ULTRA LOW POWER TECHNIQUES FOR MACHINE LEARNING ON THE EDGE

by

Md Munir Hasan

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2022

Approved by:

____

Dr. Jeremy Holleman

____

Dr. Andrew Willis

____

Dr. Hamed Tabkhi

____

Dr. A. Suzanne Boyd

ABSTRACT

MD MUNIR HASAN. Ultra Low Power Techniques for Machine Learning on the Edge. (Under the direction of DR. JEREMY HOLLEMAN)

Deep learning has become an integral part of machine learning. It has radically transformed our lives in healthcare, automotive systems, human computer interaction etc. Although, deep learning requires a tremendous amount of compute power and resources, the success of deep learning in solving complex tasks has generated a serious interest in deploying deep learning models in edge sensors and IoT devices. However, that goal presents serious challenges. Typical deep learning models require very powerful hardware with large memories and high power consumption. However, sensor systems and IoT devices at the edge are heavily resource constrained. They have a limited amount of compute power and on-board memory. That is why many efforts are being actively pursued to optimize the deep learning models so that they fit into the limited resources of edge devices.

In this dissertation, I explore different techniques for achieving ultra low power hardware for enabling machine learning at the edge. There have been numerous advances in circuit design techniques such as subthreshold analog computing, in memory computation, etc., for very low power applications. Emerging devices and circuits to integrate those devices into low power applications have shown promising results for custom hardware based edge devices. In this study, I explore neuromorphic techniques that lower the power consumption of the computation hardware without significantly degrading the performance. I draw inspiration from biology to design low power circuits, specifically spiking neurons of the biological nervous system. I explore biologically relevant neurons, circuits and

learning rules to minimize computation and power consumption for machine learning at the edge device and sensors.

I have proposed a modification to a sparse coding algorithm that decreases the number of circuits for hardware implementation. I have proposed an analog spiking neuron design which can display a variety of spiking behaviors. The circuit is compact, low power, uses low supply voltage and has high power efficiency, which improves the state of the art. Analog circuits suffer from the problem of leakage current, which makes the design of synaptic circuits difficult. I have proposed a leakage current mitigation technique in a synaptic circuit array and provide simulation experiments to show its efficacy. Spiking neural network is still an emerging branch of machine learning. Hence, there are a lack of necessary tools for simulation. Although there are many hardware neuron circuits, there are no spiking neural network simulators that can account for the hardware non-idealities in the simulation. When it comes to the performance of robust circuits and systems with predictable outcomes through simulation, the inclusion of hardware non-idealities is a must. Given the complexity of spiking neural network hardware, it is not an easy task. I propose phase-plane method for easily extracting hardware non-idealities and using them in the existing simulator to simulate spiking neural networks. The proposed method is computationally inexpensive and easily integrates with spiking network simulators. I compare the spice simulation and phase-plane simulation of spiking neural networks to show that phase-plane can indeed account for hardware non-idealities.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Jeremy Holleman for his continuous support and advice. I am grateful to him for his guidance in this work. I would like to thank the Graduate Assistant Support Plan (GASP) for financial support in my graduate studies. I would also like to thank the committee members Dr. Andrew Willis, Dr. Hamed Tabkhi and Dr. A. Suzanne Boyd for their time and constructive discussion in shaping this dissertation.

This work is dedicated to my loving parents.

Table of Contents

List of Tables

List of Figures

Glossary

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
MPW	Multi Project Wafer. This is a way to share a silicon wafer for different designs. It allows individuals to submit their chip designs for fabrication without having to buy the whole wafer.
PDK	Process Development Kit. This is a collection of silicon foundry model files which is provided by the fabrication foundry. The PDK contains the models required for circuit simulation. It also contains the design rules for layout.
RF	Receptive Field
Skywater	The name of the PDK that Google is using for open source chip fabrication
SNN	Spiking Neural Network

Chapter 1

Introduction

When it comes to designing low power, compact devices and systems, biological systems offer exciting inspiration that the engineering community can benefit from. Using biologically relevant devices, algorithms and models to solve machine learning tasks is commonly known as neuromorphic engineering. Neuromorphic computing has recently emerged as a promising alternative to von Neumann systems. In von Neumann systems, memory and computation are separate. A central processing unit is responsible for controlling the memory and computation. This architecture is based on a central clock, which executes instructions in a serial manner. As Moore's law is expected to come to an end, von Neumann-based computing systems will eventually not be able to meet the computational demand in the future. Tremendous amounts of data are being generated every day, which needs to be processed using artificially intelligent machines. Processing such vast amount data also means a greater demand on the power consuption and computing power.

On the other hand, analog computing techniques offer better power efficiency [3] compared to digital computing techniques. As a result, many neuromorphic systems [4, 5, 6] are based on analog computation techniques. On top of that, neuromorphic systems are highly parallel in nature. They also colocate memory and processing, which has the promise of overcoming the von

Neumann bottleneck [7]. A large amount of power is required to move data in and out of memory than it takes for actual computation, which is known as the von Neumann bottleneck. Instead of separating memory from computation, memory can be placed close to the computation in order to minimize data movement. This memory colocation is inspired by biology. As a result, neuromorphic engineering has become a common name in the field of machine learning.

Traditional artificial neural networks (ANN) use neurons that operate on continuous values. On the other hand, operations in neuromophic computing systems use spiking neurons where computation is based on spike events. Spiking neural networks (SNN) have emerged as a promising candidate for the next generation of neural networks [8]. Neurons in ANNs are rate-code-based models where continuous valued inputs are weighted and summed, after which a nonlinear function is applied to produce neuron output. However, in SNNs, spike events are integrated over time and an output event spike is generated when the integrated value crosses a threshold. A spiking neuron in an SNN is biologically plausible. Moreover, because of the neurons' event based nature, it is more energy efficient. There are also significant differences between the learning methods of ANNs and SNNs. Most SNNs are trained with biologically plausible learning rules such as Hebbian learning [9], spike timing dependent plasticity (STDP) learning rule [10] etc. whereas ANNs are trained using backpropagation rule [11].

1.1 Motivation

Recently, machine learning on the edge has become a very popular and practical concept [12]. There are several reasons for this popularity.

- Machine learning at the edge enables processing the data in real time. For offline processing, the data needs to be collected and then sent to the cloud servers or data processing stations. Directly processing data at the place of data collection removes a significant overhead and processing time. Critical technologies such as autonomous vehicles and medical devices can greatly benefit from real time machine learning at the edge.

- Sending data from sensor devices to cloud servers potentially presents a security risk. Cloud servers store sensitive personal user information, which is subject to adversarial attacks. By performing machine learning locally at the edge, the data storage and hence any security risk are eliminated.

There are several design considerations for machine learning on edge devices. The computing power and memory resources of the edge device are extremely limited. A typical edge sensor, for example an environmental sound detector or cough detector for biomedical data acquisition, has to operate on very limited power. These kinds of devices are typically run by coin cell batteries. If the power cost of computation is high, then the battery would run out very quickly. Furthermore, a wearable biomedical sensor has to be very compact in size. This puts a limit on the computing devices, battery size, and also the memory constraints available on board the device. Thus, edge machine learning in application specific integrated circuits (ASIC) is a very challenging task. SNN offers many desirable properties which edge machine learning can benefit from. SNNs are inherently event based systems which can provide energy efficient and robust decision making. Using the properties of biologically motivated spiking neural networks, I can develop machine learning systems that are capable of operating under strict energy and memory constraints.

1.2 Proposed Contributions

To meet the challenges of machine learning on the edge, I study and explore the following domains.

- *Develop memory efficient approximate computing algorithms:* Sparse coding is a biologically inspired unsupervised learning algorithm that potentially explains the sparse activity of the biological brain. As an engineering approach to reduce power consumption, sparse coding is gaining more and more interest. Moreover, it can be used as a feature discovery layer [13] of a convolutional neural network (CNN). Recently, a spiking version of sparse coding called SAILNet [14] has been proposed. SAILNet is particularly attractive because the learning rules are biologically plausible. Hence, a sparse coding algorithm such as SAILNet might become an important preprocessing step in spike based information processing systems. A memory-efficient version of the SAILNet algorithm is required for deployment in edge devices. I propose a modification [15] to the algorithm which reduces the memory footprint of the coding algorithm.

- *Design compact ultra low power neuron circuit for neuromorphic systems:* In order to pave the way for energy efficient intelligent edge devices based on spiking neurons, ultra low power SNN components are needed. The neuron is one key component in an SNN. For the neuron circuit an ultra low power, compact analog spiking neuron [16] in 130nm CMOS technology is presented in chapter 4.

- *Mitigation of Leakage current in Synaptic Array:* Analog circuits suffer from the problem of leakage current. For synaptic circuits, this leakage current presents a problem in the steady state response of the neuron. A technique for mitigating the leakage current and synaptic circuit array design is presented [17] in chapter 5. The synaptic array is designed using 130nm CMOS technology.

- *Develop simulation techniques to account for hardware nonidealities:* For custom analog circuit based SNN implementation, it is necessary to perform spice simulation in order to verify the expected functionality and effect of hardware non idealities. Simulation of SNNs is time consuming. Simulating an SNN in a spice simulator is even more time consuming. Even a small-sized SNN (e.g. two layer fully connected network with 100 and 10 neurons) takes 8 hours of simulation time in Cadence spectre. As a result, it makes more sense to simulate the network in an SNN simulator, adjust the network parameters and then do the final spice simulation. However, existing SNN simulators cannot take into account hardware non idealities. Analog circuits are subject to noise and device mismatch. For custom analog circuit implementation, it is necessary to incorporate device hardware nonidealities into the machine learning model so that the model can mimic performance when they are deployed in real hardware. In chapter 6, I propose a method [18] to simulate SNN that can take into account hardware non idealities and provide very close simulation output as the spice based simulator.

1.3 Reprint Permissions of Previously Published Materials

This dissertation contains materials from articles that I published previously. Reprint permission has been obtained from the corresponding publishers for the copyrighted materials. Proper copyright notices have been given in the references as directed by the publishers in entries [15, 16, 17, 18].

Chapter 2

Fundamentals of Neuromorphic Engineering

The concept of brain-inspired machines has existed since the beginning of computer engineering. Both von Neumann [19] and Turing [20] discussed machines and the brain in the 1950's. However, Dr. Carver Mead was the first scientist who recognized the similarity between the silicon electronic circuits and the biological nervous system [21]. He coined the term neuromorphic computing in 1990. The physics of the operation of a biological neuron makes use of the exponential function of the Boltzmann distribution. The Boltzmann distribution is also utilized in the operation of a silicon transistor. The nervous system operates under various constraints, such as limited energy, the presence of noise etc. Silicon electronic systems also operate under such constraints. Dr. Mead argued that it should be possible to emulate the architecture of nervous system and computational principles in silicon electronic circuits and achieve robust information processing power similar to the biological nervous system.

If we compare the processing power of a biological nervous system with digital computing systems, we see that biological systems are more efficient by many orders of magnitude. It is estimated that a human brain performs synaptic computations on the order of 3.6×10^{15} operations per second [3] while consuming only $12W$ of power. This is such an extreme computational efficiency that no supercomputer will be able to match. Below I provide an overview of biological computing components and their neuromorphic models.

2.1 Biological Spiking Neuron

A typical neuron cell is shown in Fig. 2.1. The cell is functionally divided into three sections. The dendrites, the cell body and the axon. The axon acts as the output signal branch of the neuron. The dendrites act as the input signal branch of the neuron where axons from other neurons connect. The structure of the dendrites looks like tree branches with leaf-like structures called spines. The overall structure of the neuron resembles the structure of a tree with branches, roots and trunk. When a neuron wants to talk to other neuron it forms a connection between axon of one neuron to the dendrite of other neuron. The connection between an axon and a dendrite is called a synapse. The synapse mostly forms between an axonal branch and the dendritic spine. Sometimes a synapse can form between an axon and the cell body as well.

2.1.1 Neuron Operation

Neurons are essentially electrical devices. When communicating to other neurons, the neuron sends a voltage spike called an action potential as an output down the axon. The membrane potential of a neuron is always stated with respect to the outside. At steady state the inside of the cell is more negative than the outside. Typically, the membrane potential inside the cell is $-70mV$ with respect to the outside at steady state. This is called the resting potential. When the neuron receives an input action potential at the dendrite, the membrane potential can either become more negative (polarize) or more positive (depolarize) than the resting potential. If the membrane potential becomes depolarized, the input

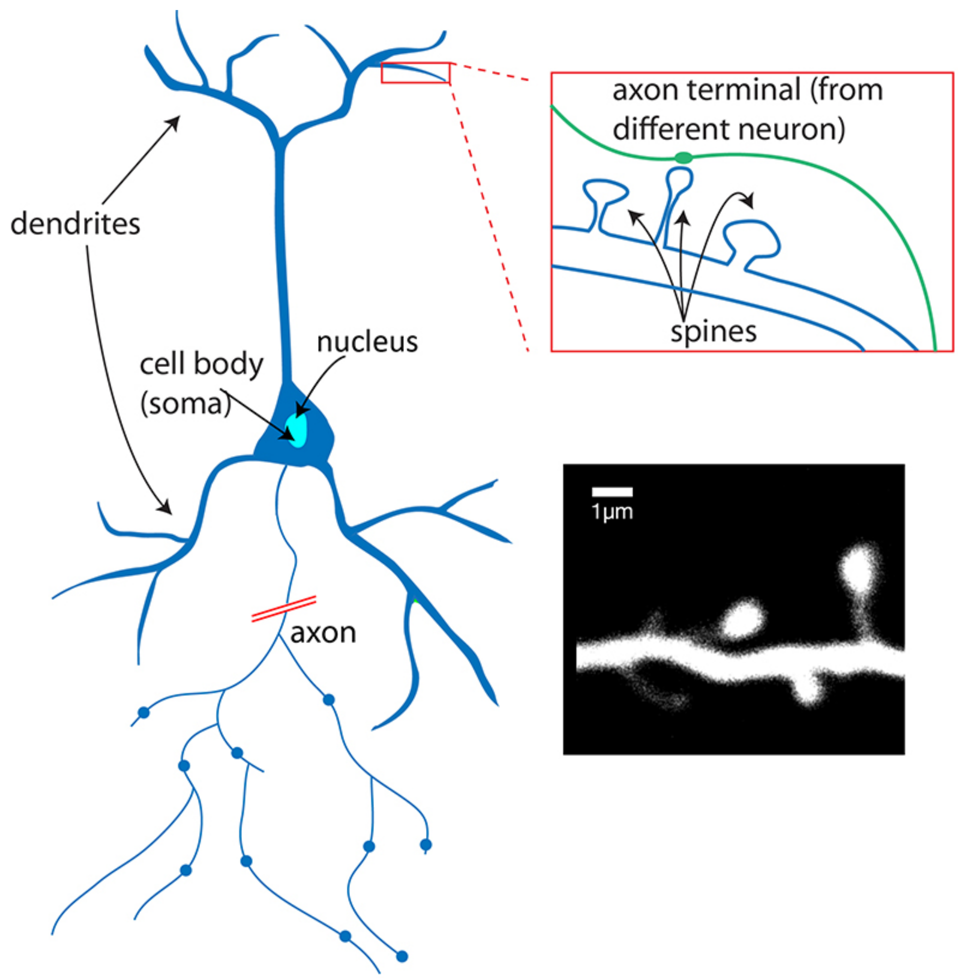


Figure 2.1: General structure of a biological neuron. Bottom-right image: microscopic image of a dendrite from which spines branch off. (Image courtesy: Queensland Brain Institute, Alan Woodruff; De Roo et. al. [1] / CC BY-SA 3.0 via Commons)

action potential is said to be excitatory. Likewise, if the membrane potential becomes polarized, the input action potential is said to be inhibitory. How much the membrane potential will polarize or depolarize depends on the strength of the synapse. As the input action potential comes in, the membrane potential changes. When the membrane potential reaches a threshold voltage, typically around $-50mV$, the membrane potential abruptly increases to a value around

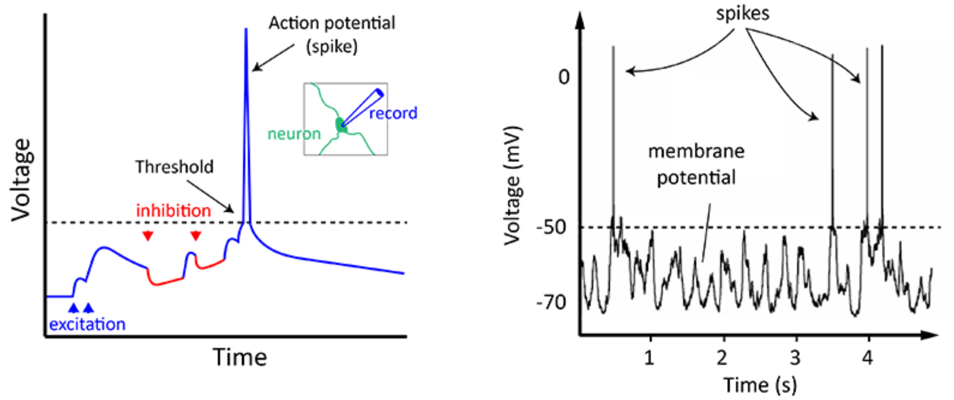


Figure 2.2: Generation of action potential in a biological neuron. Left: qualitative depiction of action potential generation. Right: membrane voltage trace recorded from an actual neuron in a mouse's cortex. (Image courtesy: Queensland Brain Institute)

$+20mV$ and then immediately falls down below threshold as shown in Fig. 2.2. This pulse of membrane potential is called an action potential which travels down the axon. The action potential is also simply referred to as *spike*. The spike typically has an amplitude of $100mV$ and a duration of $1ms$. A chain of action potentials from a neuron is called spike train. Spikes are the fundamental units of communication between neurons.

The description above presents a qualitative description of how neurons work. In reality, the operation of neuron involves complex interaction of charge-carrying ions (Na^+, K^+, Cl^-, Ca^+) and neurotransmitters (dopamin, glutamate, acetylcholin etc.). Fig. 2.3 shows a typical structure of a synapse. The neuron sending the signal is called a presynaptic neuron, and the neuron receiving the signal is called a postsynaptic neuron. When the spike reaches the presynaptic terminal, it causes voltage-gated ion channels to open, releasing the neurotransmitter in the synaptic cleft. The transmitters then bind to the receptors on the dendrite of the postsynaptic neuron. Depending on the type of neurotransmitter, the receptors cause positive or negative ion currents to flow

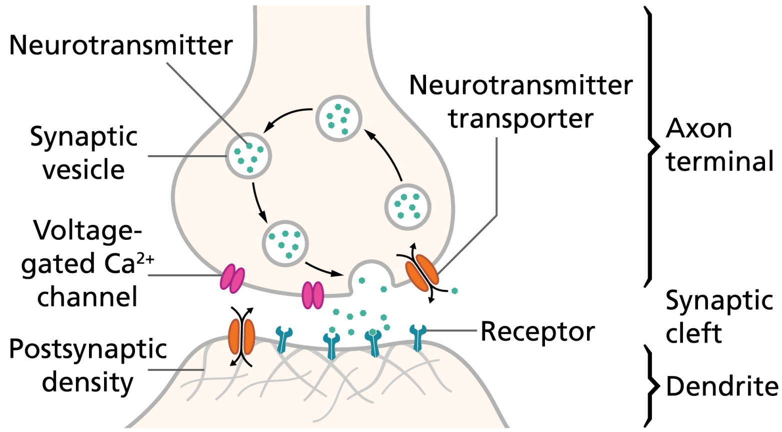


Figure 2.3: Spike, causes neurotransmitters to be released across the synaptic cleft, causing an electrical signal in the postsynaptic neuron. (Image courtesy: Queensland Brain Institute)

across the cell membrane. This ion current is accumulated on the membrane, which causes the membrane potential to increase or decrease.

When the membrane potential reaches the spiking threshold, a rush of Na^+ influx current causes a rapid increase of the membrane potential as shown in Fig. 2.4. Then immediately Na^+ influx current stops and $K+$ current flows out of the cell, thereby repolarizing the cell. This rapid rise and subsequent fall is called a spike. When a neuron generates a spike, it cannot be stopped by any inhibitory inputs. If the input current is insufficient to depolarize the membrane potential to the threshold voltage, no spike will fire. After generating a spike, the membrane potential goes below the resting potential to a voltage called the reset potential. This phase is called hyperpolarization. From the reset potential, the membrane potential reaches the resting potential again. It is very difficult to make the neuron generate another spike in the time period between the hyperpolarization and the resting state. This period is called the refractory period of the neuron.

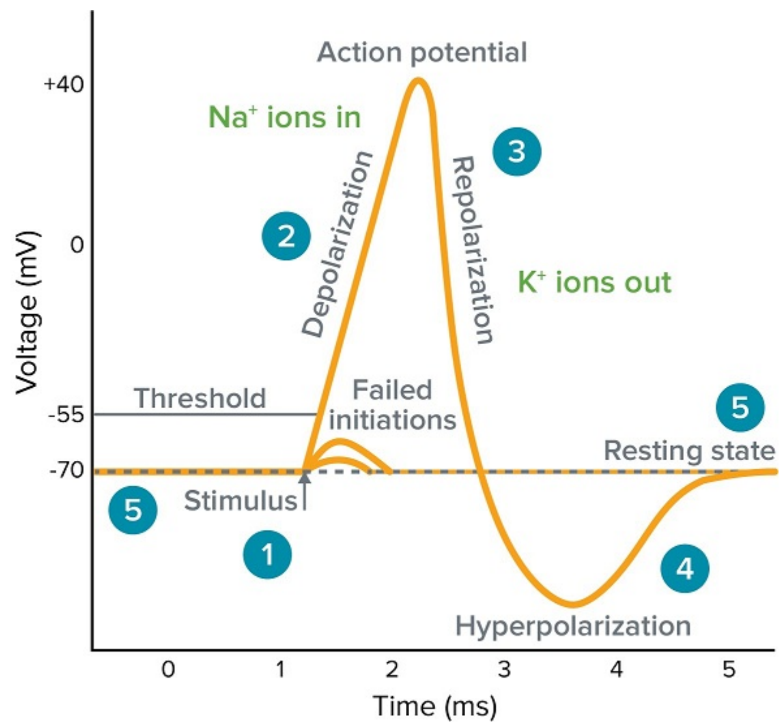


Figure 2.4: Shape of an action potential. (Image courtesy: moleculardevices)

2.2 Spiking Neuron Models

The first mathematical model for a neuron was provided by Hodgekin and Huxley [22] in 1952 which eventually led to the Nobel Prize in 1963. The neuron model accounted for the detailed dynamics of ion channels. This is very useful from the neuroscientific point of view but, at the same time, computationally expensive, which does not provide any insight into the computational power of a neuron. As a result, a simplified model of the neuron is needed, which captures the behavior of the neuron without the detailed dynamics of the ion channels. For engineering and computational purposes, the neural dynamics can be conceived as an input current charging a capacitor combined with a mechanism that triggers action potential above a critical voltage. Below, I describe two dominant classes of neuron models in the literature.

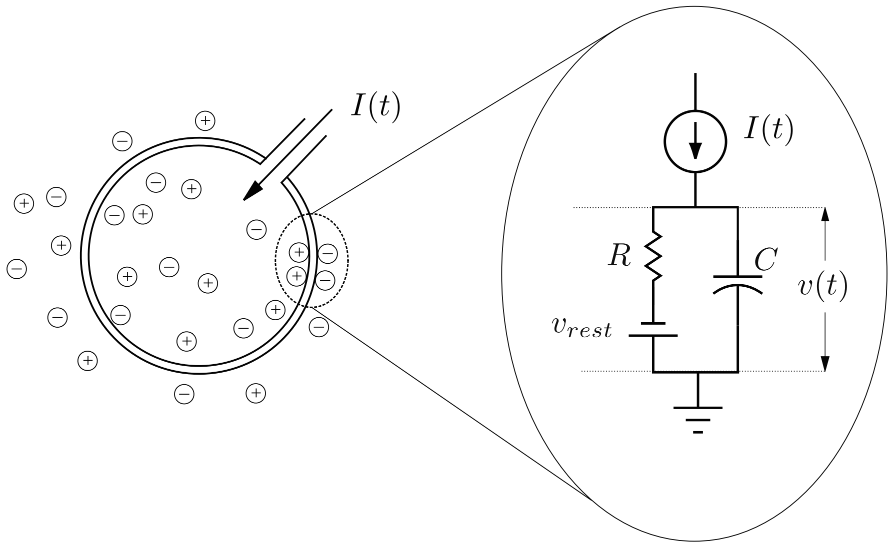


Figure 2.5: Modeling the neuron cell membrane by electrical circuit.

2.2.1 One dimensional Leaky Integrate and Fire Model

Neuron models where action potentials are described as events are called *Integrate-and-Fire* models. The shape of an action potential is not important because information is contained in the presence or absence of a spike. Integrate-and-fire models have two separate components that are both necessary to define their dynamics: first, an equation that describes the evolution of the membrane potential; and second, a mechanism to generate spikes. Fig. 2.5 shows the electrical equivalent circuit of the neuron cell membrane. The cell membrane acts like a capacitor C which can accumulate charge. The resistor R provides a path to leak the current out of the cell, which accounts for the imperfect insulator of the cell membrane. The voltage source v_{rest} allows the circuit to settle at resting voltage at steady steady state. The membrane potential is represented by $v(t)$. The input current $I(t)$ is split between the resistive current I_R and capacitive current I_C branches. The mathematical description is shown in (2.1).

$$I(t) = I_R + I_C \tag{2.1a}$$

$$I(t) = \frac{v(t) - v_{rest}}{R} + C\frac{dv(t)}{dt} \tag{2.1b}$$

$$\tau_m \frac{v(t)}{dt} = -(v(t) - v_{rest}) + RI(t) \tag{2.1c}$$

Here, $\tau_m = RC$ is the time constant of the differential equation. From the mathematical analysis of the electrical circuit, it can be seen that the neuron membrane potential can essentially be described as a linear differential equation. From the electrical engineering point of view, the model equation is a *leaky integrator*. In addition, a criterion is required to generate the spike. Whenever, the membrane potential $v(t)$ reaches or exceeds a given threshold θ, a spike is generated as a Dirac delta function as output of the neuron. The membrane potential is subsequently reset to a reset potential v_r. Whenever a spike is generated as output, the neuron is said to have *fired* a spike. The firing time is labeled as $t^{(f)}$. The firing mechanism is formally expressed as (2.2).

$$t^{(f)} : v(t) \geq \theta \tag{2.2a}$$

$$\lim_{\delta \to 0; \delta > 0} v(t^{(f)} + \delta) = v_r \tag{2.2b}$$

After a neuron has fired a spike, the dynamics again follows (2.1). When a neuron i fires multiple times, the spikes can be labeled as $t_i^{(f)}$ where $f = 1, 2, \cdots$ denote the label of spikes. The spike trains can be expressed as a sum of Dirac delta functions as in (2.3).

$$S_i(t) = \sum_f \delta(t - t_i^{(f)}) \tag{2.3}$$

2.2.2 Two or More Dimensional Model

There is another form of neuron model which integrates the mechanism of upward stroke of spike generation into the model itself. One example of this type of model is Izhikevich [23] model. In this type of model, the dynamics of the membrane potential is tailored with functions to generate the spike. In addition to the membrane potential, another variable is used called the recovery variable or slow variable in order to balance the disturbance in the membrane potential caused by the spike. The membrane potential is also termed as fast variable. The two-dimensional model consisting of the fast and a slow variable is given by (2.4).

$$\frac{dv}{dt} = \frac{1}{C}\{k(v - v_r)(v - v_t) - u + I\} \tag{2.4a}$$

$$\frac{du}{dt} = a\{b(v - v_r) - u\} \tag{2.4b}$$

$$(v, u) \leftarrow (c, u + d) \qquad \text{if } v \geq v_{peak} \tag{2.4c}$$

Here, C is membrane capacitance, $k, v_r, v_t, a, b, c, d, v_{peak}$ are modeling parameters, I is input current. When the membrane potential reaches a predefined peak potential v_{peak} the time is recorded as firing time $t^{(f)}$ and the dynamics is reset by setting the membrane potential v to a reset potential c and the recovery variable u to $u + d$.

2.3 Information Encoding

The actual mechanism of how information is encoded by a spiking neuron and how computation is performed is still unknown. Experimental evidence points to different forms of encoding mechanisms. In general, the hypothesis of information encoding can be broadly categorized as *neuron based* and *population* *based* encoding. There is support for both kinds of hypothesis. However, the universally accepted method of neural encoding is a subject of debate. There are different coding mechanisms in these broad categories. A discussion of these mechanisms is necessary from a neuromorphic perspective. Depending on the hardware, algorithm, and application, one method of encoding may be preferable over the other.

2.3.1 Neuron Based Encoding

In this encoding mechanism, each neuron is believed to encode a numerical value or a representation in its spike. This idea of a single neuron representing a single piece of information is hypothesized from the notion of *grandmother cell* [24]. The idea is that there is a single neuron that becomes active when a person sees their grandmother. In other words, a single neuron encapsulates the representation of the person's grandmother. This way different neurons in the nervous system represent different ideas or concepts. The strength of the ideas or concepts could be represented by spike rates or spike timings.

Rate Coding

Rate coding hypothesis assumes that the information is represented by the firing rate or the number of spikes over a period of time of a neuron [25]. An example of a rate code based neuron is the motor neuron in the peripheral nervous system. A muscle's contraction is controlled by the number of spikes coming onto the muscle in a short time window. The greater the number of spikes, the greater the contraction. In this regard, the spike rate can be thought of as representing numerical values.

Temporal Coding

In a given time window, a neuron can emit some spikes in quick succession and be silent, whereas another neuron can emit the same number of spikes in that window. In both cases, the spike rate is the same, but the neuron has all of the spikes bunched together near the start of the window. In this case, the spike timing is important. In temporal coding, the latency of the spike firing can encode information. An example of temporal code is the early auditory system, where spike timing is used to localize sound [26].

2.3.2 Population Based Encoding

Both rate codes and temporal codes describes encoding by individual neurons. The information can be encoded by the collective activity of a group of neurons as well. In this case, the representation is distributed across the activity of a population of neurons. An example of this coding is in the touch sensitive receptors on our skin. The more pressure is applied the more number of neurons are activated. This process of engaging more neurons as needed is called recruitment. Another form of population coding is to have individual neurons in the population to represent a part of the input. This way all of the neurons in the population can represent the whole input space. An example of this is the direction sensitive cells in the visual cortex. In a given cluster of neurons each neuron is tuned to respond to a particular direction of movement. This kind of population coding is also known as *sparse coding.*

2.4 Learning in Spiking Models

The proper learning model and algorithm for training spiking neural networks is a major open question in neuromorphic engineering. The learning algorithm varies considerably depending on the network, neuron and synapse types. There is also the issue of whether to implement the training or learning on-chip or off-chip. A more fundamental issue is the lack of efficient training algorithms. Deep learning has enjoyed the use of backpropagation [11] in training neural networks. It has largely been successful in training different kinds of networks, such as feedforward and recurrent networks. Backpropagation uses gradient descent in the cost function landscape to reach a minimum error. There are many established and optimized tools available today that implement backpropagation efficiently. Naturally, it makes sense to utilize these existing tools to train spiking neural networks as well. However, backpropagation has not been equally successful in the spiking neural network domain for several reasons. First, backpropagation requires a continuous or piece-wise continuous differentiable function in order to create a smooth cost function landscape for gradient descent to work. Spiking neuron activation function is fundamentally discontinuous and thus non-differentiable in nature. As a result, backpropagation is not directly applied in the spiking domain. Second, backpropagation is not biologically plausible. There seems to be no evidence of a backpropagation-like mechanism happening in the brain. Learning in the brain is based on local synaptic activities. However, learning in backpropagation is non-local, meaning it needs synaptic activity from all the neurons in a layer in order to adjust the synaptic weight. Backpropagation also suffers from a weight transport problem, which means that the backward network needs access to the forward weight in order to calculate the gradients. Although, research has shown that techniques such as feedback alignment [27] have the potential to make backpropagation work using random backward weights, it does not achieve competitive performance for large networks. Despite problems with backpropagation, it is still the best tool available for supervised training for spiking networks with some relaxation in the spiking activation function. Below I briefly describe the current methods available for supervised and unsupervised training algorithms for spiking neural networks.

Supervised Learning

Backpropagation is the dominant method of supervised training in spiking neural networks. There are two ways backpropagation is applied to spiking neural networks. The first method is *weight transfer* method. In this method, first a traditional artificial neural network is trained using backpropagation. Then the artificial neural network is converted to a spiking neural network by replacing the traditional artificial neurons with spiking neurons. This type of conversion does not achieve comparable classification accuracy as the original network. Some weight optimization is required in order to bridge the accuracy gap by balancing weights and thresholds [28]. However, it still fails to reach comparable accuracy.

The second method is to directly apply backpropagation with some relaxation in the spiking activation function. Since, the spiking activation is non-differentiable, a differentiable approximation is used for training. After training is complete, the actual non-differentiable activation is used for inference. This technique is known as surrogate gradient [29, 30]. This technique also fails to achieve comparable accuracy compared to the equivalent artificial neural network. It requires a long inference time window to accumulate enough spikes for decision making. Time-varying parameters such as batch normalization through time [31] can be utilized to decrease the inference time window and accuracy gap.

Unsupervised Learning

Unlike its supervised counterpart, the spiking neural network enjoys biologically plausible unsupervised learning techniques. One of the earliest methods is known as *Hopfield network*. This type of network can memorize patterns in the network dynamics and can retrieve the pattern back in the presence of noise. This type of network is often used to describe associative memory in the brain. Another type of method is called Spike Timing Dependent Plasticity (STDP) or better known as *Hebbian Rule*. In this rule, synaptic weight is increased if the postsynaptic neuron fires immediately after the firing of presynaptic neurons, and synaptic weight is decreased if the opposite happens. In popular terminology, it is known as the neurons that fire together wire together. This type of simple rule is quite powerful in finding underlying patterns and clusters in data [32].

2.5 Discussion

There is still a significant amount of work to be done within the field of learning algorithms and low-power hardware for neuromorphic systems. In order to fully realize the benefits of neuromorphic hardware, a fundamental change in approach and underlying assumptions is necessary for the training method and encoding system. Algorithms such as backpropagation and associated network models were developed with the von Neumann architecture in mind. The spike system is fundamentally different from the von Neumann system. Using surrogate backpropagation with rate coding or temporal coding only tries to imitate the working process of a traditional aritifial neural network. Rate coding encodes numerical values for the input and output of the spiking neuron. A surrogate gradient allows a differentiable activation function for backpropagation to work. None of these methods utilize the underlying spiking hardware and biological training method. As a result, at best, this imitation-based spiking system is only capable of achieving similar performance as the corresponding traditional artificial neural network while achieving increased power efficiency. From an engineering perspective, this power efficiency is very attractive in edge computing and edge machine learning.

Chapter 3

Memory Efficient Sparse Coding

3.1 Introduction

The need for low power and energy efficient intelligent circuit has led electronic circuit designers to draw inspiration from biology [21]. Advancements made by neuroscience have helped shape machine learning techniques such as artificial neural network [33] and reinforcement learning. After the seminal work by Olhausen [34] on sparse coding, several algorithms hav been proposed [35, 36, 14] which inspired a hardware implementation of sparse coding [37, 38]. SAILNet [14] provides an algorithm that have local learning which is biologically plausible. However, in SAILNet the neurons threshold voltage is a learnable parameter. Different neurons in the same layer have different threshold voltages which requires more memory hardware.

In this chapter, I show that SAILNet can be modified to have the same threshold voltage across all the neurons and the feedback matrix can be collapsed into a vector. The resulting network can still reproduce the receptive fields (RFs) of V1 simple cells of visual cortex. The modified algorithm shows more sparsity of neuronal activity and still reconstructs input that image with reliable accuracy. The rest of the paper is organized as follows. First, I present our modification to the algorithm. Second, I show how the modified algorithm sparsity compares with the original one. Third, I show how well the learned receptive fields represents the input stimuli by comparing the classification accuracy of reconstructed images in a convolutional neural network. Throughout the paper the modified algorithm is referred to as new network for simplicity.

3.2 Sparse Coding Algorithm

Sparse coding is based on the idea that an image $I(x, y)$ can be represented as a linear superposition [34] of some basis functions $\phi(x, y)$ as in Eq. 3.1.

$$I(x, y) = \sum_i n_i \phi_i(x, y) \tag{3.1}$$

where n_i is the coefficients corresponding to the basis $\phi_i(x, y)$. The basis functions are not necessarily orthogonal to each other. The basis functions are also overcomplete which means that number of basis functions are more than the total number of elements in $I(x, y)$. The goal of sparse coding is to find a set of n_i to represent $I(x, y)$ such that most of the values of n_i are zero. Which means that the image is represented by the activities of a small set of bases from the whole set of basis functions. In matrix form Eq. 3.1 can be expressed as $I = \Phi N$, where $\Phi = [\phi_1\ \phi_2\ \cdots\ \phi_m]$ and $N = [n_1\ n_2\ \cdots\ n_m]^T$. Each column of Φ is the flattened out from $\phi_i(x, y)$.

3.2.1 Network Design

Each basis function is represented by a spiking neuron. The activities of a neuron (number of spikes in a given period) represents the coefficients of the basis that the neuron represents. For sparse activity only a few neurons need to show activity and most of the other neurons need to be inactive. Lateral inhibition is a way to achieve this whereby the most active neuron prevents the other neurons

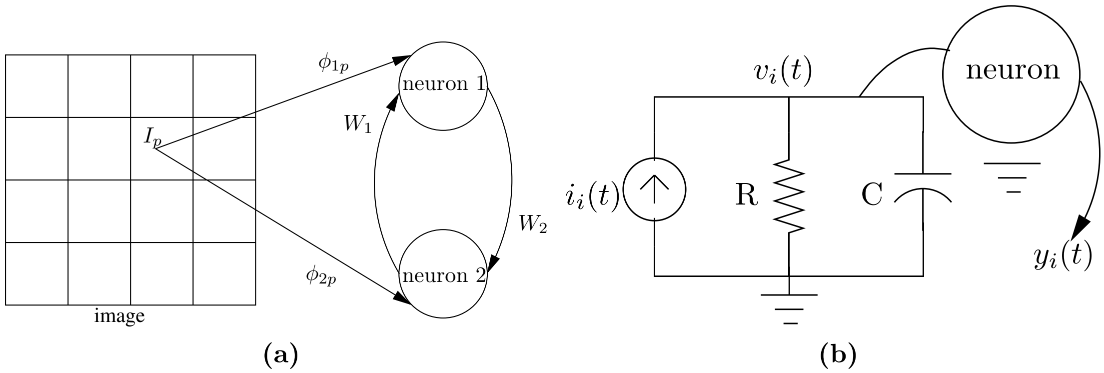


Figure 3.1: (a) Feed-forward, feedback and pixel connection (b) Integrate and fire neuron model.

from activating. Fig. 3.1 shows neuron connectivity. The input current for each neuron comprises of stimuli from pixel intensity values and activities of other neurons. The current in neuron i is as shown in Eq. 3.2.

$$i_i(t) = \frac{1}{R}(\sum_p \phi_{ip} I_p - W_i \sum_{i \neq j} y_j(t)) \tag{3.2}$$

Here I_p is the image intensity value from pixel p, R is the membrane resistance. $y_j(t)$ is neuron j output at time t. I_p and y_j act like voltages. If neuron j spikes at time t then $y_j(t) = 1$, else $y_j(t) = 0$. W_i is the lateral inhibitory connection strength between neuron i and other neurons. The $i \neq j$ means the neuron does not inhibit itself. Unlike SAILNet or LCA where each neuron has $M - 1$ inhibitory connections, here each neuron has one inhibitory connection that treats all incoming spikes from other neurons by same strength. The current changes the membrane potential v_i of neuron i according to the leaky integrate model given by the differential Eq. 3.3.

$$\tau \frac{dv_i(t)}{dt} = -v_i(t) + i_i(t)R \tag{3.3}$$

Here $\tau = RC$ is the time constant, R is the membrane resistance, C is the membrane capacitance. When v_i reaches a certain threshold θ, the neuron emits an action potential or spike. The output of each neuron is taken as number of spikes generated by the neuron, $n_i = \sum_t y_i(t)$, inside a fixed period of time following the stimulus presentation to the network. For simulation this period of time is taken as 5τ similar to [14]. For simulations in this chapter the network is taken as two times overcomplete. Input image size is chosen as $16 \times 16 = 256$ pixels. Hence the number of neurons for two times overcomplete is $2 \times 256 = 512$.

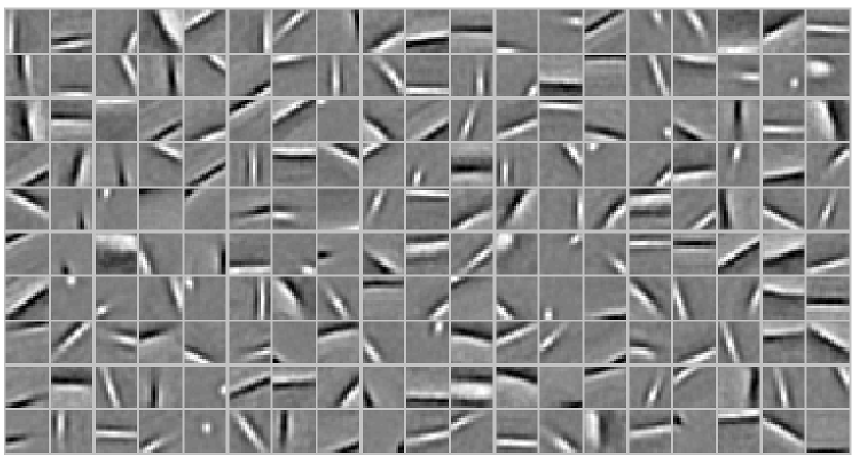


Figure 3.2: 190 randomly selected RFs out of 512 RFs learned using the rules in Eq. 3.6. Each of the RF is 16×16 size. Simulation settings: $\tau = 1$ unit, $\theta = 2$. Learning rate used in learning these RFs: $\alpha = 0.1$, $\beta = 0.01$.

3.2.2 Learning Rules

The learning rules are formed from the constrained optimization imposed on the network. First of all, the network activity must be able to reconstruct the input stimulus. From Eq. 3.1 the reconstructed pixel value is $\bar{I}_p = \sum_i n_i \phi_{ip}$. The mean squared error between the input and the reconstruction, $\sum_p (I_p - \sum_i n_i \phi_{ip})^2$, should be minimized. Secondly, the network activity has to be sparse i.e. only few neurons should produce spikes. If neuron i is active then other neurons should be ideally inactive if the input stimulus can be represented by only the activity of neuron i. Hence, the product $n_i \sum_{i \neq j} n_j$ should be zero or close to zero. This also helps to ensure that the activity minimizes L_0 norm. Using the Lagrange multiplier I can form the Lagrange function.

$$\mathcal{L} = \frac{1}{2} \sum_p (I_p - \sum_i n_i \phi_{ip})^2 - \sum_i W_i (n_i \sum_{i \neq j} n_j) \tag{3.4}$$

Here the inhibitory connection strength W_i for neuron i serves as the Lagrange multiplier. To minimize $\mathcal{L}$ I perform gradient descent with respect to ϕ_{ip} and W_i.

$$\Delta W_i = -\alpha \frac{\partial \mathcal{L}}{\partial W_i} = \alpha n_i \sum_{i \neq j} n_j \tag{3.5a}$$

$$\begin{aligned} \Delta \phi_{ip} &= -\beta \frac{\partial \mathcal{L}}{\partial \phi_{ip}} = \beta n_i (I_p - \sum_j n_j \phi_{jp}) \\ &= \beta (n_i I_p - n_i^2 \phi_{ip} - n_i \sum_{i \neq j} n_j \phi_{jp}) \end{aligned} \tag{3.5b}$$

Here α and β are learning rates. Learning rule from Eq. 3.5b is non local i.e. neuron i needs to know neuron activities from neuron j in the last term. But I notice that if network activity is sparse, only one neuron is active and others are inactive. So on average the $n_i \sum_{i \neq j} n_j$ product should be zero. Hence, I can ignore the last term of Eq. 3.5b and thus the rule becomes local. The rule from Eq. 3.5a is local because W_i connects neuron i to other neurons and it needs activities n_i and $\sum_{i \neq j} n_j$ which is local to W_i. The final learning rule as average of batch process can summarized as follows.

$$\begin{aligned}\Delta W_i &= \alpha \langle n_i \sum_{i \neq j} n_j \rangle \\ \Delta \phi_{ip} &= \beta \langle (n_i I_p - n_i^2 \phi_{ip}) \rangle \\ &= \beta (\langle n_i I_p \rangle - \langle n_i^2 \rangle \phi_{ip})\end{aligned} \tag{3.6}$$

The ϕ learning rule looks similar to SAILNet learning rule. But the assumptions made to arrive at these rules are different from those imposed in SAILNet. The threshold voltage is fixed for all the neurons here. For SAILNet the threshold voltage is also a learnable parameter.

3.2.3 Learned RFs

Training images to learn the basis functions/RFs are taken from natural image set of Olshausen and Field [34]. There are ten 512×512 images of natural scenes available preprocessed by zero-phase lowpass filter described in [34]. W is set to zero and Φ is set to random values before training as in [14]. Threshold voltage θ is set to a value of 2. Batches of 100 images each of size 16×16 with zero mean and unit standard deviation are selected randomly from the images in the database and presented to the network. Number of spikes generated from the neurons are counted over 5τ unit of time after the images are presented. With those spike counts W and Φ are updated using the rules of Eq. 3.6. This process is repeated until a stable solution is reached. Fig. 3.2 shows some of the RFs obtained after training. The RFs are spatially localized, oriented and selective to structures like edges. These are the properties of RFs of mammalian primary visual cortex and looks similar to RFs recorded from V1 simple cells of macaque monkey [39].

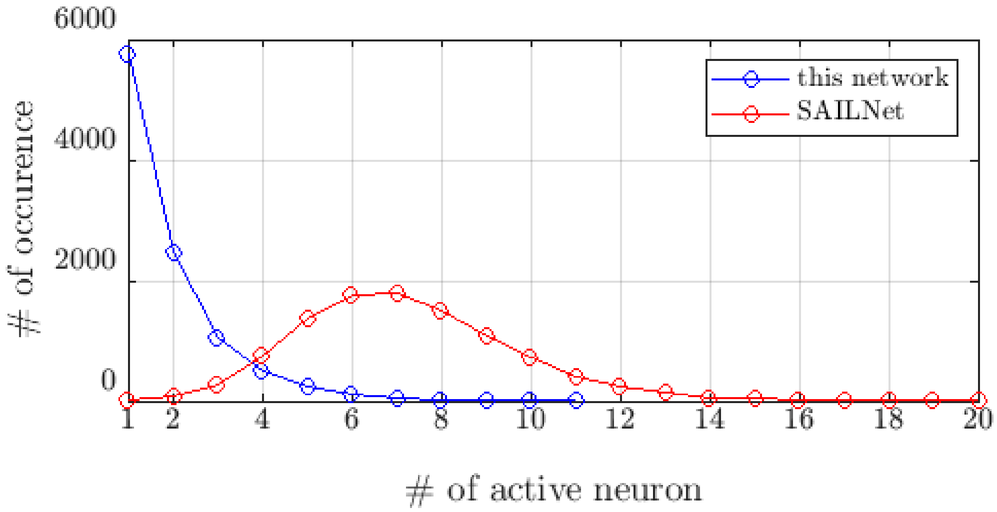
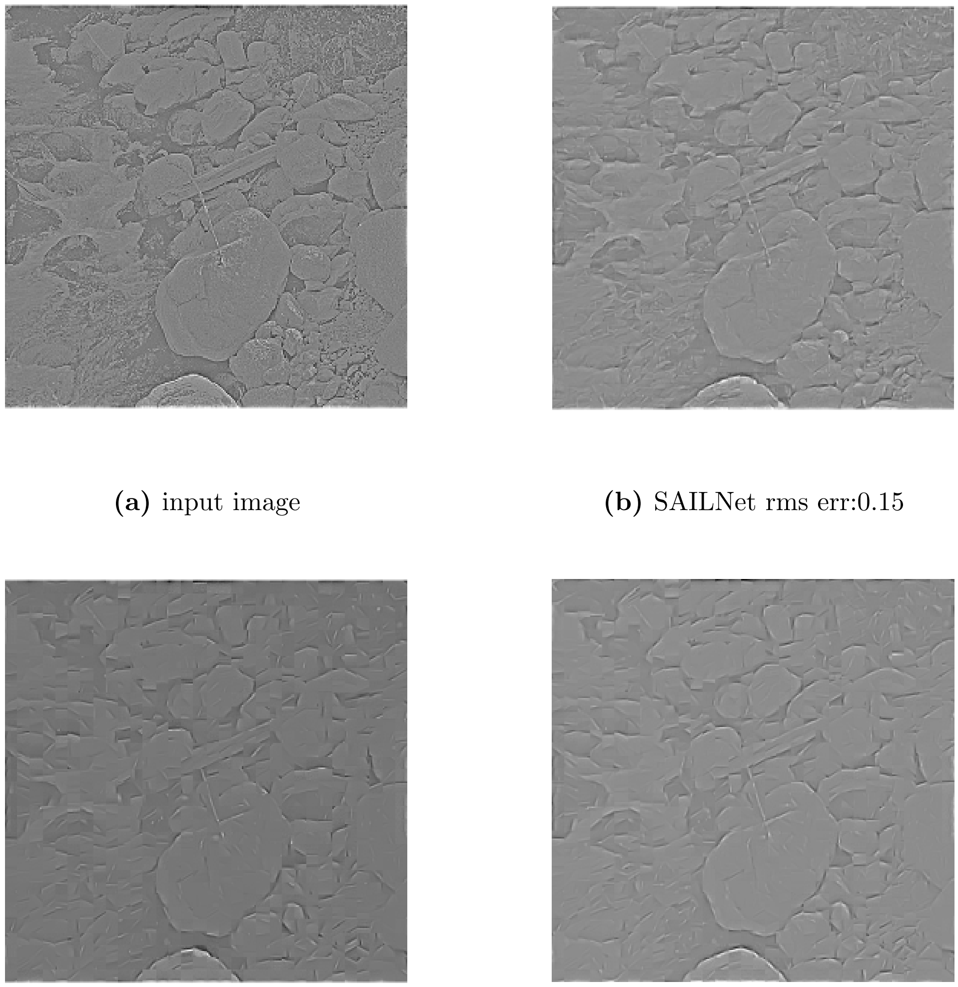


Figure 3.3: Sparsity histogram: sparsity is indicated by a 16×16 image patch being represented by small number of active neurons most of the time an image patch is presented to the network.

3.3 Sparsity of Activities

Here I compare the sparsity of the learned network with the sparsity of SAILNet. SAILNet was learned using the parameters provided in [14]. All ten images from the database are fed to both of the networks. 16×16 image patches are taken from the database images and number of spikes are counted in a 5τ unit time window. Each image is 512×512, hence with 16×16 image patches there are 1024 patches from one image and 10240 patches from all ten images. If I count the number of neurons with non-zero spike counts after each image patch presentation and plot them in a histogram I get a comparison of sparsity. Fig. 3.3 shows the result. The new network learning rules produced only one active neuron most of the time a 16×16 image patch is presented. Out of 10240 image patches around 4800 patches, almost 47% of the time, a 16×16 image patch is represented by only one neuron activity. Compared to that SAILNet produces seven active neurons most of the time a 16×16 image patch is presented. The new network is clearly



(a) input image

(b) SAILNet rms err:0.15

(c) this net rms err:0.17

(d) this net(0.1×W) rms err: 0.16

Figure 3.4: Reconstruction error comparison.

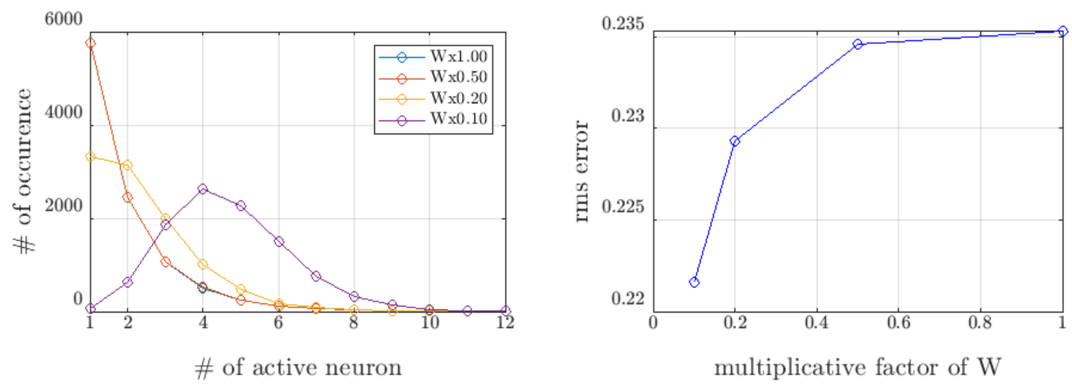
more sparse than SAILNet. This is because of the learning rule of Eq. 3.5a which encourages one neuron to be active.

3.4 Reconstruction Accuracy

There is a trade off between sparsity of activity and accurate reconstruction. For accurate reconstruction of image from linear combination of basis functions, more than one basis functions are needed to reproduce fine details of input image. Since the new network has only one active neuron most of the time, the reconstruction error is slightly higher than SAILNet. Fig. 3.4 shows a reconstructed image along with rms errors for SAILNet and new network. The rms error is just slightly higher than that of SAILNet. This is expected because in new network sparsity is higher. If more accurate reconstruction is required, it can be achieved to some degree by tuning the value of W. If I reduce the inhibitory connection strengths, neurons will not have reduction of the membrane potential as much and more neurons will fire. Thus the network activity will get less sparse i.e. more than one neuron activity will represent the input stimulus most of the time. Fig. 3.4d shows such a reconstruction with inhibitory weights set to ten percent of learned inhibitory weights where network is less sparse and more details are visible. I am trading off activity sparsity for more accurate reconstruction. Fig. 3.5a shows how multiplying W with a factor less than one, changes sparsity for images in the dataset. As the inhibitory connection gets less stronger more neurons are active most of the time and the curve begins to look like SAILNet sparsity curve as in Fig. 3.3. As the sparsity is reduced by reducing values of W, the rms error also decreases as shown in Fig. 3.5b. These two figures clearly shows the trade off between sparsity and reconstruction accuracy.

3.5 Quality of RFs and Reconstruction

Although reconstruction error is slightly higher for the new network, to a human eye reconstructed images from SAILNet and new network looks similar as in



(a) sparsity **(b)** rms error

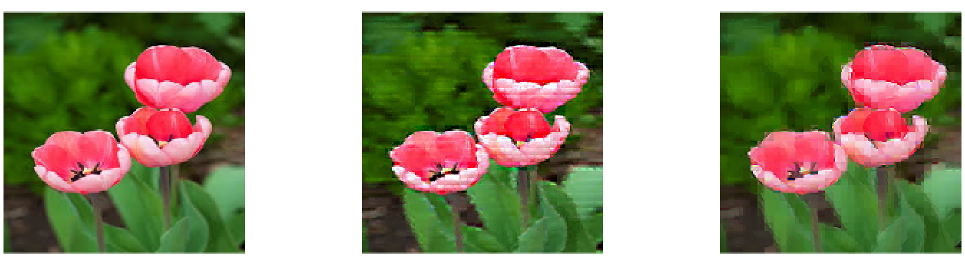
Figure 3.5: Sparsity and reconstruction rms error tuning by tuning the values of inhibitory connection strength.

Fig. 3.4. But would a computer vision program be able to tell if Fig. 3.4(b) and Fig. 3.4(c) are same and they are similar to Fig. 3.4(a)? To answer that question I devise an experiment. I feed reconstructed images to an image classifier and compare the classification error with the classification error of original images. If they are close then I can say that the reconstructed images have enough information for a computer be able to tell the difference. For this I train a convolutional neural network with flower dataset [40] which has 17 classes of flowers of each class with 80 images. This dataset is chosen because it is lightweight and has natural scene. Every image is resized to 512×512 pixels. For convolutional neural network I choose ResNet-101 [33]. For training 80% and for testing 20% of the images from each class is used. Three testing image sets are made: first set with the original testing images, second set with the reconstructed images of the first set using SAILNet, third set with the reconstructed images of the first set using new algorithm. For reconstruction of the flowers, RFs learned in section 3.2.3 are used instead of learning them again on the flower database. The reason is that since those RFs are learned on natural images, they should be able to reproduce any other natural scenes. The flower images are color images but

Table 3.1: Classification Accuracy

testing set	accuracy
original	91.3%
SAILNet reconstuction	86.2%
new network reconstuction	89.1%

the RFs are grayscale images. Hence, the reconstruction is done on R,G,B color channel separately. A reconstruction is shown in Fig. 3.6. Interestingly, the rms error for flower dataset turned out to be less than SAILNet while maintaining more sparsity. Using the three sets of testing images classification accuracy is measured. The result is shown in table 3.1. The classification accuracies for the reconstructed images are not too far from accuracy of original images. This proves that RFs can faithfully retain information for a convolutional neural network to be able classify. The classification accuracy of the new network turned out to be higher than SAILNet. I think this is because the details discarded by the new sparse coding network was helpful for the convolutional neural network for this dataset.



(a) original **(b)** SAILNet rms:0.08 **(c)** this network rms:0.06

Figure 3.6: Reconstruction images and rms error for one of the images from flower datadset. W matrix is unaltered for reconstruction.

3.6 Computation and Hardware Complexity

Since a neuron does not inhibit itself, in SAILNet or LCA each neuron needs $M-1$ feedback weights. For M neurons the feedback needs an $M\times(M-1)$ vector matrix multiplier. With N elements in each input, feed-forward computation needs an $N\times M$ vector matrix multiplier. In SAILNet there are also M threshold voltages. So total memory needed for SAILNet is $NM + M(M-1) + M = NM + M^2$. But in the modified algorithm each neuron has one feedback weight and it does not have different threshold for each neuron. Hence the memory requirement is $NM+M$. This is a huge savings in memory and associated circuits for hardware implementation. The vector matrix multiplication of $M \times (M-1)$ elements is reduced to M multiplication which can save power as well. In [38] SAILNet was implemented in 65nm digital process. It takes significant fraction of the total power for data movement from memory. In [37] LCA was implemented using analog floating gate memory. It takes considerable amount of time to fix the floating gate voltages to appropriate values. Reducing number of feedback weights and removing neuron threshold as stored memory parameter can help both digital and analog implementation of sparse coding to reduce computation and speed up operation.

3.7 Conclusion

In this chapter, I present a modification of the sparse coding algorithm, SAILNet, that reduces the number of learnable parameters without significantly affecting the reconstruction error and still reproduce the RFs of V1. Our experiments show that the modified algorithm is more sparse but can reproduce the input signal with necessary information for it to be identified by a convolutional neural network. Although there is a trade off between sparsity and rms error, this reduced memory algorithm can be useful for processes which can tolerate inaccuracies in data to a certain level.

Chapter 4

Compact Ultra Low Power Spiking Neuron Circuit

4.1 Introduction

Brain inspired neuromorphic systems use biologically plausible spiking neurons to model intelligent systems like silicon retina, cochlea and machine learning systems [41, 42, 43]. Simulation of large scale spiking neural networks in a traditional von-Neumann type digital system is not suitable because of the asynchronous nature of spiking neurons. Highly parallel nature of neuromorphic hardware makes them faster, which has led to their recently increasing popularity [44]. However, very large scale simulations of neural networks in hardware become power hungry. Hence, efforts went into designing biologically plausible spiking neuron circuits [45] with behaviors, such as adaptation and bursting while restricting power consumption of individual neurons.

While many designs implement a broad range of spiking behaviours [46], the circuits operate in strong inversion and consume high power. Other designs use subthreshold circuits [43], but they require many transistors. In this chapter I propose a leaky integrate and fire neuron that uses subthreshold device physics to implement neuron functionality, which allows us to reduce number of transistors. The circuit elements draw current only when the neuron is spiking and not at other times. The power consumption at spike time is very small. The neuron is capable of showing complex behaviour like adaptation and bursting while using only a handful of transistors. I have used a 130nm silicon CMOS process for simulation in cadence spectre.

4.2 Neuron Circuit

The circuit is shown in Fig. 4.1. The circuit consists of five sub blocks. Block a with I_{in} and M_k serves as input excitation to the membrane capacitor C_v. Block b with M_{1-3} performs thresholding and spike generation. Block c with M_{4-5} acts as the axon which generates a voltage pulse at each spike. Block d with M_{6-8} controls spike width, refractory period and resets the neuron after a spike. Block e with M_{9-11} controls adaptation and bursting. The main firing and resetting dynamics are governed by (4.1) and (4.2)

$$C_v \frac{dv}{dt} = I_{in} - I_k + I_{pos} - I_{neg} - I_a \tag{4.1}$$

$$C_u \frac{du}{dt} = I_w - I_r \tag{4.2}$$

The neuron has 12 transistors that operate in the subthreshold regime. The body of all the nFETs are grounded, and the body of all the pFETs are connected to the positive supply. The circuit has multiple levels of control over the neuron operation. It can control spiking threshold, spike width, refractory period and adaptation period.

4.3 Circuit operation

The circuit operation is described below as a step by step process.

Step 1: Input current I_{in} acts as excitatory current to the neuron. The leak transistor M_k subtracts some current I_k from I_{in} using V_k. Hence, the total input

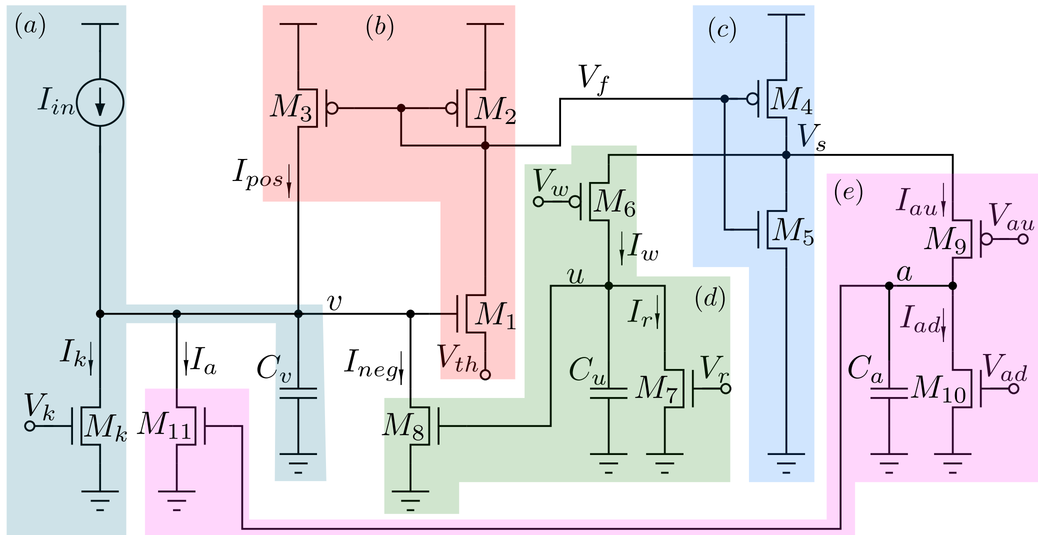


Figure 4.1: Proposed neuron circuit (a) Input block, (b) M_{1-3} for thresholding and spike generation, (c) M_{4-5} for axon, (d) M_{6-8} for reset and spike width, refractory control, (e) M_{9-11} for adaptation and bursting control.

current going into the membrane capacitor C_v is $I_{in} - I_k$. By controlling I_{in} and V_k input current to the neuron can be made excitatory or inhibitory. The net excitatory input current charges up the membrane capacitor C_v, and membrane voltage v increases.

Step 2: Membrane voltage v is applied to the gate of M_1. Once the gate voltage of M_1 increases above the source voltage V_{th} which acts as spiking threshold, M_1 starts to conduct current. This current is copied using M_{2-3} and fed back into membrane capacitor C_v thus implementing positive feedback current I_{pos}. The current through M_1 can become very large if the top rail voltage is large. Here, the top rail voltage is low which limits the maximum current through M_1 and consequently limits the power consumption for a spike. Since M_1 is operating in the subthreshold regime, the current produced is exponentially related to the gate voltage. When v exceeds threshold V_{th}, this exponential positive feedback current raises v very quickly until v reaches the top voltage rail V_{dd}.

Step 3: As long as v is higher than V_{th}, M_1 conducts current and V_f drops below V_{dd}. The axon block is essentially an inverter. Hence, V_s goes up and reaches V_{dd}. The current through an inverter can be very high when both M_4 and M_5 are conducting. But in this case V_{dd} is low, which limits the current. As V_s goes up, capacitor C_u charges through M_6 and increases voltage u. The speed of charging C_u can be controlled via V_w. Once u becomes high enough to produce a current though M_8 such that I_{neg} overpowers I_{pos}, C_v discharges, axon output V_s goes to ground and the neuron resets. Using V_r in M_7, C_u can be discharged slowly so that voltage u can continue to produce high enough I_{neg} that the input current cannot charge C_v. This implements the refractory period. Once the refractory period is over the neuron starts the operation again if there is still any input current. By controlling the charging time of C_u using V_w the spike width can be controlled. Transistors attached to C_v implement the membrane resistance.

Spike frequency adaptation is accomplished by reducing the input current to membrane capacitor. With every spike axon output, V_s reaches V_{dd} which charges capacitor C_a slowly using M_9. The slight increase in voltage a causes M_{11} to conduct current I_a and leak some input current. V_{au} and V_{ad} controls the charging and discharging of C_a. By selecting proper values of the control voltages V_w, V_r, V_{au}, V_{ad}, a wide range of spiking patterns can be achieved. The transistor sizing and capacitor values are given in Table 4.1. Individual transistors are very small in size except M_4, which is slightly larger than the others because it has to supply current to block d and e. The only large size capacitor is C_a, which controls adaptation and bursting.

Table 4.1: Transistor size, capacitor and supply voltage values

M_4 W/L	Other FET W/L	C_v	C_u	C_a	V_{dd}
800nm/260nm	260nm/260nm	50 fF	30 fF	100 fF	300mV

4.4 Spike patterns

A 130nm CMOS process is used for circuit simulation with single supply voltage voltage $V_{dd} = 300mV$. Different spiking patterns are obtained by setting appropriate control voltage values. Fig. 4.2 shows different spiking patterns for a constant input current. The voltage values used to obtain spiking patterns are given in the figure description. For a regular spiking (RS) pattern, adaptation block does not charge capacitor C_a to high voltage, thereby stopping current leakage through M_{11}.

For regular spiking but with frequency adaptation (RS-FA) spiking patterns, C_a is allowed to charge. V_{au} and V_{ad} are set such that after a few spikes, voltage a settles down to a fixed value, and the neuron continues to spike at a slow rate.

Chattering (CH) and intrinsically bursting (IB) spiking patterns for a constant input current are obtained by manipulating the control voltages. A chattering neuron generates a burst of high frequency spikes repetitively in response to a constant input current. The magnitude of input current controls the period between the burst. An intrinsically bursting neuron generates a burst of spikes at the beginning of a constant input current and then switches to tonic spiking mode. Pyramidal neurons found in cortical layers display these kinds of behaviors [47]. Another type of spiking pattern found in cortical layers is the fast spiking (FS) pattern. This kind of pattern is created by periodic trains of spikes with high frequency without adaptation. These are created by not allowing the membrane potential to reach all the way to ground when it resets. All of these firing patterns can be obtained in our circuit by adjusting the control voltages.

Fig. 4.3 shows a spiking pattern when threshold voltage and refractory period is changed. Spiking threshold can be changed by changing V_{th}. Fig. 4.3(a), (b), (c) show that spike frequency is reduced as the spiking threshold is increased from $30mV$ to $70mV$. From Fig. 4.3, it can be noticed that the onset of the

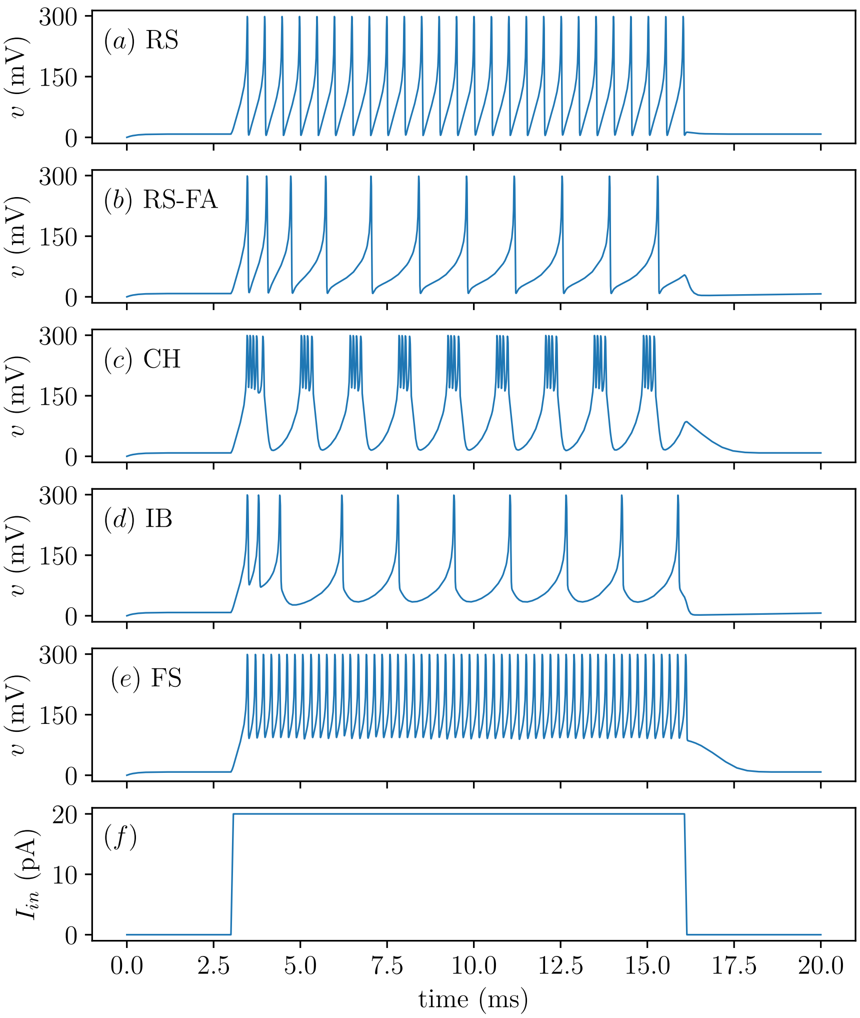


Figure 4.2: Different spiking patterns from the Neuron. For all cases $V_{dd} = 300mV$, $V_{th} = 50mV$, $V_k = 30mV$ (a) RS: $V_w = 80mV$, $V_r = 120mV$, $V_{au} = 280mV$, $V_{ad} = 3mV$, (b) RS-FA: $V_w = 80mV$, $V_r = 120mV$, $V_{au} = 120mV$, $V_{ad} = 3mV$, (c) CH: $V_w = 70mV$, $V_r = 145mV$, $V_{au} = 120mV$, $V_{ad} = 50mV$, (d) IB: $V_w = 80mV$, $V_r = 130mV$, $V_{au} = 120mV$, $V_{ad} = 3mV$, (e) FS: $V_w = 80mV$, $V_r = 135mV$, $V_{au} = 280mV$, $V_{ad} = 3mV$, (f) input current I_{in}

spike is around $150mV$, although V_{th} is below that. This is because gate to source voltage difference needs to be around $150mV$ to generate a strong positive feedback current. So, the onset of the spike is effectively slightly higher than the voltage set by V_{th}. Fig. 4.3(c) and (d) have the same spiking threshold, but the refractory period is increased in (d) by decreasing V_r. As a result the spiking frequency decreases. Fig. 4.3(d) and (e) have same settings but in (e) spike width is increased by increasing V_w. Decreasing spike width will decrease energy per spike. However, in a network of neurons, a synapse might need longer spike width to provide necessary current. Hence, it is necessary to provide varying levels of control over the neuron operation.

4.5 Power Consumption

The transistors in the circuit conduct current only during the time of spike. At other times currents through the transistors are only the leakage currents set by the process technology, which are very very low. Fig. 4.4 shows a close up trace of voltages and some currents of a spike from Fig. 4.3(a). The current traces show that current draw spikes only during the time of membrane voltage spike. The major currents are I_{pos} and I_{neg}. The limit of I_{pos} value is set by the gate voltage of M_1, which is V_{dd} at its maximum. Since V_{dd} is low, the current is also low. I_{neg} is larger than I_{pos} because it has to overpower the positive feedback current to reset the neuron. These currents themselves are very low, in this case under 3.5nA. Since there is current conduction only at the time of spike, the neuron consumes power only during the spike time. By reducing spike width using V_w, additional power savings can be achieved. Energy for each spike is calculated by integrating instantaneous power supplied by V_{dd} over the simulation time and dividing by the number of spikes produced. The resulting energy per spike is found to be 22fJ. For this process the collective leakage current is around 7pA when the neuron is not spiking. This means that the static power consumption is 2.1pW for this process. The neuron consumes 15pW of power when spiking at 1kHz.

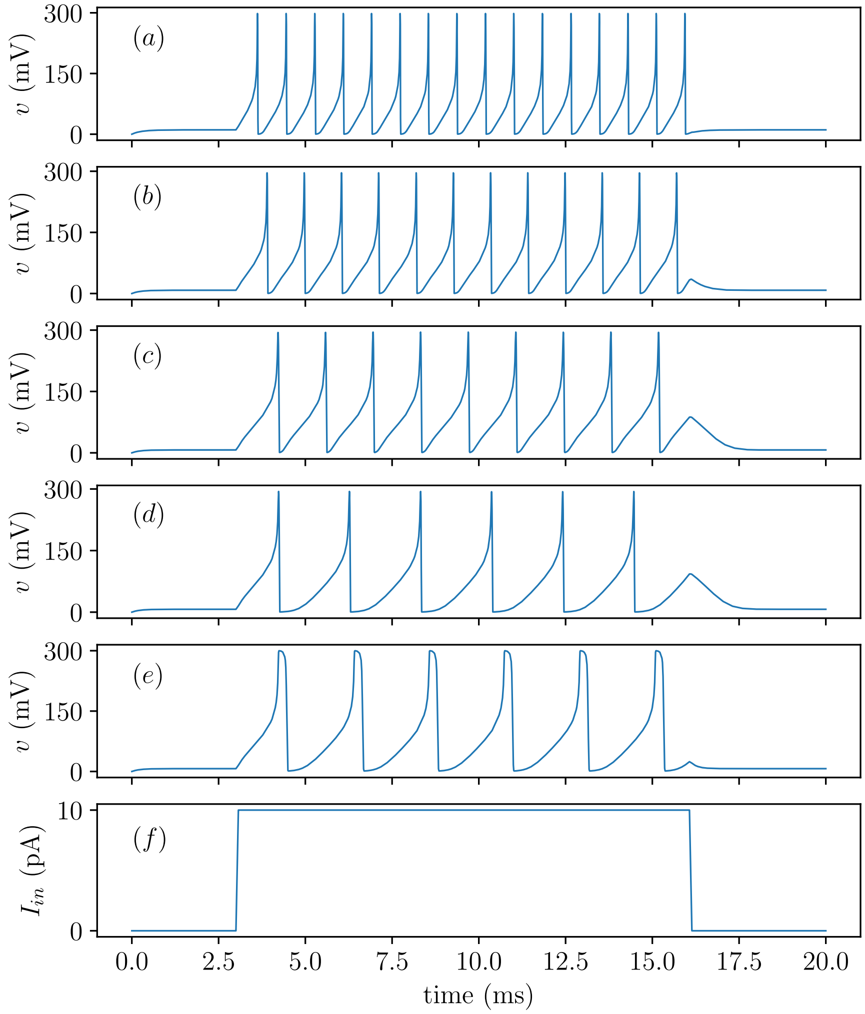


Figure 4.3: Spiking pattern when spiking threshold, refractory period and spike width changes (a) $V_{th} = 30mV$, $V_w = 80mV$, $V_r = 80mV$, (b) $V_{th} = 50mV$, $V_w = 80mV$, $V_r = 80mV$, (c) $V_{th} = 70mV$, $V_w = 80mV$, $V_r = 80mV$, (d) $V_{th} = 70mV$, $V_w = 80mV$, $V_r = 30mV$, (e) $V_{th} = 70mV$, $V_w = 170mV$, $V_r = 30mV$, (f) Input current I_{in}

4.6 Comparison

Circuits as in [48] use an operational amplifier based comparator to implement thresholding. However, a problem with operational amplifier based design is that the tail current of the operational amplifier will consume power even when

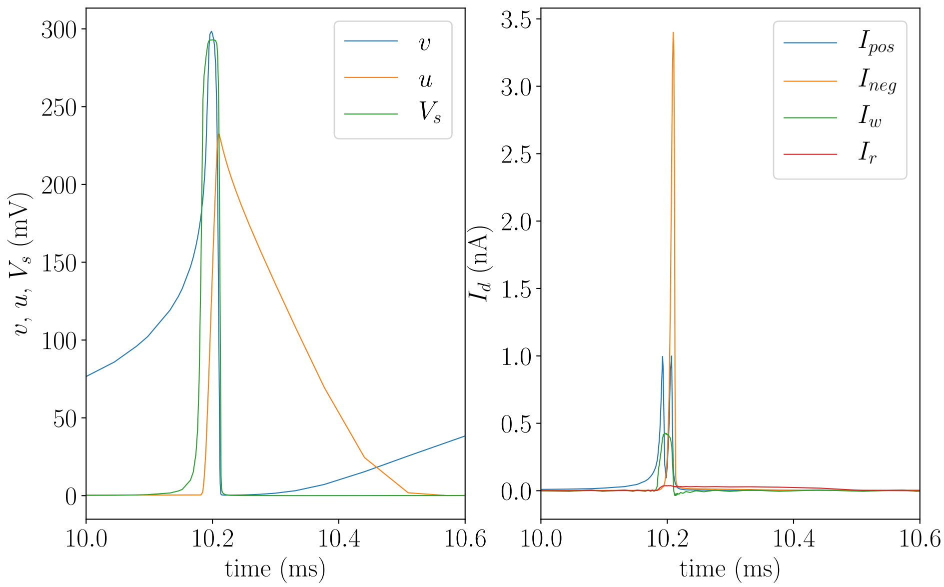


Figure 4.4: Close up view of voltage and drain current spike traces. Current spikes at the time of membrane voltage spike

Table 4.2: Circuit comparison

	Indiveri[43]	Wijekoon[46]	Arthur[45]	our circuit
V_{th} ctrl.	yes	yes	no	yes
refractory ctrl.	yes	no	yes	yes
spike width ctrl.	no	no	no	yes
adapt. & burst.	yes	yes	yes	yes
# of FETs	22	14	15	12
power @frequency	10-110μW @100Hz	8-40μW -	50-100nW @100Hz	15pW @1kHz
Energy/Spike	900pJ	8.5-9pJ	-	22fJ
Area Wμm× Lμm	83×31	70×40	-	15.5×11
Process	0.35μm	0.35μm	0.25μm	130nm
V_{dd}	3.3V	3.3V	-	300mV

there is no excitation current. Hence, for comparison purpose I choose circuits with similar basic working principles and spiking patterns. Table 4.2 compares capability of this circuit with other works. The circuit in [43] is capable of producing complex spike patterns like adaptation but it has no control over spike width and takes a large number of transistors. The circuit in [45] has fewer transistors than [43], but the power consumption for a single neuron is still prohibitively high for integration into a large scale network. The circuit in [46] operates in above threshold mode and consumes a significant amount of power. It should be noted that a 4fJ per spike neuron has been reported in [49]. However, that neuron is much simpler and lacks the variety of spiking patterns observed in biological neurons. The circuit that I propose here has considerable levels of control over the neuron operation, and it is capable of producing a variety of spiking patterns. By using low supply voltage and operating the transistors in subthreshold mode significant power reduction is achieved. Fig. 4.5 shows the layout of the circuit. It occupies $15.5\mu\text{m}\times 11\mu\text{m}$ of silicon area which is significantly less than the other circuits. Most of the area is taken by the capacitors. Ideally, the capacitors and transistors can be made smaller than reported here, but smaller devices are susceptible to process variation and mismatch. The neuron can be integrated into a system in a similar fashion as described in [43].

4.7 Effect of device mismatch

In a network of neurons, all the neurons will be tied to the same global control voltages. However, the process variation and mismatch will cause the devices to conduct a different current than intended. If the mismatch is too large, then neuron output will vary greatly from neuron to neuron. To see how the process variation and device mismatch affects neuron output, a Monte Carlo simulation is performed. For the same settings as Fig. 4.3(a), a few runs from the Monte Carlo simulation are shown in Fig. 6.15. It can be seen that the spiking process of membrane potential v reaching to V_{dd} is not affected. The process of spike generation is robust to mismatch because of the feedback mechanism.

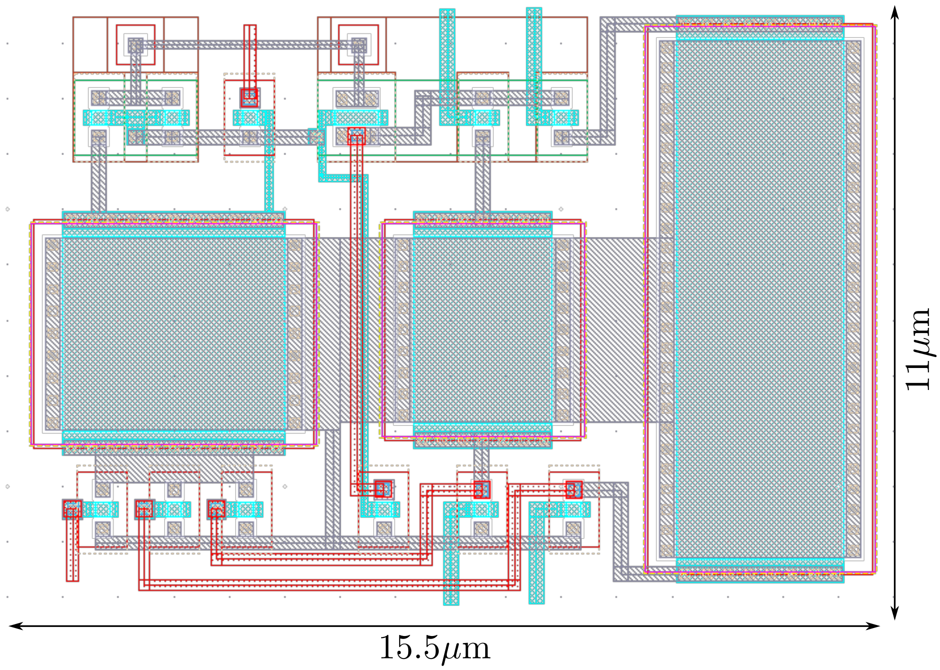


Figure 4.5: Layout of the proposed neuron circuit. Most of the area is taken by the capacitors

However, the spiking frequency is affected. This is not surprising because the device mismatch is causing the leak transistor M_k to conduct a different current than intended. Hence, the current charging C_v is different than expected. For Monte Carlo simulation of Fig. 6.15 the mean firing rate was 1.2kHz with standard deviation of 432Hz. In a chip where synapse weight can be set, this effect of device mismatch can be mitigated by adjusting synapse weight properly.

4.8 Conclusion

In this chapter, I have presented an analog implementation of a spiking neuron operating in the subthreshold regime. Exponential drain current to gate voltage relationship is used to implement positive feedback that generates spike. Device physics is used to implement the operation and reduce number of devices needed.

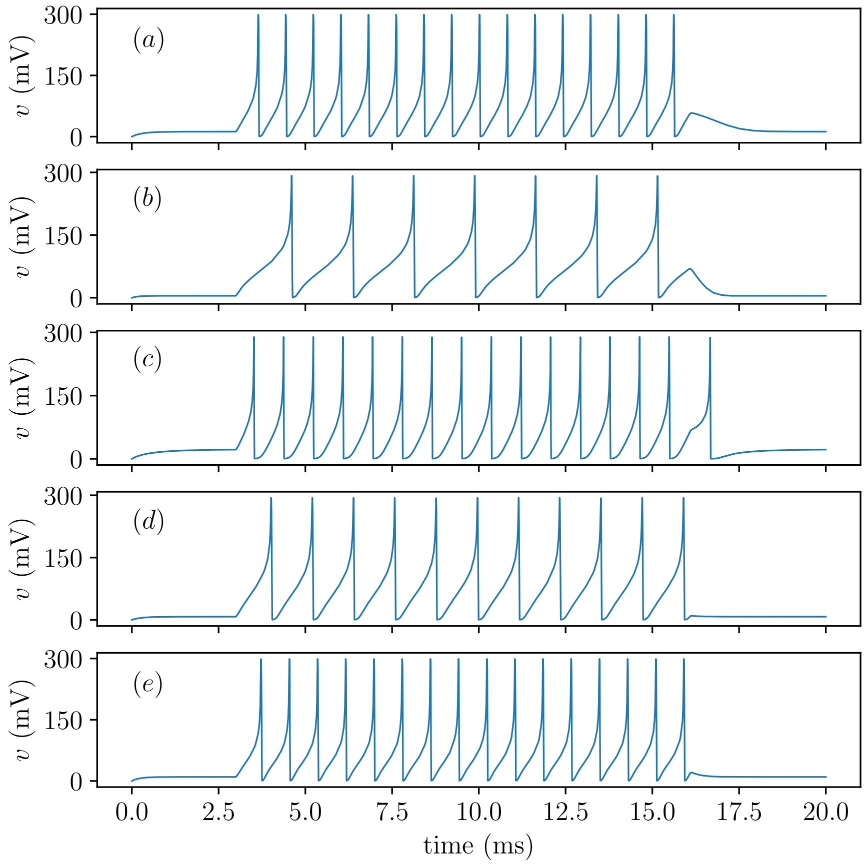


Figure 4.6: Few runs from the Monte Carlo sampling simulation. Due to device mismatch frequency of spike is affected

While being compact, the circuit can show a variety of spiking patterns. The neuron circuit I have developed has the potential to reduce power consumption and area of a large spiking neural network. Since the neuron has negligible power consumption during idle time, any chip made out of this neuron can be operated with stringent power restrictions.

Chapter 5

Synapse Circuit and Leakage Compensation

5.1 Introduction

There has been significant interest in hardware implementations of neuromorphic circuits in recent years. The parallel nature of computation in those implementations makes them ideal to implement spiking neural networks and event driven systems [50]. They are also important to investigate neuromorphic algorithms and hypotheses because of the difficulty of simulating large scale networks in traditional von-Neumann platform. However, manufacturing process of the neuromorphic computational platforms has lagged behind the latest advanced CMOS process available at any given time. Neuromorphic hardware mostly used older CMOS technology nodes. Neuromorphic circuit operations are dominated by differential equations. Subthreshold current mode circuits [51, 45] make it particularly easy to implement ordinary differential equations in transistor circuits. By its nature, subthreshold current is very low, on the level of pico-amperes to nano-amperes. This becomes a problem when implementing subthreshold circuits in smaller technology nodes. As the fabrication process down scales, the leakage currents of the transistors increase. As a result, the leakage currents become comparable to the desired operating currents of the circuit elements. For this reason, even though more advanced technology nodes

Table 5.1: Synapse Packing Size in a Single Chip

	BrainScales [54]	Neurogrid [4]	ROLLS [5]
Synapses per neuron	448	65k(shared)	128k
Process	180nm	180nm	180nm
Year	2010	2014	2015

are available, neuromorphic hardware uses older manufacturing processes as shown in Table 5.1.

To take advantage of smaller technology nodes, one requires either switching to digital circuit [44] or to use alternative circuit design techniques. One technique is to use switched capacitor circuits [52] to circumvent leakage currents. However, leaky switches still present many problems. Also, this technique takes away the flexibility and ease of design of neuron and synapse circuits. Although a subthreshold implementation of a neuron and a synapse circuit in 90nm technology can be found in [53], it is only one neuron circuit and one synapse circuit in two separate chips.

Integration of a large number of synapses from the leakage current point of view is important because in a neural network, synapses are the most abundant circuit elements. Several synapses are typically connected to a single neuron. Hence, the effect of leakage current is most prominent when a large number of synapses are connected together. A synapse injects a certain amount of current into a post synaptic neuron depending on the weight of the synapse when it is hit with a pre-synaptic spike. When the synapse is in inactive or off state, ideally it does not inject any current. However, in a circuit implementation, a synapse conducts leakage current at off state. This leakage current may be ignored when there are only ten or fifteen synapses. However, for a useful neural network hundreds of synapses are necessary. With this many synapses, leakage currents become large enough to stop a neuron from operating properly.

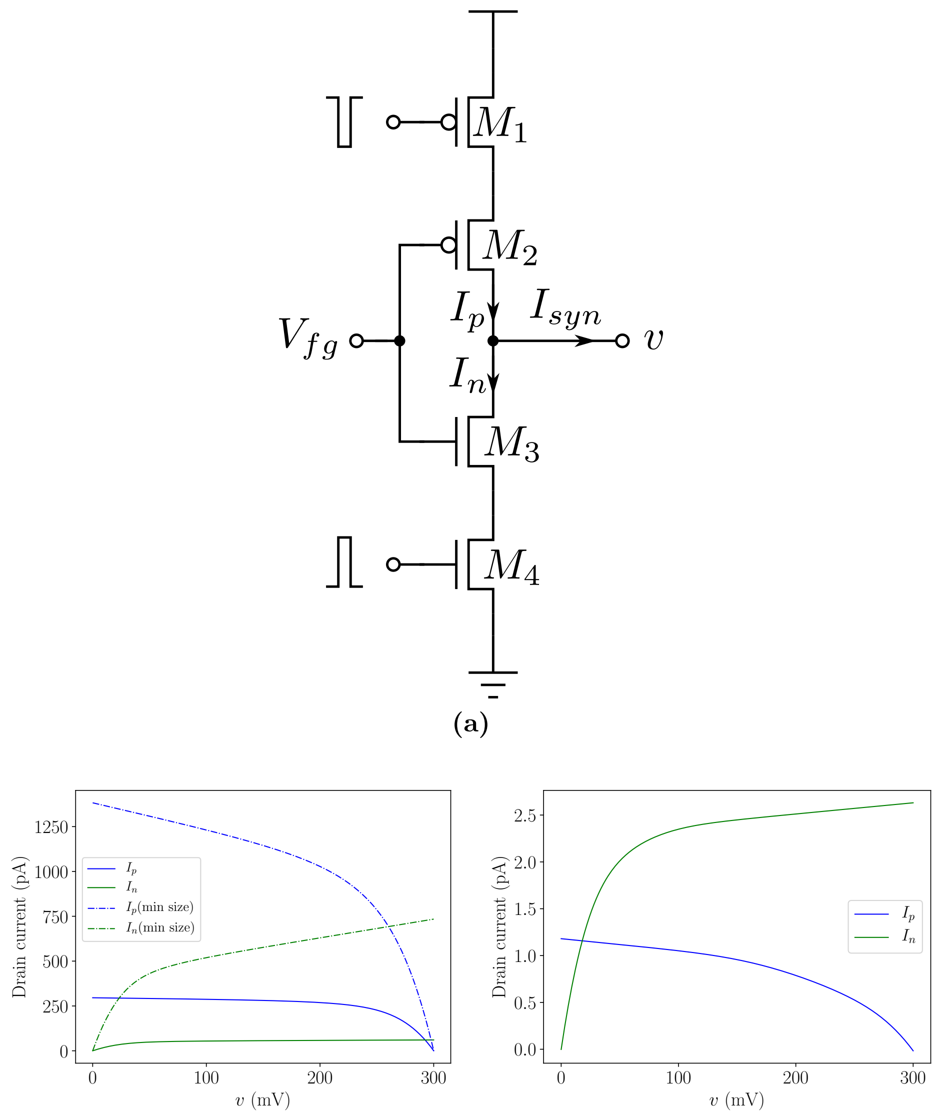
In this chapter, I propose a technique to compensate the leakage current problem in a 130nm CMOS process. I show that with a simple tweak in design of a current mode circuit, leakage currents can be compensated when a large number of synapses are connected together.

5.2 Method

In any current mode synapse circuit, the natural choice is to use a PMOS to supply a current to increase membrane potential and use an NMOS to supply a current to decrease membrane potential. Even when more complex circuits are used to implement learning functionality such as spike timing dependent plasticity (e.g. [43]), eventual current injection to the neuron is accomplished by PMOS and NMOS devices. Hence, I use a synapse circuit with very simple arrangement of NMOS and PMOS devices which can be replaced with complex synapse circuits for which the compensation technique should still hold.

5.2.1 Initial Synapse Circuit

I first start with the initial design of the synapse. The circuit is shown in Fig. 5.1(a). To make the synapse circuit compact, a single synapse designed to supply both excitatory and inhibitory current. A minimum size transistor does not act as a constant current source in saturation as shown in Fig. 5.1(b). Hence, a transistor sizing of 260nm$\times$260nm is chosen to avoid this problem and at the same time maintain small size. A supply voltage of 300mV is used to minimize power. It is assumed that the neuron also has a supply voltage of 300mV. It is not unusual for a neuron circuit to operate at such a low supply voltage because an analog neuron operating at 200mV supply voltage has already been demonstrated [49]. For an active synapse, a pre-synaptic spike is applied



(b) **(c)**

Figure 5.1: Synapse circuit and currents. All the transistors are of size 260nm×260nm. (a) synaptic current from a single synapse is $I_{syn} = I_p - I_n$, (b) active synapse current at V_{fg}=100mV for both the proposed design and a synapse using minimum-sized transistors, (c) inactive synapse current at V_{fg}=100mV. Even though the synapse is inactive there is substantial current that acts as inhibitory current. This current scales up as more synapses are added.

to the gate of M_4 and an inverse spike is applied to the gate of M_1. The voltage V_{fg} controls the drain currents I_p of M_2 and I_n of M_3. The difference of the two currents, $I_{syn} = I_p - I_n$, is injected into the neuron membrane capacitor which changes the membrane voltage v. By controlling V_{fg}, I_{syn} can be made either excitatory or inhibitory. In practice V_{fg} will come from an analog memory device such as a floating gate memory [55]. Fig. 5.1(b) shows excitatory synaptic current as the membrane potential varies. The current is excitatory because I_p is larger than I_n.

For an inactive synapse, without the presence of any spike, the gate of M_2 is pulled down to the ground and the gate of M_1 is pulled up to the supply voltage. As can be seen in Fig. 5.1(c), even when the synapse is inactive, I_{syn} is nonzero and acts as inhibitory current because typically NMOS leakage current is more than the leakage current for the same sized PMOS. There is about I_{syn}=2pA of leakage current acting as inhibitory current for almost the entire range of membrane potential. This much leakage current does not pose a problem if there is another synapse which can supply much larger active synapse current, thus overcoming the leakage current. However, when a large number of synapses are tied together at node v, the leakage current linearly increases to such a value that one active synapse is not able to overcome the leakage currents. For example, if 256 of synapses are tied together, the total leakage current becomes 512pA. Even if an active synapse is able to supply more current than 512pA and raise the membrane potential v, once the pre-synaptic spike is over, membrane potential will very quickly go down because of the large leakage current. Thus it will be almost impossible to get the membrane potential to cross the threshold voltage.

5.2.2 Leakage Current Compensating Circuit

To mitigate the leakage current problem, I propose a leakage current compensation technique. The circuit is shown in Fig. 5.2. The circuit in Fig. 5.1(a) is split at node v. Then, the NMOS current parts to a neuron are bundled

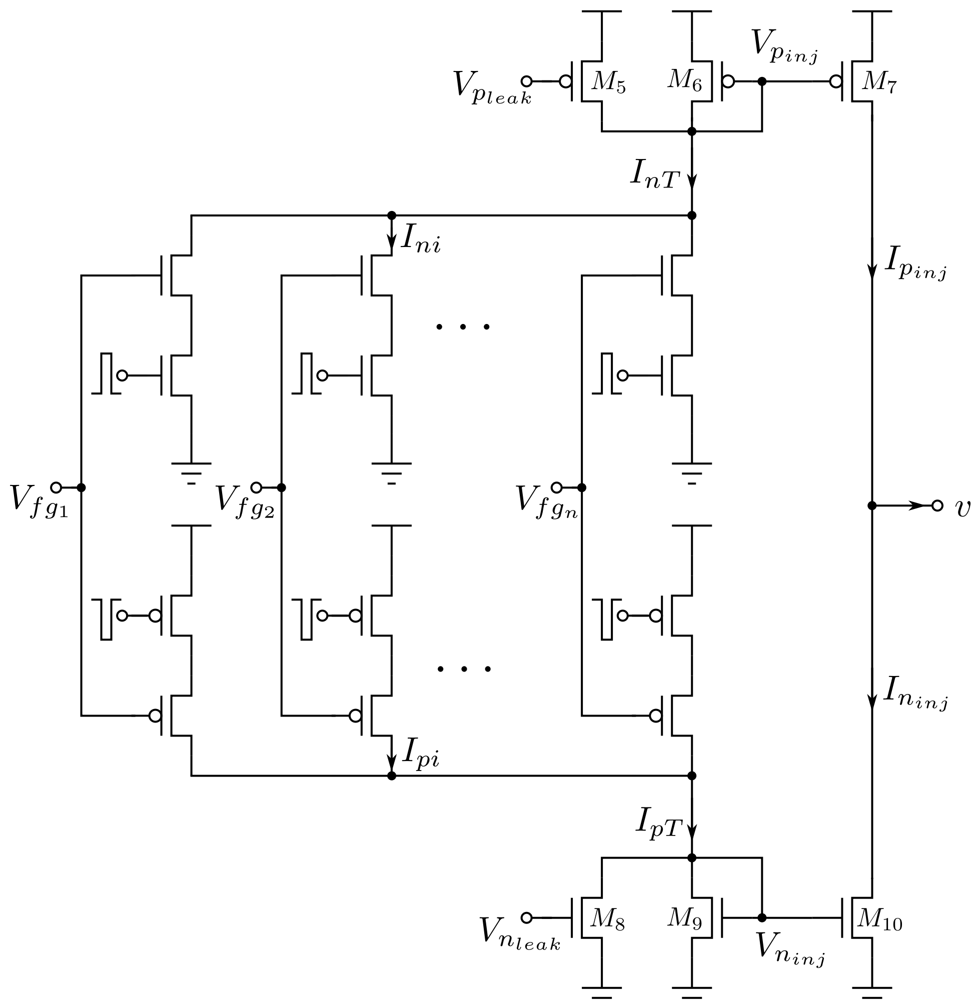


Figure 5.2: Modified synapse circuit to compensate leakage currents.

together as I_{nT}. M_5 supplies the demand of the leakage currents by setting an appropriate gate voltage $V_{p_{leak}}$. The resulting current is copied by M_6 and M_7 to produce $I_{p_{inj}}$ which is injected into the neuron membrane at node v. In a similar fashion, the PMOS current parts to a neuron are bundled together as I_{pT}. Leakage current demand is met with M_8 by setting an appropriate gate voltage $V_{n_{leak}}$. The resulting current is copied by M_9 and M_{10} to produce $I_{n_{inj}}$ which is injected into the neuron membrane. The difference of $I_{p_{inj}}$ and $I_{n_{inj}}$ now acts as total synaptic current and can be both excitatory and inhibitory.

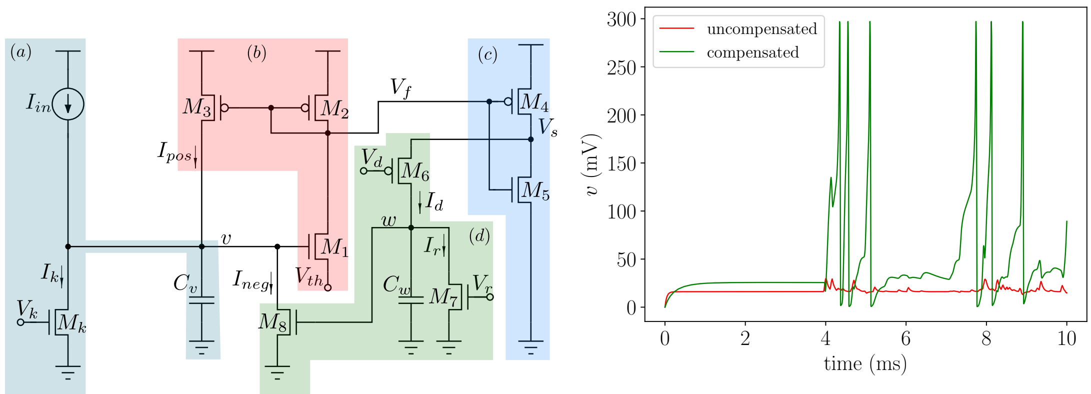
Different layers of a neural network will have different number of synapses connected to a neuron. In that case $V_{p_{leak}}$ and $V_{n_{leak}}$ has to be different for each layer to compensate different level of total leakage current. These voltages can be stored in floating gate memories just like the synapse weights are stored avoiding the need for separate pins.

5.3 Experiment, Results and Discussion

5.3.1 Experimental Setup

To compare the effectiveness of the compensation technique, two test circuits are simulated in a 130nm CMOS process. In one test circuit, 256 uncompensated synapses as shown in Fig. 5.1(a) are connected to a neuron circuit. In the other test circuit, 256 synapses with compensation technique applied as shown in Fig. 5.2, are connected to another neuron circuit. Same set of 256 random weights for the 256 synapses are set to both test circuits. The inputs to the 256 synapses are also same for both test circuits which are set by a randomly selected MNIST [56] image. MNIST is a handwritten digit dataset which is popular for machine learning. The image is resized to size 16×16. The pixel values from the MNIST image serve as the input spike frequency in Hz. Then, the spike trains are delivered to the synapses using VerilogA blocks with each spike having a 45μs spike width. The maximum spike frequency from an input pixel is 255Hz.

The neuron used for the experiment is shown in Fig. 5.3(a). The operation of the neuron circuit is presented briefly here. The neuron circuit is divided into four blocks. Block a serves as input block. An input current I_{in} charges up membrane capacitor C_v and increases membrane potential v. Block b implements positive feedback to generate a spike. When the gate of M_1 becomes larger than the source voltage V_{th} which acts as spiking threshold voltage, M_1 starts to conduct



(a) **(b)**

Figure 5.3: (a) The neuron circuit used in the experiment, (b) comparison of neuron membrane potential with leakage compensated synapses vs uncompensated synapses. With uncompensated synapses, the membrane potential barely increased by a pre-synaptic spike. Weights and inputs are same in both compensated and uncompensated cases.

current. In subthreshold regime, the drain current of M_1 increases exponentially with v which is copied using M_2 and M_3 to produce I_{pos} and injected into the membrane capacitor. This implements positive feedback and rapidly increases v to supply voltage thus generating a spike. Block c is an inverter which generates a square pulse to indicate a spike for the next layer. Block d serves the function of membrane voltage resetting, spike width and refractory period controller. Using M_6 and M_7, a second capacitor C_w charging and discharging is controlled. When voltage w increases substantially, M_8 discharges the membrane capacitor C_v. M_k compensates leakage current from M_3 thus preventing the neuron from spiking spontaneously when I_{in} is zero. The neuron circuit has a supply voltage of 300mV, same as the synapses.

Table 5.2: Overhead associated with compensation per neuron

Area	Parameter	Active Syn. Power
6 extra MOS	2 V_{leak}	2×

5.3.2 Results

The time evolution of membrane potential for both test circuits are shown in Fig. 5.3(b). It clearly shows the effect of leakage currents in a large number of synapses. For a 255Hz input, the first pre-synaptic spike occurs at 3.9ms. For compensated synapse circuit, the neuron membrane potential changed in expected ways. For large synaptic currents, the neuron spiked with just one or two pre-synaptic spikes. For smaller synaptic currents, membrane voltage increased fast but decreased slowly (around 6ms) because the leakage current is low. However, for the uncompensated synapses, the membrane potential barely increased. The same synaptic current which made the compensated circuit spike, hardly made any impact in the uncompensated circuit. Moreover, as soon as the membrane voltage increased, it also decreased very fast because of the combined large inhibitory leakage current from the synapses. These results show that the compensation technique can mitigate the leakage current problem in large scale synapses for a technology node as small as 130nm. The same technique can potentially work for smaller technology nodes as well.

5.3.3 Discussion

The number of synapses connected to a neuron is typically different for different layers of a neural network. In a fully connected network, there are same number of synapses connected to all the neurons in a given layer. Hence, total synapse leakage current for every neuron is expected to be the same. Fig. 5.4 shows the dependence an of inactive synapse leakage current on V_{fg}. For PMOS the

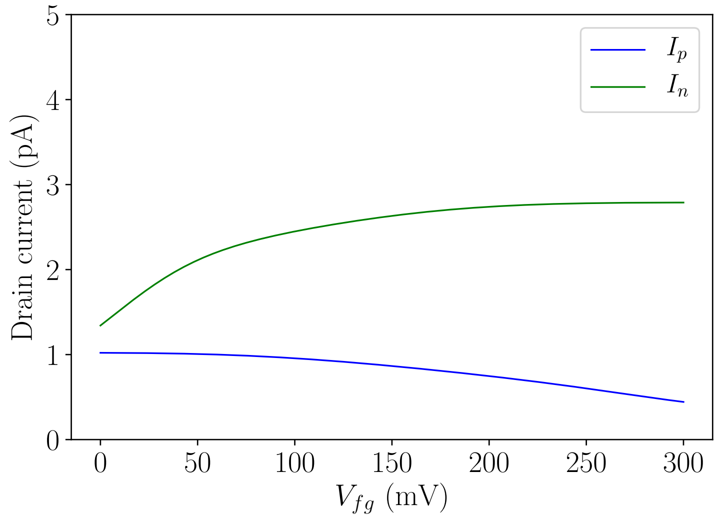


Figure 5.4: Inactive synapse leakage current dependence on V_{fg}. The currents are shown for v=150mV as V_{fg} is varied.

variation is low. For NMOS the variation is only about 1.5pA on its entire range from ground to supply voltage. Hence, the leakage current can be considered approximately independent of V_{fg}. Moreover, the distribution of weights in a large number of synapses tends to average out the total leakage current to a same value for each neuron. In that case, $V_{p_{leak}}$ and $V_{n_{leak}}$ can be shared across all the neurons in a given layer. Table. 5.2 shows the overhead associated with the compensation technique on a per neuron basis. Active synapse power is twice the uncompensated circuit because the currents that will flow in an active synapse also have to flow in M_7 and M_{10}. However, a synapse is active only momentarily hence the power overhead is not huge.

5.4 Chip Implementation

The neuron circuit and synapse design has been submitted for fabrication in the first ever open source multi project wafer (MPW) funded by Google. The

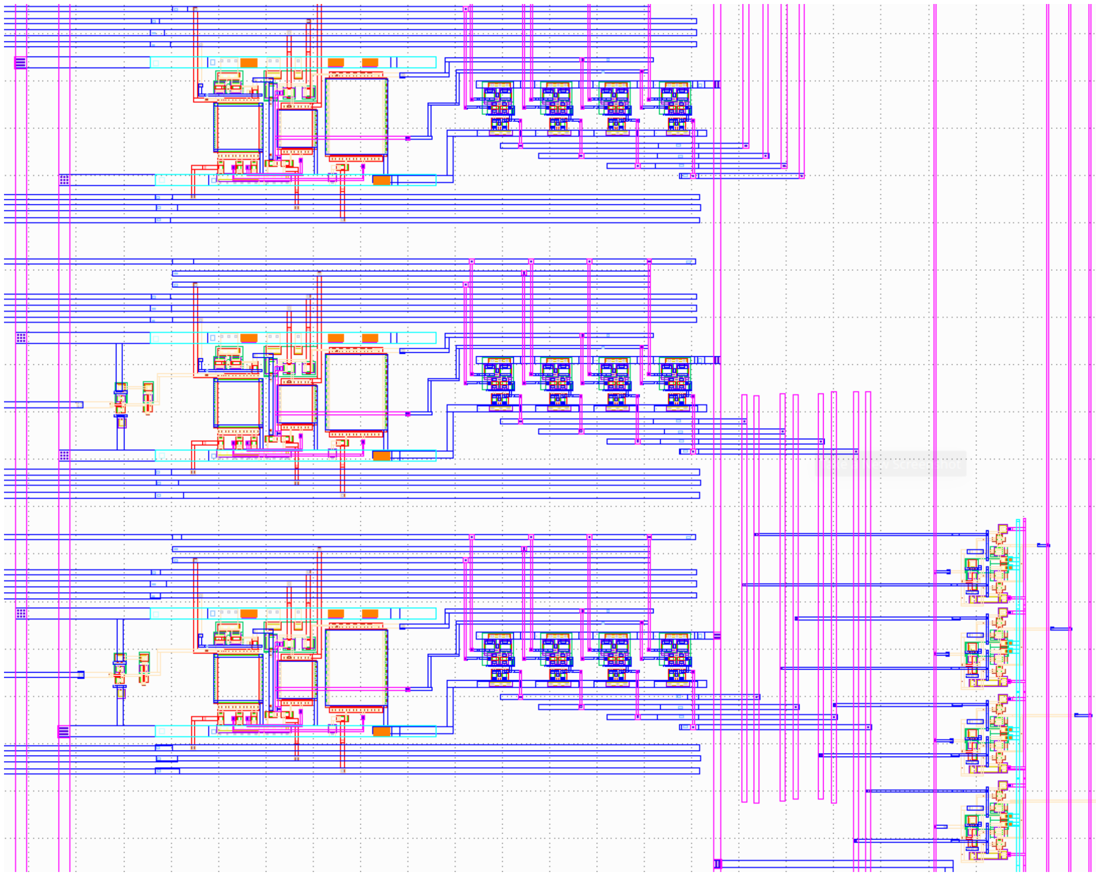


Figure 5.5: Layout of frew neuron circuits with measurement circuitry.

Skywater130 pdk has been used for the design wich is a hybrid 130nm-150nm process node. Open source design frameworks such as ngspice, magic layout, klayout and design flow from skywater project has been used to design the circuits and embed the design into the chip. Fig. 5.5 shows the layout of few neuron circuits along with the voltage measurement circuits. Analog multiplexers have been used to send the outputs of the neurons to the chip pads. The right-bottom circuits in the figure shows the multiplexers.

In order to test the behavior of the analog memory of the pdk a 4×4 SONOS cell has also been implemented in the chip design wich is shown in Fig. 5.6. SONOS stands for silicon-oxide-nitride-oxide-silicon. SONOS cells are basically

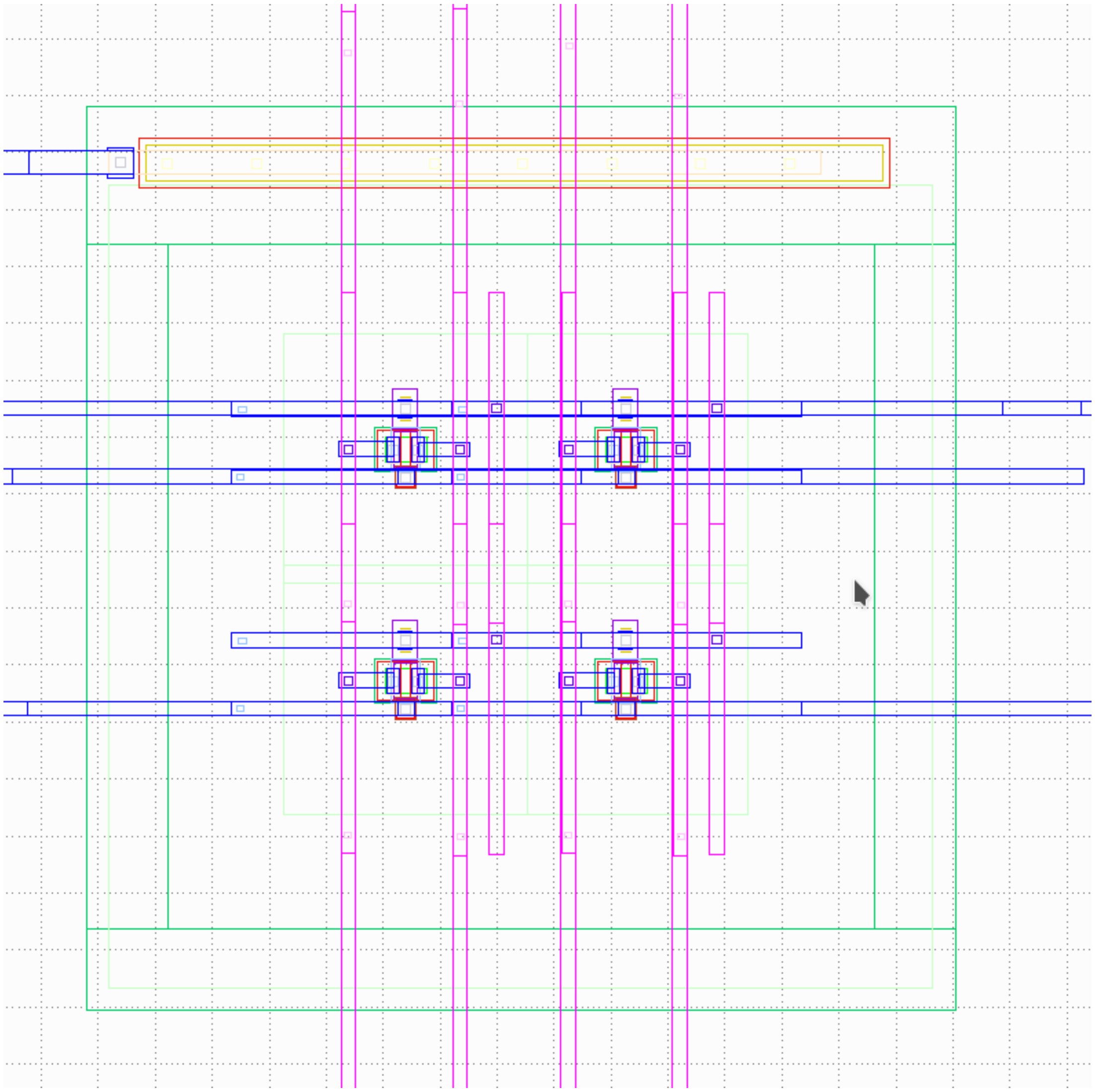


Figure 5.6: A 4 × 4 sonos cell array.

transistors which have additional layers of gate oxide and nitride materials to trap electrical charges.

5.5 Conclusion

In this chapter, I have proposed a technique to compensate the leakage currents from a large number of synapses. I have used simple synapse circuit to demonstrate the technique. This simple synapse circuit can be replaced with complex synapses without affecting the compensation technique. I have shown simulation results to demonstrate the viability of the proposed technique. It is expected that the same techniques will also work for much smaller technology nodes.

Chapter 6

Hardware Model Based Simulation of Spiking Neural Network

6.1 Introduction

Spiking neural networks have been gaining significant interest in recent times which has led to some interesting research endeavors such as machine learning tasks using spiking neurons [32], event based systems [50], silicon retina [57] etc. Since analog computation provides excellent energy efficiency, numerous hardware implementations of large scale VLSI spiking neural networks have been proposed [43, 58]. These chips are fabricated for deployment and testing of different models of learning. These models of learning, however, are formulated and refined using spiking network simulators which do not account for hardware device nonidealities. Thus, the performance of a network designed in a spiking network simulator is not generally representative of the performance of the same network deployed in real custom hardware. There are techniques such as deep modeling [59] which use deep learning frameworks to estimate circuit nonidealities in scaling and bias error parameters. Some techniques [60] map neuronal models onto hardware once the chip is fabricated. Mapping parameters are estimated from fitting chip output with neuronal model. Then these mappings are used to set biases on the chip. However, this requires fabrication of the chip first without the knowledge of how nonidealities will affect the neuron behavior. Hence, for cost saving reasons it is necessary to incorporate circuit behaviour into the simulation before manufacturing a chip.

There are many spiking network simulators [61, 62] available at this moment. Some simulators allow custom description of neuron and synapse equations. With these kind of simulators, ideally, hardware neuron and synapse models can be described and simulated. However, one would require the parameters of silicon process such as subthreshold slope factor, early voltage, body effect coefficient, diffusion capacitance etc. to formulate current and voltage equations. Many of these parameters do not have closed form representations. Also, it is not straightforward to formulate current and voltage equations from foundry-provided BSIM [63] models.

Neuron dynamics generally have the characteristics of a dynamical system which makes it possible to use the phase plane to analyze a neuron [45, 64]. However, I can also use the phase plane to account for device nonidealities. In this chapter, I describe a process of incorporating BSIM-model based device nonidealities in the simulation of spiking neurons with an existing spiking neural network simulator using phase plane. I first describe the process of using a phase plane to obtain the solution of a neuron equation. Then, I present the simulation of a hardware neuron using a spiking neural network simulator. With the aid of hardware model based simulation, behaviour of neural networks can be observed quantitatively in the presence of device nonidealities.

6.2 Overview of Phase Plane Analysis

6.2.1 Phase Plane

In this section I provide an overview of using phase plane to solve differential equation. To demonstrate this, I choose a well known neuron model called

Fitzhugh-Nagumo (FHN) model [65]. Later I will carry over the ideas developed here to our neuron circuit. The FHN model is described by two first-order Ordinary Differential Equations (ODE) given by (6.1).

$$\frac{dv}{dt} = f(v, w) = v - \frac{1}{3}v^3 - w + I \tag{6.1a}$$

$$\frac{dw}{dt} = g(v, w) = \epsilon(v + a - bw) \tag{6.1b}$$

Here v is membrane potential, w is recovery variable, I is input current to the neuron and ϵ, a, b are constants. These differential equations describe a dynamic system which can be analyzed using phase plane which is also known as state space. A phase plane represents every possible state of a dynamic system with each possible state representing a unique point in the space. Since FHN system needs two first order differential equations in v and w, its phase plane is two dimensional which is represented by $v - w$ plane. Each point in the plane represents a state $(v(t), w(t))$ at some point in time. After some time Δt that point will evolve and move to another point $(v(t+\Delta t), w(t+\Delta t))$. The direction of the movement will be determined by the velocity of v and w which are given by $f(v, w)$ and $g(v, w)$ respectively. Fig. 6.1 shows a velocity field for $I = 0$. By following the arrows I can track the trajectory of a point as it evolves over time. The points where velocity of v are zero are called v-nullcline and shown as $\dot{v} = 0$ line. The line is obtained from condition $f(v, w) = 0$. Similarly w-nullcline is shown as $\dot{w} = 0$ line. The point where two nullclines meet is a fixed point. A trajectory of a point P is shown in the figure which evolves along the direction of the arrows.

I can use the phase plane to solve for the time domain solution of the dynamical system. Any initial condition (v_0, w_0) will be a point on the phase plane. Then I can simply follow the path along the velocity arrows to find the

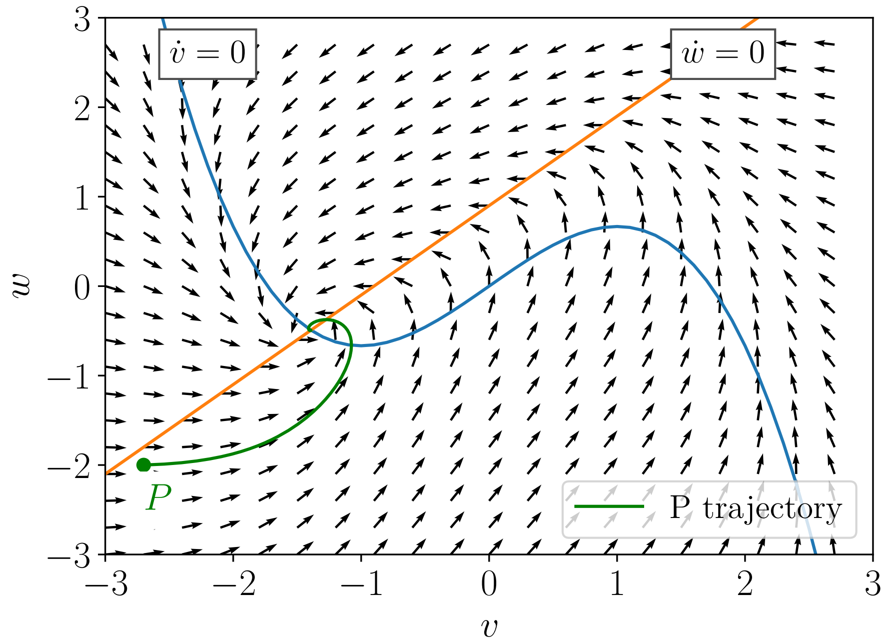


Figure 6.1: Phase plane and nullclines of FHN model for $\epsilon = 1.25$, $a = 0.9$, $b = 1$, $I = 0$. Velocities are scaled to unit value. The trajectory of point P moves in the direction of arrows.

next time step values of $(v(t), w(t))$. When the phase plane is available I do not have to calculate $f(v, w)$ and $g(v, w)$ at every time step because they are already stored in the phase plane. This property will be very useful later when I will consider hardware models.

6.2.2 Solving ODE Using Phase Plane

In a digital computing platform the phase plane is represented in the form of a 2D meshgrid array. Using this meshgrid I can solve ODE using phase plane. In this meshgrid, I define i as the row index and j as the column index. Then choosing a state (v, w) is equivalent to selecting an element (i, j) from the 2D meshgrid array. If I choose an initial condition (i_0, j_0) corresponding to (v_0, w_0),

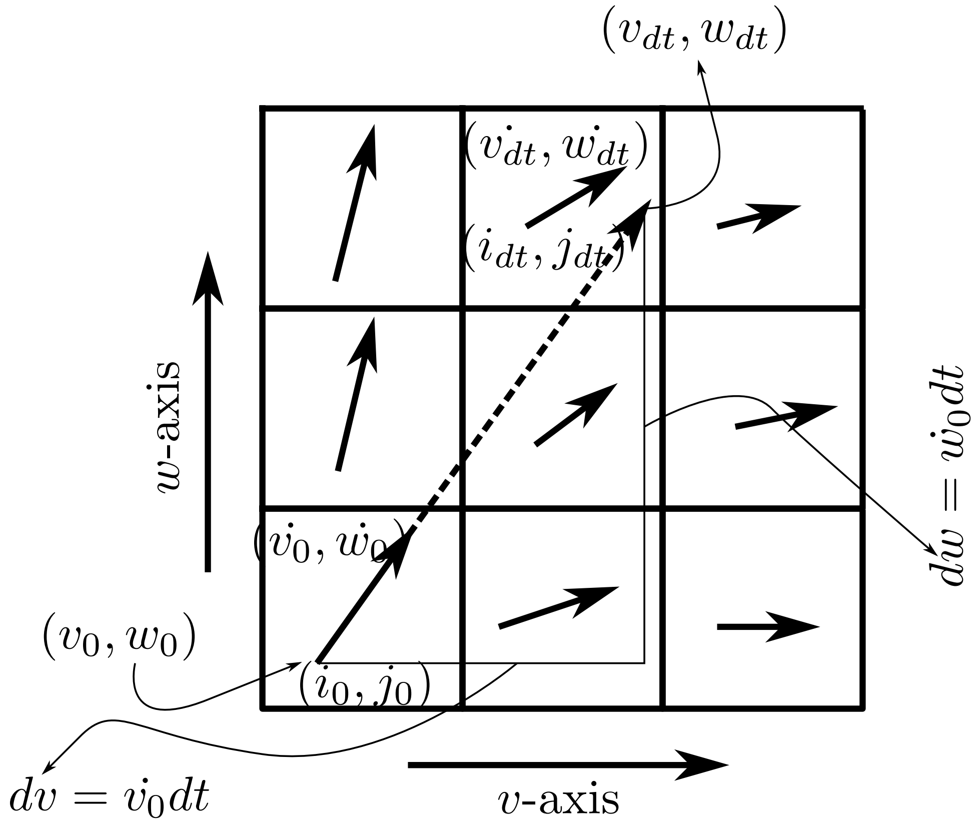
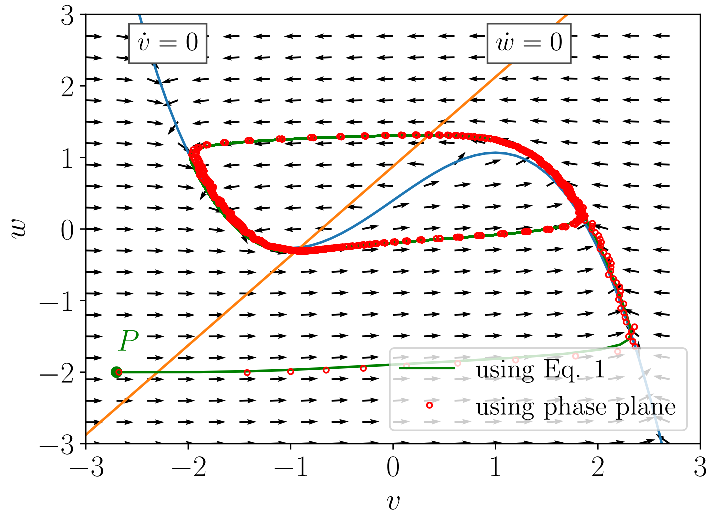


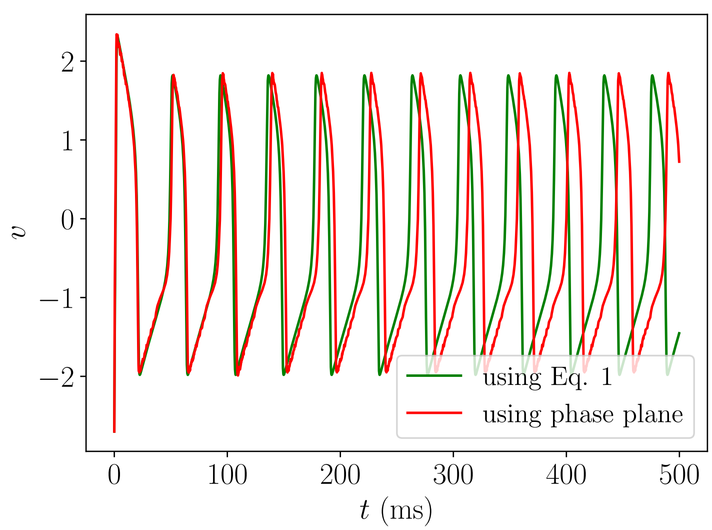
Figure 6.2: ODE solving using phase plane meshgrid. Dotted line shows jump of initial point over time step dt.

the corresponding velocity is $(\dot{v}_0, \dot{w}_0)$. Then after time step dt, finding the next state means moving a distance $(dv, dw) = (\dot{v}_0 dt, \dot{w}_0 dt)$ along the direction of the velocity. This is shown in Fig. 6.2.

After moving a distance of (dv, dw) in the 2D array, I arrive at the next state (v_{dt}, w_{dt}) and land on element (i_{dt}, j_{dt}). For next time step I must use the velocity that exists at index (i_{dt}, j_{dt}). This way I continue finding the next state and consequently the solution of the ODE. The process described here is just numerical integration. However, instead of feeding (v_{dt}, w_{dt}) to Eq. 6.1 for next time step to find the velocities, I use velocity values from 2D meshgrid array.

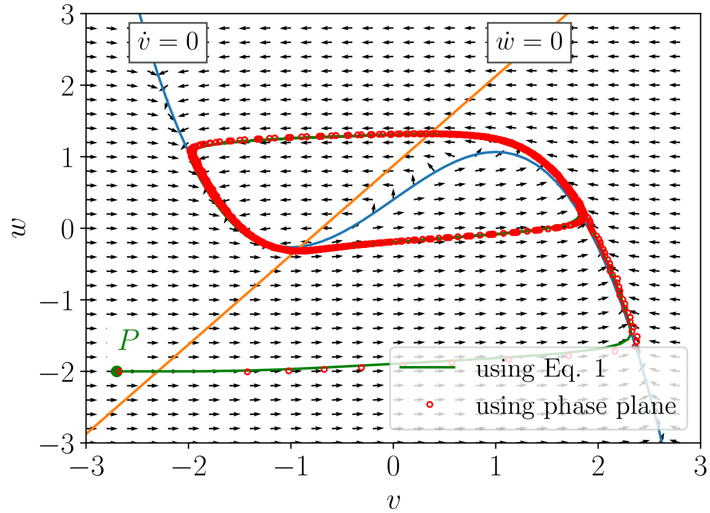


(a)

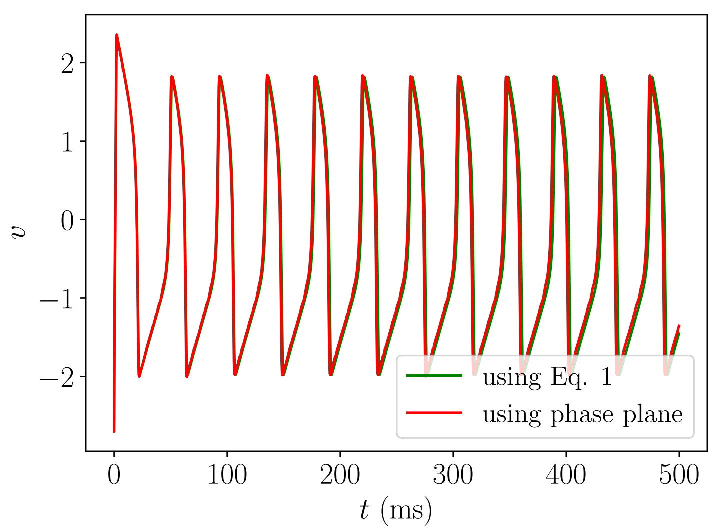


(b)

Figure 6.3: (a) Trajectory of point $P = (-2.7, -2.0)$ and (b) time domain solution obtained by solving ODE using phase plane and Eq. 6.1. FHN model parameters: $\epsilon = 0.08$, $a = 0.7$, $b = 0.8$, $I = 2$. Meshgrid step size is 0.1 on both axis.



(a)



(b)

Figure 6.4: (a) Trajectory of a point $P = (-2.7, -2.0)$ and (b) time domain solution obtained by solving ODE using phase plane and Eq. 6.1. FHN model parameters: $\epsilon = 0.08$, $a = 0.7$, $b = 0.8$, $I = 2$. Meshgrid step size is 0.05 on both axis.

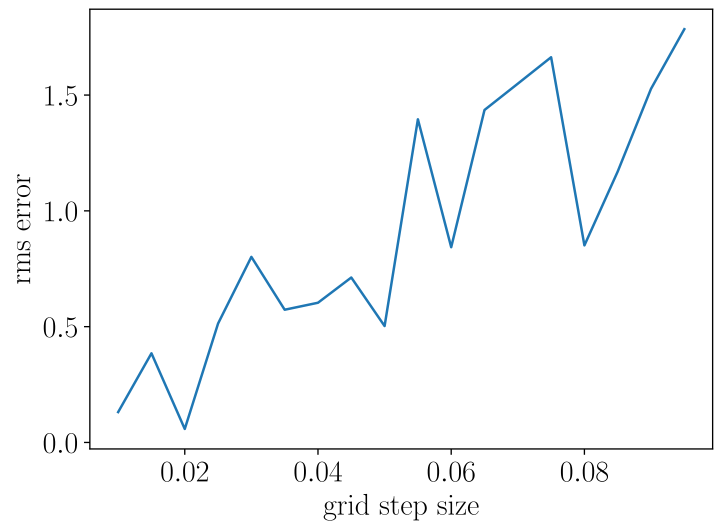


Figure 6.5: root mean squared error variation of phase plane solution with the solution from equations as meshgrid step size varies.

Fig. 6.3 shows the trajectory and time evolution of a point found by both Eq. 6.1 and phase plane method for a meshgrid created with a step size of 0.1 on both axes. On the trajectory plot we can see that solution using phase plane follows a path very close to the actual solution. However, in time domain plot the phase plane solution lags behind the actual solution as time moves forward. This is because of the meshgrid is not dense enough. Just like the case that a numerical integration produces error when the time step is large, phase plane integration produces error if the meshgrid step size is large. If we create a denser meshgird using a smaller step size and use that for solving ODE, then the error between actual time domain solution and phase plane solution should go away. Fig. 6.4(a), (b) shows the trajectory and time evolution of a point found by both Eq. 6.1 and phase plane method for a meshgrid created with a step size of 0.05 on both axes.

This analysis shows that it is possible to get reasonable accuracy in ODE solution using phase plane where velocities are stored in a meshgrid. An optimum size of the meshgrid can be found imperially by plotting the root mean squared error between the phase plane solution and solution directly from equations as a function of meshgrid step size as shown in Fig. 6.5. Depending on the speed of simulation trade off can be made between error and step size. Next, I will use this process to find the time domain solution of a spiking neuron implemented with MOSFET transistors.

6.3 Silicon Circuit Using Phase Plane

6.3.1 Neuron Circuit

The neuron circuit [2] I have used in this work is shown in Fig. 6.6. In practice any neuron circuit can be used. The circuit consists of four blocks. Block a serves as input. Block b provides thresholding and positive feedback to generate spikes. Block c is a simple inverter acting as axon. Block d controls reset, spike width and refractory period. The operation of the circuit is explained below.

Block a

The input block consists of input current I_{in} to the neuron, a leak transistor M_k and membrane capacitor C_v. The leak transistor M_k subtracts some current I_k from I_{in}. The net current $I_{in} - I_k$ charges C_v, and the membrane voltage v increases.

Block b

Membrane voltage v is applied to the gate of M_1. When v exceeds the source voltage v_{th}, M_1 starts to conduct current. This current is copied using M_{2-3} and fed back into the membrane capacitor C_v. In subthreshold regime, the drain current of M_1 is exponentially related to the gate to source voltage. Hence, as gate

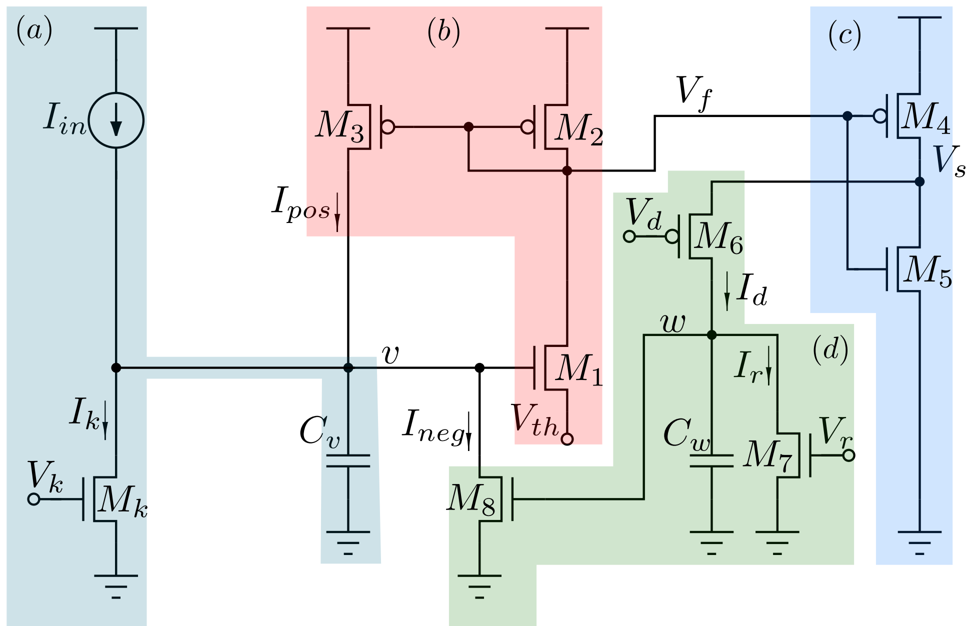


Figure 6.6: Silicon neuron circuit [2]. For $M_{1-3,5-8}$ W/L = 260nm/260nm, For M_4 W/L = 800nm/260nm, C_v = 50fF, C_u = 30fF.

to source voltage of M_1 increases, the exponential current I_{pos} further increases v and thus implements positive feedback. The exponential positive feedback current very quickly increases v to the top voltage rail V_{dd}, which generates the spike. Since M_1 is not active when v is below v_{th}, v_{th} acts as spiking threshold.

Block c

When a spike is generated because of the positive feedback, drain voltage V_f of M_1 goes down. This is applied to the inverter formed by M_{4-5}. As a result the inverter output V_s goes up. V_s goes up only when v spikes. Thus the inverter acts like an axon.

Block d

When V_s is at V_{dd}, capacitor C_w is charged by current conduction through M_6, and voltage w increases. Voltage w is connected to the gate of M_8 which draws current I_{neg} away from C_v. As w increases, I_{neg} overpowers I_{pos}, C_v is discharged and the neuron resets. After the neuron is reset, C_w is discharged using M_7 so that the neuron can start its spiking operation again. The refractory period is implemented by discharging C_w slowly using V_r. V_d controls the charging time of C_w thereby controlling the spike width.

6.3.2 Mathematical Description

The dynamics of the neuron in Fig. 6.6 can be described by (6.2). The neuron dynamics is described by two states v and w.

$$\frac{dv}{dt} = f(v, w) = \frac{1}{C_v}(I_{in} - I_k + I_{pos} - I_{neg}) \tag{6.2a}$$

$$\frac{dw}{dt} = g(v, w) = \frac{1}{C_w}(I_d - I_r) \tag{6.2b}$$

To solve the ODE in Eq. 6.2 using phase plane, 2D meshgrids of $f(v, w)$ and $g(v, w)$ need to be generated. Using Python scripting tools [66], the 2D meshgrids of these functions are generated by DC parametric sweep of v and w in Cadence spectre simulation and I_{pos}, I_{neg}, I_k, I_d, I_r are recorded in a text file for a given set of control voltages. The capacitance values used for creating the meshgrids are slightly larger than C_v and C_w. This is because gates of M_1 and M_8 are adding additional capacitance to C_v and C_w respectively. For this reason, an estimate of the parasitic gate capacitance C_p is added to both C_v and C_w. The currents I_{pos}, I_{neg}, I_k are functions of variable gate and drain voltages of their respective transistors, not a function of I_{in}. Hence, as shown in (6.3), a meshgrid F_{grid} created once, can be used for arbitrary value of I_{in}.

$$C'_v = C_v + C_p \tag{6.3a}$$

$$C'_w = C_w + C_p \tag{6.3b}$$

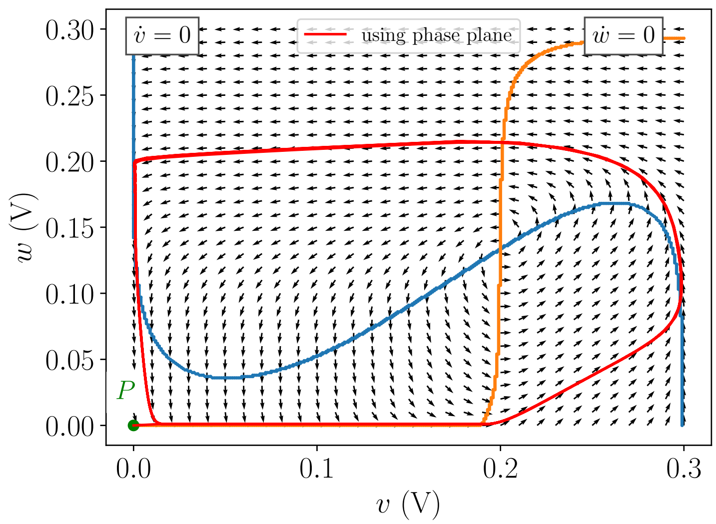
$$F_{grid}(v, w) = I_{pos}(v, w) - I_{neg}(v, w) - I_k(v, w) \tag{6.3c}$$

$$f_{grid}(v, w) = \frac{1}{C'_v}(F_{grid}(v, w) + I_{in}) \tag{6.3d}$$

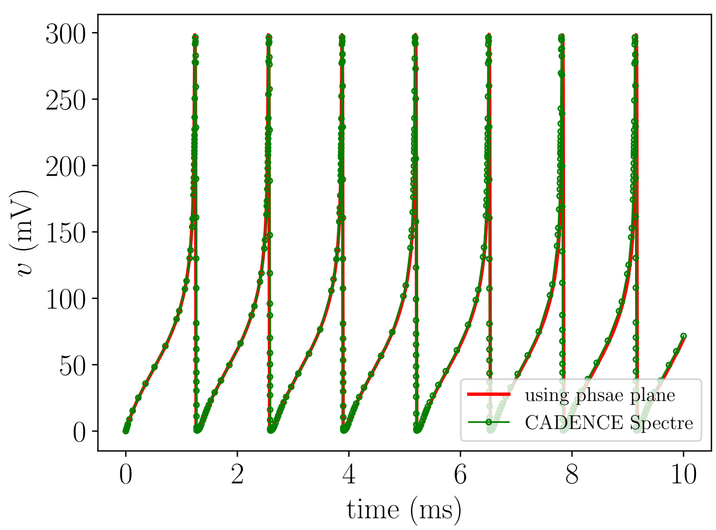
$$g_{grid}(v, w) = \frac{1}{C'_w}(I_d(v, w) - I_r(v, w)) \tag{6.3e}$$

6.3.3 Solving Circuit ODE Using Phase Plane

Using the procedure as outlined in section 6.2.2, a solution of the membrane voltage v of the circuit in Fig. 6.6 is shown in Fig. 6.7(a), (b). The resulting time domain solution is in very good agreement with Cadence spectre solution. Fig. 6.8 shows that for the same settings as in Fig. 6.7, phase plane solution and Cadence spectre solution match well for a variety of I_{in}. To capture nonidealities I would have needed to formulate equations of the transistor currents which requires the values of some parameters of the silicon process. Many of these quantities do not have simple closed from expression and require numerical methods to solve in most modern silicon processes. Thus the expression of the currents of the neuron circuit would have been very complex and intractable. In addition, the phase plane solution can be obtained with already available spiking neural network simulator Brian2 [62]. Brian2 has the functionality to accept user defined functions. With this functionality I am able to feed the meshgrid values to the



(a)



(b)

Figure 6.7: (a) Trajectory of point $P = (0, 0)$ and (b) time domain solution obtained by solving ODE using phase plane for $I_{in} = 6$pA and time domain membrane voltage trace. Voltage settings: $V_{dd} = 300$mV, $V_k = 10$mV, $V_{th} = 50$mV, $V_d = 80$mV, $V_r = 100$mV and capacitor values: $C_v = 50$fF, $C_w = 30$fF, $C_p = 5$fF.

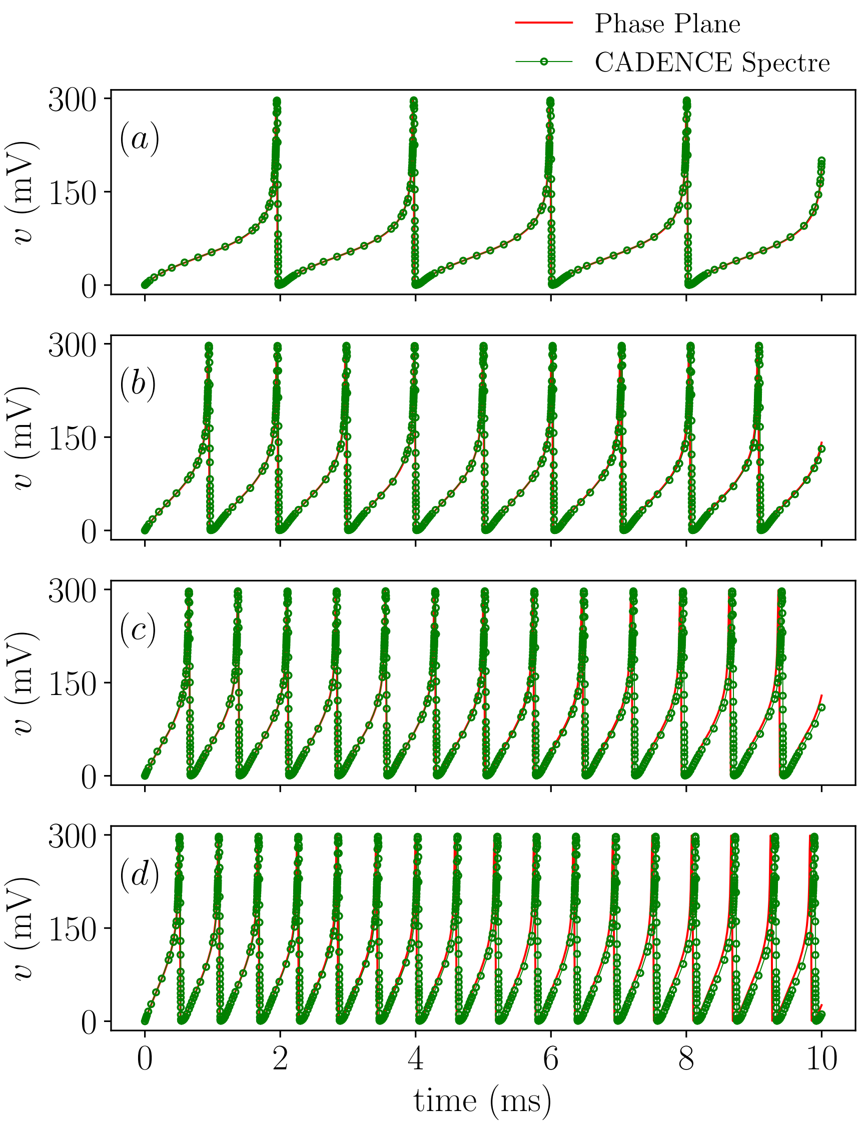


Figure 6.8: Phase plane solution and Cadence spectre solution for (a) I_{in} = 4pA, (b) I_{in} = 8pA, (c) I_{in} = 12pA, (d) I_{in} = 16pA.

simulator. The only overhead is to generate the meshgrids, which is a simple parametric sweep simulation. Using phase plane, the time domain solution of the neuron circuit ODE can be obtained with realistic device nonidealities.

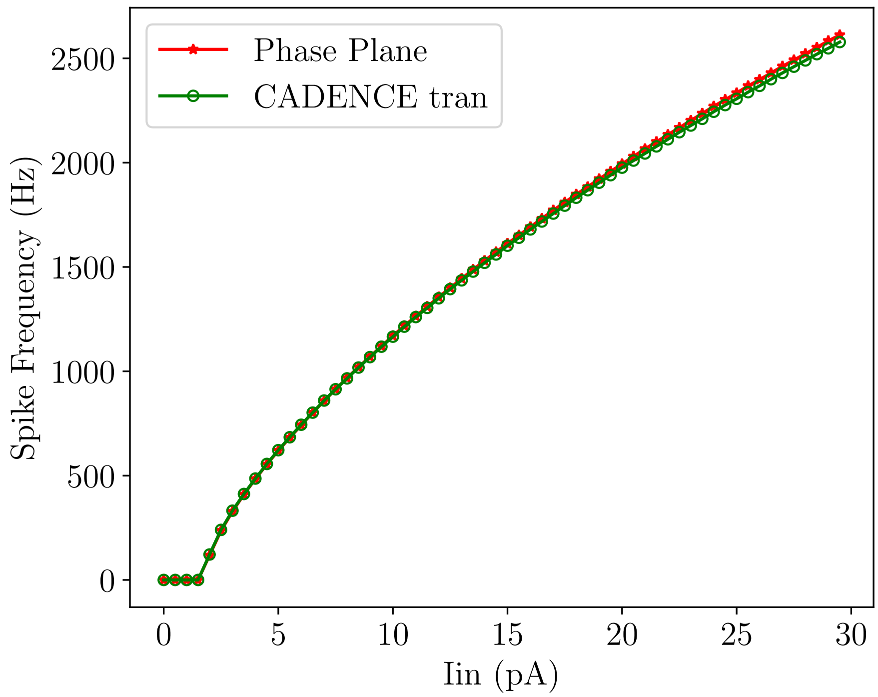


Figure 6.9: Frequency vs input current curve of the neuron circuit.

The phase plane is obtained by DC parameter sweep which does not take into account the transient effects of the transistors such as source to body, gate to drain capacitive currents. These transient effects have negligible effect on the operation of the neuron. Fig. 6.9 shows a Frequency vs Input current (F-I) curve comparison of phase plane solution and Cadence Spectre transient solution where the transient effects are manifested as a slight difference in output frequency at higher input current. The difference in output frequency at higher current is about 1-2% which can be safely ignored. Here, I have shown the solution process of a two dimensional phase plane. The process can be generalized to more than two dimensional system as well.

6.3.4 Meshgrid Size and Memory Access Time

For accurate solution using phase plane, the meshgrid needs to be dense or equivalently the step size of the grid needs to be small. This means that the meshgrid or the array will be large. The ODE solver needs to access an element from the array at each time step. The access time of an element from an array is constant and independent of the size of the array. Hence, large meshgrid will not slow down the ODE solving process. The Meshgrid used in Fig. 6.7 is generated with a step size of 1mV on both axes which produced a 301×301 element array. This was dense enough to produce a solution that is in good agreement with Cadence spectre.

6.4 Neural Network Simulation

With the aid of phase plane simulation, I can carry out simulation of a network of spiking neurons. I have chosen the MNIST [56] handwritten digit dataset to demonstrate that. To do that, I also need hardware realistic model of the synapses. This is done in a similar manner as neuron meshgrid generation which is described below.

6.4.1 Synapse Circuit

The synapse circuit I used, is shown in Fig. 6.10(a). V_{fg} comes from an analog memory device such as floating gate memory [67, 55] which controls drain current of M_{10} and M_{11}. The difference of PMOS and NMOS current acts as synaptic current which is injected to the membrane potential node v. A presynaptic spike and its inverse are applied on the gates of M_9 and M_{12} respectively. When there is no spike, M_9 and M_{12} are turned off and the synapse is inactive. When there is an incoming spike, M_9 and M_{12} are turned on and the synapse is active. This

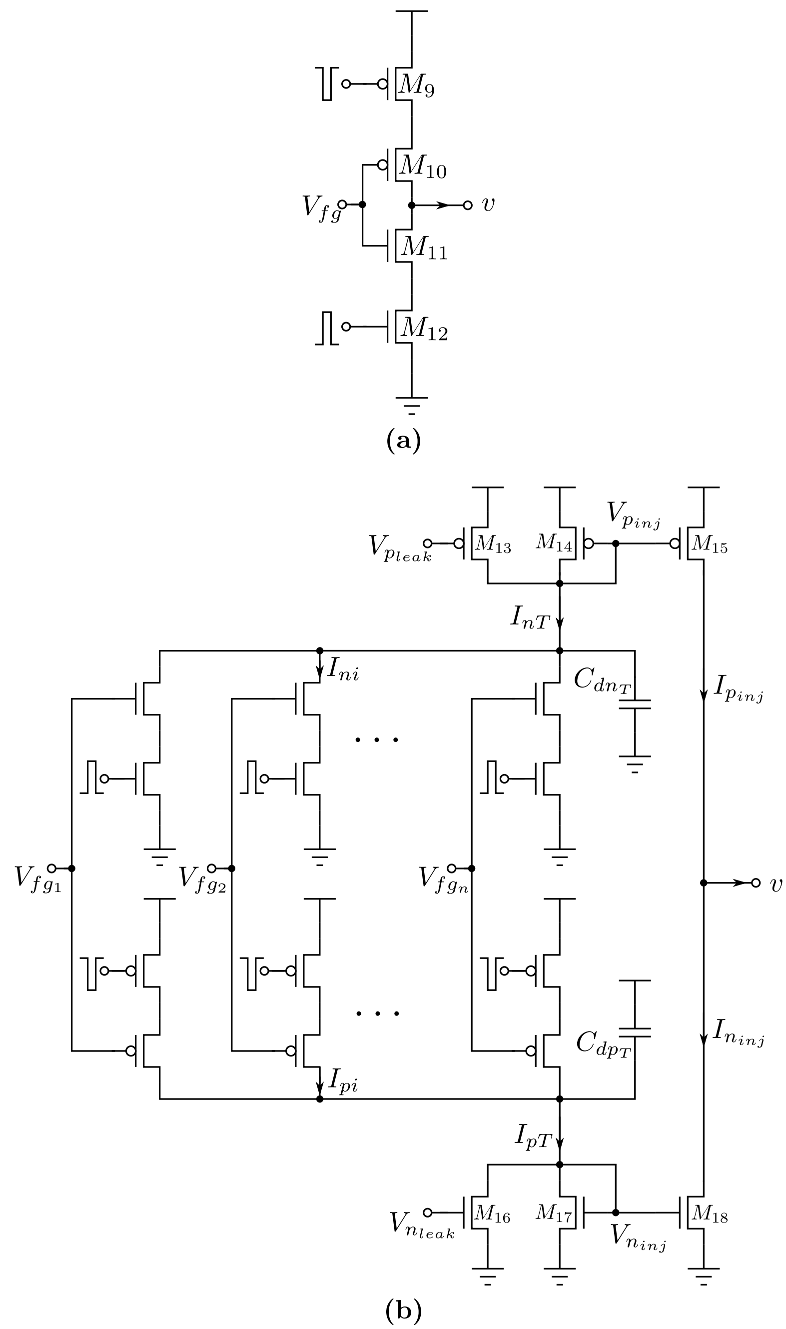


Figure 6.10: (a) Synapse circuit (b) Synapse bundle circuit to eliminate leakage current. Every transistor has size W/L = 260nm/260nm.

circuit is simple and it takes relatively small area. However, in this form of the circuit when a large number of synapses are tied together at node v, the leakage currents of the inactive synapses become so high that it acts as inhibitory current which prevents the neuron from spiking. To eliminate this problem, the NMOS current parts of all the synapses to a neuron are bundled together by tying the drain nodes, then collected by M_{14} as shown in Fig. 6.10(b). C_{dnT} represents the total NMOS drain to body capacitance. When there are large number of synapses the collective drain capacitances become large enough to affect the neural dynamics. M_{13} supplies the demand of leakage currents by setting an appropriate value of $V_{p_{leak}}$. The resulting current is copied by M_{15} with the help of injection voltage $V_{p_{inj}}$ and injected into v node of the neuron. Similarly, the PMOS current parts are bundled and collected by M_{17}, leakage current demand is met by M_{16} by setting $V_{n_{leak}}$, copied using M_{18} with the help of injection voltage $V_{n_{inj}}$ and injected in node v of the neuron. C_{dpT} represents the total PMOS drain to body capacitance. Denoting the total drain current from M_{13} and M_{14} as I_{pB} and total drain drain current from M_{16} and M_{17} as I_{nB}, the dynamics of the injection voltages are given by Eq. 6.4. Here, an estimation of 0.5fF per PMOS and 1fF per NMOS has been used for individual drain capacitance of the synapse.

$$\frac{dV_{p_{inj}}}{dt} = \frac{1}{C_{dnT}}(I_{pB} - I_{nT}) \tag{6.4a}$$

$$\frac{dV_{n_{inj}}}{dt} = \frac{1}{C_{dpT}}(I_{pT} - I_{nB}) \tag{6.4b}$$

6.4.2 Synapse Model Extraction

In every time step of a simulation, individual NMOS synapse currents I_{ni} are determined for a given V_{fg}, summed to $I_{nT} = \sum I_{ni}$ from which $V_{p_{inj}}$ is determined for a given $V_{p_{leak}}$. Finally, $I_{p_{inj}}$ is determined from the value of $V_{p_{inj}}$. Similar

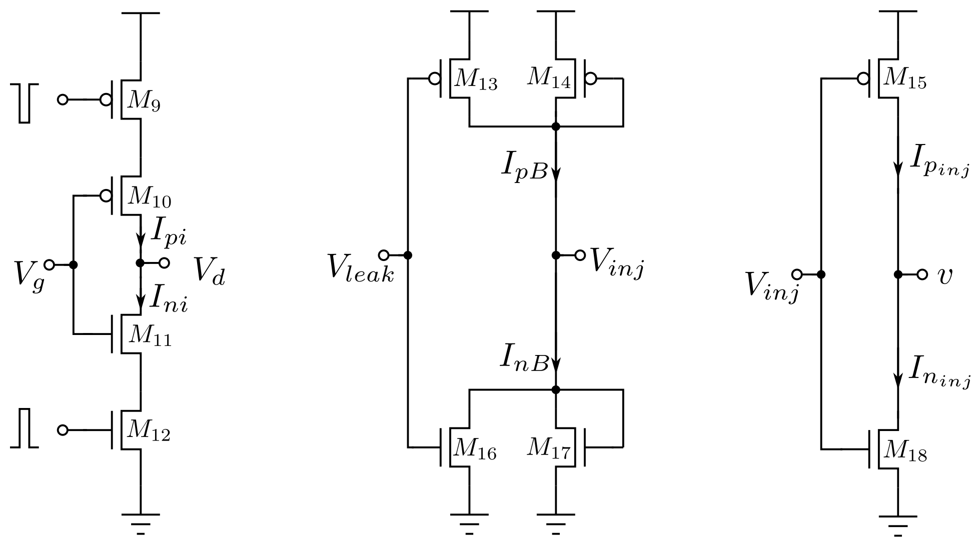


Figure 6.11: Circuits used for generating synapse meshgrids.

process is carried out for the PMOS synapse currents I_{pi}. The process of finding these quantities are done using functions where I will plug in input values and the function will return output value. In this case, the function takes the form of meshgrids or array table. The necessary meshgirds needed for the synapse circuit are given in Eq. 6.5 which are extracted using circuits in Fig. 6.11. First, the meshgrid for a single synapse current is generated. There are two components of a synapse current, I_{pi} and I_{ni}. Moreover, each current will depend on whether the neuron is active or not. Hence, each current will have two meshgrid, one for active synapse and other for inactive synapse. For inactive synapse, there will be leakage current which cannot be ignored. Hence, inactive synapse current meshgrid also needs to be generated. The circuit used for meshgrid generation of I_{pi} and I_{ni} is shown in Fig. 6.11(a). For active synapse gate voltages of M_9 and M_{12} are pulled down and pulled up respectively and vice versa for inactive synapse. The synapse currents depend of the gate voltage V_g and drain voltage V_d. Both the meshgrid of I_{pi} and I_{ni} are obtained by a single parametric sweep simulation of by sweeping

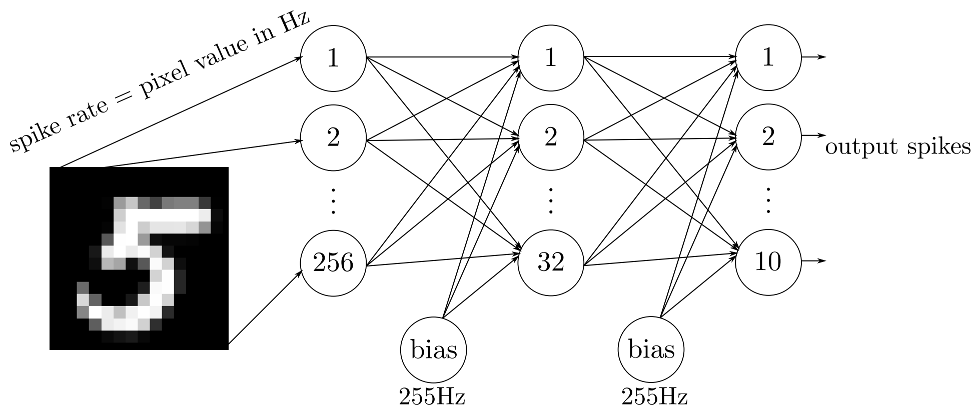


Figure 6.12: Neural network topology

V_g and V_d. Meshgrids for I_{pB} and I_{nB} are generated using circuit of Fig. 6.11(b). Both meshgrids are obtained by a single parametric sweep simulation by sweeping V_{leak} and V_{inj}. Similarly, using circuit in Fig. 6.11(c), meshgrids for $I_{p_{inj}}$ and $I_{n_{inj}}$ are obtained by sweeping V_{inj} and v.

$$I_{ni\_active} = F_{grid\_I_{ni}\_active}(V_{fg}, V_d) \tag{6.5a}$$

$$I_{pi\_active} = F_{grid\_I_{pi}\_active}(V_{fg}, V_d) \tag{6.5b}$$

$$I_{ni\_inactive} = F_{grid\_I_{ni}\_inactive}(V_{fg}, V_d) \tag{6.5c}$$

$$I_{pi\_inactive} = F_{grid\_I_{pi}\_inactive}(V_{fg}, V_d) \tag{6.5d}$$

$$I_{pB} = F_{grid\_I_{pB}}(V_{leak}, V_{inj}) \tag{6.5e}$$

$$I_{nB} = F_{grid\_I_{nB}}(V_{leak}, V_{inj}) \tag{6.5f}$$

$$I_{p_{inj}} = F_{grid\_I_{p_{inj}}}(V_{inj}, v) \tag{6.5g}$$

$$I_{n_{inj}} = F_{grid\_I_{n_{inj}}}(V_{inj}, v) \tag{6.5h}$$

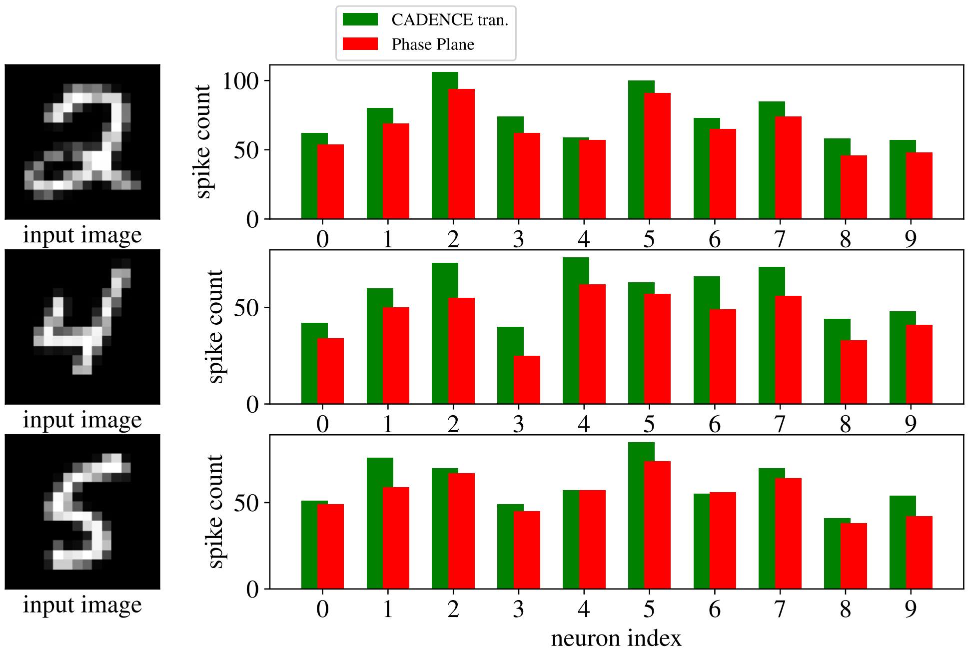


Figure 6.13: Classification results from phase plane simulation and Cadence spectre transient simulation of the network. Three examples are shown. Spike counts at the network output are closely reproduced in the phase plane simulation. Variation of the spike count at the network output are also reproduced.

6.4.3 Network Simulation for classification

I have used the Brian2 [62] simulator to simulate the neural network for a classification task shown in Fig. 6.12. The dataset for handwritten digit recognition MNIST [56] is chosen for the classification task. The dataset has 60,000 images as training set and 10,000 images as testing set.The network has one hidden layer before the output layer. Input images are resized to 16×16. Input spikes are supplied as spike trains with pixel value as the spike rate in Hz. Thus the maximum spike rate for a pixel is 255Hz. As in a deep neural network, there are bias inputs which are set at a constant 255Hz. The weights and biases are determined from a gradient descent based deep neural network training of the same network. The weights are then converted to floating gate voltages. With those weights and biases some of the inference results are shown

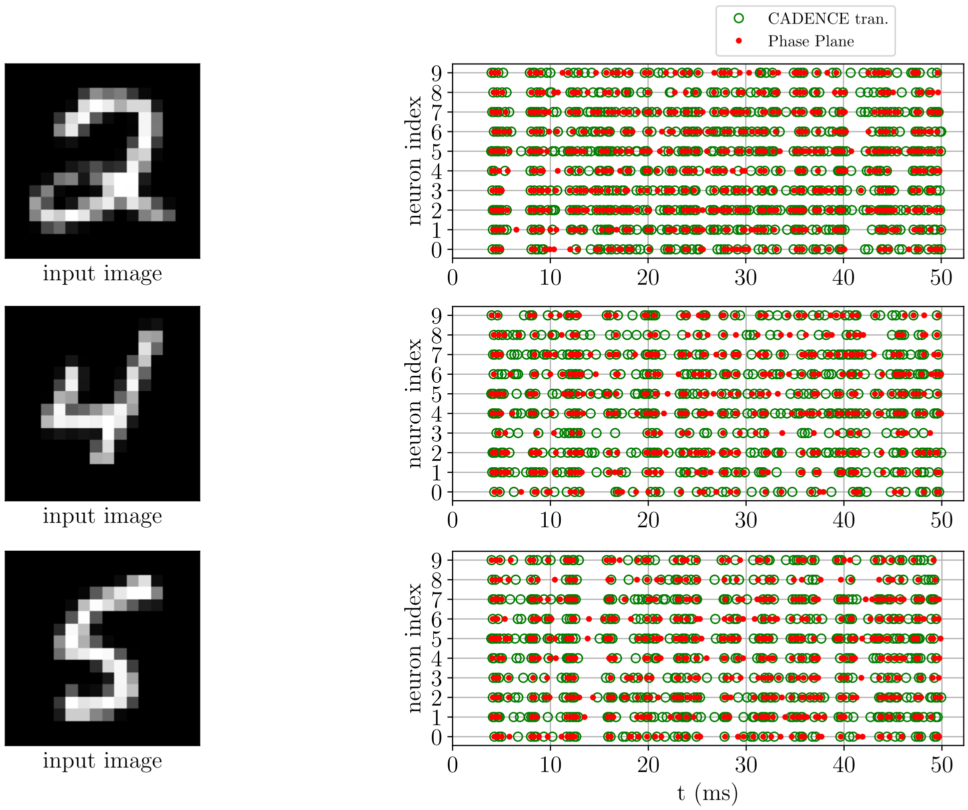


Figure 6.14: Spike timing in classification from phase plane simulation and Cadence spectre transient simulation of the network. Three examples are shown. Spike timing and spike clusters are closely reproduced.

in Fig. 6.13 and Fig 6.14. Spike timing and spike count at the network output from the phase plane simulation are compared with Cadence spectre transient simulation. Spike timings are compared in a raster plot a point is drawn at spike time for the corresponding neuron. Spike count is compared with a bar plot. It can be seen that spike timings are closely reproduced in the phase plane simulation. There are few spikes from Cadence spectre simulation that are not present in phase plane simulation and vice versa. This is because of the precision of the floating gate voltages in phase plane simulation. Floating gate voltages in phase plane simulation comes from a meshgrid. Hence, if a value falls in between the parametric sweep values, closest value is used. Also, there

Table 6.1: Speed Comparison for a 50ms of Network Simulation

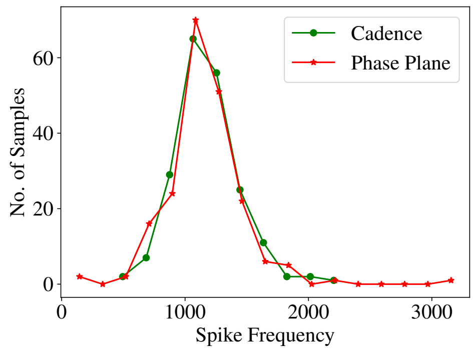
Phase Plane	Cadence Spectre
3.25 minutes	8-11 hours

are some transient effects which contributes to the difference of spike counts. Although the spike count is different at the output, the overall variation of the spike count from neuron to neuron is captured by the phase plane simulation. Hence, the classification result from the phase plane simulation can be taken as representative of BSIM circuit model based simulation result. Moreover, the phase plane simulation takes only a fraction of the time taken by a Cadence transient simulation. As shown in Table 6.1, a typical Cadence spectre transient simulation of the network shown in Fig. 6.12, takes around 8 to 11 hours to simulate 50ms of inference duration on a Red Hat desktop with 8 core CPU and 32GB of ram. Whereas, it takes only about three minutes to simulate the same network for the same duration of inference time on the same desktop to obtain similar spike counts. With the use of GPU, the simulation time can be further reduced. Brian2 simulator team has recently introduced GPU enhanced spiking neural network simulator [68] which is claimed to be 400 times faster than single CPU simulation. However, at the time of this writing, the GPU enhanced simulator does not support some features of Brian2 which have been used in phase plane simulation. Hence, GPU enhanced simulation time could not be reported. With the use of GPU, the simulation time can be reduced to milliseconds which will make it possible to learn network weights in presence of hardware nonidealities. Thus, those weights can be directly transferred to a fabricated chip.

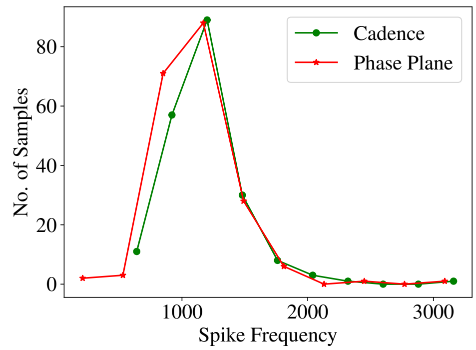
While the phase plane simulation does not replace the transistor level simulation, it can speed up debugging process of the network and estimation of network classification accuracy. In a real hardware network, signal propagation delay might significantly affect the classification accuracy. Brian2 simulator has the capability to include signal delays into account. For a small network as in Fig. 6.12, the spike propagation delay is negligible. Hence, it was not considered here. The effect of process variation on a classification result can be estimated by implementing a Monte-Carlo like simulation by randomly varying phase plane currents. For comparison of Monte-Carlo simulation in phase plane with Cadence, I consider the variation of spiking rate of the neuron circuit with a sample size of 200. Fig. 6.15 displays the results. For this simulation I have first calculated the standard deviation of a transistor current through Monte-Carlo simulation in cadence. Then I applied the standard deviation in the phase plane Monte-Carlo simulation. In order to verify the efficacy of the method, I have first included variation of all the devices in the neuron. Then excluded a single device of the neuron from applying variation. In every case the Monte-Carlo histogram closely matched the result from the phase plane method. This analysis shows that the phase plane method simulation can capture the process variation as well.

6.5 Conclusion

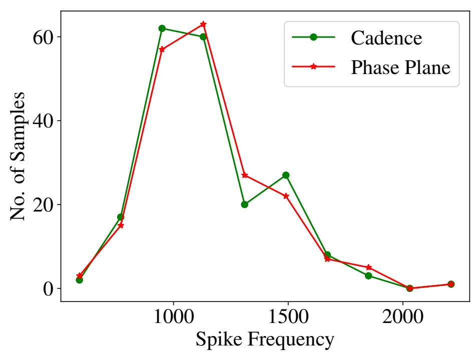
In this chapter, I have presented a method to incorporate hardware BSIM model into simulation of a neuron circuit and neural network with synapse circuits. I have used dynamical system phase plane analysis to aid us with solving circuit differential equation and synapse differential equation. I have integrated the process with an existing spiking neural network simulator. This makes it a relatively easy process to integrate hardware non-idealities into account in analog spiking neural network simulation. I have shown that the network output simulated with the phase plane method, closely follows the output of the network simulated in Cadence spectre. Moreover, phase plane simulation provides a large



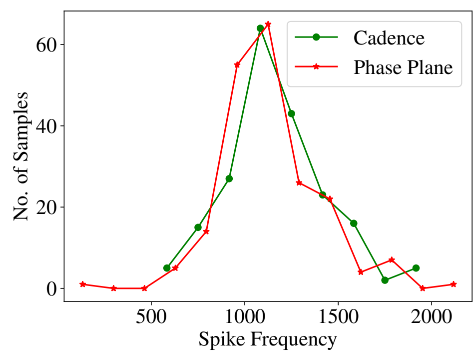
(a)



(b)



(c)



(d)

Figure 6.15: Comparison of Monte-Carlo simulation on neuron spiking frequency. Histogram results are obtained by applying process variation to devices as: (a) Including all devices (b) Excluding M_8 (c) Excluding M_6 (d) Excluding M_7.

time advantage, 160 times faster in our example, over the Cadence simulation that can be used to speed up the spiking neural network design process.

Chapter 7

Conclusion

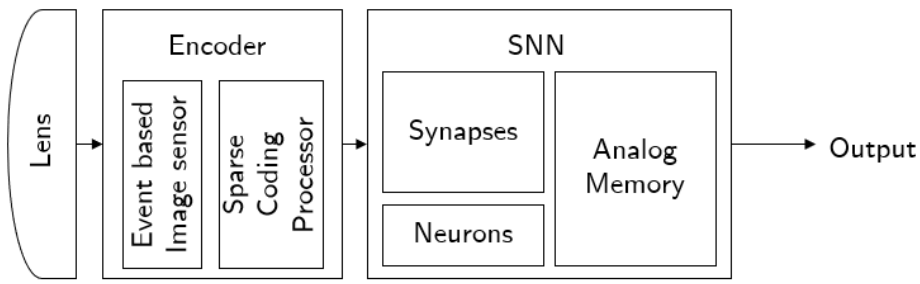


Figure 7.1: A neuromorphic image sensor processing pipeline.

In conclusion, this dissertation has presented circuit design and algorithmic techniques to minimize circuit components and power dissipation, which can be applied to machine learning on the edge. Edge systems necessitate small-scale systems with low power dissipation. Neuromorphic hardware has the promise of providing low power, compact systems that can function in the presence of noise. This dissertation has mainly focused on the hardware implementation of a neuromorphic spiking neural system. Fig. 7.1 shows an example neuromorphic image sensor with on board image processor. In this dissertation I have addressed the energy and area efficiency of sparse coding processor, neuron and synapses in the spiking neural network processor and simulation technique of the spiking neural network processor. It is shown that hardware complexity can be decreased by optimizing learning algorithms such as sparse coding. A hardware designer who wants to implement sparse coding on a chip will benefit from the increased area efficiency of the proposed algorithm. A compact, low power spiking neuron circuit is presented. A synaptic array circuit is also presented with the mitigation of leakage current. A fully connected spiking neural network can be implemented with the proposed neurons and synapses. In order to pave the way for circuit simulation with spiking neural networks for custom circuits, a phase plane method of simulation is presented which can reliably account for the hardware non-idealities. More research and experiments are required both in algorithms and hardware in order to make the neuromorphic edge machine learning competitive with the digital edge machine learning. There is signal communication complexity associated with implementing convolutional neural networks. The efficiency of the digital hardware can be utilized for the communication of signals, whereas the analog circuit can be utilized for computation to take advantage of energy efficiency. Hence, an efficient neuromorphic processor typically consists of both digital and analog circuits. The neurons and synapse circuits presented here improve the energy efficiency of the analog compute domain.

As a closing thought, I would like to express my own point of view as a researcher. As electrical engineers, we need to utilize the findings of neuroscience and biological research. Biological systems have optimized themselves over the course of millions of years. Their system is robust to noise. They consume a small amount of energy to make intelligent decisions. The problems we want to solve as engineers, the chances are very high that a biological system has already solved them with far greater efficiency than we could. One example is the information encoding system as binary number representation vs. population coding. The binary number system is the foundation of modern digital computation. Using an analog to digital converter, an analog value is converted to a binary number.

Population coding can be treated as the equivalent of binary encoding. Larger analog values require more bits to represent them. Similarly, larger analog inputs recruit more neurons for representation. Biological systems are thought to make internal models of the world based on the small amounts of information they receive through vision, hearing or touch. It recreates the outside world in the brain. Our electronic screens also recreate images based on the binary encoding. Overall, biological systems do similar things but with different computational approaches.

Biological systems are subject to stringent resource constraints. This can be attributed to why the nervous system can solve problems efficiently. I think when trying to solve a problem, biological systems resort to a fundamental computational principle without which artificial intelligence cannot move forward. This, I think, is the reason why the blackbox model of the neural network is difficult to interpret. In A.6, I point out there may exist other techniques to perform supervised learning where gradient descent fails. There are still dark areas in the realm of intelligent computation where light needs to be shed. I am hoping that these discoveries will eventually make neurmorphic systems as ubiquitous as digital systems are today.

Bibliography

Bibliography

[1] Dominique Muller Mathias De Roo, Paul Klauser. Ltp promotes a selective long-term stabilization and clustering of dendritic spines. *PLOS Biology*, 2008. xii, 8

[2] M. Munir Hasan and Jeremy Holleman. Low power compact analog spiking neuron circuit using exponential positive feedback with adaptation and bursting capability. In *2020 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2020. xv, 66, 67

[3] Rahul Sarpeshkar. Analog versus digital: Extrapolating from electronics to neurobiology. *Neural Computation*, 10(7):1601–1638, October 1998. 1, 6

[4] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R. Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V. Arthur, Paul A. Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, May 2014. 1, 46

[5] Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in Neuroscience*, 9, April 2015. 1, 46

[6] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):106–122, February 2018. 1

[7] Don Monroe. Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, 57(6):13–15, June 2014. 2

[8] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, December 1997. 2

[9] Donald Hebb. The organization of behavior. emphnew york, 1949. 2

[10] Guo qiang Bi and Mu ming Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of Neuroscience*, 18(24):10464–10472, December 1998. 2

[11] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. 2, 17, 108

[12] Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. Benchmarking tinyml systems: Challenges and direction, 2020. 2

[13] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. 111:47–63, March 2019. 4

[14] Joel Zylberberg, Jason Timothy Murphy, and Michael Robert DeWeese. A sparse coding model with synaptically local plasticity and spiking neurons can account for the diverse shapes of v1 simple cell receptive fields. *PLoS Computational Biology*, 7(10):e1002250, October 2011. 4, 20, 23, 25, 26

[15] ©2020 IEEE. Reprinted, with permission, from Md Munir Hasan and Jeremy Holleman. Spiking sparse coding algorithm with reduced inhibitory feedback weights. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1040–1043. IEEE, August 2020. 4, 5

[16] ©2020 IEEE. Reprinted, with permission, from Md Munir Hasan and Jeremy Holleman. Low power compact analog spiking neuron circuit using exponential positive feedback with adaptation and bursting capability. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 452–455. IEEE, August 2020. 4, 5

[17] ©2020 IEEE. Reprinted, with permission, from Md Munir Hasan and Jeremy Holleman. Leakage current compensation in large number of inactive synapses in a 130nm CMOS process. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 460–463. IEEE, August 2020. 5

[18] ©2021 IEEE. Reprinted, with permission, from Md Munir Hasan and Jeremy Holleman. Hardware model based simulation of spiking neuron using phase plane. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, May 2021. 5

[19] John Von Neumann. *The computer and the brain.* Yale Nota Bene. Yale University Press, New Haven, CT, 2 edition, September 2000. 6

[20] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, October 1950. 6

[21] Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. 6, 20

[22] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. 11

[23] E.M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003. 14

[24] Charles G. Gross. Genealogy of the "grandmother cell". *The Neuroscientist*, 8(5):512–518, 2002. 15

[25] Wulfram Gerstner, Andreas K. Kreiter, Henry Markram, and Andreas V. M. Herz. Neural codes: Firing rates and beyond. *Proceedings of the National Academy of Sciences*, 94(24):12740–12741, November 1997. 15

[26] S. M. Chase. Spike-timing codes enhance the representation of multiple simultaneous sound-localization cues in the inferior colliculus. *Journal of Neuroscience*, 26(15):3889–3898, April 2006. 16

[27] Timothy P. Lillicrap, Daniel Cownden, Douglas Blair Tweed, and Colin J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7, 2016. 17

[28] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. 11, December 2017. 18

[29] Eric Hunsberger and Chris Eliasmith. Training spiking deep networks for neuromorphic hardware. *arXiv:1611.05141*, 2016. 18

[30] Jason K. Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. Training spiking neural networks using lessons from deep learning, 2021. 18

[31] Youngeun Kim and Priyadarshini Panda. Revisiting batch normalization for training low-latency deep spiking neural networks from scratch, 2020. 18

[32] Peter U. Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9, August 2015. 19, 58

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. 20, 29

[34] Bruno A. Olshausen and David J. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, June 1996. 20, 21, 25

[35] Martin Rehn and Friedrich T. Sommer. A network that uses few active neurones to code visual input predicts the diverse shapes of cortical receptive fields. *Journal of Computational Neuroscience*, 22(2):135–146, October 2006. 20

[36] Christopher J. Rozell, Don H. Johnson, Richard G. Baraniuk, and Bruno A. Olshausen. Sparse coding via thresholding and local competition in neural circuits. *Neural Computation*, 20(10):2526–2563, October 2008. 20

[37] Samuel Shapero, Christopher Rozell, and Paul Hasler. Configurable hardware integrate and fire neurons for sparse approximation. *Neural Networks*, 45:134–143, September 2013. 20, 31

[38] Phil Knag, Jung Kuk Kim, Thomas Chen, and Zhengya Zhang. A sparse coding neural network ASIC with on-chip learning for feature extraction and encoding. *IEEE Journal of Solid-State Circuits*, 50(4):1070–1079, April 2015. 20, 31

[39] Dario L. Ringach. Spatial structure and symmetry of simple-cell receptive fields in macaque primary visual cortex. *Journal of Neurophysiology*, 88(1):455–463, July 2002. 25

[40] M-E. Nilsback and A. Zisserman. A visual vocabulary for flower classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 1447–1454, 2006. 29

[41] Paul Merolla and Kwabena Boahen. A recurrent model of orientation maps with simple and complex cells. In *in Advances in Neural Information Processing Systems*, pages 995–1002. MIT Press, 2004. 33

[42] Rock Z. Shi and Timothy K. Horiuchi. A neuromorphic VLSI model of bat interaural level difference processing for azimuthal echolocation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(1):74–88, January 2007. 33

[43] G. Indiveri, E. Chicca, and R. Douglas. A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17(1):211–221, January 2006. 33, 41, 42, 47, 58

[44] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, August 2014. 33, 46

[45] John V. Arthur and Kwabena A. Boahen. Silicon-neuron design: A dynamical systems approach. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(5):1034–1043, May 2011. 33, 41, 42, 45, 59

[46] Jayawan H.B. Wijekoon and Piotr Dudek. Compact silicon neuron circuit with spiking and bursting behaviour. *Neural Networks*, 21(2-3):524–534, March 2008. 33, 41, 42

[47] C. M. Gray and D. A. McCormick. Chattering cells: Superficial pyramidal neurons contributing to the generation of synchronous oscillations in the visual cortex. *Science*, 274(5284):109–113, October 1996. 37

[48] A. van Schaik. Building blocks for electronic spiking neural networks. *Neural Networks*, 14(6-7):617–628, July 2001. 40

[49] Ilias Sourikopoulos, Sara Hedayat, Christophe Loyez, François Danneville, Virginie Hoel, Eric Mercier, and Alain Cappy. A 4-fJ/spike artificial neuron in 65 nm CMOS technology. *Frontiers in Neuroscience*, 11, March 2017. 42, 47

[50] Iulia Alexandra Lungu, Shih-Chii Liu, and Tobi Delbruck. Incremental learning of hand symbols using event-based cameras. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(4):690–696, December 2019. 45, 58

[51] J.V. Arthur and K. Boahen. Recurrently connected silicon neurons with active dendrites for one-shot learning. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*. IEEE. 45

[52] Christian Mayr, Johannes Partzsch, Marko Noack, Stefan Hanzsche, Stefan Scholze, Sebastian Hoppner, Georg Ellguth, and Rene Schuffny. A biological-realtime neuromorphic system in 28 nm CMOS using low-leakage switched capacitor circuits. *IEEE Transactions on Biomedical Circuits and Systems*, 10(1):243–254, February 2016. 46

[53] J. M. Cruz-Albrecht, M. W. Yung, and N. Srinivasa. Energy-efficient neuron, synapse and STDP integrated circuits. *IEEE Transactions on Biomedical Circuits and Systems*, 6(3):246–256, June 2012. 46

[54] Johannes Schemmel, Daniel Briiderle, Andreas Griibl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, May 2010. 46

[55] M. Munir Hasan and Jeremy Holleman. Implementation of linear discriminant classifier in 130nm silicon process. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018. 49, 73

[56] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. 51, 73, 78

[57] Kareem A Zaghloul and Kwabena Boahen. A silicon retina that reproduces signals in the optic nerve. *Journal of Neural Engineering*, 3(4):257–267, September 2006. 58

[58] E. Chicca, D. Badoni, V. Dante, M. D. Andreagiovanni, G. Salina, L. Carota, S. Fusi, and P. Del Giudice. A vlsi recurrent network of integrate-and-fire neurons connected by plastic synapses with long-term memory. *IEEE Transactions on Neural Networks*, 14(5):1297–1307, September 2003. 58

[59] David Bolme, Aravind Mikkilineni, Derek Rose, Srikanth Yoginath, Mohsen Judy, and Jeremy Holleman. Deep modeling: Circuit characterization using theory based models in a data driven framework. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2017. 58

[60] Peiran Gao, Ben V. Benjamin, and Kwabena Boahen. Dynamical system guided mapping of quantitative neuronal models onto neuromorphic hardware. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(10):2383–2394, October 2012. 58

[61] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398, July 2007. 59

[62] Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, August 2019. 59, 69, 78

[63] B.J. Sheu, D.L. Scharfetter, P.-K. Ko, and M.-C. Jeng. BSIM: Berkeley short-channel IGFET model for MOS transistors. *IEEE Journal of Solid-State Circuits*, 22(4):558–566, August 1987. 59

[64] Arindam Basu and Paul E. Hasler. Nullcline-based design of a silicon neuron. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(11):2938–2947, November 2010. 59

[65] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, 1(6):445–466, July 1961. 60

[66] Skillbridge, a seamless python to cadence virtuoso skill interface. 68

[67] P. Hasler and J. Dugger. An analog floating-gate node for supervised learning. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(5):834–845, May 2005. 73

[68] Marcel Stimberg, Dan F. M. Goodman, and Thomas Nowotny. Brian2genn: accelerating spiking neural network simulations with graphics hardware. *Scientific Reports*, 10(1), January 2020. 80

[69] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980. 99

[70] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, March 1968. 99

[71] Rahul Sarpeshkar. *Ultra Low Power Bioelectronics*, pages 42–43. Cambridge University Press, 2009. 100, 101

[72] Shih-Chii Liu, Jörg Kramer, Giacomo Indiveri, Tobias Delbrück, and Rodney Douglas. *Analog VLSI: Circuits and Principles*, pages 126–127. The MIT Press, 2002. 101

[73] Karl Åström. *Feedback Systems : an introduction for scientists and engineers, second edition*, page 268. Princeton University Press, 2008. 104

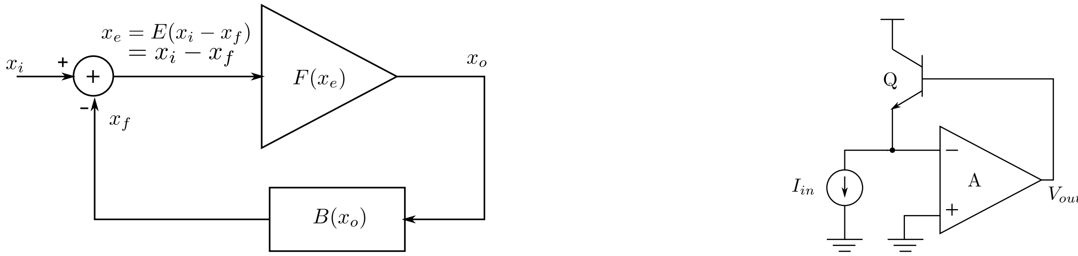
Appendices

Appendix A

Supervised Learning as Negative Feedback

A.1 Introduction

Gradient descent has long been the dominant method for optimizing weights in neural networks. It is constructed purely from a mathematical point of view with the goal to minimize a loss function. Many mathematical formulations are modeled after a physical process. The most relevant example is the deep neural network which is modeled after a biological process. Having a physical process behind a mathematical model has the advantage that the behavior of the physical process can provide intuition for the mathematical model. For example, the convolutional neural network [69], which now forms the backbone of image recognition, is inspired by the receptive field of the mammalian visual cortex [70] . Gradient descent with momentum is developed by analogy with stabilizing a heavy ball rolling down a hill. I believe that studying the physical process which describes the optimization should help us design a better optimizer. Here I present a negative feedback system as a physical analogy of optimization and show a close relationship to gradient descent. This optimization method is based on the ability of a negative feedback system to perform the inverse operation of a function. This principle is well known in the analog circuits and systems



(a) (b)

Figure A.1: (a) A generic negative feedback system (b) An Operational amplifier with an exponential element in the feedback path realizes a logarithmic input-output function. The transistor Q has exponential voltage to current relationship. The feedback system implements inverse of the exponential i.e. logarithmic function.

community and many useful analog circuits have been constructed [71] using this principle.

A.2 Theoretical Background

For a negative feedback system as shown in Fig. A.1a, if I define the forward function, backward function, and the error function with Eq.s (A.1), (A.2) and (A.3) respectively, then the input output relationship is expressed by Eq. (A.4). For a forward function of the form $y = F(x) = Ax$ where A is the gain, the inverse of the forward function is $x = F^{-1}(y) = y/A$. If the gain A is large then $F^{-1} \rightarrow 0$. For error function of the form $y = E(x) = ux$ where u is the gain, $E^{-1}(F^{-1}) \rightarrow 0$ for high forward gain A. Then the output of the feedback system becomes inverse of the backward function as in Eq. (A.7). Effectively, the negative feedback system is implementing the inverse of the function that is in the backward path.

$$x_o = F(x_e) \quad \text{(forward function)} \tag{A.1}$$

$$x_f = B(x_o) \quad \text{(backward function)} \tag{A.2}$$

$$x_e = E(x_i - x_f) \quad \text{(error function)} \tag{A.3}$$

$$F^{-1}(x_o) = E(x_i - B(x_o)) \tag{A.4}$$

$$x_i = E^{-1}(F^{-1}(x_o)) + B(x_o) \tag{A.5}$$

$$x_i \approx B(x_o) \quad (\text{for large } A,\ E^{-1}(F^{-1}(x_o)) \to 0) \tag{A.6}$$

$$x_o = B^{-1}(x_i) \tag{A.7}$$

This property is commonly used in analog circuits in order to perform inverse operation of the transistor function [71, 72]. An example circuit is shown in Fig. A.1b. In a transistor an input voltage creates an exponential output current. However, the transistor is an uni-directional device which means that pushing a current at the output of the transistor will not produce a voltage at the input. In order to make that operation work, a negative feedback system using an operational amplifier of gain A is used which implements that inverse operation. This way an input current I_{in} into the feedback system produces the corresponding transistor voltage V_{out}.

It should be noted that even if the backward function B is not completely invertible (which is the case for an uni-directional transistor), the overall system appears to be performing B^{-1}. This is because the system is not using x_i (the range of B) as input to the function B^{-1} directly. Rather, as in Fig. A.1a, the output of the system x_o acts as the domain of B. The output of B is then compared with the target range of B i.e. x_i. When the difference of x_i and x_f is zero, the overall system output x_o is approximately the output of B^{-1}.

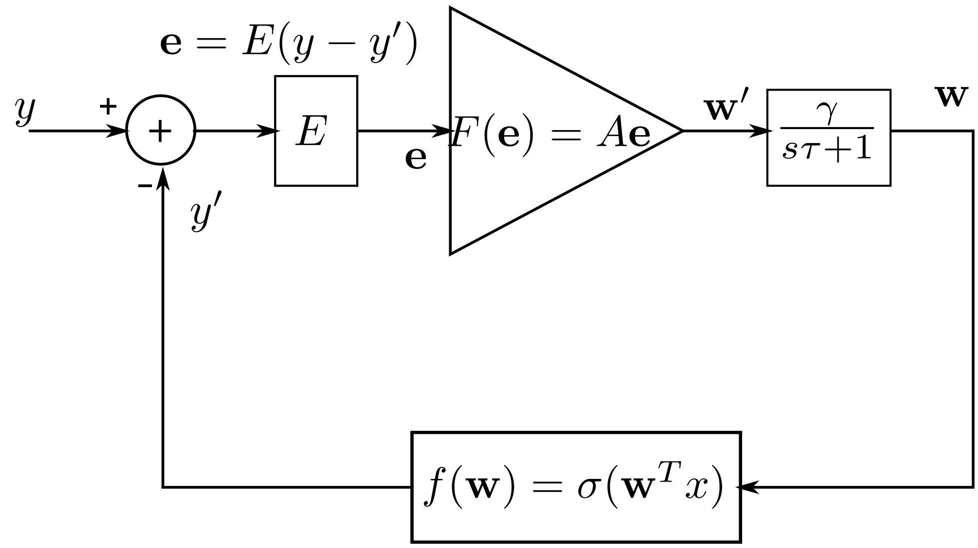


Figure A.2: A negative feedback system as optimizer for machine learning system.

A.3 Method

A.3.1 System Setup

To frame optimization as a negative feedback problem, I express the a layer as a function of the weights, with the inputs held constant. In a neural network, a single layer can be expressed as a function of a linear combination of $\mathbf{x}$ with a weight vector $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$ as shown in Eq. (A.8). There can be linear or non-linear activation function σ inside the function f. A bias term can be easily implemented by setting an element of the $\mathbf{x}$ vector to 1. The variables x_i and y are training samples which are known quantities for a problem. By implementing the inverse operation of the function in Eq. (A.9) I can find the weights w_i, which effectively implements an optimization operation.

$$y = f(\mathbf{w}) = \sigma(\sum_i w_i x_i) \tag{A.8}$$

$$\mathbf{w} = f^{-1}(y) \tag{A.9}$$

To implement the inverse operation using negative feedback, the function f is placed in the feedback path as shown in Fig. A.2, x training samples are used in the backward function, weights are initialized randomly and y training samples are set as input to the feedback system. An initial prediction of the weight vector $\mathbf{w}$ is used by the backward function to produce y'. Using the difference $y - y'$ an error $\mathbf{e}$ is generated. The process of generating a vector $\mathbf{e}$ with a scalar $y - y'$ is described in the following subsection.

A.3.2 Stability Criteria

In order for a feedback system to be stable, the bandwidth of the system should be limited, meaning that the output should change slowly (a low frequency system). Hence, instead of changing the weight from the previous value to the new value predicted by the forward function instantly (infinite bandwidth), a small increment is made from the previous value toward the predicted value by using a first order low pass filter as shown below.

$$\begin{aligned}
\frac{\mathbf{w}}{\mathbf{w}'} &= \frac{\gamma}{s\tau + 1} \quad \text{(Laplace transformed low pass filter transfer function)} \\
\tau\frac{\partial \mathbf{w}}{\partial t} &= -\mathbf{w} + \gamma\mathbf{w}' \\
\mathbf{w}^t &= \mathbf{w}^{t-1} + (\gamma\mathbf{w}' - \mathbf{w}^{t-1})\frac{\partial t}{\tau} = \mathbf{w}^{t-1} + (A\gamma\mathbf{e} - \mathbf{w}^{t-1})\eta
\end{aligned} \tag{A.10}$$

This is similar to using a small learning rate in gradient descent. The prediction labeled $\mathbf{w}'$ from the forward function goes into a low pass filter characterised by a time constant τ and arbitrary constant γ which outputs slowly varying $\mathbf{w}$. This new value of $\mathbf{w}$ goes around the feedback loop again and with consecutive iterations around the feedback loop, the output converges to the optimum value of $\mathbf{w}$. The weight update method because of the low pass filter is given in Eq. (A.10) where the quantity $\eta = \partial t/\tau$ acts as the learning rate. The superscript t denotes the weight at time t during iteration.

Another important criterion for stability is that the gain around the feedback loop must be negative when the magnitude is greater than unity [73]. From Fig. A.2, the forward gain is A and the backward gain is $\boldsymbol{\beta} = \partial y'/\partial \mathbf{w}$. The loop gain of the system is $-1 \times A\boldsymbol{\beta}$. Hence, I have to make sure that the product of the forward and backward gain for each component of $\boldsymbol{\beta}$ is always positive. The forward gain A is typically positive. If any component of $\boldsymbol{\beta}$ is negative for a training sample then the corresponding element of the gain product becomes negative. In general, if I use a forward gain of $A\boldsymbol{\beta}$, then the element-wise product of forward and backward gain is $A\boldsymbol{\beta}\times\boldsymbol{\beta} = A\boldsymbol{\beta}^2$ which is guarantied to be positive. With $A\boldsymbol{\beta}$ as the forward gain, the forward function can now take scalar error $y-y'$ and produce vector $\mathbf{w}'$ as shown below.

$$\mathbf{w}' = A\boldsymbol{\beta} \times (y - y') = A \times \boldsymbol{\beta}(y - y') \tag{A.11}$$

In (A.11), I can separate $\boldsymbol{\beta}$ from the forward gain and attach it to $y - y'$. This way I can keep using a forward gain of A and use $\mathbf{e} = \boldsymbol{\beta}(y - y')$ as the new error. The error is now a function of scalar $y - y'$ which is shown by an error function block E in Fig. A.2. I also notice that the error function is of the form $e = E(x) = ux$ as assumed in section A.2. The error is calculated by multiplying the difference $y - y'$ with backward gain $\boldsymbol{\beta}$. Thus the gain of the error function is $\mathbf{u} = \boldsymbol{\beta}$.

A.4 Application in Machine Learning

In the following sections, I apply this method starting with simpler regression problems and then gradually develop methods for complex problems such as deep neural networks.

A.4.1 Regression

In machine learning, the activation functions can be unity, ReLU, tanh etc. The backward gain of the feedback system for any activation function is $\boldsymbol{\beta} = \partial y'/\partial \mathbf{w} = \sigma' \mathbf{x}$ where σ' is the gain of the activation function. For a single training sample, the error corresponding to i^{th} weight is $e_{w_i} = u_i(y - y') = \sigma'_i x_i (y - y')$. With many training samples the error is the sum of the errors from all the samples. The error for all the weights can be expressed as matrix multiplication, as in Eq. (A.12), where $u_{w_i}^{[k]} = \sigma_i'^{[k]} x_i^{[k]}$ is the error gain for i^{th} weight and k^{th} sample. For all the training samples the error function gain becomes a matrix $\mathbf{U}$.

$$\mathbf{e} = E(y - y') = \mathbf{U}(\mathbf{y} - \mathbf{y}')^T = \begin{bmatrix} u_{w_1}^{[1]} & u_{w_1}^{[2]} & \dots & u_{w_1}^{[m]} \\ u_{w_2}^{[1]} & u_{w_2}^{[2]} & \dots & u_{w_2}^{[m]} \\ \vdots & \vdots & \ddots & \vdots \\ u_{w_n}^{[1]} & u_{w_n}^{[2]} & \dots & u_{w_n}^{[m]} \end{bmatrix} \begin{bmatrix} y^{[1]} - y'^{[1]} \\ y^{[2]} - y'^{[2]} \\ \vdots \\ y^{[m]} - y'^{[m]} \end{bmatrix} = \begin{bmatrix} e_{w_1} \\ e_{w_2} \\ \vdots \\ e_{w_n} \end{bmatrix} \tag{A.12}$$

A.4.2 Single Layer Classifier

The regression problem can be turned into a perceptron classifier by using softmax or tanh as the activation function. Hence, Eq. (A.12) also represents the error

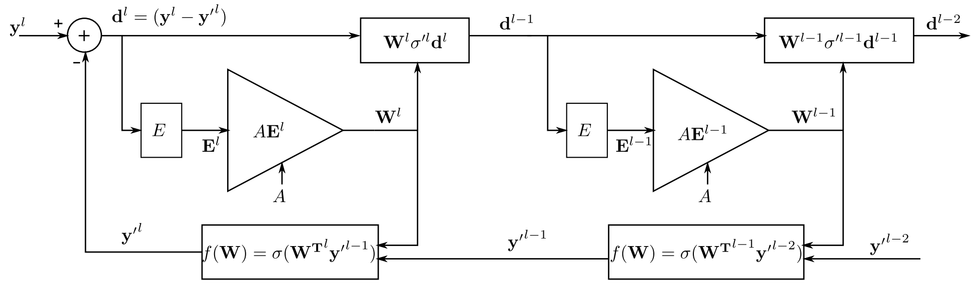


Figure A.3: Backpropagating the difference vector to previous layers.

function for a single layer perceptorn. For a multi class classifier the error function is simply the extension of Eq. (A.12). The single row of $\mathbf{y} - \mathbf{y}'$ becomes a matrix with $y - y'$ of different classes stacked as rows.

A.4.3 Deep Network

To use this system in deep networks, a method for error backpropagation is needed. A network is shown in Fig. A.3 with l denoting layer number. The low pass filters haven been omitted in the figure for simplicity. The input from second to last layer $\mathbf{y}'^{l-2}$ generates the final output $\mathbf{y}'^{l}$. I treat the output $\mathbf{y}'^{l}$ as a result of the input $\mathbf{y}'^{l-2}$ as in Eq. (A.13). For c^{th} class output it can be expressed as Eq. (A.14). The backward gain for a weight is given by Eq. (A.15). Multiplying the difference $d_c^l = (y_c^l - y_c'^l)$ with the backward gain, I can write the error for

$e_{w_{ji}^{l-1}}$ as Eq. (A.16).

$$\mathbf{y'}^{l} = \sigma(\mathbf{W^T}^{l}\sigma(\mathbf{W^T}^{l-1}\mathbf{y'}^{l-2})) \tag{A.13}$$

$$y_c^l = \sigma(\sum_i w_{ic}^l(\sigma(\sum_j w_{ji}^{l-1}y_j^{l-2}))_i) \tag{A.14}$$

$$\beta_{w_{ji,c}^{l-1}} = \sigma'^{l}w_{ic}^{l}\sigma'^{l-1}y_j^{l-2} \tag{A.15}$$

$$e_{w_{ji}^{l-1}} = \sigma'^{l-1}y_j^{l-2}\sum_c \sigma'^{l}w_{ic}^{l}d_c^{l} \tag{A.16}$$

$$e_{w_{ji}^{l-1}} = \sigma'^{l-1}y_j^{l-2}d_i^{l-1} = u_j^{l-1}d_i^{l-1} \tag{A.17}$$

The sum over c expresses the fact that every class output is influenced by w_{ji}^{l-1}. With $d_i^{l-1} = \sum_c \sigma'^{l}w_{ic}^{l}d_c^{l}$ in Eq. (A.17), d_i^{l-1} can be thought of as the difference error for layer $l-1$. Also, u_j^{l-1} represents the error function gain. The outcome is shown in Fig. A.3. The difference vector of the last layer is multiplied with $\sigma'^{l}\mathbf{W}^{l}$ which produces the difference vector for the previous layer. This way error is backpropagated to all the previous layers.

A.5 Comparison with Gradient Descent

For a negative feedback system it is important that the forward and backward gain product for each weight is positive. The gain of the error function as $\mathbf{u} = \boldsymbol{\beta}$ satisfies that condition. In fact I can use $\mathbf{u} = \boldsymbol{\beta}^n$ as the gain as well where n is an odd positive integer. This way the negative feedback system represents an infinite number of optimizers. The reason for odd positive n is that it preserves the sign of $\boldsymbol{\beta}$. When $n = 1$, the negative feedback system error implements the error gradient of the gradient descent optimization method. The gradient descent method minimizes a loss function, e.g. squared error as in Eq. (A.18). The weight parameters are updated by going in the opposite direction of the gradient which is given by Eq. (A.19) for a weight w_i. Using $\mathbf{u} = \boldsymbol{\beta}$ in Eq. (A.12), the feedback error for a weight w_i is given by Eq. (A.20). I see that both expressions are same except for a factor of $2/m$. The relationship between the two is $e_{w_i} = (m/2)(-\nabla q_{w_i})$. In gradient descent with a weight decay factor λ, the update rule is given by $w^t = w^{t-1} - \eta(\nabla q_{w_i} + \lambda w^{t-1})$. If I let $\eta \leftarrow \eta\lambda$, $\gamma \leftarrow 2/(Am\lambda)$ and substitute $e_{w_i} = (m/2)(-\nabla q_{w_i})$ in Eq. (A.10) I get Eq. (A.21) which is exactly the same as gradient descent update rule.

$$q = \frac{1}{m}\sum_k (y^{[k]} - y'^{[k]})^2 \tag{A.18}$$

$$-\nabla q_{w_i} = \frac{2}{m}\sum_k (y^{[k]} - y'^{[k]}).\sigma'.x_i \tag{A.19}$$

$$e_{w_i} = \sum_k (y^{[k]} - y'^{[k]}).\sigma'.x_i \tag{A.20}$$

$$w_i^t = w_i^{t-1} - \eta(\nabla q_{w_i} + \lambda w_i^{t-1}) \tag{A.21}$$

At this stage I can see that with $\mathbf{u} = \boldsymbol{\beta}$ which is the condition for squared error minimization, the negative feedback system and gradient descent method are equivalent. Also, by noticing Fig. A.3, one can easily realize that the error propagation to previous layers is the same as the backpropagation technique in gradient descent method [11]. I have derived it only using the properties of the negative feedback system. Thus, the negative feedback system allows us to look at and analyze the optimization problem from a different perspective. In gradient descent the objective is to minimize a loss function. However, in negative feedback system, the objective is inverse the backward function.

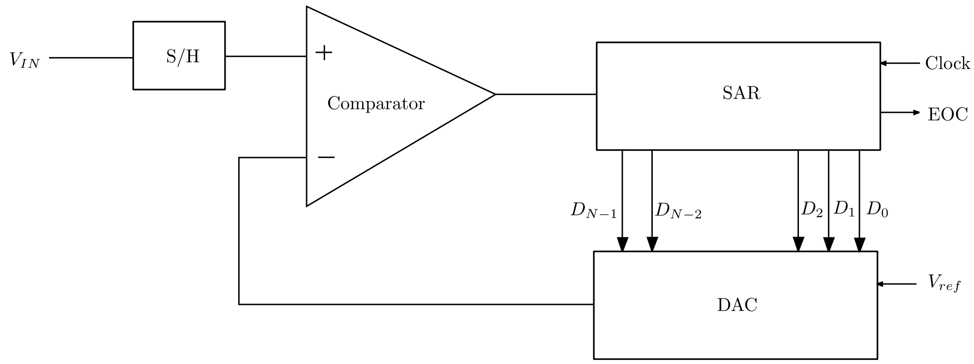


Figure A.4: SAR ADC as supervised learning system.

A.6 Analog to Digital Converter as Supervised Learning System

At this point, it can be established that negative feedback is the computational principle for supervised learning. From the mathematical analysis, it is also seen that the gradient of the backward function is needed in order to establish the error function. However, a counter example can be given where gradient of the backward function is not required in order to establish the error function. The Successive Approximation Register (SAR) Analog to Digital Converter (ADC) is well known as a digital converter. However, when looked at closely, it is apparent that it is a supervised learning system. Fig. A.4 shows a SAR ADC system. S/H is the sample and hold block that samples the analog voltage. SAR is the control logic block that generates the output digital binary bits. DAC is the digital to analog converter block that generates analog voltage with the digital bits and a reference voltage V_{ref} as input. The comparator compares the input and feedback voltage. The comparator outputs a binary error signal. Depending on the error signal, the SAR block implements a binary search algorithm to generate a guess of digital output. The DAC then converts the digital output back to an analog value, which is compared with the original analog value. Once all the bits have been generated, SAR outputs an End of Conversion (EOC) signal.

$$M = D_{N-1}x^{N-1} + D_{N-1}x^{N-1} + \cdots + D_2x^2 + D_1x^1 + D_0x^0 \tag{A.22}$$

Overall, the entire system is a negative feedback system that tries to learn the binary representation of the given input. DAC functions as the backward function. SAR functions as the error function and high gain block. The binary to decimal conversion is given by (A.22) where N is the number of bits, $x = 2$ and M is the analog decimal. $[x^{N-1}, x^{N-2}, \cdots, x^1, x^0]$ is the input to the backward function, M is the teacher signal and $[D_{N-1}, D_{N-2}, \cdots, D_1, D_0]$ is the learned weights. What in interesting is that gradient descent fails to learn the binary representation. The weights $[D_{N-1}, D_{N-2}, \cdots, D_1, D_0]$ are binary which makes (A.22) non-differentiable. However, the SAR ADC can learn binary representation by using binary search as the error function. This clearly shows that there may exist error functions other than pure mathematical expressions as given by (A.20) and still function as a negative feedback learning system. This insight is not readily obtained purely from the gradient descent point of view.

Appendix B

Codes Used in Simulation

B.1 Meshgrid Generation

Listing B.1: Neuron phase plane

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 21 21:49:04 2019

@author: mhasan13
"""


from skillbridge import Workspace
from skillbridge.client.translator import Symbol
import numpy as np
import matplotlib.pyplot as plt
import utils
import pickle as pkl

########### design variables ############
vdd = 300e-3
vk = 10e-3
```

```
vr = 100e-3
vth = 50e-3
vw = 80e-3
vss = 0
########## sweep variables ##############
step = 0.1e-3
extended_zone = 0.0
#########################################


# connect to server
ws = Workspace.open()

# set simulator
ws['simulator'](Symbol('spectre'))
# set schematic
ws['design']('/tmp/simulation/neuron_0p3/spectre/schematic/netlist/netlist')
# results directory
ws['resultsDir']( '/tmp/simulation/neuron_0p3/spectre/schematic')
# set model files
ws['modelFile'](utils.model_files[0],utils.model_files[1],utils.model_files[2],utils.model_files[3],utils.model_files[4],utils.model_files[5],utils.model_files[6],utils.model_files[7],utils.model_files[8],utils.model_files[9],
                utils.model_files[10],utils.model_files[11],utils.model_files[12],utils.model_files[13],utils.model_files[14],utils.model_files[15],utils.model_files[16],utils.model_files[17],utils.model_files[18],utils.model_files[19],
```

```
                    utils.model_files[20],utils.model_files[21],utils.model_files[22],utils.model_files[23],utils.model_files[24],utils.model_files[25],utils.model_files[26],utils.model_files[27],utils.model_files[28]
                    )
# dc analysis
ws['analysis'](Symbol('dc'),'?param', 'v', '?start', vss-extended_zone,'?stop', vdd+extended_zone, '?step', step)

# set design variables
ws['desVar'](   "v", 0)
ws['desVar'](   "u", 0)
ws['desVar'](   "vdd", vdd  )
ws['desVar'](   "vk", vk  )
ws['desVar'](   "vr", vr  )
ws['desVar'](   "vth", vth  )
ws['desVar'](   "vw", vw  )
# analysis order in case of multiple analysis
ws['envOption'](Symbol('analysisOrder'), ['dc'])
# to be saved currents
ws['save']( Symbol('i'), "/pos_feed/D", "/neg_feed/D", "/width_p/D", "/refrac_n/D", "/Mk/D" )
# set temp
ws['temp'](27)
# param sweep
dummy= ws['paramAnalysis']('u', start=vss-extended_zone, stop=vdd+extended_zone, step=step) # values not string

# run
ws['paramRun']()
# skillbridge cannot parse stdobj@0xhexnumber type data. but assigning return value to a variable prevents error
dummy = ws['selectResult'](Symbol('dc'))
```

```

waves = [ws.get.data('/pos_feed/D'), ws.get.data('/neg_feed/D'),
    ws.get.data('/width_p/D'), ws.get.data('/refrac_n/D'), ws.get.
    data('/Mk/D'), ws.get.data('/axon')]
data = []
n_param = 2
for wave in waves:
     mgrid = utils.n_param_wave_to_meshgrid(ws, wave, [None for _
    in range(n_param)], n_param, n_param)
     data.append(mgrid)

utils.meshgrid_to_pickle(data, n_param, 'neuron-dense.pickle')

############ draw phase space #############
#with open ('neuron.pickle', 'rb') as fp:
#    itemlist = pkl.load(fp)
#
#data = np.array(itemlist)
#u_range = data.shape[1]
#v_range = data.shape[2]
#Cv = 50e-15
#Cu = 30e-15
#u = data[0,::10,::10]
#v = data[1,::10,::10]
## -ve sign has to be fixed for pmos currents now
## as cadence introduced a -ve sign for outgoing current
#dvdt = (1/Cv)*(-data[2,::10,::10] - data[3,::10,::10])
#dudt = (1/Cu)*(-data[4,::10,::10] - data[5,::10,::10])
#
#r = np.sqrt(dvdt**2 + dudt**2)
#dvdt = dvdt / r
#dudt = dudt / r
#
```

```
#fig = plt.figure()
#ax = fig.gca()
#ax.quiver(v,u,dvdt,dudt)
```

Listing B.2: Active/Inactive synapse current phase plane using Fig. 6.11(a)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 21 21:49:04 2019

@author: mhasan13
"""


from skillbridge import Workspace
from skillbridge.client.translator import Symbol
import numpy as np
import matplotlib.pyplot as plt
import utils
import pickle as pkl

########### design variables ############
vdd = 300e-3
vss = 0
vfg = 100e-3
vref = 100e-3
v = 150e-3
spike_p = vss
spike_n = vdd
########## sweep variables ##############
step = 100e-6
extended_zone = 0.0
#########################################
```

```


# connect to server
ws = Workspace.open()

# set simulator
ws['simulator'](Symbol('spectre'))
# set schematic
ws['design']('/tmp/simulation/synapse_0p3/spectre/schematic/netlist/netlist')
# results directory
ws['resultsDir']( '/tmp/simulation/synapse_0p3/spectre/schematic')
# set model files
ws['modelFile'](utils.model_files[0],utils.model_files[1],utils.model_files[2],utils.model_files[3],utils.model_files[4],utils.model_files[5],utils.model_files[6],utils.model_files[7],utils.model_files[8],utils.model_files[9],
                utils.model_files[10],utils.model_files[11],utils.model_files[12],utils.model_files[13],utils.model_files[14],utils.model_files[15],utils.model_files[16],utils.model_files[17],utils.model_files[18],utils.model_files[19],
                utils.model_files[20],utils.model_files[21],utils.model_files[22],utils.model_files[23],utils.model_files[24],utils.model_files[25],utils.model_files[26],utils.model_files[27],utils.model_files[28]
                )
# dc analysis
ws['analysis'](Symbol('dc'),'?param', 'v', '?start', vss-extended_zone,'?stop', vdd+extended_zone, '?step', step)

# set design variables
ws['desVar'](   "v", 0)
```

```
ws['desVar'](   "vfg", 0)
ws['desVar'](   "vdd", vdd  )
ws['desVar'](   "vref", vref  )
ws['desVar'](   "spike_p", spike_p  )
ws['desVar'](   "spike_n", spike_n)
# analysis order in case of multiple analysis
ws['envOption'](Symbol('analysisOrder'), ['dc'])
# to be saved currents
ws['save']( Symbol('i'), "/syn_p_v3/D", "/syn_n_v3/D" )
# set temp
ws['temp'](27)

dummy = ws['paramAnalysis']('vfg', Symbol('?start'), vss-extended_zone, Symbol('?stop'), vdd+extended_zone, Symbol('?step'), step) # values not string

# run
ws['paramRun']()
# skillbridge cannot parse stdobj@0xhexnumber type data.
dummy = ws['selectResult'](Symbol('dc'))


waves = [ws.get.data('/syn_p_v3/D'), ws.get.data('/syn_n_v3/D')]
data = []
n_param = 2
for wave in waves:
    mgrid = utils.n_param_wave_to_meshgrid(ws, wave, [None for _ in range(n_param)], n_param, n_param)
    data.append(mgrid)

utils.meshgrid_to_pickle(data, n_param, 'synapse-active.pickle')

####################
```

```
with open ('synapse-active.pickle', 'rb') as fp:
    itemlist_active = pkl.load(fp)


# =============================================================================
#  for non active synapse
# =============================================================================

# non spike
spike_n = vss
# set design variable
ws['desVar'](   "spike_n", spike_n)

# run again
ws['paramRun']()
# skillbridge cannot parse stdobj@0xhexnumber type data.
dummy = ws['selectResult'](Symbol('dc'))

waves = [ws.get.data('/syn_p_v3/D'), ws.get.data('/syn_n_v3/D')]
data = []
n_param = 2
for wave in waves:
    mgrid = utils.n_param_wave_to_meshgrid(ws, wave, [None for _ in range(n_param)], n_param, n_param)
    data.append(mgrid)

utils.meshgrid_to_pickle(data, n_param, 'synapse-inactive.pickle')

```

```
####################
with open ('synapse-inactive.pickle', 'rb') as fp:
    itemlist_nonactive = pkl.load(fp)
```

Listing B.3: Synapse leakage bypass phase plane using Fig. 6.11(b)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 21 21:49:04 2019

@author: mhasan13
"""


from skillbridge import Workspace
from skillbridge.client.translator import Symbol
import numpy as np
import matplotlib.pyplot as plt
import utils
import pickle as pkl

########### design variables ############
vdd = 300e-3
vss = 0
v_leak = 100e-3
idc_max = 30e-9
idc = 1e-9
########## sweep variables ##############
v_leak_step = 0.1e-3
v_inj_step = 100e-6
#########################################


```

```
# connect to server
ws = Workspace.open()

# there is a bug in IC6.18
# https://community.cadence.com/cadence_technology_forums/f/custom-ic-skill/42502/ocean-script-nested-parametric-analysis-problem
# envSetVal("spectre.envOpts" "controlMode" 'string "batch")
# the above command has to be set to get paramRun() working
ws['envSetVal']('spectre.envOpts', 'controlMode', Symbol('string'), 'batch')

# set simulator
ws['simulator'](Symbol('spectre'))
# set schematic
ws['design']('/tmp/simulation/synapse_0p3_modified/spectre/schematic/netlist/netlist')
# results directory
ws['resultsDir']( '/tmp/simulation/synapse_0p3_modified/spectre/schematic' )
# set model files
ws['modelFile'](utils.model_files[0],utils.model_files[1],utils.model_files[2],utils.model_files[3],utils.model_files[4],utils.model_files[5],utils.model_files[6],utils.model_files[7],utils.model_files[8],utils.model_files[9],
                utils.model_files[10],utils.model_files[11],utils.model_files[12],utils.model_files[13],utils.model_files[14],utils.model_files[15],utils.model_files[16],utils.model_files[17],utils.model_files[18],utils.model_files[19],
                utils.model_files[20],utils.model_files[21],utils.model_files[22],utils.model_files[23],utils.model_files[24],utils.model_files[25],utils.model_files[26],utils.model_files[27],utils.model_files[28]
```

```
                    )
# dc analysis
ws['analysis'](Symbol('dc'),'?param', 'v_inj', start=0, stop=vdd, step=v_inj_step)

# set design variables
ws['desVar'](   "vm", vdd/2)
ws['desVar'](   "v_inj", 0)
ws['desVar'](   "vdd", vdd  )
ws['desVar'](   "v_leak", 0)
ws['desVar'](   "idc", idc  )
# analysis order in case of multiple analysis
ws['envOption'](Symbol('analysisOrder'), ['dc'])
# to be saved currents
ws['save']( Symbol('i'), "/PM12/D", "/PM13/D", "/NM10/D", "/NM11/D")
# set temp
ws['temp'](27)

dummy = ws['paramAnalysis']('v_leak', start=vss, stop=vdd, step=v_leak_step) # values not string

# run
ws['paramRun']()
# skillbridge cannot parse stdobj@0xhexnumber type data.
dummy = ws['selectResult'](Symbol('dc'))


waves = [ws.get.data('/PM12/D'), ws.get.data('/PM13/D'), ws.get.data('/NM10/D'), ws.get.data('/NM11/D')]
data = []
n_param = 2
for wave in waves:
```

```
    mgrid = utils.n_param_wave_to_meshgrid(ws, wave, [None for _
    in range(n_param)], n_param, n_param)
    data.append(mgrid)

utils.meshgrid_to_pickle(data, n_param, 'synapse-bundle-current.
    pickle')

####################
with open ('synapse-bundle-current.pickle', 'rb') as fp:
    itemlist = pkl.load(fp)
```

Listing B.4: Synapse current mirror phase plane using Fig. 6.11(c)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 21 21:49:04 2019

@author: mhasan13
"""


from skillbridge import Workspace
from skillbridge.client.translator import Symbol
import numpy as np
import matplotlib.pyplot as plt
import utils
import pickle as pkl

########### design variables ############
vdd = 300e-3
vss = 0
v_leak = 100e-3
idc_max = 10e-9
```

```
idc = 1e-9
########## sweep variables ##############
vm_step = 0.5e-3
v_inj_step = 100e-6
########################################


# connect to server
ws = Workspace.open()

# there is a bug in IC6.18
# https://community.cadence.com/cadence_technology_forums/f/custom-ic-skill/42502/ocean-script-nested-parametric-analysis-problem
# envSetVal("spectre.envOpts" "controlMode" 'string "batch")
# the above command has to be set to get paramRun() working
ws['envSetVal']('spectre.envOpts', 'controlMode', Symbol('string'), 'batch')

# set simulator
ws['simulator'](Symbol('spectre'))
# set schematic
ws['design']('/tmp/simulation/synapse_0p3_modified/spectre/schematic/netlist/netlist')
# results directory
ws['resultsDir']( '/tmp/simulation/synapse_0p3_modified/spectre/schematic' )
# set model files
ws['modelFile'](utils.model_files[0],utils.model_files[1],utils.model_files[2],utils.model_files[3],utils.model_files[4],utils.model_files[5],utils.model_files[6],utils.model_files[7],utils.model_files[8],utils.model_files[9],
```

```
                    utils.model_files[10],utils.model_files[11],
    utils.model_files[12],utils.model_files[13],utils.model_files
    [14],utils.model_files[15],utils.model_files[16],utils.
    model_files[17],utils.model_files[18],utils.model_files[19],
                    utils.model_files[20],utils.model_files[21],
    utils.model_files[22],utils.model_files[23],utils.model_files
    [24],utils.model_files[25],utils.model_files[26],utils.
    model_files[27],utils.model_files[28]
                    )
# dc analysis
ws['analysis'](Symbol('dc'),'?param', 'v_inj', start=0, stop=vdd,
     step=v_inj_step)

# set design variables
ws['desVar'](   "vm", vdd/2)
ws['desVar'](   "v_inj", 0)
ws['desVar'](   "vdd", vdd  )
ws['desVar'](   "v_leak", 0)
ws['desVar'](   "idc", idc  )
# analysis order in case of multiple analysis
ws['envOption'](Symbol('analysisOrder'), ['dc'])
# to be saved currents
ws['save']( Symbol('i'), "/PM8/D", "/NM8/D" )
# set temp
ws['temp'](27)

dummy = ws['paramAnalysis']('vm', start=vss, stop=vdd, step=
    vm_step) # values not string

# run
ws['paramRun']()
# skillbridge cannot parse stdobj@0xhexnumber type data.
dummy = ws['selectResult'](Symbol('dc'))
```

```
waves = [ws.get.data('/PM8/D'), ws.get.data('/NM8/D')]
data = []
n_param = 2
for wave in waves:
    mgrid = utils.n_param_wave_to_meshgrid(ws, wave, [None for _
    in range(n_param)], n_param, n_param)
    data.append(mgrid)

utils.meshgrid_to_pickle(data, n_param, 'synapse-bundle-injection
    .pickle')

####################
with open ('synapse-bundle-injection.pickle', 'rb') as fp:
    itemlist = pkl.load(fp)
```

Listing B.5: Utlity functions

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 16 21:51:14 2019

@author: mhasan13
"""

from skillbridge import Workspace
import numpy as np
import pickle as pkl

# eny values in ocean containing apostrophe like 'tran use Symbol
    ('tran') in python
```

```
# use Symbol('tran') to set 'tran ; client.translator import Symbol


model_files = [
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/design.scs", ""],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "bjt_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "diode_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "res_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "moscap_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "mimcap_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108001_30.scs", "typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm142005-3.scs", "typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm142005-3.scs", "diode_typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Def"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typ_DNW"],
```

```
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typical_1V2"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typical_LVT"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typical_HVT"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typical_2V5"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typical_3V3"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "diode_typical"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "NMOSVAR_Typical"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typ_PNVar"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "Typ_RFESD"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "MIM_Typical"],
 ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "SPI_OCT_Typical"],
```

```
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "SYM_Typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "CT_Typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "X_Typical"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "BALUN"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "RFBP_Typ"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "CPW"],
    ["/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/pdk/gf-130/chrt13rf_7LM/models/Spectre/sm108002_24.scs", "MSL"]
        ]


def waveform_to_vector(ws, waveform):
    y_wave = ws.dr.get_waveform_y_vec(waveform)
    y_vec = []
    for i in range(ws.dr.vector_length(y_wave)):
        y_vec.append(ws.dr.get_elem(y_wave, i))

    x_wave = ws.dr.get_waveform_x_vec(waveform)
    x_vec = []
    for i in range(ws.dr.vector_length(x_wave)):
        x_vec.append(ws.dr.get_elem(x_wave, i))

    return x_vec, y_vec
```

```

def param_waveform_to_vector(ws, wave):
    x_waveform = ws.dr.get_waveform_x_vec(wave) # contains the value of param sweeps
    y_waveform = ws.dr.get_waveform_y_vec(wave) # list of waveforms each entry for param sweep

    x_vector_list = []
    for i in range(ws.dr.vector_length(x_waveform)):
        x_vector_list.append(ws.dr.get_elem(x_waveform, i))

    y_vector_list = []
    for i in range(ws.dr.vector_length(y_waveform)):
        y_vector_list.append(waveform_to_vector(ws, ws.dr.get_elem(y_waveform, i)) )

    return x_vector_list, y_vector_list
    ####################
    # y_vector_list data looks like this
    # [...,[  ith sweep plot   ],..]
    # [...,[ [x vect], [yvect] ],..]

# ============================================================================
# structure of cadence waveform
# (v1,(v2,(v3,y))))
# ============================================================================
def param_waveform_to_meshgrid(ws, wave):
    var_1_vector = ws.dr.get_waveform_x_vec(wave) # 1st parameter sweep
```

```
    var_2_pack = ws.dr.get_waveform_y_vec(wave) # 2nd parameter sweep+output content values=>waveform

    var_1_list = []
    var_2_list = []
    content_list = []
    for i in range(ws.dr.vector_length(var_1_vector)):
        var_1 = ws.dr.get_elem(var_1_vector, i)
        var_2_waveform = ws.dr.get_elem(var_2_pack, i)
        var_2_vector = ws.dr.get_waveform_x_vec(var_2_waveform) # 2nd parameter sweep
        content_vector = ws.dr.get_waveform_y_vec(var_2_waveform) # output values

        cnt_list = []
        v1_list = []
        v2_list = []
        for j in range(ws.dr.vector_length(var_2_vector)):
            var_2 = ws.dr.get_elem(var_2_vector, j)
            content = ws.dr.get_elem(content_vector, j)
            cnt_list.append(content)
            v2_list.append(var_2)
            v1_list.append(var_1)

        content_list.append(cnt_list)
        var_2_list.append(v2_list)
        var_1_list.append(v1_list)

    return np.array(var_1_list), np.array(var_2_list), np.array(content_list)
```

```python
# https://stackoverflow.com/questions/7186518/function-with-varying-number-of-for-loops-python
def n_param_wave_to_meshgrid(ws, waveform, param_passing, ith_param, n_param):
    '''
    param_passing = [None for _ in range(n_param)] when called
    ith_param = n_param when called
    '''
    # create empty list of size n_param
    # https://stackoverflow.com/questions/10617045/how-to-create-a-fix-size-list-in-python
    n_param_storage = [ [] for _ in range(n_param+1)] # nparam + content
    x = ws.dr.get_waveform_x_vec(waveform)
    y = ws.dr.get_waveform_y_vec(waveform)

    for i in range(ws.dr.vector_length(x)):
        x_var = ws.dr.get_elem(x, i)
        y_var = ws.dr.get_elem(y, i)
        if ith_param > 1:
            param_passing[n_param-ith_param] = x_var
            returned_n_param = n_param_wave_to_meshgrid(ws, y_var, param_passing, ith_param-1, n_param)
            for j in range(n_param+1):
                n_param_storage[j].append(returned_n_param[j])
        else:
            for j in range(n_param-1):
                n_param_storage[j].append(param_passing[j])
            n_param_storage[j+1].append(x_var)
            n_param_storage[j+2].append(y_var)

    return n_param_storage

```

```

def meshgrid_to_pickle(data, n_param, file_name):
    '''
    take the list returned by n_param_wave_to_meshgrid()
    remove redundant params and make on list
    save them in pickle

    the first dimension packs the last thing that was appened
    hence the earliest things appened are accessed by highest
    dimension
    '''
    params = data[0][0:n_param]
    for content in data:
        params.append(content[n_param])

#     https://stackoverflow.com/questions/899103/writing-a-list-to
    -a-file-with-python
    with open(file_name, 'wb') as file:
        pkl.dump(params, file)

def transient_waveform_to_vector(ws, waveforms):
    vectors =[]
    for wave in waveforms:
        y_wave = ws.dr.get_waveform_y_vec(wave)
        y_vec = []
        for i in range(ws.dr.vector_length(y_wave)):
            y_vec.append(ws.dr.get_elem(y_wave, i))
        vectors.append(y_vec)

    # x vector is same for all these y vector
    x_wave = ws.dr.get_waveform_x_vec(wave)
    x_vec = []
    for i in range(ws.dr.vector_length(x_wave)):
```

```
        x_vec.append(ws.dr.get_elem(x_wave, i))

    return vectors, x_vec


def transient_to_pickle(vectors, file_name):
    with open(file_name, 'wb') as file:
        pkl.dump(vectors, file)
```

B.2 Spiking Neural Network Simulation

Listing B.6: SNN simulation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May 19 22:20:51 2020

@author: mhasan13
"""
from ObjectClass import *
import numpy as np
import matplotlib.pyplot as plt
import random
import cv2 as cv
import pickle as pkl
from scipy.interpolate import interp1d
import time
import brian2 as br

br.prefs.codegen.target = 'numpy'
br.start_scope()
dt = br.defaultclock.dt = 1*br.us

```

```

# ==========================================================================
# # mnist data preparation
# ==========================================================================
reduced_row = reduced_col = 16
mnist_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/mnist_test.csv'
mnist_data = np.loadtxt(mnist_file, delimiter=',')
images = mnist_data[:,1:]
labels = mnist_data[:,0]
# fetch a random digit
random_idx = random.choice(range(len(labels)))
image = images[random_idx,:].reshape((28,28))
image = cv.resize(image,(reduced_row,reduced_row),cv.INTER_CUBIC)

# ==========================================================================
# # TF weights in transposed state => #rows=input, #cols=output
# ==========================================================================
weight_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_0_weights.csv'
weight_1 = np.loadtxt(weight_file, delimiter=',')
weight_1_max = np.max(np.abs(weight_1))
```

```
weight_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_1_weights.csv'
weight_2 = np.loadtxt(weight_file, delimiter=',')
weight_2_max = np.max(np.abs(weight_2))

# biases
bias_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_0_biases.csv'
bias_1 = np.loadtxt(bias_file, delimiter=',')
bias_1_max = np.max(np.abs(bias_1))
bias_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_1_biases.csv'
bias_2 = np.loadtxt(bias_file, delimiter=',')
bias_2_max = np.max(np.abs(bias_2))

# normalize weights or not
weight_max = np.max([weight_1_max, weight_2_max, bias_1_max, bias_2_max])
weight_layer_1 = weight_1/weight_1_max
weight_layer_2 = weight_2/weight_2_max
bias_layer_1 = bias_1/bias_1_max
bias_layer_2 = bias_2/bias_2_max

# weight to floating gate voltage
synapse_meshgrid = SynapseMeshGrid('../../meshgrid-generation/v3/synapse-active.pickle',
                                   '../../meshgrid-generation/v3/synapse-inactive.pickle')
vdp_at = 50*br.mV
vdn_at = 50*br.mV
```

```
Ip_at = synapse_meshgrid.Ip_active[:, int(synapse_meshgrid.j_per_vd*vdp_at)]
In_at = synapse_meshgrid.In_active[:, int(synapse_meshgrid.j_per_vd*vdn_at)]
I_syn = In_at - Ip_at
vfg_syn = synapse_meshgrid.vfg[:, int(synapse_meshgrid.j_per_vd*vdn_at)]
f_w_to_vfg = interp1d(I_syn, vfg_syn)
I_max = 500e-12
vfg_layer_1 = f_w_to_vfg(weight_layer_1*I_max)
vfg_layer_2 = f_w_to_vfg(weight_layer_2*I_max)
bias_vfg_layer_1 = f_w_to_vfg(bias_layer_1*I_max)
bias_vfg_layer_2 = f_w_to_vfg(bias_layer_2*I_max)
# =============================================================================
# meshgrid data
# =============================================================================
neuron_meshgrid = NeuronMeshGrid('../../meshgrid-generation/v3/neuron.pickle')

@br.check_units(i=1, j=1, result=1)
def Cv_current(i:int, j:int) -> float:

    return neuron_meshgrid.iCv[i,j]

@br.check_units(i=1, j=1, result=1)
def Cu_current(i:int, j:int) -> float:

    return neuron_meshgrid.iCu[i,j]

```

```python
synapse_meshgrid = SynapseMeshGrid('../../meshgrid-generation/v3/synapse-active.pickle',
                                   '../../meshgrid-generation/v3/synapse-inactive.pickle')

@br.check_units(i=1, j=1, result=1)
def syn_active_p(i:int, j:int) -> float:

    return synapse_meshgrid.Ip_active[i,j]
@br.check_units(i=1, j=1, result=1)
def syn_active_n(i:int, j:int) -> float:

    return synapse_meshgrid.In_active[i,j]

@br.check_units(i=1, j=1, result=1)
def syn_inactive_p(i:int, j:int) -> float:

    return synapse_meshgrid.Ip_inactive[i,j]
@br.check_units(i=1, j=1, result=1)
def syn_inactive_n(i:int, j:int) -> float:

    return synapse_meshgrid.In_inactive[i,j]

bundle_synapse_meshgrid = BundleSynapseMeshGrid('../../meshgrid-generation/v3/synapse-bundle-current.pickle',
                                                '../../meshgrid-generation/v3/synapse-bundle-injection.pickle')

@br.check_units(i=1, j=1, result=1)
def ip_bundle(i:int, j:int) -> float:

    return bundle_synapse_meshgrid.Ip_bundle[i,j]

```

```
@br.check_units(i=1, j=1, result=1)
def in_bundle(i:int, j:int) -> float:

    return bundle_synapse_meshgrid.In_bundle[i,j]

@br.check_units(i=1, jp=1 , jn=1, result=1)
def i_injection(i:int, jp:int, jn:int) -> float:

    return bundle_synapse_meshgrid.Ip_injection[i,jp] - bundle_synapse_meshgrid.In_injection[i,jn]

@br.check_units(Ip_bundle=br.amp, In_bundle=br.amp, Ip=br.amp, In=br.amp, vp_inj=br.volt, vn_inj=br.volt, result=1)
def debug(Ip_bundle, In_bundle, Ip, In, vp_inj, vn_inj):
#     print(Ip_bundle, In_bundle, Ip, In, vp_inj, vn_inj)
    return 0
# =============================================================================
# network preparation
# =============================================================================
f_factor = 1
L0 = InputGroupBrian(reduced_row*reduced_col)
L0.L.pulse_width = 45e-6
L0.L.frequency = image.flatten()*f_factor
L0_mon = br.StateMonitor(L0.L, ('s'), record=True)
L0_spk = br.SpikeMonitor(L0.L, record=True)
# next layer
L1 = NeuronGroupBrian(neuron_meshgrid, bundle_synapse_meshgrid, 32)
L1.L.vp_leak = 68*br.mV
```

```
L1.L.vn_leak = 170*br.mV
L1_mon = br.StateMonitor(L1.L, ('v','u','Isyn','IpT','InT','vp_inj','vn_inj'), record=True)
L1_spk = br.SpikeMonitor(L1.L, record=True)
# next layer
L2 = NeuronGroupBrian(neuron_meshgrid, bundle_synapse_meshgrid, 10)
L2.L.vp_leak = 130*br.mV
L2.L.vn_leak = 100*br.mV
L2_mon = br.StateMonitor(L2.L, ('v','u','Isyn','IpT','InT','vp_inj','vn_inj'), record=True)
L2_spk = br.SpikeMonitor(L2.L, record=True)
# bias generator
B0 = InputGroupBrian(1)
B0.L.pulse_width = 45e-6
B0.L.frequency = 255*f_factor
B1 = InputGroupBrian(1)
B1.L.pulse_width = 45e-6
B1.L.frequency = 255*f_factor
# =============================================================================
# synapse
# =============================================================================
W1 = SynapseGroupBrian(synapse_meshgrid, L0,L1)
W1.S.vg_p = vfg_layer_1.flatten(order='C')*br.volt
W1.S.vg_n = vfg_layer_1.flatten(order='C')*br.volt
W1_b = SynapseGroupBrian(synapse_meshgrid, B0,L1)
W1_b.S.vg_p = bias_vfg_layer_1.flatten(order='C')*br.volt
W1_b.S.vg_n = bias_vfg_layer_1.flatten(order='C')*br.volt
# next layer
```

```
W2 = SynapseGroupBrian(synapse_meshgrid, L1,L2)
W2.S.vg_p = vfg_layer_2.flatten(order='C')*br.volt
W2.S.vg_n = vfg_layer_2.flatten(order='C')*br.volt
W2_b = SynapseGroupBrian(synapse_meshgrid, B1,L2)
W2_b.S.vg_p = bias_vfg_layer_2.flatten(order='C')*br.volt
W2_b.S.vg_n = bias_vfg_layer_2.flatten(order='C')*br.volt
# ============================================================================
# fix capacitor
# ============================================================================
#L1.L.Cdp_bundle = 642e-15*br.farad
#L1.L.Cdn_bundle = 642e-15*br.farad
#L2.L.Cdp_bundle = 5.5e-15*br.farad
#L2.L.Cdn_bundle = 5.5e-15*br.farad
# ============================================================================
# run and record
# ============================================================================
start_time = time.time()
sim_time = 50*br.ms
net = br.Network()
net.add(L0.L, L1.L, L2.L, B0.L, B1.L, W1.S, W2.S, W1_b.S,W2_b.S, L0_mon, L1_mon, L2_mon, L0_spk, L1_spk, L2_spk) #W1_b.S,W2_b.S,
net.run(sim_time)
stop_time = time.time()
print('time to run() ', stop_time-start_time)
```

```
plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.gca().xaxis.set_major_locator(plt.NullLocator())
plt.gca().yaxis.set_major_locator(plt.NullLocator())
plt.subplot(122)
plt.plot(L2_spk.t/br.ms,L2_spk.i,marker='.',linestyle='none')
plt.xlabel('time (ms)'), plt.ylabel('neuron index')
plt.gca().set_yticks(range(10))
plt.grid(True)
plt.gcf().set_size_inches(10,3)
plt.gcf().set_tight_layout(True)
plt.figure()
plt.bar(range(10),L2_spk.count)

# save for later comparison with cadence
#with open(str(random_idx)+'.pickle' , 'wb') as file:
#    itemlist = [list(L2_spk.i), list(L2_spk.t/br.second)]
#    pkl.dump(itemlist, file)

#plt.plot(L1_mon.t/br.ms,L1_mon.v[0])
#plt.figure()
#plt.plot(L1_mon.t/br.ms,L1_mon.Isyn[0])
plt.show()
```

Listing B.7: SNN simulation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May 19 22:20:51 2020

@author: mhasan13
"""
from ObjectClass import *
```

```
import numpy as np
import matplotlib.pyplot as plt
import random
import cv2 as cv
import pickle as pkl
from scipy.interpolate import interp1d
import time
import brian2 as br

br.prefs.codegen.target = 'numpy'
br.start_scope()
dt = br.defaultclock.dt = 1*br.us


# =============================================================================
# # mnist data preparation
# =============================================================================
reduced_row = reduced_col = 16
mnist_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/
    mhasan13/fc-spiking-mnist/smaller/data/mnist_test.csv'
mnist_data = np.loadtxt(mnist_file, delimiter=',')
images = mnist_data[:,1:]
labels = mnist_data[:,0]
# fetch a random digit
random_idx = random.choice(range(len(labels)))
image = images[random_idx,:].reshape((28,28))
image = cv.resize(image,(reduced_row,reduced_row),cv.INTER_CUBIC)

```

```
# =============================================================================
# # TF weights in transposed state => #rows=input, #cols=output
# =============================================================================
weight_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_0_weights.csv'
weight_1 = np.loadtxt(weight_file, delimiter=',')
weight_1_max = np.max(np.abs(weight_1))
weight_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_1_weights.csv'
weight_2 = np.loadtxt(weight_file, delimiter=',')
weight_2_max = np.max(np.abs(weight_2))

# biases
bias_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_0_biases.csv'
bias_1 = np.loadtxt(bias_file, delimiter=',')
bias_1_max = np.max(np.abs(bias_1))
bias_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/hidden_layer_1_biases.csv'
bias_2 = np.loadtxt(bias_file, delimiter=',')
bias_2_max = np.max(np.abs(bias_2))

# normalize weights or not
weight_max = np.max([weight_1_max, weight_2_max, bias_1_max, bias_2_max])
```

```
weight_layer_1 = weight_1/weight_1_max
weight_layer_2 = weight_2/weight_2_max
bias_layer_1 = bias_1/bias_1_max
bias_layer_2 = bias_2/bias_2_max

# weight to floating gate voltage
synapse_meshgrid = SynapseMeshGrid('../../meshgrid-generation/v3/synapse-active.pickle',
                                   '../../meshgrid-generation/v3/synapse-inactive.pickle')
vdp_at = 50*br.mV
vdn_at = 50*br.mV
Ip_at = synapse_meshgrid.Ip_active[:, int(synapse_meshgrid.j_per_vd*vdp_at)]
In_at = synapse_meshgrid.In_active[:, int(synapse_meshgrid.j_per_vd*vdn_at)]
I_syn = In_at - Ip_at
vfg_syn = synapse_meshgrid.vfg[:, int(synapse_meshgrid.j_per_vd*vdn_at)]
f_w_to_vfg = interp1d(I_syn, vfg_syn)
I_max = 500e-12
vfg_layer_1 = f_w_to_vfg(weight_layer_1*I_max)
vfg_layer_2 = f_w_to_vfg(weight_layer_2*I_max)
bias_vfg_layer_1 = f_w_to_vfg(bias_layer_1*I_max)
bias_vfg_layer_2 = f_w_to_vfg(bias_layer_2*I_max)
# ==============================================================================
# meshgrid data
# ==============================================================================
```

```
neuron_meshgrid = NeuronMeshGrid('../../meshgrid-generation/v3/neuron.pickle')

@br.check_units(i=1, j=1, result=1)
def Cv_current(i:int, j:int) -> float:

    return neuron_meshgrid.iCv[i,j]

@br.check_units(i=1, j=1, result=1)
def Cu_current(i:int, j:int) -> float:

    return neuron_meshgrid.iCu[i,j]

synapse_meshgrid = SynapseMeshGrid('../../meshgrid-generation/v3/synapse-active.pickle',
                                   '../../meshgrid-generation/v3/synapse-inactive.pickle')

@br.check_units(i=1, j=1, result=1)
def syn_active_p(i:int, j:int) -> float:

    return synapse_meshgrid.Ip_active[i,j]
@br.check_units(i=1, j=1, result=1)
def syn_active_n(i:int, j:int) -> float:

    return synapse_meshgrid.In_active[i,j]

@br.check_units(i=1, j=1, result=1)
def syn_inactive_p(i:int, j:int) -> float:

    return synapse_meshgrid.Ip_inactive[i,j]
@br.check_units(i=1, j=1, result=1)
def syn_inactive_n(i:int, j:int) -> float:
```

```

    return synapse_meshgrid.In_inactive[i,j]

bundle_synapse_meshgrid = BundleSynapseMeshGrid('../../meshgrid-generation/v3/synapse-bundle-current.pickle',
                                                '../../meshgrid-generation/v3/synapse-bundle-injection.pickle')

@br.check_units(i=1, j=1, result=1)
def ip_bundle(i:int, j:int) -> float:

    return bundle_synapse_meshgrid.Ip_bundle[i,j]

@br.check_units(i=1, j=1, result=1)
def in_bundle(i:int, j:int) -> float:

    return bundle_synapse_meshgrid.In_bundle[i,j]

@br.check_units(i=1, jp=1 , jn=1, result=1)
def i_injection(i:int, jp:int, jn:int) -> float:

    return bundle_synapse_meshgrid.Ip_injection[i,jp] - bundle_synapse_meshgrid.In_injection[i,jn]

@br.check_units(Ip_bundle=br.amp, In_bundle=br.amp, Ip=br.amp, In=br.amp, vp_inj=br.volt, vn_inj=br.volt, result=1)
def debug(Ip_bundle, In_bundle, Ip, In, vp_inj, vn_inj):
#    print(Ip_bundle, In_bundle, Ip, In, vp_inj, vn_inj)
    return 0
# ==============================================================================
# network preparation
```

```
#
    ============================================================================
f_factor = 1
L0 = InputGroupBrian(reduced_row*reduced_col)
L0.L.pulse_width = 45e-6
L0.L.frequency = image.flatten()*f_factor
L0_mon = br.StateMonitor(L0.L, ('s'), record=True)
L0_spk = br.SpikeMonitor(L0.L, record=True)
# next layer
L1 = NeuronGroupBrian(neuron_meshgrid, bundle_synapse_meshgrid,
    32)
L1.L.vp_leak = 68*br.mV
L1.L.vn_leak = 170*br.mV
L1_mon = br.StateMonitor(L1.L, ('v','u','Isyn','IpT','InT','
    vp_inj','vn_inj'), record=True)
L1_spk = br.SpikeMonitor(L1.L, record=True)
# next layer
L2 = NeuronGroupBrian(neuron_meshgrid, bundle_synapse_meshgrid,
    10)
L2.L.vp_leak = 130*br.mV
L2.L.vn_leak = 100*br.mV
L2_mon = br.StateMonitor(L2.L, ('v','u','Isyn','IpT','InT','
    vp_inj','vn_inj'), record=True)
L2_spk = br.SpikeMonitor(L2.L, record=True)
# bias generator
B0 = InputGroupBrian(1)
B0.L.pulse_width = 45e-6
B0.L.frequency = 255*f_factor
B1 = InputGroupBrian(1)
B1.L.pulse_width = 45e-6
B1.L.frequency = 255*f_factor
```

```
# =============================================================================
# synapse
# =============================================================================
W1 = SynapseGroupBrian(synapse_meshgrid, L0,L1)
W1.S.vg_p = vfg_layer_1.flatten(order='C')*br.volt
W1.S.vg_n = vfg_layer_1.flatten(order='C')*br.volt
W1_b = SynapseGroupBrian(synapse_meshgrid, B0,L1)
W1_b.S.vg_p = bias_vfg_layer_1.flatten(order='C')*br.volt
W1_b.S.vg_n = bias_vfg_layer_1.flatten(order='C')*br.volt
# next layer
W2 = SynapseGroupBrian(synapse_meshgrid, L1,L2)
W2.S.vg_p = vfg_layer_2.flatten(order='C')*br.volt
W2.S.vg_n = vfg_layer_2.flatten(order='C')*br.volt
W2_b = SynapseGroupBrian(synapse_meshgrid, B1,L2)
W2_b.S.vg_p = bias_vfg_layer_2.flatten(order='C')*br.volt
W2_b.S.vg_n = bias_vfg_layer_2.flatten(order='C')*br.volt
# =============================================================================
# fix capacitor
# =============================================================================
#L1.L.Cdp_bundle = 642e-15*br.farad
#L1.L.Cdn_bundle = 642e-15*br.farad
#L2.L.Cdp_bundle = 5.5e-15*br.farad
#L2.L.Cdn_bundle = 5.5e-15*br.farad
```

```
# ================================================================================
# run and record
# ================================================================================
start_time = time.time()
sim_time = 50*br.ms
net = br.Network()
net.add(L0.L, L1.L, L2.L, B0.L, B1.L, W1.S, W2.S, W1_b.S,W2_b.S, L0_mon, L1_mon, L2_mon, L0_spk, L1_spk, L2_spk) #W1_b.S,W2_b.S,
net.run(sim_time)
stop_time = time.time()
print('time to run() ', stop_time-start_time)
plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.gca().xaxis.set_major_locator(plt.NullLocator())
plt.gca().yaxis.set_major_locator(plt.NullLocator())
plt.subplot(122)
plt.plot(L2_spk.t/br.ms,L2_spk.i,marker='.',linestyle='none')
plt.xlabel('time (ms)'), plt.ylabel('neuron index')
plt.gca().set_yticks(range(10))
plt.grid(True)
plt.gcf().set_size_inches(10,3)
plt.gcf().set_tight_layout(True)
plt.figure()
plt.bar(range(10),L2_spk.count)

# save for later comparison with cadence
#with open(str(random_idx)+'.pickle' , 'wb') as file:
#    itemlist = [list(L2_spk.i), list(L2_spk.t/br.second)]
```

```
#    pkl.dump(itemlist, file)

#plt.plot(L1_mon.t/br.ms,L1_mon.v[0])
#plt.figure()
#plt.plot(L1_mon.t/br.ms,L1_mon.Isyn[0])
plt.show()
```

Listing B.8: Spike count from Cadence simulation output

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 11 20:52:55 2020

@author: mhasan13
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle as pkl

digit = 8592


# 6947 is based on vfg at Imax=100pA, vref=130mV, vd=150mV,
    global weight normalization
# 5513, 5763 is based on vfg at Imax=50pA, vref=130mV, vd = 100mV
    , global weight normalization
file = pd.read_csv(str(digit)+'.csv', header=0)
data = file.values

t = []
neuron = []
```

```

n = data.shape[1]
for i in range(int(n/2)):
    spikes = data[:,2*i+1]
    time = data[:,2*i]
    spikes[spikes>0.1] = 1
    spikes[spikes<0.1] = 0
    difference = np.diff(spikes)
    idxs = np.where(difference==1)[0] + 1
    t.extend(time[idxs])
    neuron.extend([i]*len(idxs))

t = np.array(t)
neuron = np.array(neuron)
plt.plot( t/1e-3,neuron,marker='o',markersize=6,fillstyle='none',
    linestyle='none')

neuron_count = []
for i in range(int(n/2)):
    neuron_count.append(np.sum(neuron==i))

#plt.bar(range(10),neuron_count)
```

Listing B.9: Spike count from Cadence simulation output

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 11 20:52:55 2020

@author: mhasan13
"""

import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import pickle as pkl
import cv2 as cv


# #########
# plt.rc('text', usetex=True)
# plt.rc('font', family='Times')
# plt.rc('font', size=16)
# #########

def find_spikes(data):

    t = []
    neuron = []

    n = data.shape[1]
    for i in range(int(n/2)):
        spikes = data[:,2*i+1]
        time = data[:,2*i]
        spikes[spikes>0.1] = 1
        spikes[spikes<0.1] = 0
        difference = np.diff(spikes)
        idxs = np.where(difference==1)[0] + 1
        t.extend(time[idxs])
        neuron.extend([i]*len(idxs))

    return np.array(neuron), np.array(t)


```

```python
# =============================================================================
# # mnist data preparation
# =============================================================================
reduced_row = reduced_col = 16
mnist_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/mnist_test.csv'
mnist_data = np.loadtxt(mnist_file, delimiter=',')
images = mnist_data[:,1:]
labels = mnist_data[:,0]
###########################
digits = [8592, 6585, 8747]
#digits = [digits[0]]
# https://stackoverflow.com/questions/34162443/why-do-many-examples-use-fig-ax-plt-subplots-in-matplotlib-pyplot-python
# https://www.delftstack.com/howto/matplotlib/how-to-make-different-subplot-sizes-in-matplotlib/
fig = plt.figure()
axs = []
for i in range(len(digits)):
    ax = []
    for j in range(2):
        ax.append(fig.add_subplot(len(digits),2,i*2+j+1))
    axs.append(ax)

d = 0
width = 0.5
x_dig = np.array( [i*2 for i in range(10)] )
for digit in digits:
    file = pd.read_csv(str(digit)+'.csv', header=0)
```

```
    data = file.values

    neuron, t = find_spikes(data)

    neuron_count = []
    for i in range(10):
        neuron_count.append(np.sum(neuron==i))


    axs[d][1].bar(x_dig - width/2, neuron_count, color='g', label='CADENCE tran.')
############################################################
    with open(str(digit)+'.pickle', 'rb') as file:
        itemlist = pkl.load(file)

    br_neuron = np.array(itemlist[0])
    br_t = np.array(itemlist[1])
    br_neuron_count = []
    for i in range(10):
        br_neuron_count.append(np.sum(br_neuron==i))

    image = images[digit,:].reshape((28,28))
    image = cv.resize(image,(reduced_row,reduced_row),cv.INTER_CUBIC)
    axs[d][0].imshow(image, cmap='gray')
    axs[d][0].set_xlabel('input image')
    axs[d][0].xaxis.set_major_locator(plt.NullLocator())
    axs[d][0].yaxis.set_major_locator(plt.NullLocator())

    axs[d][1].bar(x_dig + width/2, br_neuron_count, color='r', label='Phase Plane')
    axs[d][1].set_xlabel('neuron index')
    axs[d][1].set_xticks( x_dig, [str(i) for i in range(10)] )
```

```
    axs[d][1].set_ylabel('spike count')
    axs[d][1].legend(bbox_to_anchor=(1,0.6),fontsize=11)

    d += 1




# #!/usr/bin/env python3
# # -*- coding: utf-8 -*-
# """
# Created on Thu Jun 11 20:52:55 2020

# @author: mhasan13
# """

# import pandas as pd
# import numpy as np
# import matplotlib.pyplot as plt
# import pickle as pkl
# import cv2 as cv


# #########
# plt.rc('text', usetex=True)
# plt.rc('font', family='Times')
# plt.rc('font', size=16)
# #########

# def find_spikes(data):

#     t = []
#     neuron = []
```

```
#     n = data.shape[1]
#     for i in range(int(n/2)):
#         spikes = data[:,2*i+1]
#         time = data[:,2*i]
#         spikes[spikes>0.1] = 1
#         spikes[spikes<0.1] = 0
#         difference = np.diff(spikes)
#         idxs = np.where(difference==1)[0] + 1
#         t.extend(time[idxs])
#         neuron.extend([i]*len(idxs))

#     return np.array(neuron), np.array(t)


# # =============================================================================
# # # mnist data preparation
# # =============================================================================
# reduced_row = reduced_col = 16
# mnist_file = '/nfs/users/mhasan13/linux/Desktop/iss-research_nfs/mhasan13/fc-spiking-mnist/smaller/data/mnist_test.csv'
# mnist_data = np.loadtxt(mnist_file, delimiter=',')
# images = mnist_data[:,1:]
# labels = mnist_data[:,0]
# ##########################
# digits = [8592, 6585, 8747]
# #digits = [digits[0]]
```

```
# # https://stackoverflow.com/questions/34162443/why-do-many-examples-use-fig-ax-plt-subplots-in-matplotlib-pyplot-python
# # https://www.delftstack.com/howto/matplotlib/how-to-make-different-subplot-sizes-in-matplotlib/
# fig = plt.figure()
# axs = []
# for i in range(len(digits)):
#     ax = []
#     for j in range(3):
#         ax.append(fig.add_subplot(len(digits),3,i*3+j+1))
#     axs.append(ax)

# d = 0
# width = 0.5
# x_dig = np.array( [i*2 for i in range(10)] )
# for digit in digits:
#     file = pd.read_csv(str(digit)+'.csv', header=0)
#     data = file.values

#     neuron, t = find_spikes(data)

#     neuron_count = []
#     for i in range(10):
#         neuron_count.append(np.sum(neuron==i))

#     axs[d][1].plot(t/1e-3,neuron, marker='o',markersize=6, fillstyle='none',linestyle='none', color='g', label='CADENCE tran.')
#     axs[d][1].set_yticks(range(10))
#     axs[d][2].bar(x_dig - width/2, neuron_count, color='g', label='CADENCE tran.')
# ##########################################################
#     with open(str(digit)+'.pickle', 'rb') as file:
```

```
#         itemlist = pkl.load(file)

#     br_neuron = np.array(itemlist[0])
#     br_t = np.array(itemlist[1])
#     br_neuron_count = []
#     for i in range(10):
#         br_neuron_count.append(np.sum(br_neuron==i))

#     image = images[digit,:].reshape((28,28))
#     image = cv.resize(image,(reduced_row,reduced_row),cv.INTER_CUBIC)
#     axs[d][0].imshow(image, cmap='gray')
#     axs[d][0].set_xlabel('input image')
#     axs[d][0].xaxis.set_major_locator(plt.NullLocator())
#     axs[d][0].yaxis.set_major_locator(plt.NullLocator())
#     axs[d][1].plot(br_t/1e-3, br_neuron, marker='.', linestyle='none', color='r', label='Phase Plane')
#     axs[d][1].set_xlabel('t (ms)')
#     axs[d][1].set_xticks(range(0,60,10))
#     axs[d][1].set_ylabel('neuron index')
#     axs[d][1].legend(bbox_to_anchor=(0.5,1), fontsize=11)
#     axs[d][1].set_yticks(range(10))
#     axs[d][1].grid(True)
#     axs[d][2].bar(x_dig + width/2, br_neuron_count, color='r', label='Phase Plane')
#     axs[d][2].set_xlabel('neuron index')
#     axs[d][2].set_xticks( x_dig, [str(i) for i in range(10)] )
#     axs[d][2].set_ylabel('spike count')
#     axs[d][2].legend(bbox_to_anchor=(1,0.6),fontsize=11)

#     d += 1
```

Listing B.10: Class definitions

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun  2 03:23:55 2020

@author: mhasan13
"""

import pickle as pkl
import numpy as np
import brian2 as br

class NeuronMeshGrid:
    '''
    Data on neuron phase plane
    '''
    def __init__(self, pickle_path:str) -> None:
        with open(pickle_path, 'rb') as fp:
            itemlist = pkl.load(fp)

        data = np.array(itemlist)
        self.u = data[0,:,:]
        self.v = data[1,:,:]
# =============================================================================
#         -ve sign has to be fixed for pmos currents now
#         as cadence introduced a -ve sign for outgoing current
# =============================================================================
        self.iCv = -data[2,:,:] - data[3,:,:] - data[6,:,:] # Ipos_feed - Ineg_feed - I_leak
```

```
        self.iCu = -data[4,:,:] - data[5,:,:] # Iw - Ir
        self.axon = data[7,:,:] # axon output
# ==============================================================================
#          i=>y axis index, j=>x axis index
# ==============================================================================
        self.vmax, self.vmin = np.max(self.v), np.min(self.v)
        self.umax, self.umin = np.max(self.u), np.min(self.u)
        self.j_per_x = (self.v.shape[1]-1)/(self.vmax-self.vmin)
        self.i_per_y = (self.u.shape[0]-1)/(self.umax-self.umin)

class SynapseMeshGrid:
    '''
    Data on synpase meshgrid
    '''
    def __init__(self, active_path:str, inactive_path:str) -> None:
        # active synapse
        with open (active_path, 'rb') as fp:
            itemlist = pkl.load(fp)

        data = np.array(itemlist)
        self.vfg = data[0,:,:]
        self.vd = data[1,:,:]
# ==============================================================================
#          -ve sign has to be fixed for pmos currents now
#          as cadence introduced a -ve sign for outgoing current
```

```
# ==============================================================================
        self.Ip_active = -data[2,:,:]
        self.In_active = data[3,:,:]

        with open (inactive_path, 'rb') as fp:
            itemlist = pkl.load(fp)

        data = np.array(itemlist)
        self.Ip_inactive = -data[2,:,:]
        self.In_inactive = data[3,:,:]
# ==============================================================================
#          i=>y axis index, j=>x axis index
# ==============================================================================
        self.vfg_max, self.vfg_min = np.max(self.vfg), np.min(self.vfg)
        self.vd_max, self.vd_min = np.max(self.vd), np.min(self.vd)
        self.i_per_vfg = (self.vfg.shape[0]-1)/(self.vfg_max-self.vfg_min)
        self.j_per_vd = (self.vd.shape[1]-1)/(self.vd_max-self.vd_min)

class BundleSynapseMeshGrid:
    '''
    Data on bundle synapse injection current
    '''
```

```python
    def __init__(self, i_bundle_path:str, i_inj_path:str) -> None:
        with open(i_bundle_path, 'rb') as fp:
            itemlist = pkl.load(fp)

        data = np.array(itemlist)
        self.v_leak = data[0,:,:]
        self.vd = data[1,:,:]
# ============================================================================
#          -ve sign has to be fixed for pmos currents now
#          as cadence introduced a -ve sign for outgoing current
# ============================================================================
        self.Ip_bundle = -(data[2,:,:]+data[3,:,:])
        self.In_bundle =   data[4,:,:]+data[5,:,:]
# ============================================================================
#          i=>y axis index, j=>x axis index
# ============================================================================
        self.v_leak_max, self.v_leak_min = np.max(self.v_leak), np.min(self.v_leak)
        self.vd_max, self.vd_min = np.max(self.vd), np.min(self.vd)
        self.i_per_v_leak = (self.v_leak.shape[0]-1)/(self.v_leak_max-self.v_leak_min)
        self.j_per_vd = (self.vd.shape[1]-1)/(self.vd_max-self.vd_min)
```

```

        with open(i_inj_path, 'rb') as fp:
            itemlist = pkl.load(fp)

        data = np.array(itemlist)
        self.vm = data[0,:,:]
        self.v_inj = data[1,:,:]
# ==============================================================================
#          -ve sign has to be fixed for pmos currents now
#          as cadence introduced a -ve sign for outgoing current
# ==============================================================================
        self.Ip_injection = -data[2,:,:]
        self.In_injection = data[3,:,:]
# ==============================================================================
#          i=>y axis index, j=>x axis index
# ==============================================================================
        self.v_inj_max, self.v_inj_min = np.max(self.v_inj), np.min(self.v_inj)
        self.vm_max, self.vm_min = np.max(self.vm), np.min(self.vm)
        self.i_per_vm = (self.vm.shape[0]-1)/(self.vm_max-self.vm_min)
        self.j_per_v_inj = (self.v_inj.shape[1]-1)/(self.v_inj_max-self.v_inj_min)

```

```
class InputGroupBrian:
    '''
    Input spike generation from frequency
    '''
    def __init__(self, n:int) -> None:
        self.dt = br.defaultclock.dt

        self.input_neuron_model='''
                            dx/dt = 1/second : 1
                            s : 1
                            frequency : 1
                            t_period = 1/(frequency+1e-15) : 1
                            pulse_width : 1
                            '''
        self.input_spike_event_action = '''
                                    s += 1
                                    '''
        self.input_reset_event_action = '''
                                    x = pulse_width
                                    s = 0
                                    '''
        self.input_neuron_events={
                            'spike':'s==1',
                            'spike_event':'x>t_period',
                            'resetting':'x>t_period+pulse_width',
                            'reset_event':'x<t_period'
                            } # threshold='s==1' also works

        self.L = br.NeuronGroup(n,
                                model=self.input_neuron_model,
                                events=self.input_neuron_events,
                                dt=self.dt)

```

```
        self.L.run_on_event('spike_event',self.input_spike_event_action)
        self.L.run_on_event('resetting',self.input_reset_event_action)

class NeuronGroupBrian:
    '''
    Pack all the components of brian NeuronGroup
    '''
    def __init__(self, neuron_meshgrid:NeuronMeshGrid, bundle_synapse_meshgrid:BundleSynapseMeshGrid, n:int) -> None:
        self.dt = br.defaultclock.dt

        self.model = '''
                    i_per_u : 1
                    j_per_v : 1
                    vmax : volt
                    vmin : volt
                    umax : volt
                    umin : volt
                    Cv : farad
                    Cu : farad
                    Cp : farad
                    Cdp_bundle : farad
                    Cdn_bundle : farad

                    vp_leak : volt
                    vn_leak : volt
                    i_per_v_leak : 1
                    j_per_vd : 1
                    i_per_vm : 1
                    j_per_v_inj : 1

```

```
                    IpT : amp
                    InT : amp
                    IpB = ip_bundle( int(i_per_v_leak*vp_leak/volt), int(j_per_vd*vp_inj/volt) )*amp : amp (constant over dt)
                    InB = in_bundle( int(i_per_v_leak*vn_leak/volt), int(j_per_vd*vn_inj/volt) )*amp : amp (constant over dt)
                    dvp_inj/dt = (IpB - InT)/Cdn_bundle : volt
                    dvn_inj/dt = (IpT - InB)/Cdp_bundle : volt
                    Isyn = i_injection( int(i_per_vm*v/volt), int(j_per_v_inj*vp_inj/volt), int(j_per_v_inj*vn_inj/volt) )*amp : amp (constant over dt)
                    dv/dt = dvdt : volt
                    dvdt=( Cv_current(int(i_per_u*u/volt),int(j_per_v*v/volt))*amp + Isyn )/(Cv+Cp) : amp/farad (constant over dt)
                    du/dt = dudt : volt
                    dudt=Cu_current(int(i_per_u*u/volt),int(j_per_v*v/volt))*amp/(Cu+Cp) : amp/farad (constant over dt)
                    s : 1
                    '''
        self.spike_event_action = '''
                                  s += 1
                                  '''
        self.reset_event_action = '''
                                  s = 0
                                  '''
        self.neuron_events={
                              'vdd_rail':'v>vmax',
```

```
                           'vss_rail':'v<vmin',
                           'udd_rail':'u>umax',
                           'uss_rail':'u<umin',
                           'vp_inj_rail_up':'vp_inj>vmax',
                           'vp_inj_rail_down':'vp_inj<vmin',
                           'vn_inj_rail_up':'vn_inj>vmax',
                           'vn_inj_rail_down':'vn_inj<vmin',
                           't_step':'t>0*second',
                           'spike':'s==1',
                           'spike_event':'v>200*mV',
                           'reset_event':'v<200*mV'
                           }

        self.L = br.NeuronGroup(n,
                    model=self.model,
                    events=self.neuron_events,
                    dt=self.dt
                    )

        self.L.vmax = neuron_meshgrid.vmax*br.volt
        self.L.vmin = neuron_meshgrid.vmin*br.volt
        self.L.umax = neuron_meshgrid.umax*br.volt
        self.L.umin = neuron_meshgrid.umin*br.volt
        self.L.i_per_u = neuron_meshgrid.i_per_y
        self.L.j_per_v = neuron_meshgrid.j_per_x
        self.L.Cv = 50e-15*br.farad
        self.L.Cu = 30e-15*br.farad
        self.L.Cp = 5e-15*br.farad
        self.L.Cdp_bundle = 2e-15*br.farad
        self.L.Cdn_bundle = 2.5e-15*br.farad
        self.L.i_per_v_leak = bundle_synapse_meshgrid.i_per_v_leak
        self.L.j_per_vd = bundle_synapse_meshgrid.j_per_vd
```

```
        self.L.i_per_vm = bundle_synapse_meshgrid.i_per_vm
        self.L.j_per_v_inj = bundle_synapse_meshgrid.j_per_v_inj
        self.L.vp_inj = 300*br.mV # set initial value
        self.L.vn_inj = 0*br.mV # set initial value

        self.L.run_on_event('vdd_rail','v=vmax')
        self.L.run_on_event('vss_rail','v=vmin')
        self.L.run_on_event('udd_rail','u=umax')
        self.L.run_on_event('uss_rail','u=umin')
        self.L.run_on_event('vp_inj_rail_up','vp_inj=vmax')
        self.L.run_on_event('vp_inj_rail_down','vp_inj=vmin')
        self.L.run_on_event('vn_inj_rail_up','vn_inj=vmax')
        self.L.run_on_event('vn_inj_rail_down','vn_inj=vmin')
        self.L.run_on_event('spike_event',self.spike_event_action)
        self.L.run_on_event('reset_event',self.reset_event_action)


class SynapseGroupBrian:
    '''
    Pack all the components of synapse
    '''
    def __init__(self, synapse_meshgrid:SynapseMeshGrid, pre_group:NeuronGroupBrian, post_group:NeuronGroupBrian) -> None:
        self.syn_model = '''
                    i_per_vg_syn : 1
                    j_per_vd_syn : 1

                    vg_p : volt
                    vg_n : volt
```

```
                    Isyn_active_p = syn_active_p( int(i_per_vg_syn*vg_p/volt), int(j_per_vd_syn*vn_inj/volt) )*amp : amp (constant over dt)
                    Isyn_active_n = syn_active_n( int(i_per_vg_syn*vg_n/volt), int(j_per_vd_syn*vp_inj/volt) )*amp : amp (constant over dt)
                    Isyn_inactive_p = syn_inactive_p( int(i_per_vg_syn*vg_p/volt), int(j_per_vd_syn*vn_inj/volt) )*amp : amp (constant over dt)
                    Isyn_inactive_n = syn_inactive_n( int(i_per_vg_syn*vg_n/volt), int(j_per_vd_syn*vp_inj/volt) )*amp : amp (constant over dt)
                    Ip_syn_previous_t_step : amp
                    In_syn_previous_t_step : amp
                    '''
# ==============================================================================
# I += Isyn will keep increasing I for the duration of spike. but this is wrong.
# i need to keep I same as Isyn for the duration of spike.
# with I_syn_previous_t_step variable previous timestep current can be subtracted
# from I before adding new timestep current and thus prevents I from increasing
# ==============================================================================
        self.syn_active_action = '''
                            IpT -= Ip_syn_previous_t_step
                            InT -= In_syn_previous_t_step
                            IpT += Isyn_active_p
                            InT += Isyn_active_n
```

```
                            Ip_syn_previous_t_step = Isyn_active_p
                            In_syn_previous_t_step = Isyn_active_n
                            '''
        self.syn_inactive_action = '''
                            IpT -= Ip_syn_previous_t_step
                            InT -= In_syn_previous_t_step
                            IpT += Isyn_inactive_p
                            InT += Isyn_inactive_n
                            Ip_syn_previous_t_step = Isyn_inactive_p
                            In_syn_previous_t_step = Isyn_inactive_n
                            '''
        self.on_pre_action={
                        'syn_active_path':self.syn_active_action,
                        'syn_inactive_path':self.syn_inactive_action,
                        }
        self.event_assignment={
                        'syn_active_path':'spike_event',
                        'syn_inactive_path':'reset_event',
                        }
        self.S = br.Synapses(pre_group.L, post_group.L,
                self.syn_model,
                on_pre=self.on_pre_action,
                on_event=self.event_assignment
                )
        self.S.connect()
        self.S.i_per_vg_syn = synapse_meshgrid.i_per_vfg
        self.S.j_per_vd_syn = synapse_meshgrid.j_per_vd
        # set drain capacitance of the bundle synapse
```

```
        # += because bais synpase is added seperately
        post_group.L.Cdp_bundle += 0.5e-15*br.farad*pre_group.L.N
        post_group.L.Cdn_bundle += 1.05e-15*br.farad*pre_group.L.N

class SimpleNeuronGroupBrian:
    '''
    Pack all the components of brian NeuronGroup
    '''
    def __init__(self, neuron_meshgrid:NeuronMeshGrid, n:int) -> None:
        self.dt = br.defaultclock.dt

        self.model = '''
                    i_per_u : 1
                    j_per_v : 1
                    vmax : volt
                    vmin : volt
                    umax : volt
                    umin : volt
                    Cv : farad
                    Cu : farad
                    Cp : farad

                    I : amp
                    dv/dt = dvdt : volt
                    dvdt=( Cv_current(int(i_per_u*u/volt),int(j_per_v*v/volt))*amp + I )/(Cv+Cp) : amp/farad (constant over dt)
                    du/dt = dudt : volt
                    dudt=Cu_current(int(i_per_u*u/volt),int(j_per_v*v/volt))*amp/(Cu+Cp/2) : amp/farad (constant over dt)
                    s : 1
```

```
                    '''
        self.spike_event_action = '''
                                  s += 1
                                  '''
        self.reset_event_action = '''
                                  s = 0
                                  '''
        self.neuron_events={
                           'vdd_rail':'v>vmax',
                           'vss_rail':'v<vmin',
                           'udd_rail':'u>umax',
                           'uss_rail':'u<umin',
                           't_step':'t>0*second',
                           'spike':'s==1',
                           'spike_event':'v>200*mV',
                           'reset_event':'v<200*mV'
                           }

        self.L = br.NeuronGroup(n,
                    model=self.model,
                    events=self.neuron_events,
                    dt=self.dt
                    )

        self.L.vmax = neuron_meshgrid.vmax*br.volt
        self.L.vmin = neuron_meshgrid.vmin*br.volt
        self.L.umax = neuron_meshgrid.umax*br.volt
        self.L.umin = neuron_meshgrid.umin*br.volt
        self.L.i_per_u = neuron_meshgrid.i_per_y
        self.L.j_per_v = neuron_meshgrid.j_per_x
        self.L.Cv = 50e-15*br.farad
        self.L.Cu = 30e-15*br.farad
        self.L.Cp = 5e-15*br.farad
```

```


        self.L.run_on_event('vdd_rail','v=vmax')
        self.L.run_on_event('vss_rail','v=vmin')
        self.L.run_on_event('udd_rail','u=umax')
        self.L.run_on_event('uss_rail','u=umin')
        self.L.run_on_event('spike_event',self.spike_event_action
    )
        self.L.run_on_event('reset_event',self.reset_event_action
    )
```