Towards Efficient On-Chip Learning With Equilibrium Propagation

Zhengyun Ji

Electrical and Computer Engineering
McGill University, Montreal

December 2019

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Engineering

Anyone who considers arithmetical methods
of producing random digits is, of course,
in a state of sin.

— JOHN VON NEUMANN

Abstract

With the growing research and application of deep learning, there are increasing demands on the ability to re-train or improve models with new data in the field. The popular back-propagation algorithm is very effective when training large models offline; however, it requires considerable computational resources. As an alternative to the traditional back propagation algorithm, Equilibrium Propagation is an energy-based learning algorithm for neural networks. Using almost the same computation for the forward and backward phase, the algorithm is an interesting candidate for implementing on-chip learning. As a first step towards building the hardware, quantizations are applied on the algorithm to study the feasibility of a digital implementation. We then introduce a hardware-oriented network pruning method to reduce the number of computations and the memory usage by a factor of 2.7. We also propose a digital hardware architecture for the pruned network for the MNIST hand written digit recognition task using 500 hidden nodes with a maximum throughput of 3418 images per second. Finally, we introduce the use of stochastic computing by replacing the state update logic with a Tracking Forecast Memory. This thesis studies the Equilibrium Propagation algorithm from multiple angles and proposes various modifications towards a hardware friendly implementation.

Résumé

Avec l'augmentation de la recherche et de l'application de l'apprentissage approfondi, la capacité de former ou d'améliorer les modèles à l'aide de nouvelles données sur le terrain est de plus en plus sollicitée. L'algorithme populaire de rétropropropagation est très efficace lors de la formation de grands modèles hors ligne, cependant, il nécessite des ressources informatiques considérables. Comme alternative à l'algorithme traditionnel de propagation en retour, la propagation en équilibre est un algorithme d'apprentissage basé sur l'énergie pour les réseaux de neurones. En utilisant presque le même calcul pour la phase avant et la phase arrière, l'algorithme est un candidat intéressant pour la mise en œuvre de l'apprentissage sur puce. Comme première étape vers la construction du matériel, des quantifications sont appliquées sur l'algorithme pour étudier la faisabilité d'une implémentation numérique. Nous introduisons ensuite une méthode d'élagage réseau orientée matériel pour réduire le nombre de calculs et l'utilisation de mémoire par un facteur 2,7. Nous avons également proposé une architecture matérielle numérique pour le réseau taillé pour la tâche de reconnaissance de chiffres manuscrits MNIST utilisant 500 nœuds cachés avec un débit maximum de 3418 images par seconde. Enfin, nous introduisons l'utilisation de la calcul stochastique en remplaçant la logique de mise à jour d'état par une mémoire de prévision de suivi. Cette thèse étudie l'algorithme de propagation d'équilibre sous de multiples angles et propose diverses modifications en vue d'une implémentation adaptée au matériel.

Acknowledgement

I would like to thank my supervisor Prof. Warren Gross for his constant support and motivation for my study and research in this challenging yet rewarding Masters program. His course on VLSI for Machine Learning introduced me to key concepts and techniques that laid the foundation to my research and the completion of this thesis.

This thesis and degree could not have been possible without the support and company of my colleagues and friends. I would like to thank especially Dr. Arash Ardakani, for his mentorship, technical guidance, and inspirations. I had the honor to collaborate with him on three conference papers, and he has never failed to amaze me with his ingenuity and insights.

I wish to also thank Dr. Thibaud Tonnellier and Dr. Siting Liu for patiently reading the drafts of my thesis and papers. Their feedback and advice are instrumental to the completion of this thesis.

I would also like to thank Harsh Aurora for introducing me to McGill Pizza, exchanging Linux tricks, as well as countless entertaining and intellectual conversations.

Finally, I would like to thank my family and friends for their moral support that kept me sane for all these years.

Contributions

The work in Chapter 3 was submitted to and under review in a conference paper for which I am the primary author.

List of Figures

List of Acronyms

ASIC Application-Specific Integrated Circuit

CNN Convolutional Neural Network

EP Equilibrium Propagation

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

LFSR Linear-Feedback Shift Register

LIF Leaky-Integrate-and-Fire

MAC Multiply and Accumulate

MLP Multi-Layer Perceptron

PVT Process, Voltage, and Temperature

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

SNG Stochastic Number Generator

SNN Spiking Neural Network

TFM Tracking Forecast Memory

VLSI Very-Large-Scale Integration

Contents

CHAPTER 1

Introduction

1.1 Motivations

Deep learning with neural networks has achieved state-of-the-art performance in various tasks including image classification and language modelling [1]. These networks [2, 3] rely on the now ubiquitous back-propagation algorithm [4], where the gradient of the loss function is propagated backwards using the chain rule to update the weights at each layer.

However, these methods often require computational resources including multiple Graphics Processing Units (GPUs) running for hours or sometimes even weeks. Various works [5–8] have proposed efficient hardware architectures targeting devices such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), most of which, however, are focused on the inference computation. Training of the networks is still mostly done offline on traditional computing devices such as general-purpose GPUs.

In applications that require energy-efficient learning algorithms witincludingthe ability

to adapt to new inputs in the field, an on-line learning chip can be advantageous.

Equilibrium Propagation (EP) [9] was proposed as a biologically plausible energy-based deep learning algorithm. Energy-based models, including Hopfield Networks [10], are models that associate the overall network state with a scalar value, denoted as the energy [11]. The models are then relaxed to a preferred (low energy) state given some input, and predictions are made based on that state. The EP algorithm requires similar computations for both the inference and the error propagation. The hardware computation elements used for the inference computation can then be reused to perform the backward error propagation, leading to an on-line learning architecture with better hardware utilization.

1.2 Objectives

In this thesis, we explore one direction to design such an on-line learning system using the energy-based EP algorithm.*

In order to reduce the hardware complexity, memory usage, and computation latency, we investigate the effect of quantization, as well as pruning of the EP network. We then develop and synthesize a hardware architecture as a proof-of-concept to study its performance. Additionally, we propose the use of stochastic computing to replace the computations in EP to further simplify the computing hardware, and examine its feasibility.

*While proposed and implemented on conventional Von Neumann CPU, the algorithm could be more efficiently implemented as analog circuits [12]. On digital hardware, we use Euler's Method to discretize the state evolution, which usually converges in about 20 iterations for a small network with one hidden layer, and up to 500 iterations for a bigger network [9]. There has been work that attempts to reduce the required number of iterations by predicting and initializing the states to be close to the fixed point [13].

However, despite the advantage of analog circuits, they suffer from poor scalability and portability. A chip designed for a small network cannot be easily adapted to support larger networks. It is also not trivial to transfer designs to new process nodes. Additionally, analog memory, e.g., one used to store the weights, cannot be easily transferred to other devices in order to deploy pre-trained network.

1.3 Thesis organization

In this thesis, we explore the implementation of the EP algorithm using digital hardware. We will introduce briefly the EP algorithm and its computations involved in Section 2.2. Quantization and pruning methods will then be introduced in Sections 3.2 and 3.3. We will show the results and effect of the quantization and pruning method proposed in Section 3.4. We will then introduce an example hardware architecture for the digital EP algorithm, and its performance in Chapter 4. We will propose the application of stochastic computing in Chapter 5 and show some preliminary results. Finally, we will conclude with some discussion and propose future works on this subject in Chapter 6.

CHAPTER 2

Background

2.1 Artificial Neural Networks

Machine learning is a type of algorithm that is able to improve its performance for certain tasks through experience. A common class of machine learning algorithm is supervised learning. As shown in Fig. 2.1, typically, the algorithm uses a model to make predictions based on the input. As opposed to manually programming the mathematical or logical relationships, learning algorithms optimize a set of model parameters through training data (experience). Using the trained parameters, the model is able to predict outputs from new inputs that it has never seen before.

Among many learning algorithms, artificial neural networks are a class of models that mimic the behavior and structure of that of the human nervous system. The model involves a set of nodes, or neurons, connected by weighted edges to form a network. Examples of such has been proposed as early as the birth of modern computers [14, 15].

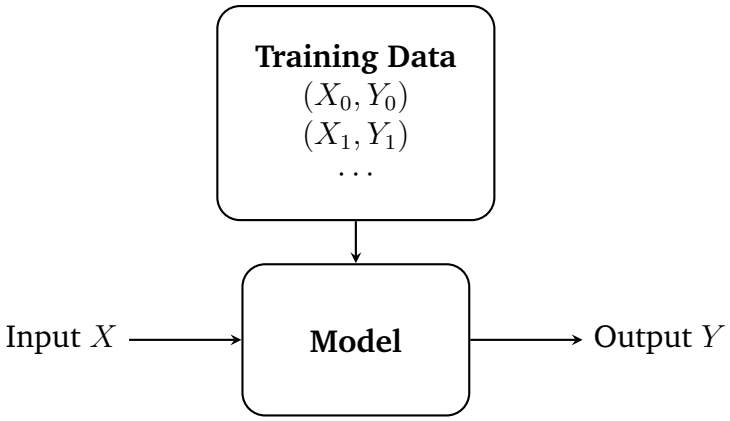

Figure 2.1: Typical Supervised Learning System

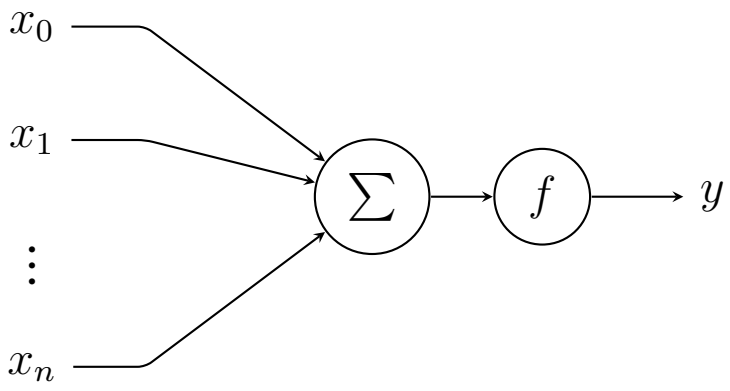

Figure 2.2: The Perceptron Model, where f is a non-linear function.

2.1.1 Multi-Layer Perceptron Networks

The perceptron model shown in Fig. 2.2 is the building block of many artificial neural networks. The perceptron neuron accumulates a weighted sum of the inputs, and applies a non-linear function to the result. Typically, the functionality of one perceptron is limited, e.g., it cannot implement functions such as the exclusive-OR operation. However, many perceptrons can be stacked and chained to form a network of neurons such as the MLP network, as shown in Fig. 2.3.

Training such a neural network often involves a forward and backward phase. In the forward phase, an input vector $\mathbf{x}$ from the training dataset is presented to the model with

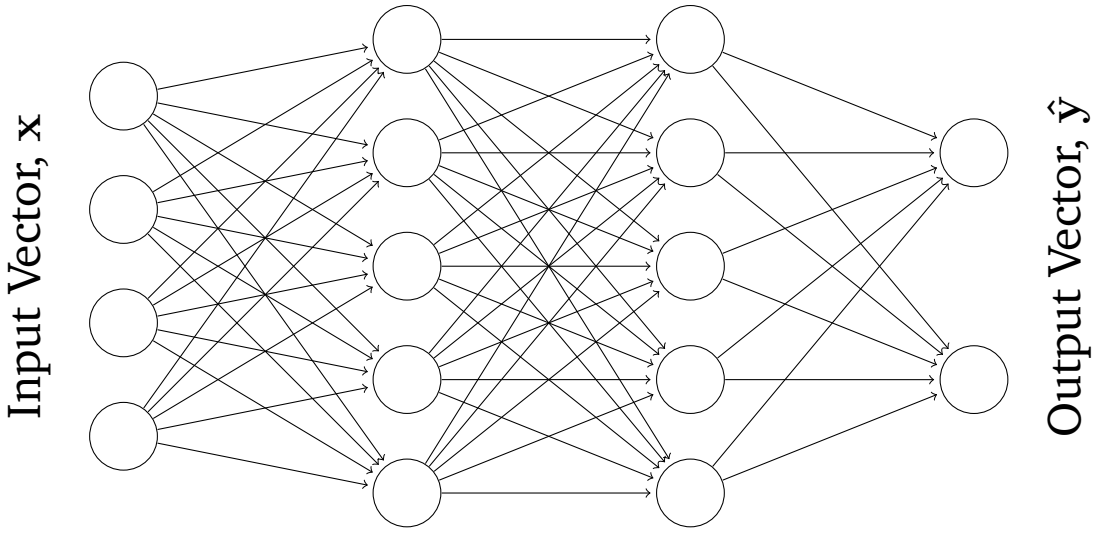

Figure 2.3: A multi-layer perceptron network.

parameters $\mathbf{w}$, which predicts an output vector

$$\hat{\mathbf{y}} = g(\mathbf{x}, \mathbf{w}). \tag{2.1}$$

This output vector is then used to make the predictions. In classification tasks, the output vector is generally one-hot encoded, and the index for the maximum element is the predicted class. By comparing with the true target $\mathbf{y}$ of the output from the dataset, we can establish an error or cost function $C(\hat{\mathbf{y}}, y)$. In the backward phase, the parameters of the model, such as edge weights and biases, are updated to minimize the cost function. For example, in back-propagation [4], we differentiate the cost function with respect to the model parameters $\mathbf{w}$ to obtain the gradient. The gradient are then used in conjunction with optimizers such as Stochastic Gradient Descent (SGD) [16] and Adam [17] to update the parameters.

2.1.2 Spiking Neural Networks

A Spiking Neural Network (SNN) refers to a neural network model that uses spiking neurons

as the computational units [18]. Commonly used neural network models such as the MLP and Convolutional Neural Network (CNN) are, albeit brain-inspired, based on statistical models. SNNs are not only inspired by the central nervous system, but also model it as closely as our knowledge and technology allow. Due to the inherent non-linearity and non-deterministic nature, implementation with matrix multiplication or convolution is not feasible.

A SNN model defines the network as a finite directed graph of neurons, with the edges as synapses. Neuron activity is modeled as message passing on the edges. Artificial neurons are created as models of biological neurons. Ideally, a precise model of the biochemistry process of a neuron should yield an accurate behavioral model.

Indeed, models such as Hodgkin-Huxley [19], describe the generation of action potentials using a set of differential equations. When implementing a neural network with thousands of neurons, the processing power required to implement these artificial neurons is significant. Even though the Hodgkin-Huxley model uses biologically meaningful and measurable parameters, a single neuron would require 1200 floating point operations for a single one-millisecond time step [20].

On the other hand, models have been developed to capture the input and output relationships, treating neurons as black boxes. Lapicque's integrate-and-fire model [21] has been the basis of many other computation or behavior-oriented models. The integrate-and-fire model is based on the accumulation of Excitatory Post-Synaptic Potentials (EPSPs), and Inhibitory Post-Synaptic Potentials (IPSPs), leading to the generation of an action potential, or spike, once the membrane potential crosses a certain threshold. In practice, the synaptic weights from synapses that spiked are integrated, then compared to a threshold value. If the value crosses the threshold, an output spike is generated.

Researchers have proposed adding a leakage of membrane potential in the Integrate-and-Fire model, i.e., LIF [22]. The new model takes into account the tendency of the biological neurons toward their resting membrane potential by including an exponential

decay when the membrane potential is below the threshold. Some implementations, however, use linear decay instead [23].

We will discuss in Section 5.3 the relationship of the LIF neurons with the EP algorithm using stochastic computing.

2.2 Overview of Equilibrium Propagation

In this section, we will provide an introduction to the EP [9] algorithm. In Equilibrium Propagation, we define a Hopfield-like network [10] consisting of nodes connected with symmetrical and bi-directional edges. For each node i, we have an associated scalar state value s_i, and a bias parameter b_i. A weight value W_{ij} is associated with each edge between node i and a neighboring node j.

2.2.1 Energy Function

The training process includes the following major steps: Free Phase, Nudge Phase, and Parameter Update. In both the Free Phase and the Nudge Phase, the network relaxes to a fixed point by minimizing the energy function

$$E := \frac{1}{2}\sum_i s_i^2 - \frac{1}{2}\sum_{i \neq j} W_{ij}\rho(s_i)\rho(s_j) - \sum_i b_i \rho(s_i). \tag{2.2}$$

Here $\rho(\cdot)$ is a non-linear activation function. In [9], $\rho(\cdot)$ was chosen to be the hard sigmoid function:

$$\rho(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } 0 < x < 1 \\ 1, & \text{if } x \geq 1 \end{cases}. \tag{2.3}$$

While the use of other activation functions $\rho(\cdot)$ is possible, we assume the hard sigmoid activation function for its simplicity and effectiveness for the rest of this thesis.

During the Free Phase, Eq. (2.2) gives the overall energy of the network, and the states are gradually updated to minimize the overall energy. During the Nudge Phase, a subset of the nodes designated as output nodes are weakly clamped to the target values by including the cost function:

$$C := \frac{1}{2} \|y - d\|^2, \tag{2.4}$$

where y is the present state associated with the output nodes, and d is the target value.

We can then define the total energy for output nodes to be

$$F := E + \beta C, \tag{2.5}$$

where β is a small clamping factor. As mentioned previously, the inference (Free Phase) and error propagation (Nudge Phase) uses similar computation elements. In fact, the Free Phase computation is equivalent to the Nudge Phase with $\beta = 0$.

2.2.2 State Dynamics

We can minimize the energy function using gradient descent on the state variables,

$$\frac{ds_i}{dt} = -\frac{\partial F}{\partial s_i}, \tag{2.6}$$

where the states are changed over time such that the overall energy function is minimized.

In the case of the hard sigmoid function, we have the state gradient for $s_i \in (0, 1)$ as

$$\frac{ds_i}{dt} = \sum_{j \neq i} W_{ij} s_j + b_i - s_i + \beta(d_i - y_i). \tag{2.7}$$

Using Euler's method with step size ϵ for weight updating we would have

$$s_i[t+1] = s_i[t] + \epsilon \frac{ds_i[t]}{dt}. \tag{2.8}$$

We can then write the update rule for $s_i \in (0, 1)$ as follows:

$$s_i + \epsilon \frac{ds_i}{dt} = s_i + \epsilon \left(\sum_{j \neq i} W_{ij} s_j + b_i - s_i \right) + \epsilon \beta (d_i - y_i). \tag{2.9}$$

2.2.3 Weight Update

In EP [9], the weights are updated locally based on the final states of the Free Phase and the Nudge Phase. For the weights between nodes, the change in weights is

$$\Delta w_{ij} \propto \frac{1}{\beta} \left(\rho \left(s_i^{\beta} \right) \rho \left(s_j^{\beta} \right) - \rho \left(s_i^{0} \right) \rho \left(s_j^{0} \right) \right), \tag{2.10}$$

where s_i^0 are free-phase states and s_i^β are nudge-phase states.

Similarly, the biases are updated according to the difference between the free and nudge phase states:

$$\Delta b_i \propto \frac{1}{\beta} \left(\rho \left(s_i^{\beta} \right) - \rho \left(s_i^{0} \right) \right). \tag{2.11}$$

Now with these state dynamics and weight update schemes, we can train the network by first clamping a set of nodes to the input, and then run the Free Phase until the network converges to a fixed point. We then run the nudge phase until convergence, and update the weights based on the final state of both phases. To perform inferences, we only run the free-phase updates and read out the state values associated with the output nodes.

2.2.4 Comparison with MLP

To evaluate the baseline baseline performance of the EP algorithm, we trained a network for the MNIST [24] hand-written digits recognition task with one layer of 500 nodes. An MLP network with one layer of 500 nodes was also trained with back-propagation. The hidden layer in the MLP network uses a Rectified Linear Unit (ReLU) activation function, and the output layer uses a softmax function as the activation function. Table 2.1 compares

Table 2.1: Training Hyper-Parameters and Test Performance of EP and MLP

Parameter	EP	MLP
Step Size ϵ	0.5	–
Clamping Force β	0.5	–
Free Phase	20	–
Nudge Phase	5	–
Optimizer	–	SGD
Loss Function	Square Error	Cross Entropy
Batch Size	1	64
Learning Rate γ	0.03125	0.1
Epochs	15	15
Test Error	**1.98%**	**1.68%**

the performance of the two algorithms on a network with similar number of parameters.

The above table shows that the EP algorithm can achieve a classification accuracy close to that of the MLP network trained with back-propagation.

CHAPTER 3

Quantization and Pruning of Equilibrium Propagation

In Section 2.2 we introduced the Equilibrium Propagation algorithm as an alternative learning algorithm for neural networks, and showed some benchmarks performed using floating point numbers. Here we are interested in quantization in order to reduce the memory usage and hardware complexity. In addition, numerous studies have shown that full precision is not necessary in multiple neural network architectures and applications [25–30]. We are also interested in pruning the network as an additional form of model compression [25,31].

Thus, in this chapter, we will first introduce how the network is implemented in Section 3.1. The effect of quantization are then studied in Section 3.2. In Section 3.3, we will show the pruning method used, and we will then discuss the result from both the quantization and pruning process in Section 3.4.

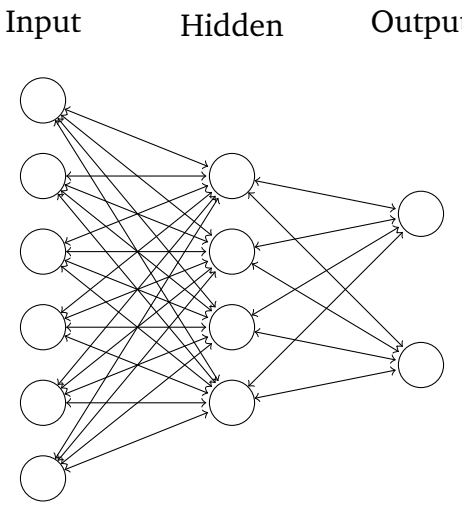

Figure 3.1: Example of a simple multi-layer network topology with six inputs, two outputs, and four hidden nodes.

3.1 Network Topology Representation

Prior work [32] hard codes a layered network topology, where the state vectors corresponding to each layer are updated individually and sequentially. In this work, to easily accommodate alternative network topologies, we operate on the state vector $\mathbf{s}$ for the whole network directly. With certain connection weights set to zero, we can define the network topology.

3.1.1 Example of Topology Mapping

Given a simple multi-layer topology, as shown in Fig. 3.1, we can define the corresponding network topology using the adjacency matrix as shown in Eq. (3.1). Since the connections are bidirectional, the matrix is symmetrical.

Here each row of the adjacency matrix represents the existence of connection from each node to the node represented by that row. In this case, the first six columns and the first six rows represent the six input nodes, the next two the outputs nodes, and the rest the hidden nodes.

$$M = \left[\begin{array}{cccccc|cc|cccc} 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ \hline 0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ \hline 1&1&1&1&1&1&1&1&0&0&0&0 \\ 1&1&1&1&1&1&1&1&0&0&0&0 \\ 1&1&1&1&1&1&1&1&0&0&0&0 \\ 1&1&1&1&1&1&1&1&0&0&0&0 \end{array}\right]. \tag{3.1}$$

The top-left block, for example, represents the connection among all the nodes in the input layer. Since we know that the input nodes are not interconnected, the block is thus all-zero. Similarly, since there are also no direct connections among the output nodes, nor between nodes in the input and the output layer, the corresponding blocks also contain only zeros.

In this particular case of a multi-layer network topology, weights are defined for the connections between layers. A network defined in Fig. 3.1 will have two weight matrices $M^{6\times4}$ and $M^{4\times2}$. However, when considering the adjacency matrix for all the nodes, we can see that the two block matrix of ones are indeed the two aforementioned weight matrices (not counting transposed copies due to the symmetry).

Implementation Using Network Topology Mask

Here we can use the adjacency matrix M, with ones and zeros representing the existence and non-existence of connections, respectively, as a mask for the network topology. We simply multiply a weight matrix of the same dimension element-wise with the mask to satisfy the topology constraint.

Thus, we can rewrite the update rule for the Free Phase in Eq. (2.9) as

$$\mathbf{s} + \epsilon \frac{d\mathbf{s}}{dt} = \mathbf{s} + \epsilon \left((W \odot M)\mathbf{s} + \mathbf{b} - \mathbf{s} \right) \tag{3.2}$$

$$= \mathbf{s} + \epsilon \left((W \odot M - I_n)\mathbf{s} + \mathbf{b} \right). \tag{3.3}$$

where $\mathbf{s}$ is a vector of every node in the network. Here $\odot$ is an element-wise product.

We can further simplify the gradient to

$$\frac{d\mathbf{s}}{dt} = W'\mathbf{s} + \mathbf{b}, \tag{3.4}$$

where $W' = W \odot M - I_n$.

Here the new weight matrix W' is simply the adjacency matrix with the diagonal terms set to -1. During the Nudge Phase, the error term is then added to the output nodes after the state gradient is accumulated.

After the Free Phase and Nudge Phase, two vectors of the node states are obtained. The weight update is then performed by:

$$\Delta W = \frac{1}{\beta} \left(\mathbf{s}_\beta {\mathbf{s}_\beta}^T - \mathbf{s}_0 {\mathbf{s}_0}^T \right), \tag{3.5}$$

where $\mathbf{s}_0$ and $\mathbf{s}_\beta$ are the Free Phase and Nudge Phase state vectors, respectively.

Similarly, the bias update is the difference of the two vectors:

$$\Delta \mathbf{b} = \frac{1}{\beta} \left(\mathbf{s}_\beta - \mathbf{s}_0 \right). \tag{3.6}$$

3.2 Network Quantization

During the Free Phase and the Nudge Phase, the network undergoes an iterative numerical optimization process aimed to reduce the overall energy function in Eq. (2.5). In order to

reduce the hardware footprint, various degrees of numerical quantization are evaluated for their performance.

In a digital hardware setting, all variables — in this case the state values and weights — are converted to fixed-point representation. Binary fixed-point numbers generally consist of a fixed number of bits for both the integer part and the fractional part, separated by the implied radix point. In this algorithm, since all the state values are clamped between 0 and 1, we will keep all but the sign bit for the fractional part.

Even though valid state values are bounded between 0 and 1, we keep the extra sign bit to simplify the computation, as intermediate results could be negative. In practice, floating-point numbers are converted to q-bit fixed-point numbers by scaling and flooring:

$$x_{\mathrm{fixed-point}} = \left\lfloor x \times 2^{q-1} \right\rfloor . \tag{3.7}$$

While the state values range between 0 and 1, the weight values are often much smaller in scale. Thus, as the number of bits used decreases, the weight gradient may be smaller than the quantization step, making it impossible to train the network.

In order to achieve finer granularity for the weights while maintaining the same number of bits used, it is necessary to sacrifice the dynamic range of the possible weight values. E.g., if we wish to quantize a number with 3 bits in the fractional part and one sign bit for the integer part, the smallest granularity is 2^{-3} with a dynamic range of $(-1, 1)$. To achieve a granularity of 2^{-4}, we must add an extra bit to the right. However, we can also remove a bit from the left which results in a smaller dynamic range of $(-0.5, 0.5)$.

In this thesis, we tested the quantization with weight scaling factors of 1, 2, and 4, where the weights have a corresponding dynamic range of $(-1, 1)$, $(-0.5, 0.5)$, and $(-0.25, 0.25)$.

3.3 Network Pruning

With the adjacency matrix representation it is easy to define an arbitrarily connected network. It means we can also optimize the network topology for hardware implementation. Network pruning [25] is a technique often used to simplify neural network systems by skipping ineffectual computations [31]. Many weight values are so small, that removing the corresponding edge has little to no impact to the overall accuracy of the network.

The hardware design of the equilibrium propagation requires us to understand the data flow of the network. To start, let us consider the variables in the network: the states, the inputs, and the weights. If we layout all the nodes, each node must include memory elements storing the state of that node and weights corresponding to each input source.

With a fully-connected multi-layer topology, each node in the first hidden layer is connected to all the input nodes. Equivalently, each node in the first hidden layer accumulates a weighted sum of the inputs. Thus, each node generally keeps a copy of the weights associated with each input source. The input values are then streamed to the nodes one at a time, while iterating through the weight table. In EP, since the network is run multiple times to converge to a fixed point, the same input vector is therefore required to be read multiple times.

We can reduce the complexity and latency of the hardware by pruning the network topology. By limiting the fan-out and fan-in of the nodes, we can effectively reduce the size of the memory element required to store the weights associated with each incoming connection.

Take the example network as shown in Fig. 3.1, we may restrict the fan-in and fan-out as shown in Fig. 3.2. The adjacency matrix mask for this network can thus be derived as Eq. (3.8).

A band matrix is a matrix which has its non-zero entries constrained to positions around the diagonal of the matrix. We can notice that the mask consists of blocks of such band matrices.

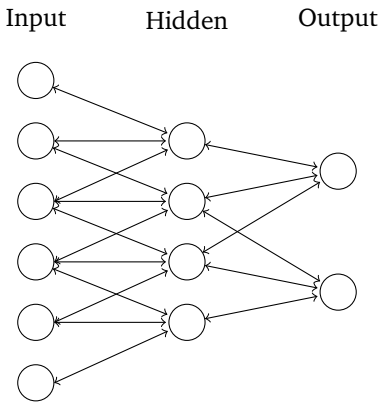

Figure 3.2: Multi-layer network pruned to the bandwidth limit.

$$M_{\text{stripe}} = \left[\begin{array}{cccccc|cc|cccc} 0&0&0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&1&1&0&0\\ 0&0&0&0&0&0&0&0&1&1&1&0\\ 0&0&0&0&0&0&0&0&0&1&1&1\\ 0&0&0&0&0&0&0&0&0&0&1&1\\ 0&0&0&0&0&0&0&0&0&0&0&1\\ \hline 0&0&0&0&0&0&0&0&1&1&1&0\\ 0&0&0&0&0&0&0&0&0&1&1&1\\ \hline 1&1&1&0&0&0&1&0&0&0&0&0\\ 0&1&1&1&0&0&1&1&0&0&0&0\\ 0&0&1&1&1&0&1&1&0&0&0&0\\ 0&0&0&1&1&1&0&1&0&0&0&0 \end{array}\right]. \tag{3.8}$$

Table 3.1: Test Accuracy for the MNIST Network with 500 Hidden Nodes

	Full Connection			Pruned		
q-bit	Scale=1	Scale=2	Scale=4	Scale=1	Scale=2	Scale=4
8-bit	9.8%	76.05%	80.98%	9.8%	52.24%	84.01%
10-bit	54.35%	89.06%	77.46%	9.92%	85.93%	82.14%
12-bit	92.26%	92.01%	80.82%	78.01%	92.91%	83.56%
14-bit	96.8%	93.24%	81.82%	95.48%	94.31%	87.54%
16-bit	97.28%	91.51%	81.59%	97.23%	95.03%	88.61%
Float		97.27%			97.01%	

Band matrices have their bandwidths reflected by the number of non-zero elements per row or per column. Here, when representing the network topology, the bandwidth corresponds to the fan-in and fan-out of nodes in the network. Consequently, the size of the weight table associated with each node directly corresponds with the bandwidth.

3.4 Results and Discussion

The following results are based on the MNIST [24] hand-written digits dataset. The network implemented has 500 hidden nodes as one layer, in addition to the 784 input nodes and 10 output nodes. We use 20 iterations in the Free Phase, and 5 iterations in the Nudge Phase, with a step size $\epsilon = 0.5$ and the clamping force $\beta = 0.5$.

The bandwidth of the pruned network is $784 - 500 = 284$. We quantize both the fully connected network and the band matrix pruned network at various numbers of bits. We also test for different degree of weight scaling, as explained in Section 3.2.

Table 3.1 summarizes the accuracy of the trained EP networks on the test set. Each row corresponds to a particular precision, where for both the fully-connected and the pruned network, weights are scaled by a factor of 1, 2, or 4.

We notice that without weight scaling, i.e., when the scale factor is one, the network fails to train when quantized to 8 bits. While increasing the learning rate can increase the training accuracy temporarily, the model quickly over-fits. Consequently, we can see that

the performance improves dramatically with the introduction of weight scaling. In fact, with 8-bit quantization, the model produces higher accuracy than that of 10-bit quantization without weight scaling. On the other hand, for higher resolutions such as 14-bit and 16-bit results, weight scaling adversely affects the test accuracy, as the effect of reduced dynamic range of weights is evident.

When training in full 16-bit precision, we notice that the proposed method of pruning the network does not significantly reduce the test accuracy. However, quantization does limit the accuracy achievable. Without weight scaling, the performance of the pruned network decays much faster than that with full connections. As we can see in Table 3.1, for the fully-connected case, the test accuracy dropped to 54.35% when the resolution is lowered to 10-bit from 12-bit. With weight scaling, however, the test accuracy only drops to 89.06%.

To better visualize the effect of quantization and pruning, Fig. 3.3 shows the error rates for the networks and the memory required to store the weights. Fig. 3.3 shows the pareto optimal results, including the pruned networks for 12, 14, and 16 bits.

Pruning the network reduces the memory usage by 2.7$\times$, while achieving an accuracy within one percentage point from that of the fully connected network. With the reduced memory footprint, the pruned network topology also requires 2.7$\times$ fewer operations.

3.5 Conclusion

EP is a novel training method that uses the same computation elements for both forward and backward propagation. While the algorithm involves a relatively long process of numerical optimization, the ability to reuse hardware for error propagation provides an opportunity to build efficient on-chip learning systems.

In this chapter, we have studied the impact of quantization and pruning on the EP algorithm. From the simulation results, we see that to effectively train the network for

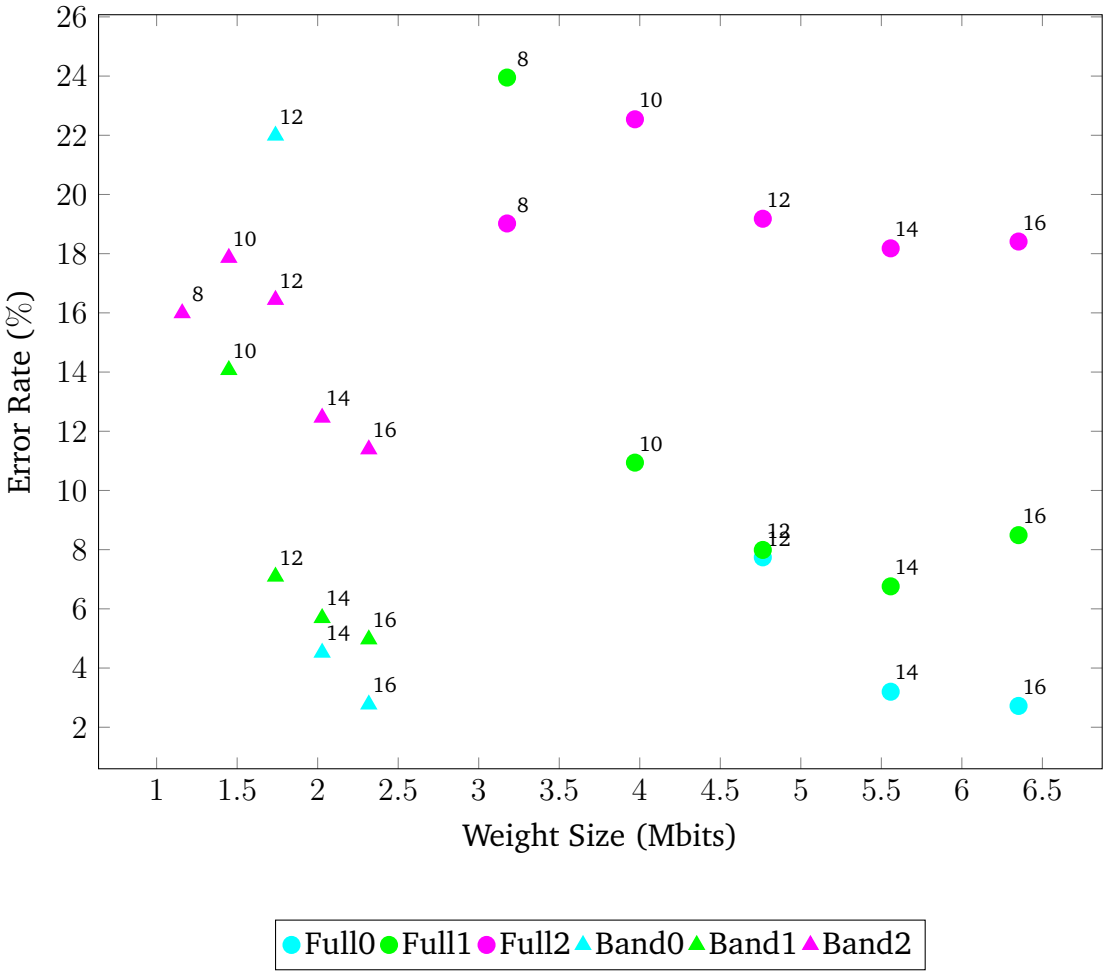

Figure 3.3: Error rate of various configurations and the memory required for the weights. Configurations that results in very high error rates are excluded from this chart to improve readability. Labels by the data point represents the number of bits.

MNIST task, at least 12 or 14 bits of resolution should be used to achieve accuracy of over 90%, which is less than ideal. However, by pruning the network to limit the bandwidth required, we can reduce the number of operations and memory footprint by 2.7×. We can then take advantage of the reduced number of operations when designing the hardware architecture. As the first step in a digital hardware implementation, the bandwidth limited pruning of the network allows an efficient architecture to take advantage of the special network topology.

CHAPTER 4

Digital Hardware Implementation

In Chapter 3 we studied the effect of quantization and introduced a pruning method. In this chapter we propose a proof-of-concept hardware architecture for the pruned EP. Similar to the software simulation in the previous chapter, we choose a network topology with one hidden layer of 500 nodes, as shown in Fig. 4.2. The connections are pruned with the proposed band matrix based technique. Fig. 4.1 renders the mask matrix for the connections between the input and the hidden layer.

4.1 Architecture and Dataflow

4.1.1 Hidden Node Architecture

The majority of the hardware is for the hidden nodes, as there are 500 hidden nodes; in contrast there are 10 output nodes. No dedicated computation element is needed for the input node, since we are always clamping the state associated with the input nodes to the

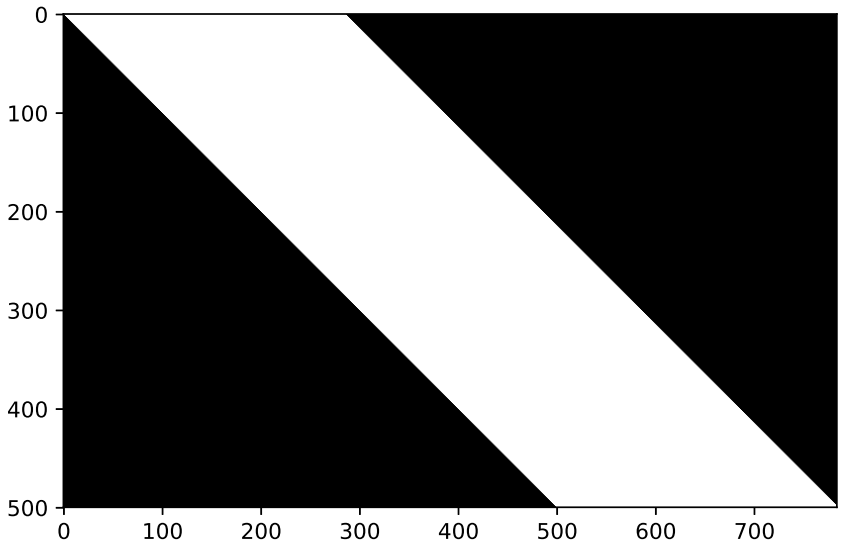

Figure 4.1: Bandwidth limit pruning mask for the proposed 784-500-10 network. Here a white pixel shows the existance of a connection.

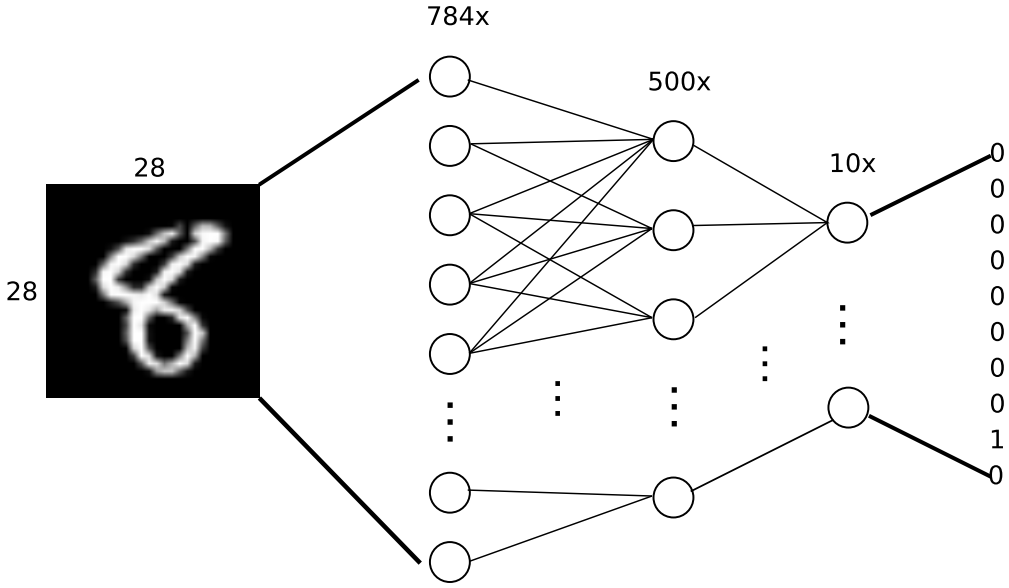

Figure 4.2: The network used in this design.

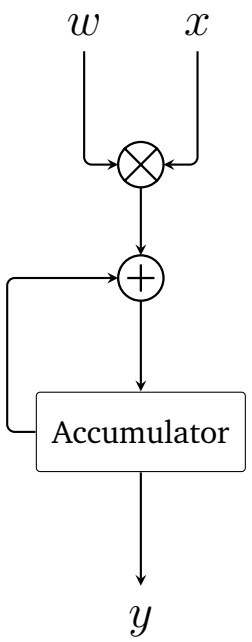

Figure 4.3: An example MAC unit — partial sums are accumulated in a register to perform an inner product.

input values. Effectively, the input nodes can be considered as constant-valued neighbors connecting to the hidden nodes.

During Free and Nudge Phase, each hidden node implements the update rule introduced in Eq. (2.9):

$$s_i := s_i + \epsilon \times (\mathbf{w} \cdot \mathbf{x} + b_i - s_i)\,, \tag{4.1}$$

where $\mathbf{w}$ is an array of the weights corresponding to the list of connected inputs $\mathbf{x}$. The inner product is part of the matrix vector multiplication, which represents nodes in the network accumulating inputs from each of their neighboring nodes via weighted edges.

Such matrix multiplications are often performed using a parallel array of MAC units [5, 7, 33], as shown in Fig. 4.3. Each MAC unit computes an inner product serially by accumulating the product of each input and its corresponding weight.

Fig. 4.4 is a simplified architecture for the hidden node. A MAC unit is used to compute the inner product $\mathbf{w} \cdot \mathbf{x}$. The difference of the bias and the current state value is first loaded

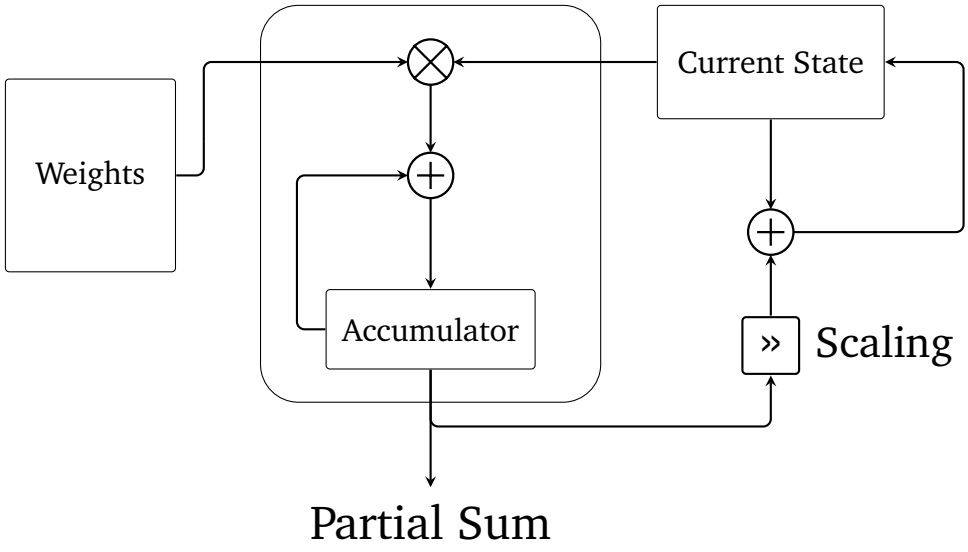

Figure 4.4: Simplified Diagram of the Hidden Node.

into the partial sum register in the MAC unit. After the products from all the connected inputs are accumulated, the current state value is updated by adding a scaled version of the partial sum. If we choose the step size ϵ as a negative power of two, we can achieve the scaling by truncating the least significant bits.

The hidden nodes are responsible for computing partial sums for edges originated from the input nodes. For each incoming edge, a weight value is stored in the weight table. The size of these weight tables depends on the number of incoming edges. With the band matrix pruning applied, we effectively limit the size of the weight table to the bandwidth ($784 - 500 + 1bias = 285$ in this case).

When training with EP, both the free phase and the nudge phase states needs to be saved. We can replace the current state register in Fig. 4.4 with the circuits as shown in Fig. 4.5. By asserting 0 or 1 on the state select signal, we can load and read from the correct state register.

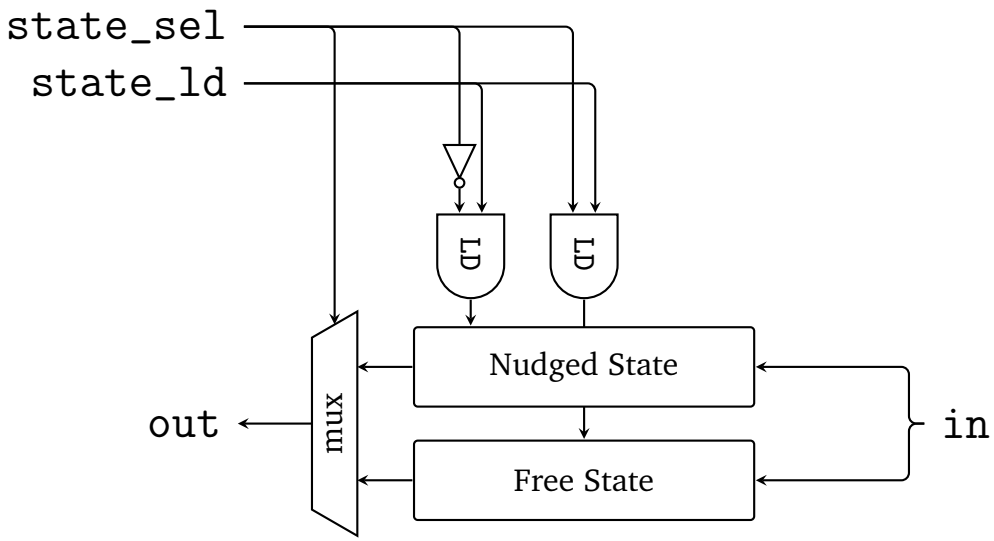

Figure 4.5: Circuits to Select Between Two States of EP Training.

4.1.2 Band Matrix Data Flow

As introduced in Section 3.3, the network connections are pruned to limit the required bandwidth. The hidden nodes accumulate weighted sums of connected input values, and the connection is characterized by a band matrix with a bandwidth of 284. In order to better explain the data flow, we use the small-network as shown in Fig. 3.2 in Section 3.3 as an example.

The mask matrix corresponding to the edges between the input and hidden layer is the lower left block in Eq. (3.8):

$$M_{\text{ih}} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \tag{4.2}$$

In this case, each of the four hidden nodes accumulates partial sums from three input nodes. For example, the first hidden node accumulates partial sums from the first three input nodes, and the second hidden node accumulates partial sums from the three input nodes starting

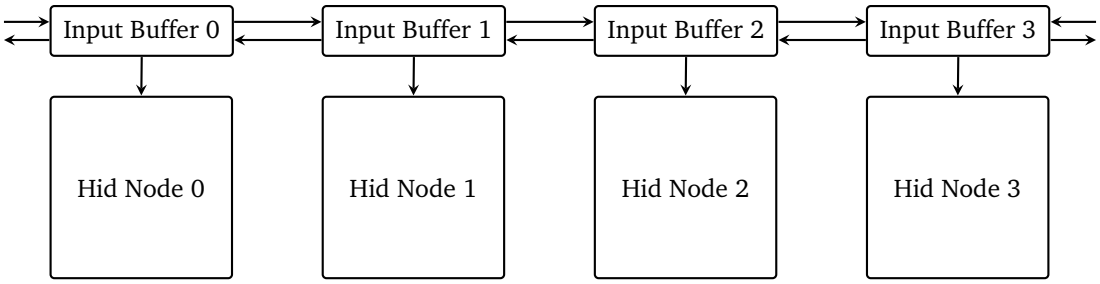

Figure 4.6: Data path for Loading the Input State Value.

from the second input node. Each of the individual hidden nodes accumulates hidden sums serially, and so we can split the corresponding matrix into three time steps:

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \; M_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, and \; M_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.3)$$

We can see that the input values are passed from one node to the next. Fig. 4.6 shows the simplified model for the example network. Here hidden nodes 0 through 3 correspond to the four rows of the matrix M_{ih}. After an initial loading period, where inputs are shifted into the buffers, the input buffer 0 contains the value from the first input node, input buffer 1 contains the value from the second input node, etc. The hidden nodes then accumulate the partial sum derived from the input value in the input buffer. For the second time step, the input buffers shift every value to the left, and hidden node 0 now reads the input value that was in input buffer 1.

When the last input value is shifted into the input buffer, the system has completed one iteration for the free or nudge phase. For the second iteration, instead of flushing the input buffer, we shift the data to the right to avoid the overhead. Equivalently, we are processing the matrices in Eq. (4.3) in the reverse order: $M_3 \rightarrow M_2 \rightarrow M_1$.

We can thus expand the idea for the EP hardware, where we shift in the input values through a buffer queue.

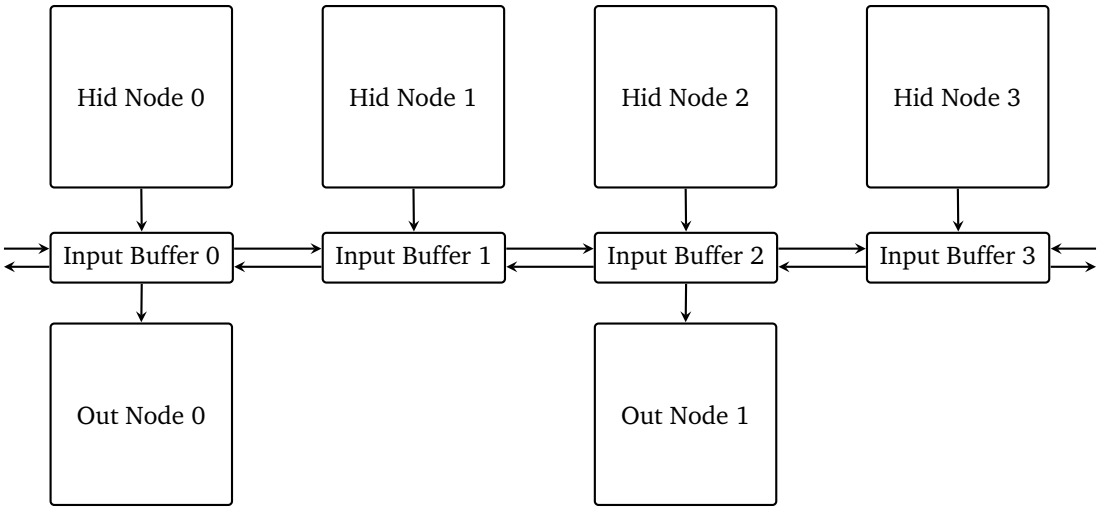

Figure 4.7: Data path for Loading the Output State Value.

4.1.3 Output Nodes

The output nodes are responsible for applying the nudge in EP. In addition to the update equation in Eq. (4.1), an error term, $\beta(d_i - y_i)$, needs to be added to the partial sum. Again, if we choose a $beta$ value to be a power of two, the scaling can be done simply with a bit shift.

Since there are many fewer output nodes than hidden nodes (10 as opposed to 500), we compute the partial sums between the hidden layer and the output layer in the output node. While the output node states are updated similarly to the hidden nodes, except the error term, the product of the edge weight with the current state is simultaneously produced. The hidden node accumulates the product to complete its state update.

As shown in Fig. 4.7, before each iteration, we keep a copy of the hidden states in a shift register queue similar to that of the input values. These states are shifted together with the input queue as the output nodes compute partial sums from these values.

4.1.4 Weight Update

As the final phase of the EP on-chip learning, the weight update takes place within each node. Firstly, the bias is updated based on the difference of the final state values from nudge and free phase, as shown in Eq. (2.11). Now for the hidden nodes, the equation for the hidden nodes can be simplified, since the neighboring nodes, i.e., the input nodes, are clamped to the input values. Effectively, for edges connecting input nodes to the hidden nodes, we have $s_i^\beta = s_i^0$. Consequently, the weight update equation for the hidden nodes can be simplified as:

$$\Delta w_{ij} = \frac{\gamma}{\beta} \cdot s_i \left(s_j^\beta - s_j^0 \right), \tag{4.4}$$

where γ is the learning rate, and s_i is the input value. Hence, we can compute both the weight update and bias update for the hidden nodes from the difference between the two final states.

4.2 Implementation Results and Discussion

Table 4.1: Number of cycles in each phase of processing for one input image.

Process	Load	Free0	Free1	...	Free19	Nudge0	...	Nudge4	Weight	...
#Cycles	500	286	286	...	286	286	...	286	286	...

The scheduling of processing one input image is shown in Table 4.1. When initiated, the system loads part of the input image to fill in the input queue. Then, before each iteration, the system first loads the initial value of the partial sum, The initial value is determined by the difference between the current state and the bias. The queue then begin shifting the inputs through the hidden nodes, as a counter serving as the address for the weight table in each node increments. When the counter reaches the end of the table, i.e., 284, the partial sum is then used to update the current state in the next two cycles (a total of 286). After the new state is loaded in to the state register, we proceed with the next iteration.

After both the free and nudge phases are complete, we shift the input buffer again to update the weights instead of the states. Thus, with 20 iterations for the free phase and 5 for the nudge phase, an image requires

$$500 + 286 \cdot (20 + 5 + 1) = 7936 \text{ Cycles} \tag{4.5}$$

to complete, with a clock speed of 5 MHz, we can achieve a throughput of:

$$\text{Throughput} = \frac{5 \text{ MHz}}{7936 \text{ Cycles per image}} \approx 630 \text{ Images per second.} \tag{4.6}$$

We synthesized the hardware architecture targeting a Xilinx Kintex-7 (xc7k355tffg901–3) FPGA device using Vivado 2018.2. Using 16-bit, 14-bit, or 12-bit quantization, we have the following implementation results in Table 4.2.

Table 4.2: Implementation Results and Corresponding Test Accuracy from Table 3.1.

Precision	LUT	FF	BRAM (36Kb)	DSP	Power (mW @5MHz)	Fmax (MHz)	Accuracy (%)
16-bit	87,483	64,097	255	1,030	376	24.104	97.23
14-bit	79,239	58,019	255	1,030	367	26.127	95.48
12-bit	70,983	51,941	255	1,030	358	27.223	92.91

With the 14-bit quantization, we can achieve a maximum frequency of 26.127 MHz, which corresponds to a maximum throughput of

$$\text{Throughput} = \frac{27.127 \text{ MHz}}{7936 \text{ Cycles per image}} \approx 3418 \text{ Images per second.} \tag{4.7}$$

The back-propagation algorithm was implemented on FPGAs in [36, 37], however, they are limited to small-scale networks with fewer than 100 inputs due to the complexity of the back-propagation algorithm. There are works that attempted to reduce complexity and implemented on-chip learning on FPGAs and ASICs for tasks such as the MNIST image

[†]Approximated from figure
[†]Approximated from figure

Table 4.3: ASIC Synthesis Results

	EP-14	EP-16	EP-16	MLP [34]	MLP [35]
Technology	65 nm	65 nm	32 nm (scaled)	28 nm	FPGA Spartan-6
Learning Capability	Yes	Yes	Yes	Yes	Yes
Total Area (mm^2)	32.17	36.76	11.14	0.8†	9,123 LUTs, 4,771 Regs, 16 BRAMs
Total Power (W)	2.04	2.57	1.02	–	–
Nominal Frequency (MHz)	100	100	100	120.5	78
Latency (μs/sample)	79.36	79.36	79.36	2.17	12000
Energy Per Sample (nJ)	161.81	203.88	80.58	80†	–
Bit Width	14	16	16	8	8
Test Error on MNIST (%)	4.52	2.77	2.77	1.9	2.05

recognition task. To facilitate the comparison with these works, the designs with 14-bit and 16-bit precision are synthesized for TSMC 65-nm technology using Cadence Genus Synthesis Solution 18.10, and the results are shown in Table 4.3.

[35] implemented on-line learning on an MLP network (784-600-600-10). With binary activations, the training algorithm reduces the required memory overhead. Using 8-bit weights at 78 MHz, the network can achieve an average training time of 12 milliseconds per example (83.33 examples per second). Although this work can achieve a 41$\times$ speedup in latency of training on FPGA in comparison with [35], the hardware utilization is much higher in this work in terms of the number of lookup tables and registers used in the FPGA implementation as shown in Table 4.2.

[34] implemented a smaller MLP network of (784-200-100-10) using stochastic computing, which performs arithmetic operations using random binary bit streams in order to reduce the area and power. To achieve similar test accuracy, the EP algorithm requires 16-bit precision, which results in 2.5 $\times$ the energy per training sample as required in the MLP network. It is worth noting that [34] is synthesized for a 28 nm process. To improve the comparison, the area and power results are scaled from 65 nm@1V to 32 nm LP@0.97V according to Table 4 and Table 5 in [38]. The scaled energy per sample is comparable with

that of [34], but the total area is still 14× that of the MLP.

The high energy consumption and hardware cost mainly stem from the inherent complexity of the EP algorithm. E.g., while it takes multiple iterations (20 in this case) to achieve convergence in the Free Phase, an MLP network accomplishes the same inference task in one pass. What is worse, the number of operations in one iteration of the EP algorithm is similar to or larger than that of the MLP with the same network topology.

Furthermore, with the current quantization method, the EP algorithm requires high numerical precision in weights to train effectively. We can see that a two-bit increase in precision results in a 14.3% increase in area, as well as a 25.98% increase in the estimated power. The larger bit width contributes to the hardware cost as well.

To improve the hardware efficiency, further research must be conducted to optimize the algorithm to reduce the number of operations, as well as the required bit width. In addition to the quantization method proposed in Chapter 3, alternative rounding methods or stochastic pruning could be attempted to reduce the number of bits needed for the target error rate. Simulations could be performed to segment the network in order to reduce data dependency. Furthermore, improvements to the EP algorithm itself could potentially reduce the number of iterations required for convergence, thus reducing the total number of operations required for each training example.

Moreover, [34] has shown promising results in using stochastic computing for the computation of back propagation. By eliminating conventional array multipliers with simple logic gates, the total area can be significantly reduced. In the next chapter, we will introduce a preliminary study to apply stochastic computing techniques in EP.

4.3 Conclusion

In this chapter we introduced a digital hardware architecture to implement the EP architecture. We show that we can take advantage of the band matrix pruning technique by shifting

the input values through the hidden nodes. Finally, we synthesized the architecture on both an FPGA and an ASIC platform to investigate and compare the hardware performance.

From the synthesis results, we see that the system is not yet competitive against MLP networks and back-propagation due to the iterative process of EP. It is evident that further optimization is required to deploy EP systems in the field.

CHAPTER 5

Towards a Stochastic Computing Implementation

In Chapter 3, we introduced the quantization and pruning of the EP algorithm, and we subsequently proposed a hardware architecture for a digital hardware implementation in Chapter 4. In this chapter, we will introduce Stochastic Computing [39] as a way to further optimize the implementation of EP algorithm in hardware.

5.1 Basics of Stochastic Computing

Stochastic Computing is a technique to simplify computing hardware. Real numbers are encoded as streams of independent random bits. Computations such as multiplications are conducted by bitwise operations on the bit streams. These bitwise operations can be implemented with simple logic gates, and this simplifies the circuit and reduces the hardware cost.

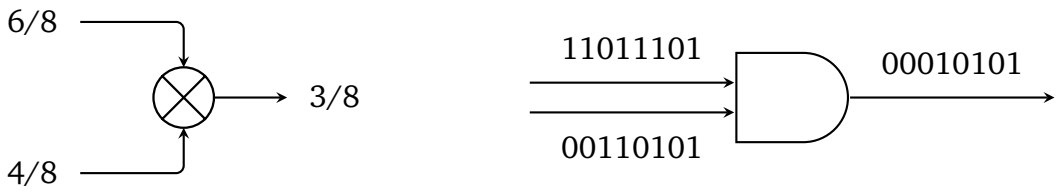

Figure 5.1: Example of performing multiplication with an AND-gate.

5.1.1 Unipolar Representation

The most basic form of stochastic computing uses the unipolar representation, which converts a real number $x \in [0, 1]$ to a stream of randomly generated bits, $\underline{X} = [q_0 q_1 \dots q_n]$, where the probability of generating a 1 is the real number x, i.e., $\Pr(q_i = 1) = x$. Since the joint probability of two independent random variables equals the product of their probabilities, we know that given two independent Bernoulli random variables $\underline{a} \perp\!\!\!\perp \underline{b}$,

$$\Pr(\underline{a} = 1 \text{ and } \underline{b} = 1) = \Pr(\underline{a} = 1) \cdot \Pr(\underline{b} = 1). \tag{5.1}$$

With this property, instead of a conventional array multiplier, we can use a single AND-gate to multiply two numbers.

The expected frequency of ones in a series of independent Bernoulli random variables is the probability parameter of the distribution. Fig. 5.1 shows an example of a two-input multiplication. For demonstration purpose, we choose a stream length of eight bits. Here the two inputs can be converted to stochastic bit streams, and we can see that the bit stream corresponding to the number 6/8 or 0.75 has 6 ones out of the 8 bits. Similarly, the number 4/8 or 0.5 has 4 ones out of the 8 bits. When we apply the bitwise AND to the bit stream, the generated bit stream contains 3 one's out of the eight bits.

Now it is worth noting that since the bits are stochastically generated, the bit stream may not result in the precise frequency of ones corresponding to the product. However, as the stream length increases, the expected value of the frequency of ones approaches the

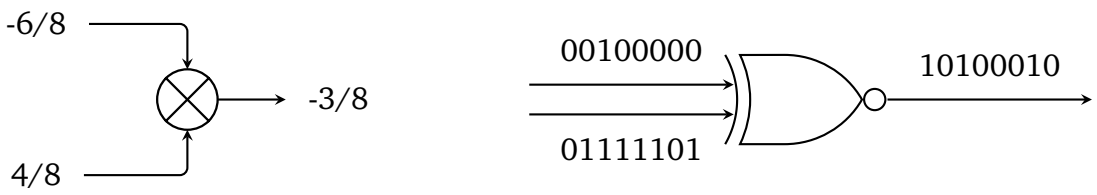

Figure 5.2: Example of performing multiplication with an AND-gate.

product of the two probabilities.

5.1.2 Bipolar Representation

With the most basic form of stochastic computing, we can only represent numbers from zero to one. However, a bipolar representation allows us to represent numbers from -1 to 1. If we map a number $x \in [-1, 1]$ into $[0, 1]$ using $f(x) = (x + 1)/2$, we can use an XNOR-gate instead of an AND-gate to perform the multiplication of two bit stream.

Fig. 5.2 shows an example bipolar computation. Effectively, a zero in the bipolar bit stream represents -1, and the real number represented by the bit stream is the sample mean of the stream of -1 and 1.

5.1.3 Length of Stochastic Bit Stream

While stochastic computing can significantly reduce the area for a multiplication, replacing full array multipliers with simple logic gates, it comes with a cost of additional latency. Exponentially longer bit streams are required to achieve higher accuracy. Fig. 5.3 shows the mean square error of representing 10000 floating numbers with stochastic bit streams. It is worth noting that the bipolar representation requires longer bit streams to achieve the same accuracy.

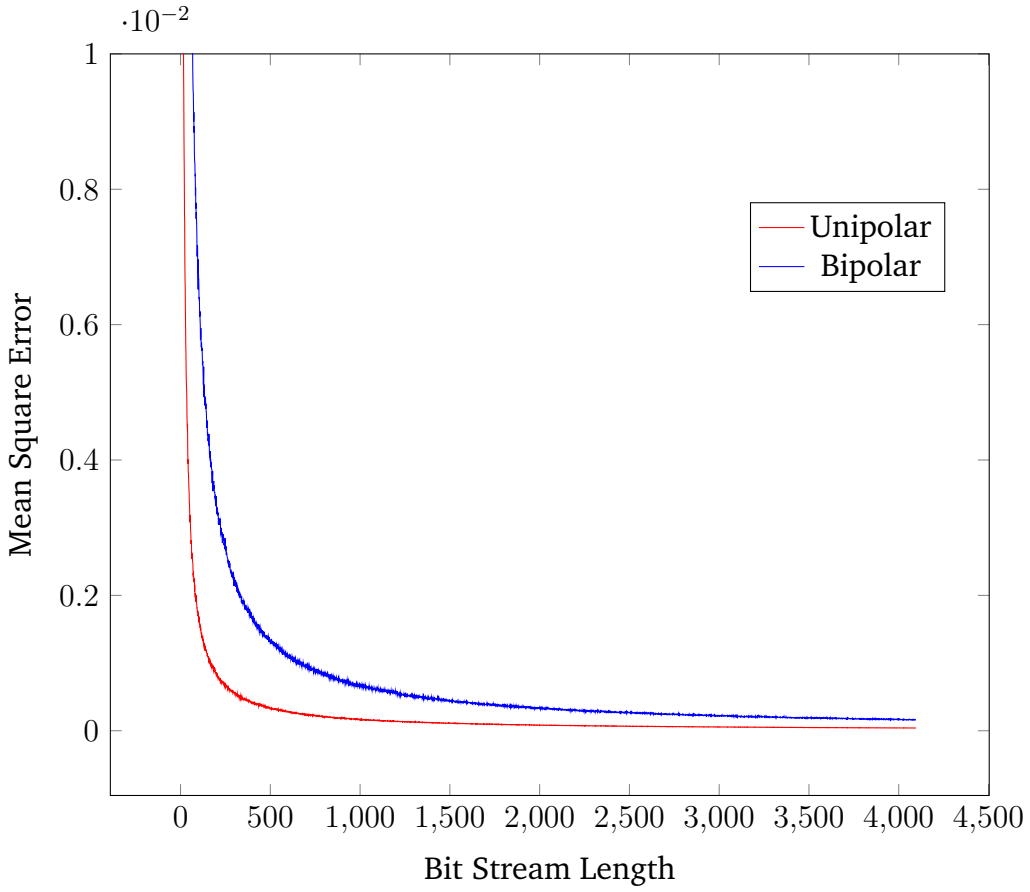

Figure 5.3: Accuracy of Representing Floating Point Numbers with Various Lengths of Stochastic Bit Streams.

5.2 Equilibrium Propagation with Stochastic Computing

As mentioned in Section 1.2 and [9], the EP training algorithm involves an iterative optimization process, for up to 500 iterations to converge for a network with 1500 hidden nodes. It may initially seem ill-advised to introduce additional latency to replace the computations with lengthy bit stream operations. However, when applying stochastic computing to existing algorithms, it is important to adapt the algorithm itself instead of replacing individual operations directly. In this section, we propose the modified EP algorithm, and show some preliminary results.

5.2.1 Algorithm Modification

Eq. (4.1) shows the computation involved during the iterative optimization process for one node, from which we can derive an alternative view of the computation:

$$s_i := s_i + \epsilon \times (\mathbf{w} \cdot \mathbf{x} + b_i - s_i) \tag{5.2}$$

$$:= s_i - \epsilon s_i + \epsilon (\mathbf{w} \cdot \mathbf{x} + b_i) \tag{5.3}$$

$$:= (1 - \epsilon) s_i + \epsilon (\mathbf{w} \cdot \mathbf{x} + b_i) . \tag{5.4}$$

Suppose that we call the weighted sum $s' = \mathbf{w} \cdot \mathbf{x} + b_i$, then the weight update can be written as:

$$s_i := (1 - \epsilon) s_i + \epsilon s'. \tag{5.5}$$

Here the equation resembles that of an exponential moving average. In other words, each state tracks the moving average of the weighted sum of the neighboring states, where the weighted sum from the more recent states has a higher weight than that from past states.

TFM [40] is used to de-correlate stochastic bit streams in stochastic decoders. It also closely relates to the exponential moving average as shown in Eq. (5.5). Now, for the moment, if we assume that we can produce an accurate and uncorrelated bit stream repre-

senting the weighted sum s', we should be able to replace the real value s' with realizations of the random variable $\underline{s}'$, where $\mathbb{E}(\underline{s}') = s'$. Essentially, the single node reduces to a TFM with scaling factor ϵ for the bit stream $\underline{s}'$.

Technically, this modification so far does not perform any stochastic arithmetic mentioned before. However, if we naively adopt stochastic computing to replace individual computations, the latency of the algorithm grows with the length of the bit stream, which in turn grows exponentially with the required precision. When considering the iterative process in Eq. (4.1) as a TFM, we no longer treat each iteration as individual computations, but as the processing of a single bit out of the bit stream.

It is worth noting that the weighted sum s' can be negative. However, bipolar stochastic numbers are particularly inaccurate around zero [41]. In fact, we have not been successful in training the model using simply the bipolar representation. Instead, we can use a two-stream representation [42] including a sign stream and a magnitude stream.

As mentioned, for this preliminary study, we assume an accurate and uncorrelated bit stream for $s' = \mathbf{w} \cdot \mathbf{x} + b_i$ since performing an inner products using stochastic computing has been well studied [8, 41, 43, 44].

5.2.2 Experiments and Preliminary Results

To simulate the accurate and uncorrelated bit stream $\underline{s}'$, we first compute s' in floating point. We then extract the sign of the number, and sample its absolute value from a Bernoulli distribution. The sign stream and the magnitude stream are then sent to the TFM to updating the states using Eq. (5.5). Fig. 5.4 shows the circuit for the TFM replacing the state update logic. As we can see, the sign and magnitude bit streams are converted to two's complement numbers using an AND-gate.

The network is recurrent in nature, and is sensitive to noise in the state update. If we pick a larger step size, the numerical error propagates throughout the network and back to the node itself. Here we pick a small step size of $2^{-6} = 0.015625$, which corresponds to

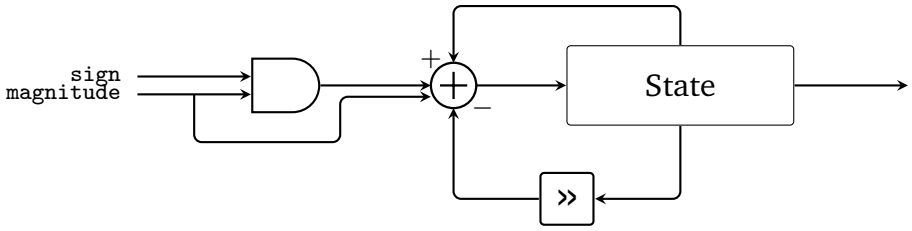

Figure 5.4: Tracking Forecast Memory, assuming a 1 in the sign stream means negative.

an arithmetic shift of 6 positions to the right. In hardware, we can directly get the number by truncating the least significant six bits. During simulation, we also need to quantize the state values.

We first evaluate the effectiveness during inference with pre-trained weights. Fig. 5.5 shows an example of the state evolution during one inference pass. Fig. 5.6 shows the same input and weight with the TFM quantized to 8 bits. We trained the network using floating-point numbers, and then quantized the parameters to 8, 10, and 12 bits. The performance with and without using the TFM are obtained from the test set. In the experiment with the TFM, the number of iterations for the Free and Nudge phases are chosen to be 400 and 200, respectively. Table 5.1 compares the test set error rate for the quantized model using TFM with binary radix simulation in Section 3.4.

Table 5.1: Test Performance of Pre-Trained Model.

Quantization	Error Rate	
	TFM	Binary
Floating Point	2.87%	2.42%
8-bit	4.7%	4.25%
10-bit	3.01%	2.54%
12-bit	2.88%	2.42%

We then train the model using quantized TFM and generating sign and magnitude stochastic bit streams on the fly. The step size is again chosen to be 2^{-6}, with 400 iteration for the free phase, and 200 iterations for the nudge phase.

As shown in Table 5.2, when training with TFM as the state update logic, the error

rate is better than that of the quantized model as shown in Table 3.1 since during this experiment the weights are kept as floating point.

Table 5.2: Test Performance of Training Model using TFM

Method	Error Rate	
	TFM	Binary
8-bit TFM	13.49%	19.02%
10-bit TFM	5.31%	10.04%
12-bit TFM	5.73%	7.74%

Now if we replace each operation with the stochastic computing equivalent, we will need at least 1024 bits in each bit stream for a 10-bit precision. When multiplied by the 20 iterations required, each image requires 20480 total cycles to process. By replacing the state update with a TFM, even though more iterations are needed, much fewer cycles are required to process each image. Meanwhile, the use of stochastic logic significantly reduces the hardware footprint.

5.3 Relationship to Spiking Neural Network

Stochastic Computing has been shown to improve SNNs [45]. By converting the state update mechanism to a TFM, we implement the exponential moving average in Eq. (5.5). It is worth noting that using stochastic computing in the EP algorithm draws similarities to SNNs with LIF neurons by considering the state values as membrane potentials.

In a digital LIF neuron j, the membrane potential $V_i(t)$ is updated at each time period:

$$V_i(t) = V_i(t-1) + \left(\sum_{j=0}^{N-1} x_j(t) w_j\right) - \lambda V_i(t-1), \tag{5.6}$$

where the synaptic weights w_j are integrated if the corresponding neuron x_j spiked at that time step. A leak $\lambda V_i(t-1)$ proportional to the current state is then applied regardless of the spikes. The neuron will then spike and reset if the membrane potential exceeds a

certain threshold.

The EP algorithm with stochastic computing has a state update equation similar to Eq. (5.6), however, the modified EP does not have a mechanism to fire based on a threshold. Instead, a stream of bits is generated stochastically based on the state value. Since the state values are always clamped between zero and one, the unipolar bit stream parallels the series of spikes in the LIF neuron.

Fig. 5.7 compares the threshold function of LIF neurons with the stochastic bit stream generators. It is worth noting that some LIF models include an optional additive noise to the threshold value [23], which produces stochastic spikes that resemble those of a stochastic bit stream used in stochastic computing.

5.4 Conclusion

In this chapter we applied stochastic computing techniques to Equilibrium Propagation. Since directly replacing computations with bit stream computation is inefficient with high latency, we modified the algorithm itself by replacing the state update computation with a Tracking Forecast Memory unit. Preliminary results are obtained on the effect of this modification that shows similar classification accuracy on MNIST as the quantized counterpart. Finally, we indicated that the EP algorithm with stochastic computing technique is closely related to Spiking Neural Networks with LIF neurons.

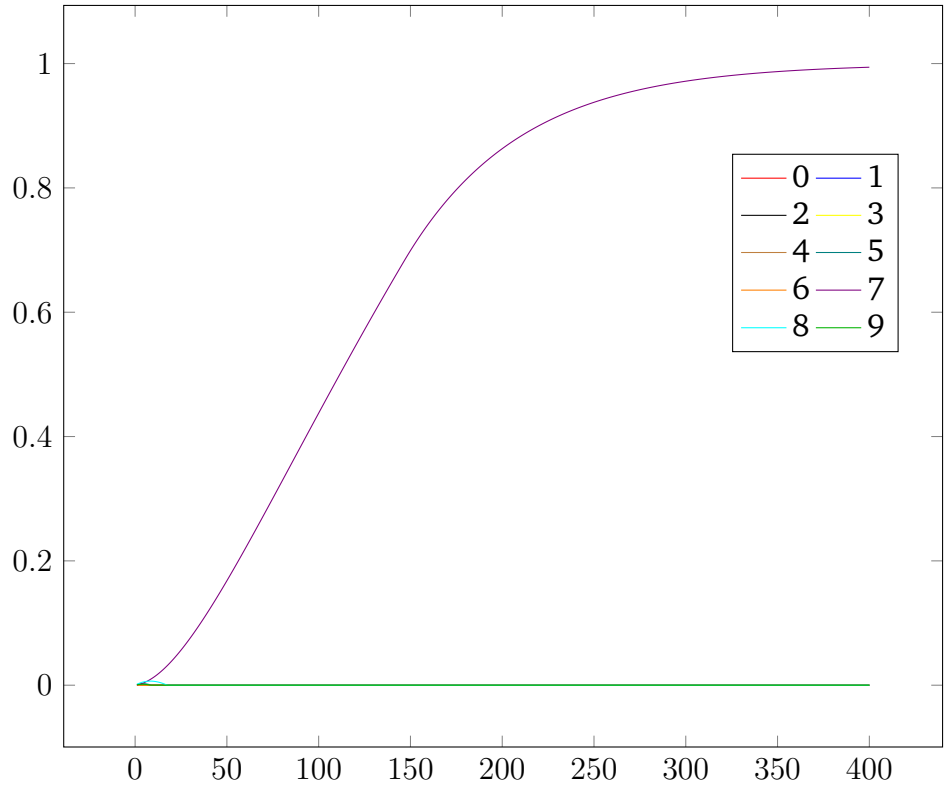

Figure 5.5: State Evolution for the Output Layer with Floating Point

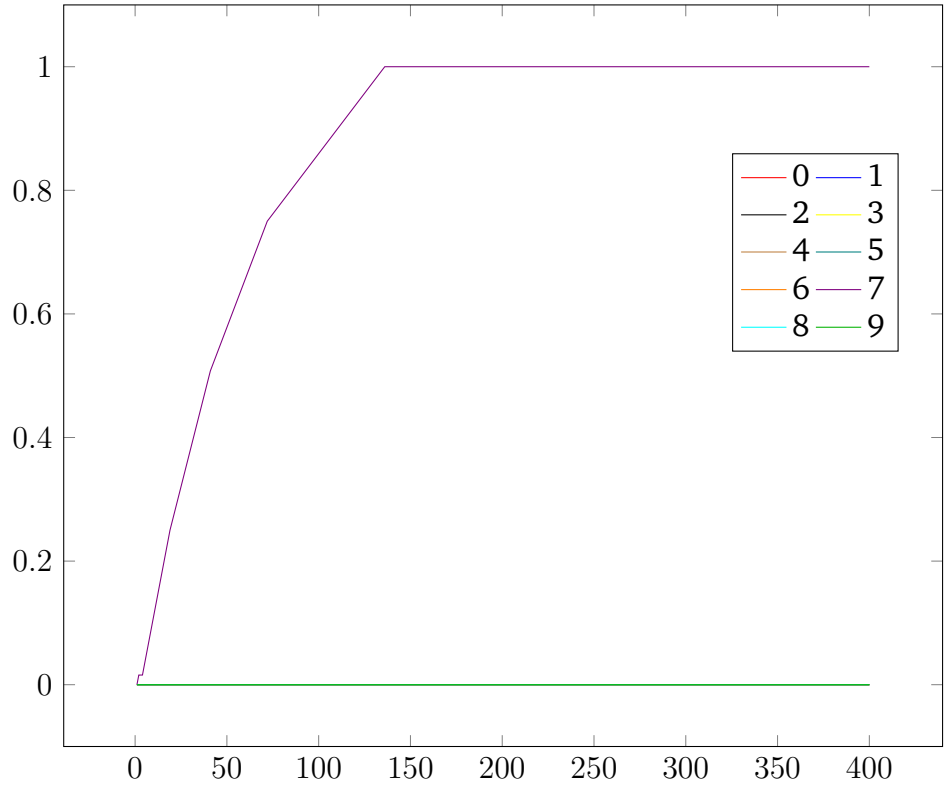

Figure 5.6: State Evolution for the Output Layer Using TFM

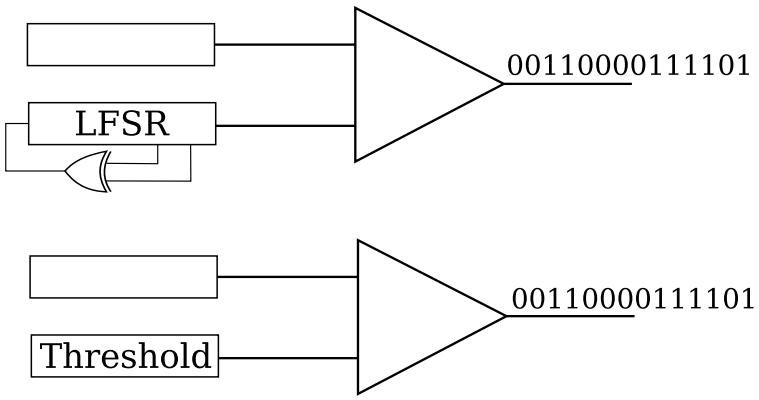

Figure 5.7: TOP: In Stochastic computing, the bit stream is generated by comparing the state value with a random source. BOTTOM: In an LIF neuron, the membrane potential is compared with a fixed threshold function.

CHAPTER 6

Conclusions and Future Work

6.1 Summary

In this thesis, we studied the Equilibrium Propagation algorithm in detail. By applying quantization to the algorithms, we found that training the network effectively requires at least 12 or 14 bits. In order to further reduce the network size for hardware implementation, we then pruned the network topology to limit the fan-in and fan-out of each node. With many fewer parameters, the pruned network can still achieve comparable results to the fully connected network.

With quantization and pruning, we then proposed a hardware architecture and data flow optimized for the band matrix pruned network. With 14-bit precision, we are able to achieve a throughput of 630 images per second with 367 mW @5 MHz on a Xilinx Kintex-7 FPGA. However, due to the limitation of the algorithm itself, such a digital implementation using binary radix arithmetic is not competitive in comparison with other on-chip learning algorithms.

Hence, we next attempted to further optimize the system by introducing the stochastic computing technique. While the length of the bit streams increases the latency of computation, it reduces a conventional array multiplier to an AND-gate or an XNOR-gate. Using a Tracking Forecast Memory in place of the state update logic, we combined the state update time steps with the bit stream time steps. As preliminary results, we showed that by the modification does not significantly degrade the performance of the network.

6.2 Future Work

While this thesis provides the foundations to build an efficient on-chip learning hardware using Equilibrium Propagation, there is much more work to be done. In this section we introduce a few directions to extend upon this work.

6.2.1 Improved Binary Quantization

The results in Section 3.4 indicate that the algorithm depends on sufficiently high precision to effectively train the network. The main reason, as the simulations have shown, is that the weight gradient is often less than the granularity of the quantization level. As a result of truncation, no parameters can be updated. To address this issue there are a few techniques that can be used to quantize the network parameters.

- High resolution weight with low resolution state values: the number of bits for the weight values can be greater than that for the state values.

- Quantization with rounding: for hardware simplicity, when a number is scaled, the least significant bits are currently truncated. If rounding is used, lower resolution may be used since the weight gradients could be registered even when less than the granularity.

The above methods as well as other quantization techniques could improve the classification performance of the network at a cost of additional hardware complexity. Further simulations need to be performed to evaluate whether the improvement justifies the extra cost.

6.2.2 Flexible and Tile-able Hardware

The hardware architecture proposed in this thesis is optimized for a fixed network topology. However, to accommodate other topologies and size, we could design a hardware architecture that is both flexible and expandable.

One key advantage of EP for hardware implementation is the ability to update weights locally. If we wish to tile multiple chips to implement a larger network, we must consider the communication between chips. With EP, each chip only needs to transmit and receive the state values on the boundary during the free and nudge phase. Since the weight gradients only depend on the final states of two connected nodes, most of the weights in the one chip can be updated without requiring any information from neighboring chips.

6.2.3 Full Stochastic Computing Implementation

Chapter 5 demonstrated a proof-of-concept of using stochastic computing in EP computations. We assumed that the weighted sum from neighboring nodes is an accurate and uncorrelated bit stream, however, further experimentation is needed to investigate the best way to compute the inner product, which was performed in conventional method for in this preliminary study.

There are alternatives to compute the exponential moving average to the TFM. The stochastic bit stream generated from the state value itself could be used as the leakage in place of the truncated state value used in this thesis. Further experiments are required to compare the performance and hardware cost.

In order to build a complete stochastic computing EP system, additional simulations are also needed to determine the number and placement of Stochastic Number Generators (SNGs). Experiments show that SNGs and stochastic-to-binary converters take approximately 80% of the hardware resource of a stochastic computing system [46]. To reduce the hardware cost, pseudo-random sources are often shared among computation elements. When sharing the pseudo-random sources, the accuracy could be expected to degrade due to the violation of Eq. (5.1).

With a full stochastic computing implementation, we can then design a tile-able EP system requiring much less bandwidth. Each node in the tile would only need one wire to transmit its state value as a bit stream to neighboring tiles, instead of the full binary representation numbers.

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[3] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[5] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[6] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the*

47th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622.

[7] A. Ardakani, C. Condo, M. Ahmadi, and W. J. Gross, "An Architecture to Accelerate Convolution in Deep Neural Networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 4, pp. 1349–1362, Apr. 2018.

[8] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2688–2699, Oct. 2017.

[9] B. Scellier and Y. Bengio, "Equilibrium Propagation: Bridging the Gap between Energy-Based Models and Backpropagation," *Frontiers in Computational Neuroscience*, vol. 11, May 2017.

[10] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences*, vol. 81, no. 10, pp. 3088–3092, May 1984.

[11] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. J. Huang, "A Tutorial on Energy-Based Learning," *Predicting structured data*, vol. 1, no. 0, p. 59, 2006.

[12] B. Scellier, A. Goyal, J. Binas, T. Mesnard, and Y. Bengio, "Generalization of Equilibrium Propagation to Vector Field Dynamics," *arXiv:1808.04873 [cs, stat]*, Aug. 2018.

[13] P. O'Connor, E. Gavves, and M. Welling, "Initialized Equilibrium Propagation for Backprop-Free Training," in *International Conference on Learning Representation*, New Orleans, LA, Sep. 2018.

[14] N. Rochester, J. Holland, L. Haibt, and W. Duda, "Tests on a cell assembly theory of the action of the brain, using a large digital computer," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 80–93, Sep. 1956.

[15] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, p. 386, May 1959.

[16] T. Zhang, "Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: ACM, 2004, pp. 116–.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[18] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[19] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, Aug. 1952.

[20] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.

[21] L. Lapicque, "Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation," *J. Physiol. Pathol. Gen.*, vol. 9, pp. 620–635, 1907.

[22] R. B. Stein, "A theoretical analysis of neuronal variability," *Biophysical Journal*, vol. 5, pp. 173–194, Mar. 1965.

[23] A. S. Cassidy, P. Merolla, J. V. Arthur, S. K. Esser, B. Jackson, R. Alvarez-Icaza, P. Datta, J. Sawada, T. M. Wong, V. Feldman, A. Amir, D. B.-D. Rubin, F. Akopyan,

E. McQuinn, W. P. Risk, and D. S. Modha, "Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, Aug. 2013, pp. 1–10.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, Feb. 2016.

[26] A. Ardakani, Z. Ji, S. C. Smithson, B. H. Meyer, and W. J. Gross, "Learning Recurrent Binary/Ternary Weights," *arXiv:1809.11086 [cs, stat]*, Jan. 2019.

[27] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3123–3131.

[28] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," *arXiv:1511.06393 [cs]*, Jun. 2016.

[29] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *arXiv:1606.06160 [cs]*, Feb. 2018.

[30] L. Hou and J. T. Kwok, "Loss-aware Weight Quantization of Deep Networks," in *International Conference on Learning Representations*, Feb. 2018.

[31] A. Ardakani, Z. Ji, and W. J. Gross, "Learning to Skip Ineffectual Recurrent Computations in LSTMs," *arXiv:1811.10396 [cs]*, Nov. 2018.

[32] A. Jasem, "Equilibrium Propagation," https://github.com/Abzollo/eqprop, Nov. 2019.

[33] A. Ardakani, C. Condo, and W. J. Gross, "A Multi-Mode Accelerator for Pruned Deep Neural Networks," in *2018 16th IEEE International New Circuits and Systems Conference (NEWCAS)*, Jun. 2018, pp. 352–355.

[34] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A Stochastic Computational Multi-Layer Perceptron with Backward Propagation," *IEEE Transactions on Computers*, vol. 67, no. 9, pp. 1273–1286, Sep. 2018.

[35] H. Mostafa, B. Pedroni, S. Sheik, and G. Cauwenberghs, "Hardware-Efficient Online Learning through Pipelined Truncated-Error Backpropagation in Binary-State Networks," *Frontiers in Neuroscience*, vol. 11, Sep. 2017.

[36] F. Ortega-Zamorano, J. M. Jerez, D. Urda Munoz, R. M. Luque-Baena, and L. Franco, "Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 9, pp. 1840–1850, Sep. 2016.

[37] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 78–89, Feb. 2011.

[38] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74 – 81, 2017.

[39] B. R. Gaines, "Stochastic Computing," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 149–156.

[40] S. S. Tehrani, A. Naderi, G. Kamendje, S. Mannor, and W. J. Gross, "Tracking Forecast Memories in stochastic decoders," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, Apr. 2009, pp. 561–564.

[41] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6.

[42] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, vol. 1. Geneva, Switzerland: Presses Polytech. Univ. Romandes, 2000, pp. 599–602.

[43] B. D. Brown and H. C. Card, "Stochastic neural computation. I. Computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, Sep. 2001.

[44] P.-S. Ting and J. P. Hayes, "Stochastic Logic Realization of Matrix Operations," in *2014 17th Euromicro Conference on Digital System Design*, Aug. 2014, pp. 356–364.

[45] S. C. Smithson, K. Boga, A. Ardakani, B. H. Meyer, and W. J. Gross, "Stochastic Computing Can Improve Upon Digital Spiking Neural Networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct. 2016, pp. 309–314.

[46] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, pp. 1–19, May 2013.