of about 10 processing stages. Furthermore the firing rates in the areas involved in such complex computations are well below 100 spikes per second. But to estimate such low firing rates one has to wait at least 30 to 50 ms which is a contradiction to the finding that a computation involving 10 processing stages can be carried out within 150 ms. Further experimental results indicate that some

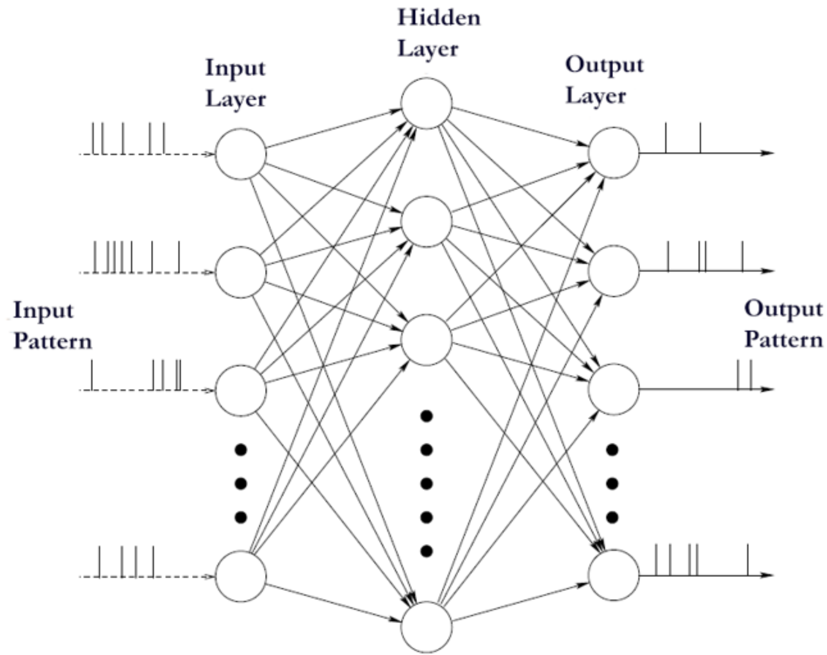

Figure 2.4: Spiking neural network architecture; Input neurons fire spikes; Output spikes are measured

biological neural systems indeed use the exact timing of individual spikes which further confirms the idea that the firing rate alone does not carry all the relevant information [17]. This leads to the advent of temporal coded neural networks.

2.4 Third Generation Neural Networks

Signals that carry information in our brain are spikes. Brain works really well on pattern recognition. Not only its accuracy is outstanding but it consumes very less power. These positive aspects of computations in our brain gave rise to a new class of neural network models where one also incorporates the timing of individual spikes [9]. Thus time plays a central role in SNNs whereas in most other neural network models there is even no notion of time.

When precise spike timing or high-frequency firing-rate fluctuations are found

to carry information, the neural code is often identified as a temporal code. A number of studies have found that the temporal resolution of the neural code is on a millisecond time scale, indicating that precise spike timing is a significant element in neural coding [17]. Neurons exhibit high-frequency fluctuations of firing-rates which could be noise or could carry information. Rate coding models suggest that these irregularities are noise, while temporal coding models suggest that they encode information. If the nervous system only uses rate codes to convey information, a more consistent, regular firing rate would have been evolutionarily advantageous, and neurons would have utilized this code over other less robust options. Temporal coding supplies an alternate explanation for the noise" suggesting that it actually encodes information and affects neural processing. To model this idea, binary symbols can be used to mark the spikes: 1 for a spike, 0 for no spike. Temporal coding allows the sequence 000111000111 to mean something different from 001100110011, even though the mean firing rate is the same for both sequences, at 6 spikes/10 ms. There are some neural encoding schemes which utilize aspects of rate and time both. The issue of temporal coding is distinct and independent from the issue of independent-spike rate coding. If each spike is independent of all the other spikes in the train, the temporal characters of the neural code is determined by the behavior of time-dependent firing rate r(t). If r(t) varies slowly with time, the code is typically called a rate code, and if it varies rapidly, the code is called temporal.

There can be multiple varieties in temporal coding. The simplest way of encoding is time-to-first spike coding [18]. In this coding scheme, each neuron only needs to fire a single spike to transmit information. If it emits several spikes, only the first spike after the reference signal counts. All following spikes would be irrelevant. Since each neuron in such a scenario transmits exactly one spike per stimulus, it is clear that only the timing conveys information and not the number of spikes. In time-to-

first spike coding, the firing time can be set as proportional to analog input data. Many researches have been happened that utilizes time-to-first spike coding scheme [21, 7, 3]. In [8], it is shown that using spike train instead of single spike, gives better result. But multiple spikes add more complexity in computing of network parameters. Because of maintaining simplicity, time-to-first spike is used in this research. To avoid using a lengthy term “temporal coded spiking neural network”, we would simply term it as “Spiking Neural Network” (SNN) next chapter onwards.

3. SPIKING NEURAL NETWORK ARCHITECTURE

Following earlier research in this area [3, 27, 26, 4, 8, 20, 7], the spiking neural network architecture used in this thesis work is a three layer feedforward network of spiking neurons (as shown in figure 2.4). All neurons from two neighboring layers are fully connected to each other with K number of synapses. Each synapse consists of a delay and weight associated with it (see figure 3.1). Synapses between two particular neurons have different delays associated with it. Input patterns are encoded as firing time and fed to input neurons. We used one extra neuron as bias neuron. In [29], it is explained in detail why bias neuron is compulsory to have at the input. Spike firing time at the bias neuron is assumed to be always 0 ms. In this research, firing times are considered in the order of ms although simulation time need not be in the same order. Therefore, we can compress or elongate the time scale.

The spike response function which is used for calculating the internal state variables is defined by equation 3.1 and shown in the figure 3.2 [6, 3, 7]. τ models the membrane potential decay time constant that determines the rise-time and decay-time of the post synaptic potential (PSP). A point to be noted is that spike response function is maximum when $t = \tau$. There have been different implementations of spike response function as well [4, 8]. In [4], a spike response function is defined as the difference of two exponential functions. In [8], multiple spikes along with the refractory period are incorporated. In this research, we will stick to the traditional exponential spike response function defined by equation 3.1 for simplicity.

$$\varepsilon(t) = \begin{cases} \frac{t}{\tau}\exp\left(1 - \frac{t}{\tau}\right) \; when \; t > 0 \\ 0 \; when \; t < 0 \end{cases} \tag{3.1}$$

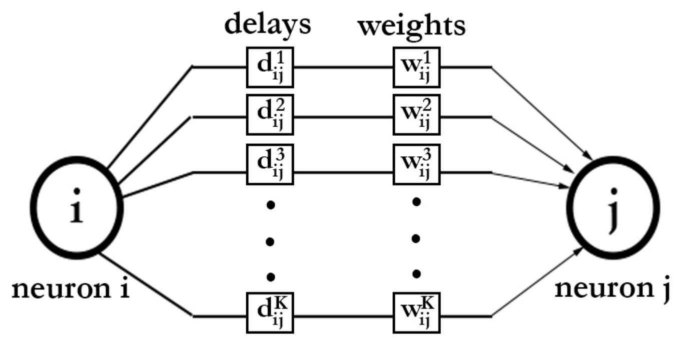

Figure 3.1: Synaptic connections between two neurons of neighboring layers

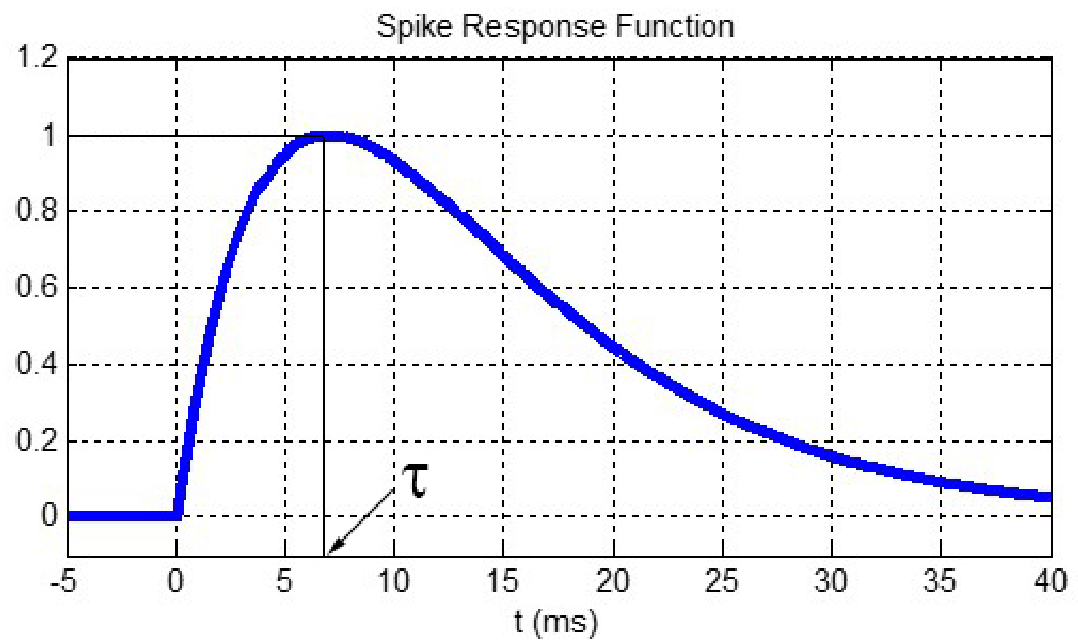

Figure 3.2: Spike response function; it reaches maximum at τ

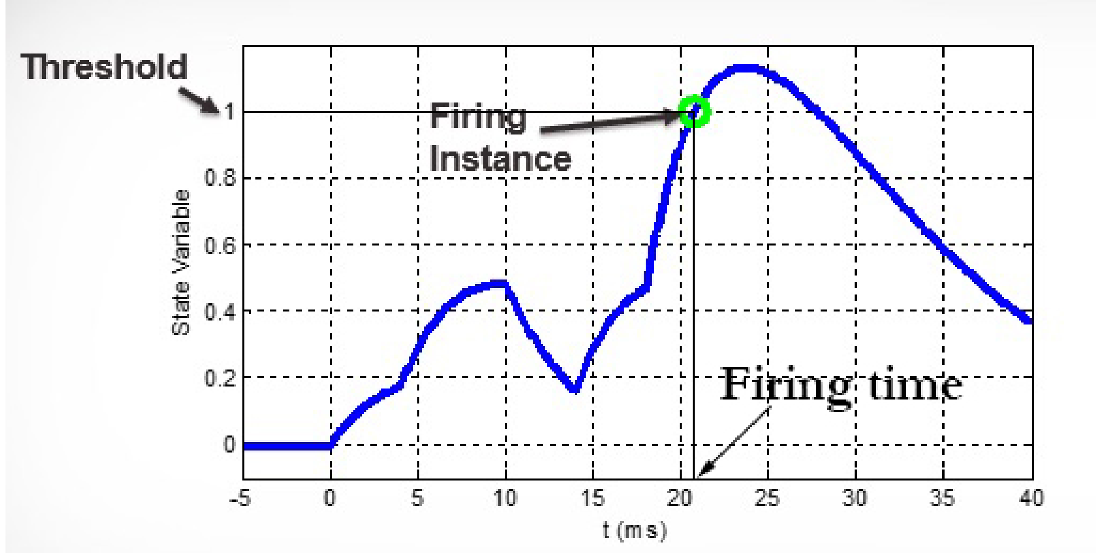

Figure 3.3: Demonstrating calculation of firing time

Internal state variable is defined as:

$$x_j(t) = \sum_{i \epsilon \Gamma_j} \sum_k w_{ij}^k \varepsilon(t - t_i - d_{ij}^k) \tag{3.2}$$

Here, Γ_j represents the set of neurons in the previous layer of the layer neuron i belongs to and t_i is the firing time of neuron i. d_{ij}^k and w_{ij}^k are the delay and weight associated with the synapse connected between neuron i and neuron j. Once internal state reaches the threshold, we say that particular time as firing time for neuron. In this work we use only the first spike time. Studies have shown that first spike carries useful information [18]. But later in [8], it is shown that neural network with multiple spikes carries more information and gives better result. On the other hand, simulating multiple spikes would create more complexity in the system. For simplicity, time-to-first spike is considered in this research work. The firing time t_j of neuron j is determined as the first instant when the state variable crosses the

threshold (as shown in figure 3.3). Thus, the firing time t_j is a non-linear function of the state variable x_j.

4. BACKPROPAGATION ALGORITHM FOR SNNS

In 2002, Bohte [3] presented a back propagation learning algorithm for SNNs. This algorithm is named as SpikeProp. The concept is similar to the backpropagation algorithm used in sigmoidal neural networks. This chapter begins with deriving original SpikeProp algorithm and later on, many other improved variations are described and analyzed.

4.1 SpikeProp Algorithm

Before going into equations of SpikeProp algorithm, let's label the layers of SNNs. H denotes input layer or set of input neurons, I denotes the hidden layer or set of neurons in hidden layer and J denotes output layer or set of neurons in the output layer. The main goal of the training algorithm is to learn a set of desired firing times, denoted $\{t_j^d\}$, where subscript j denotes jth output neuron. In other way, we can define an error function which is a function of actual firing time $\{t_j\}$ and goal of the training is to minimize the error function by updating the synaptic weights in each iteration.

Let's define the error function as

$$E = \frac{1}{2} \sum_{j \in J} \left(\mathrm{t_j} - \mathrm{t_j^d}\right)^2 \tag{4.1}$$

Now using gradient descent with respect to w_{ij}^k, where w_{ij}^k is synaptic weight of kth synapse between neuron i to output neuron j.

$$\triangle w_{ij}^k = -\eta \frac{\partial E}{\partial w_{ij}^k}, \tag{4.2}$$

Where, η is the learning rate of the algorithm. The above partial derivative term can be further expanded using the chain rule at the output neuron spike time instant, $t = t_j$.

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial t_j} \frac{\partial t_j}{\partial w_{ij}^k} \tag{4.3}$$

It can further be expanded as following because t_j is a function of x_j, which depends on the weights w_{ij}^k.

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial t_j} \frac{\partial t_j}{\partial x_j(t_j)} \frac{\partial x_j(t_j)}{\partial w_{ij}^k} \tag{4.4}$$

The first derivative on the right hand side of equation 4.4 is computed as,

$$\frac{\partial E}{\partial t_j} = \frac{\partial \left[\frac{1}{2}\sum_{j\in J}\left(\mathrm{t_j}-\mathrm{t_j^d}\right)^2\right]}{\partial t_j} = (t_j - t_j^d) \tag{4.5}$$

The third derivative term on the right hand side of equation 4.4 can be computed as

$$\frac{\partial x_j(t_j)}{\partial w_{ij}^k} = \frac{\partial \left[\sum_{n\epsilon\Gamma_j}\sum_l w_{nj}^l \varepsilon(t_j - t_n - d^l)\right]}{\partial w_{ij}^k} \tag{4.6}$$

Since the weight of any synapse is independent of the weights of other synapses, the two summation terms on the right hand side would be vanished,

$$\frac{\partial x_j(t_j)}{\partial w_{ij}^k} = \frac{\partial \left[w_{ij}^k \varepsilon(t_j - t_i - d^k)\right]}{\partial w_{ij}^k} = \varepsilon(t_j - t_i - d^k) \tag{4.7}$$

The second derivative term, on the right hand side of equation 4.4, $\partial t_j / \partial x_j(t_j)$ is not simple to calculate because t_i cannot be expressed as a continuous and differentiable function of $x_j(t_j)$. Bohte in [3] made an assumption that $x_j(t_j)$ is a linear function of t_i around the output spike time instant, $t = t_j$, In [8], this assumption is explained in much detail and it is shown graphically small change in threshold can give a huge

jump in firing time. This explains the abrupt changes in error value while training the network. To avoid these unwanted jumps, we use some heuristic rules on our simulation, which will be explained later. By implicit function theorem $\partial t_j/\partial x_j(t_j)$ is the negative inverse of the derivative $\partial x_j(t_j)/\partial t_j$. Hence,

$$\frac{\partial t_j}{\partial x_j\left(t_j\right)} = \frac{-1}{\partial x_j(t_j)/\partial t_j} \tag{4.8}$$

$$\frac{\partial x_j(t_j)}{\partial t_j} = \frac{\partial\left[\sum_{i\epsilon\Gamma_j}\sum_l w_{ij}^l \varepsilon(t_j - t_i - d^l)\right]}{\partial t_j} \tag{4.9}$$

$$\frac{\partial x_j(t_j)}{\partial t_j} = \sum_{i\epsilon\Gamma_j}\sum_l w_{ij}^l \frac{\partial \varepsilon(t_j - t_i - d^l)}{\partial t_j} \tag{4.10}$$

The above equation can be further simplified as,

$$\frac{\partial x_j(t_j)}{\partial t_j} = \sum_{i\epsilon\Gamma_j}\sum_l w_{ij}^l \varepsilon(t_j - t_i - d^l)\left(\frac{1}{t_j - t_i - d^l} - \frac{1}{\tau}\right) \tag{4.11}$$

For convenience, let's define δ_j as (where $j\epsilon J$):

$$\delta_j = \frac{\partial E}{\partial t_j}\frac{\partial t_j}{\partial x_j(t_j)} = \frac{-(t_j - t_j^d)}{\sum_{i\epsilon\Gamma_j}\sum_l w_{ij}^l \varepsilon(t_j - t_i - d^l)(\frac{1}{t_j - t_i - d^l} - \frac{1}{\tau})} \tag{4.12}$$

Now, equation 4.2 can be expressed as,

$$\Delta w_{ij}^k = -\eta \frac{\partial E}{\partial w_{ij}^k} = -\eta \delta_j \varepsilon(t_j - t_i - d^k) \tag{4.13}$$

Similar to δ_j, we can derive δ_i where $i\epsilon I$,

$$\delta_i = \frac{\partial E}{\partial t_i}\frac{\partial t_i}{\partial x_i(t_i)} \tag{4.14}$$

Since E is dependent on all the firing times t_j where $j\epsilon\Gamma^i$, the first derivative term in the above equation can be expanded as

$$\frac{\partial E}{\partial t_i} = \sum_{j\epsilon\Gamma^i} \frac{\partial E}{\partial t_j} \frac{\partial t_j}{\partial x_j(t_j)} \frac{\partial x_j(t_j)}{\partial t_i} = \sum_{j\epsilon\Gamma^i} \delta_j \frac{\partial x_j(t_j)}{\partial t_i} \tag{4.15}$$

The derivative term $\partial x_j(t_j)/\partial t_i$ can be expanded as,

$$\frac{\partial E}{\partial t_i} = \frac{\delta[\sum_{n\epsilon\Gamma_j} \sum_l w_{nj}^l \varepsilon(t_j - t_n - d^l)]}{\partial t_i} = \sum_l w_{ij}^l \frac{\partial \varepsilon(t_j - t_i - d^l)}{\partial t_i} \tag{4.16}$$

Another derivative term $\partial t_i/\partial x_i(t_i)$ can be calculated similar to equation 4.8 and equation 4.11 and expressed as,

$$\frac{\partial t_i}{\partial x_i(t_i)} = \frac{-1}{\sum_{h\epsilon\Gamma_i} \sum_l w_{hi}^l \varepsilon(t_i - t_h - d^l)(\frac{1}{t_h - t_i - d^l} - \frac{1}{\tau})} \tag{4.17}$$

Now, δ_i can be expresses as,

$$\delta_i = \frac{\sum_{j\epsilon\Gamma^i}[\delta_j \sum_l w_{ij}^l \varepsilon(t_j - t_i - d^l)(\frac{1}{t_j - t_i - d^l} - \frac{1}{\tau})]}{\sum_{h\epsilon\Gamma_i} \sum_l w_{hi}^l w_{hi}^l \varepsilon(t_i - t_h - d^l)(\frac{1}{t_h - t_i - d^l} - \frac{1}{\tau})} \tag{4.18}$$

Using above result, Δw_{hi}^k can be expressed as,

$$\Delta w_{hi}^k = -\eta \frac{\partial E}{\partial w_{hi}^k} = -\eta \frac{\partial E}{\partial t_i} \frac{\partial t_i}{\partial x_i(t_i)} \frac{\partial x_i(t_i)}{\partial w_{hi}^k} = -\eta \delta_i \varepsilon(t_i - t_h - d^k) \tag{4.19}$$

Important to note that this equation is quite similar to what we had calculated for Δw_{ij}^k.

SpikeProp algorithm is similar to the back-propagation algorithm where the synaptic weights are adjusted in either batch or incremental processing modes [7]. In the incremental mode, synaptic weights are updated after each training sample

processing. In the batch mode, the weights are updated after all the training samples are processed. In both cases, one epoch is counted when all the training samples are processed. In the original work by Bohte [3], SpikeProp was applied on the incremental mode. Later, both modes are analyzed in detail and a comparative result has been shown in [7]. In this research work, only batch mode is used for simulations.

4.2 Fast Learning based on Momentum [35]

Convergence of SpikeProp algorithm can be made faster by adding momentum to it. Change of weight in any particular iteration can be formulated as,

$$\Delta w_{ij}^k = -\eta \frac{\partial E}{\partial w_{ij}^k} + \alpha(\Delta w_{ij}^k)_{prev} \tag{4.20}$$

Where α is the momentum parameter. It is used to control the convergence. High value of α might lead to instability whereas low value of α causes slow convergence.

4.3 Fast Learning based on RProp [20]

RProp is the abbreviation for resilient propagation. It is a learning rate adjustment algorithm which works on existing neural network training algorithm. RProp is based on the sign of the gradient $\partial E/\partial w_{ij}^k$ but doesn't depend on the magnitude.

The learning rate in RProp is updated according to following formulas,

$$\Delta_{ij}^k = \begin{cases} \eta^+ . (\Delta_{ij}^k)_{prev}, & \text{if } (\frac{\partial E}{\partial w_{ij}^k})_{prev} \cdot \frac{\partial E}{\partial w_{ij}^k} > 0. \\ \eta^- . (\Delta_{ij}^k)_{prev}, & \text{if } (\frac{\partial E}{\partial w_{ij}^k})_{prev} \cdot \frac{\partial E}{\partial w_{ij}^k} < 0. \\ (\Delta_{ij}^k)_{prev}, & \text{otherwise.} \end{cases} \tag{4.21}$$

$$\Delta w_{ij}^k = \begin{cases} -\Delta_{ij}^k, & \text{if } \frac{\partial E}{\partial w_{ij}^k} > 0. \\ +\Delta_{ij}^k, & \text{if } \frac{\partial E}{\partial w_{ij}^k} < 0. \\ 0, & \text{otherwise.} \end{cases} \quad (4.22)$$

η^+ is generally kept between 1 and 2 and η^- should be between 0 and 1. The first equation says that learning rate keeps on increasing with rate η^+ if minimum is not crossed. Once minimum is crossed, learning rate decreases and is governed by η^-s. Second equation determines the sign of the weight change.

4.4 Fast Learning based on QuickProp [20]

Back-propagation works by calculating the partial first derivative of overall error with respect to each weight. If we take infinitesimal steps down the gradient, we are guaranteed to reach local minimum. If we want the solution in the shortest possible time, we don't want to take infinitesimal step, instead we should take largest steps possible without overshooting the solution. Partial first derivative tells very little about how large a step can be taken. In QuickProp, we utilize second order derivative to perform large steps. There are two main assumptions: first that the error vs weight curve for which weight can be approximated by a parabola whose arms open upwards; second, that change in slope with respect to a weight is not affected by other weight changes. The computation of QuickProp follows following simple formula,

$$\Delta w_{ij}^k = \begin{cases} \eta^Q.(w_{ij}^k)_{prev}, & \text{when } (\Delta w_{ij}^k)_{prev} \neq 0. \\ -\eta.\frac{\partial E}{\partial w_{ij}^k}, & \text{when } (\Delta w_{ij}^k)_{prev} = 0. \end{cases} \quad (4.23)$$

Where η^Q the QuickProp learning rate is computed adaptively based on formula,

$$\eta^Q = \frac{\frac{\partial E}{\partial w_{ij}^k}}{(\frac{\partial E}{\partial w_{ij}^k})_{prev} - \frac{\partial E}{\partial w_{ij}^k}} \tag{4.24}$$

There is a problem with this rate adaption technique. It occurs when $\frac{\partial E}{\partial w_{ij}^k}$ and $(\frac{\partial E}{\partial w_{ij}^k})_{prev}$ have same sign but $\frac{\partial E}{\partial w_{ij}^k}$ is larger in magnitude. This leads to moving away from local minimum and towards local maximum. We need to set a heuristic limitation to avoid this problem. We can set a rule such that we reset the weight change value (equal to zero) whenever such condition occurs. Another problem comes when $(\frac{\partial E}{\partial w_{ij}^k})_{prev}$ and $\frac{\partial E}{\partial w_{ij}^k}$ are almost same in value. This can cause a very high learning rate which can lead to instability. To avoid this problem, we can set a parameter μ, the maximum growth parameter. Now weight step is not allowed to be greater in magnitude than μ times the previous step for that weight.

5. PROPOSED ALGORITHMS FOR TRAINING DISCRETE WEIGHTS

The SpikeProp algorithm derived by Bohte [3], works on continuous parameters. What if we would like to implement it on digital VLSI chips? The only way to do that is by making all the parameters discrete that includes weights, firing times & internal parameters of neurons. To maintain the accuracy of SNNs, we need to use large number of bits to represent discrete weights. Since, power consumed in multiplication block is proportional to the size of multiplication factor. That implies that large number of bits for weights would consume lots of power on silicon chip. On the other hand fewer number of bits would make SNN less accurate. This happens because the error surface is not smooth, making all the weights discrete might result in a significant change in the behavior of SNNs. To overcome this problem, new techniques to train SNNs with discrete weights are proposed in this chapter.

5.1 Motivation

Recently, several efforts have been made to implement SNNs on FPGA [23, 24]. There has not been any research work for training discrete weight SNNs. However, there are some works to train discrete weight sigmoidal neural networks [33, 15]. The works in [33, 15] use traditional backpropagation and find out the weight that gives maximum benefit in error. Unlike SNNs, inputs and outputs in sigmoidal neural networks are monotonically related. In the case of SNNs, error vs weight curve would be non-linear and rough because a slight change in weight can cause big change in firing time (as shown in the figure 5.2).

The overall system (as shown in figure 5.1) indicates that training algorithm works on software and weights are fed to the digital FPGA or VLSI chip. There is a possibility that this training algorithm can be implemented in digital VLSI but that

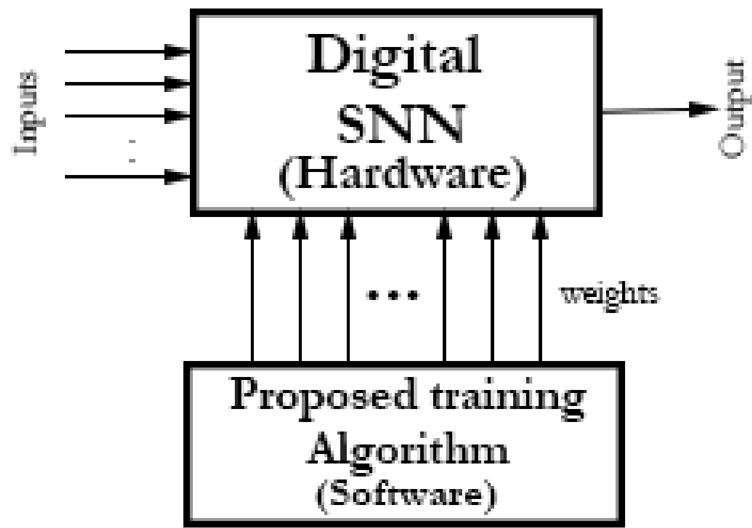

Figure 5.1: Discrete SNN system

is not intended in this research work. This aspect of research is kept for future work in this area.

5.2 Basic Idea

A set of input vectors is applied to the network and in response, a set of output vectors is produced by the network. The error is the mean squared difference between desired outputs and actual outputs (equation 4.1). The network is trained in the following manner. A set of input vectors is presented to the network, each vector being propagated forward through the network to produce an output vector. A set of error vectors is then presented to the network and propagated backwards. After all the input vectors are presented, an update phase is initiated. During the update phase, in accordance with the results of the derived algorithm, the selective change selects the other weight value if selecting the other weight value will decrease the total error. This approach finds local minima nearest to the starting point on the

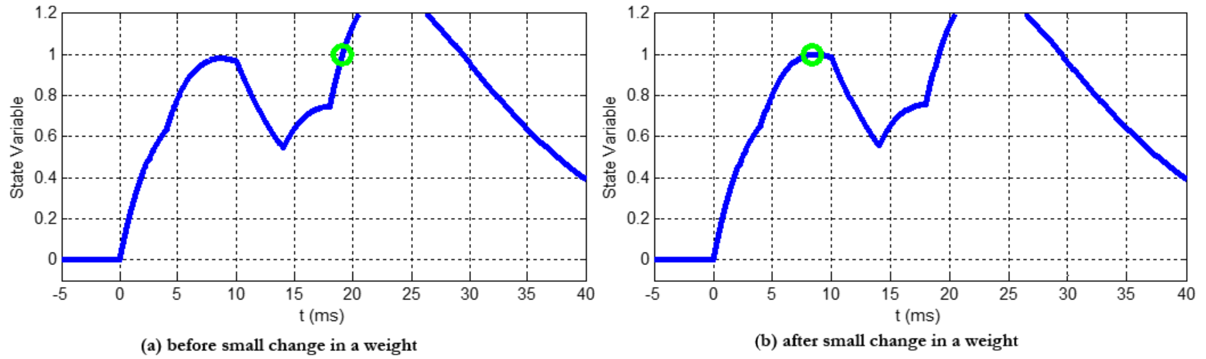

Figure 5.2: Figure showing a big change in firing time of a neuron because of a small change in synaptic weight, the circle indicates the firing instant

error surface. The result is very similar to what we get in SpikeProp algorithm.

How to choose a weight change that would decrease the overall error? There are different techniques that can trigger this change. The most basic approach would be to check with all the weight values. Pick a weight, change it by a quantization step and check the overall error. We need to do this for each weight and then we choose the weight, which gives the most decrement in the overall error. This approach works but it is an extensive search among the weights. And also we are changing only one weight in an iteration. Obviously, this is not an efficient solution.

Another approach is to use result of SpikeProp algorithm and use it for guided search among weights. Our proposed algorithm is based on this idea. Four algorithms, which are explained in this chapter are analyzed for this purpose. In all of these algorithms, predicted error $(\Delta Epred)_{ij}^k$ corresponding to weight w_{ij}^k is calculated as:

$$(\Delta Epred)_{ij}^k = (\frac{\partial E}{\partial w_{ij}^k}) * (\pm \Delta w), \tag{5.1}$$

Where Δw is the quantization step assumed for the discrete weights and is gradient calculated by SpikeProp algorithm (see equation 4.13 and 4.19). In this research, Δw is assumed equal to 0.1. For each w_{ij}^k, two $(\Delta Epred)_{ij}^k$ are calculated: first for increment of $+\Delta w$ and second for increment of $-\Delta w$. Consequently, there would be $2 * N$ number of $(\Delta Epred)_{ij}^k$ for N number of weights.

5.3 Single Weight Single Step Algorithm

Algorithm 1 Single Weight Single Step Algorithm

1: Calculate predicted Error change $(\Delta Epred)_{ij}^k$ for each weight (w_{ij}^k) using SpikeProp Algorithm.
2: Arrange all weights (w_{ij}^k) in increasing order of their $\Delta Epred$ values.
3: Find the first weight and its change $(\pm\Delta w)$ that gives negative $\Delta Eactual$.
4: Apply the change $(\pm\Delta w)$ to the chosen weight (w_{ij}^k).

In this algorithm, we chose a single weight that gives benefit in error value with the help of SpikeProp algorithm. Then we change that weight by one quantization step. Since this algorithm changes only one weight in one iteration, its convergence is very slow. Few more modifications need to be inserted into the above algorithm to speed up the convergence.

5.4 Single Weight Multiple Steps Algorithm

This algorithm is a little modification of Single Weight Single Step. In this algorithm we don't stop once we pick a weight. We further analyze that particular weight for number of steps it can allow to give benefit in the error. This algorithm stops when we reach a quantization step that gives less benefit in error value.

Algorithm 2 Single Weight Multiple Steps Algorithm

1: Calculate predicted Error change $(\Delta Epred)^k_{ij}$ for each weight (w^k_{ij}) using Spike-Prop Algorithm.
2: Arrange all weights (w^k_{ij}) in increasing order of their $\Delta Epred$ values.
3: Find the first weight and its change $(\pm\Delta w)$ that gives negative $\Delta Eactual$.
4: Start increasing the weight step $(\pm\Delta w * k)$ as long as $\Delta Eactual$ is decreasing. (here $k = 1, 2, 3...$)
5: Apply the change $(\pm\Delta w * k)$ to the chosen weight (w^k_{ij}).

5.5 Multiple Weights Single Step Algorithm

This algorithm is another improved version of Single Weight Single Step algorithm. In this, we don't stop after choosing a weight. We keep on looking for more weights until we encounter a series of weight change that doesn't improve the $(\Delta Eactual)^k_{ij}$. That means this algorithm changes one or more than one weight in one iteration. Change in all of these weights is only one quantization step. This algorithm stops when we get less benefit in error value. To speed up overall execution time, we included some heuristics that causes algorithm to stop when total number of weights that does not give benefit in error is more than particular value (look at the algorithm 3).

5.6 Multiple Weights Multiple Steps Algorithm

This algorithm utilizes quality of both Algorithm 2 and Algorithm 3. That means, it not only chooses the weights that need to be changed, it also look for the maximum quantization step that can be taken for each chosen weight.

Algorithm 3 Multiple Weights Single Step Algorithm

1: Calculate predicted Error change $(\Delta Epred)_{ij}^{k}$ for each weight (w_{ij}^{k}) using SpikeProp Algorithm.
2: Arrange all weights (w_{ij}^{k}) in increasing order of their $\Delta Epred$ values.
3: Find the first weight and its change $(\pm\Delta w)$ that gives negative $\Delta Eactual$.
4: total missing = 0
5: **while** ($\Delta Eactual$ or total missing < 20) **do**
6: Select the next weight and its change $(\pm\Delta w)$. Calculate $\Delta Eactual$.
7: **if** $\Delta Eactual < (\Delta Eactual)_{prev}$ **then**
8: Choose the current weight
9: total missing = 0
10: **else** total missing = total missing +1
11: **end if**
12: **end while**
13: Apply the change $(\pm\Delta w)$ to all the chosen weights (w_{ij}^{k}).

Algorithm 4 Multiple Weights Multiple Steps Algorithm

1: Figure out the weights that need to be changed using Multiple Weight Single Step Algorithm
2: **for** each w_{ij}^{k} selected in previous step **do**
3: Start increasing the weight step $(\pm\Delta w * k)$ as long as $\Delta Eactual$ is decreasing. (here $k = 1, 2, 3...$)
4: Apply the change $(\pm\Delta w * k)$ to the chosen weight (w_{ij}^{k}).
5: **end for**

6. RESULTS AND CONCLUSION

This chapter includes the benchmarks and results of some standard datasets. Platform used for this research is MATLAB. The machine used for simulation has 12 GB memory and no GPU. The maximum execution time is set to 72 hours.

6.1 XOR Problem

The XOR problem is the most basic non-linear classification problem. This is usually used to test training algorithm. This data contains 2 binary inputs and one binary output. If two inputs are identical, i.e. $\{0, 0\}$ or $\{1, 1\}$, then output is 0. If two inputs are different, i.e. $\{0, 1\}$ or $\{1, 0\}$, then output is 1.

6.1.1 Input Encoding

Since SNNs considers inputs and outputs as spike times, we need to encode inputs and outputs as spike times. Bohte in [3], used encoding as input *bit* 1 assigning to 0 ms spike time and input *bit* 0 assigning to 6 ms spike time. Difference between these two spike times is called *encoding interval*, which is 6ms in this case.

6.1.2 Network Setup

Bohte [3] selected the time decay constant τ for spike response function (in equation 3.1) by trial and error. A small value of τ results in a narrow-shaped post synaptic potential (PSP). If the PSP is too narrow, PSPs resulting from presynaptic spikes from various presynaptic neurons delayed by different synapses may not overlap. In such a case, summation of the PSPs does not increase the internal state and, as a result, the postsynaptic neuron does not fire. Too large a value of τ results in a very wide PSPs and the internal state of the postsynaptic neuron takes longer to decrease to the resting potential. The increased width of the PSP leads to too much

overlap between subsequent PSPs. As a result, the postsynaptic neuron fires equally early for both small and large values of synaptic weights thus becoming insensitive to changes in synaptic weights during the SNN training [7]. Bohte suggested that value of τ, which is membrane potential decay time constant, should be selected slightly higher than encoding interval, which is 6 ms. Therefore, τ is kept equal to 7 ms.

16 synapses are used between the two neurons. Delays of these synaptic connections are spread between 1 ms and 16 ms with 1 ms interval. We can calculate maximum and minimum possible spike time for the output neuron.

Network structure is 3 layered SNN (see figure 6.1). Apart from the input neurons carrying input information, there must be an extra bias neuron. In [29], it is explained that it is compulsory to have a bias neuron at the input because SNN carry information in the form of difference in firing time instances. Without the bias neuron, the problem becomes impossible to solve independent of the learning algorithm. Therefore, the input layer contains 3 neurons (2 for input data and 1 for bias) and output layer contains 1 neuron. Hidden layer contains 5 neurons similar to [3].

Discrete weights are selected as 4 bits wide (from -3.5 to 4.0 with 0.5 step).

6.1.3 Output Encoding

As a result of delays, maximum possible spike time for the output neuron would be $6 + 2 \times 16 = 38$ ms (6 ms is the latest firing time at the input and each synapse can cause a maximum of 16 ms of delay) and minimum possible spike time would be $0 + 2 \times 1 = 2$ ms. Bohte [3] selected the desired output spike times (t_j) for SNN training towards the middle of this range using a trial and error process. When a small (early) value is selected for the desired output spike time, the delayed input spikes will not have any effect on the output spike computation. When a large (late)

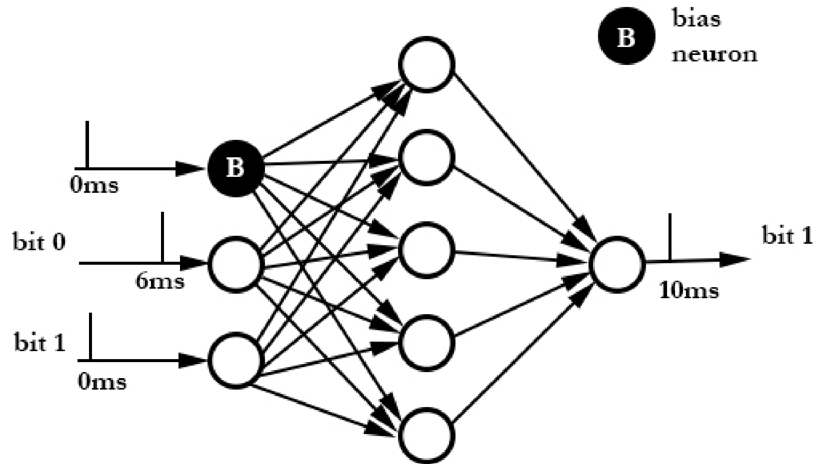

Figure 6.1: One training sample for XOR problem

value is selected for the desired output spike time the simulation time becomes large which reduces the computational efficiency [7]. Output values 1 and 0 are encoded as output spike times 10 ms and 16 ms respectively. Table 6.1 shows input and output encoding for all 4 samples.

Table 6.1: Input and output patterns for XOR problem

First Input Firing time	Second Input Firing Time	Output Firing Time
0 ms (bit 1)	0 ms (bit 1)	16 ms (bit 0)
0 ms (bit 1)	6 ms (bit 0)	10 ms (bit 1)
6 ms (bit 0)	0 ms (bit 1)	10 ms (bit 1)
6 ms (bit 0)	6 ms (bit 0)	16 ms (bit 0)

6.1.4 Convergence Criteria

Most results in the literature are reported for a limiting convergence mean square error (MSE) value of 0.5 with the exception of Bohte [3] who reported a sum squared error (SSE) of 1.0 for the original SpikeProp algorithm within 250 epochs. For the XOR problem, since there are only four training instances and one output neuron and SSE 1.0 is equivalent to an MSE of $SSE/(Number\ of\ training\ instances) = 1.0/4 = 0.25$. In this research work, MSE value of 0.5 is used and the upper limit for the number of epochs is set to 500.

6.1.5 Comparison of Discrete Weight Algorithms in terms of Execution Time

Table 6.2: Comparison of different training algorithm on XOR problem

Algorithm	Number of iterations	execution time
SpikeProp Algorithm (Rprop)	25	18.72 *sec*
Single Weight Single Step	66	47.39 *sec*
Single Weight Multiple Steps	29	31.58 *sec*
Multiple Weights Single Step	10	23.87 *sec*
Multiple Weights Multiple Steps	6	20.76 *sec*

Figure 6.2 and table 6.2 shows a comparison among four proposed algorithms. It is clearly seen that Multiple Weights Multiple Steps performs the best.

6.2 Other Standard Benchmarks

To analyze the proposed algorithms, 3 classification benchmarks are analyzed. 1. Fisher Iris classification problem, and 2. Wine recognition problem. All benchmarks

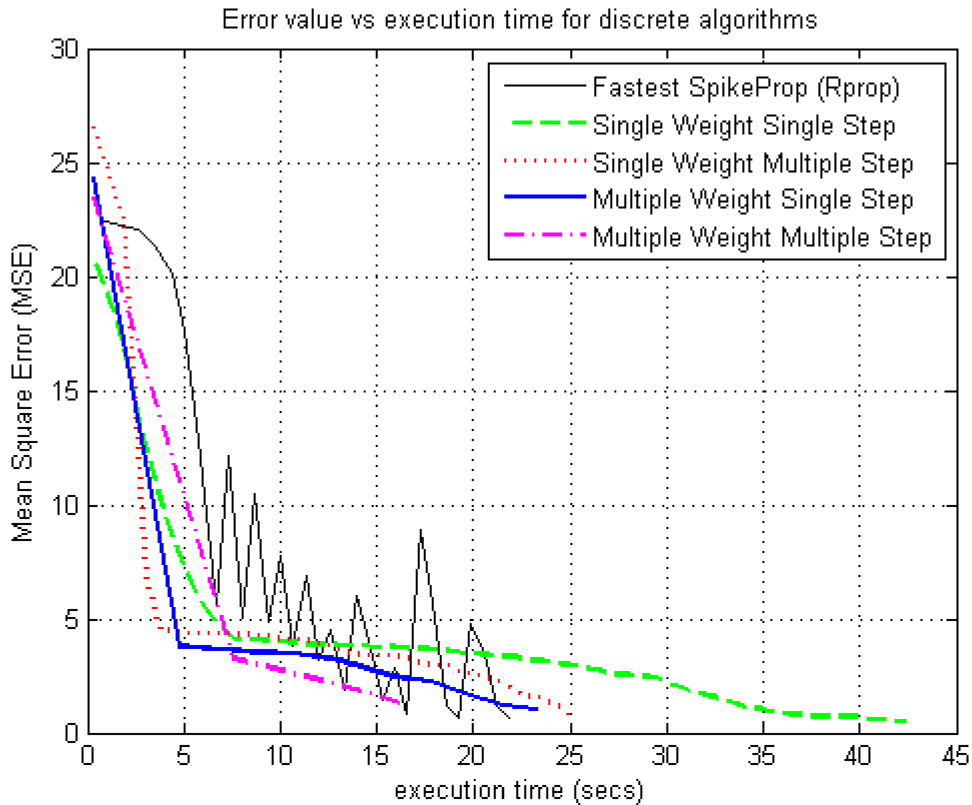

Figure 6.2: Comparison of discrete weight training algorithms

were taken from the UCI Machine Learning Repository [2].

6.3 Fisher Iris Dataset

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced as an example of discriminant analysis [5]. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.

6.3.1 Input Encoding

Unlike the XOR problem, features of this classification problem are real numbers. Therefore, it cannot be encoded easily. Ghosh [7], analyzed a linear encoding scheme where value of each input feature is converted proportionately to a spike time in the range of 0-10 ms. However, for SpikeProp this scheme resulted in classification accuracies approximately 30-50 % lower than those obtained using population encoding, where range of spike times seems to represent the inputs more accurately and is therefore used in this research (similar to [3, 7]).

In population encoding, each feature is encoded separately by $M > 2$ identically shaped overlapping Gaussian functions centered at M locations (see figure 6.3). In the figure, analog input on x axis is the input feature. 4 values on the output are the values of guassian curve at analog input. The spread of the Gaussian function is $(1/\gamma)(I_{max} - I_{min})/(M - 2)$ where I_{max} and I_{min} are the maximum and minimum values for the encoded features, respectively and γ is adjustment factor. The center of the ith Gaussian curve is located at $I_{min} + [(2i - 3)/2][(I_{max} - I_{min})/(M - 2)]$ where $i \epsilon \{1, 2, ..., M\}$.

Due to compression, two different input feature values close to each other may yield the same spike time response and, therefore, may not be differentiable in the encoding process. This is not desirable when classes are not linearly separable. Population encoding transforms each input feature value into M different spike times which proportionately increases the differentiability. There is a trade-off between differentiability and computational effort. Because as M increases, differentiability increases but number of input neurons and synaptic weights increases too. That leads to higher computational effort.

For Fisher Iris problem, $M = 4$ is used. That means 4 features would generate

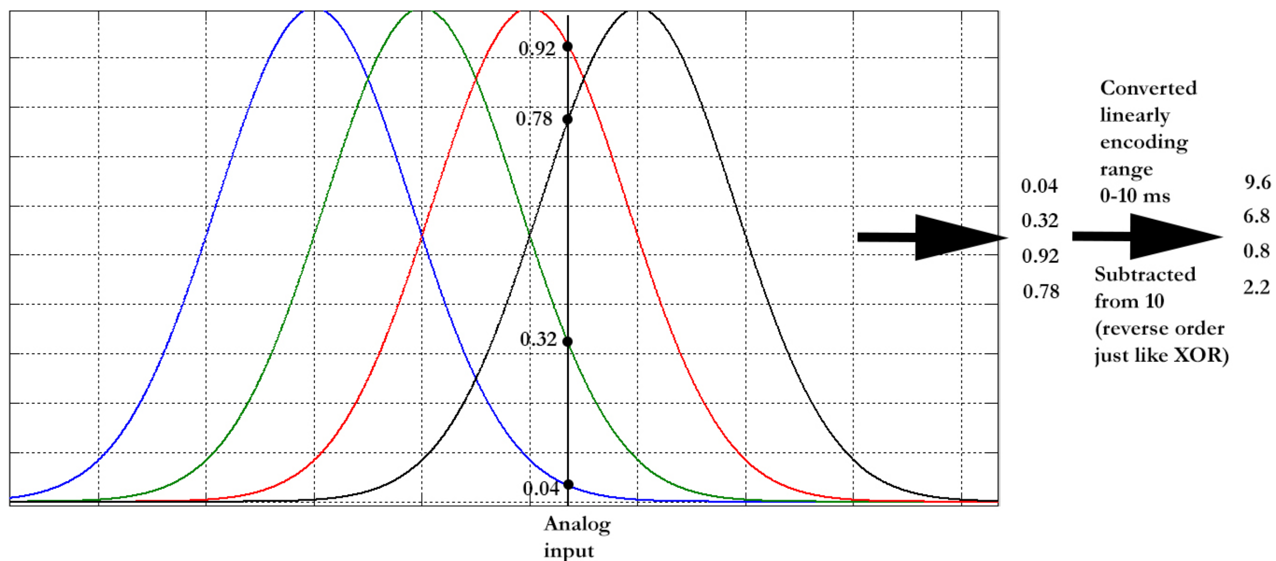

Figure 6.3: Population encoding scheme

16 input spikes pattern.

6.3.2 Network Setup

Number of input neurons would be $16(input\ spikes) + 1(bias) = 17$. In two ways, we can setup output neurons. First, three output neurons are separately encoded as three classes. Second, only one output neuron represent three classes by three spike times. It is shown in [7], that second scheme yields higher classification accuracy. Therefore, only one output neuron is used for the research. Number of neurons in the hidden layer is kept as 8 (nearly middle of number of input neurons and number of output neurons).

Synaptic connections are kept same as the XOR problem (section 6.1.2). That means we used 16 synaptic connections with delay varying from 0 ms to 15 ms with 1 ms step.

Discrete weights are kept similar to the XOR problem (section 6.1.2). That means

discrete weights are selected as 4 bit wide (from -3.5 to 4.0 with 0.5 step).

As explained in section 6.1.2, τ is selected a little higher than encoding interval (10 ms in this case). So, τ is made equal to 11 ms.

6.3.3 Output Encoding

Output spikes are encoded as values which are close to midpoints of the range of output spike times. As a result of delays, maximum possible spike time for the output neuron would be $10 + 2 \times 16 = 42$ ms (10 ms is the latest firing time at the input, 16 ms is the maximum delay at the synapse and there are 2 synaptic connection between input and output neuron) and minimum possible spike time would be $0 + 2 \times 1 = 2$ ms (0 ms is the earliest firing time at the input, 16 ms is the minimum delay at the synapse and there are 2 synaptic connection between input and output neuron). Therefore output spikes of 15 ms, 20 ms and 25 ms are encoded to three distinct classes.

6.3.4 Convergence Criteria

Mean square error (MSE) is kept below MSE corresponding to 95% accuracy. Using equation 4.1, for calculating average MSE for 95% accuracy: $MSE < (1/2) \times (100 - 95) \times \frac{3^2}{100} = 0.225 \approx 0.25$. In this dataset setup, minimum error at the output, when the prediction is inaccurate, would be 2.5 ms. Because output spike times are set as 15 ms, 20 ms and 25 ms, which is 5 ms apart. Therefore error has to be more than 2.5 ms for wrong prediction. To be on the safe side, we approximated it to 3 ms.

6.3.5 Results

Figure 6.4 shows that Multiple Weights Multiple Steps algorithm performs best among other proposed algorithms. The numerical results are shown in the table 6.3.

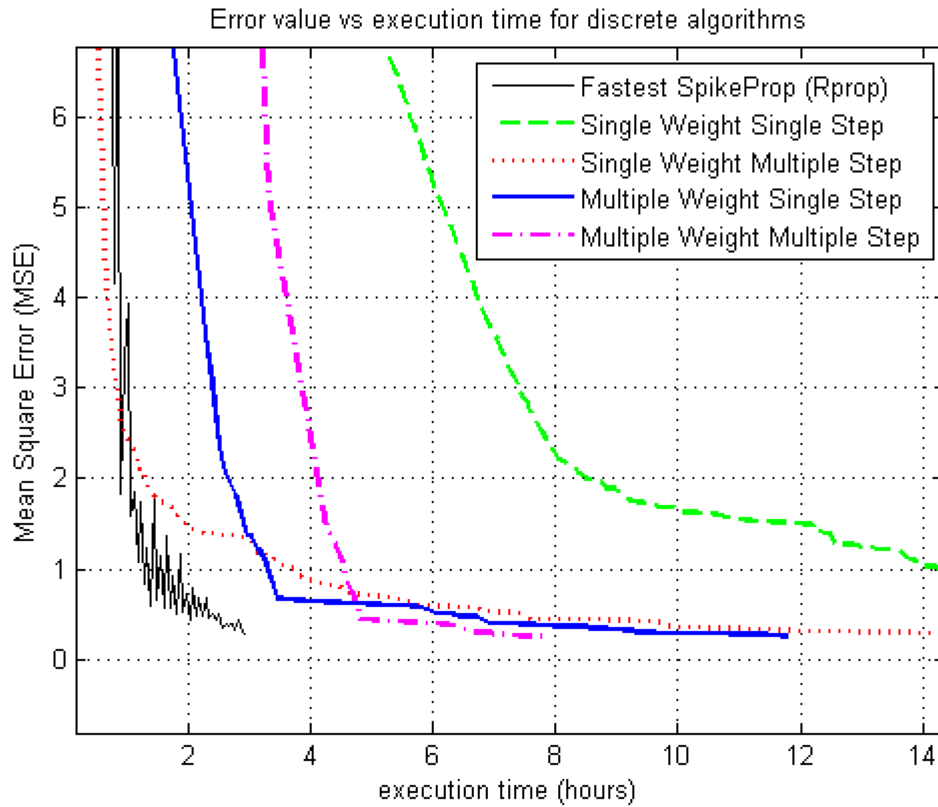

Figure 6.4: Comparison of discrete weight training algorithms for Fisher Iris classification problem

Table 6.3: Comparison of different training algorithm on Fisher Iris Problem

Algorithm	Number of iterations	execution time	accuracy in training set	accuracy in test set
SpikeProp Algorithm (Rprop)	66	2.98 *hours*	96%	92%
Single Weight Single Step	250	26.50 *hours*	98%	89%
Single Weight Multiple Steps	189	19.42 *hours*	97%	83%
Multiple Weights Single Step	59	12.59 *hours*	99%	94%
Multiple Weights Multiple Steps	29	8.05 *hours*	97%	89%

6.4 Wine Recognition Dataset

This dataset is using chemical analysis to determine the origin of wines. The number of samples is 178. There are 13 features for each sample. All feature values are real numbers. Samples are divided among 3 classes.

Since number of classes is 3 which is similar to Fisher Iris. Output encoding is similar to Fisher Iris, which is 15 ms, 20 ms and 25 ms. Number of features are 13 which is quite higher than previous datasets. In population encoding, value of M is reduced to 3. This is to avoid high computational effort. Table 6.4 shows the results comparing all the proposed algorithms and Rprop variation of SpikeProp algorithm.

Table 6.4: Comparison of different training algorithm on Wine Recognition Problem

Algorithm	Number of iterations	execution time	Accuracy	Convergence in 72 hour
SpikeProp Algorithm (Rprop)	89	12.54 *hours*	89%	converged
Single Weight Single Step	> 361	> 72 *hours*	87%	did not converge
Single Weight Multiple Steps	> 187	> 72 *hours*	93%	did not converge
Multiple Weights Single Step	211	61.35 *hours*	96%	converged
Multiple Weights Multiple Steps	> 238	> 72 *hours*	93%	did not converge

6.5 Speed Up Ideas

The results of the Wine Recognition dataset show that proposed discrete weight algorithms takes a huge amount of time when executed on SNNs of large network size. There are multiple ways we can make runtime shorter. First of all, the algorithms are implemented in MATLAB. If we implement the same algorithms on a lower level programming language such as C/C++ then execution time is expected to be much lower. There are parallelisms in these algorithms. We may utilize GPUs to

reduce the execution time further. During execution, most of the time is consumed in the feedforward phase of the algorithm. This is because we are looking into small timestamps while calculating the firing time. Longer timestamp would definetely benefit execution time but there is a tradeoff between execution time and stability of the algorithm. Also, instead of looking for whole timescale window, we can reduce our window based on firing times of neurons in the previous layer.

6.6 Conclusion

By looking at the results, it is quite clear that Multiple Weights Single Step and Multiple Weights Multiple Steps work well for training of spiking neural networks with discrete weights. Training of discrete weight SNN takes a little more computational effort but is amenable to the desired digital implementation on silicon. In future, approaches should be evaluated for larger dataset. Current proposed algorithm becomes slow exponentially with size of dataset. Training of discrete weight SNN is a hard problem therefore other sophisticated discrete algorithms (such as genetic algorithm [30]) should be analyzed as well.

REFERENCES

[1] James A Anderson and Edward Rosenfeld. *Neurocomputing*, volume 2. MIT press, 1993.

[2] K. Bache and M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2013.

[3] Sander M Bohte, Joost N Kok, and Han La Poutre. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17–37, 2002.

[4] Olaf Booij et al. A gradient descent rule for spiking neurons emitting multiple spikes. *Information Processing Letters*, 95(6):552–558, 2005.

[5] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[6] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[7] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Improved spiking neural networks for eeg classification and epilepsy and seizure detection. *Integrated Computer-Aided Engineering*, 14(3):187–212, 2007.

[8] Samanwoy Ghosh-Dastidar and Hojjat Adeli. A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection. *Neural Networks*, 22(10):1419–1431, 2009.

[9] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking neural networks. *International journal of neural systems*, 19(04):295–308, 2009.

[10] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605. IEEE, 1989.

[11] Suzana Herculano-Houzel. The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 3, 2009.

[12] Geoffrey Hinton. The next generation of neural networks (google techtalk). `https://www.youtube.com/watch?v=AyzOUbkUf3M`, 2007.

[13] Michio Kaku. *The future of the mind: The scientific quest to understand, enhance, and empower the mind*. Random House LLC, 2014.

[14] Kiyoshi Kawaguchi. A multithreaded software model for backpropagation neural network applications. *ETD Collection for University of Texas, El Paso*, Jan 2000.

[15] Keisuke Kojima, Kazuo Kyuma, Toshio Shinnishi, Shuichi Tai, and Masanobu Takahashi. Learning method for neural network having discrete interconnection strengths, July 1 1997. US Patent 5,644,681.

[16] Wolfgang Maass. On the computational power of noisy spiking neurons. *Advances in neural information processing systems*, pages 211–217, 1996.

[17] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[18] Wolfgang Maass and Christopher M Bishop. *Pulsed neural networks*. MIT press, 2001.

[19] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[20] Sam McKennoch, Dingding Liu, and Linda G Bushnell. Fast modifications of the spikeprop algorithm. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 3970–3977. IEEE, 2006.

[21] Thomas Natschläger and Berthold Ruf. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9(3):319–332, 1998.

[22] Thomas Natschlger. Networks of spiking neurons: A new generation of neural network models. `http://www.igi.tugraz.at/tnatschl/online/3rd_gen_eng/3rd_gen_eng.html`, dec 1998.

[23] Marco Nuño-Maganda and Cesar Torres-Huitzil. A temporal coding hardware implementation for spiking neural networks. *ACM SIGARCH Computer Architecture News*, 38(4):2–7, 2011.

[24] Martin J Pearson, Anthony G Pipe, Benjamin Mitchinson, Kevin Gurney, Chris Melhuish, Ian Gilhespy, and Mokhtar Nibouche. Implementing spiking neural networks for real-time signal-processing and control applications: a model-validated fpga approach. *Neural Networks, IEEE Transactions on*, 18(5):1472–1487, 2007.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 1988.

[26] Benjamin Schrauwen and Jan Van Campenhout. Extending spikeprop. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 1. IEEE, 2004.

[27] Benjamin Schrauwen and Jan Van Campenhout. Improving spikeprop: enhancements to an error-backpropagation rule for spiking neural networks. In

Proceedings of the 15th ProRISC workshop, volume 11, 2004.

[28] Sergio M Silva and Antonio E Ruano. Application of levenberg-marquardt method to the training of spiking neural networks. In *Neural Networks and Brain, 2005. ICNN&B'05. International Conference on*, volume 3, pages 1354–1358. IEEE, 2005.

[29] Ioana Sporea and André Grüning. Reference time in spikeprop. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1090–1092. IEEE, 2011.

[30] Kenneth O Stanley and Risto Miikkulainen. Efficient evolution of neural network topologies. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1757–1762. IEEE, 2002.

[31] Rand S Swenson. Review of clinical and functional neuroscience. *Dartmouth Medical School. Retrieved November*, 18:2012, 2006.

[32] Simon Thorpe, Arnaud Delorme, and Rufin Van Rullen. Spike-based strategies for rapid processing. *Neural networks*, 14(6):715–725, 2001.

[33] Max S Tomlinson Jr. Discrete weight neural network, April 17 1990. US Patent 4,918,618.

[34] Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. An fpga platform for on-line topology exploration of spiking neural networks. *Microprocessors and microsystems*, 29(5):211–223, 2005.

[35] Jianguo Xin and Mark J Embrechts. Supervised learning with spiking neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, pages 1772–1777. IEEE, 2001.