2018

Implementation Costs of Spiking versus Rate-Based ANNs

Lacie Renee Stiffler
University of South Carolina - Columbia

Implementation Costs of Spiking versus Rate-Based ANNs

by

Lacie Renee Stiffler

Bachelor of Science in Engineering
University of South Carolina, 2016

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2018

Accepted by:

Jason D. Bakos, Director of Thesis

Duncan Buell, Reader

Gabriel A. Terejanu, Reader

Cheryl L. Addy, Vice Provost and Dean of the Graduate School

Dedication

To my parents, my husband, and the clan that welcomed me as one of their own; in short, my family.

Acknowledgments

This thesis is the culmination of several years of scholarship and would not have been possible without the generous contributions of many different people.

My mentor, Jason Bakos, had the most profound impact on my development as a research scientist and provided valuable advice along the way.

Gabriel Terejanu generously gave his knowledge throughout the various developmental stages of this thesis.

Even though most of my friends and family did not always understand what I was working on, they consistently asked me about progress, listened when I had tough days, and celebrated with me when I had breakthroughs.

The most deserving of my gratitude is my husband, Nick. Without his support, I most likely would not have pursued a Master's degree. I definitely would not have been able to overcome the tough times such a venture entails without his solidarity.

Abstract

Artificial neural networks are an effective machine learning technique for a variety of data sets and domains, but exploiting the inherent parallelism in neural networks requires specialized hardware. Typically, computing the output of each neuron requires many multiplications, evaluation of a transcendental activation function, and transfer of its output to a large number of other neurons. These restrictions become more expensive when internal values are represented with increasingly higher data precision. A spiking neural network eliminates the limitations of typical rate-based neural networks by reducing neuron output and synapse weights to one-bit values, eliminating hardware multipliers, and simplifying the activation function. However, a spiking neural network requires a larger number of neurons than what is needed in a comparable rate-based network. In order to determine if the benefits of spiking neural networks outweigh the costs, we designed the smallest spiking neural network and rate-based artificial neural network that achieved 90% or comparable testing accuracy on the MNIST data set. After estimating the FPGA storage requirements for synapse values of each network, we concluded rate-based neural networks need significantly fewer bits than spiking neural networks.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

ANN ..... Artificial Neural Network

ASIC ..... Application Specific Integrated Circuit

BNN ..... Binarized Neural Network

CDF ..... Cumulative Distribution Function

CNN ..... Convolutional Neural Network

CPU ..... Central Processing Unit

DNN ..... Deep Neural Network

FPGA ..... Field Programmable Gate Array

GPGPU ..... General-Purpose Graphical Processing Unit

GPU ..... Graphics Processing Unit

RAM ..... Random Access Memory

SNN ..... Spiking Neural Network

STDP ..... Spike Time Dependent Plasticity

List of Symbols

$S(x)$	Sigmoid solution of input x
x	Network Input
i	Input Node/Neuron Index
j	Target Neuron Index
I_j	Summed Input to Neuron j
x_i	Binary Spike State of Input Node/Neuron i
$\widetilde{x}_i$	A Continuous Value in Range [0,1] Representing x_i
c_{ij}	Binary Synaptic Connection Between Node/Neuron i and Neuron j
$\widetilde{c}_{ij}$	A Continuous Value in Range [0,1] Representing c_{ij}
s_{ij}	Synaptic Strength Between Node/Neuron i and Neuron j, +1 or -1
b_j	Bias Term for Neuron j
n_j	Binary Neuron j Output
$\widetilde{n}_j$	A Continuous Value in Range [0,1] Representing n_j
μ_j	Gaussian Mean of Neuron j's Summed Input
σ_j^2	Gaussian Variance of Neuron j's Summed Input
k	Class of Output
y_k	Binary Class Label
p_k	Probability the Average Spike Count for Class k is Greater Than 0.5
a	Learning Rate
erf	Error Function

Chapter 1

Introduction

Interest in artificial neural networks (ANNs) for tasks such as computer vision and image, video, and audio classification has increased in recent years. However, ANNs are expensive in terms of computing and memory resources across all platforms used for deployment: Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs). CPU and GPU implementations are aided by libraries and frameworks to be as efficient as possible in terms of computing time and resource management. Some popular tools include NVIDIA's cuDNN [12], TensorFlow [31], Torch [34], Caffe [4], and Theano [32]. ASICs are designed to have all necessary resources for a narrow range of ANNs. This limitation on network type and topology can make an ASIC cost prohibitive for research purposes. ANN resource demands are most noticeable on FPGAs. As a mobile embedded platform, FPGAs have the most limited computing and memory resources. Even though ANNs have been implemented on FPGAs since 1992 [11] and FPGAs have grown in size and amount of resources, these networks generally have fewer neurons and layers than those deployed on CPUs, GPUs, and ASICs.

One way FPGAs can make efficient use of available resources and make room for the largest possibly network is to reduce data precision used within an ANN, which can be arbitrarily set. Reduced numerical precision poses to significant advantages: decreased memory needed for a single value and smaller multipliers for the multiply-accumulate operations inherent to ANNs. These benefits help to in-

crease performance and reduce power consumption of FPGAs and ASICs alike. The minimum precision required by a neural network varies depending on type and desired application, requiring some amount of experimentation to determine the exact precision.

The smallest possible data precision is 1-bit that can only communicate information with zeros and ones. An example of neural network using this extreme is the Spiking Neural Network (SNN). Until IBM's TrueNorth ASIC [16] [13], SNNs had not been considered for use in machine learning due to lack of training methods that suited with binary neural network. IBM devised a unique training algorithm for SNNs that uses the familiar backpropagation method of learning. The 1-bit needed for data eliminates hardware multipliers on FPGAs since multiply-accumulate operations are replaced by simple accumulates, uses a simpler neuron activation function, and lends itself to a simpler neuron interconnect. However, these benefits come at a cost. SNNs generally have more neurons than comparative ANNs that use higher data precisions. This trade-off has not been explored, leaving in question if the resource savings of SNNs outweighs the cost of additional neurons.

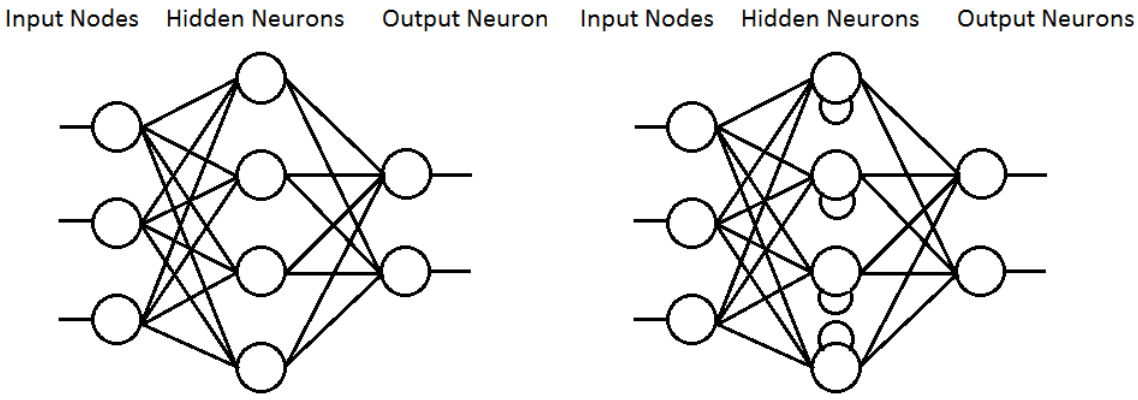
In order to determine if the SNN trade-off is beneficial, this thesis estimates the FPGA storage requirements for synapse weights of a rate-based ANN and a SNN that achieve a 90% or comparable testing accuracy for the MNIST data set [23]. There are currently no publicly available tools for training SNNs, so first we develope an algorithm based on IBM's backpropagation technique [13] presented with their TrueNorth ASIC while keeping FPGA architecture in mind. We conclude that the SNN requires so many more neurons than a comparable ANN using higher data precision to negate the resource savings signified by the 1-bit data precision.

Chapter 2

Background

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are a fundamental algorithm in machine learning most commonly used for classification, e.g. pattern recognition. The theory concerning the structure, usage,and training of ANNs has become well understood after several decades of research, but researchers have not been able to deploy the networks until recently due to the high computational costs associated with training the networks. The introduction and rapid advancement of general-purpose graphical processor units (GPGPUs) and field programmable gate arrays (FPGAs) have allowed the widespread use of ANNs.



(a) Feed Forward Neural Network Topology (b) Recurrent Neural Network Topology

Figure 2.1: Common Neural Network Topologies

ANNs are primarily modeled as a graph consisting of interconnected neurons. Each neuron accepts multiple inputs called synapses and gives a single value as output.

The chosen network topology must be tailored to the desired application and training algorithm to be used. Commonly used topologies are acyclic and cyclic, which are referred to as "feed forward" and "recurrent" networks, respectively, seen in Figure 2.1.

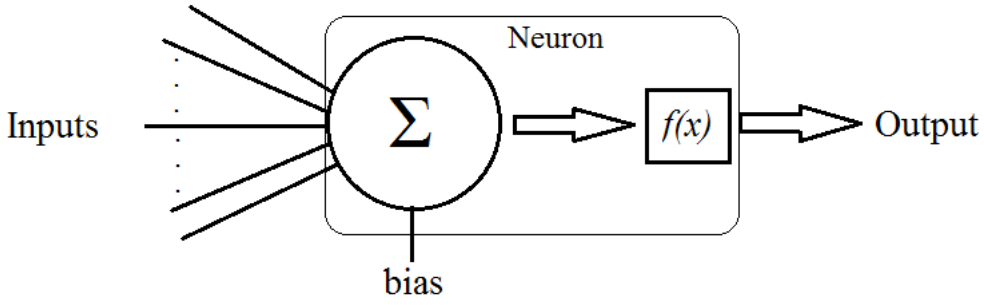


Figure 2.2: Basic Neuron Structure

Generally, each neuron uses the weighted sum of synapse values to compute a continuous activation function $f(x)$, Figure 2.2. The output from the function is the neuron's final output, either to subsequent neurons or the end of the network. Continuous activation functions give results between 0 and 1, making them suitable for both analog and digital functions. A popular activation function is the sigmoid function, $S(x) = \frac{1}{1+e^{-x}}$, which approaches 0 for negative input and approaches 1 for positive input [30], see Figure 2.3. Implementing the sigmoid function on an FPGA is nontrivial and typically requires a polynomial approximation.

2.2 Spiking Neural Networks

Neural networks have been separated into three generations. The first generation contains the McCulloch and Pitts-type neurons whose output signals were set to zero or one. Rate-based ANNs make up the second generation where continuous activation functions give values between zero and one. The third and current generation is timing-based ANNs (Spiking Neural Networks, SNNs) that store information in the timing of spikes [18] and more closely model the behavior of biological neurons.

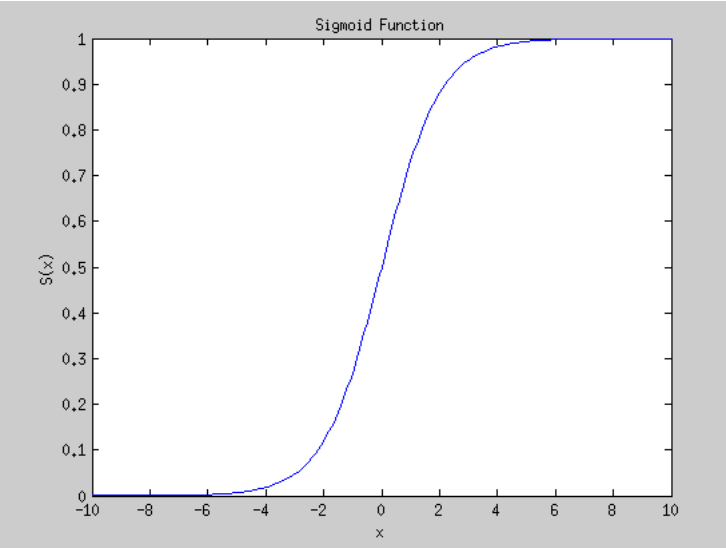


Figure 2.3: Graph of Sigmoid Function

Neurons in rate-based ANNs communicate with real numerical values created by the activation functions, such as the sigmoid function discussed in the previous section. Timing based networks communicate with spikes carried on the synapses, where the frequency and spacing of the spikes convey the necessary information. Spikes are typically represented with ones and zeros are used in the absence of a spike. This behavior closely resembles biological neurons. Each neuron collects signals from surrounding neurons that change the ionic level (or membrane potential) in the neuron. Once the level reaches a threshold, the neuron fires and transmits a signal downstream in the system, demonstrated in Figure 2.4.

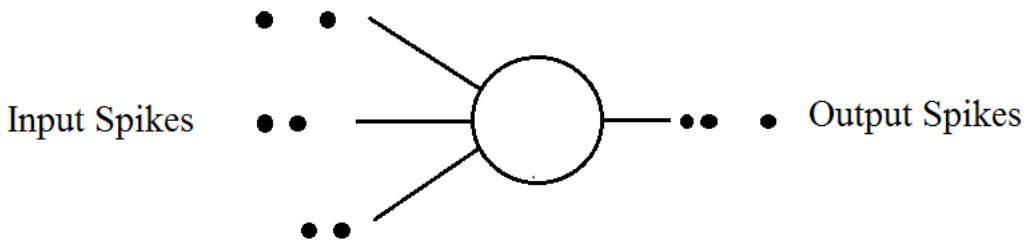


Figure 2.4: A Neuron in a SNN

In order to model the membrane potential of biological neurons, each neuron

in a SNN maintains and updates internal state with a neuron update function. The function combines the weighted pulses arriving on the synapses with the current state and produces a series of spikes to send downstream. Several update functions exist, each with varying degrees of compatibility with biological neurons and associated computational costs [27]. The update function has to be chosen with care since it will greatly impact the SNN's performance, implementation, and training method. Popular choices are the "leaky integrate ad fire," Hodgkins-Huxley, and Izhikevich models [21].

2.2.1 Training SNNs

Rate-based ANNs are typically trained with gradient descent, a method that cannot be directly applied to SNNs due to the discontinuous-in-time nature of spiking neurons. While training methods for SNNs are not as well developed as those for traditional ANNs, there are several supervised and unsupervised approaches available. One unsupervised approach that has received a lot of attention recently is Spike Time Dependent Plasticity (STDP). This method searches for relationships between firing neurons and adjusts the weights to strengthen those relationships [18]. Supervised training methods include [22]:

- SpikeProp [2],
- ReSuMe [29],
- statistical methods that optimize synapse weights to maximize the likelihood of firing at the desired times [15],
- linear algebra methods where the approximate target firing patterns are determined from the input patterns and solve for the weights using iterative methods [5],

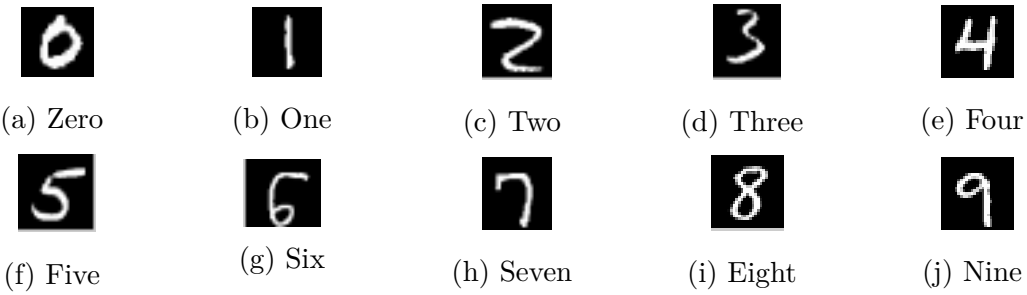
- evolutionary methods [1], and
- spike-based Hebbian methods where specific neurons are associated with training samples. The weights of the neurons are adjusted so that the neuron fires when its corresponding training sample is given to the network [24].

There is an emerging approach that uses probabilistic backpropagation to bridge the gap between gradient descent for traditional ANNs and the discontinuous-in-time nature of spiking neurons. Probabilistic backpropagation uses probabilities of events occuring, such as a neuron firing, during training then converts the probabilities to binary or trinary values for run time. IBM used this method for digit classification on their TrueNorth chip [13]. Probabilistic backpropagation has also been used to train fully connected networks with binary neurons and binary or trinary synapses [38] [8].

2.3 Binnarized Neural Networks

There is a neural network that is similar to an SNN called a Binarized Neural Network (BNN) that uses binary weights and activations at run time [9] [35] [10]. Values are typically constrained to -1 and +1, similar to SNNs restriction to zero and one. For both networks, this limitation replaces multiply-accumulate operations with simple accumulations, saving space and power in hardware implementations.

While BNNs and SNNs behave similarly on hardware, they are implemented differently and for varying reasons. BNNs are inteded to reduce the precision of Deep Neural Networks (DNNs), specifically Convolutional Neural Networks (CNNs), for deployment on FPGAs, which are known to perform better when using binary operations over floating point [35]. SNNs are binary by nature and are not a means to translate DNNs into low precision. Communicating via spikes easily translates to using ones to indicate a spike and zeros otherwise. With this in mind, SNNs cannot be trained with methods used for DDNs like a BNN can with the necessary consid-



(a) Zero (b) One (c) Two (d) Three (e) Four

(f) Five (g) Six (h) Seven (i) Eight (j) Nine

Figure 2.5: Examples of digits in the MNIST dataset

erations for binarization included during training. SNNs represent a biological brain and keep an internal state to replicate the ionic level of a neuron to do so in many implementations. BNNs do not keep such a state.

2.4 MNIST Dataset

In a collaborative effort, LeCun, Cortes, and Burges created the MNIST dataset of handwritten digits [23]. There is a training set of 60,000 examples and a testing set with 10,000 examples, all of which have been normalized with respect to size and centered in a 28x28 pixel image. The MNIST dataset is beneficial to use when exploring various learning techniques and pattern recognition methods while desiring to use real-world data because there is minimal effort needed to preprocess and format the data into usable files. A sample of the MNIST dataset is shown in Figure 2.5.

Chapter 3

Related Work

While this thesis does not deploy a neural network on an FPGA or ASIC, we want to know how networks perform on each platform and compare estimated resource usage of rate-based ANNs and SNNs as if they are to be deployed on an FPGA.

3.1 FPGA-Based ANNs

The first published work of an FPGA-based ANN occured in 1992 [11] [39] where the authors used a feed forward network with one hidden layer. Cox and Blanz used 8-bit integer synapse weights and had to support 224 8-bit integer multiplies per cycle [11]. The authors minimized resource usage to allow for the largest ANN possible to fit on the FPGA by implementing the multipliers as single-input fixed-multipliers. These new multipliers had constant weights encoded into their design. In this implementation, each synapse is associated with a specific multiplier, meaning that it can only support ANNs with synapses less than or equal to the maximum number of possible multipliers. This limitation of multipliers on synapses greatly impairs the size of ANNs that can be implemented.

It has been mentioned before that recurrent ANNs and some SNNs both maintain internal state. For recurrent networks to accomplish this, additional synapse connections store data dependencies for later use, causing increased computation time during training and deployment. Recent work has been conducted on FPGA-based recurrent neural networks, applied to natural language processing [25]. In this work, the recurrent connections occur in the hidden layer as seen in Figure 2.1b. Training

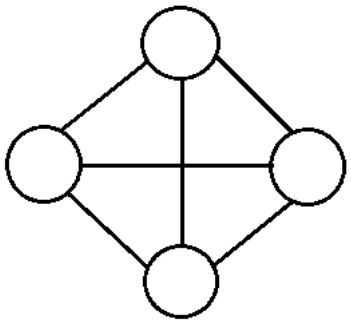


Figure 3.1: Fully-Connected Unlayerd Neural Network Topology

this network used backpropagation through time to unfold the network over three time steps that can then follow the typical training of feed forward networks.

A way to increase the size of an FPGA-based layered feed forward ANN is to use layer multiplexing [20]. Layer multiplexing only implements the largest layer, excluding the input layer, and gives each neuron the maximum number of inputs. For example, a network structured with eight input neurons, five hidden neurons, and three output neurons would be implemented using five neurons with eight inputs each. The single layer is multiplexed by a control block to execute the behavior of all layers of the network. This method allowed Himavathi et. al. to implement 31 neurons that only used 64% of the available slices on the Xilinx XCV400hq240.

3.2 FPGA-Based SNNs

Thomas and Luk designed a SNN with 1,024 neurons arranged in a fully-connected unlayered configuration [33] seen in Figure 3.1. The RAM (Random Access Memory) holds synapse weights on-chip in a 1,024 element by 36-bit array, requiring 1,024 clock cycles just to read weights and complete calculations. Performance of the design was evaluated using synthetic workloads instead of an application. Thomas and Luk found a range of 35% slow down to a 17% speedup when comparing the FPGA SNN to a GPU implementation due to the varying firing activity generated by the synthetic workload.

The primary limitation to SNNs on FPGAs is space, i.e. how many neurons can fit on the FPGA device? Thomas and Luk were able to fit 1,024 neurons on their device with careful planning [33] but this is not enough for sufficiently large-scale SNNs. Recent research has been focused on creating a better neuron architecture in order to improve the scalability of FPGA-based SNNs [37]. Wan et al. developed an efficient neuron architecture that uses a sharing mechanism at the synapse and neuron levels to reduce the silicon area and resources occupied by each neuron.

At the center of Wan et al.'s design is a neuron computing core that emulates the synaptic behavior within the neuron [37]. Each neuron block has a computing core shared by multiple synapses connected to the neuron. To complete the efficient neuron architecture, the neuron block is enclosed in a layer module with a RAM, a decoder, a controller, and a packet generator. A layer module has multiple neurons associated with it and all neurons share the neuron block where the computations take place. With this sharing mechanism in the architecture, Wan et al. were able to accommodate up to 181 neurons in each layer module, totalling 3,982 neurons on the FPGA [37].

3.3 ASIC-Based SNNs

One of the recent chips that implements SNNs is the IBM TrueNorth [16] [13] [6]. The architecture of the chip is an array of neural cores arranged in a 64x64 square. Each core contains 256 neurons, which brings a total of 2^{20} neurons on the TrueNorth, and has 256 inputs and 256 outputs, equaling 65,536 synapses per core. The TrueNorth is able to achieve such high density because it is an ASIC, which typically have 10x more density that FPGAs.

Esser et al. describe how they train the SNN for the IBM TrueNorth chip using backpropagation in [13]. Spikes and discrete synapses are treated as continuous probabilities, which are sampled to create one or more networks that are merged together

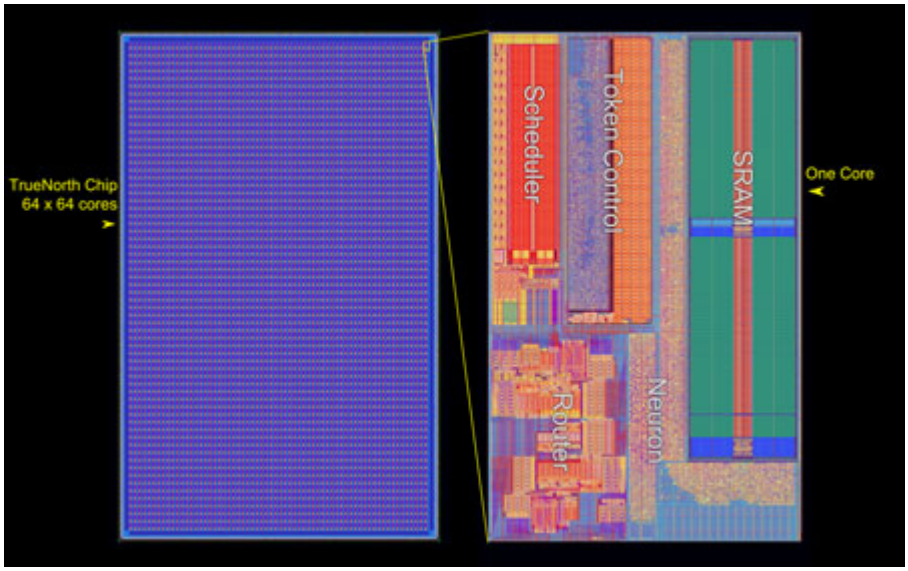


Figure 3.2: IBM TrueNorth

using ensemble averaging. They tested the training method with the MNIST dataset and were able to achieve 99.42% accuracy in a high performance network and 92.7% accuracy in a high power efficiency network.

The Google TPU [19], Nvidia Volta [36], SpiNNaker [17] [14], DianNao [7], and Darwin [26] are other platforms and co-processors working to implement SNNs on a large scale to meet research needs.

Table 3.1: Summary of Related Work

	FPGA-based ANNs			**FPGA-based SNNs**		**ASIC-based SNNs**
Source	[11]	[20]	[25]	[33]	[37]	[13]
Number of Neurons	30	31	1024 Hidden	1024	3982	2^{20}
Network Connectivity	Fully interconnected, feedforward	Fully interconnected, feedforward	Fully interconnected, feedforward	Fully connected	Fully interconnected, feedforward	Each core is fully connected but sparse connections between cores.
Number of Synapses	224	130	10,836,992	$1,024^2$	398,200	intra-core: 64*64*65536
Platform	9U VME card	Xilinx XCV400hq240	Xilinx Virtex6 LX760	Xilinx Virtex5 xc5vlx330t	Xilinx XC7Z020	IBM TrueNorth

Chapter 4

SNN Training Algorithm

Although training tools for rate-based ANNs are relatively common, there are currently no off-the-shelf training tools for SNNs. As a result, part of this thesis was to develop a training tool for the SNN based on IBM's methods for the TrueNorth chip [13] in Matlab. To our knowledge, IBM has not released a library for their technique to be used by researchers. Most of the details of this chapter come from [13] but there were some aspects of the IBM SNN training implementation that were unclear or not fully specified so we have detailed everything here to show the full training process. Our recreation targets FPGA boards as the deployment platform instead of being restricted to the TrueNorth, allowing for a broader range of neural network topologies.

Our SNN uses zeros and ones for most values: inputs, synapse connections, and outputs; one value, the synapse strength, can be -1 or +1. In order to train this type of network, we have to use probablistic backpropagation. This requires a separate training network that uses probabilistic interpretations of events; we focus on using probabilities of a value being one. These probabilities are then translated to zeros and ones when creating the deployment network that would be be used on the desired platform.

4.1 Neuron Model

Each neuron in the network sums its input using $I_j = \sum_x x_i c_{ij} s_{ij} + b_j$ and the activation function uses a history-free thresholding equation

$$n_j = \begin{cases} 1 & if\ I_j > 0, \\ 0 & otherwise \end{cases},$$

where x_i is the input to the neuron, c_{ij} indicates if the synapse is connected, s_{ij} is the synaptic strength, and b_j is the bias term, to determine output. Figure 4.1 shows the neuron structure.

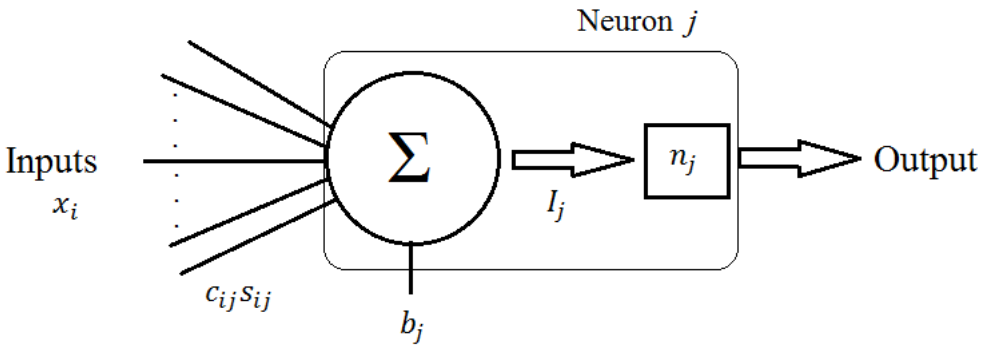


Figure 4.1: SNN Neuron Structure

4.2 Data Pre-Processing

In [13], IBM briefly mentions rescaling the input image data into a continuous value in the range [0,1] without detailing how the rescaling is completed. Dr. Brownlee describes two methods for scaling data, normalization and standardization [3]. Normalization rescales the input to the range [0,1] while standardization centers data distribution on 0 and sets standard deviation to 1. Based on these definitions, it appears that IBM used normalization to rescale the MNIST dataset.

Normalization requires the minimum and maximum values of each attribute; in the MNIST dataset, the attributes are the individual pixels in the 28x28 pixel images.

The pixels of an image are arranged in a single row with 784 attributes total and a training set has an arbitrary number of images up to 60,000. The result is a matrix where the number of rows is the number of images in the set and the number of columns is consistently 784. Performing Matlab's min and max functions on the matrix gives a row vector of the minimum and maximum value of each column (pixel), respectively. Algorithm 1 shows how each scaled value is calculated, resulting in a matrix where the dimensions remain as the number of training images by 784. Normalization is also performed on the full set of testing images.

Algorithm 1 Rescaling Input Data

```
%Note: min and max row matrices are generated prior to the nested for loops
for each row in matrix, r do
    for each column, c do
        if max(c) == 0 then
            scaled_value(r,c) = 0
        else
            scaled_value(r,c) = (value(r,c) - min(1,c))/(max(1,c) - min(1,c))
        end if
    end for
end for
```

4.3 Network Initialization

This thesis used a fully inter-connected feed forward topology with one hidden layer. In order to train this network with backpropagation, continuous values have to be used for the properties to be learned, namely the weight of each synapse. Part of the weight is represented as synapse strength, either -1 or +1, that is established before training begins and remains constant throughout the process. The strengths are set up as a matrix for the hidden and output layers with the neurons in each layer as the rows and the input synapses to each neuron as the columns. We assume an even number of neurons per layer so that a neuron can have an equal number of -1 and +1 synapse strengths. For example, a network with two input nodes, four hidden

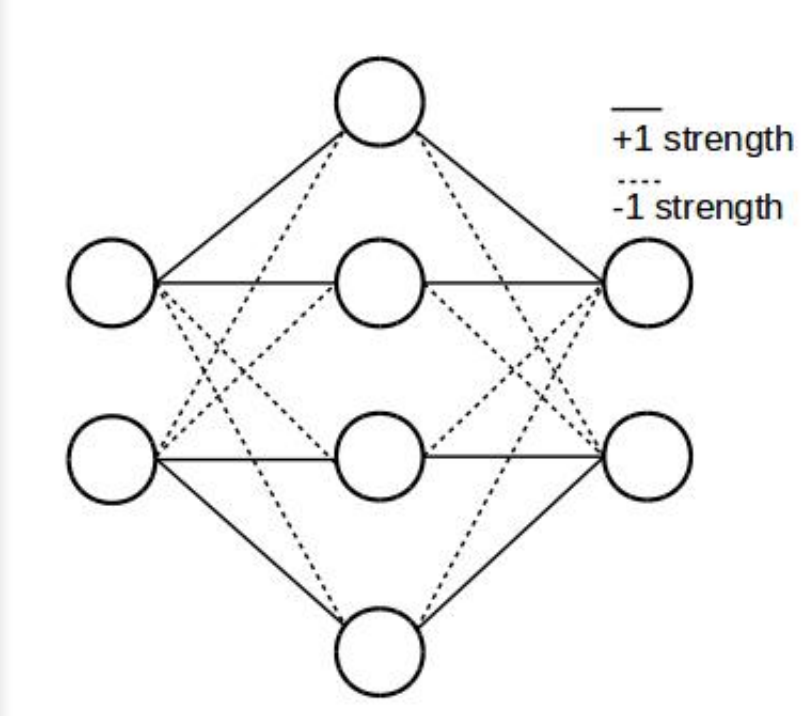


Figure 4.2: The Distribution of +1 and -1 Synapse Strengths

neurons, and two output neurons will look like Figure 4.2. The synapse connections are represented by the solid and dashed lines, where a solid line indicates a synaptic strength of +1 and a dashed line shows a strength of -1. Because of our restriction to even numbers of neurons per layer, Figure 4.2 shows how each neuron has an equal number of +1 and -1 strengths coming in and going out.

The rest of a synapse weight is the connection, c. During training, connection is a probability initialized from a uniform random distribution over the range [0,1] multiplied by 0.1 and interpreted as the probability of the connection being one. Connection probabilities are also setup in a matrix with the same dimensions as the corresponding synapse strength matrix.

4.4 Training

4.4.1 Input

The matrix generated from pre-processing the MNIST training images is used as input to the training network. The normalized values are interpreted as the probability of the pixel being 1 in the deployment network. There are input nodes in the network that only pass the normalized values of a single image to the hidden layer. Since the network topology is fully inter-connected, each hidden neuron receives all input values. Subsequently, input to each output neuron is the output from all hidden neurons as seen in Figure 2.1a.

4.4.2 Forward Pass

Based on the inputs, the probability of each neuron in each subsequent layer firing is calculated as a function of the neuron's synaptic connection probabilities using a Cumulative Distribution Function (CDF) of a Gaussian. Each synapse between target neuron j and all input nodes or neurons i has both a synaptic probability of being connected, $\widetilde{c}_{ij}$, and synapse strength, s_{ij}. $\widetilde{x}_i$ is the probability of the input being 1. Mean and variance are used as parameters to determine if a neuron fires:

$$\mu_j = b_j + \sum_i \widetilde{x}_i \widetilde{c}_{ij} s_{ij}$$

$${\sigma_j}^2 = \sum_i \widetilde{x}_i \widetilde{c}_{ij} (1 - \widetilde{x}_i \widetilde{c}_{ij}) {s_{ij}}^2$$

The probability of a neuron firing is derived using the complimentary CDF of a Gaussian:

$$\widetilde{n}_j = 1 - \frac{1}{2}\left[1 + erf\left(\frac{\Theta - \mu_j}{\sqrt{2{\sigma_j}^2}}\right)\right]$$

where erf is the error function and $\Theta = 0$.

4.4.3 Backward Pass

The backward pass adjusts the synapse connection probabilities using the gradient descent of a log-loss function:

$$E = -\sum_k [y_k log(p_k) + (1 - y_k) log(1 - p_k)]$$

where, for class k, y_k is the binary label indicating if this is the correct class of the input image and p_k is the probability that the average spike count for k is greater than 0.5 ($\widetilde{n}_j$ with $\Theta = 0.5$). The partial derivative of the loss function with respect to the synapse connection probabilities $\widetilde{c}_{ij}$ is used to find the gradient at each synapse. It is computed using the chain rule

$$\frac{\partial E}{\partial \widetilde{c}_{ij}} = \frac{\partial E}{\partial \widetilde{n}_j} \frac{\partial \widetilde{n}_j}{\partial \widetilde{c}_{ij}},$$

shown in Appendix A. The training algorithm then calculates the change in the synapse connection probabilities at each neuron:

$$\Delta \widetilde{c}_{ij} = -a \frac{\partial E}{\partial \widetilde{c}_{ij}},$$

where a is the learning rate.

Using the gradient from the log-loss function only works for the output layer since we know what each neuron should fire. For the hidden layer, the training algorithm has to backpropagate the output neuron errors through summations, shown in Algorithm 2. The summation replaces the $\frac{\partial E}{\partial \widetilde{n}_j}$, represented by e_n, portion of the gradient descent calculation. n_mu and mu_c correspond to $\frac{\partial \widetilde{n}_j}{\partial \mu_j}$ and $\frac{\partial \mu_j}{\partial \widetilde{c}_{ij}}$, respectively. Algorithm 2 was developed by comparing the equations used for the SNN to those used in the traditional rate-based ANN we developed, shown in Appendix B.

The bias of each hidden and output neuron is also trained using gradient descent:

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial \widetilde{n}_j} \frac{\partial \widetilde{n}_j}{\partial b_j} \; and \; \Delta b_j = -a \frac{\partial E}{\partial b_j}.$$

More details are given in Appendix A.

Algorithm 2 Error Propagation to Hidden Layer

for each hidden neuron, l **do**
  sum = 0
  for each output neuron, n **do**
    % Note: out_bias = e_n * n_mu and out_$\tilde{c}$ = e_n * n_mu * mu_c
    % from the calculation of the gradient descent at the output layer
    sum = sum + (out_bias(n,l) * out_$\tilde{c}$(n,l) * out_strengths(n,l))
  end for
  n_mu = $(1/(\sigma_l\sqrt{2\pi}))$ * exp(-$(\Theta$-$\mu_l)^2/(2\sigma_l^2)$)
  mu_c = input_data .* hid_strengths(l)
  hid_$\tilde{c}$(l) = sum * n_mu * mu_c
  hid_bias(l) = sum * n_mu
end for

Connection probabilities and bias terms are then checked to determine if they need to be snapped to their appropriate ranges, [0,1] and [-255,255], respectively.

4.5 From Training to Deployment

The training algorithm works on a training network where each synapse has been assigned a connection probability. In order to create the network that is deployed on hardware, the deployment network, each synapse connection in the training network is randomly sampled to determine if the synapse is zero or one in the deployment network, as seen in Algorithm 3. If a higher accuracy is desired, multiple deployment networks can be created, called ensembles, and operated in parallel. Bias terms can have fractional portions during training that must be dropped for the deployment network, we used the floor of each number.

4.6 Output Neurons and Classification

Each class label, 0 to 9, is associated with multiple output neurons and the prediction is based on the average of all neurons assigned to each label. This thesis used eight neurons per class, totalling 80 output neurons. After the average for each class is calculated, the highest average determines the prediction. An example is shown in

Algorithm 3 Random Sampling

```
for each row, r do
    for each column, c do
        y = 0.1 * rand
        if y <= probability(r,c) then
            spike(r,c) = 1
        else
            spike(r,c) = 0
        end if
    end for
end for
```

Table 4.1 where the digit classes are listed at the top of the table, the output from all 80 neurons are listed in the middle, and the average of each class is listed at the bottom. Based on the averages, the prediction in this example is digit 9.

Table 4.1: Example of output from SNN Output Neurons

Digit Classes	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	0	0	0	0	1	0	0	0	0	1
	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	1	0	0	0	0	0
Output	0	0	0	0	1	0	0	0	0	1
	0	0	0	0	0	1	0	0	0	1
	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	1	0	0	0	0	0
Average	0	0	0	0	1/2	1/8	0	0	0	3/4

4.7 Network Optimization

In an effort to achieve a high testing accuracy, we implemented momentum and learning rate decay. Momentum is used only on the hidden layer neurons at a rate of 0.9. Our learning rate decay occurs on a fixed schedule starting with alpha equal to 0.1 and multiplying by 0.1 every 250 epochs. More details are given in Chapter 7.

Chapter 5

Preliminary Data

As mentioned in Chapter 4, IBM does not provide all details of their SNN training algorithm in [13]. In order to fill in the missing information, we developed our training algorithm in steps by starting with the simplest network configuration then progressing to the full feedforward network. Each configuration builds on the previous one and helped answer questions IBM had left open.

5.1 No Hidden Layer, One Output Neuron

The first SNN configuration only has the 784 input nodes and one output neuron. The output neuron fires 1 if a zero digit image is given to the network and 0 otherwise. We used this configuration to confirm the proper input image data rescaling method and the proper bias update equation since these were the first two uncertainties we came across. We had three possible rescaling options and two possible bias update equations:

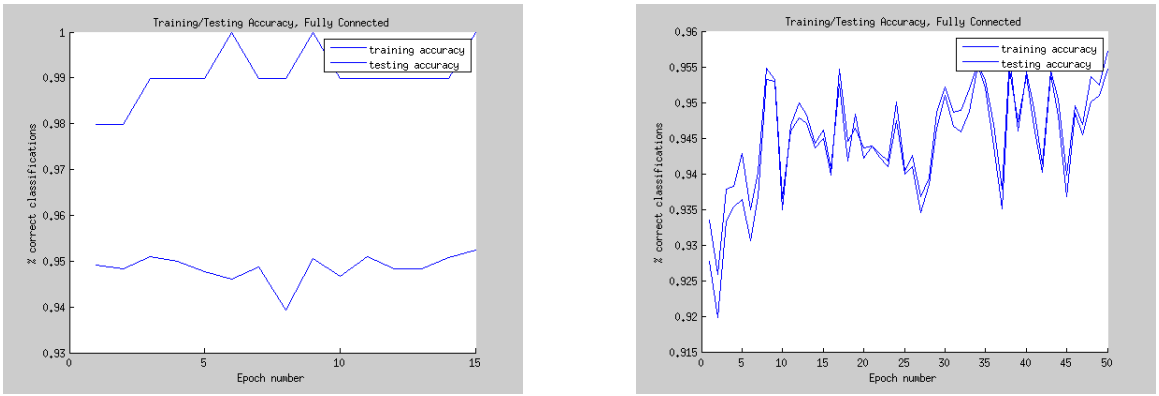
- Rescaling
  - Normalization
  - Standardization
  - Averaging Each Pixel Across All Training/Testing Images
- Bias Update Equation
  - $\Delta b_j = -a * \frac{\partial E}{\partial b_j}$ where $\frac{\partial E}{\partial b_j} = 1$

– $\Delta b_j = -a * (\frac{\partial E}{\partial \widetilde{n}_j} * \frac{\partial \widetilde{n}_j}{\partial \mu_j} * \frac{\partial \mu_j}{\partial b_j})$ where $\frac{\partial \mu_j}{\partial b_j} = 1$

We concluded that normalization is the best way to rescale the input image data and that $\Delta b_j = -a*(\frac{\partial E}{\partial \widetilde{n}_j} * \frac{\partial \widetilde{n}_j}{\partial \mu_j} * \frac{\partial \mu_j}{\partial b_j})$, where $\frac{\partial \mu_j}{\partial b_j} = 1$, is the appropriate way to update a neuron's bias term. We also added the bias bounds check during training and dropped the fractional bits from the bias terms during evaluation of the deployment SNN using Matlab's floor method. Figure 5.1a shows the resulting training and testing accuracies of this configuration and Table 5.2 includes testing parameters such as the number of epochs used during training this configuration and subsequent ones.

5.2 Hidden Layer, One Output Neuron

We next chose to add a hidden layer to the SNN, wanting to perfect the backward pass and properly backpropagate the output error to the hidden layer. It was because of this configuration that we compared ANN and SNN equations, seen in Appendix B, to create Algorithm 2. With this algorithm, we were able to correctly perform gradient descent on the hidden layer. Unfortunately, "not a number" values began to appear in the SNN during training. This was due to zeros and ones as whole numbers being assigned to variables instead of decimal values during training. We rectified



(a) No Hidden Layer (b) Hidden Layer

Figure 5.1: SNNs with Only One Output Neuron

this issue by checking for zero and one as special values, discussed in more depth in Appendix B. Figure 5.1b shows the accuracies of this configuration.

5.3 No Hidden Layer, 10 Output Neurons

For this configuration, we took out the hidden layer again to focus solely on the output layer. We added enough output neurons to have one for each digit class in the MNIST dataset, 0-9. A neuron fires one if an image of its digit is presented to the SNN and zero otherwise. The question we explored here is what do with the Σ in $E = -\sum_k [y_k log(p_k) + (1 - y_k) log(1 - p_k)]$, the log-loss function used to determine the gradient at each synapse. The two scenarios that we explored:

- Derive the gradient at each synapse with the summation included and p_k is replaced by $\widetilde{n}_j$:

$$\frac{\partial E}{\partial \widetilde{n}_j} = -\frac{1}{ln(10)} \sum_k \left[y_k \frac{1}{\widetilde{n}_j} + (1 - y_k) \frac{1}{\widetilde{n}_j - 1} \right].$$

  This would use the output of the specific neuron being analyzed, $\widetilde{n}_j$, and the binary class label of each output neuron, y_k, that indicates whether the neuron should have fired.

- Remove Σ and derive the gradient at each synapse such that it only focuses on one neuron at a time, giving

$$\frac{\partial E}{\partial \widetilde{n}_j} = -\frac{1}{ln(10)} \left[y_j \frac{1}{\widetilde{n}_j} + (1 - y_j) \frac{1}{(\widetilde{n}_j - 1)} \right].$$

We concluded that it made more sense to remove the Σ and use the second partial derivative shown in the list. While the summation is important when determining the overall error of the SNN, it is not necessary when calculating the error of one specific neuron. More details are given in Appendix A. Figure 5.2a shows the accuracies for this configuration, which at approximately 50% testing average leaves room for improvement.

5.4 No Hidden Layer, Multiple Output Neurons per Class

The jaggedness seen in Figure 5.2a does not represent a learning curve typically expected for a neural network. We decided that there are two possible explanations, the missing hidden layer or having only one output neuron per digit class. We chose to test multiple output neurons first since IBM says they used multiple output neurons per digit class to improve the prediction performance of the network [13]. The previous configuration only had one output neuron per digit class where the fist neuron in the order of zero to nine to fire a 1 is used as the prediction. To quickly determine the affect of multiple output neurons we added a second neuron to each digit class. The prediction for each class becomes the average of all neurons assigned to the class and the highest average is used as the network prediction. Our results are shown in Figure 5.2b. The overall accuracy of the SNN improved but there is still jaggedness in the learning that needs to be fixed.

5.5 Hidden Layer, Multiple Output Neurons per Class

In order to create the complete SNN we added the hidden layer to the previous configuration, what became known as the full SNN. We started with 10 hidden neurons and tested up to 512 to find the highest testing accuracy we could achieve with two output neurons per class, 100 epochs, and 1,000 training samples. Table 5.1a shows the tests and corresponding results. Since 512 hidden neurons did not improve significantly from 256 hidden neurons, we moved on to testing the impact of the number of epochs and output neurons used per digit class with 512 hidden neurons, shown in Tables 5.1b and 5.1c. At the end of these tests, we increased the number of hidden neurons to 1,024 to ensure maximum accuracy. We had a SNN with 1,024 hidden neurons, that learned over 1,000 epochs, used 1,000 training samples, and used eight output neurons per digit class that had approximately 70% testing accuracy at the

Table 5.1: Summary of Methods Used to Maximize Testing Accuracy

Number of Hidden Neurons	Testing Accuracy
10	10.36%
100	46.19%
256	53.91%
512	~54%

(a) Testing Various Amounts of Hidden Neurons with Two Output Neurons per Class, 100 epochs, and 1,000 Training Samples

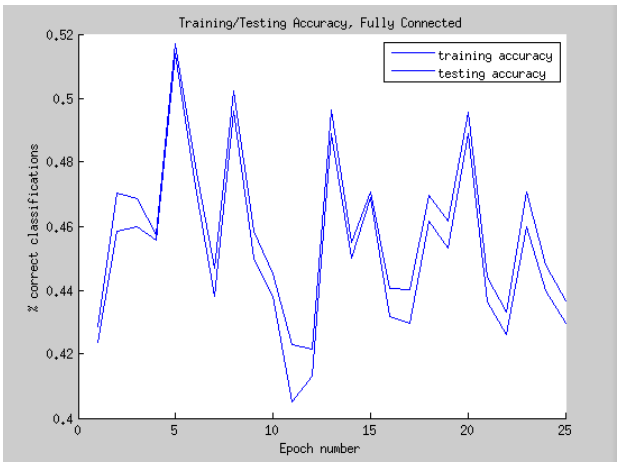
Number of Epochs	Testing Accuracy
250	~60%
500	60%
1,000	64% - 65%

(b) Testing Various Numbers of Epochs with Two Output Neurons per Class, 512 Hidden Neurons, and 1,000 Training Samples

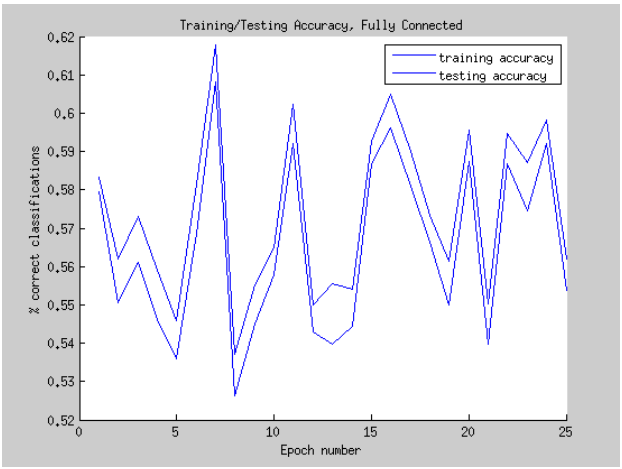
Number of Output Neurons per Class	Testing Accuracy
4	69.25%
8	70.05%
16	~70%
25	~70%

(c) Testing Various Numbers of Output Neurons per Class with 512 Hidden Neurons, 1,000 Epochs, and 1,000 Training Samples

end of this phase. Figure 5.3 shows that the SNN now has the expected learning curve.



(a) SNN with 10 Output Neurons, One Per Digit Class



(b) SNN with Two Output Neurons Per Digit Class

Figure 5.2: Effect of Multiple Output Neurons per Digit Class, No Hidden Layer

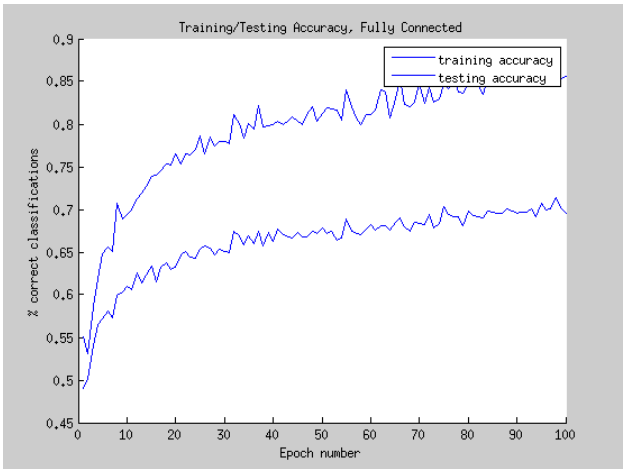


Figure 5.3: Full SNN with Eight Output Neurons Per Digit Class

Table 5.2: Testing Parameters Used During the Design Process

Configuration /Optimization	**No Hidden Layer, One Output Neuron**	**Hidden Layer, One Output Neuron**	**No Hidden Layer, 10 Output Neurons**	**No Hidden Layer, Multiple Output Neurons per Class**	**Hidden Layer, Multiple Output Nerons per Class**
Number of Inputs	784	784	784	784	784
Number of Hidden Neurons	N/A	10	N/A	N/A	1,024
Number of Epochs	250	50	100	250	1,000
How Often Evaluated (Epochs)	10	1	10	10	10
Number of Training Samples Used	100	100/1,000	100	100/60,000	1,000
Number of Output Neurons per Class	N/A	N/A	1	2	8
Number of Testing Samples	10,000	10,000	10,000	10,000	10,000

Chapter 6

Methodology

This thesis performs a direct comparison of hardware efficiency between rate-based ANNs and SNNs for a benchmark data set, MNIST [23]. There are recent efforts to develop algorithms for training SNNs to perform machine learning tasks. When used for machine learning, SNNs offer the potential advantages of not requiring multipliers and reductions in storage for synapse weights and intermediate results. Despite this, there are currently no direct comparisons in hardware efficiency between rate-based ANNs and SNNs. For the comparison, two neural networks were trained and evaluated, one rate-based ANN and one SNN. The network parameters were adjusted such that both achieved comparable testing accuracy. Afterward, we estimated their hardware cost according to their corresponding synapse storage requirements.

6.1 Rate-Based ANN

In order to compare hardware efficiency, we designed the smallest fully connected feed forward rate-based ANN with one hidden layer that achieves a 90% testing accuracy. We are targeting embedded deployment platforms where space is a significant limiting factor for both on-chip and off-chip memory storage. For this thesis, the amount of space, in bits, needed to store synapse weights will be used to approximate the size of a network. This approximation does not fully represent hardware resource usage, especially for floating point data precision since they require extensive hardware to function. The total number of bits for a fully connected network with one hidden layer is determined with the number of input nodes, the number of neurons in the

hidden layer, the number of output neurons, and the number of bits used to store the weights: $TotalBits_{rate} = ((input * hidden) + (hidden * output)) * bitWidth$. The MNIST images are 28x28 pixels, giving us the number of input nodes the network requires. We will only use one output neuron per class label (digit), giving us a total of 10 output neurons. This leaves the number of hidden neurons and the bitWidth unknown. We experimentally found the combination of hidden neurons and bitWidth that achieved 90% testing accuracy while using as few total bits as possible, determined by $TotalBits_{rate} = ((784 * hidden) + (hidden * 10)) * bitWidth$.

6.1.1 Testing Accuracy Criteria

To determine if a rate-based ANN successfully achieved 90% testing accuracy we looked at the testing accuracy reported for the 40^{th} and 50^{th} epochs, out of 50 total. If the average of the two values was greater than or equal to 90% for at least three of five tests, the network was determined to have satisfied the testing accuracy goal.

6.1.2 Floating Point Precision Implementation

Testing double and single floating point precisions were the least complicated tests we performed. Double precision is Matlab's default precision so we did not have to make any considerations during training calculations. Single precision only needed to be type-casted at the initialization stage and during training.

6.1.3 Fixed Point Precision Implementation

Our fixed point precision implementation required significant modification from floating point. We initially attempted training in fixed point but encountered difficulties. The results reported in Table 7.1 used double floating point during training and converted to fixed point during network evaluation. We knew there would be a reserved sign bit in our fixed point precision, some number of bits reserved for the whole

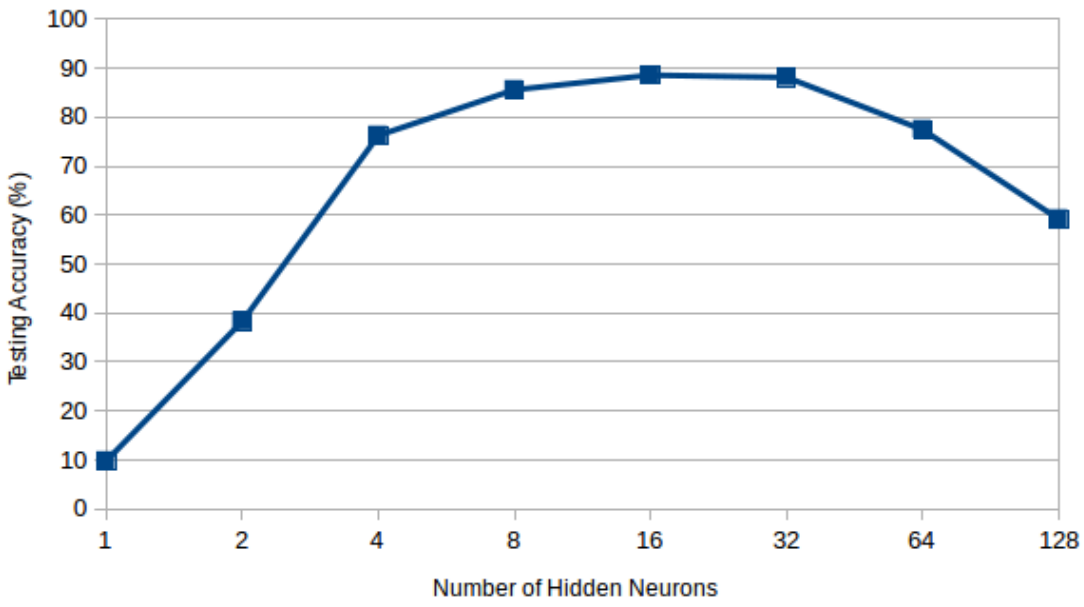


Figure 6.1: Testing Accuracy Trend for 6-bit Fixed Point

number portion, and the remaining bits of the chosen precision would be used as the fractional portion.

To determine the necessary number of bits needed for the whole number portion, we looked at the minimum and maximum values synapse weights can reach during training. We looked at synapse weight values because this thesis is determining the storage requirements for these numbers and wanted to preserve the learned values as much as possible. The smallest learned synapse weight we observed was -30.2 and the largest was 19.7. Since 30 is the largest whole number we saw, we reserved $ceiling(log_2(30))$, or five, bits for the whole number portion of our fixed point precision.

We then tested 6-bit fixed point as the smallest precision we can use, one bit for the sign and five bits for the whole number with no fractional bits. We wanted to know if you could leave out the fractional portion of the synapse weights without penalty. Unfortunately, there is a penalty. When 6-bit fixed point was used with 64 or more hidden neurons, the testing accuracy began to fall compared to what was found when

using smaller amounts of hidden neurons. Figure 6.1 shows this behavior. Because of these results, we decided to enforce at least one bit for the fractional portion, making 7-bit fixed point the smallest precision we tested and reported.

6.2 SNN

After the smallest possible rate-based ANN was found, we designed a comparable SNN discussed in detail in Chapters 4 and 5. It has the same fully connected feed forward topology as the ANN with 784 input nodes from the 28x28 pixel images. Due to the spiking nature of the network output, we had to use eight neurons per digit class, resulting in 80 total output neurons. Since the synapse weights are only stored using one bit, bitWidth is not taken into account for the total bits used for the SNN. Instead, multiple deployment SNNs, called ensembles, can be used to increase testing accuracy so the number of ensembles needed is measured instead. The number of hidden neurons required to achieve 90% or comparable testing accuracy is also unknown. The total number of bits required by the SNN will be determined by $TotalBits_{spike} = ((784 * hidden) + (hidden * 80)) * ensembles$.

6.2.1 Testing Accuracy Criteria

To determine if a SNN configuration successfully achieved the target testing accuracy, we looked at the accuracy reported for the 900^{th} and $1,000^{th}$ epochs, out of 1,000 total. If the average of these two values was greater than or equal to the target, the network was decided to have satisfied the testing accuracy goal. We did not require that the accuracy be reached at least three of five tests.

6.2.2 Optimizations

After Chapter 5 we had a full SNN that achieved approximately 70% testing accuracy. We investigated the optimizations discussed below and implemented those that helped

Algorithm 4 Learning Rate Decay

```
%Note: alpha is initialized to 0.1 and epoch starts at 1
if epoch % 250 == 0 then
    alpha = alpha * 0.1;
end if
```

increase the SNN testing accuracy closer to our 90% goal.

Learning Rate Decay

This is the first optimization we explored since its implementation had the least impact on the code, only requiring a simple `if` statement shown in Algorithm 4. We used IBM's method for learning rate decay which followed "a fixed schedule across training iterations starting at 0.1 and multiplying by 0.1 every 250 epochs [13]".

Momentum

The next optimization we looked at was momentum. IBM reported they used a momentum of 0.9 [13] so we did also without experimenting with other values. Momentum is useful because of its smoothing affect on the learning process, due to the fact that momentum helps a network extricate itself from local minima that can occur during training [28].

Mini Batching

After momentum, we looked into implementing mini batches. The SNN had been designed using stochastic gradient descent up to this point and we investigated mini batches since IBM reported using it for their TrueNorth SNN application [13]. Successfully implementing mini batches would reduce the number of times the network was updated and overall calculation time. However, adding mini batches to the SNN was error prone and the effort to debug quickly proved to be intensive. We decided to pursue 90% testing accuracy by other means.

Ensembles

The last optimization we added to the SNN was ensembles. This is implemented in the deployment network where a number of networks, the desired number of ensembles, is created by independently sampling the training network. Each ensemble evaluates input data and the output from all of them are averaged together before determining the prediction, similar to the example in Table 4.1.

Chapter 7

Results

We designed a rate-based ANN and SNN with matching topologies. Both have 784 input nodes (28x28 pixel images) fully connected to a hidden layer with some number of neurons. The hidden layer is then fully connected to the output layer. Our rate-based ANN has 10 output neurons, one for each digit, and our SNN has 80 total output neurons, each digit has eight neurons assigned to it. Figure 2.1a shows the general topology of these networks.

7.1 Rate-Based ANN Results

To determine the smallest possible rate-based ANN, we decided to pick the bit widths we wanted to test then determine the number of hidden neurons needed to reach 90% testing accuracy on the MNIST dataset. We tested double and single floating point and a variety of fixed point precisions under 32 bits. These results are shown in Table 7.1.

Table 7.1: Rate-based ANN Results

bitWidth	**Minimum Hidden Neurons to Achieve 90% Testing Accuracy**	$TotalBits_{rate}$
Double Floating Point	8	406,528
Single Floating Point	8	203,264
24-bit Fixed Point	8	152,448
16-bit Fixed Point	8	101,632
9-bit Fixed Point	8	57,168
8-bit Fixed Point	8	50,816
7-bit Fixed Point	11	61,138

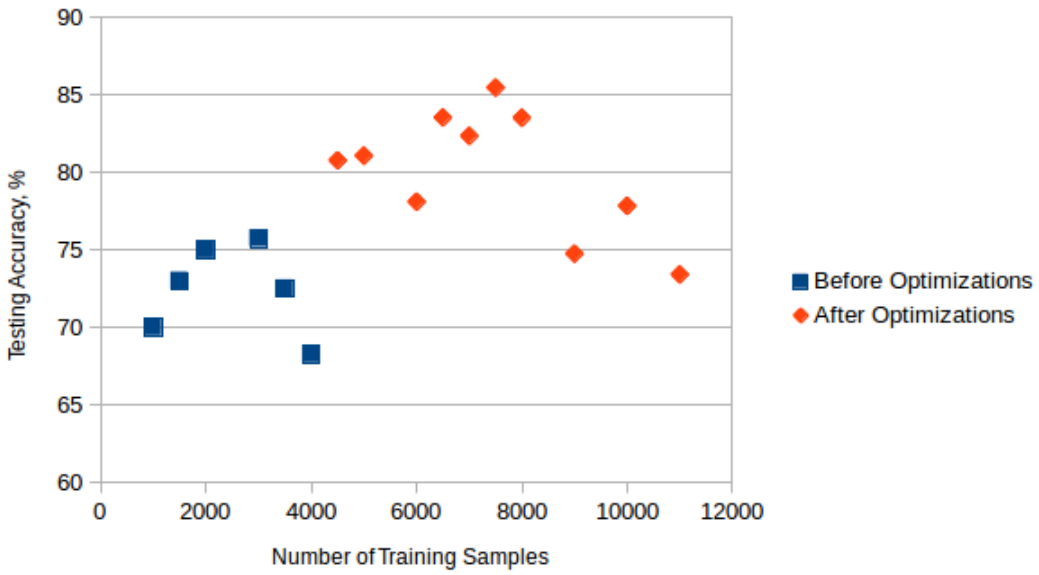


Figure 7.1: Increasing Training Sample Size for SNN

7.2 Increasing Training Sample Size for SNN

Considering that the MNIST dataset comes with 60,000 training samples, using less than 10,000 is not ideal. However, throughout the process of developing our SNN training algorithm, testing with 10,000 and 60,000 training samples failed. Learning would progress normally then plummet. The accuracies would greatly oscillate as well. This behavior is possibly a result of over learning so we decided to test various training sample sizes until we reached the maximum amount we could use before seeing a decline in testing accuracies. Figure 7.1 shows the results of these tests. The overall testing accuracy of a test is determined by averaging the accuracies reported at the 900^{th} and $1,000^{th}$ epochs, out of 1,000 total, for 3,000 training samples and beyond. Tests before 3,000 had their testing accurcay estimated from their graphs. We concluded 7,500 training samples is the optimal amount to use for training our SNN with 85.48% testing accuracy but this is short of our goal of 90% testing accuracy.

7.3 SNN Optimizations

Optimizations were added to the SNN as the training sample size was increased once we noticed that the learning curve was still on an upward trajectory after 1,000 epochs starting at approximately 3,500 training samples. We anticipated that the optimizations would cause the learning curve to plateau before the $1,000^{th}$ epoch.

7.3.1 Momentum

IBM did not elaborate on where they used momentum, on the hidden layer, output layer, or both. After some experimentation with 1,024 hidden neurons, shown in Figure 7.2a, we decided to only use momentum on the hidden layer.

7.3.2 Ensembles

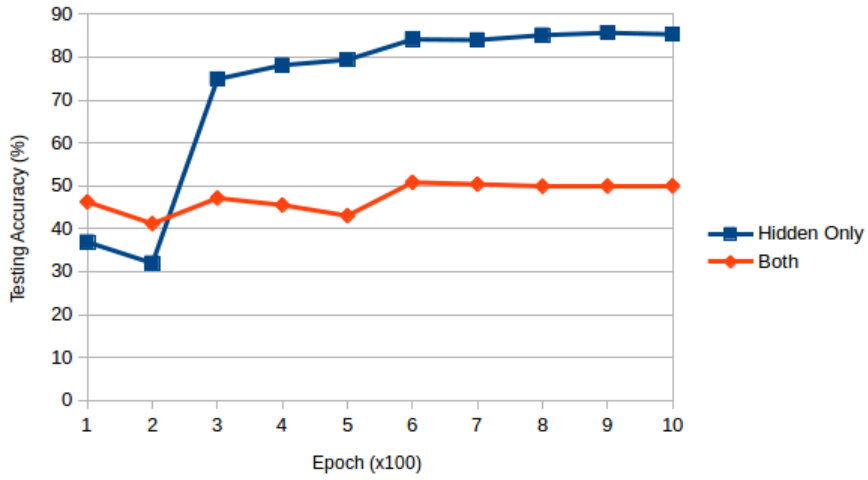
We explored the affect of ensembles on a SNN with 1,024 hidden neurons and found a small improvement in the testing accuracy when four ensembles were used during deployment instead of only one, Figure 7.2b.

7.3.3 Learning Rate Decay

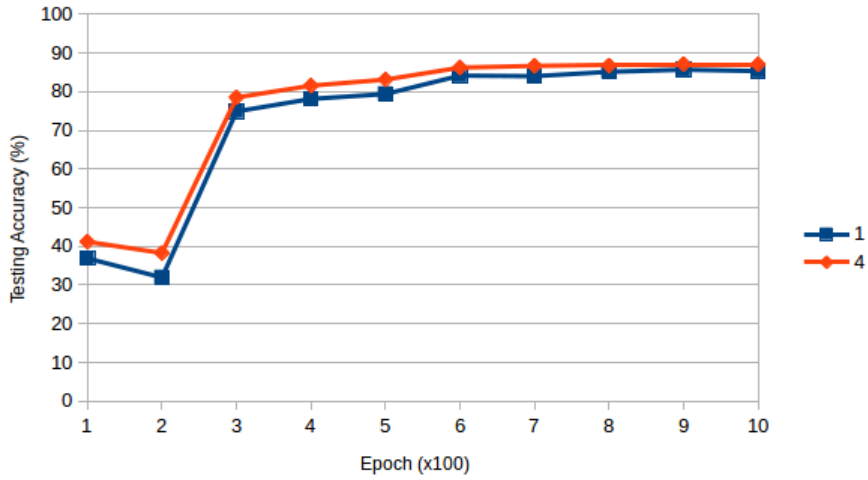
Including learning rate decay in the SNN training algorithm resulted in the jump in testing accuracy between the 200^{th} and 300^{th} epochs seen in Figures 7.2a and 7.2b. You can also discern a slight jump in testing accuracy between the 500^{th} and 600^{th} epochs in these figures.

7.4 SNN Results

We were unable to reach 90% testing accuracy with our fully-connected feedforward SNN. Table 7.2 shows the most successful configurations we were able to obtain for one, two, three, and four ensembles. We attempted to minimize the number of hidden



(a) Momentum



(b) Ensembles

Figure 7.2: SNN Optimizations

neurons needed to reach 87% testing accuracy for four ensembles but smaller amounts of hidden neurons resulted in decreased testing accuracy.

Table 7.2: SNN Results

Ensembles	**# Hidden Neurons**	**Testing Accuracy**	$TotalBits_{spike}$
1	256	85.18%	221,184
2	512	86.42%	884,736
3	256	87.06%	663,552
4	1,024	87.05%	3,538,944

Chapter 8

Conclusion and Discussion

This thesis examined the hardware efficiency of a SNN trained with IBM's probabilistic backpropagation algorithm [13] with respect to a rate-based ANN. Since SNNs do not require multipliers or transcendental activation functions, the purpose of this thesis is to determine if these benefits outweigh the cost of increased neurons. For the analysis, a SNN and rate-based ANN were trained and evaluated on the MNIST dataset [23]. Network parameters, such as the number of hidden neurons, data precision, and ensembles, were adjusted until both networks achieved comparable testing accuracy with 90% as the goal. Hardware efficiency was estimated by the number of bits required to store synapse weights.

Based solely on the storage requirements for synapse values, the rate-based ANN is more memory efficient than the SNN. The smallest rate-based ANN we found that achieved 90% testing accuracy on the MNIST dataset used 8-bit fixed point to represent synapse weights and only needed eight hidden neurons. This configuration required 50,816 bits for storage. Comparatively, the highest testing accuracy we were able to obtain for the SNN was 87% which required three ensembles, 256 hidden neurons, and 663,552 bits for synapse value storage.

For the SNN, we noticed that some of the network parameters we tweaked during experimentation had varying affects on the testing accuracy. In Chapter 5, we learned that increasing the number of hidden neurons, epochs, and output neurons per class improved testing accuracy up to a point, after which, there was no increase. In Chapter 7, we explored the affects of a few neural network optimizations. Momentum

had the greatest impact by producing a smoother learning curve and only having momentum on the hidden layer instead of both hidden and output layers gave a higher testing accuracy. Learning rate decay had significant impact on the testing accuracy by accelerating the learning. Ensembles was the least predictable parameter we experimented with. It was not guaranteed that increasing the number of ensembles would increase the testing accuracy. Increasing the number of hidden neurons was not guaranteed to increase the testing accuracy either.

Storage requirements are not the only aspect of hardware that can be investigated to determine which neural network is most hardware efficient. This thesis can be extended in a future project to estimate the area of an FPGA implementation for both a rate-based ANN and a SNN. An in depth look at the total hardware necessary for both networks would give better understanding of the trade-offs between the two. One important piece of hardware to compare is the multiplier. Binary networks eliminate multiply-accumulate operations inherent to rate-based ANNs and therefore eliminate hardware mulitpliers. This difference could provide a way for SNNs to be more efficient than rate-based ANNs.

Another point of comparison for rate-based ANNs and SNNs is their activation functions. SNNs only require an addition and a comparison while rate-based ANNs need an estimation of a transcendental function. The implementation of the transcendental function could cause a drop in hardware efficiency for the rate-based ANN.

Bibliography

[1] A. Belatreche, L. P. Maguire, M. McGinnity, and Q. X. Wu. A method for supervised training of spiking neural networks. In *Proc. IEEE Conf. Cybernetics Intelligence Challenges and Advances, CICA*, 2003.

[2] S. M. Bohte, J. N. Kok, and H. La Poutré. Spike-prop: error-backpropagation in multi-layer networks of spiking neurons. *Neurocomputing*, 48(1):17–37, 2002.

[3] Jason Brownlee. How to scale machine learning data from scratch with python, Mar 2018.

[4] Caffe. http://caffe.berkeleyvision.org/.

[5] Andrew Carnell and Daniel Richardson. Linear algebra for time series of spikes. In *Proc. 13th European Symposium on Artificial Neural Networks*, 2005.

[6] Andrew S. Cassidy, Paul Merolla, John V. Arthur, Steven K. Esser, Bryan L. Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M. Wong, Vitaly Feldman, Arnon Amir, Daniel Ben Dayan Rubin, Filipp Akopyan, Emmett McQuinn, William P. Risk, and Dharmendra S. Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *IJCNN*, pages 1–10. IEEE, 2013.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Oliver Temam. A high-throughput neural network accelerator. *IEEE Micro*, 35(3).

[8] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhen-zhong Lan. Training binary multilayer neural networks for image classification using expectation backpropagation. *CoRR*, abs/1503.03562, 2015.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3123–3131. Curran Associates, Inc., 2015.

[10] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training neural networks with weights and activatios constrained to +1 or -1. March 2016.

[11] C. E. Cox and E. Blanz. Ganglion - a fast field-programmable gate array implementation of a connectionist classifier. *IEEE Journal of Solid-State Circuits*, 28(3):288–299, 1992.

[12] Nvidia's cudnn. http://developer.nvidia.com/cudnn.

[13] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc., 2015.

[14] A. D. Brown et al. Spinnaker - programming model. *IEEE Trans. on Computers*, 64(6):1769 – 1782, 2015.

[15] Jean-Pascal Pfister et al. Optimal spike-timing-dependent plasticity for precise action potential firing in supervised learning. *Neural computation*, 18(6):1318–1348, 2006.

[16] Paul A. Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668 – 673, August 2014.

[17] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the spinnaker system architecture. *IEEE Trans. on Computers*, 63(12), December 2013.

[18] Ankur Gupta and Lyle N. Long. Hebbian learning with winner take all for spiking neural networks. In *Proc. International Joint Conference on Neural Networks*, 2009.

[19] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 6 edition, 2019.

[20] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, May 2007.

[21] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. on Neural Networks*, 14(6), 2003.

[22] Andrezej Kasinski and Filip Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *Int. J. Appl. Math. Comput. Sci.*, 16(1):101–113, 2006.

[23] Y. LeCun, L. Bottou, and Y. Bengio an P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[24] Robert Legenstein, Christian Naeger, and Wolfgang Maass. What can a neuron learn with spike-timing-dependent plasticity? *Neural computation*, 17(11):2337–2382, 2005.

[25] Sicheng Li, Chunpeng Wu, Hai (Helen) Li, Boxun Li, Yu Wang, and Qinru Qui. Fpga acceleration of recurrent neural network based language model. In *Proc. IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[26] De Ma, Juncheng Shen, Zonghua Gu, Ming Zhang, Xiaolei Zhu, Xiaoqiang Xu, Qi Xu, Yangjing Shen, and Gang Pan. Darwin: A neuromorphic hardware co-processor based on spiking neural networks. *Journal of Systems Architecture*, 2017.

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

[28] Momentum and learning rate adaptation. http://www.willamette.edu/~gorr/classes/cs449/momrate.html.

[29] Filip Ponulak. *Supervised learning in spiking neural networks with ReSuMe method*. PhD thesis, Poznan University of Technology, 2006.

[30] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Publishing, 3rd edition.

[31] Tensorflow. http://www.tensorflow.org.

[32] Theano. http://www.deeplearning.net/software/theano/.

[33] David B. Thomas and Wayne Luk. Fpga accelerated simulation of biologically plausible spiking neural networks. In *17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009.

[34] Torch. torch.ch.

[35] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74, New York, NY, USA, 2017. ACM.

[36] Nvidia volta. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[37] Lei Wan, Yuling Luo, Shuxiang Song, J. Harkin, and Junxiu Liu. Efficient neuron architecture for fpga-based spiking neural networks. In *2016 27th Irish Signals and Systems Conference (ISSC)*, pages 1–6, June 2016.

[38] Jieyu Zhao, John Shawe-Taylor, and Max van Daalen. Learning in stochastic bit stream neural networks. *Neural Networks*, 9(6):991 – 998, 1996.

[39] J. Zhu and P. Sutton. Fpga implementation of neural networks-a survey of a decade of progress. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, pages 1062–1066, 2003.

Appendix A

Derivation of Neuron Gradient

The probability of target neuron j firing is represented as $P(n_j = 1) \equiv \widetilde{n}_j$. $\widetilde{n}_j$ is the complimentary Cumulative Distribution Function (CDF) of a Gaussian:

$$\widetilde{n}_j = 1 - \frac{1}{2}\left[1 + erf\left(\frac{\Theta - \mu_j}{\sqrt{2{\sigma_j}^2}}\right)\right]$$

where erf is the error function and $\Theta = 0$. The log-loss function is used since training was observed by IBM to converge fastest with this approach [13],

$$E = -\sum_k [y_k log(p_k) + (1 - y_k) log(1 - p_k)]$$

where, for class k, y_k is the binary label indicating the presence of a class and p_k is the probability that the average spike count for k is greater than 0.5 ($\widetilde{n}_j$ with $\Theta = 0.5$). During training, the derivative of the loss function with respect to the synapse connection probabilities $\widetilde{c}_{ij}$, where i is the input neuron index, is used to find the gradient at each synapse. It is computed using the chain rule

$$\frac{\partial E}{\partial \widetilde{c}_{ij}} = \frac{\partial E}{\partial \widetilde{n}_j} \frac{\partial \widetilde{n}_j}{\partial \widetilde{c}_{ij}}.$$

In [13], IBM derived the right most fraction and concluded that

$$\frac{\partial \widetilde{n}_j}{\partial \widetilde{c}_{ij}} \approx \frac{\partial \widetilde{n}_j}{\partial \mu_j} \frac{\partial \mu_j}{\partial \widetilde{c}_{ij}}$$

where

$$\frac{\partial \widetilde{n}_j}{\partial \mu_j} = \frac{1}{\sigma_j \sqrt{2\pi}} e^{-\frac{(\Theta - \mu_j)^2}{2{\sigma_j}^2}},$$

and

$$\frac{\partial \mu_j}{\partial \widetilde{c}_{ij}} = \widetilde{x}_i s_{ij}.$$

IBM did not derive $\frac{\partial E}{\partial \widetilde{n}_j}$ in [13] so this derivation is included below where p_k is replaced with $\widetilde{n}_j$ of neuron j. Since only one neuron is being analyzed at a time, the summation is dropped and y_k becomes y_j, the expected output of neuron j in the output layer.

$$\frac{\partial E}{\partial \widetilde{n}_j} = -\frac{\partial}{\partial \widetilde{n}_j} \sum_k [y_k log(p_k) + (1 - y_k) log(1 - p_k)]$$

$$\frac{\partial E}{\partial \widetilde{n}_j} = -\frac{\partial}{\partial \widetilde{n}_j} y_j log(\widetilde{n}_j) + \frac{\partial}{\partial \widetilde{n}_j}(1 - y_j) log(1 - \widetilde{n}_j)$$

$$\frac{\partial E}{\partial \widetilde{n}_j} = -y_j \frac{1}{\widetilde{n}_j ln(10)} + (1 - y_j) \frac{1}{(\widetilde{n}_j - 1) ln(10)}$$

$$\frac{\partial E}{\partial \widetilde{n}_j} = -y_j \frac{1}{2.303} \frac{1}{\widetilde{n}_j} + (1 - y_j) \frac{1}{2.303} \frac{1}{(\widetilde{n}_j - 1)}$$

$$\frac{\partial E}{\partial \widetilde{n}_j} = -\frac{1}{2.303} \left[y_j \frac{1}{\widetilde{n}_j} + (1 - y_j) \frac{1}{(\widetilde{n}_j - 1)} \right]$$

Putting everything together, the gradient at each synapse is

$$\frac{\partial E}{\partial \widetilde{c}_{ij}} = -\frac{1}{2.303} \left[y_j \frac{1}{\widetilde{n}_j} + (1 - y_j) \frac{1}{(\widetilde{n}_j - 1)} \right] * \frac{1}{\sigma_j \sqrt{2\pi}} e^{-\frac{(\Theta - \mu_j)^2}{2{\sigma_j}^2}} * \widetilde{x}_i s_{ij}.$$

In [13], IBM makes a comment that "a similar treatment can be used to show that [the] corresponding gradient with respect to the bias term equals 1" after deriving $\frac{\partial \mu_j}{\partial \widetilde{c}_{ij}}$. I took this to mean that

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial \widetilde{n}_j} \frac{\partial \widetilde{n}_j}{\partial \mu_j} \frac{\partial \mu_j}{\partial b_j} \; where \; \frac{\partial \mu_j}{\partial b_j} = 1.$$

Appendix B

SNN Design Decisions

B.1 ANN Breakdown

In order to understand how the equations I have for the SNN relate to the traditional ANN, I broke down our ANN equations into smaller pieces and matched their function to those of the SNN equations. ANN equations are shown with the round bullets and to the left of the $\equiv$ symbol in the dash bullets. The equivalent SNN equations are shown to the right of the $\equiv$ symbol in the dash bullets.

Output Layer

- $out\_\widetilde{c} = \frac{\partial}{\partial x} S(x) * (out\_output - out\_expected)$
  - $out\_output - out\_expected \equiv \frac{\partial E}{\partial \widetilde{n}_j}$
  - $\frac{\partial}{\partial x} S(x) \equiv \frac{\partial \widetilde{n}_j}{\partial \mu_j}$
- $out\_weights = out\_weights - (a * out\_\widetilde{c} * out\_input)$
  - $out\_input \equiv \frac{\partial \mu_j}{\partial \widetilde{c_{ij}}}$
  - $-a * out\_\widetilde{c} * out\_input \equiv -a * \frac{\partial E}{\partial \widetilde{c_{ij}}}$

Hidden Layer

- $hid\_\widetilde{c} = \frac{\partial}{\partial x} S(x) * sum \ where \ sum = \Sigma(out\_\widetilde{c} * out\_weights)$
  - Sum replaces output comparison to expected $\equiv$ Sum to replace $\frac{\partial E}{\partial \widetilde{n}_j}$
  - $\frac{\partial}{\partial x} S(x) \equiv \frac{\partial \widetilde{n}_j}{\partial \mu_j}$

- $hid\_weights = hid\_weights - (a * hid\_\widetilde{c} * hid\_input)$
  - $hid\_input \equiv \frac{\partial \mu_j}{\partial \widetilde{c}_{ij}}$
  - $-a * hid\_\widetilde{c} * hid\_input \equiv -a * \frac{\partial E}{\partial \widetilde{c}_{ij}}$

B.2 Value Checks

It is possible for some values to be 0 or 1 during training and this can cause problems in calculations within the SNN by introducing infinity or "not a number". I have provided checks for these specific values and altered them with a small number; 0.0001 was chosen arbitrarily based on the default precision Matlab displays in the Command Window. Algorithm 5 shows the checks I have for the output neurons and Algorithm 6 shows the checks for the variance of the hidden neurons, which are the only two values I observed to become 0 or 1 during training.

Algorithm 5 Check for Special Values of out_n

```
1: if out_n == 0 then
2:   | out_n = out_n + small_num;
3: else if out_n == 1 then
4:   | out_n = out_n - small_num;
5: end if
6: e_n=(-1/log(10))*((y*(1/out_n))+((1-y)*(1/(out_n-1))));
```

Algorithm 6 Check for Special Values of hid_var

```
1: if hid_var == 0 then
2:   | hid_var = hid_var + small_num;
3: end if
4: hid_n_mu=(1/(σ_l√(2π))) * exp(-(Θ-μ_l)^2/(2σ_l^2));
```