BACKPROPGATION FOR CONTINUOUS THETA NEURON NETWORKS

A Dissertation

by

DAVID FAN

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

MASTER OF SCIENCE

Chair of Committee,	Peng Li
Committee Members,	Yoonsuck Choe
	Arum Han
	Xiaoning Qian
Head of Department,	Miroslav M. Begovic

May 2015

Major Subject: Electrical Engineering

ABSTRACT

The Theta neuron model is a spiking neuron model which, unlike traditional Leaky-Integrate-and-Fire neurons, can model spike latencies, threshold adaptation, bistability of resting and tonic firing states, and more. Previous work on learning rules for networks of theta neurons includes the derivation of a spike-timing based backpropagation algorithm for multilayer feedforward networks. However, this learning rule is only applicable to a fixed number of spikes per neuron, and is unable to take into account the effects of synaptic dynamics. In this thesis a novel backpropagation learning rule for theta neuron networks is derived which incorporates synaptic dynamics, is applicable to changing numbers of spikes per neuron, and does not explicitly depend on spike-timing. The learning rule is successfully applied to XOR, cosine and sinc function mappings, and comparisons between other learning rules for spiking neural networks are made. The algorithm achieves 97.8 percent training performance and 96.7 percent test performance on the Fischer-Iris dataset, which is comparable to other spiking neural network learning rules. The algorithm also achieves 99.0 percent training performance and 99.14 percent test performance on the Wisconsin Breast Cancer dataset, which is better than the compared spiking neural network learning rules.

DEDICATION

To my family and friends, for their unconditional love and support.

ACKNOWLEDGEMENTS

It is a pleasure to thank those who made this thesis possible. I would like to thank my advisor, Dr. Peng Li, for his wise and tactful support and guidance throughout this whole process. I would also like to thank my committee members, Dr. Xiaoning Qian, Dr. Yoonsuck Choe, and Dr. Arum Han, for taking their time to review and critique my work. Further thanks goes to Jimmy Jin for many interesting discussions and help. Lastly, this thesis would not have been possible without the support of my family and friends, whose love and encouragement has enabled me to keep moving in a positive direction.

NOMENCLATURE

AdEx	Adaptive Exponential Integrate and Fire
BP	Backpropagation
CD	Contrastive Divergence
CEC	Constant Error Carousel
DBN	Deep Belief Network
ESN	Echo State Network
LIF	Leaky Integrate and Fire
LSM	Liquid State Machine
LSTM	Short Long Term Memory
NN	(Sigmoidal) Neural Network
QIF	Quadratic Integrate and Fire
RBM	Restricted Boltzmann Machine
ReSuMe	Remote Supervision Method
RNN	Recurrent Neural Network
SI	Sparse Initialization
SNN	Spiking Neural Network
SRM	Spike Response Model

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

Mankind has always had a fascination with building machines that can emulate human characteristics. The ancient Greeks and Chinese envisioned and built clever mechanical contraptions to do the work of men; these contraptions could move on their own, make sounds, eat, tell time, etc. In just the past 50 years, these rudimentary attempts to create artificial intelligence have been completely eclipsed by modern advances. Developments in our understanding of the brain along with the technological achievements of computing have led researchers to develop brain-inspired computational algorithms. In 1943, Warren S. McCulloch and Walter Pitts, a neuroscientist and a logician, formulated a logical analysis of a network of neurons under certain simplifying assumptions[59]. Their analysis demonstrated that neurons could be considered a type of universal Turing machine; this was one of the first steps forward in the creation of artificial neural networks. In 1957 Frank Rosenblatt developed the perceptron algorithm, which trains a layer of weighted sums combined with a threshold, to produce a binary output[77]. This algorithm was implemented in hardware with a series of photocells, potentiometers, and electric motors[6]. These developments created a huge wave of optimism about the possibilities of constructing machines which could think like humans. Unfortunately, in 1969 Marvin Minsky and Seymour Papert published their book *Perceptrons*, wherein they analyzed the perceptron algorithm and showed some critical limitations, including that fact that a single layer of perceptrons is unable to handle linearly non-separable problems (e.g. the XOR function). Furthermore, large-scale implementation of the perceptron algorithm was limited by hardware constraints at that time. These setbacks, along with a variety of other factors, culminated in an "AI winter", during which researchers saw a severe decline in funding which lasted until the 1980s. Despite these setbacks, in 1982 John Hopfield published his work on what are now known as hopfield nets: a recurrent binary neural network that can serve as content-addressable memory. This work as well as others gradually helped to restore credibility to the field. Neural network research again found new popular support in the 1990s, particularly with the popularization of the backpropagation algorithm applied to multilayered sigmoidal networks[79]. The field again took a downward swing as researchers found it difficult to train networks with many layers. Furthermore, the advent of support vector machines and other linear classifier methods created a shift in focus in machine learning towards these new and powerful techniques. The advent of deep learning methods in the early 2000's allowed the training of networks with many hidden layers, and thus neural network research took off again. In short, the history of neural networks, and artificial intelligence in general, has been filled with repeated cycles of overenthusiastic hype followed by disappointed disillusionment. Nevertheless, research in neural networks continues to expand and influence other related lines of research, including spiking neural networks (SNN), deep learning, probabilistic neural network methods, brain structure modeling, and more. This introduction will give a brief summary of some of these topics which are relevant to this work, beginning with a discussion of computation and learning in the human brain, as well as some of the approaches used for investigating this end. Following this, a discussion of sigmoidal neural networks, temporal processing in neural networks, and spiking neural networks will be given. Chapter 2 will present a novel backpropagation algorithm for continuously coupled theta neuron networks. Chapter 3 will examine the application of this algorithm to machine learning benchmarks. Chapter 4 will discuss the results obtained in Chapter 3 and compare them to other spiking neural network algorithms. Finally, Chapter 5 will give a summary of this thesis.

1.1 Learning in the Brain

The human brain contains roughly 86 billion neurons and 84 billion glial cells[2]. With each neuron connecting to on average 7,000 other neurons, this yields an estimate for the number of synapses at about 0.6x10^{15}. This high level of complexity is to be expected, since a brain must store the memories, skills, impressions, and abilities accumulated over a lifetime, which may last up to 4x10^{9} seconds. How the brain accumulates and stores this vast amount of information remains a subject of intense study. Indeed, due in part to these complex unknowns, the brain has sometimes been called the last frontier of human scientific understanding. Measuring instruments for in-vivo studies of the human brain are limited in that either their resolution is too low (e.g. fMRI, Magneto-tomography, EEG), or their bandwidth is too low (e.g. patch clamp recordings, microelectrode recordings, two-photon microscopy, optogenetic recordings). Therefore the investigation of the brain generally takes a two-pronged approach: top-down methods, where the large-scale anatomy and structures of the brain are correlated with functional attributes, and bottom-up methods, where the molecular and cellular processes which comprise the function of individual neurons are modeled and studied. A good example of this two-pronged approach is in the study of the neocortex: the thin outermost layer of cells in the mammalian brain which is associated with sensory processing, motor skills, reasoning, planning, and much more. Investigations into the global function and structure of the neocortex reveal that it is made up of several distinct layers with different cell types and distributions in each layer. The neocortex can be spatially mapped to different sensory and reasoning modalities - for instance, auditory processing occurs in the temporal lobes (on the sides), while visual processing occurs in the occipital lobe (in the back). Despite this specialization, the overall topology and structure of the neocortex is conserved across these domains. This suggests that the neocortex implements a general kind of processing algorithm, and can handle computations which span a wide range of modalities[22]. This hypothesis is further supported by the observation that when one region stops receiving input (e.g. in a loss of vision), it can be "recruited" to enhance other sensory modalities (hearing, tactile senses, etc.)[75]. This kind of general learning and adaptability is one of the characteristics which some artificial intelligence research strives to achieve[99]. Further insight into the function of the neocortex can be gleaned from its granularity and interconnectivity. Its granular structure is defined by cortical columns which range from 300-600μm in diameter, with each column showing distinct functional organization with respect to the processing of sensory input or motor output (for instance, the tonotopic organization of the auditory cortex, or columns in the visual cortex responding to different features)[64]. Although finding an exact definition for what constitutes a column can be difficult, the clustering of both the termination of long-range connections as well as receptive fields for sensory processing support the hypothesis that computation is both locally specialized and massively parallel[44]. This hypothesis is further supported by the interconnectivity of the neocortex. Connections between cortical regions are organized in a small-world topology - i.e. the connection probabilities as a function of distance follow a power-law distribution, with many short/local connections and comparatively sparse (˜11%) region-to-region connectivity[34]. This topology has been shown to be optimal for modularized information processing, where different sensory and motor modalities are processed locally and in parallel[87].

While these top-down approaches to investigating the brain can provide broad insight and general theoretic direction, the elucidation of the precise mechanisms by which learning, memory, and computation occur requires a more detailed investigation of the brain on the cellular and molecular level. These "bottom-up" approaches have produced some powerful insight into how the brain may adapt and learn. Perhaps the best example of this is the discovery of synaptic plasticity: where memory and learning is associated with changes in the connectivity and connection strength between individual neurons. This idea dates back to the first half of the 20th century when Donald Hebb proposed the now well-known Hebbian rule, which can be summarized as "neurons that fire together, wire together"[84]. More recent work has shed light on the original Hebbian hypothesis, leading to the discovery of Spike-Timing-Dependent-Plasticity(STDP), where synapses are strengthened or weakened depending on the relative timing of the firing of the neurons involved[16]. STDP has been shown to be crucial for learning,[18] working memory,[91] spatial navigation,[39] and more. Additionally, many theories of learning and computation in neural circuits revolve around exploiting STDP-like learning rules[56][57][78][72][51]. Despite these advances, however, a complete understanding of learning in the brain is far from complete. Indeed, a prerequisite for understanding how STDP allows for learning is a complete picture of how information is encoded in the firing activity of individual neurons or populations of neurons. This is a difficult and convoluted question. It is generally recognized that the brain can encode information in a diverse number of ways: through average firing rates of individual or populations of neurons,[83] through the timing of individual spikes,[88] through the phase of spikes firing relative to background voltage oscillations,[53] or through correlations between populations of neurons,[70] to name but a few. This diverse range of observed encodings makes it difficult to create a unified theory of learning and computation in the brain. Therefore, computer simulation techniques have been especially useful in trying to understand particular aspects of this complex problem.

1.2 Neuron Models

Neurons can exhibit a wide range of dynamic behaviors resulting in a diverse set of transformations of input spike trains into output spike trains. Some of these behaviors include: tonic spiking, class 1/2 excitability, spike latency, threshold variability, and a bistability of resting and spiking states. Tonic spiking refers to the ability of some neurons to fire continuously in the presence of persistent input[20]. Class 1 excitability is a neuron's ability to fire at a low frequency when input is weak, and to fire at a higher rate in proportion to the input. This is in contrast to class 2 excitable neurons, which have a more binary character, either remaining quiescent or firing at a high frequency[76]. Spike latency refers to the delay that occurs between an input and the resulting spiking event. For most cortical neurons, this latency is inversely proportional to the strength of the input. Threshold variability refers to the fact that a neuron's voltage threshold to firing is not static, but depends on past activity. For example, an excitatory input may not be strong enough to cause a neuron to fire on its own, but an inhibitory input quickly followed by that same excitatory input may cause a spike to occur. Furthermore, some neurons can exhibit two stable modes of behavior, shifting between quiescence and tonic firing. This switch can be caused by carefully timed input[46].

When conducting a computational study involving the modeling of neurons, the researcher must choose a neuron model which not only captures the essential dynamic behaviors relevant to the work, but one which also remains computationally and analytically tractable. This is usually a trade-off. For instance, the classic description of a neuron's membrane voltage dynamics is the Hodgkin-Huxley model, which captures the transmission probabilities of different ions through ion channels in the neuron's cell membrane[43]. The model is comprised of four first order differential equations, and a dozen or so parameters, and directly models the flow of ions as the neuron fires, recovers, and responds to various current input. While the model is excellent in terms of biophysical accuracy and can capture all of the previously mentioned dynamic behaviors and more, this accuracy comes at a severe computational and analytical cost.

An attractive alternative to the Hodgkin-Huxley model is the Izhikevich model which consists of two first order differential equations and four parameters which can be tuned to allow the model to exhibit the same dynamic behaviors as the Hodgkin-Huxley model[45]. The simplicity of the Izhikevich model makes it attractive for simulating large populations of neurons. In contrast to the Hodgkin-Huxley model, the Izhikevich model does not explicitly model the change of membrane voltage as the neuron spikes. Instead it models firing events as a reset of variables when a threshold is reached and creates an impulse output. Another similar model is the Adaptive-Exponential-Integrate-and-Fire (AdEx) model which can exhibit most of the same biologically realistic behaviors and also depends on two first order differential equations with a threshold and reset. The AdEx model can be more easily mapped to biological values such as membrane capacitances[14].

A slightly less complex model is the Quadratic-Integrate-and-Fire (QIF) model, which models the neuron membrane voltage response to inputs with a quadratic differential equation. This produces dynamics which more closely emulates the behavior of real neurons[25]. The QIF model can be mapped onto the related Theta neuron model, which is a focus of this thesis and will be discussed in further detail in Chapter 2.

Yet another neuron model, and one of the simplest, is the Leaky-Integrate-and-Fire (LIF) neuron, which is often used in neuron simulations, as well as for computational purposes. This model integrates the weighted sum of inputs over time, and generates a “spike” - usually an impulse or a decaying exponential - when a threshold is reached. After a spike occurs, the membrane voltage is reset. The spike is then passed onto other neurons, which then convolve this impulse with a kernel function to simulate the dynamics of synapses. The synaptic dynamics can be modeled with different time constants to match different synapse types. The LIF model is easy to simulate, but can be more difficult to analyze due to the many nonlinear constraints in the model, such as thresholding, resetting, etc.

Neuron models are often used to understanding learning and information processing in the brain, in both large-scale simulations and smaller network analyses. Large scale simulations seek to create detailed simulations of large populations of biological neurons in order to examine population dynamics and test various hypotheses. For example, by simulating a simplified model of the entire brain, researchers can try to characterize of the conditions under which large scale abnormal dynamic states may occur, e.g. epilepsy, seizures, etc. [101]. Other studies study how networks of neurons exhibit emergent properties of computation and self-organization. Unfortunately, it is difficult to know ahead of time which properties of neurons should be modeled in order to create the desired global effects. Other studies using neuron models focus on the detailed computational mechanisms of a small population of neurons or even an individual neuron, as in the case of the study of dendritic computation[100][13][92]. These studies try to create a mathematical framework for explaining the information processing ability of even small networks of neurons, choosing neuron models which are analytically tractable with respect to this goal. This thesis will focus on deterministic models of neural computation, which have led to many powerful advancements in machine learning and artificial intelligence, including neural networks, deep learning, and more. Some of these methods will be discussed in more detail in the follow sections.

1.3 Sigmoidal Neural Networks

Sigmoidal neural networks (NN) refer to the classic method of computation in which an alternating series of weighted sums and nonlinear thresholding operations are used to transform input vectors to desired output vectors. The biological inspiration for NNs comes from the idea that a neuron sums up the activity of presynaptic neurons and produces a saturated response. However, NNs can only be considered to very loosely approximate the function of real neurons. Perhaps the most useful comparison is that both NNs and biological neurons are believed to operate via the "connectionist" paradigm - in which the connections between many small units of computation create a computationally powerful system[30]. While learning in biological neurons is difficult to elucidate, learning in NNs is comparatively much easier. In 1973 Paul Werbos in his PhD thesis described a method for training a neural network via gradient descent; this became known as backpropagation[96]. Rumelhart et al.'s paper in 1986 helped to popularize this method, greatly enhancing the field of neural network research[79]. In NNs, each "unit" is described as follows:

$$y_j = \sigma(\sum_i w_{ij} y_i + b_j) \tag{1.1}$$

where y_j is the j^{th} unit's output, y_i are input units, w_{ij} is the weight from the i^{th} unit to the j^{th} unit, b_j is the j^{th} unit's bias, and σ is a smooth, monotonically increasing "squashing" or "activation" function. The logistic function or the hyberbolic tangent function is often used for σ. The backpropagation algorithm relies on the continuously differentiable nature of the squashing function to efficiently calculate the gradient of the error at the output layer.

It has been shown that a feed-forward NN with a single hidden layer can approximate any continuous function; this is known as the universal approximation theorem[52]. Although NNs possess powerful abilities to learn arbitrary function mappings, their training has several issues which can make application difficult. One major issue is the difficulty in training networks with many layers or with recurrent topologies, as in the case of recurrent neural networks (RNN). As error is back-propagated from the output layer back towards the input layer, the influence of the parameters on the output shrinks or grows exponentially[41]. This is known as the "long time lag" problem. In feed forward networks with many hidden layers, this can be partially overcome by carefully setting the learning rates in earlier layers, but in recurrent neural networks this issue cannot be so easily resolved. For RNNs this long time lag problem results in difficulties when training networks to learn relationships between inputs which are separated by many time steps. A discussion of attempts to solve this problem follows in the sections below. A second significant issue is that the error surface is often highly non-convex, with the result being that during the learning process the trajectory of the parameters may get trapped in local minima. A variety of tricks can be useful to overcome this problem, including the use of momentum in the gradient descent and setting the initial weights to some optimal values[23]. For RNNs, initializing weights such that the recurrent network is in an "edge of chaos" state (i.e. such that the hidden-to-hidden weight matrix has a spectral radius of slightly greater than 1.0) allows the network to converge more easily to the global minimum[47].

Another way to overcome the long time lag problem for networks with many hidden layers is the use of Restricted Boltzmann Machines (RBM). An RBM is formed from two sets of nodes, labeled as a "hidden" layer and a "visible" layer, each node is a binary random variable whose probability of being 1 is the logistic sigmoid of the weights sum of its inputs. An RBM is "restricted" in the sense that its topology is limited to each layer having no recurrent connections - furthermore, connections are bidirectional between the hidden and visible layers. In 1985, Ackley, Hinton, and Sejnowski published a fast learning algorithm for RBMs called Contrastive Divergence (CD)[1]. By stacking layers of RBMs together and training them sequentially, deep hierarchical representations of features can be constructed - these are known as Deep Belief Networks (DBN). DBNs have been behind much of the recent boom in interest in “deep learning”, having been applied with great success to handwriting recognition,[40] speech recognition,[24] as well as being the focus of AI research being conducted by Google, Baidu, Microsoft, etc. While DBNs are very good for learning the structure underlying large amounts of data, they are not as well suited for sequential or temporal data. DBNs have been extended to a temporal context by liking up sequences of DBNs, with one DBN for each timestep[90].

DBNs could explain certain aspects of computation in the brain, namely, the learning of deep hierarchical representations of sensory input. Indeed, it has been shown that by modeling neurons as stochastic processes and with certain topological assumptions, networks of spiking neurons can perform probabilistic inference via Markov chain Monte Carlo sampling[15]. This theoretical framework has been used to implement RBMs with LIF neurons, appropriate for implementation on large-scale neuromorphic hardware platforms[73][65]. Interest in related stochastic models of neural computation have seen an abundance of progress in the last few years, with models of Hidden Markov Model learning([48]) and other bayesian computations([66]) being implemented with stochastic spiking neuron models. However, these approaches often sacrifice the biological plausibility of the neuron models in order to perform the desired computations. Indeed, biological neurons are inherently temporal in nature - with diverse dynamical behavior occurring on a wide spectrum of timescales, from the 100-nanosecond reaction times of fish in response to electric signals ([17]) to the tens of minutes of the time course of neuromodulators[71]. Therefore neural computation algorithms which explicitly model time dependencies are of great interest from both an application and a theoretical neuroscience standpoint.

1.4 Temporal Processing in Neural Networks

The processing of temporal data with neural networks takes the form of applying sequential input to the network, and training the network to produce the desired sequential output. For example, in speech recognition the input might be sequential Mel-frequency cepstral coefficients (MFCCs), while the output could be phone probabilities. For such a task, the network must have some memory of recent input and be able to model the temporal structure of this input. Naive attempts at solving this problem with neural networks involved using RNNs trained with backpropagation, since RNNs possess fading memory which scales with the size of the recurrent layer[31]. Unfortunately, due to the long time lag problem, this approach often fails for time lags larger than 10 steps[49].

One workaround for this problem is to give each neuron unit a fading memory. Each unit can be modeled as a leaky integrator, as follows:

$$y_j[n] = \sigma(\sum_{k=1}^{n}(\sum_{i}\lambda^{k-1}w_{ij}y_i[n-k]+b_j)) \tag{1.2}$$

where $0 < \lambda < 1$ is the time constant for the integration. This setup allows backpropagation to directly calculate the influence of data several timesteps away, via direct connections from temporally delayed neurons[89]. However, this approach requires keeping track of the history of each unit $y_j[n], n = 1...N$, which can be computationally impractical. Nevertheless, this approach moves closer towards giving each neural unit a memory of past inputs.

Another notable attempt to overcome this long time lag issue is the Long Short-Term Memory (LSTM)[42]. The basic idea behind LSTM is to modify a neural unit to store a value for an infinite duration by connecting it to itself with a weight of 1.0, and setting its activation function to the identity. The unit refreshes its own value at each time step, creating a memory cell; this is known as a Constant Error Carousel (CEC). The CEC is controlled by multiplicative gates which determine when the CEC should accept input, provide output, and reset its value. A network of LSTM units can be combined with traditional sigmoidal neurons and trained via backpropagation[32]. In theory, the network can then learn when to memorize, forget, or recall certain features across an infinite number of time steps. Beginning with Alex Graves' PhD thesis which applied LSTM to the TIMIT speech recognition dataset,[36] LSTM has been shown to achieve excellent results on a variety of applications including robot localization,[27] robot control,[58] handwriting recognition,[19] and especially speech[35]. Although LSTM networks have been shown to be computationally powerful in their own right, they fall short in terms of biological plausibility[69]. Spiking neural networks, which directly model the spiking nature of real neurons, may be more promising in terms of both providing insight into the biological basis of computation and learning, as well as naturally being capable of temporal processing.

1.5 Spiking Neural Networks

Spiking neural networks have been described as the third generation of neural network models. They attempt to capture the firing behavior of biological neurons for computational use. There are several immediate advantages to this approach. First, SNNs have been shown to have greater computational abilities than traditional NNs[55]. The temporal nature of spiking neuron models allows for robust computation in a continuous time domain. Second, SNNs are desirable in terms of power consumption for hardware implementation. Information is encoded in single pulses which may occur sparsely, resulting in significant savings in power consumption and information transmission. Third, SNNs are close to biological neurons in their function and behavior. This makes them useful for implementing theories of computation in the brain, as well as providing a framework from which to find new insights into neuronal computation.

The most commonly used spiking neuron model is the Leaky-Integrate-and-Fire(LIF) neuron, which takes input spike trains of the form $y(t) = \sum_{t_n<t} \delta(t - t_n)$, where δ is the Dirac delta function, and spikes occur at times t_n. The dynamics are described by:

$$\frac{1}{\tau}\frac{dx_j(t)}{dt} = -x_j(t) + \sum_j w_{ij} y_i(t) \tag{1.3}$$

where τ is the membrane time constant, $x_j(t)$ is the j^{th} neuron's membrane voltage, and $y_i(t)$ is the input spike train from the i^{th} neuron. When $x_j(t)$ is greater than a firing threshold x_{th}, then the neuron outputs a spike and $x_j(t)$ is reset to zero or a resting potential. Instead of a spike train, synaptic dynamics can be modeled instead, and are typically given by the alpha function:

$$y(t) = \alpha \sum_{t_n<t} (e^{-\frac{t-t_n}{\tau_1}} - e^{-\frac{t-t_n}{\tau_2}}) \tag{1.4}$$

where α is a scaling constant.

Unfortunately, training SNNs is difficult. The main difficulty lies in the fact that spikes are generated when the membrane voltage crosses a threshold, and the membrane potential is reset. This discontinuous function cannot be easily differentiated, and the method of gradient descent via backpropagation can no longer be directly applied. Nonetheless, there have been many attempts at deriving learning rules for spiking neural networks. These attempts can be roughly grouped into three broad categories: spike-timing-based methods, rate-based methods, and other model-specific methods.

1.6 Spike-Timing Methods

A popular method for training spiking neural networks is the SpikeProp method, which can be considered a spike-timing-based method[9]. This method calculates the gradient of spike firing times with respect to synaptic weights, and is applicable to multilayer feedforward architectures. It models neurons with the Spike Response Model (SRM), which is a generalization of the LIF model[33]. However, in order for this method to work well, this method must utilize multiple connections between each pair of neurons (e.g. up to 16) with varying delays, in order to allow for a complete range of temporal computation. This is due to the fact that synaptic time constants are very short, removing the possibility for post synaptic potentials (PSPs) to interact across large timescales. Creating a large number of delayed synaptic connections shifts the burden of computation from the neuron model itself to the synapses[61]. SpikeProp has been extended in a variety of ways, including supporting multiple spikes per neuron and recurrent network topologies[12].

Another spike-timing based approach utilizes the Theta Neuron model. The Theta Neuron model is convenient for spike-timing calculations since the time to the neuron spiking is a differentiable function of the times that input spikes arrive at the neuron. With the simplification that synapses are modeled as spikes, a gradient descent rule can be derived which finds the dependence of the timing of the output spikes on the weights of the network[61]. This approach was applied to several machine learning benchmarks, as well as a robotic task[60]. However, there are several drawbacks to this method which will be discussed in further detail in the next chapter.

If one assumes that the initial number of output spikes is equal to the number of target spikes, a simple error function consisting of a sum of the difference in timing between each pair of output and target spikes can be constructed. The gradient with respect to the output layer's weights can then be taken for simple LIF spiking neuron models. Extending this approach to multi-layer networks requires some heuristic assumptions. Xu et al. 2013 analyzed this approach and applied it to several machine learning benchmarks, calling it MuSpiNN[97].

Memmesheimer et al. took a more analytical approach to finding weights in a spiking neural network[62]. They derived a perceptron-like learning rule which is guaranteed to find weights which cause a layer of spiking neurons to fire at the desired times, if such a solution exists. Their learning rule is applicable to a recurrent network; however, it requires that exact target spike times are specified for each neuron. Therefore it is not applicable to networks with hidden layers, where target spike times are not given, and need to be learned. Nevertheless, they showed that this approach can find weights in a recurrent network to generate periodic output, create delay lines, and solve other interesting tasks.

1.7 Rate-Based Methods

Rate-based approaches are also useful for training SNNs. One simple approach is to use SNNs to approximate NNs, after which traditional NN learning methods can be used, e.g. backpropagation. For example, SNNs can be made to encode sigmoidal values with their average firing rates. Smooth nonlinear transfer functions between firing rates of neurons can be constructed, so that traditional NN learning methods can be applied[80]. Using spiking neurons to perform NN computations shows that spiking neurons at least can perform NN computations, but these methods of training spiking neural networks do not take advantage of the increased computational possibilities afforded by using spiking neurons.

More specific rate-based learning methods which do take into account individual spikes can also be constructed. One example of this is the Remote Supervision Method (ReSuMe)[74]. ReSuMe applies a heuristic approximation based on instantaneous firing probabilities to derive a Spike Timing Dependent Plasticity(STDP)-like rule. It has the advantage that it can be applied to a wide range of spiking neuron models, as long as the model is operating in a linear input-to-output firing rate regime. It is also applicable to multilayer and recurrent network topologies, and can handle multiple spikes per neuron[85].

The appeal of rate-based methods is that they are more amenable to simplifying assumptions which allow for taking the gradient on a smooth error surface. It is also possible to construct an error surface which takes into account both spike-timing and the presence/absence of spikes. One can use the Victor & Purpura distance, which is defined as the minimum cost of transforming one spike train into another by creating, removing, or shifting spikes[93]. This distance measure was used to construct learning rules for a single layer of spiking neurons - termed the Chronotron learning rule[29].

1.8 Other Methods

An interesting approach to using SNNs for computation is to avoid training the network weights altogether. This is the fundamental idea behind Liquid State Machines (LSM). Input is applied to a "reservoir" - a recurrent network of spiking neurons with fixed parameters. As the reservoir responds to the input in a dynamic, nonlinear way, an output layer can be trained to map the reservoir's activity to the desired target via linear regression or some other simple method. This approach has the benefit that the reservoir parameters do not need to be adjusted during learning - only one output layer needs to be trained. The parameters are set such that the reservoir is at the "edge of chaos" - i.e. its connections are neither too weak (which results in a fast decay of activity, i.e. a short fading memory), nor too strong (which results in the recurrent feedback overwhelming any input, i.e. a state of chaos)[50]. Reservoir Computing (RC) is the general study of such systems: the spiking neuron implementation of RC is known as LSMs, while the NN implementation is known as Echo State Networks (ESM)[54]. Using a fixed reservoir avoids the difficult problem of training weights in a recurrent network; however, recent advances in training recurrent NN networks have caused Echo state networks to fall by the wayside[23]. Efforts to create learning rules for recurrent SNN networks have similarly overtaken research in LSMs, although the state of research in training recurrent SNNs has yet to mature[81].

Yet another unique approach to training spiking neural networks is through a clever math trick. Bohte and Rombouts observed that a sum of shifted dirac delta functions is the fraction derivative of a sum of shifted power-law kernels, and that one can consider the spike-creating threshold function of a spiking neuron as a fractional derivative[11]. This allows the derivation of backpropagation-style gradient descent learning rules[8]. Unfortunately, this trick requires that input take the form of a sum of power-law kernels, which restricts its usefulness to more general categories of problems.

One final interesting approach to training spiking neurons worth mentioning is by Schrauwen et al. which tries to bridge spiking neural networks with sigmoidal neural networks. By replacing the hard threshold function which creates impulse spikes in a spiking neuron model with a sigmoidal function that outputs an analog value, they showed that the approximated spiking system can be trained with the traditional backpropagation algorithm used for NNs[82]. With this method a trade-off exists between how much "spiking" is being modeled by the network and how effectively the NN backpropagation learning can be applied.

2. CONTINUOUS BACKPROPAGATION FOR THETA NEURON NETWORKS

2.1 Theta Neuron Networks

The Theta neuron model is a specific implementation of the class of quadratic spiking neuron models, which are described by the differential equation:

$$\frac{du}{dt} = u^2 + I \tag{2.1}$$

where u is the state variable and I is the input. The solution is the tangent function; therefore a spike is said to occur when the state u blows up to infinity. The following change of variables yields the theta neuron model:[25]

$$u(t) = tan\left(\frac{\theta(t)}{2}\right). \tag{2.2}$$

The quadratic spiking model, and by extension the theta neuron model has the property that the neuron's response to incoming synaptic input depends on the internal state of the neuron. This creates the important property that spikes incoming to a neuron at different times will produce different effects. This temporal nonlinearity in the neuron's behavior could allow for more robust and interesting temporal computation to occur. Indeed, the theta neuron exhibits several characteristics found in real neurons, including first-spike latency (where the latency between an input current and a spike event depends on the strength of the input), tonic firing (where the neuron can fire repeatedly in response to constant positive input, and activity-dependent thresholding (where the amount of input current needed to make the neuron fire depends on the recent firing history of the neuron).

The theta neuron model is described by

$$\tau \frac{d\theta_j(t)}{dt} = (1 - \cos\theta_j(t)) + \alpha I_j(t)(1 + \cos\theta_j(t)) \tag{2.3}$$

where $\theta_j(t) \in [0, 2\pi]$ is the j^{th} neuron's phase at time t, τ is the time constant, α is a scaling constant, and $I_j(t)$ is the input current, which is given by a sum of a baseline current I_0 and weighted input spike trains:

$$I_j(t) = I_0 + \sum_{i=1}^{N} w_{ij} S_i(t). \tag{2.4}$$

The neurons $i = 1 \ldots N$ provide spike trains to neuron j; the spike trains are given by $S_i(t) = \sum_{t'_i<t} \delta(t - t'_i)$, which are then weighted by w_{ij}. A theta neuron produces a spike whenever its phase equals π (i.e. $\forall t = t'_j, \theta_j(t) = \pi$). When $I_0 > 0$, the neuron exhibits tonic firing in the absence of other input, with its phase cycling around the phase circle, spiking at periodic intervals. When $I_0 < 0$, the phase circle has two fixed points, one attracting and one repelling (Figure 2.1). In the absence of input, the phase settles to the attracting fixed point. If positively weighted input spikes cause the phase to increase enough to cross the repelling fixed point, then the neuron will "fire" when its phase eventually crosses π. Thus, the repelling fixed point is equivalent to a firing threshold. After firing, the neuron exhibits relative refractoriness, during which a much larger excitatory input is required to cause the neuron to quickly fire again. In the figure as well as in the rest of this work, the phase is shifted by a constant $\cos^{-1}(\frac{\alpha I_0+1}{\alpha I_0-1})$ to set the attracting fixed point to be at $\theta = 0$. This shift is convenient for implementation of the learning rule, as will be shown.

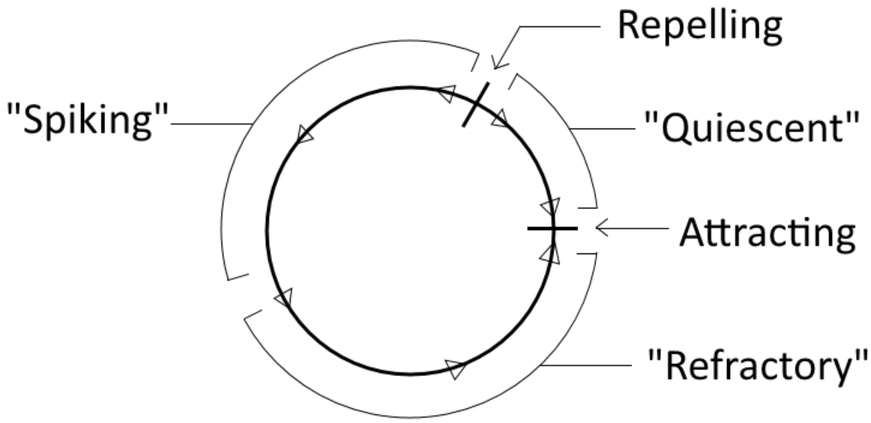

Figure 2.1: Theta neuron phase circle. Shown for when the baseline current I_0 is less than zero, resulting in two fixed points. The attracting fixed point is equivalent to the resting membrane potential, while the repelling fixed point is equivalent to the firing threshold.

The theta neuron model has the attractive property that it does not need to reset the state variables after firing a spike, in contrast to many spiking neuron models. This allows for the derivative to be directly taken when deriving gradient descent learning rules. Indeed, such a method was first developed for a single-layer of neurons ([94]) and was applied to a robotic application[60]. The single layer gradient descent rule was later extended to multiple layers by explicitly calculating the dependence of each spike time on the parameters,[61] as follows: the change of phase of a neuron before and after a spike is given by

$$\theta_j^+ = 2tan^{-1}(\alpha w_{ij} + tan(\frac{\theta_j^-}{2})). \tag{2.5}$$

where θ_j^- is the phase before the input spike, and θ_j^+ is the phase after the input spike. The effect of the input spike on the phase is dependent on the current phase and the weight from the i^{th} neuron to the j^{th} neuron. Furthermore, the remaining time until the neuron fires a spike itself is a function of its current phase:

$$F(t) = \int_{\theta(t)}^{\pi} \frac{d\theta}{(1-\cos\theta) + \alpha I(t)(1+\cos\theta)}. \tag{2.6}$$

If there are input spikes, the integral can be broken up into the sum of several pieces with the integration bounds of each piece determined by equation 2.5. McKennoch et al. applied this learning rule to multilayer feedforward networks and demonstrated their superior performance on a variety of machine learning benchmarks when compared to other spiking neural network algorithms[61]. However, this learning rule is only applicable to networks where spikes are transmitted via impulses, and therefore neglects the modeling of synapses and synaptic currents. Furthermore, this approach assumes a fixed number of spikes during the course of learning.

In this thesis, a gradient descent learning rule for theta neuron networks which approximates synaptic interactions and can handle changing numbers of spikes per neuron as the weights are changed is derived.

2.2 Gradient Descent Backpropagation Algorithm

2.2.1 Continuous Spiking Approximation

Commonly in spiking neural network models, several significant discontinuities exist which impede the use of gradient descent techniques. First, spikes modeled as impulse functions are obviously discontinuous. Second, spikes are transmitted only when the membrane voltage crosses a threshold. Therefore the output is a hard-threshold function of the membrane voltage. Finally, in many models the membrane voltage is reset to some baseline voltage immediately after a spike; this reset is discontinuous as well. The theta neuron model avoids this last type of discontinuity because after a spike is fired, the phase naturally will approach the attracting fixed point. However, spikes still are modeled as impulses, being generated when the phase crosses π. Many spiking neural network models attempt to mitigate the discontinuities when modeling spikes as impulses by instead modeling spikes as synaptic currents. Synaptic models generally consist of convolving spikes with a first or second order decaying exponential function. Sometimes a reset of the membrane voltage is modeled as a strong hyperpolarizing synaptic current as well. Nevertheless, these approaches do not avoid the hard-threshold issue for creating spikes. Indeed, a spike is instantly created whenever the threshold is crossed. The other extreme is to use a soft threshold with no spiking dynamics, as found in NNs. To avoid these problems but still model realistic spiking behavior, the phases of theta neurons can be coupled through a kernel, as follows:

$$\kappa(\theta_i(t)) = K\left(\exp\left[-\frac{1}{2}\left(\frac{\cos(\frac{\theta_i(t)}{2})}{\sigma}\right)^2\right] - \exp(-\frac{1}{4\sigma^2})\right). \tag{2.7}$$

where σ controls the spread of the function and K is a normalizing constant such that $\int_0^{2\pi}\kappa(\theta)\mathrm{d}\theta = 1$ (Figure 2.2). Also, $\lim_{\sigma\to 0^+}\kappa(\theta_i(t)) = \sum_{t_k<t}\delta(t_k)$, where $\theta_i(t_k) = \pi$. This kernel value can then be directly fed to a downstream neuron j, after being scaled by a weight w_{ij}:

$$\tau\frac{d\theta_j(t)}{dt} = (1 - \cos\theta_j(t)) + \alpha I_j(t)(1 + \cos\theta_j(t)) \tag{2.8}$$

$$I_j(t) = I_0 + \sum_{i=1}^{N} w_{ij}\kappa(\theta_i(t)). \tag{2.9}$$

This smooth approximation of spikes can be used to derive learning rules for the network using gradient descent, while still retaining spiking dynamics. The kernel function acts as an approximation of synaptic dynamics, and there are no discontinuities introduced by thresholding or resets. Additionally, the direct coupling of subthreshold dynamics can be interpreted as a type of gap junction connectivity[21].

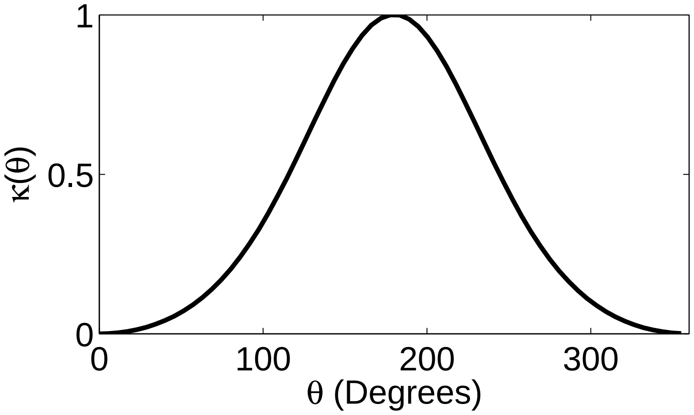

Figure 2.2: Plot of kernel as a function of phase. The peak occurs when $\theta = \pi$.

The system can be discretized via Euler's method:

$$\theta_j[n] = \theta_j[n-1] + \frac{\Delta t}{\tau}\left[(1-\cos\theta_j[n-1]) + \alpha I_j[n-1](1+\cos\theta_j[n-1])\right] \tag{2.10}$$

$$I_j[n-1] = I_0 + \sum_{i=1}^{N} w_{ij}\kappa(\theta_i[n-1])). \tag{2.11}$$

To gain a more intuitive picture of the dynamics, a single theta neuron receiving input and producing output was simulated (Figure 2.3). This demonstrates both the synaptic currents roughly modeled by the kernel and the biologically realistic properties of the theta model. The particular choice of the kernel (an exponential of a cosine) is motivated by the desire to first map the phase (which may take values in $[0, 2\pi]$) to the interval $[-1, 1]$. A square and exponential can then be applied to obtain a Gaussian-like shape, after which $\kappa(0)$ is subtracted to set the output to a maximum when $\theta = \pi$ and 0 when $\theta = 0$. Note that the shift of θ by the constant $\cos^{-1}(\frac{\alpha I_0+1}{\alpha I_0-1})$ as mentioned earlier allows the output to be 0 when θ is at the attracting fixed point. This prevents output being sent to downstream neurons when the neuron is in a resting phase.

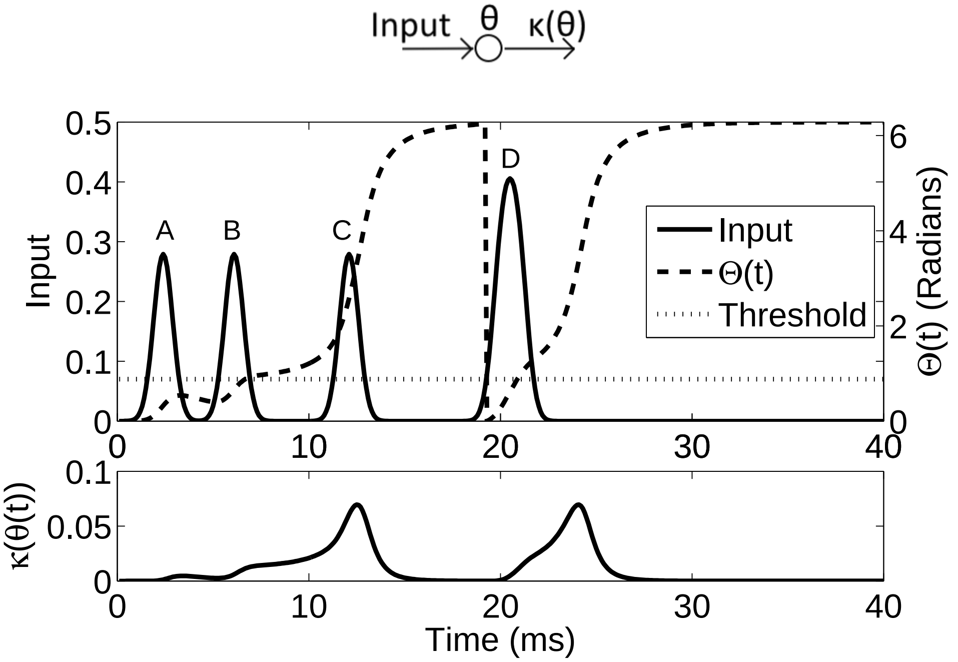

Figure 2.3: Response of single theta neuron with kernel. Top: Input is applied to a single theta neuron, and the output is a function of the neuron's phase. Middle: Plots of input, theta nueron phase, and the threshold (i.e. repelling fixed point). At A a kernelized input is applied. This input alone is not large enough to cause the phase to cross the threshold, and the phase begins to return to 0. At B another input spike is recieved, this time the phase is moved past the threshold, and the neuron begins to fire. At C the neuron is fully spiking, as evidenced by the output kernel in the lower panel. As the phase crosses π, the output reaches a maximum. Input is applied, but this does not affect the dynamics very much, since the phase is current in the "spiking" regime. The phase wraps around to $2\pi = 0$. Finally, at D, a larger input is applied which is enough to cause the neuron to fire again. Bottom: Kernelized θ, which is the output of the neuron. This output will be fed to downstream neurons.

2.2.2 Network Topology

Learning rules are derived for a three-layered network, with input, hidden, and output layers. This could be extended to networks with additional layers. A feed-forward topology is assumed here (as in Figure 2.4), see the Appendix for derivation of learning rules for a recurrent topology. Let $\mathcal{I}$, $\mathcal{H}$, and $\mathcal{O}$ denote the set of input, hidden, and output neurons indexed over i,j, and k, respectively, with $|\mathcal{I}| = M$, $|\mathcal{H}| = N$, and $|\mathcal{O}| = P$. Input neurons $\mathcal{I}$ are connected to hidden neurons $\mathcal{H}$ via weights w_{ij} in a feed-forward manner and hidden layer neurons are connected to the output layer $\mathcal{O}$ via weights w_{jk}, again in a feed-forward manner. Input and target spike trains are denoted by $X_i[n]$ and $T_k[n]$ respectively. The error function is given by the sum squared error of the output layer and the targets:

$$E[n] = \frac{1}{2}\sum_{k-1}^{P}(\kappa(\theta_k[n]) - T_k[n])^2 \tag{2.12}$$

$T_k[n] = 1$ if a spike is desired during the n^{th} time bin, and $T_k[n] = 0$ otherwise. Similarly, the input spike trains are 0 or 1 for each $X_i[n]$.

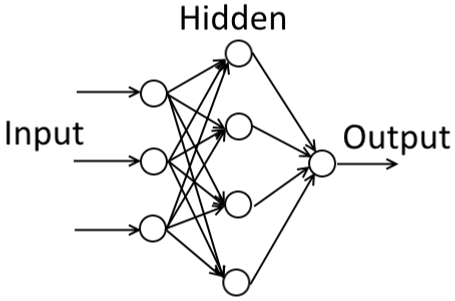

Figure 2.4: Feed-forward neural network consisting of an input, hidden, and output layer.

2.2.3 Forward Pass

Denote $\theta_k[n]$ as θ_k^n for convenience. The forward pass for the input neurons $i \in \mathcal{I}$, hidden neurons $j, l \in \mathcal{H}$, and output neurons $k \in \mathcal{O}$ are, respectively:

$$\theta_i^n = \theta_i^{n-1} + \frac{\Delta t}{\tau}\left[(1-\cos\theta_i^{n-1}) + \alpha(1+\cos\theta_i^{n-1})(I_0 + X_i[n-1])\right] \tag{2.13}$$

$$\theta_j^n = \theta_j^{n-1} + \frac{\Delta t}{\tau}\left[(1-\cos\theta_j^{n-1}) + \alpha(1+\cos\theta_j^{n-1})\left(I_0 + \sum_{i\in\mathcal{I}} w_{ij}\kappa(\theta_i^{n-1})\right)\right] \tag{2.14}$$

$$\theta_k^n = \theta_k^{n-1} + \frac{\Delta t}{\tau}\left[(1-\cos\theta_k^{n-1}) + \alpha(1+\cos\theta_k^{n-1})\left(I_0 + \sum_{j\in\mathcal{H}} w_{jk}\kappa(\theta_j^{n-1})\right)\right]. \tag{2.15}$$

2.2.4 Backward Pass

The dependence of the error on the weights w_{ij} and w_{jk} needs to be found. Denote $\frac{\partial\theta_k[n]}{\partial w_{jk}}$ as $\delta_{k,jk}^n$ for convenience. Starting with hidden-to-output layer weights w_{jk}, we have:

$$\frac{\partial E[n]}{\partial w_{jk}} = (\kappa(\theta_k^n) - T_k[n])\kappa'(\theta_k^n)\delta_{k,jk}^n \tag{2.16}$$

where $\kappa'(\theta)$ is the derivative with respect to θ:

$$\kappa'(\theta_m(t)) = K \exp\left[-\frac{1}{2}\left(\frac{\cos(\frac{\theta_m(t)}{2})}{\sigma}\right)^2\right]\left(\frac{\sin(\theta_m(t))}{4\sigma^2}\right). \tag{2.17}$$

Continuing the derivative:

$$\begin{aligned}
\delta^n_{k,jk} = \delta^{n-1}_{k,jk} + \frac{\Delta t}{\tau}\Bigg[&\delta^{n-1}_{k,jk}\sin\theta^{n-1}_k + \alpha(1+\cos\theta^{n-1}_k)\kappa(\theta^{n-1}_j) & (2.18)\\
&-\alpha\delta^{n-1}_{k,jk}\sin\theta^{n-1}_k\left(I_0 + \sum_{j\in\mathcal{H}} w_{jk}\kappa(\theta^{n-1}_j)\right)\Bigg] & (2.19)
\end{aligned}$$

The backward pass for input layer weights w_{ij} follows similarly:

$$\frac{\partial E[n]}{\partial w_{ij}} = \sum_{k\in\mathcal{O}}(\kappa(\theta^n_k) - T_k[n])\kappa'(\theta^n_k)\delta^n_{k,ij} \tag{2.20}$$

$$\delta^n_{k,ij} = \delta^{n-1}_{k,ij} + \frac{\Delta t}{\tau}\left[\delta^{n-1}_{k,ij}\sin\theta^{n-1}_k - \alpha\delta^{n-1}_{k,ij}\sin\theta^{n-1}_k(I_0 + \sum_{j\in\mathcal{H}} w_{jk}\kappa(\theta^{n-1}_h))\right] \tag{2.21}$$

$$\begin{aligned}
\delta^{n-1}_{j,ij} = \delta^{n-2}_{j,ij} + \frac{\Delta t}{\tau}\Bigg[&\delta^{n-2}_{j,ij}\sin\theta^{n-2}_j + \alpha(1+\cos\theta^{n-2}_j)\kappa(\theta^{n-2}_i) & (2.22)\\
&-\alpha\delta^{n-2}_{j,ij}\sin\theta^{n-2}_j(I_0 + X_i[n-2])\Bigg] & (2.23)
\end{aligned}$$

It is also possible to find the gradient with respect to the baseline current I_0 (see Appendix). The learning rules can be summarized as:

$$\Delta w_{ij} = -\eta_h \sum_{n=0}^{\mathcal{N}} \frac{\partial E[n]}{\partial w_{ij}} \tag{2.24}$$

$$\Delta w_{jk} = -\eta_o \sum_{n=0}^{\mathcal{N}} \frac{\partial E[n]}{\partial w_{jk}} \tag{2.25}$$

where $\mathcal{N}$ is the number of time steps and η_h, η_o are the learning rates for each layer of connections, respectively.

During simulation we need to keep track of the errors δ, which are updated at each simulation time step. Each neuron keeps track of the errors which describe the effect of a weight change on that neuron's activity. The errors are then passed on to downstream neurons. Finally, at the output layer, the dependence of the total error on the weights can be calculated, and the appropriate weight updates are then applied.

2.2.5 Nesterov-Style Momentum

There are several heuristic methods used for training recurrent sigmoidal neural networks which can be applied to the current setup. Recurrent neural networks suffer from the long time lag problem, which may also be a problem here, since both algorithms backpropagate errors back in time. Recent work by Hinton et. al. suggests several key methods for training recurrent neural networks, one of which is the use of Nesterov-style momentum[23]. Classical momentum takes the form of:

$$\mathbf{v}_{t+1} = \mu \mathbf{v}_t - \eta \bigtriangledown E(\mathbf{w}_t) \tag{2.26}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1} \tag{2.27}$$

where μ is the momentum coefficient. At each weight update step t, the gradient is calculated at the current weight vector $\mathbf{w}_t$, and added to the momentum vector $\mathbf{v}_t$. The new momentum term is then used to update the weight vector. In contrast, Nesterov-style momentum is applied as follows:

$$\mathbf{v}_{t+1} = \mu\mathbf{v}_t - \eta \bigtriangledown E(\mathbf{w}_t + \mu\mathbf{v}_t) \tag{2.28}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1} \tag{2.29}$$

The difference between classical momentum and Nesterov-style momentum is that instead of calculating the gradient at the current weight vector, Nesterov-style momentum first performs a partial update to the weight vector by adding $\mu\mathbf{v}_t$, then calculating the gradient at this new location (Figure 2.5). This allows the gradient descent to be more responsive to changes in the objective function as the weights are updated. For instance, consider the example where the weight vector is poised to enter a narrow valley in the error function. Classical momentum would calculate the gradient at the point before entering the valley, pushing the weights past the valley and onto the other side. Only at the next update would the reversed gradient cause the weight trajectory to slow down. In contrast, Nesterov-style momentum first calculates a partial update which crosses the valley, then calculates the gradient at the new point. This gradient points in the opposite direction, and the weight trajectory is immediately slowed. This subtle difference compounds across weight updates and allows for faster convergence, particularly for when high values of μ are desired to overcome large initial plateaus or abundant local minima. Indeed, Sutskever et al. (2013) show that Nesterov-style uses a smaller effective momentum for directions with high curvature relative to classical momentum. This prevents oscillations and allows the use of a large μ[23].

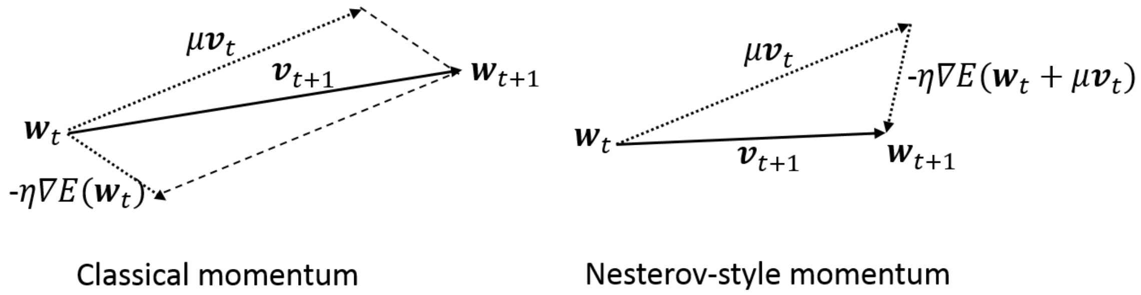

Figure 2.5: Comparison of classical momentum and Nesterov-style momentum. In classical momentum, weight updates are performed by adding the vectors $-\eta \bigtriangledown E(\mathbf{w}_t)$, the gradient, and $\mu\mathbf{v}_t$, the momentum. The next epoch's momentum vector is simply the difference of the new weight and the old weight. In Nesterov-style momentum, weight updates are performed by adding the momentum vector $\mu\mathbf{v}_t$ first, then adding the gradient calculated at the new point. The next epoch's momentum vector is calculated in the same way.

The partial update in Nesterov-style momentum is not optimal for implementation with the backpropagation calculations because the partial update of the weight vector demands resimulating the network with the partially updated weights, calculating the gradient, then simulating again at the final updated weight vector. A simple change of variables makes the algorithm easier to implement. Let $\tilde{\mathbf{w}}_t = \mathbf{w}_t + \mu\mathbf{v}_t$. Then after some calculation we obtain:

$$\mathbf{v}_{t+1} = \mu\mathbf{v}_t - \eta \bigtriangledown E(\tilde{\mathbf{w}}_t) \tag{2.30}$$

$$\tilde{\mathbf{w}}_{t+1} = \tilde{\mathbf{w}}_t + \mu\mathbf{v}_{t+1} - \eta \bigtriangledown E(\tilde{\mathbf{w}}_t) \tag{2.31}$$

Now the gradient is taken at the current weight vector and the partial weight update is no longer explicitly necessary.

Weight initialization is another important consideration in the convergence of RNNs. For the hidden-to-hidden layer connections, setting the initial weights such that that the hidden layer is near the "edge of chaos" aids in finding good solutions. Input-to-hidden layer weight initialization is an important consideration as well. Hinton et al. suggests setting initial weights small enough to not cause the activity in the hidden layer to saturate, but large enough to speed learning. Weights are chosen randomly from a Gaussian distribution with a mean and variance that is chosen through experimentation. Generally, the mean is chosen to be positive and the variance is chosen to be large enough such that there will be a significant fraction of negative weights. A few more tricks are employed: learning rates for each weight layer are adjusted independently to compensate for errors shrinking as they backpropagate towards the input layer, and weight updates are performed in an online manner, where weights are updated after the presentation of each data sample, rather than after presenting all data in the training set. This helps to speed convergence.

All experiments used the constants outlined in Table 2.1.

Constant Name	*Symbol*	*Value*
Time constant	τ	20ms
Input scaling	α	1
Baseline current	I_0	-0.005
Output layer learning rate	η_o	Varies
Hidden layer learning rate	η_h	$1.0e3 * \eta_o$
Kernel Spread	σ	2.0
Timestep	Δt	0.5ms

Table 2.1: Table of constants used for simulation and experiments.

3. IMPLEMENTATION AND MACHINE LEARNING BENCHMARKS

3.1 Demonstration of Learning Rule

A simple two-weight network was used to demonstrate the efficacy of the learning rule. The network consists of one input neuron, one hidden neuron, and one output neuron. The input neuron fires one spike, and the output neuron was trained to fire at a desired time by setting the target value to 1 at the desired time and 0 otherwise. In order to compare the use of momentum versus no momentum, the target values are set such that the error surface contains a local minimum. Figure 3.1 examines the result, and demonstrates successful implementation for this simple example.

One advantage of using a smoothed kernel is that spikes can be created and added naturally during gradient descent. To examine in more detail what occurs when a spike is created by the learning algorithm, the following setup was used: One output neuron was trained to fire in response to 100 input neurons. Each input neuron fired a spike in succession with a short delay between each firing event. The target spike time for the output neuron was set to occur when the 70th input neuron fired. The weights were initialized to the same small value, $w_{init} = 0.01$, so that each input neuron contributed equally and the output neuron did not fire. After 100 learning epochs, the output neuron successfully created a spike and placed it to occur at the correct target time.

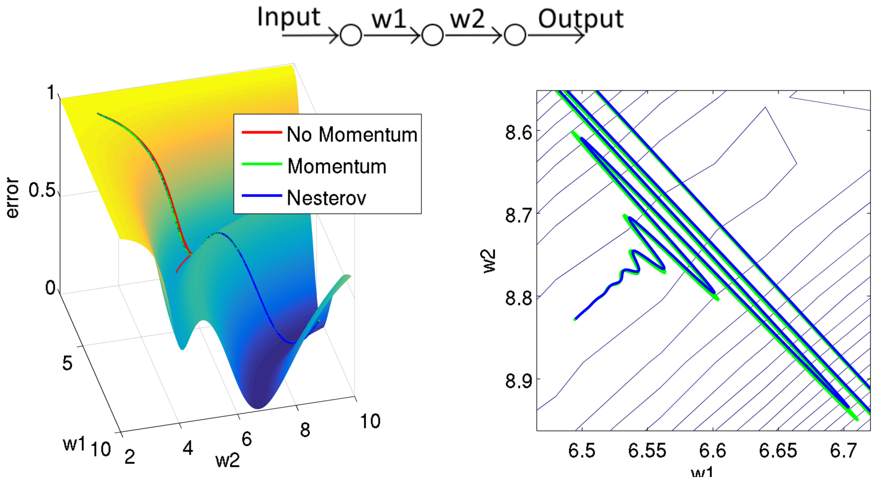

Figure 3.1: Demonstration of learning rule for network with two weights. Top: Topology of network (one input, one hidden, and one output neuron), with weights labeled. Left panel: Weight trajectory on error surface, comparing learning with and without momentum. The trajectory which does not use momentum gets stuck in the local minimum, while the trajectories which use momentum does not. Right: Zoomed in contour plot of the weight trajectory. After a few oscillations the weights converge, with Nesterov-style momentum converging slightly faster and oscillating less than classic momentum.

The change of the 100 input weights across the training epochs reveals how the learning algorithm is deciding to create and shift a spike (Figure 3.2). Before the spike has been created, weights are updated slowly, with weights corresponding to input neurons which fire before the target firing time increasing, and weights corresponding to input neurons which fire after the target time decreasing. This trend eventually causes the output neuron to fire in response to input spikes that come before the target spike time. After the new spike is created there is a sudden change in the trajectories of the weights as they rapidly change to shift the newly created spike to the target time, since the newly created output spike does not initially occur at the target time, but later. This delayed spike creation is due to the “spike latency” dynamics of the theta neuron model; it takes some time for the phase of the output neuron to move around the phase circle beginning from the repelling fixed point. As the output spike is shifted, oscillations occurring around 30-50 learning epochs are a result of overshooting the target firing time until it settles at the correct time. After learning has stabilized, the main weights which contribute to the neuron are those which correspond to inputs that fire in a relatively narrow time window before the target spike time (pink traces). These inputs correspond to those which maximally contribute to causing the output neuron to fire at the target time, taking into account the theta neuron dynamics.

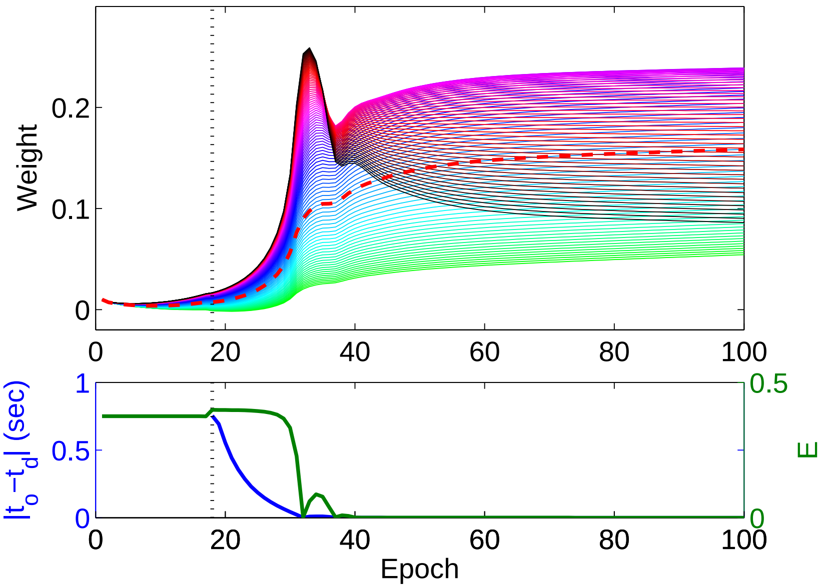

Figure 3.2: Examination of weight changes as the creation of a new spike occurs. Top: 100 weights are plotted as they change over the course of 100 learning epochs. Colors indicate the which input neurons the weights correspond to - black traces correspond to input neurons which fire earliest, followed by pink, blue, cyan, and lastly green. The red dotted line indicates the weight corresponding to the input neuron which fires exactly at the target time. The neuron begins to fire a spike after about 18 learning epochs, marked by the dotted black line. Bottom: Two measures of error over the course of learning. The left axis corresponds to the timing error of the newly created spike - the absolute difference between the output spike time t_o and the target spike time t_d. The right axis corresponds to the raw SSE value used in the backpropagation algorithm.

3.2 XOR Task

The XOR task is commonly used as a test for classification algorithms since it is a simple linearly non-separable problem. If an algorithm can solve the XOR task then presumably it can scale to other more difficult linearly non-separable tasks. In order to apply the XOR problem to theta neurons, the binary values of 0 and 1 need to be encoded as spike times. This is done by having input and output spikes fire early or late, which correspond to 1 or 0 respectively. Three input neurons were used: two neurons encoded the binary inputs of 0 or 1 and the remaining neuron marked the beginning of the trial by firing a single spike. Input neurons fire at 20ms or 40ms. The output layer has one neuron, which was trained to fire at either 60m or 100ms depending on the input, while the hidden layer had 4 neurons. Weights were randomly initialized with a Gaussian distribution with mean 2.5 and variance 2.0. Momentum was set to 0.9, with the learning rate of the output layer set to $\eta_o = 1.0e3$ After 220 epochs, the algorithm converged to the solution. (Figure 3.3)

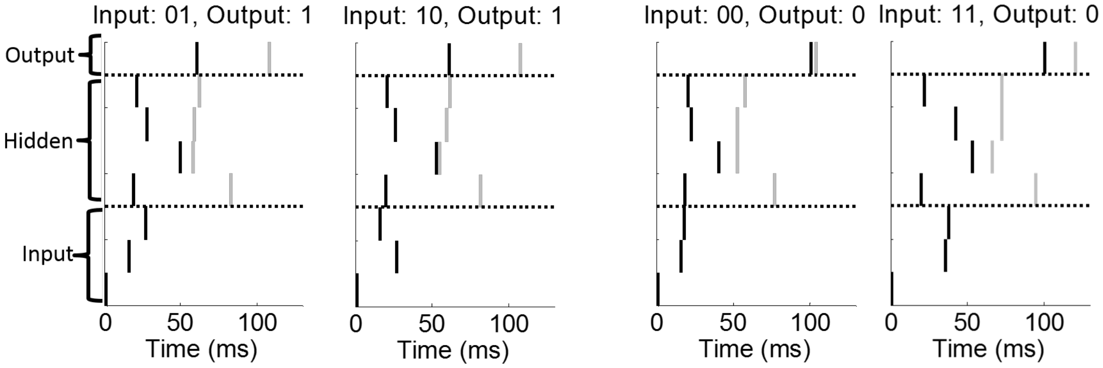

Figure 3.3: Comparison of network spiking activity for the XOR task before and after learning. Four panels correspond to each of the four possible inputs to the XOR function: 01,10,00, and 11. Spikes occur at the times indicated by the rasters. Grey spikes show activity before learning, black spikes are after learning. Before learning has occurred, the output spikes occur at roughly the same time for each input, indicating that the network has not yet learned to discriminate the inputs. After learning, the output spike fires early for input 01 and 10, and late for input 00 and 11. The hidden layer spike times change dramatically after learning.

3.3 Cosine and Sinc Tasks

To test a small continuous theta neuron network's ability to approximate nonlinear functions, regression tasks on data generated by the cosine and sinc functions were used. To encode real-valued inputs as spikes, the value was scaled and shifted, then encoded as the latency to the first spike firing time. Output values were similarly encoded as a single target firing time for each input. For the cosine task, a network with 2 input neurons, 5 hidden neurons, and one output neuron was used. One input neuron was used to mark the start of a trial by firing once, while the other input neuron fired at a later time proportional to the desired input value. The output neuron is trained to fire at a time proportional to the value of the cosine function such that the earliest possible target time came immediately after the latest possible input spike. 50 data points, chosen randomly from a uniform distribution on $[0, 2\pi]$ were used. Weights were randomly initialized with a Gaussian distribution with mean 12.0 and variance 6.0. Momentum was set to 0.99, with $\eta_o = 1.0e3$. After 1433 epochs, the algorithm converged to the solution (Figure 3.4). A closer examination of the activity of the hidden layer after training reveals that after learning, some hidden units learned to fire multiple spikes or to not fire at all in response to different inputs (Figure 3.5). This suggests that the learning algorithm can take advantage of a changing number of spikes in the hidden layer to achieve more varied computations.

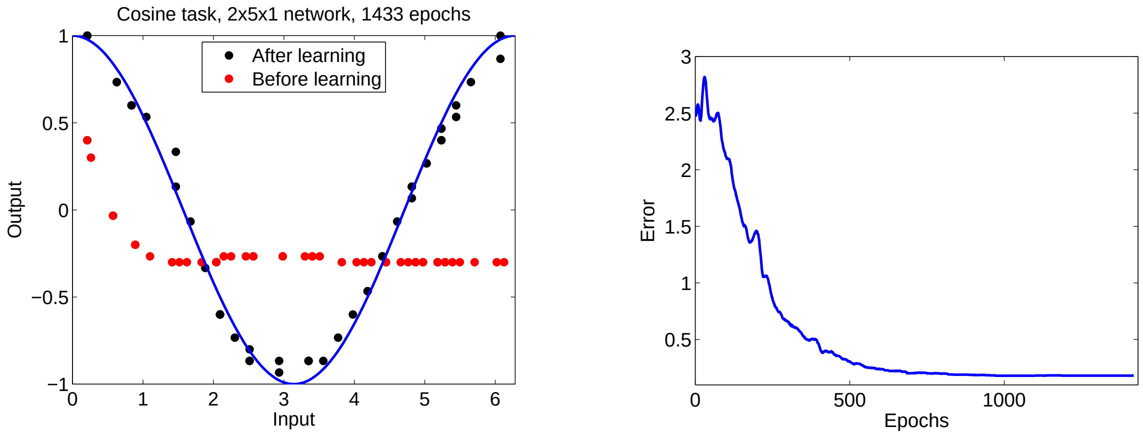

Figure 3.4: Results for Cosine regression task. Left: Comparison of regression before and after learning. Right: Error as a function of learning epochs. The algorithm quickly converges to the solution.

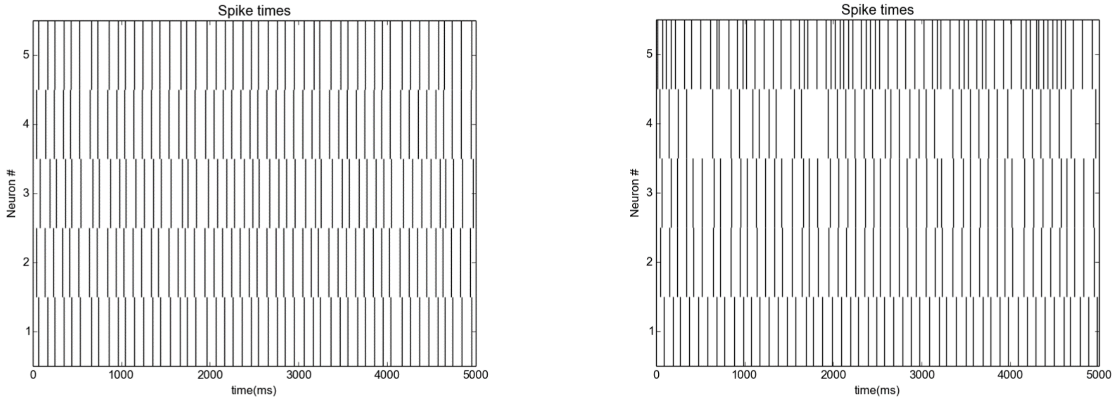

Figure 3.5: Hidden neuron activity before and after learning for the Cosine regression task. Left: Before learning, five hidden neurons each spike once in response to each of the 50 input data samples. Right: After learning, neurons 1-3 still spike once in response to each input, but neuron 4 has learned to not produce a spike for certain inputs, while neuron 5 produces multiple spikes for some inputs. This varied activity is summed and combined to cause the output neuron to fire at the desired times, enabling the entire network to accurately map the cosine function.

The sinc function was also used to provide more of a challenge. For the sinc test, a network topology of two input neurons, 11 hidden neurons, and one output neuron was used. 150 data samples from the interval $[-3\pi, 3\pi]$ were taken, the other parameters were the same as the cosine task. After 2200 epochs the learning rule converged (Figure 3.6). Additional networks with different numbers of hidden units were tested: from 5 to 20, with 11 hidden units yielding the best performance.

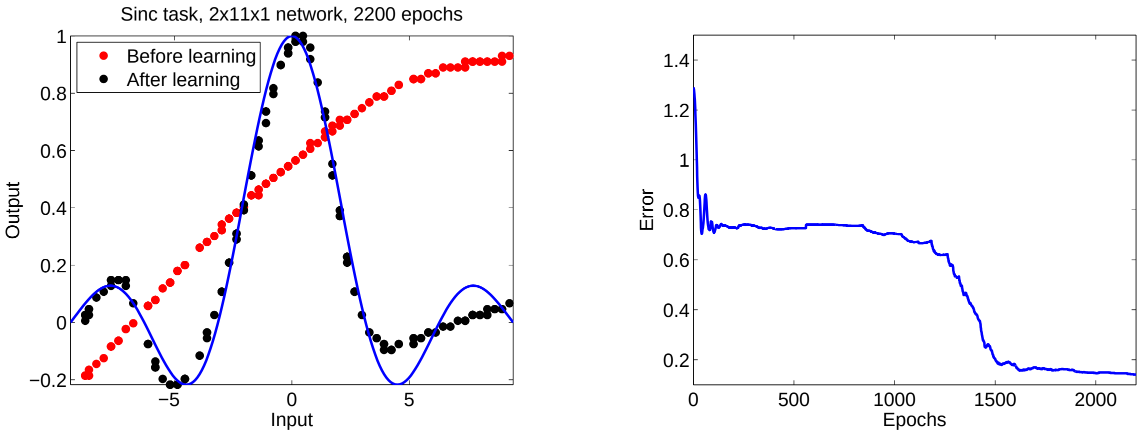

Figure 3.6: Results for Sinc regression task. Left: Comparison of regression before and after learning. The right-hand side tail fails to accurately map the curves, instead converging to a sloped line. Right: Error as a function of learning epochs. The error plateaus for a long period before rapidly decreasing. This plateau corresponds to the local minimum when every input is mapped to the mean of the sinc function. After a rapid decline, the error slowly approaches a minimum.

3.4 Fischer-Iris Dataset

In order to compare the learning algorithm's performance to other spiking neural network learning algorithms, a network was trained to perform classification on the Fischer-Iris dataset. [28]. The Fischer-Iris dataset consists of 3 classes of 50 samples, each sample has 4 features, and some samples are not linearly separable from the incorrect class. The features describe petal and septal lengths and widths for three different classes of flower species. The network used for this task consists of four input neurons, 8 hidden neurons, and 1 output neuron. In contrast to the XOR task, no additional input neuron was used to signal the start of a trial; instead, two spikes were used for each of the 4 input neurons: the first signaled the start of the trial, and the second encoded the analog feature value in its timing. The output neuron was trained to fire at one of three times, corresponding to the input data's class. 5-fold cross-validation was used to improve the robustness of the results, since the size of each class was relatively small. Weights were initialized with a random gaussian distribution having mean 2.0 and variance 4.0, momentum was set to 0.99, and the learning rate η_o was set at 5.0e1. Classification error was calculated by determining whether or not the output spike occurred closest to the target spike time corresponding to the correct class, as opposed to the other classes' target spike times. Results in comparison to other spiking neural network algorithms are shown in Table 3.1. Although comparison is complicated by the use of different cross-validation or holdout procedures, it can be seen that the Continuous Theta BP method performs comparably with other spiking neural network methods. The hold-out method was also used, in which 50 data samples were set aside for the test dataset. The use of this method resulted in deceptively good numbers, even reaching 98% training and 100% test performance. However, these numbers highly depend on the selection of data samples for the test and training data sets.

Algorithm	*Topology*	*Epochs*	*Train*	*Test*
SpikeProp	50x10x3	1000	97.4	96.1
Dynamic Synapse SNN	4x10x1	n/a	96	97.3
NN A	50x10x3	2.60E+06	98.2	95.5
NN B	4x8x1	1.00E+05	98	90
MuSpiNN	4x5x1	192	99.96	94.44
Spike-timing Theta BP	4x8x1	1080	100	98
Continuous Theta BP	**4x8x1**	**3000**	**97.8**	**96.7**

Table 3.1: Comparison of performance of spiking neural network algorithms on the Fischer-Iris Dataset. SpikeProp and NN A results are from Bohte et al. 2000[10], Dynamic Synapse SNN result is from Belatreche et al. 2006[3], Spike-Timing Theta BP and NN B results are from McKennoch et al. 2009[61], and results for MuSpiNN are from Xu et al. 2013[97]. Continuous Theta BP refers to this work. NN A and NN B refer to sigmoidal neural networks trained with classical backpropagation. The closest comparison is with the Spike-timing Theta BP method, from the same work. Note that while the Continuous Theta BP results (this work) are obtained with 5-fold cross-validation, the Spike-timing Theta BP results are obtained via 1/3 holdout, while SpikeProp and NN experiments are conducted via 2-fold cross-validation.

3.5 Wisconsin Breast Cancer Dataset

The Wisconsin Breast Cancer Dataset consists of 699 samples from two classes (malignant/benign), and each sample consists of 9 measurements (radius, texture, perimeter, etc)[7]. Like the Fischer-Iris dataset, it contains both linearly and non-linearly separable data points. A similar network to the Fischer-Iris task was used -

this time with 9 input neurons, 8 hidden neurons, and 1 output neuron. Encoding input values and target classes were done in the same manner as well. Weights were initialized with a mean of 1.0 and variance 5.0, momentum was set to 0.9, and the learning rate η_o was set at 2.0e1. Results of the continuous theta learning algorithm in comparison to other spiking neural network algorithms are shown in Table 3.2.

Algorithm	*Topology*	*Epochs*	*Train*	*Test*
SpikeProp	64x15x2	1500	97.6	97
Dynamic Synapse SNN	9x6x1	n/a	97.2	97.3
NN A	64x15x2	9.20E+06	98.1	96.3
NN B	9x8x1	1.00E+05	97.2	99
MuSpiNN	9x5x1	209	100	95.32
Spike-timing Theta BP	9x8x1	3130	98.3	99
Continuous Theta BP	**9x8x1**	**5000**	**99.0**	**99.14**

Table 3.2: Comparison of performance of spiking neural network algorithms on the Wisconsin Breast Cancer Dataset. SpikeProp and NN A results are from Bohte et al. 2000[10], Dynamic Synapse SNN result is from Belatreche et al. 2006[3], Spike-Timing Theta BP and NN B results are from McKennoch et al. 2009[61], and results for MuSpiNN are from Xu et al. 2013[97]. Spike-timing Theta BP results are obtained via hold out of 1/7 of the data, while Continuous Theta BP results are obtained with 5-fold cross validation.

4. DISCUSSION

4.1 Investigation of Gradient Descent

The application of the derived learning rule for continuous theta neural networks to a simple two-weight problem demonstrates the importance of momentum for successful learning (Section 3.1). Without momentum, the gradient descent will get caught in shallow local minima and converge to suboptimal solutions. Here, no significant difference between Nesterov-style momentum and classical momentum was observed. This is because the benefits of Nesterov-style momentum shines when the gradient must rapidly change directions. For the simple two-weight example, the gradient is largely in one direction, and does not need to turn.

The importance of rapidly changing directions is demonstrated with the spike-creation example (same section). In this example, 100 input weights contribute to the creation and placement of one output spike. Before the output spike is created, the learning rule moves the weights in the direction which creates a spike. Once the spike is created, however, the direction of the gradient descent must quickly change to shift the spike to the correct time. Because Nesterov-style momentum calculates the gradient after a partial weight update, it is more sensitive to sudden changes in the direction of steepest descent. In contrast, classical momentum is slower to adjust, and may be more prone to overshooting. In experiments comparing classical momentum to Nesterov-style momentum with all other parameters controlled for, Nesterov style-momentum would correctly shift the trajectory from spike-creation to spike-shifting. On the other hand, classical momentum would often be slower to adapt, sometimes resulting in a spike being created then immediately removed as the negative weights corresponding to input spikes occurring after the target spike time would continue to decrease.

While the use of a kernel for the theta neuron model removes discontinuities related to modeling spikes as impulses as well as eliminating the need for a hard-threshold and reset function to model refractoriness, a significant discontinuity still remains. The discontinuity caused by the bifurcation in the dynamics of the theta model at the repelling fixed point results in difficulties for gradient descent to be able to choose whether or not to create/remove spikes or shift them. The use of Nesterov-style momentum may be helpful in overcoming this problem, since it can better account for sudden and dramatic changes in the error surface when spikes are created or removed. However, this remains a significant issue in the training of spiking neural networks, and has been the subject of many studies. For example, one approach is to construct an error function by directly taking into account both the number and timing of spikes, which results in an error surface resembling a patchwork of piecewise smooth surfaces adjoined by discontinuous boundaries. Gradient descent on this error function results in a learning rule can find the minimum within the support of a piecewise smooth surface, but may have trouble crossing boundaries[29]. Other attempts have focused on analytical solutions for the input weights to a single layer network given desired spike times. While this approach is guaranteed to find a solution if one exists, it is not applicable to networks with hidden layers, where target spike times are not explicitly specified[62].

In this work's approach, the fact that spikes are kernelized and smoothed out in time results in gradient descent naturally choosing whether to create a new spike or create a new one. This is because the width of the kernel function determines the temporal width of a spike. When a spike occurs it has a long "tail" on either side, and target spikes which occur within this tail will cause the gradient to shift the spike towards the target. At the same time, the target spike will influence the gradient to create a new spike by increasing weights corresponding to neurons that have fired recently before it. These influences compete continuously, with spike-shifting winning when spikes are close to the target, and spike-creation winning when existing spikes are far away.

4.2 Weight Initialization

The learning rule was successfully applied to a small network trained to solve the XOR task (Section 3.2). The XOR task has a local minimum corresponding to correctly identifying two out of the four possible inputs. Without the use of momentum, the learning rule would easily get stuck at this minimum. Furthermore, the ability of the algorithm to find a good solution was sensitive to how the weights were initialized, in agreement with the observations of Sutskever et al[23]. Faster convergence was obtained when weights were not too large (which would result in the network being saturated with a large number of spikes) or too small (which would result in few initial numbers of spikes). Convergence was fastest when the initial number of output spikes produced matched the number of target output spikes. If the number of output spikes was too many or too few, convergence was still possible, but slower and more susceptible to being trapped in local minima. This issue of sensitivity to weight initialization (and of hyper-parameters in general) has sometimes been a point of criticism of neural networks in general, and it is therefore not surprising to find the same issue here.

4.3 Universal Function Approximation and Other Considerations

It has been shown that a neural network with a single hidden layer can act as a "universal function approximator", i.e. it can approximate any continuous function with compact support[52]. Spiking neural networks have been shown to possess the same property[55]. To try to verify this property for continuous theta neuron networks, two regression tasks were attempted with the derived learning rule - the cosine and the sinc function (Section 3.3). A network with 2 input neurons, 5 hidden neurons, and 1 output neuron was able to map a cosine function over a small interval. Observation of the hidden neuron activity before and after learning reveals that over the course of learning, the number of spikes produced per hidden neuron per trial changes. At the initial weights before learning has begun, each hidden neuron fires one spike in response to the input. When learning is finished, one hidden neuron produces either one or two spikes depending on the input, while another hidden neuron produces either one or no spike at all in response to the input. This variable number of spikes per trial shows that the learning algorithm is able to train hidden units to produce a variable number of spikes in response to input. This offers a distinct advantage over the exact-spike BP approach to training theta neuron networks.

The successful learning of a cosine mapping may indicate that a network could learn any arbitrary function comprised of a sum of cosines. This idea was put to the test with the sinc function mapping. It was more difficult to train a network to correctly map this function, indeed, after trying different numbers of hidden units (from 5 to 20), the best performance achieved was for a network with 11 hidden neurons. For this case, the network was able to learn to map the function reasonably well, with a slight mistake in one of the tails of the function. This may be due to getting stuck in a local minimum which was too deep to be overcome with momentum. The plot of the error over epochs reveals that after an initial drop in error in the first 10 or so epochs, there is not any significant improvement for nearly 1000 epochs. This plateau corresponds to the local minimum where each input is mapped to the mean of the sinc function. Here, the use of a large momentum coefficient greatly aids convergence, reducing the time spent navigating this plateau. Eventually the network finds a way to descend closer to the global minimum, rapidly decreasing then leveling out as it searches the valley for a good solution. Unfortunately, further iterations did not improve the error rate. This may be due to the relatively large step size of the learning rate when it is close to the global minimum. There is a trade-off when choosing a fixed learning rate: if the rate is large, the algorithm may be able to find solutions farther from the initial starting point, but may be unable to accurately converge to a global minimum in a narrow valley. On the other hand, while a small learning rate can accurately descend narrow valleys, it way be slow or more easily get stuck in local minima. This trade-off has encouraged the use of learning rate schedules, which starts the learning rate at a large value and gradually decreases it over the course of learning. The momentum coefficient can also be placed on a similar schedule to aid in converging to the elusive global minimum[98].

It is worth considering why performance decreased when adding hidden neurons for this sinc task. For small number of hidden neurons (5-8), the solution often took on a bell-shape approximating the sinc function, but was unable to model the tails. Adding more hidden neurons (9-13) enabled the network to model these tails more successfully. However, adding even more hidden neurons caused performance to decrease, resulting in “messy” solutions where the error of individual output values had a larger variance. One hypothesis for this behavior is that the large number of hidden neurons feeding into one output neuron causes the output neuron to be easily saturated, reducing its sensitivity to individual inputs. This kind of problem can occur in sigmoidal neural networks, where it has been shown that using sparse initialization (SI), where each neuron only receives input from a subset of neurons from the previous layer, can be beneficial[67].

4.4 Machine Learning Tasks

In section 3.4 and 3.5, the continuous theta backpropagation algorithm was used to train theta networks to classify the Fischer-Iris dataset and the Wisconsin Breast Cancer Dataset. Although these datasets are relatively simple, they provide a way to compare the current algorithm to other neural network algorithms. The algorithms being compared are the SpikeProp algorithm,[10] dynamic synapse SNN,[3] sigmoidal neural networks trained with backpropagation (NN A and NN B), and the spike-timing theta BP algorithm discussed in Chapter 2. The SpikeProp algorithm uses a large number of input neurons because each input feature is encoded by multiple neurons, with each neuron having graded and slightly overlapping "receptive fields". This method of encoding analog values in spikes is biologically plausible and well-studied, but comes at the cost of a much larger network with many more parameters to train[26]. It also neglects the appeal of encoding values using spike timing, instead opting for a population code. Furthermore, the SpikeProp algorithm uses multiple synaptic connections between neurons - for the results shown, 16 synaptic connections are used between each pair of neurons. On the Fischer-Iris task, SpikeProp is training $50 * 10 * 3 * 16 = 24,000$ parameters, while the continuous theta method is training 32. The improved performance of both Theta neuron methods relative to SpikeProp indicate that the burden of computation can be successfully shifted from the synapses to the nonlinear dynamics of the theta neuron model.

Sigmoidal neural networks of different sizes were also used in the comparison, with NN A being the same size as the SpikeProp network, and NN B the same size as the spike-timing theta and continuous theta algorithms. However, NN algorithms achieved lower performance than the theta neuron methods. Theoretical comparison of the difference between NN algorithms vs algorithms using the theta neuron model may be difficult. However, the improved performance of the theta neuron model could be due to the robustness of computation obtained from the nonlinear temporal dynamics modeled by the theta model. In comparison, a sigmoidal neuron consists solely of a sigmoidal threshold of a weighted sum. While the simplicity of the sigmoidal neural network makes analysis easy, this may come at the cost of computational power, particularly in a temporal context.

The main comparison to be made is with the spike-timing theta BP algorithm investigated by McKennoch et al., since it also uses the theta neuron model[61]. However, results for the machine learning tasks should be taken with a grain of salt since McKennoch et al. do not use cross-validation, instead separating the Fischer-Iris dataset into 100 training and 50 test samples, and the Wisconsin Breast Cancer dataset into 599 training and 100 test samples. In contrast, this work presents results for 5-fold cross-validation. Running tests in the same hold-out manner as the spike-timing theta BP work can result in 100% test performance for both the Fischer-Iris task and the Wisconsin Breast Cancer dataset using continuous theta BP, depending on the choice of training and test subsets. The small size of the dataset precipitated the decision to present results using 5-fold cross-validation to give a better idea of how well the algorithm is able to learn the complete dataset.

The improved performance of the continuous theta method over the spike-timing method could be explained in several ways. First, the learning rules for the spike-timing method assumes that spikes are modeled as impulses. This is a crucial assumption in the derivation, since the times of spikes occurring form the bounds of integrals which are summed to calculated the gradient. In contrast, the continuous theta method presented here models spikes as smooth Gaussian-shaped kernels whose width can be adjusted to loosely match the synaptic dynamics of real neurons. Second, the spike-timing method assumes a fixed number of spikes, and calculates the effect of shifting each spike by a certain amount. Spikes can still be added or removed over the course of learning as the weights change, but these occurrences are not taken into account in the learning rules and account for significant discontinuities in the error surface. Furthermore, if during the course of learning via spike-timing theta BP a neuron stops firing any spikes, it will not be able to recover, and will effectively drop out of the network. This is because if a neuron does not fire there is no way for the learning rule to determine the effect of changing that neuron's weight on the timing of the spikes that it fires. Therefore all weights to the silent neuron will stop being adjusted. For tasks where each neuron fires a single spike, the computational power of the network may progressively decrease as neurons stop firing during the course of learning.

In contrast to the spike-timing theta BP method, the continuous theta algorithm presented here is able to both create and remove spikes over the course of learning. This allows the spiking network to take advantage of a much greater realm of computations where the number of times a neuron fires can carry discriminatory information. At the same time, learning the timing of individual spikes is not neglected. The overall effect is that the continuous theta backpropagation algorithm is able to bridge the gap between spike-timing coding and rate-coding. The ability to make use of a broader realm of computations may help to explain the improved performance on the Wisconsin Breast Cancer dataset.

4.5 Recurrent Network Topologies and Baseline Current Learning

The continuous nature of a coupled theta neural network allows for the gradient to be calculated for more complex topologies. In the Appendix, backpropagation for a continuous theta neural network with a recurrent hidden layer is derived. This theoretically would allow for computations with long time lags to be made, in which the recurrent hidden layer could sustain its own activity. For sigmoidal neural networks, recurrent topologies have been used to successfully model complex temporal functions on relatively long timescales. However, in this case convergence to acceptable solutions proved difficult, for several possible reasons. First, training a reasonably-sized network was very slow, especially since the number of weights in the hidden layer is $\mathcal{H}^2$. This made it difficult to find parameters which could encourage good convergence by trial and error. Second, it is possible that using a recurrent topology would be more useful on longer timescales on the order of tens or hundreds of time constants. Because of the computational expense, this was difficult to assess. Finally, it is possible that the long-time lag problem, where errors do not backpropagate significantly far back in time within a recurrent or deep topology, prevents the learning rule from updating weights quickly enough[4]. Although more investigation is needed, the ability to apply backpropagation to recurrent topologies for theta neural networks could still be a potential advantage.

Presented in the Appendix are learning rules for the baseline current of each neuron. As discussed in Chapter 2, when the theta neuron's baseline current is positive, the fixed points of dynamics disappear, and the dynamics take on periodic oscillations around the phase circle - i.e. the neuron fires continuously without the need for additional input. This bistable behavior could be exploited to allow neurons to encode a binary "memory", with some neurons learning to produce sustained activity to encode some information which can be accessed later. Another potential benefit of adjusting the baseline current would be to allow each neuron to customize its sensitivity to inputs, if, for example, some neurons receive a high degree of input and other receive smaller values. However, there were difficulties in demonstrating these potential benefits. As the baseline current crosses from a negative value to a positive one, a bifurcation of the neuron's dynamics occurs. This results in a major discontinuity in the error surface. This shift to tonic firing is very disruptive the the gradient descent process. A more intelligent way of deciding how to store long term memories would be needed to take advantage of this feature of the theta neuron model. Something closer to the Long Short Term Memory LSTM architecture might be needed, where memory cells are guarded with differentiable "gates" for reading, writing, and resetting. [32]. An interesting recent development in this line of neural network research is the Neural Turing Machine, in which a recurrent neural network is coupled with an external memory bank and given differentiable reading and writing operations[37]. Restricting the baseline current to negative values to prevent this catastrophic bifurcation resulted in marginal improvements on the XOR task and the Fischer-Iris task (not shown), but these were sporadic and inconsistent. It may be that learning both weights and baseline currents simultaneously create unfavorable interactions in the gradient. Indeed, convergence generally took longer when applying learning rules to both weights and baseline currents. Nevertheless, being able to learn additional parameters which control each neuron's sensitivity to input could prove to be a benefit in certain situations.

4.6 Biological Feasibility

The question of how biologically feasible a neuron model and its accompanying learning algorithms may be is important for advancing understanding of how the brain performs computations. One of the primary questions relevant which is relevant to this work to consider is the biological feasibility of backpropagation. Many studies have considered this question (e.g. see Stork 1989,[86] which is dedicated to this question alone). The main problem with the biological feasibility of backpropagation is that it requires a neural mechanism for keeping track of and transmitting errors from the last output layer to the hidden and input layers. The needed large-scale organization of many neurons precisely transmitting errors to the correct neurons seems like a daunting requirement. However, recent work by Bengio et al. suggests that layers of denoising auto-encoders in a deep learning framework only need to propagate errors forward and backwards one layer[5]. The biologically realistic propagation of errors from one layer to another is a more reasonable demand. Indeed, O'Reilly explores such an idea in his work on a biologically plausible generalization of contrastive divergence[68]. Nevertheless, although backpropagation and other error-driven algorithms are sometimes criticized for being biologically infeasible, the use of backpropagation in spiking neural networks remains useful for assessing the computational possibilities of spiking neuron models. For example, showing that theta neurons can perform robust computations by utilizing both rate-coding and spike-timing coding supports the idea that the brain could be doing the same, helping to bridge the old rate-coding vs. timing-coding debate[38].

The question of whether or not the learning rules derived in this work are biologically plausible is an interesting one as well. Admittedly, the equations presented in Section 2 do not directly model any known cellular or molecular processes. However, there must be biological processes which do accomplish the overall goal of temporal and structural credit assignment. For example, spike timing dependent plasticity (STDP) is a well-studied mechanism which changes synaptic weights based on the relative spike timing of the presynaptic and postsynaptic neurons. A wide variety of STDP types and mechanisms exist and have been shown to be affected by neuromodulators of various kinds[71]. This high level of complexity creates a rich realm of possibilities for STDP-based learning rules and computation, placing the possibility of credit assignment via these types of processes on the table[51]. In addition to the various forms of STDP, the various forms of dendritic computation adds another dimension of possibility for complex and robust learning rules in the brain[100].

In this work, to create a continuously differentiable spiking neural network, the theta neurons are directly coupled such that information about subthreshold membrane dynamics are transmitted to other neurons. While this is not traditionally modeled in spiking neural networks algorithms, it is certainly biologically plausible. Gap junctions are the means by which cells can share ions and small organic molecules, and form the basis for electrical synapses between neurons in the brain (as distinct from chemical synapses, which use neurotransmitters)[21]. Gap junctions are particularly prevalent during postnatal development when the brain is rapidly developing, and remain present in certain classes of neurons in adulthood[95]. The interaction of chemical and electrical synapses have been shown to play a large role in the population dynamics of interneurons in the cortex, affecting synchrony and stability of activity[63]. Therefore, the derivation of the algorithm in this work can be said to take into account some of the effects of both chemical and electrical synapses. While these effects here are modeled qualitatively and not as strictly biologically realistic mechanisms, they may provide some insight into their importance for computation in the brain.

Finally, the biological plausibility of the theta neuron model itself should be considered. The use of the theta neuron model provides an additional level of biological realism when compared to the LIF model or sigmoidal neural networks. This may be an additional reason for its improved performance presented in the results. For example, the ability to model first-spike latencies gives the possibility of spikes being "canceled" by later inhibitory input after the firing threshold has been crossed. While the theta model lacks certain characteristics of more complex models and therefore does not provide a complete model of neuron dynamics, it retains an analytical simplicity which allows for concrete computational performance evaluations.

5. CONCLUSION

In this thesis a novel backpropagation learning rule for theta neuron networks coupled by smoothed spikes was derived. This learning rule was shown to be effective for training spiking neural networks where changes in both the timing and the number of spikes were beneficial for the task at hand. Nesterov-style momentum was used to improve convergence. Several benchmarks were used to compare the learning rule against other spiking neural network learning algorithms; these benchmarks included the XOR task, the Fischer-Iris dataset, and the Wisconsin Breast Cancer Dataset. Comparable performance for the Fischer-Iris Dataset and improved performance on the Wisconsin Breast Cancer dataset was shown. The successful demonstration of a continuously coupled theta neuron network to perform computations suggests that computation in biological neural networks can simultaneously make use of the properties modeled. These properties include those modeled by the theta neuron model itself (such as spike latencies, activity-dependent thresholding, and tonic firing), as well as those modeled by coupling neurons with a smoothing kernel (such as synaptic dynamics and coupled sub-threshold dynamics via gap junctions). Although the biological plausibility of these learning rules is not strong, it has been demonstrated that a network of nonlinear dynamic nodes modeling biological neurons are able to perform complex spike-timing and classification tasks.

REFERENCES

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann Machines. *Cognitive Science. Special Issue: Connectionist models and their applications*, 9(1):147–169, March 1985.

[2] Frederico a C Azevedo, Ludmila R B Carvalho, Lea T. Grinberg, José Marcelo Farfel, Renata E L Ferretti, Renata E P Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, April 2009.

[3] Ammar Belatreche, Liam P. Maguire, and Martin McGinnity. Advances in design and application of spiking neural networks. *Soft Computing*, 11(3):239–248, 2007.

[4] Y Bengio, P Simard, and P Frasconi. Learning long term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[5] Yoshua Bengio, Dong-Hyun Lee, Jörg Jorg Bornschein, and Zhouhan Lin. Towards Biologically Plausible Deep Learning. *CoRR*, abs/1502.0, February 2015.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] C L Blake and C J Merz. UCI Repository of machine learning databases. *University of California*, page http://archive.ics.uci.edu/ml/, 1998.

[8] Sander M. Bohte. Error-backpropagation in networks of fractionally predictive spiking neurons. *Lecture Notes in Computer Science (including subseries*

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6791 LNCS(PART 1):60–68, June 2011.

[9] Sander M. Bohte, Joost N. Kok, and Han La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1-4):17–37, October 2002.

[10] Sander M. Bohte, Joost N. Kok, and Han La Poutre. SpikeProp: Backpropagation for networks of spiking neurons. *Neurocomputing*, 48(1-4):17, 2002.

[11] Sander M. Bohte and Jaldert O. Rombouts. Fractionally Predictive Spiking Neurons. *Advances in Neural Information Processing Systems*, 23:13, October 2010.

[12] Olaf Booij and Hieu Tat Nguyen. A gradient descent rule for spiking neurons emitting multiple spikes. *Information Processing Letters*, 95(6 SPEC. ISS.):552–558, September 2005.

[13] Tiago Branco and Michael Häusser. The single dendritic branch as a fundamental functional unit in the nervous system. *Current Opinion in Neurobiology*, 20(4):494–502, August 2010.

[14] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, 94(5):3637–3642, November 2005.

[15] Lars Buesing, Johannes Bill, Bernhard Nessler, and Wolfgang Maass. Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons. *PLoS Computational Biology*, 7(11):e1002211, November 2011.

[16] Natalia Caporale and Yang Dan. Spike timing-dependent plasticity: a Hebbian learning rule. *Annual review of neuroscience*, 31:25–46, January 2008.

[17] C E Carr, W Heiligenberg, and G J Rose. A time-comparison circuit in the electric fish midbrain. I. Behavior and physiology. *The Journal of Neuroscience*, 6(1):107–119, 1986.

[18] Stijn Cassenaer and Gilles Laurent. Corrigendum: Conditional modulation of spike-timing-dependent plasticity for olfactory learning. *Nature*, 487(7405):128–128, March 2012.

[19] Mohamed Cheriet and Reza Farrahi Moghaddam. Guide to OCR for Arabic Scripts. In *Guide to OCR for Arabic Scripts*, pages 453–484, 2012.

[20] B W Connors and M J Gutnick. Intrinsic firing patterns of diverse neocortical neurons. *Trends in neurosciences*, 13(3):99–104, March 1990.

[21] Barry W Connors and Michael A Long. Electrical synapses in the mammalian brain. *Annual Review of Neuroscience*, 27:393–418, January 2004.

[22] O D Creutzfeldt. Generality of the functional structure of the neocortex. *Die Naturwissenschaften*, 64(10):507–517, October 1977.

[23] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, volume 28, pages 8609–8613, 2013.

[24] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech and Language Processing*, 20(1):30–42, 2012.

[25] G. B. Ermentrout and N. Kopell. Parabolic Bursting in an Excitable System Coupled with a Slow Oscillation. *SIAM Journal on Applied Mathematics*, 46(2):233–253, April 1986.

[26] C W Eurich and S D Wilke. Multidimensional encoding strategy of spiking neurons. *Neural Computation*, 12(7):1519–1529, 2000.

[27] Zheng Fang. A robust and effcient algorithm for mobile robot localization. *Acta Automatica Sinica*, 33(1):0048, 2007.

[28] Ra Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[29] Rzvan V. Florian. The chronotron: A neuron that learns to fire temporally precise spike patterns. *PLoS ONE*, 7(8):e40233, January 2012.

[30] J a Fodor and Z W Pylyshyn. Connectionism and cognitive architecture: a critical analysis. *Cognition*, 28(1-2):3–71, March 1988.

[31] Surya Ganguli, Dongsung Huh, and Haim Sompolinsky. Memory traces in dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 105(48):18970–18975, December 2008.

[32] F a Gers, J Schmidhuber, and F Cummins. Learning to forget: continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, October 2000.

[33] Wulfram Gerstner. Time structure of the activity in neural network models. *Physical Review E*, 51(1):738–758, January 1995.

[34] Gaolang Gong, Yong He, Luis Concha, Catherine Lebel, Donald W. Gross, Alan C. Evans, and Christian Beaulieu. Mapping anatomical connectivity patterns of human cerebral cortex using in vivo diffusion tensor imaging tractography. *Cerebral Cortex*, 19(3):524–536, March 2009.

[35] a Graves, a Mohamed, and G Hinton. Speech recognition with deep recurrent neural networks. *Acoustics*, March 2013.

[36] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks. *Proceedings of the International Joint Conference on Neural Networks*, 4(5-6):2047–2052, 2005.

[37] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *CoRR*, abs/1410.5, October 2014.

[38] Robert Gütig. To spike, or when to spike? *Current Opinion in Neurobiology*, 25:134–139, April 2014.

[39] Hatsuo Hayashi and Jun Igarashi. LTD windows of the STDP learning rule and synaptic connections having a large transmission delay enable robust sequence learning amid background noise. *Cognitive Neurodynamics*, 3(2):119–130, June 2009.

[40] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science (New York, N.Y.)*, 313(5786):504–507, 2006.

[41] S Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen.* Diploma thesis, Technische Universität München, 1991.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[43] a. L. Hodgkin and a. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bulletin of Mathematical Biology*, 52(1-2):25–71, August 1990.

[44] Jonathan C Horton and Daniel L Adams. The cortical column: a structure without a function. *Philosophical Transactions of the Royal Society of London. Series B, Biological sciences*, 360(1456):837–862, April 2005.

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

[46] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, September 2004.

[47] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. *Science (New York, N.Y.)*, 304(5667):78–80, 2004.

[48] David Kappel, Bernhard Nessler, and Wolfgang Maass. STDP installs in winner-take-all circuits an online approximation to Hidden Markov Model learning. *PLoS Computational Biology*, 10(3):e1003511, March 2014.

[49] Jan Koutník, Klaus Greff, Faustino J. Gomez, Jurgen Jürgen Schmidhuber, Jan Koutnik, Klaus Greff, Faustino J. Gomez, and Jurgen Jürgen Schmidhuber. A Clockwork RNN. *CoRR*, abs/1402.3, February 2014.

[50] Robert Legenstein and Wolfgang Maass. Edge of chaos and prediction of computational performance for neural circuit models. *Neural Networks*, 20(3):323–334, April 2007.

[51] Robert Legenstein, Christian Naeger, and Wolfgang Maass. What can a neuron learn with spike-timing-dependent plasticity?, March 2005.

[52] G. Lewicki and G. Marino. Approximation of functions of finite variation by superpositions of a sigmoidal function. *Applied Mathematics Letters*, 17(10):1147–1152, 2004.

[53] Evgueniy V Lubenov and Athanassios G Siapas. Hippocampal theta oscillations are travelling waves. *Nature*, 459(7246):534–539, May 2009.

[54] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, August 2009.

[55] Wolfgang Maass and Henry Markram. On the computational power of circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, December 2004.

[56] Timothée Masquelier, Rudy Guyonneau, and Simon J. Thorpe. Spike timing dependent plasticity finds the start of repeating patterns in continuous spike trains. *PLoS ONE*, 3(1):e1377, January 2008.

[57] Timothée Masquelier, Etienne Hugues, Gustavo Deco, and Simon J Thorpe. Oscillations, phase-of-firing coding, and spike timing-dependent plasticity: an efficient learning scheme. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 29(43):13484–13493, October 2009.

[58] Hermann Mayer, Faustino Gomez, Daan Wierstra, Istvan Nagy, Alois Knoll, and Jürgen Schmidhuber. A system for robotic heart surgery that learns to tie knots using recurrent neural networks. *IEEE International Conference on Intelligent Robots and Systems*, pages 543–548, April 2006.

[59] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, December 1943.

[60] Sam McKennoch, Preethi Sundaradevan, and Linda G. Bushnel. Theta neuron networks: Robustness to noise in embedded applications. In *IEEE In-*

ternational Conference on Neural Networks - Conference Proceedings, pages 2330–2335. IEEE, August 2007.

[61] Sam McKennoch, Thomas Voegtlin, and Linda Bushnell. Spike-Timing Error Backpropagation in Theta Neuron Networks. *Neural Computation*, 0(0):080804143617793–37, January 2008.

[62] Raoul Martin Memmesheimer, Ran Rubin, BenceP Ölveczky, and Haim Sompolinsky. Learning precisely timed spikes. *Neuron*, 82(4):925–938, April 2014.

[63] Elliott B Merriam, Theoden I Netoff, and Matthew I Banks. Bistable network behavior of layer I interneurons in auditory cortex. *The Journal of Neuroscience*, 25(26):6175–6186, June 2005.

[64] Vernon B. Mountcastle. The columnar organization of the neocortex. *Brain*, 120(4):701–722, April 1997.

[65] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in Neuroscience*, 7(January):1–14, November 2014.

[66] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, and Wolfgang Maass. Bayesian computation emerges in generic cortical microcircuits through Spike-Timing-Dependent Plasticity. *PLoS Computational Biology*, 9(4):e1003037, April 2013.

[67] Ryusuke Niwa and Yuko S. Niwa. The fruit fly drosophila melanogaster as a model system to study cholesterol metabolism and homeostasis. *Cholesterol*, 2011:735–742, 2011.

[68] Randall C. O'Reilly. Biologically plausible error-driven learning using local activation differences: The Generalized Recirculation Algorithm. *Neural Com-*

putation, 8(5):895–938, 1996.

[69] Randall C O'Reilly and Michael J Frank. Making working memory work: a computational model of learning in the prefrontal cortex and basal ganglia. *Neural computation*, 18(2):283–328, March 2006.

[70] Stefano Panzeri and Mathew E. Diamond. Information carried by population spike times in the whisker sensory cortex can be decoded without knowledge of stimulus time. *Frontiers in Synaptic Neuroscience*, 2(JUN):17, January 2010.

[71] Verena Pawlak, Jeffery R. Wickens, Alfredo Kirkwood, and Jason N D Kerr. Timing is not everything: Neuromodulation opens the STDP gate. *Frontiers in Synaptic Neuroscience*, 2(OCT):146, January 2010.

[72] Dejan Pecevski, Lars Buesing, and Wolfgang Maass. Probabilistic inference in general graphical models through sampling in stochastic networks of spiking neurons. *PLoS Computational Biology*, 7(12):e1002294, December 2011.

[73] Bruno U. Pedroni, Srinjoy Das, Emre Neftci, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Neuromorphic adaptations of restricted Boltzmann machines and deep belief networks. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, August 2013.

[74] Filip Ponulak and Andrzej Kasiski. Supervised learning in spiking neural networks with ReSuMe: sequence learning, classification, and spike shifting. *Neural Computation*, 22(2):467–510, February 2010.

[75] Michael J. Proulx, David J. Brown, Achille Pasqualotto, and Peter Meijer. Multisensory perceptual learning and sensory substitution. *Neuroscience and Biobehavioral Reviews*, 41:16–25, April 2014.

[76] John Rinzel and G. Bard Ermentrout. Analysis of neural excitability and oscillations. *Methods in Neuronal Modeling*, pages 251–292, 1998.

[77] F Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[78] Timothy Rumbell, Susan L. Denham, and Thomas Wennekers. A Spiking Self-Organizing Map combining STDP, oscillations, and continuous learning. *IEEE Transactions on Neural Networks and Learning Systems*, 25(5):894–907, May 2013.

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

[80] T D Sanger. Probability density methods for smooth function approximation and learning in populations of tuned spiking neurons. *Neural computation*, 10(6):1567–1586, August 1998.

[81] J Schmidhuber. Deep learning in neural networks: an overview. *arXiv preprint arXiv:1404.7828*, abs/1404.7:1–66, April 2014.

[82] B. Schrauwen and J. Van Campenhout. Backpropagation for Population-Temporal Coded Spiking Neural Networks. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 1797–1804. IEEE, 2006.

[83] M N Shadlen and W T Newsome. The variable discharge of cortical neurons: implications for connectivity, computation, and information coding. *The Journal of Neuroscience*, 18(10):3870–3896, May 1998.

[84] H Söderlund, L Kääriäinen, and C H von Bonsdorff. *Properties of Semliki Forest virus nucleocapsid.*, volume 53. 1975.

[85] Ioana Sporea and Andre Gruning. Supervised learning in multilayer Spiking Neural Networks. *Neural Computation*, 25(2):473–509, February 2012.

[86] D.G. Stork. Is backpropagation biologically plausible? In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 241–246 vol.2. IEEE, 1989.

[87] S H Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, March 2001.

[88] Y Sugase, S Yamane, S Ueno, and K Kawano. Global and fine information coded by single neurons in the temporal visual cortex. *Nature*, 400(6747):869–873, August 1999.

[89] Ilya Sutskever and Geoffrey Hinton. Temporal-Kernel Recurrent Neural Networks. *Neural Networks*, 23(2):239–243, March 2010.

[90] Ilya Sutskever, Geoffrey Hinton, and Graham Taylor. The Recurrent Temporal Restricted Boltzmann Machine. *Neural Information Processing Systems*, 21(1):1601–1608, 2008.

[91] Botond Szatmáry and Eugene M. Izhikevich. Spike-timing theory of working memory. *PLoS Computational Biology*, 6(8):11, January 2010.

[92] Yuki Todo, Hiroki Tamura, Kazuya Yamashita, and Zheng Tang. Unsupervised learnable neuron model with nonlinear interaction on dendrites. *Neural Networks*, 60:96–103, August 2014.

[93] J D Victor and K P Purpura. Nature and precision of temporal coding in visual cortex: a metric-space analysis. *Journal of neurophysiology*, 76(2):1310–1326, 1996.

[94] Thomas Voegtlin. Temporal coding using the response properties of spiking neurons. *Advances in neural information processing systems*, page 8, 2007.

[95] Xiao-Jing Wang. Neurophysiological and computational principles of cortical rhythms in cognition. *Physiological reviews*, 90(3):1195–1268, July 2010.

[96] P Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, 1974.

[97] Yan Xu, Xiaoqin Zeng, Lixin Han, and Jing Yang. A supervised multi-spike learning algorithm based on gradient descent for spiking neural networks. *Neural Networks*, 43:99–113, July 2013.

[98] Y. Nesterov. A method of solving a convex programming problem with convergence rate O(1/k2).pdf. *Soviet Mathematics Doklady*, 1983.

[99] Roman V. Yampolskiy. Turing test as a defining feature of AI-completeness. *Studies in Computational Intelligence*, 427:3–17, 2013.

[100] Danke Zhang, Yuanqing Li, Malte J Rasch, and Si Wu. Nonlinear multiplicative dendritic integration in neuron and network models. *Frontiers in computational neuroscience*, 7(May):56, January 2013.

[101] Yong Zhang, Boyuan Yan, Mingchao Wang, Jingzhen Hu, Haokai Lu, and Peng Li. Linking brain behavior to underlying cellular mechanisms via large-scale brain modeling and simulation. *Neurocomputing*, 97:317–331, November 2012.

APPENDIX A

BACKPROPAGATION WITH A RECURRENT HIDDEN LAYER AND BASELINE CURRENTS

A.1 Network Topology

The learning rules presented in Chapter 2 only cover the case when the hidden layer is not recurrent, and assumes a fixed baseline current I_0. Here the case where the hidden layer is recurrent is considered, and the gradient with respect to each neuron's baseline current is considered. I_0 is replaced with I_j, and backpropagation for each baseline current is calculated.

Let $\mathcal{I}$, $\mathcal{H}$, and $\mathcal{O}$ denote the set of input, hidden, and output neurons indexed over i,j, and k, respectively, with $|\mathcal{I}| = M$, $|\mathcal{H}| = N$, and $|\mathcal{O}| = P$. The indices l, h, and p are also used for hidden units. Input neurons $\mathcal{I}$ are connected to hidden neurons $\mathcal{H}$ via weights w_{ij} in a feed-forward manner, neurons within the hidden layer $\mathcal{H}$ are connected recurrently via weights w_{jl}, and hidden layer neurons are connected to the output layer $\mathcal{O}$ via weights w_{jk}, again in a feed-forward manner. Input and target spike trains are denoted by $X_i[n]$ and $T_k[n]$ respectively. The error function is given by the sum squared error of the output layer and the targets:

$$E[n] = \frac{1}{2}\sum_{k-1}^{P}(\kappa(\theta_k[n]) - T_k[n])^2 \tag{A.1}$$

$T_k[n] = 1$ if a spike is desired during the n^{th} time bin, and $T_k[n] = 0$ otherwise. Similarly, the input spike trains are 0 or 1 for each $X_i[n]$.

A.2 Forward Pass

Denote $\theta_k[n]$ as θ_k^n for convenience. The forward pass for the input neurons $i \in \mathcal{I}$, hidden neurons $j, l \in \mathcal{H}$, and output neurons $k \in \mathcal{O}$ are, respectively:

$$\theta_i^n = \theta_i^{n-1} + \frac{\Delta t}{\tau} \left[(1 - \cos \theta_i^{n-1}) + \alpha (1 + \cos \theta_i^{n-1})(I_i + X_i[n-1]) \right] \tag{A.2}$$

$$\begin{aligned} \theta_j^n = \theta_j^{n-1} + \frac{\Delta t}{\tau} \Bigg[& (1 - \cos \theta_j^{n-1}) + \alpha (1 + \cos \theta_j^{n-1}) \\ & \left(I_j + \sum_{i \in \mathcal{I}} w_{ij} \kappa(\theta_i^{n-1}) + \sum_{\substack{l \in \mathcal{H} \\ l \neq j}} w_{lj} \kappa(\theta_l^{n-1}) \right) \Bigg] \end{aligned} \tag{A.3}$$

$$\theta_k^n = \theta_k^{n-1} + \frac{\Delta t}{\tau} \left[(1 - \cos \theta_k^{n-1}) + \alpha (1 + \cos \theta_k^{n-1}) \left(I_k + \sum_{j \in \mathcal{H}} w_{jk} \kappa(\theta_j^{n-1}) \right) \right]. \tag{A.4}$$

A.3 Backward Pass for Baseline Currents

We want to find the dependence of the error on the baseline currents I_i,I_j, and I_k, as well as the weights w_{ij},w_{lj}, and w_{jk}. Beginning with output baseline currents I_k:

$$\frac{\partial E[n]}{\partial I_k} = \sum_{k \in \mathcal{O}} (\kappa(\theta_k^n) - T_k^n) \kappa'(\theta_k^n) \delta_{k,k}^n \tag{A.5}$$

$$\begin{aligned} \delta_{k,k}^n = \delta_{k,k}^{n-1} + \frac{\Delta t}{\tau} \Bigg[& \delta_{k,k}^{n-1} \sin \theta_k^{n-1} + \alpha (1 + \cos \theta_k^{n-1}) \\ & - \alpha \delta_{k,k}^{n-1} \sin \theta_k^{n-1} \left(I_k + \sum_{j \in \mathcal{H}} w_{jk} \kappa(\theta_j^{n-1}) \right) \Bigg] \end{aligned} \tag{A.6}$$

The backward pass for hidden layer baseline currents I_j is:

$$\frac{\partial E[n]}{\partial I_j} = \sum_{k \in \mathcal{O}} (\kappa(\theta_k^n) - T_k^n)\kappa'(\theta_k^n)\delta_{k,j}^n \tag{A.7}$$

$$\begin{aligned}\delta_{k,j}^n = \delta_{k,j}^{n-1} + \frac{\Delta t}{\tau}\Bigg[&\delta_{k,j}^{n-1}\sin\theta_k^{n-1} + \alpha(1+\cos\theta_k^{n-1})\sum_{l \in \mathcal{H}} w_{lk}\delta_{l,j}^{n-1}\kappa'(\theta_l^{n-1}) \\ &-\alpha\delta_{k,j}^{n-1}\sin\theta_k^{n-1}(I_k + \sum_{l \in \mathcal{H}} w_{lk}\kappa(\theta_l^{n-1}))\Bigg]\end{aligned} \tag{A.8}$$

$$\begin{aligned}\delta_{l,j}^{n-1} = \delta_{l,j}^{n-2} + \frac{\Delta t}{\tau}\Bigg[&\delta_{l,j}^{n-2}\sin\theta_l^{n-2} + \alpha(1+\cos\theta_l^{n-2})\left(\mathbf{1}_l(j) + \sum_{\substack{p \in \mathcal{H} \\ p \neq l}} w_{pl}\delta_{p,j}^{n-2}\kappa'(\theta_p^{n-2})\right) \\ &-\alpha\delta_{l,j}^{n-2}\sin\theta_l^{n-2}\left(I_l + \sum_{i \in \mathcal{I}} w_{il}\kappa(\theta_i^{n-2}) + \sum_{\substack{p \in \mathcal{H} \\ p \neq j}} w_{pl}\kappa(\theta_p^{n-2})\right)\Bigg]\end{aligned} \tag{A.9}$$

where $\mathbf{1}_l(j)$ denotes the indicator function which equals 1 when $l = j$ and 0 otherwise.

The backward pass for input layer baseline currents I_i is:

$$\frac{\partial E[n]}{\partial I_i} = \sum_{k \in \mathcal{O}} (\kappa(\theta_k^n) - T_k^n)\kappa'(\theta_k^n)\delta_{k,i}^n \tag{A.10}$$

$$\begin{aligned}\delta_{k,i}^n = \delta_{k,i}^{n-1} + \frac{\Delta t}{\tau}\Bigg[&\delta_{k,i}^{n-1}\sin\theta_k^{n-1} + \alpha(1+\cos\theta_k^{n-1})\sum_{j \in \mathcal{H}} w_{jk}\delta_{j,i}^{n-1}\kappa'(\theta_j^{n-1}) \\ &-\alpha\delta_{k,i}^{n-1}\sin\theta_k^{n-1}(I_k + \sum_{j \in \mathcal{H}} w_{jk}\kappa(\theta_j^{n-1}))\Bigg]\end{aligned} \tag{A.11}$$

$$\begin{aligned}\delta_{j,i}^{n-1} = \delta_{j,i}^{n-2} + \frac{\Delta t}{\tau} \bigl[& \delta_{j,i}^{n-2} \sin\theta_j^{n-2} \\ & +\alpha(1+\cos\theta_j^{n-2}) \left(w_{ij}\delta_{i,i}^{n-2}\kappa'(\theta_i^{n-2}) + \sum_{\substack{l\in\mathcal{H}\\ l\neq j}} w_{lj}\delta_{l,i}^{n-2}\kappa'(\theta_l^{n-2}) \right) \\ & -\alpha\delta_{j,i}^{n-2}\sin\theta_j^{n-2} \left(I_j + \sum_{i'\in\mathcal{I}} w_{i'j}\kappa(\theta_i'^{n-2}) + \sum_{\substack{l\in\mathcal{H}\\ l\neq j}} w_{lj}\kappa(\theta_l^{n-2}) \right) \Biggr] \end{aligned} \tag{A.12}$$

$$\delta_{i,i}^{n-2} = \delta_{i,i}^{n-3} + \frac{\Delta t}{\tau}\left[\delta_{i,i}^{n-3}\sin\theta_i^{n-2} + \alpha(1+\cos\theta_i^{n-3}) - \alpha\delta_{i,i}^{n-3}\sin\theta_i^{n-3}(I_i + X_i[n-3])\right] \tag{A.13}$$

A.4 Backward Pass for Network Weights

The dependence of the error function on the weights needs to be calculated. Denote $\frac{\partial\theta_k[n]}{\partial w_{jk}}$ as $\delta_{k,jk}^n$ for convenience. Starting with output layer weights w_{jk}, we have:

$$\frac{\partial E[n]}{\partial w_{jk}} = (\kappa(\theta_k[n]) - T_k[n])\kappa'(\theta_k^n)\delta_{k,jk}^n \tag{A.14}$$

$$\begin{aligned}\delta_{k,jk}^n = \delta_{k,jk}^{n-1} + \frac{\Delta t}{\tau}\Biggl[& \delta_{k,jk}^{n-1}\sin\theta_k^{n-1} + \alpha(1+\cos\theta_k^{n-1})\kappa(\theta_j^{n-1}) \\ & - \alpha\delta_{k,jk}^{n-1}\sin\theta_k^{n-1}\left(I_k + \sum_{j\in\mathcal{H}} w_{jk}\kappa(\theta_j^{n-1}) \right) \Biggr]\end{aligned} \tag{A.15}$$

The backward pass for hidden layer weights w_{lj} is:

$$\frac{\partial E[n]}{\partial w_{lj}} = \sum_{k\in\mathcal{O}} (\kappa(\theta_k^n) - T_k^n)\kappa'(\theta_k^n)\delta_{k,lj}^n \tag{A.16}$$

$$\begin{aligned}\delta^{n}_{k,lj} = \delta^{n-1}_{k,lj} + \frac{\Delta t}{\tau}\Bigg[&\delta^{n-1}_{k,lj}\sin\theta^{n-1}_{k} + \alpha(1+\cos\theta^{n-1}_{k})\sum_{h\in\mathcal{H}} w_{hk}\delta^{n-1}_{h,lj}\kappa'(\theta^{n-1}_{h}) \\ &-\alpha\delta^{n-1}_{k,lj}\sin\theta^{n-1}_{k}(I_k + \sum_{h\in\mathcal{H}} w_{hk}\kappa(\theta^{n-1}_{h}))\Bigg]\end{aligned} \tag{A.17}$$

$$\begin{aligned}\delta^{n-1}_{h,lj} = \delta^{n-2}_{h,lj} + \frac{\Delta t}{\tau}\Bigg[&\delta^{n-2}_{h,lj}\sin\theta^{n-2}_{h} \\ &+\alpha(1+\cos\theta^{n-2}_{h})\left(\sum_{\substack{p\in\mathcal{H}\\ p\neq h}}\big(\mathbf{1}_l(p)\mathbf{1}_j(h)\kappa(\theta^{n-2}_{l}) + w_{ph}\delta^{n-2}_{p,lj}\kappa'(\theta^{n-2}_{p})\big)\right) \\ &-\alpha\delta^{n-2}_{h,lj}\sin\theta^{n-2}_{h}\left(I_h + \sum_{i\in\mathcal{I}} w_{ih}\kappa(\theta^{n-2}_{i}) + \sum_{\substack{p\in\mathcal{H}\\ p\neq h}} w_{ph}\kappa(\theta^{n-2}_{p})\right)\Bigg]\end{aligned} \tag{A.18}$$

The backward pass for input layer weights w_{ij} is:

$$\frac{\partial E[n]}{\partial w_{ij}} = \sum_{k\in\mathcal{O}}(\kappa(\theta^{n}_{k}) - \hat{\Upsilon}^{n}_{k})\kappa'(\theta^{n}_{k})\delta^{n}_{k,ij} \tag{A.19}$$

$$\begin{aligned}\delta^{n}_{k,ij} = \delta^{n-1}_{k,ij} + \frac{\Delta t}{\tau}\Bigg[&\delta^{n-1}_{k,ij}\sin\theta^{n-1}_{k} + \alpha(1+\cos\theta^{n-1}_{k})\sum_{h\in\mathcal{H}} w_{hk}\delta^{n-1}_{h,ij}\kappa'(\theta^{n-1}_{h}) \\ &-\alpha\delta^{n-1}_{k,ij}\sin\theta^{n-1}_{k}(I_0 + \sum_{h\in\mathcal{H}} w_{hk}\kappa(\theta^{n-1}_{h}))\Bigg]\end{aligned} \tag{A.20}$$

$$
\begin{aligned}
\delta_{h,ij}^{n-1} = \delta_{h,ij}^{n-2} + \frac{\Delta t}{\tau}\Bigg[& \delta_{h,ij}^{n-2}\sin\theta_h^{n-2} \\
& + \alpha(1+\cos\theta_h^{n-2})\left(\mathbf{1}_j(h)\kappa(\theta_i^{n-2}) + \sum_{\substack{p\in\mathcal{H}\\ p\neq h}}\left(w_{ph}\delta_{p,ij}^{n-2}\kappa'(\theta_p^{n-2})\right)\right) \\
& -\alpha\delta_{h,ij}^{n-2}\sin\theta_h^{n-2}\left(I_h + \sum_{i'\in\mathcal{I}} w_{i'h}\kappa(\theta_i'^{n-2}) + \sum_{\substack{p\in\mathcal{H}\\ p\neq h}} w_{ph}\kappa(\theta_p^{n-2})\right)\Bigg]
\end{aligned}
\tag{A.21}
$$

Therefore the learning rules can be summarized as:

$$\Delta w_{ij} = -\frac{\eta_w}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial w_{ij}} \tag{A.22}$$

$$\Delta w_{lj} = -\frac{\eta_w}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial w_{lj}} \tag{A.23}$$

$$\Delta w_{jk} = -\frac{\eta_w}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial w_{jk}} \tag{A.24}$$

$$\Delta I_i = -\frac{\eta_I}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial I_i} \tag{A.25}$$

$$\Delta I_j = -\frac{\eta_I}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial I_j} \tag{A.26}$$

$$\Delta I_k = -\frac{\eta_I}{\mathcal{N}}\sum_{n=0}^{\mathcal{N}}\frac{\partial E[n]}{\partial I_k} \tag{A.27}$$

where $\mathcal{N}$ is the number of time steps and η_w, η_I are the learning rates for the weights and baseline currents, respectively.