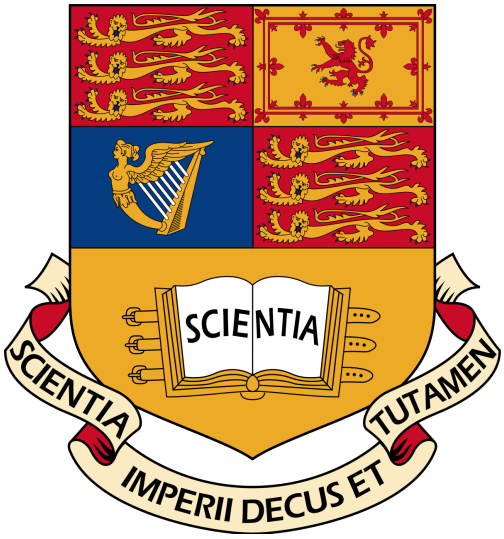
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2018



Project Title:	**Stochastic Computing for Deep Neural Networks**
Student:	**Adamos E. Solomou**
CID:	**00984498**
Course:	**EEE4**
Project Supervisor:	**Prof George A. Constantinides**
Second Marker:	**Dr Christos-Savvas Bouganis**

To the memory of my mother

Acknowledgements

Foremost, I would like to thank my supervisor Prof. George Constantinides for his patience, guidance and advice throughout the course of this project. It has been an honour to be his student.

A special thanks to all the friends that I've made throughout my time at university. I wish them all the best in their future endeavours.

Last but not least, I would like to thank my family for their unconditional love and support. Without them, I would not have had the opportunity to attain an exceptional education at Imperial College London.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence.

ANN Artificial Neural Network.

ASIC Application Specific Integraded Circuit.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DNN Deep Neural Network.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

IoT Internet of Things.

MLP Multilayer Perceptron.

ReLU Rectified Linear Unit.

SC Stochastic Computing.

SGD Stochastic Gradient Descent.

Abstract

Recently, Deep Neural Networks (DNNs) have been revolutionizing machine learning research and applications. These methods have made unprecedented progress in several recognition and detection tasks, achieving accuracy close to, or even better, than human perception. However, their complex topologies require a large amount of computational resources, restricting the widespread deployment of DNNs in mobile devices and embedded systems with limited area and power budget.

This project considers Stochastic Computing (SC) as a novel computing paradigm to provide significantly low hardware footprint with high scalability. In contrast to conventional binary computing, SC operates on random bit-streams where the signal value is encoded by the probability of a bit in the bit-stream being one. Although not compact, this representation allows the implementation of key arithmetic operations such as multiplication and addition with very simple logic. Hence, SC has the potential to implement fully parallel and highly scalable DNNs with significantly reduced hardware footprint.

First, a detailed investigation of the stochastic processing elements employed in DNNs is conducted. Amongst them, a stochastic approximation of the max function is presented, and subsequently used to approximate the rectified linear unit in SC. As addition in SC is performed in a scaled manner, saturation arithmetic architectures are proposed to alleviate large down-scaling parameters that undermine precision in the computations. Combining several building blocks, a scheme is proposed for the implementation of neural network inference in SC. Finally, a modified neuron architecture is presented for training DNNs which are SC compatible and can be implemented efficiently using SC hardware. Experimental results using the MNIST dataset demonstrate that the proposed inference scheme can implement a neural network in SC without increasing the error by more than 2.34%. Finally, the modified SC neuron architecture when employed in training improves classification accuracy on the MNIST dataset by 3.36%.

Chapter 1

Introduction

Humans have longed dreamed of creating machines that think. Today, Artificial Intelligence (AI) is a thriving field with many active research topics and practical applications. People seek to intelligent software to automate routine labor, recognize speech and images, aid medical diagnoses, develop self-driving cars and many more. The capability of an AI system to acquire its own knowledge, by extracting meaningful patterns from raw data is known as machine learning [18]. Deep learning has emerged as a new area of machine learning research that allows a computer to automatically learn complex functions directly from the data by extracting representations at multiple levels of abstraction [10, 27]. Deep Neural Networks (DNNs) have achieved unprecedented success in many machine learning applications such as speech recognition [2] and visual object recognition [44]. Although such tasks are intuitively solved by humans, they originally proved to be the true challenge to artificial intelligence.

Despite their success, when compared with other machine learning techniques, DNNs require more computations due to the deep architecture of the model. Furthermore, developer's ambition for better performance tends to increase the size of the network, leading to longer training times as well as a larger number of computational resources needed for implementation. Currently, researches and practitioners rely on the use of high performance servers to practically implement large scale DNNs. However, such high performance computing clusters incur high power consumption and a large hardware cost, thereby limiting their suitability for low-cost applications such as embedded and wearable IoT devices that require low power consumption and small hardware footprint [41]. These applications increasingly utilise machine learning algorithms to perform fundamental tasks such as natural language processing, speech to text transcription as well as image and video recognition [10, 27] and play nowadays an important role to our everyday life. Hence, to implement such compute-intensive models in resource constraint systems an alternative implementation needs to be found. In some cases, specialised hardware has been designed using Field Programmable Gate Arrays (FPGAs) and Application Specific Integraded Circuits (ASICs) [4, 41, 45, 48]. Nevertheless, there still exists a margin of improvement if the inherent properties and structure of DNNs are further exploited.

This project considers Stochastic Computing (SC) as a low-cost alternative to conventional binary computing. This computing paradigm operates on random bit-streams, where the signal value is encoded by the probability of an arbitrary bit in the sequence being one. Such a representation is particularly attractive as it enables very low-cost implementations of arithmetic operations using simple logic circuits [3]. For example, multiplication and addition can be performed using an AND gate and a multiplexer (MUX) respectively. Stochastic computing offers very low computation hardware area, high degree of error tolerance and the capability to trade-off computation time and

accuracy without any hardware changes [6]. It therefore has the potential to implement DNNs with significantly reduced hardware footprint and low power consumption. On the other hand, SC has several disadvantages including accuracy issues due to the inherent variance in estimating the probability represented by the stochastic sequence. Furthermore, an increase in the precision of a stochastic computation requires an exponential increase in the length of the bit-stream [3], thereby increasing the overall computation time. In general, stochastic arithmetic will be more suitable for an application where the accuracy requirements in the individual computations are relatively low.

Artificial Neural Networks (ANNs) are characterised by a natural error resiliency [20], which distinguishes them from other machine learning methods that require precise computations and exact number representations. Furthermore, the authors of [5] and [35] demonstrate that the addition of noise during the training of a neural network improves the network's performance. One can intuitively reinforce this error resilience by considering the variant of the gradient descent algorithm, the stochastic gradient descent which is widely used for training DNNs. The method of stochastic gradient descent provides an unbiased estimate of the true gradient based on a batch of training samples. Nevertheless, the randomization seems to benefit the minimization of the objective function as it allows to escape local minima on the loss surface. Hence, ANNs could tolerate less accurate individual numerical computations without significant degradation in their performance. This makes SC an attractive candidate for implementing DNNs as the computations in stochastic arithmetic are characterised by some degree of uncertainty and at the same time are computationally cheap and simple. In fact, the use of stochastic arithmetic is in a sense, similar to the use of the stochastic gradient descent. A representation of a quantity in stochastic computing provides an unbiased estimate of the true value, a property similar to that of the stochastic gradient descent.

The remainder of this report is structured as follows. Chapter 2 presents background information related to the project, including related work and a brief introduction to DNNs followed by fundamental principles of general computer arithmetic as well as stochastic arithmetic. Project specifications are captured in Chapter 3. A number of stochastic processing elements employed in ANNs are presented and analysed in Chapter 4. Thereinafter, Chapter 5 is concerned with the implementation of neural network inference in stochastic computing, whereas Chapter 6 proposes alternative neuron architectures for training SC compatible neural networks. Chapter 8 reports the experimental results, followed by a critical appraisal of the work presented in this project in Chapter 9. The report is concluded in Chapter 10.

Chapter 2

Background

This part of the report serves as the literature review of the project. Related work is presented, followed by a discussion on deep feedforward neural networks. Thereinafter, fundamental principles of digital and stochastic arithmetic are presented.

2.1 Related Work

Deep learning principles have been known for many years. However, it wasn't until the start of the 21st century were advances in hardware technology enabled the development of capable deep learning models. Even today, the training of large scale DNNs is often constraint by the available computational resources.

CPU platforms are in general unable to provide enough computation capacity for training large scale neural networks. Nowadays, GPU platforms are the default choice for neural network training due to the high computation capacity and easy to use development frameworks [19, 23]. Krizhevsky et al. [26] and Facebook AI Group [47] train *AlexNet*, a Convolutional Neural Network (CNN), on multiple GPUs. The authors of [29] study the memory efficiency of various CNN layers and reveal the performance implication from both data layouts and memory access patterns. Finally, [32, 37] present a GPU based implementation of a Cellular Neural Network, a locally connected recurrent neural network which is widely used in image processing applications.

FPGA based neural network acceleration is an emerging research topic as well. FPGAs can implement high parallelism and potentially surpass GPU in speed and energy efficiency [19]. A main challenge in FPGA based acceleration design is the lack of of development frameworks such as TensorFlow and Caffe. To aid the development of deep learning models on FPGAs, Venieris and Bouganis [46] propose a framework for mapping CNNs on FPGAs. Furthermore, the authors of [34] and [48] propose an FPGA base accelerator to leverage the sources of parallelism in order to achieve an efficient implementation of a deep convolutional neural network. Finally, [49] presents a reconfigurable framework for training CNNs. While viable, the FPGA and GPU based implementations still exhibit a large margin of improvement, mainly because these are general purpose computing devices not specifically optimized for executing DNNs.

In addition to such acceleration techniques, DNNs can significantly benefit from the SC technology which allows implementation of complex functions with very simple logic. Stochastic computing has the potential to implement DNNs with significantly reduced hardware footprint when compared to a fixed or floating-point implementation. There have been prior attempts to implement ANNs using stochastic computing. The authors in [22] utilise SC to implement a radial basis function (RBF) neural network significantly reducing the required hardware. However, RBF

neural networks are no longer widely used in deep learning applications as the RBF unit saturates to zero for most of its inputs, making gradient-based optimization challenging. [24] presents a neuron design in SC for DNNs and exploits the energy-accuracy trade-off. Reconfigurable large scale deep learning systems based on SC were designed in [40]. Furthermore, the authors in [30] and [41] present stochastic computing hardware designs for the implementation of CNNs. In [41], focus is given on weight storage schemes and optimization techniques to reduce area and power consumption of weight storage in hardware. On the other hand, [30] proposes a structure optimization method for a general CNN architecture aiming to minimize area and power consumption while maintaining adequate network accuracy.

In summary, the aforementioned works have proposed certain neuron designs using SC in order to satisfy the computing limitations in resource-constraint applications such as embedded systems. However, they only consider the implementation of neural network inference using SC hardware. Moreover, only a certain activation, namely the hyperbolic function is considered whose usage has reduced significantly since the introduction of the rectified linear unit. Despite previous work, there still lacks a detailed investigation regarding the scaling scheme used to implement a neural network using SC hardware. Finally, there is no existing work that investigates comprehensively how stochastic computing can be incorporated during the training stage of a DNN and how this can affect the performance of the neural network on the recognition task.

2.2 Deep Feedforward Networks

This section presents a detailed discussion on deep feedforward neural networks. Starting from the fundamental unit of neural networks, the perceptron, followed by an overview of the feedforward networks and an analysis of their architecture. Finally, backpropagation, the most common algorithm for computing the gradients during training is briefly presented.

2.2.1 The Perceptron

Frank Rosenblatt in 1958 invented the perceptron learning algorithm, consisting of a single neuron which at that time worked sufficiently well as a binary classifier [42]. The perceptron would perform a weighted sum of its inputs, add a bias and pass the result through an activation function, like the Heaviside function. In that case, the sign of the output could be used to predict the category in which the test sample belongs to (i.e. classify the test sample). Further research revealed that the perceptron was only able to represent linear functions. A classic example where the perceptron fails to replicate a non-linear function is the XOR gate [18]. Later, it was found that combining and chaining perceptrons in the form of layers could replicate non-linear functions, creating eventually what is now known as the multilayer perceptron [33].

2.2.2 Overview of Feedforward Networks

Multilayer Perceptrons (MLPs) also known as deep feedforward networks are the quintessential deep learning models. As any other machine learning model the objective of a feedforward network is to approximate some unknown function f^*, commonly referred to as the target function. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the set of parameters $\boldsymbol{\theta}$ that yield the best function approximation [18]. These networks are of extreme importance as they form the basis of any other deep learning architecture such as convolutional or recurrent neural networks.

A feedforward network is associated with a directed acyclic graph describing how the neurons are combined with each other [18]. Figure 2.1 illustrates such a graph for a fully connected feedforward

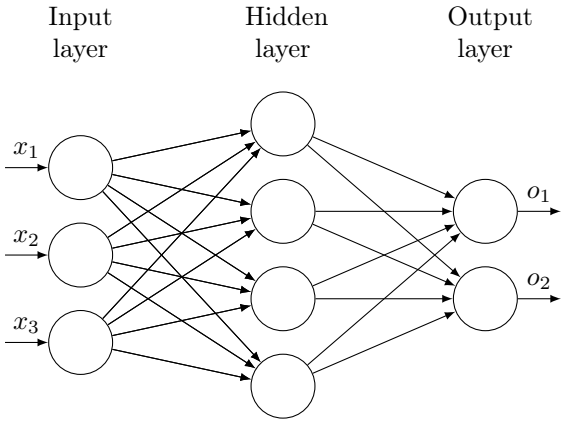


Figure 2.1: Directed acyclic graph of a feedforward network. The network has 3 inputs, a single hidden layer with 4 units and two output units.

neural network with a single hidden layer. The network has three input units, four hidden units and two output units. For this network, the relationship $\boldsymbol{y} = f(\boldsymbol{x}) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$ represents the chained structure. $f^{(1)}$ represents the first layer of the network and $f^{(2)}$ the second and output layer of the network. The overall length of the chain defines the depth of the model. Similarly, the dimensionality of each layer (i.e. the number of neurons) determines the width of the model. During training, the aim is to choose model parameters such that $f(\boldsymbol{x}) \simeq f^*(\boldsymbol{x})$. The training data, $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), .., (\boldsymbol{x}_n, y_n)\}$, essentially provide noisy approximations of the target function f^* evaluated at different points [18]. The training data specify directly what the behaviour of the output layer should be to produce an output that is close to y_i. On the other hand, the behaviour of the rest of the layers is not directly specified by the training data [18]. Their behaviour is entirely specified by the learning algorithm which must decide how to utilise those layers to achieve an implementation that best approximates f^*. Since the internal layers are not directly influenced by the training data are called hidden layers.

One way to understand feedforward networks is to consider how they overcome the limitations of linear models [18]. As already mentioned, linear models, such as the perceptron or linear regression, are limited in the sense that they can only represent linear functions. A possible way to extend linear models, is to apply a non-linear transformation ϕ to the input data $\boldsymbol{x}$. A linear model can then process the transformed features $\phi(\boldsymbol{x})$. This is similar to the kernel trick. A natural question that then arises, is how to choose a suitable non-linear transformation ϕ. An option is to manually engineer ϕ, however this requires significant human effort and expertise knowledge. The approach of deep learning is to learn ϕ. Consider a network with a single hidden layer defined by $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$. The parameters $\boldsymbol{\theta}$ are used to learn ϕ and parameters $\boldsymbol{w}$ to map $\phi(\boldsymbol{x})$ to the output [18]. This is indeed a powerful approach. However, the non-linearity causes most loss functions to become non-convex, in contrast with linear models which can be fit either through convex optimization or in closed form. Nevertheless, the benefits of this approach usually outperform this drawback.

2.2.3 Gradient-Based Learning

The non-convexity of the loss function in the context of neural networks, drives practitioners to use iterative optimization algorithms to minimize the loss function, i.e. train the neural network. Usually, variants of the gradient method are used. That is first-order, gradient-based minimization

algorithms. This implies that the optimization will most likely yield a local minimum on the loss surface rather than a global optimum as in the case of linear models where the loss function is convex. In practice, the gradient descent algorithm has proved to be slow or unreliable [18]. Hence, the stochastic gradient descent (SGD) algorithm is usually used in practice which is an extension of the original method. The SGD provides an unbiased estimate of the true gradient, calculated using a minibatch of samples, $\mathcal{B} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_m\}$, drawn uniformly from the the training set [18]. Generally, the SGD has no convergence guarantees and depends on the initialisation of the model's parameters such as weights and bias coefficients [18]. Prior training, it is important to initialise the weights of the network to small non-zero random values. As with any other machine learning model, when applying gradient-based learning key decisions are the selection of the loss function and the type of the output unit. These are briefly discussed below.

Loss Function

The loss function quantifies how close the predictions of the model are to the true labels. Ideally, this should be zero and in that case the model perfectly classifies all the samples. In practice, a regularization term is usually added to the loss function in order to avoid overfitting the training data. A typical approach to train many modern networks is to apply maximum likelihood. This implies that the loss function is the negative log-likelihood which is equivalent to the cross entropy between the training data and the model distribution [18]. For the purpose of neural network inference, the loss function is only computed once at the end of the forward propagation. On the other hand, during the training phase of a network the loss function needs to be computed in every iteration of the training algorithm. Nevertheless, for the purpose of this project, it is assumed, without loss of generality, that the loss function can be computed in floating-point arithmetic. Thus, it will not be further considered in terms of computing it in stochastic arithmetic. Other common loss functions are the mean squared error and the mean absolute error. However, these often lead to poor generalization results when used with gradient-based learning [18].

Output Units

The choice of the output unit dictates how the output will be represented, which in turn determines the form of the cross-entropy function [18]. The selection of the output unit is therefore driven by the task of the network (e.g. Classification/Regression). For binary classification tasks, the sigmoid unit can be combined with maximum-likelihood to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$ [18]. The sigmoid function is given by,

$$\sigma(x) = \frac{1}{1 + exp(-x)} \tag{2.1}$$

Note that $\sigma(x) \in (0, -1)$, hence it satisfies the range of the Bernoulli distribution. The sigmoid output unit is given by,

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + \boldsymbol{b}) \tag{2.2}$$

where $\hat{y}$ denotes the predicted probability over the output and the class of the sample can be predicted as sign($\hat{y}$). Extending the sigmoid unit to multi-class classification problems, the softmax unit can be used to represent a probability distribution over L distinct classes. In general, any output unit can be used as a hidden unit as well, although rarely a softmax function will be used as a hidden unit.

2.2.4 Hidden Units

In general, a hidden unit takes a vector of inputs $\boldsymbol{x}$, computes a weighted sum, adds an offset and passes the result through a non-linear function. This can be expressed as $g(z) = \boldsymbol{w}^T\boldsymbol{x} + b$, and has the same form as the perceptron except from the fact that the function $g(z)$ can be of any form. Expressing the computation as $g(\boldsymbol{z}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{W}$ is the weight matrix and applying the function $g(\boldsymbol{z})$ element-wise generalises the previous computation to multiple hidden units in a layer of a fully connected network. The key characteristic that distinguishes hidden units is the form of the activation function $g(\boldsymbol{z})$.

There is no specific guideline for choosing hidden units, although rectified linear units (ReLUs) are an excellent default choice since they are simple and easy to optimise [18]. ReLUs use the activation function $g(z) = max\{0, z\}$. Other well established hidden units use the logistic sigmoid activation function $g(z) = \sigma(z)$ as defined by equation (2.1) or the hyperbolic tangent activation function $g(z) = tanh(z)$. Unlike rectified linear units, sigmoidal units saturate across most of their domain making gradient-based learning very difficult [18]. In this project, an implementation in stochastic computing for both the ReLU as well as the hyperbolic tangent is given.

2.2.5 DNN Architecture Overview

Summarising, a deep feedforward neural network consists of a chain of fully connected layers. Neurons in each layer calculate a weighted sum of the incoming feature values and the result is sent to an activation function such as the ReLU or the hyperbolic tangent. Finally, the output layer computes the output of the network and the loss function is calculated indicating by how much the predicted output deviates from the true output. In this project, focus is given on a SC-based implementation of the basic operations of a feedforward network, including inner product and activation functions.

2.2.6 Back-Propagation for Gradient Computation

The goal of any training algorithm such as the stochastic gradient descent is to minimize the value of the loss function by adjusting the weights and biases of the neurons. Gradient based learning is therefore an iterative process that consists of three parts:

- Forward Propagation: Given an input $\boldsymbol{x}$, the output $\hat{y}$ of the model is calculated and the loss function $\mathcal{J}$ is computed.
- Backward Propagation: Propagate the information from the loss function at the output backwards through the network in order to compute the gradient of $\mathcal{J}$ with respect to model parameters at each layer.
- Update Rule: Update weights and biases at each layer based on the update rule of the minimisation algorithm.

When training starts, weight values are usually initialised to small random values. Then, a training example is presented to the network and a forward propagation is performed producing the network's estimate for that input. In addition, the error is computed based on the designed loss function. At this point, if the network was a single layer model it would be possible to apply the update rule directly to optimise the weights. However, for a multi-layer network this is not feasible as one needs to know the error at the output of each internal neuron which is unknown. Hence, another technique must be applied that utilizes the network's output to compute the derivative of the loss function with respect to each internal neuron parameter.

Let $w_{ij}^{(l)}$ denote the weight connecting node i to node j during the lth iteration of the training algorithm. The update rule using the gradient method is given by

$$w_{ij}^{(l+1)} = w_{ij}^{(l)} - \eta \frac{\partial \mathcal{J}}{\partial w_{ij}^{(l)}}$$

where $\eta > 0$ is the learning rate. The above equation implies that at each iteration of the training algorithm, the weights are updated by taking a step along the direction of steepest descent of the loss function $\mathcal{J}$ with respect to the weight $w_{ij}^{(l)}$. Since η is a constant, the only parameter that needs to be determined is $\frac{\partial \mathcal{J}}{\partial w_{ij}^{(l)}}$.

Rumelhart, in 1986 [43] used the idea of the chain rule of calculus to express the partial derivative of $\mathcal{J}$ with respect to any given weight $w_{ij}^{(l)}$ at an arbitrary hidden layer as

$$\frac{\partial \mathcal{J}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{J}}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial net_j^{(l)}} \frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}}$$

where $o_j^{(l)}$ is the output of neuron j and $net_j^{(l)}$ is the weighted sum of the inputs of neuron j during the lth training epoch. That is , $o_j^{(l)} = \phi(net_j^{(l)})$, where ϕ is the activation function. This approach, starts from the value of the loss function value at the output and progressively propagates information backwards through the network to calculate the derivative of $\mathcal{J}$ with respect to any weight and apply the update rule. The same procedure also applies for the biases, allowing the entire network to be trained.

2.3 Digital Arithmetic Principles

In this section, basic number representation systems and algorithms used in digital arithmetic are briefly reviewed. The treatment is very concise and the main objective is to present general principles of digital arithmetic that will allow to compare and contrast features of general computing to stochastic computing throughout the remaining of the report.

Principles of computer arithmetic (a.k.a digital arithmetic) trace back to 17th century when Gottfried Leibniz formally developed the binary system. Since then, digital arithmetic has tremendously evolved and nowadays it plays an important role in the design of general-purpose digital processors as well as embedded systems for signal processing, graphics and communications. Digital arithmetic is concerned with the study of number representations, algorithms for operations on numbers as well as implementation of arithmetic units in hardware [11]. An arithmetic unit is a system that performs operations on numbers, such as addition, multiplication, division etc. Most commonly, these numbers are [11]:

1. Fixed-point numbers
   - Integers $I = \{-N, ..., N\}$
   - Rational numbers of the form $x = a/2^f$, where $a \in I$ and f a positive integer
2. Floating-point numbers $x \times b^E$, where x is a rational number, b the integer base and E the integer exponent. Floating-point numbers approximate real numbers and facilitate computations over a wide dynamic range.

2.3.1 Fixed-Point Number Representation Systems

To operate on fixed-point numbers by means of an algorithm, a specific number representation is required. Generally, in a digital representation such a number is represented by an ordered n-tuple[1] called the *digit-vector*. Each of the elements of the digit-vector is called a *digit* and the number of digits is called the *precision* of the representation [11].

Without loss of generality, a non-negative integer x is represented by the digit-vector

$$X = (X_{n-1}, X_{n-2}, ..., X_1, X_0) \tag{2.3}$$

A number system representing x is associated with several elements such as [11]:

1. The precision of the digit-vector X
2. A set of values $\{D_i\}$ for the digits $\{X_i\}$. For example, the binary set $\{0, 1\}$ is the digit set for the conventional binary number system.
3. A rule of interpretation that defines the mapping between the set of digit-vector values and the set of integers.

Depending on the selection of the aforementioned elements, different number systems can be devised. Clearly, the set of integers that can be represented by a digit-vector with n digits is finite, with at most $\prod_{i-0}^{n-1} |D_i|$ different elements[2] [11].

General purpose computing architectures typically use a *weighted* number system [11]. For such a system, the representation mapping is

$$x = \sum_{i=0}^{n-1} W_i X_i \tag{2.4}$$

where $W = (W_{n-1}, ..., W_0)$ is the weight vector.

A *radix number system* is a weighted number system in which the weight-vector is related to the radix-vector $R = (R_{n-1}, ..., R_0)$ as follows

$$W_0 = 1, \quad W_i = \prod_{j=0}^{i-1} R_j \tag{2.5}$$

There exist two main classes of radix-number systems: 1) Fixed-radix systems and 2) Mixed-radix systems [11]. In a *fixed-radix* system all elements of the radix vector are fixed to the same value r. Thus, the weight vector is $W = (r^{n-1}, r^{n-2}, ..., r, 1)$ and the integer x is related to the digit vector by

$$x = \sum_{i=0}^{n-1} r^i X_i \tag{2.6}$$

On the other hand, in a mixed-radix system the elements of the radix vector are different [11]. Frequently, powers of two are used as radices. Although humans are more familiar with the decimal number system, the simplicity of many arithmetic algorithms in the binary number system motivates the use of alternative number systems.

Depending on the set of digit values $\{D_i\}$, the radix number systems are classified into canonical and non-canonical systems. In a canonical system the set of values $\{D_i\}$ is $\{0, 1, .., R_{i-1}\}$ with

[1] An n-tuple is a sequence (or ordered list) of n elements, where n is a non-negative integer

[2] $|D_i|$ denotes the cardinality of the set $\{D_i\}$, i.e. the number of elements in the set

$|D_i| = R_i$ [11]. In a non-canonical system the set of digit values is not canonical. For example, in the binary number system the canonical set of digits is $\{0, 1\}$. On the other hand, the digit set $\{-1, 0, 1, \}$ is non-canonical. For a system with n radix-r digits, the range of values of x that can be represented is

$$0 \leq x \leq r^n - 1$$

By far the most commonly used systems are those with fixed positive radix r and a canonical set of digit values. Such systems are termed *radix-r conventional number systems* and perhaps the most widely used system is the conventional binary system, i.e. radix-2 conventional number system.

While so far the discussion was concerned with the representation of non-negative integers, such concepts can be extended to represent signed integers as well. Two of the most common representations are [11]: 1) The sign-and-magnitude (SM) representation and 2) The true-and-complement (TC) representation.

In the SM system, a signed integer is represented by a pair (x_s, x_m), where x_s is the *sign* and x_m is the *magnitude* [11]. Clearly, only two sign values exist (+,−) and they are represented by a binary variable. Traditionally, $x_s = 0$ represents a positive sign and $x_s = 1$ represents a negative sign. The magnitude of the number can be represented by any system for the representation of positive integers. For a radix-r conventional number system, where n digits are used in the representation of the magnitude, the range of signed integers is

$$0 \leq x_m \leq r^n - 1$$

On the contrary, in the TC system there is no distinction between the representation of the sign and the magnitude. Instead, the whole signed number is represented by a positive integer, thus an additional mapping between signed and positive integers needs to be defined [11]. This process is illustrated in Figure 2.2. Effectively, a signed integer x is represented by means of a positive integer x_R, which in turn is represented by a digit-vector X. Although detailed consideration of the mapping between signed and unsigned integers in the context of a TC system is in general out of the scope of this project, a noteworthy example, is the mapping illustrated in Table 2.1 which defines the so called *two's complement system*. For this system, the range of signed integers is

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

Finally, a fixed-point representation of a non-integer number $x = x_{INT} + x_{FR}$ consists of integer and fraction components represented by m and f digits respectively. The following notation is commonly used

$$X = (X_{(m-1)}, ..., X_1, X_0.X_{-1}, ..., X_{-f}) \tag{2.7}$$

Thereby,

$$x = \sum_{i=-f}^{m-1} r^i X_i \tag{2.8}$$

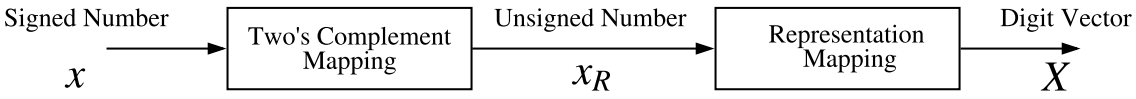


Figure 2.2: Mapping process in a TC representation system

Table 2.1: Mapping in the two's complement system

	x	x_R
True forms ($x_R = x$)	0	0
	1	1
	2	2
	-	-
	-	-
	-	-
	$2^{n-1}-1$	$2^{n-1}-1$
Complement Forms forms ($x_R = 2^n - \|x\|$)	-2^{n-1}	2^{n-1}
	$-(2^{n-1}-1)$	$2^{n-1}+1$
	-	-
	-	-
	-	-
	-2	2^n-2
	-1	2^n-1

Apart from the study of number representation systems, a major concern of digital arithmetic is the development of arithmetic units that perform operations on numbers. Throughout the years, extensive research has been devoted to development of such processing units. Ercegovac and Lang [11, 12, 13, 14, 15] present a comprehensive analysis of algorithms and implementations for addition and subtraction (of two and more than two operands) as well as multiplication and division of fixed-point numbers. While, the analysis and design of algorithms for operations on fixed-point radix-2 binary numbers is outside the scope of this project, it is useful to have such a background in place especially when it comes to compare those units to the corresponding arithmetic units employed in stochastic arithmetic.

As a case study, the multiplication of positive numbers is considered. An $n \times m$ multiplier has two inputs: $x = (x_{n-1}, ..., x_0)$ in the range $[0, 2^n - 1]$ and $y = (y_{m-1}, ..., y_0)$ in the range $[0, 2^m - 1]$. The product $p = (p_{n+m}, ..., p_0)$ is in the range $[0, 2^{n+m} - 1]$. If the operands and the result are represented in radix 2, then the opration is defined as

$$p = \sum_{i=0}^{m-1} xy_i 2^i \tag{2.9}$$

which indicates that to compute the product, the shifted multiples $xy_i 2^i$ are added [9]. Since $y_i \in \{0, 1\}$, the multiples xy_i are either 0 or x and each can be computed by means of an AND gate. Generally, multiplication methods consist of three major stages: 1) Partial product generation, 2) Reduction of partial products to two digit-vectors and 3) A final adder which computes the product. Further details of each stage can be found in [9]. Figure 2.3 illustrates the implementation of a 5×5 radix-2 multiplier using a linear array of carry-ripple multipliers. The primitive modules, shown in Figure 2.3a, incorporate the AND gates needed to compute the partial products. Clearly, when compared to the multiplier unit in stochastic computing, the complexity of the architecture shown in Figure 2.3 is significantly larger. The fixed-point multiplier obviously occupies significantly larger hardware area than a single AND gate. Finally, in contrast to stochastic computing, an increase in the precision of the fixed-point computation requires to modify the hardware infrastructure of the multiplier to introduce additional computational units.

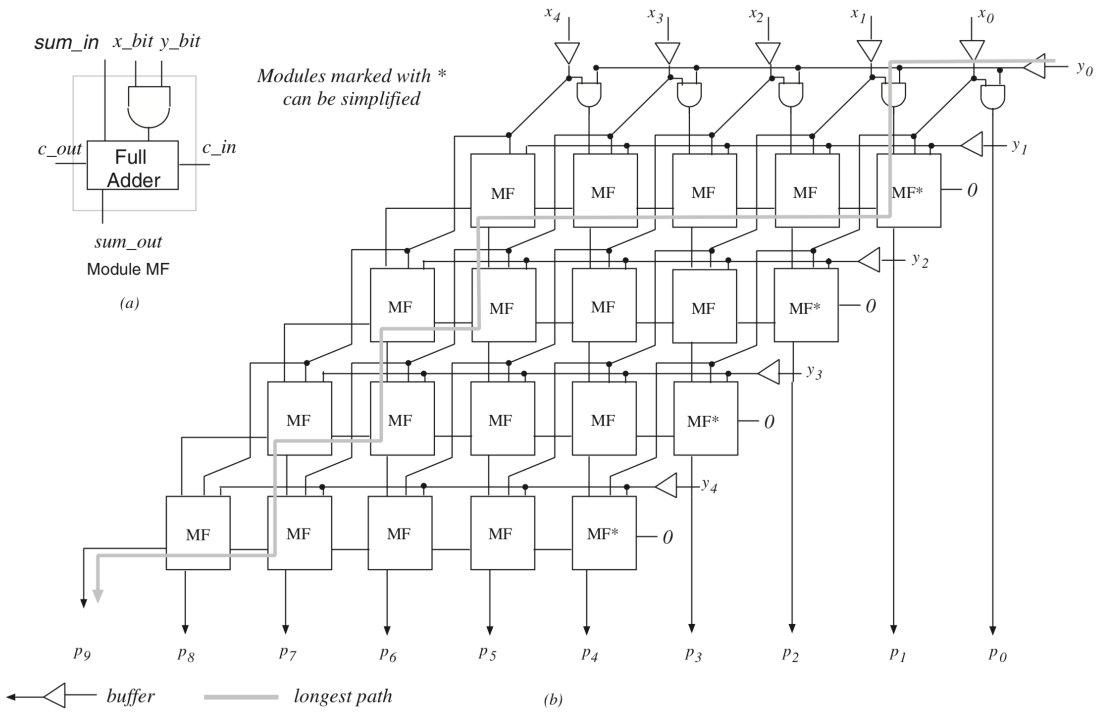


Figure 2.3: Implementation of a 5×5 multiplier. (a) Primitive Modules (b) Network. (Adapted from Dinechin et al. [9])

2.3.2 Floating-Point Representation

Many scientific and engineering applications require computations with real numbers. While a fixed-point representation can be used, in many cases the range of this representation does not correspond to the range required by the applications, resulting in overflows and underflows[3] [16]. Therefore, in order to avoid overflow fixed-point numbers must be appropriately scaled. Such a process can sometimes be complicated, hence an alternative, standard solution for such a problem is the use of floating-point representations.

In general, the representation of a floating-point number x consists of two components, the *mantissa* M_x^* (also known as the *significant*) and the *exponent* E_x, such that

$$x = M_x^* \times b^{E_x} \tag{2.10}$$

where b is called the *base* and is a constant [16]. The sign of the number is determined by the sign of the mantissa, and the exponent is a signed integer.

Nowadays, the sign-and-magnitude representation is typically used to represent the signed significant. In that case, a floating-point number is represented by the set of parameters (S_x, M_x, E_x), such that

$$x = (-1)^{S_x} \times M_x \times b^{E_x} \tag{2.11}$$

where $S_x \in \{0, 1\}$ is the sign bit and M_x denotes the magnitude of the mantissa [16].

The main advantage of a floating-point representation compared to a fixed-point representation, is the larger dynamic range of the former. According to Ercegovac and Lang [16], the dynamic

[3] An overflow exists whenever the magnitude of the result of addition or subtraction exceeds the largest representable magnitude. If this occurs, the result is incorrect. In the two's complement representation, overflow results in a "wrap-around" phenomenon.

range is defined as the ratio between the largest and smallest (non-zero and positive) number that can be represented. For a fixed-point representation using n radix-r digits for the magnitude, the dynamic range is [16]

$$DR_{fxpt} = r^n - 1 \tag{2.12}$$

On the other hand, for the floating point representation, the dynamic range is [16]

$$DR_{flpt} = \frac{M_{max} b^{E_{max}}}{M_{min} b^{E_{min}}} \tag{2.13}$$

To compare the two, assume that the n digits of the floating-point representation are partitioned so that m digits are used for the significant, $n - m$ digits for the exponent and $b = r$. Then,

$$DR_{flpt} = (r^m - 1) \times r^{(r^{n-m}-1)} \tag{2.14}$$

To illustrate the concept, consider the case where $n = 32$, $m = 24$, $r = 2$. Then,

$$DR_{fxpt} = 2^{32} - 1 \approx 4.3 \times 10^9$$

$$DR_{flpt} = (2^{24} - 1) \times 2^{(2^8-1)} \approx 9.7 \times 10^{83}$$

Since floating-point numbers can effectively approximate real numbers without overflow issues, a natural question that could then arise is why fixed-point representation systems are still studied and used? A floating-point representation does indeed have some disadvantages compared to a fixed-point representation such as less precision, roundoff errors and a more complex hardware implementation.

The precision of a representation corresponds to the number of digits of the significant [16]. As in the floating-point representation the total number of digits is partitioned between the significant and the exponent, then for the same number of digits, the floating-point representation has a smaller precision than the fixed-point representation.

Furthermore, in a specific floating-point system, real numbers that are not exactly represented in the system either fall outside the range of the representation (overflow and underflow) or they are represented by floating-point numbers that have a value that approximates the real number. Such an approximation process is termed *roundoff* and it produces a roundoff error.

Finally, an additional disadvantage of a floating-point representation is that the logic circuits used to realise floating-point operations are in general more complicated than the corresponding circuits of fixed-point hardware. Thereby, the hardware footprint of floating-point units is larger than that of fixed-point hardware incurring a higher power consumption and longer execution time.

2.4 Stochastic Computing

In the 1960's a novel computing paradigm, namely Stochastic Computing, that made use of random pulse sequences as information carriers was introduced by Brian Gaines [17] and Poppelbaum et al. [36]. Further research revealed that the use of random pulse sequences to represent numerical quantities allows fundamental arithmetic operations such as multiplication and addition to be performed by means of very simple logic. Notwithstanding this advantage and also its inherent error tolerance, SC was later on seen impractical due to its relatively low accuracy. In fact, following the aforementioned work by Gaines and Poppelbaum, a few general-purpose stochastic computers were built around that time, and they uncovered several deficiencies of the technology [3].

Binary computing devices have now become smaller, faster and more reliable. Furthermore, modern technology is capable of fabricating billions of computational units on a single silicon chip with very low cost. While this may seem to make stochastic processing irrelevant, there exists a class of systems which requires virtually an unlimited number of computational elements and is prepared to sacrifice accuracy of the computations in exchange for hardware simplicity [6]. ANNs are massively parallel systems, characterised by a certain tolerance for much less accurate individual computations [6]. ANNs involve thousands of multiply and add operations, thus they can tremendously benefit from a technology, such as stochastic computing, that allows arithmetic operations to be implemented with very simple logic.

2.4.1 Fundamental Principles

Stochastic computing relies on probability theory, where a probability number is represented by a bit-stream of chosen length and its value is determined by the probability of an arbitrary bit in the bit-stream being one. For example, a stochastic bit-stream containing 75% of ones and 25% of zeros represents the number $p = 0.75$, reflecting the fact that the probability of observing a one in an arbitrary bit position is 0.75. Clearly, when compared to a binary radix representation, the stochastic representation is not very compact. However, it leads to very low-complexity arithmetic units which was a primary concern in the past. For example, multiplication in stochastic arithmetic can be performed by a single AND gate. Consider two input stochastic streams that are logically ANDed and assume that the probability of observing a one in each stream is p_1 and p_2 respectively. Then, assuming that the inputs are suitably uncorrelated or independent, the probability of any bit in the output of the AND gate being a one is $p_1 \times p_2$. Figure 2.4 illustrates this operation.

Another attractive feature of SC is its inherent error tolerance. The probability p depends on the ratio of ones to the length of the stochastic bit-stream and not on their exact position. For example, $(1, 1, 0, 0)$, $(1, 0, 0, 1)$ as well as $(1, 0, 0, 1, 1, 1, 0, 0)$ are all possible and valid representations of $p = 0.5$. This type of representation offers a high degree of error tolerance, as a single bit-flip in a long sequence will have a small impact on the stochastic number that is represented. Consider for example a bit-flip in the input S_2 of the AND gate in Figure 2.4. This would change the value represented from 6/8 to 5/8 or 7/8. That is an error of 1/8. On the other hand, a single bit-flip in a conventional radix-2 representation can cause a significant error especially if it affects a high-order bit [3]. For example, a single bit-flip in the fixed-point representation of S_2, 0.11, would result a much larger error since 0.01 represents 0.25. So that is an error of 4/8. This property of SC is exactly due to the fact that all of the bits in a stochastic sequence are equally weighted.

Despite these attractive features, SC has some disadvantages that have limited its practical application. Specifically, stochastic arithmetic has an inherent variance in the computations [6].

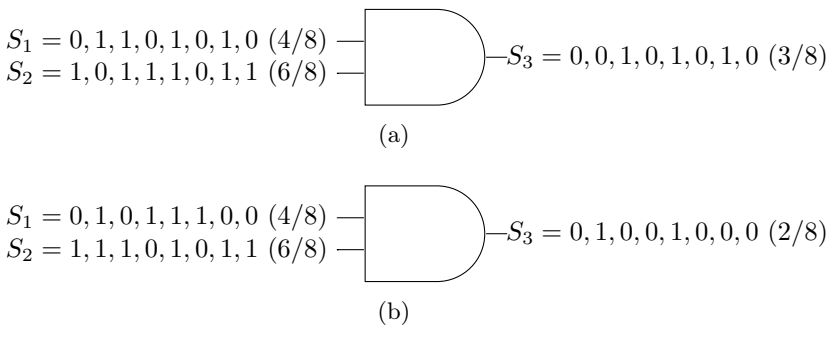


Figure 2.4: Multiplication in stochastic arithmetic: (a) Exact and (b) Approximate computation

This arises from the fact that the stochastic representation of a number is not unique. Specifically, for a stochastic sequence of length n with p ones and $n-p$ zeros, where $p \in [0, n]$, there are $\binom{n}{p}$ possible representations of the value p/n [3]. This property can lead to inaccuracy in computations. Consider the multiplication operation illustrated in Figure 2.4. The stochastic representation of $S_1 = 4/8$ and $S_2 = 6/8$ in Figure 2.4a yields the exact result. On the other hand, the stochastic representation in Figure 2.4b yields a wrong result. This illustrates the kind of inaccuracies that arise in SC due to existence of alternative SC representations of the same value. Furthermore, an n-bit sequence can exactly represent only the numbers that belong to the set $\{0/n, 1/n, 2/n, ..., n/n\}$, which is a very small subset of the real numbers in $[0, 1]$. Therefore, the set of representable values is limited by the length of the stochastic sequence.

Another downside of SC is that more accurate computations require an exponential increase in the number of digits, causing in turn an exponential increase in the number of clock cycles needed to accomplish a computation [6]. Informally, one can define precision of a quantity as the number of bits required to represent that quantity [3]. Having m bits of precision, 2^m different values can be represented. Applying this idea to SC, the set of real numbers within $[0, 1]$ represented with 4-bits if precision reduces to the set $\{1/16, 2/16, ..., 16/16\}$, and their stochastic representation requires bit-streams of length 16. To increase the precision of a stochastic number from 4 to 5 bits requires twice as many stochastic bits, that is a bit-stream of length 32. This exponential increase in the bit-stream length with respect to precision requirements is what reduces the speed of computation in SC as more clock cycles are required to complete a certain computation. Note however, that an increase in precision in SC does not require additional computational resources. The same AND gate can perform the multiply operation with higher precision. The only downside is that the latency in the computation will be larger. In contrast, in fixed-point arithmetic an increase in precision causes an increase in the resources required.

In general, inaccuracy in SC has several sources such as random fluctuations in the stochastic number representation or similarities (i.e. correlations) between numbers that are involved in a computation [3]. According to [21], two n-bit stochastic sequences $S_1 = (S_1(1), S_1(2), ..., S_1(n))$ and $S_2 = (S_2(1), S_2(2), ..., S_2(n))$ are uncorrelated or independent if and only if

$$\sum_{i=1}^{n} S_1(i)S_2(i) = \frac{\sum_{i=1}^{n} S_1(i) \times \sum_{i=1}^{n} S_2(i)}{n} \tag{2.15}$$

In any other case, the two sequences are correlated. To illustrate how correlation between two stochastic bit-streams can cause inaccuracy in the computation consider the following example. Let $S_1 = (0, 0, 0, 0, 1, 1, 1, 1)$ and $S_2 = (1, 0, 1, 0, 1, 0, 1, 0)$ both represent the value $p = 0.5$. According to the definition in equation 2.15, the two sequences are uncorrelated and their product $S_1 \times S_2$ obtained by the element-wise AND operation is $(0, 0, 0, 0, 1, 0, 1, 0) = 0.25$, which is the correct result. On the other hand, $S_2 = S_3 = (0, 1, 0, 1, 0, 1, 0, 1)$ are correlated and their product $(0, 0, 0, 0, 0, 0, 0, 0) = 0$ is clearly wrong. To mitigate such phenomena, it is essential to generate stochastic numbers that are suitably uncorrelated or independent.

In summary, SC offers several advantages over other computing techniques [6]: 1) Very low hardware footprint 2) High degree error tolerance 3) Simple hardware implementations that can allow very high clock rates and 4) Capability to trade-off latency in computation with accuracy without any hardware modifications. Accuracy issues have limited the use of SC over the past. Nevertheless, as already discussed ANNs are characterised by a tolerance for much less accurate individual computations. Hence, SC has the potential to facilitate fully parallel, low-cost hardware implementations of ANNs.

2.4.2 Linear Mapping from Analog Quantities to Probabilities

This section considers the two most popular representations of stochastic numbers. These are the unipolar and bipolar representations as defined by Gaines in [17] which represent numbers in the range $[0, 1]$ and $[-1, 1]$ respectively.

Unipolar Representation

Given a quantity $v \in [0, 1]$, represent it by a binary random variable X with generating probability x, that is

$$x = P(X = 1) = P_X = v \tag{2.16}$$

Suppose that a sequence of such binary variables is observed, one per clock cycle, and denote by $X_i = \{0, 1\}$ the value of X at the ith iteration. Assuming that the resulting random process is ergodic, the statistical average of the sequence can be approximate by a time average. Thus, the estimated value of the generating probability $\hat{x}$ is given by

$$\hat{x} = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{2.17}$$

The expected value of the estimated quantity $\hat{x}$ is given by

$$\mathbb{E}[\hat{x}] = \mathbb{E}\Big[\frac{1}{n} \sum_{i=1}^{n} X_i\Big] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{1}{n} \sum_{i=1}^{n} x = x \tag{2.18}$$

Hence, $\hat{x}$ is an unbiased estimate of the true quantity and the expected value of the estimate is independent of the length of the stochastic sequence. However, the realization of $\hat{x}$ is not in general exactly equal to x due to random fluctuations in the bit-streams. This source of error in the representation can be measured by the variance of the estimate $\hat{x}$. If the sequence $\{X_i\}$ is a stationary Bernoulli process, then the binary random variables are independent and identically distributed. The variance of the estimate is therefore given by

$$Var[\hat{x}] = Var\Big[\frac{1}{n} \sum_{i=1}^{n} X_i\Big] = \frac{1}{n^2} \sum_{i=1}^{n} Var[X_i] = \frac{x(1-x)}{n} \tag{2.19}$$

Since $Var[\hat{x}] = \mathbb{E}[(\hat{x} - \mathbb{E}[\hat{x}])^2] = \mathbb{E}[(\hat{x} - x)^2]$, the error $e_r = |\hat{x} - x|$ due to random fluctuations in the bit-stream is approximately given by the standard deviation of the estimated quantity $\hat{x}$. That is,

$$e_r \approx \sigma(\hat{x}) = \sqrt{\frac{x(1-x)}{n}} \tag{2.20}$$

Thus, the error due to random fluctuations in the bit-stream is inversely proportional to the square root of the number of binary random variables, and increasing the length of the bit-stream will reduce this error. Unipolar coding is usually used in unsigned arithmetic computations.

Bipolar Representation

Given a bipolar quantity $v \in [-1, 1]$, represent it by a binary random variable X using the linear mapping

$$x = P(X = 1) = \frac{v + 1}{2} \tag{2.21}$$

where x is the generating probability [17]. The inverse transformation is given by

$$v = 2x - 1 \tag{2.22}$$

This is simply a linear transformation of the expected value of the generating probability x. If the estimate $\hat{x}$ of x is computed using the equation (2.17), then the best estimate of v is

$$\hat{v} = 2\hat{x} - 1 \tag{2.23}$$

The standard deviation of the estimate has a similar form to (2.20) and decreases by the square root of the number of stochastic samples [17]. Generally, the bipolar representation is used in signed arithmetic calculations.

Other Considerations

The practical difference of the two representations can be best understood by noting that a stochastic sequence full of zeros represents, in the unipolar format the value of 0 whereas in the bipolar format the value of -1. Similarly, a sequence containing half zeros and half ones represents, in the unipolar format the number 0.5 and in the bipolar representation the value 0. The trade-off between the two coding formats is that the bipolar representation is able to deal directly with negative numbers while, given the same bit-stream length the precision of the unipolar representation is twice that of the bipolar representation. As both signed and unsigned numbers are likely to appear in a neural network application, the bipolar representation will be used in the context of this project. Finally, to represent a number outside the range $[-1, 1]$ in stochastic arithmetic, a pre-scaling is applied to the number such that the scaled value lies within $[-1, 1]$ which in turn is encoded in stochastic arithmetic.

Chapter 3

Requirements Capture

The primary objective of this project is to investigate the circumstances under which stochastic computing could be coupled with deep neural networks. At this point, the motivation for considering such an implementation should be clear, as well as the potential advantages and disadvantages of this approach. The investigation will involve customising the arithmetic to the problem of neural networks and will be oriented around two scenarios in which stochastic computing can incorporated with deep neural networks: 1) Neural Network Inference 2) Neural Network Training. The former consists of completing a forward propagation of a trained neural network to infer information about unknown input data. The latter is an iterative procedure aiming to learn optimal model parameters. Although the aforementioned objectives are quite broad, for each task research can be oriented around the following issues:

Neural Network Inference

1. Identify the arithmetic units required to implement a forward propagation of a multi-layer perceptron. Thereinafter, investigate whether such units exist in stochastic computing and if not whether they can be devised.

2. Implement the required computational units in stochastic computing, empirically verify their operation and identify their strengths and weaknesses.

3. Having the necessary individual arithmetic units in place, study as a whole the implementation of the forward propagation of a trained neural network in stochastic computing.

4. For the implementation of neural network inference using stochastic computing, investigate the amount of samples needed for the SC-based neural network to achieve the same level of recognition accuracy as the conventional network.

5. Is there a break-even point? That is, a point where the execution time incurred by the SC network in order to achieve the required level of accuracy is so long that a conventional implementation would be preferred.

6. How does the performance of a SC neural network implementation compares, in terms of hardware area, power and energy dissipation, to a neural network implementation using conventional binary computing?

Neural Network Training

1. A training epoch involves both a forward and a backward propagation of the neural network graph. Hence, investigate how the backward propagation changes if computations are performed using stochastic arithmetic.

2. Investigate if there is any benefit from training the neural network using stochastic computing.

3. If the batch size of stochastic gradient descent is selected dynamically during training time, then at each epoch we have the choice to either proceed to the next batch or to add another sample (i.e. gradient estimate) to this batch. The latter is known to reduce the variance of the gradient estimate. In the event where training is performed using stochastic arithmetic and the bit-stream length is selected dynamically, then at each epoch we have a similar design choice. We can keep adding stochastic samples over the same gradient estimate or proceed to the next epoch. The former is likely to have a similar effect to the mini-batch size selection.

While it seems natural to consider a hardware implementation of stochastic computing on an FPGA, in the context of this project a software implementation will be considered simulating effectively the functionality of stochastic computing in the software framework. As the project does not aim to accelerate a proven concept by means of a hardware implementation, but instead to investigate whether such a concept is indeed feasible, then a software implementation is preferred as it allows greater flexibility and the opportunity to quickly experiment with stochastic computing. Further details regarding the selection of programming language and the development environment are given in Chapter 7.

Chapter 4

Design and Analysis of Stochastic Processing Elements

This chapter of the report analyses a number of stochastic processing units employed in artificial neural networks. As already discussed, operating on random bit-streams enables complex operations to be performed with very simple logic circuits. Over the past, both combinational and sequential circuits have been proposed for processing stochastic bit-streams. Prior work has shown that combinational logic can efficiently implement multiplication, squaring, scaled addition and subtraction both in unipolar and bipolar formats [17]. On the other hand, linear finite state machines (FSMs) can efficiently implement complex non-linear functions such as exponentiation, sigmoid nonlinearity mappings and linear gain with saturation [17]. In this project, a stochastic comparator is used to efficiently implement a max unit in stochastic computing. This is subsequently used to implement a bipolar SC-based ReLU function which is nowadays widely used as the main activation function in several deep neural networks.

Without loss of generality, the bipolar format of the stochastic processing units is mainly considered. The processing blocks are presented in terms of standard logic elements but the algorithmic interpretation of certain processing blocks is also discussed. A mathematical analysis of each block is presented and at the same time the processing units are implemented and verified empirically in the target language. The analysis of all of the processing elements is based on the assumption that the input bit-streams are Bernoulli sequences. This implies that the probability of each bit in the sequence being a 1 is independent from all the previously observed bits. However, this does not imply that the output of a stochastic processing element is necessarily a Bernoulli sequence. Finally, it is assumed that in the case of a unit with multiple input signals, the inputs are uncorrelated or in independent.

4.1 Conversion Circuits

In the stochastic computing domain, logical operations are performed on stochastic bit-streams. Circuits that convert ordinary binary numbers to stochastic bit-streams and vice versa are fundamental elements of SC. Such circuits have been thoroughly studied throughout the years and the designs presented are adapted by [3].

Algorithm 1: Stochastic Number Generator

Input : Float $x \in [-1, 1]$
Integer L: Number of stochastic samples
Output: $X \in \mathbb{R}^L$: Generated stochastic bit-stream

```
for i = 0 to L − 1 do
    Draw r ∼ U(−1, 1)
    if x > r then
        X[i] = 1
    else
        X[i] = 0
```

4.1.1 Stochastic Number Generator

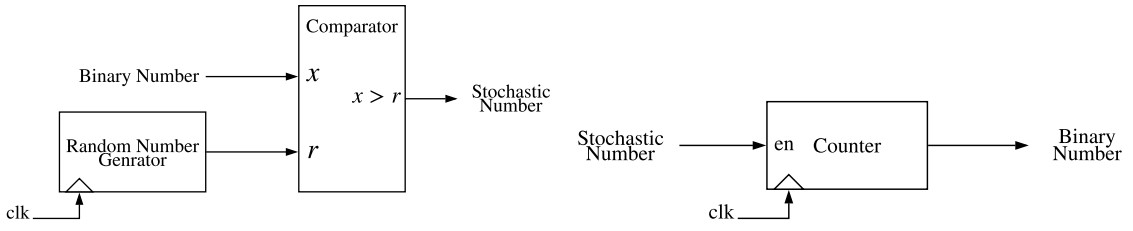
A circuit that converts a binary number to a stochastic bit-stream is known as a stochastic number generator (SNG). Figure 4.1a shows an example of such a circuit, the functionality of which is captured in Algorithm 1. By construction, the probability of a digit being a one in X is $(x+1)/2$ which is in agreement with the representation given by equation (2.21). It is important to note that through the conversion process described by Algorithm 1, the probability $p = (x+1)/2$ where $x \in [-1, 1]$, will not in general be exactly represented in the stochastic computing domain. Instead, p will be rounded to the closest number p' in the set of discrete probabilities represented by stochastic computing. Let $S = \{0, \frac{1}{L}, ..., \frac{L-1}{L}, 1\}$ denote the set of digitized probabilities represented in stochastic computing, where L is the length of the bit-stream. Whenever $p' \neq p$, then a non-zero *quantization* error, e_q, is introduced. This error is such that

$$e_q = |p - p'| \leq \frac{1}{2L} \tag{4.1}$$

Throughout the years, research has been devoted to design random number generators that will generate stochastic numbers that are suitably uncorrelated or independent [3]. Linear feedback shift registers are widely used to realise random number generation in hardware. Nevertheless, for the purpose of this project pseudo-random number generators implemented in software will be used.

4.1.2 Stochastic to Binary Conversion

Converting a stochastic bit-stream to a binary one is relatively simple. Figure 4.1b shows a circuit that can realise this conversion. This is essentially an accumulator that counts the number of ones in the stochastic bit-stream. This is then be divided by the number of clock cycles observed to compute the estimate of the generating probability $\hat{x}$. This computation is equivalent to the one given by equation (2.20) and is simply the time average of the stochastic sequence. In the



(a) Binary to stochastic (b) Stochastic to binary

Figure 4.1: Number conversion

bipolar representation, the estimate of the represented quantity can be calculated through the linear mapping given by (2.23).

4.2 Combinational Logic-based Computational Elements

Basic arithmetic operations including multiplication, scaled addition and subtraction can be implemented efficiently in the stochastic domain using very simple combinational logic circuits. The design and analysis of combinational logic-based processing elements is mainly based on prior work presented by Gaines in [17].

4.2.1 Multiplication

As already seen, multiplication of two stochastic numbers is extremely simple. This is due to the fact that multiplication is a closed operation on the interval $[0, 1]$ or $[-1, 1]$ for unipolar and bipolar signals respectively. In the bipolar representation an XNOR gate performs multiplication between two Bernoulli sequences. The XNOR output is logic 1 whenever the two inputs are either both logic 0 or logic 1. Denoting by A and B the inputs to the XNOR gate and Y the output, then for the bipolar representation one has

$$\begin{aligned} P_Y &= P_A \cdot P_B + P_{\bar{A}} \cdot P_{\bar{B}} \\ &= P_A \cdot P_B + (1 - P_A) \cdot (1 - P_B) \\ &= 2P_A \cdot P_B - P_A - P_B + 1 \end{aligned}$$

Using the fact that $P_A = \frac{(a+1)}{2}$ and $P_B = \frac{(b+1)}{2}$, then

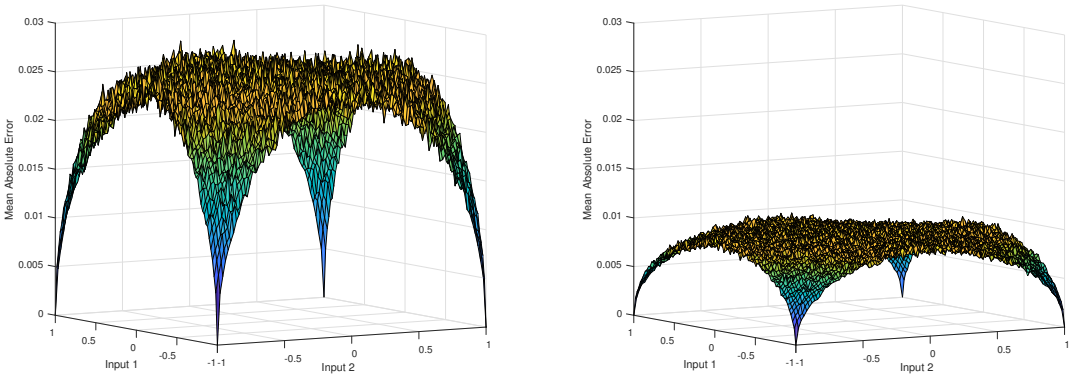
$$P_Y = \frac{ab + 1}{2}$$

For bipolar signals, $y = 2P_Y - 1$ therefore

$$y = 2\Big(\frac{ab + 1}{2}\Big) - 1 = a \times b \tag{4.2}$$

The output of the stochastic multiplier provides an unbiased estimate of the exact result and if A and B are independent Bernoulli sequences, then the output Y is also a Bernoulli sequence. While there is no error in the approximation of the desired function, the realization of y is not in general exactly equal to the product $a \times b$. This could be either due to random fluctuations in the bit-streams or due to quantisation error while rounding the probabilities a and b to the set of discrete probabilities represented by stochastic computing.

The stochastic multiplier is implemented in the target language and its operation is verified empirically. Figure 4.2 shows the multiplication errors for the bipolar multiplier. Each point represents the absolute error averaged over 500 multiplication results and a bit-stream of length 2^{10} is used in Figure 4.2a whereas 2^{13} stochastic samples are used in Figure 4.2b. As expected, increasing the bit-stream length reduces the error in the computation. This is due to the fact that for a longer bit-stream the variance in the representation decreases but also the set of digitized probabilities that can be exactly represented by stochastic computing increases. The former consequence reduces the error due to random fluctuations in the bit-stream, whereas the later reduces the quantisation error introduced by the SNG, thereby reducing the overall error in the multiply operation. Interestingly, the error is zero for the four combinations where $a = \pm 1$ and $b = \pm 1$.



(a) 2^{10} random bits (b) 2^{13} random bits

Figure 4.2: Mean absolute errors for multiplication of two numbers in stochastic computing

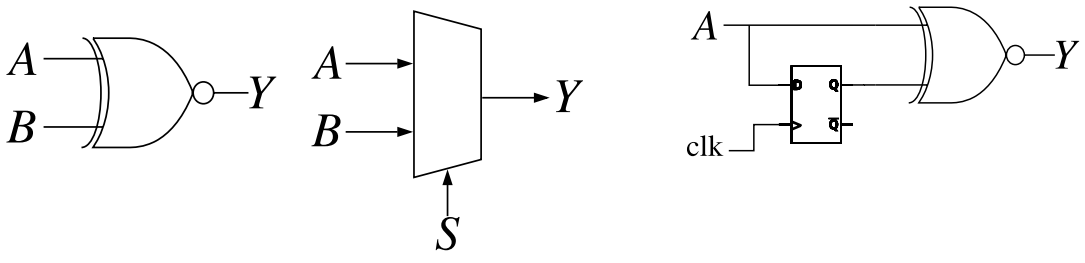
For these inputs, the generating probability of the bit-stream is either zero or one. Thus, the variance in the representation of both inputs is zero. For those inputs the output y is also equal to ± 1, therefore the variance in the representation of the output is also zero. Effectively, for those input combinations the multiplication in SC is deterministic, i.e. the variance associated with the computation is zero. Hence, no error is introduced.

4.2.2 Squaring

The squaring operation is very similar to that of multiplication. However, attempting to square a stochastic signal by connecting it to both inputs of a XNOR gate results in a sequence that is always logic 1. This is because the two input signals are correlated with each other [6]. This effect can be avoided by multiplying a stochastic sequence with its delayed, by one clock cycle, copy sequence. In that case, the two inputs are uncorrelated and the output sequence will approximate the square value of the input. The delay can be realised in hardware by placing a D-type flip-flop in one of the inputs of the XNOR gate as illustrated in Figure 4.3c. Flip-flops used in this context perform no computation. Instead, they are used to statistically isolate two cross-correlated sequences [17].

4.2.3 Change of Sign

The sign of a bipolar number can be changed by simply passing the bit-stream through a NOT logic gate performing element-wise inversion of the signal.



(a) Bipolar Multiplier (b) Scaled Adder (c) Squaring Circuit

Figure 4.3: Stochastic arithmetic units

4.2.4 Addition and Subtraction

Addition and subtraction in stochastic arithmetic are slightly more complex operations than multiplication. This is due to the fact that addition and subtraction are not closed operations on the interval $[0, 1]$ or $[-1, 1]$. The result of adding two numbers that lie within [-1,1] does not necessarily lie within [-1,1]. For this reason, a *scaled add* operation is used in SC in order to map the output of the adder from $[-2, 2]$ to $[-1, 1]$ [3]. The weighted sum of two probabilities, $\alpha p_1 + (1 - \alpha)p_2$, where $0 \leq \alpha \leq 1$, lies within $[-1, 1]$ and is representable in the stochastic computing domain. Such a computation can be realised using a two-input multiplexer where the select line is driven by the selecting probability α [17]. Consider the MUX in Figure 4.3b. The probability of a logic one appearing at the output is equal to

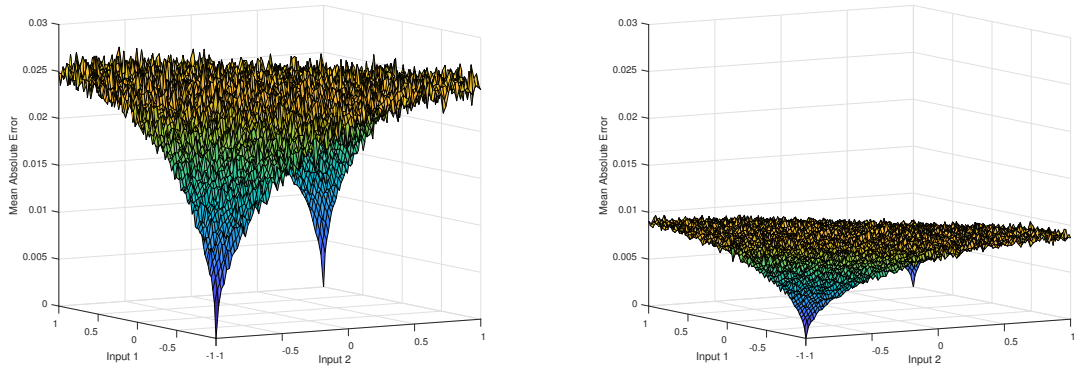
$$P_Y = P_A \cdot P_S + P_B \cdot P_{\bar{S}}$$

By choosing $P_S = 0.5$, for bipolar signals one has

$$y = 2P_Y - 1 = \frac{a + b}{2} \tag{4.3}$$

In other words, the MUX generates an output with a generating probability that is the weighted sum of the input probabilities. $y = a \oplus b = (a + b)/2$ will be used to denote the scaled addition operation in stochastic computing. Note that for bipolar signals $P_S = 0.5$ corresponds to a stochastic bit-stream having 50% zeros and 50% ones. Scaled subtraction can be implemented using the same MUX unit simply by inverting the input to be subtracted.

The scaled adder is simulated in the target environment and its operation is verified empirically. Figure 4.4 shows the addition errors for the scaled adder using the bipolar representation. Each point represents the absolute error averaged over 500 scaled addition results. As in the case of the stochastic multiplier, increasing the bit-stream length reduces the error in the computation. On the contrary though, a scaled addition in stochastic computing occurs with no error only when $a = b = \pm 1$. Once again, when a and b are equal to ± 1 the variance in the bipolar representation of the inputs is zero. However, for the input combination $a = 1$, $b = -1$ and vice versa, the result of the (scaled) addition is equal to zero. Thus, the variance associated with the bipolar representation of the output is non-zero, resulting to a finite but non-zero error in the computation. On the other hand, in the event where $a = b = \pm 1$ the output $y = a \oplus b = \pm 1$, thus the variance in the representation of the output is also zero, implying that the overall addition in SC is deterministic.



(a) 2^{10} random bits

(b) 2^{13} random bits

Figure 4.4: Mean absolute errors for scaled addition of two numbers in stochastic computing

4.2.5 Inner Product

The inner product is the core operation of artificial neurons both for feedforward networks but also for convolutional networks. Hence, to effectively implement neural networks using stochastic computing an efficient stochastic inner product unit is required. Similar to addition and subtraction, the inner product is not a closed operation on the interval $[-1, 1]$, hence a *scaled inner product* is utilised in the context of SC. As proposed by Gaines in [17], the two-input scaled adder can be extended to the weighted sum of an arbitrary number of input signals using the same MUX architecture. For a MUX unit with N inputs, this is done by selecting one of the input lines at random, with a certain probability of selecting each one, and connecting the selected input line to the output line for a single clock cycle, i.e. for a single bit. Formally, a MUX which randomly selects an input $i \in \{1, ..., N\}$ with some probability α_i such that $\sum_i \alpha_i = 1$ will produce an output whose generating probability is

$$P_Y = \sum_{i=1}^{N} \alpha_i P_{X_i} \tag{4.4}$$

where P_{X_i} for $i = 1, ..., N$ is the generating probability of each input signal [17]. For bipolar signals (4.4) yields

$$y = 2P_Y - 1 = 2\left[\sum_{i=1}^{N} \alpha_i \Big(\frac{x_i + 1}{2}\Big)\right] - 1 = \sum_{i=1}^{N} \alpha_i x_i \tag{4.5}$$

In the special case of an equally weighted sum, i.e. $\alpha_i = 1/N$, and for input values $x_i \in [-1, 1]$ for all $i = 1, ..., N$, the inner product calculated by the N-input MUX is scaled to y/N. Clearly, in the formulation given by (4.5) the selecting probabilities resemble the role of weight coefficients in a typical inner product computation given by $\sum_i^N w_i x_i$. Nevertheless, in a typical neural network, inner product computations will not involve equally weighted inputs. In fact, it is very unlikely that the weights $\{w_i\}$ in a neuron computation will define a probability distribution. That is, $\sum_i w_i \neq 1$ and there may exist weights such that $w_i < 0$.

Consider for example the inner product $y = \boldsymbol{x}^T \boldsymbol{w}$ where $\boldsymbol{x} \in \mathbb{R}^N$ and $\boldsymbol{w} \in \mathbb{R}^N$ and assume, without loss of generality, that the inputs x_i are such that $x_i \in [-1, 1]\ , \forall i = 1, ..., N$. The architecture proposed by Gaines can be extended to consider the general inner product between the inputs x_i and any set of weights $\{w_i\}$. This extension is presented in Algorithm 2[1]. In summary, the weight values are used to define a probability distribution over the inputs that specifies the probability of selecting a particular input signal. Thereinafter, for each output sample an input bit-stream is selected at random according to this probability distribution and the selected input is connected to the output for one clock cycle, i.e. the corresponding bit of the selected input is copied to the output bit-stream.

It is important to note that the output of the inner product specified by Algorithm 2 is scaled by $s_{out} = \sum_{i=1}^{N} |w_i|$ and not N. In the special case where $\{w_i\}$ are such that $w_i = 1\ \forall i = 1, ..., N$, i.e. the inputs are equally weighted, then Algorithm 2 reduces to the simple inner product architecture proposed by Gaines and $s_{out} = N$. For notation purposes, let $y = \boldsymbol{x} \odot \boldsymbol{w}$ denote the scaled inner product computed by Algorithm 2 in stochastic computing.

As a side remark, an alternative way to implement the inner product would be to use XNOR multiply units to compute the products $w_i x_i$ for all i followed by an equally weighted N-input MUX unit to accumulate the results. In contrast to the implementation above, this approach requires to convert weight values into stochastic bit-streams. As the implementation given by Algorithm 2 provides greater flexibility in software, it is preferred in the context of this project.

[1] Indices in algorithms will always start from 0 to maintain consistency with indexing in programming languages

Algorithm 2: Stochastic Inner Product

Input : Number of inputs N
Floating-point weight coefficients $w_i \in \mathbb{R}$
Stochastic bit-streams $X_i \in \mathbb{R}^L$ representing $x_i \in [-1, 1]$
Output: Bit-stream $Y \in \mathbb{R}^L$
Integer $s_{out} \in \mathbb{R}$

for $i = 0$ **to** $N - 1$ **do**
  Invert inputs corresponding to negative weights
  if $w_i < 0$ **then**
    $Z_i = \bar{X}_i$
  else
    $Z_i = X_i$
  Define a probability distribution over the inputs
  $\alpha_i = \frac{|w_i|}{\sum_j |w_j|}$
Generate the output bit-stream
for $j = 0$ **to** $L - 1$ **do**
  $i \leftarrow$ Draw a sample from $\{0, ..., N - 1\}$ with replacement according to $\{\alpha_i\}$
  $Y[j] = Z_i[j]$
$s_{out} = \sum_{i=0}^{N-1} |w_i|$

4.3 FSM-based Computational Elements

Combinational logic can efficiently implement basic arithmetic operations in stochastic computing, namely polynomial functions that map the interval $[0, 1]$ to $[0, 1]$ for the unipolar representation or $[-1, 1]$ to $[-1, 1]$ for the bipolar one [31]. However, as illustrated in Chapter 2.2 artificial neural networks employ highly non-linear activation and output units such as the hyperbolic tangent function or the rectified linear unit. The implementation of such non-linear functions with combinational logic is sometimes impossible and is in general not straightforward [38].

Gaines described the used of an ADaptive Digital Element (ADDIE) for the generation of arbitrary functions based on a saturating counter, that is a counter which will not decrement below its minimum state value and will not increment beyond its maximum state value [17]. In the ADDIE the state of the counter is controlled in a closed loop fashion. Such a system certainly has advantages, however it requires the output of the counter to be converted into a stochastic bit-stream in order to implement the closed loop feedback [6]. An approach which is highly inefficient and intensive in terms of hardware requirements. Adopting the idea of the saturating counter Brown and Card [6] proposed the use of linear finite state machines as the basis to implement highly non-linear functions in stochastic computing without using closed loop feedback leading to a much more efficient hardware implementation. The following section presents some important properties of the linear FSM proposed by Brown and Card that are directly relevant to the design and analysis of FSM-based computational elements in stochastic computing.

4.3.1 Analysis of the linear FSM

The basic form of the proposed FSM is illustrated in Figure 4.5. It consists of a set of N states arranged in a linear form (i.e. a saturating counter). Usually, $N = 2^K$ is chosen where K is a positive integer. This is a no skips model. That is transitioning from the first to the last state must occur through a set of transitions through all of the intermediate states [17]. Additionally, the state transitions are controlled by the input stochastic sequence X which is assumed to be a

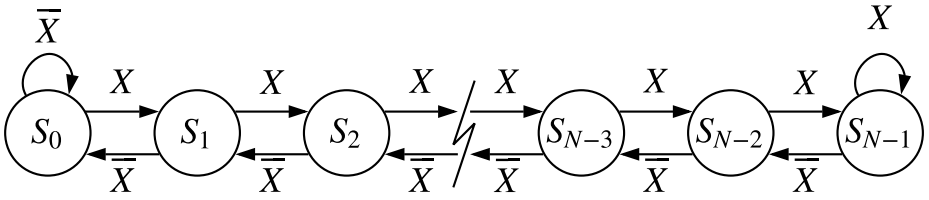


Figure 4.5: A generic linear state machine transition diagram

Bernoulli sequence and the output[2] Y at each clock cycle is determined entirely by the current state. Note that the states S_0 and S_{N-1} have saturating effects.

The system in Figure 4.5 has an interesting mathematical interpretation. The fact that the input X is a Bernoulli process implies that the bits (binary random variables) in the sequence are independent to each other. Therefore, given the present state, the next transition is entirely determined by the current state and does not dependent on sequence of states that preceded it. Such a memoryless property is known as the Markov property of a stochastic process. Hence, the system in Figure 4.5 can be modelled as a Markov chain and according to Brown and Card such a system is ergodic and will have a single stable hyperstate [6]. The existence of a single stable hyperstate implies that the individual state probabilities in the hyperstate must sum to unity, and that the probability of transitioning from state S_{i-1} to state S_i must equal the probability of transitioning from state S_i to state S_{i-1} [6].

Let P_X denote the probability that each bit in the input bit-stream X is one, P_Y denote the probability that each bit in the output bit-stream is one and $P_{i(P_X)}$ denote the probability that the current state is S_i under the input probability P_X. The above conclusions translate to

$$P_{i(P_X)} \cdot (1 - P_X) = P_{i-1(P_X)} \cdot P_X \tag{4.6}$$

$$\sum_{i=0}^{N-1} P_{i(P_X)} = 1 \tag{4.7}$$

$$P_Y = \sum_{i=0}^{N-1} s_i \cdot P_{i(P_X)} \tag{4.8}$$

where s_i in (4.8) takes two possible values, 0 or 1, and specifies the output Y of the FSM when the current state is S_i [31]. That is, if the current state is S_i then the output $Y = s_i$. It is important to note that depending on how the output function is formed, there can be significant correlations between the samples of output sequence. Therefore, the sequence generated by such a FSM is not in general a Bernoulli process. As stated in [31], $P_{i(P_X)}$ and P_X are related as follows

$$P_{i(P_X)} = \frac{\left(\frac{P_X}{1-P_X}\right)^i}{\sum_{j=0}^{N-1} \left(\frac{P_X}{1-P_X}\right)^j} \tag{4.9}$$

Proof. To prove (4.9), rearrange equation (4.6) as

$$P_{i(P_X)} = \frac{P_X}{1 - P_X} \cdot P_{i-1(P_X)} \tag{4.10}$$

[2]Not shown on Figure 4.5

and recursively eliminate $P_{i-1(P_X)}$ from (4.10) to obtain

$$P_{i(P_X)} = \Big(\frac{P_X}{1-P_X}\Big)^i \cdot P_0 \tag{4.11}$$

Furthermore, substituting (4.11) in (4.7) yields

$$\sum_{i=0}^{N-1} \Big(\frac{P_X}{1-P_X}\Big)^i \cdot P_0 = 1 \tag{4.12}$$

Equation (4.9) follows directly from (4.11) and (4.12). □

An alternative formulation of $P_{i(P_X)}$, based different intervals of P_X can be computed as follows

$$P_{i(P_X)} = \begin{cases} \frac{\left(\frac{P_X}{1-P_X}\right)^i \cdot \left(\frac{1-2P_X}{1-P_X}\right)}{1-\left(\frac{P_X}{1-P_X}\right)^N}, & 0 \leq P_X < 0.5 \\ \frac{1}{N}, & P_X = 0.5 \\ \frac{\left(\frac{1-P_X}{P_X}\right)^{N-i-1} \cdot \left(\frac{2P_X-1}{P_X}\right)}{1-\left(\frac{1-P_X}{P_X}\right)^N}, & 0.5 < P_X \leq 1 \end{cases} \tag{4.13}$$

Proof. For compactness, let $r = P_X/(1-P_X)$ and consider the interval $0 \leq P_X < 0.5$. Then,

$$P_{i(P_X)} = \frac{r^i}{\sum_{j=0}^{N-1} r^j} \tag{4.14}$$

The denominator of (4.14) is in the form of geometric series and since $r \neq 1$ (in fact $r < 1$), $P_{i(P_X)}$ can be re-written as

$$P_{i(P_X)} = \frac{r^i \cdot (1-r)}{1-r^N}$$

Substituting for r and re-arranging leads to

$$P_{i(P_X)} = \frac{\left(\frac{P_X}{1-P_X}\right)^i \cdot \left(\frac{1-2P_X}{1-P_X}\right)}{1-\left(\frac{P_X}{1-P_X}\right)^N} \tag{4.15}$$

In the interval $0.5 < P_X \leq 1$, $r > 1$. Expressing and expanding (4.14) in terms of r^{-1} leads to

$$P_{i(P_X)} = \frac{\left(\frac{1-P_X}{P_X}\right)^{-i} \cdot \left(1-(\frac{1-P_X}{P_X})^{-1}\right)}{1-\left(\frac{1-P_X}{P_X}\right)^{-N}}$$

Re-arranging and simplifying the above equation yields

$$P_{i(P_X)} = \frac{\left(\frac{1-P_X}{P_X}\right)^{N-i-1} \cdot \left(\frac{2P_X-1}{P_X}\right)}{1-\left(\frac{1-P_X}{P_X}\right)^N} \tag{4.16}$$

Finally, note that when $P_X = 0.5$, $P_{i(P_X)} = 1/N$, which in combination with (4.15) and (4.16) prove (4.13). □

As proposed in [31], if we define

$$t_{(P_X)} = \frac{0.5 - |P_X - 0.5|}{0.5 + |P_X - 0.5|} \tag{4.17}$$

equation (4.13) can be re-written as follows,

$$P_{i(P_X)} = \begin{cases} \frac{t^i_{(P_X)} \cdot (1 - t_{(P_X)})}{1 - t^N_{(P_X)}}, & 0 \leq P_X < 0.5 \\ \frac{1}{N}, & P_X = 0.5 \\ \frac{t^{N-i-1}_{(P_X)} \cdot (1 - t_{(P_X)})}{1 - t^N_{(P_X)}}, & 0.5 < P_X \leq 1 \end{cases} \tag{4.18}$$

Based on the above equation, the authors of [31] introduce the following properties of the linear FSM used in stochastic computing.

Property 1

$P_{i(P_X)}$ and $P_{N-i-1(P_X)}$ are symmetric about $P_X = 0.5$. That is, $P_{i(P_X)} = P_{N-i-1(P_X)}$.

Property 2

As $N \to \infty$ (i.e. the number of states is large enough)

- P_Y will be mainly determined by the configuration of the states from S_0 to $S_{N/2-1}$ when $0 \leq P_X < 0.5$
- P_Y will be mainly determined by the configuration of the states from $S_{N/2}$ to S_{N-1} when $0.5 < P_X \leq 1$

Hence, the output of the linear FSM, given by (4.8) can be re-written as follows,

$$P_Y \begin{cases} \approx \sum_{i=0}^{N/2-1} s_i \cdot P_{i(P_X)}, & 0 \leq P_X < 0.5 \\ = \sum_{i=0}^{N-1} \frac{s_i}{N}, & P_X = 0.5 \\ \approx \sum_{i=N/2}^{N-1} s_i \cdot P_{i(P_X)}, & 0.5 < P_X \leq 1 \end{cases}$$

In conclusion , the preceded analysis holds for the general linear FSM proposed by Brown and Card and will be used as the basis to analyse several FSM-based computational elements introduced in the subsequent subsections.

4.3.2 Stochastic Hyperbolic Function

Prior to the introduction of the rectified linear unit, the hyperbolic tangent function, *tanh*, was the most popular activation function along with the logistic sigmoid. Although the ReLU has been widely adopted by many researches and practitioners as the default non-linear activation, the hyperbolic tangent is still used as an activation function in many deep learning applications.

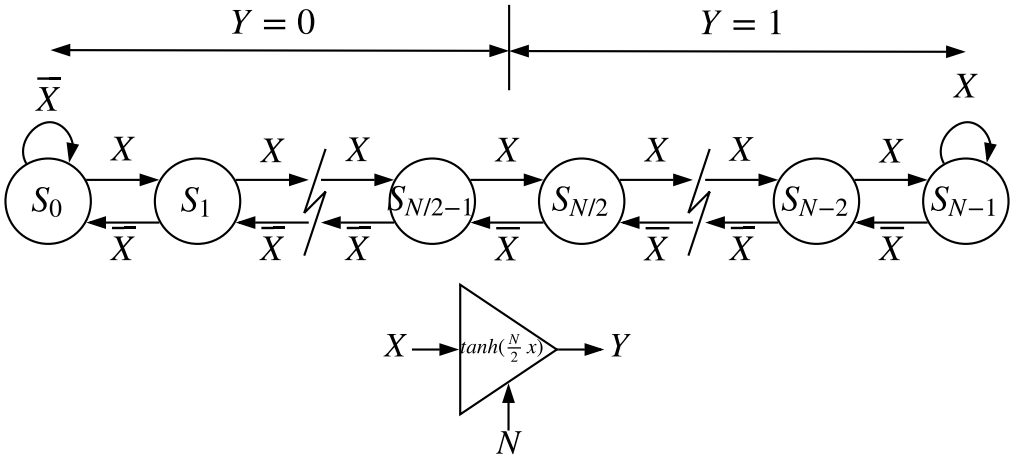


Figure 4.6: State transition diagram of the FSM-based *Stanh* function along with its circuit symbol

Consequently, a stochastic approximation of the hyperbolic function, namely *Stanh*, is considered in this section. As proposed in [6] the stochastic approximation to the *tanh* function can be achieved by setting s_i in (4.8) as follows,

$$s_i = \begin{cases} 0, & 0 \leq i \leq \frac{N}{2} - 1 \\ 1, & \frac{N}{2} \leq i \leq N - 1 \end{cases} \tag{4.19}$$

The corresponding state machine configuration is shown in Figure 4.6, where both input and output signals are represented in the bipolar format. This configuration, *approximates* the *tanh* function in stochastic computing as follows,

$$y = tanh(\frac{N}{2}x) \tag{4.20}$$

A proof based on the analysis presented in [31] is subsequently given.

Proof. Based on (4.19) the output probability P_Y is given by

$$P_Y = \sum_{i=N/2}^{N-1} P_{i(P_X)} \tag{4.21}$$

Substituting for $P_{i(P_X)}$ in (4.21) with (4.9) yields

$$P_Y = \frac{\sum_{i=N/2}^{N-1} \left(\frac{P_X}{1-P_X}\right)^i}{\sum_{j=0}^{N-1} \left(\frac{P_X}{1-P_X}\right)^j} = \frac{\sum_{i=0}^{N/2-1} \left(\frac{P_X}{1-P_X}\right)^{i+\frac{N}{2}}}{\sum_{j=0}^{N-1} \left(\frac{P_X}{1-P_X}\right)^j} \tag{4.22}$$

For compactness, let $r = P_X/(1 + P_X)$ and using the finite duration geometric series expansion (4.22) can be re-written as

$$P_Y = \frac{r^{N/2}(1 - r^{N/2})}{1 - r^N} = \frac{r^{N/2}}{1 + r^{N/2}} = \frac{\left(\frac{P_X}{1-P_X}\right)^{N/2}}{1 + \left(\frac{P_X}{1-P_X}\right)^{N/2}} \tag{4.23}$$

For bipolar signals, $P_Y = \frac{y+1}{2}$ and $P_X = \frac{x+1}{2}$. Substituting in (4.23) and solving for y, one obtains

$$y = \frac{\left(\frac{1+x}{1-x}\right)^{N/2} - 1}{\left(\frac{1+x}{1-x}\right)^{N/2} + 1} \tag{4.24}$$

Using a first order Taylor series expansion for e^x, that is $e^x \approx 1 + x$ and $e^{-x} \approx 1 - x$, (4.24) can be re-written as

$$y \approx \frac{(e^{2x})^{N/2} - 1}{(e^{2x})^{N/2} + 1} \approx \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}} = tanh(\frac{N}{2}x) \tag{4.25}$$

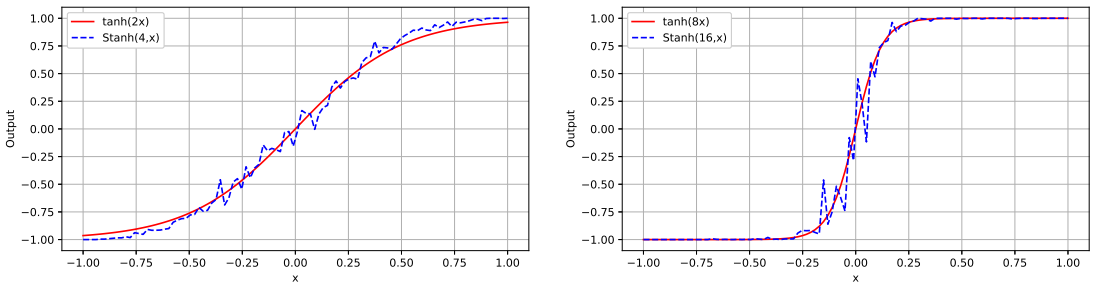
Hence, the approximate function in stochastic computing, Stanh, is given by

$$Stanh(N, x) \approx tanh(\frac{N}{2}x) \tag{4.26}$$

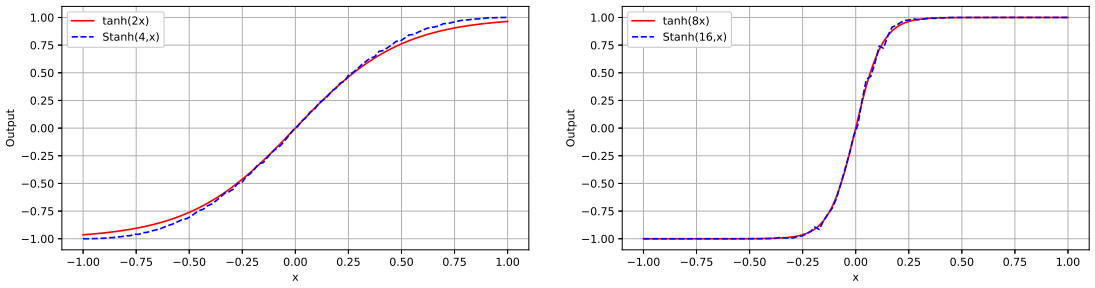
which proves the claim. □

In contrast to combinational logic-based processing elements studied in Section 4.2, where there was no error in the approximation of the desired function, the *Stanh* FSM only provides an approximation of the desired function. Thus, an approximation error e_a is introduced in addition to the error due to random fluctuations in the bit-streams (e_r) and the quantization error (e_q).

The claim can be supported both mathematically as well as empirically. The relation given by (4.25) is based on a first order Taylor series expansion, thus *Stanh* is clearly an approximation of the desired transfer function. To support the claim empirically, the *Stanh* function is simulated in the target language and results for $N = 4$ and $N = 16$ are shown in Figure 4.7. Clearly, the approximation depends on the number of states, N, in the FSM. For small N, the approximation at the saturation intervals is relatively poor, whereas for large N the approximation is much better. Furthermore, as expected, increasing the bit-stream length improves the overall estimate as both the error due to random fluctuations in the bit-streams as well as quantization errors decrease. In



(a) Bit-stream length = 2^{10}



(b) Bit-stream length = 2^{15}

Figure 4.7: Simulation results for the stochastic approximation of the tanh function

Number of States	4	8	16	32
Approximation Error	0.5213	0.1133	0.0502	0.049

Table 4.1: Approximation error of the *Stanh* function versus the number of states N

fact, for sufficiently long bit-streams and an appropriate number of states, the stochastic computing estimate approximates the true function almost exactly.

The approximation error can be quantitatively computed as the euclidean distance between the true and approximate function given by stochastic computing. For the *Stanh* function, the approximation error versus different number of states is listed in Table 4.1. The error is computed over 200 linearly spaced points within $[-1, 1]$ using a bit-stream length of 2^{20} samples. The reason for using such a long bit-stream is to effectively eliminate quantization errors as well as errors due to random fluctuations in the bit-streams, so that the error computed by the euclidean distance describes as best as possible the approximation error of the *Stanh* function. The results in Table 4.1 verify the aforementioned relationship between the quality of the approximation and the number of states N.

4.3.3 Stochastic Exponentiation Function

Although not directly utilised in the context of this project, a stochastic implementation of the exponentiation function, *Sexp*, is illustrated in this section based on the FSM configuration proposed by Brown and Card [6]. The approximation can be achieved by setting s_i in (4.8) as follows,

$$s_i = \begin{cases} 1, & 0 \leq i \leq N - G - 1 \\ 0, & N - G \leq i \leq N - 1 \end{cases} \tag{4.27}$$

where N is the total number of states and G is an integer gain parameter such that $G \ll N$. The corresponding FSM configuration is shown in Figure 4.8. It is important to note that unlike the *Stanh* approximation, *Sexp* is configured with the input as a bipolar signal and the output as a unipolar signal. Hence, if the output of such a block is to be further processed in a bipolar system one must take into account the difference in the representation. The reason for representing the output in the unipolar format is that the exponentiation function yields only non-negative values, hence the stochastic approximation is valid only for inputs greater than zero. Based on

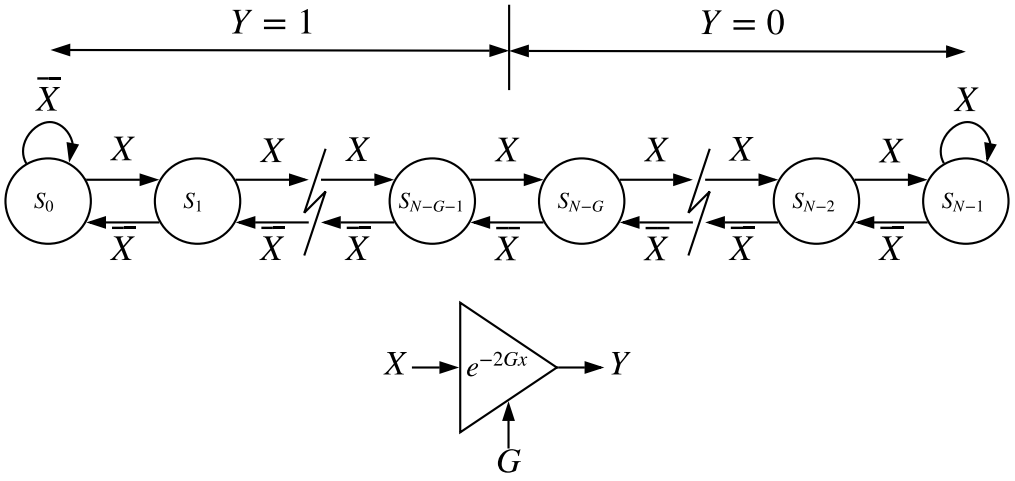


Figure 4.8: State transition diagram and circuit symbol of the FSM-based *Sexp* function

the properties of the linear FSM presented in Section 4.3.1, a similar analysis as in the case of the *Stanh* is illustrated for the *Sexp* function. The configuration in Figure 4.8, approximates the exponentiation function in stochastic computing as follows,

$$y \approx \begin{cases} 1, & -1 \leq x \leq 0 \\ e^{-2Gx}, & 0 < x \leq 1 \end{cases} \tag{4.28}$$

Following the work in [31], the claim is subsequently proven.

Proof. Based on (4.27) the output probability P_Y of *Sexp* is given by

$$P_Y = \sum_{i=0}^{N-G-1} P_{i(P_X)} \tag{4.29}$$

Substituting for $P_{i(P_X)}$ in (4.29) with (4.18) yields

$$P_Y = \begin{cases} \sum_{i=0}^{N-G-1} \frac{t_{(P_X)}^i \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N}, & 0 \leq P_X < 0.5 \\ \frac{N-G}{N}, & P_X = 0.5 \\ \sum_{i=0}^{N-G-1} \frac{t_{(P_X)}^{N-i-1} \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N}, & 0.5 < P_X \leq 1 \end{cases} \tag{4.30}$$

Based on Property 2 and since $G \ll N$, (4.30) can be re-written as follows,

$$P_Y \approx \begin{cases} \sum_{i=0}^{N/2-1} \frac{t_{(P_X)}^i \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N}, & 0 \leq P_X < 0.5 \\ 1, & P_X = 0.5 \\ \sum_{i=N/2}^{N-G-1} \frac{t_{(P_X)}^{N-i-1} \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N}, & 0.5 < P_X \leq 1 \end{cases} \tag{4.31}$$

Next, consider the approximation of P_Y given by (4.31) in each interval of P_X.

1. When $P_X < 0.5$, $t_{(P_X)} = \frac{P_X}{1-P_X} < 1$. Applying a geometric series expansion to (4.31) yields

$$P_Y \approx \frac{1 - t_{(P_X)}}{1 - t_{(P_X)}^N} \cdot \frac{1 - t_{(P_X)}^{N/2}}{1 - t_{(P_X)}} = \frac{1 - t_{(P_X)}^{N/2}}{1 - t_{(P_X)}^N}$$

Assuming that N is large enough, then $P_Y \approx 1$. As Y is coded in the unipolar format, it follows that $y \approx 1$.

2. When $P_X > 0.5$, $t_{(P_X)} = \frac{1-P_X}{P_X} < 1$. In this interval, P_Y can be re-written as follows,

$$P_Y \approx \frac{t_{(P_X)}^{N-1} \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N} \cdot \sum_{i=N/2}^{N-G-1} t_{(P_X)}^{-i} = \frac{t_{(P_X)}^{N/2-1} \cdot \left(1 - t_{(P_X)}\right)}{1 - t_{(P_X)}^N} \cdot \sum_{i=0}^{N/2-G-1} \left(t_{(P_X)}^{-1}\right)^i$$

Applying a geometric series expansion and simplifying the result yields

$$P_Y \approx \frac{t^G_{(P_X)} - t^{N/2}_{(P_X)}}{1 - t^N_{(P_X)}}$$

Since $t_{(P_X)} < 1$ and $N \gg G$, then $t^G_{(P_X)} - t^{N/2}_{(P_X)} \approx t^G_{(P_X)}$ and $1 - t^N_{(P_X)} \approx 1$. Hence, $P_Y \approx t^G_{(P_X)}$. Substituting for $t_{(P_X)} = (1 - P_X)/P_X$ and since the bit-stream X is coded in the bipolar representation (i.e. $P_X = (x + 1)/2$), P_Y can be re-written as follows,

$$P_Y \approx t^G_{(P_X)} = \left(\frac{1 - x}{1 + x}\right)^G \tag{4.32}$$

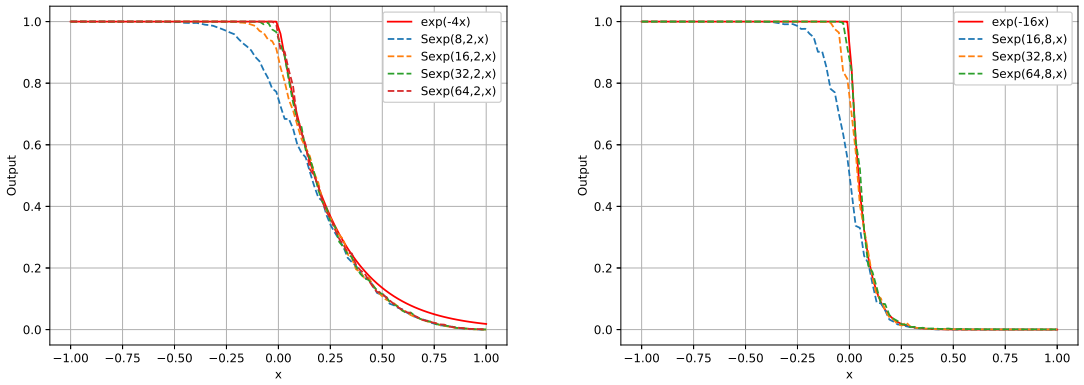
Using a first order Taylor series expansion for e^x that is, $e^x \approx 1+x$ and $e^{-x} \approx 1-x$, (4.32) simplifies to $P_Y \approx e^{-2Gx}$. Finally, recall that the output Y is coded in the unipolar format. Therefore,

$$y \approx \begin{cases} 1, & -1 \leq x \leq 0 \\ e^{-2Gx}, & 0 < x \leq 1 \end{cases} \tag{4.33}$$

The last equation proves that the configuration in Figure 4.8 proposed by Brown and Card [6] does indeed approximate the exponential function provided that $G \ll N$. □

To empirically verify the relationship between the approximation error of the *Sexp* function and the number of states N, the configuration given in Figure 4.8 is implemented in the target language and simulation results for $G = 2$ and $G = 8$ are shown in Figure 4.9. Clearly, for the approximation to be accurate the number of states N must be significantly greater than the gain factor G. Unsurprisingly, the largest error in the approximation occurs around $x = 0$.

Despite the fact that the *Sexp* is not directly utilised in the context of this project for the implementation of SC-based neural networks, it is a processing unit that can form the basis for the stochastic implementation of other non-linear functions such as the logistic sigmoid function or even the softmax function which is heavily utilised as an output unit in many neural network classifiers.



(a) $G = 2$ (b) $G = 8$

Figure 4.9: Simulation results for the stochastic approximation of the exponentiation function, $Sexp(N, G, x)$ using a bit-stream with 2^{15} samples

4.3.4 Linear Gain Function

Although the multiplication in stochastic arithmetic of a sequence with a number whose absolute value is less than one is pretty simple, multiplication by a factor whose absolute value is greater than unity (i.e. gain factor) is nontrivial. An approximation of a linear gain function with saturation was proposed by Brown and Card in 2001 [6] and the corresponding FSM configuration is shown in Figure 4.10. Note that both the input and output signals are encoded in the bipolar format and that an additional control parameter, the bit-stream K, is introduced. The control K determines the gain factor imposed by the FSM to the input bit-stream X.

The linear gain processing unit is implemented in the target language and simulation results are shown in Figure 4.11. According to Brown and Card [6], the gain imposed by the system is relatively independent to N, the number of states in the FSM. Nonetheless, as the gain factor increases (by increasing k, where $k \in [-1, 1]$), the number of states N must be increased to ensure a reasonably linear transition from -1 to 1 is maintained. The values of the bit-stream K required to achieve the particular gain factors were determined empirically by trial and error. Finally, note that in the limit, as k becomes one, the linear gain element reduces to the stochastic approximation of the *tanh* function with the gain factor being dependent on the number of states N [6].

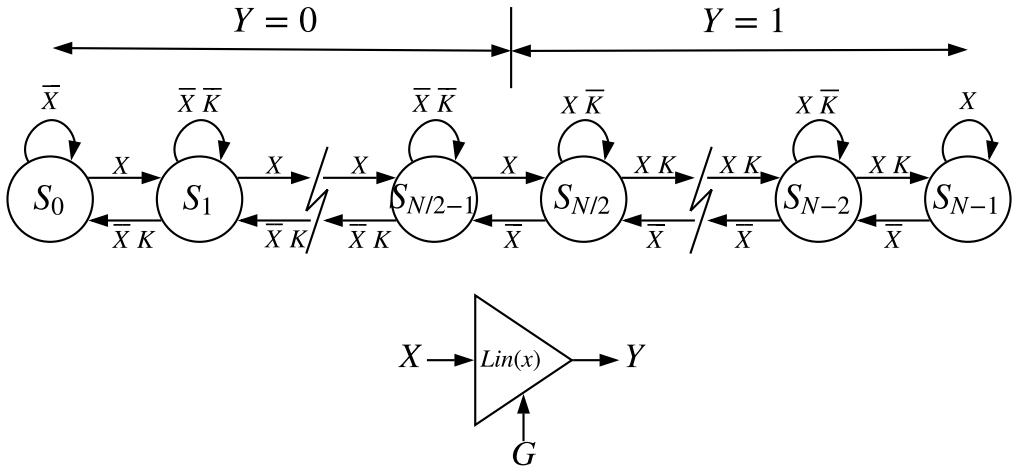
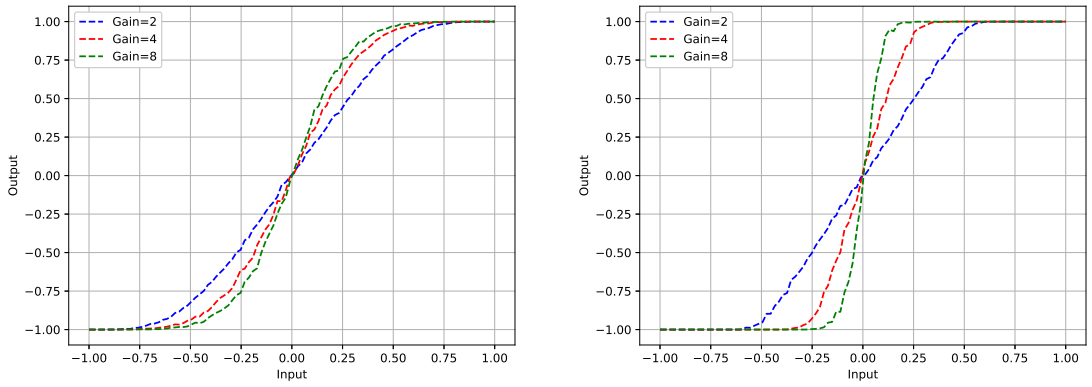


Figure 4.10: State transition diagram and circuit symbol of the linear gain block with saturation



(a) $N = 8$ (b) $N = 32$

Figure 4.11: Simulation results for the stochastic approximation of a linear gain function using a bit-stream with 2^{16} samples

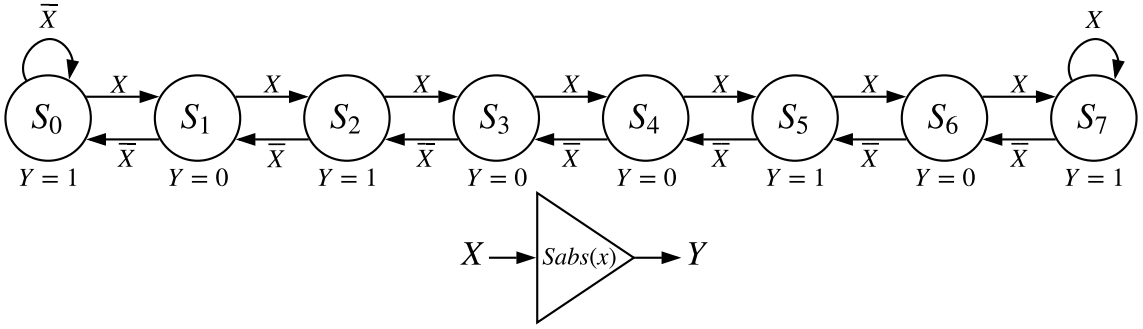


Figure 4.12: State transition diagram and circuit symbol of the *Sabs* function

4.3.5 Stochastic Absolute Value

As proposed by Li et al. in [31], an approximation to the absolute value function in stochastic computing can be achieved by setting s_i in (4.8) as follows,

- When $0 \leq i \leq \frac{N}{2} - 1$,

$$s_i = \begin{cases} 1, & i \text{ is even} \\ 0, & i \text{ is odd} \end{cases} \tag{4.34}$$

- When $\frac{N}{2} \leq i \leq N - 1$,

$$s_i = \begin{cases} 1, & i \text{ is odd} \\ 0, & i \text{ is even} \end{cases} \tag{4.35}$$

The corresponding FSM configuration for a state machine with eight states is shown in Figure 4.12. For this processing element both the input and output signals are coded in the bipolar format. The approximate transfer function, $y = |x|$, is subsequently proven.

Proof. Substituting for s_i in (4.8) with (4.34) and (4.35) yields

$$P_Y = \sum_{i=0}^{N/4-1} P_{2i(P_X)} + \sum_{i=N/4}^{N/2-1} P_{2i+1(P_X)} \tag{4.36}$$

To show that the proposed configuration implements the absolute value in stochastic arithmetic, consider P_Y over different intervals of P_X and substitute for $P_{i(P_X)}$ using (4.13).

1. When $0 \leq P_X < 0.5$, $P_X/(1 - P_X) < 1$. Substituting for $P_{i(P_X)}$ in (4.36) with (4.13) yields

$$P_Y = \frac{\frac{1-2P_X}{1-P_X}}{1 - \left(\frac{P_X}{1-P_X}\right)^N} \sum_{i=0}^{N/4-1} \left(\frac{P_X}{1-P_X}\right)^{2i} + \frac{\left(\frac{P_X}{1-P_X}\right) \cdot \left(\frac{1-2P_X}{1-P_X}\right)}{1 - \left(\frac{P_X}{1-P_X}\right)^N} \sum_{i=N/4}^{N/2-1} \left(\frac{P_X}{1-P_X}\right)^{2i} \tag{4.37}$$

Applying a geometric series expansion leads to

$$P_Y = \frac{\frac{1-2P_X}{1-P_X}}{1 - \left(\frac{P_X}{1-P_X}\right)^N} \cdot \left(\frac{1 - \left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}}}{1 - \left(\frac{P_X}{1-P_X}\right)^2}\right) + \frac{\left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}+1} \cdot \left(\frac{1-2P_X}{1-P_X}\right)}{1 - \left(\frac{P_X}{1-P_X}\right)^N} \cdot \left(\frac{1 - \left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}}}{1 - \left(\frac{P_X}{1-P_X}\right)^2}\right) \tag{4.38}$$

Using the fact that $P_X/(1 - P_X) < 1$ and assuming that the number of states N is large enough,

all powers of $P_X/(1-P_X)$ can be ignored. Thus, (4.38) can be approximated as

$$P_Y \approx \frac{\frac{1-2P_X}{1-P_X}}{1-\left(\frac{P_X}{1-P_X}\right)^2} = 1 - P_X \tag{4.39}$$

For bipolar signals, $y = 2P_Y - 1$ and $x = 2P_X - 1$, hence

$$y = 2P_Y - 1 \approx 2(1-P_X) - 1 = -x \tag{4.40}$$

2. Consider now the case when $0.5 < P_X \leq 1$. Substituting for $P_{i(P_X)}$ in (4.36) with (4.13) yields

$$P_Y = \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{N-1}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\sum_{i=0}^{N/4-1}\left(\frac{1-P_X}{PX}\right)^{-2i} + \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{N-2}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\sum_{i=N/4}^{N/2-1}\left(\frac{1-P_X}{P_X}\right)^{-2i} \tag{4.41}$$

Applying a geometric series expansion leads to

$$P_Y = \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{N-1}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\cdot\left(\frac{1-\left(\frac{1-P_X}{P_X}\right)^{-\frac{N}{2}}}{1-\left(\frac{1-P_X}{P_X}\right)^{-2}}\right) + \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}-2}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\cdot\left(\frac{1-\left(\frac{1-P_X}{P_X}\right)^{-\frac{N}{2}}}{1-\left(\frac{1-P_X}{P_X}\right)^{-2}}\right)$$

Simplifying the above equation leads to

$$P_Y = \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}+1}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\cdot\frac{1-\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}}}{1-\left(\frac{1-P_X}{P_X}\right)^2} + \frac{\frac{2P_X-1}{P_X}}{1-\left(\frac{1-P_X}{P_X}\right)^N}\cdot\frac{1-\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}}}{1-\left(\frac{1-P_X}{P_X}\right)^2} \tag{4.42}$$

Since $P_X > 0.5$, $(1-P_X)/P_X < 1$. Provided that N is large enough, all the powers of $(1-P_X)/P_X$ in (4.42) can be ignored. Hence, P_Y can be approximated as

$$P_Y \approx \frac{\frac{2P_X-1}{P_X}}{1-\left(\frac{1-P_X}{P_X}\right)^2} = P_X \tag{4.43}$$

Therefore,

$$y = 2P_Y - 1 \approx 2P_X - 1 = x \tag{4.44}$$

3. Finally, when $P_X = 0.5$, and by substituting for $P_{i(P_X)} = 1/N$ in (4.36) it easily follows that

$$P_Y = \sum_{i=0}^{N/4-1} P_{2i(P_X)} + \sum_{i=N/4}^{N/2-1} P_{2i+1(P_X)} = \frac{1}{2} \tag{4.45}$$

Note that $P_X = P_Y = \frac{1}{2}$, thus

$$y = 2P_Y - 1 = 2P_X - 1 = x = 0 \tag{4.46}$$

Combining (4.40), (4.44) and (4.46) over the different intervals of P_X yields $y = |x|$. □

The preceded analysis supports the claim that the configuration in Figure 4.12 approximates the absolute value function in stochastic arithmetic. As with the previously introduced FSM-based processing units, it is expected that as the number of states N increases the stochastic approximation will improve. Simulation results for the empirical verification of the proposed processing unit are shown in Figure 4.13. Unsurprisingly, the approximation is poorest around $x = 0$ and

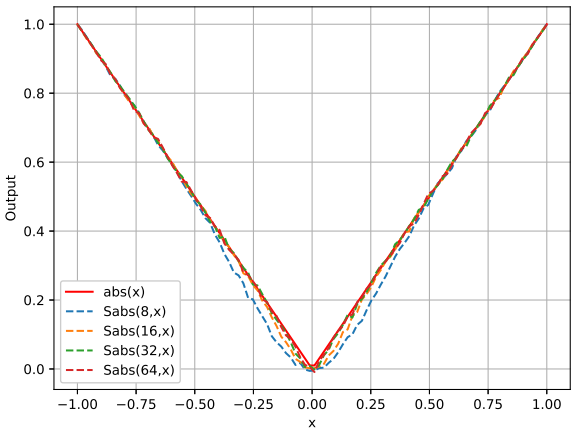


Figure 4.13: Simulation result of the stochastic implementation of the absolute value function, $Sabs(N, x)$

improves as the number of states increases. The last observation supports the preceded mathematical analysis which proves the approximate transfer function based on the assumption that N is sufficiently large.

Although the particular computational element will not be further utilised in the context of this project for the implementation of neural networks using stochastic computing, it is presented in this section as the absolute value function is widely used in several machine learning applications.

4.3.6 Stochastic Maximum Function

Undoubtedly, a stochastic implementation of a max function is of particular importance for the purpose of implementing modern deep neural networks in stochastic computing. However, in contrast to a conventional radix-2 representation, where individual bits are weighted by their position in the digit-vector, in stochastic arithmetic all the bits in the stochastic sequence are equally weighted. Thus, neither the value nor the sign of the stochastic signal is related to the exact position of the ones and zeros in the bit-stream. Instead, the ratio of ones to the length of the bit-stream determines both the sign and value of the signal. Thereby, a processing element that computes the maximum (or minimum) between two stochastic signals cannot rely on the position of the individual bits in the input bit-streams, as a binary equivalent could do.

An approximation of the max function in stochastic arithmetic, namely $Smax$, with both input and output signals encoded as bipolar stochastic bit-streams may be implemented using the configuration shown in Figure 4.14. The basic idea of the stochastic max unit is to compute the difference between the inputs A and B, i.e. $A-B$, and based on that to generate a select line signal that will choose the maximum between the two inputs. Based on the architecture in Figure 4.14, the difference is computed using the leftmost MUX unit (in combination with the NOT gate). The resulting bit-stream is fed into the $Stanh$ unit which is implemented using the FSM introduced in Section 4.3.2. Thus, if P_A is larger than P_B, then $Stanh$ tends to stay on the high state side, whereas if P_B is larger than P_A, then $Stanh$ tends to stay on the low state side. Finally, the rightmost MUX unit in Figure 4.14 selects A if the $Stanh$ output is at the high state and B if the $Stanh$ output is at the low state. Thus, the output of the circuit shown in Figure 4.14 is equal to $max(A, B)$. The claim is proven subsequently using similar arguments as the ones used in the preceded sections.

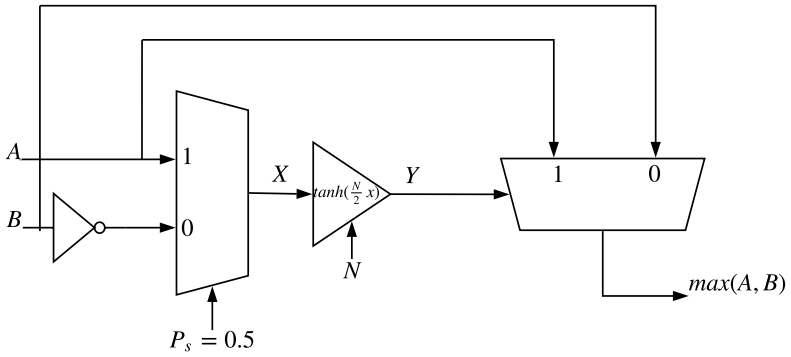


Figure 4.14: Stochastic max unit

Proof. From the configuration of the *Stanh* function it follows that

$$P_Y = \sum_{i=N/2}^{N-1} P_{i(P_X)} \tag{4.47}$$

Substituting for $P_{i(P_X)}$ in (4.47) with (4.13) when $0 \leq P_X < 0.5$ yields

$$P_Y = \frac{\frac{1-2P_X}{P_X}}{1-\left(\frac{P_X}{1-P_X}\right)^N} \sum_{i=N/2}^{N-1} \left(\frac{P_X}{1-P_X}\right)^i = \frac{\left(\frac{1-2P_X}{P_X}\right)\cdot\left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}}}{1-\left(\frac{P_X}{1-P_X}\right)^N} \sum_{i=0}^{N/2-1} \left(\frac{P_X}{1-P_X}\right)^i \tag{4.48}$$

Applying a geometric series expansion leads to

$$P_Y = \frac{\left(\frac{1-2P_X}{P_X}\right)\cdot\left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}}}{1-\left(\frac{P_X}{1-P_X}\right)^N} \cdot \left(\frac{1-\left(\frac{P_X}{1-P_X}\right)^{\frac{N}{2}}}{1-\frac{P_X}{1-P_X}}\right) \tag{4.49}$$

When $P_X < 0.5$, $P_X/(1-P_X) < 1$ thus if N is sufficiently large then (4.49) will approximate to zero[3].

On the other hand, substituting for $P_{i(P_X)}$ in (4.47) with (4.13) when $0.5 < P_X \leq 1$ yields

$$P_Y = \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{N-1}}{1-\left(\frac{1-P_X}{P_X}\right)^N} \sum_{i=N/2}^{N-1} \left(\frac{1-P_X}{P_X}\right)^{-i} = \frac{\left(\frac{2P_X-1}{P_X}\right)\cdot\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}-1}}{1-\left(\frac{1-P_X}{P_X}\right)^N} \sum_{i=0}^{N/2-1} \left(\frac{1-P_X}{P_X}\right)^{-i}$$

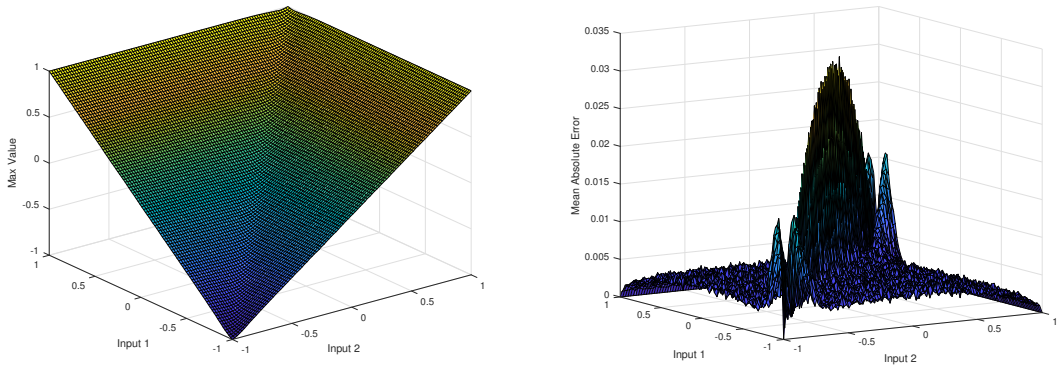
Applying a geometric series expansion and simplifying the result leads to

$$P_Y = \frac{\frac{2P_X-1}{P_X}}{1-\left(\frac{1-P_X}{P_X}\right)^N} \cdot \frac{1-\left(\frac{1-P_X}{P_X}\right)^{\frac{N}{2}}}{1-\left(\frac{1-P_X}{P_X}\right)} \tag{4.50}$$

For sufficiently large N and since $(1-P_X)/P_X < 1$, (4.50) will approximate to one. Finally, note that when $P_X = 0.5$, $P_{i(P_X)} = 1/N$. Thus, $P_Y = 1/2$. In conclusion, the output select line (i.e. the stochastic bit-stream Y) is driven by the probability P_Y which is given by

$$P_Y = \begin{cases} 0, & 0 \leq P_X < 0.5 \\ \frac{1}{2}, & P_X = \frac{1}{2} \\ 1, & 0.5 < P_X \leq 1 \end{cases} \tag{4.51}$$

[3]Terms of the form $(\frac{P_X}{1-P_X})^{\alpha N}$, for some α sufficiently large will approximate to zero.



(a) (b)

Figure 4.15: Stochastic max function. (a) Result of $Smax$ (b) Mean absolute error

Thus, if $P_A < P_B$ ($a < b$) then $P_X < 0.5$ and the output MUX selects B. Else if $P_A > P_B$ ($a > b$) then $P_X > 0.5$ and the output MUX selects A. □

The stochastic max unit is implemented in the target language and Figures 4.15a and 4.15b show $Smax$ simulation results and mean absolute errors respectively. The former shows the result of the $Smax$ function using $N = 32$ states and is consistent with the output of a conventional max unit. Each point in the latter, represents the absolute error averaged over 50 max results, while using a bit-stream length of 2^{15} samples to encode the input values. Unsurprisingly, the errors are larger on the diagonal where $a = b$. Still, the absolute errors are very small, indicating that the proposed $Smax$ unit well approximates the conventional max function. As a final remark, note that a stochastic minimum function can be achieved by simply permuting the inputs of the output (i.e. rightmost) MUX unit in Figure 4.14.

4.3.7 Stochastic Rectified Linear Unit

The sigmoid, hyperbolic tangent and rectified linear unit (ReLU) are the most popular activation functions for deep neural network applications. Nevertheless, the ReLU has nowadays been adopted as a default choice by many developers as it is simple and easy to optimise compared to the sigmoid and hyperbolic tangent function which saturate across most of their domain, making gradient-based learning difficult. An implementation of the ReLU in stochastic arithmetic, namely $SReLU$, is proposed in this section based on the max unit introduced the preceded section. The unit is illustrated in Figure 4.16[4].

The stochastic unit has both the input and output signals encoded in the bipolar representation. Since the approximation of the max operation given by $Smax$ is a function of the number of states N, the stochastic approximation of the ReLU will also be a function of N. Simulation results

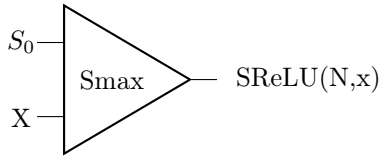


Figure 4.16: The stochastic ReLU

[4] S_0 in the first input of the $Smax$ stands for stochastic bit-streams that represent the value zero, i.e. the probability represented by the bipolar signal is equal to 0.5.

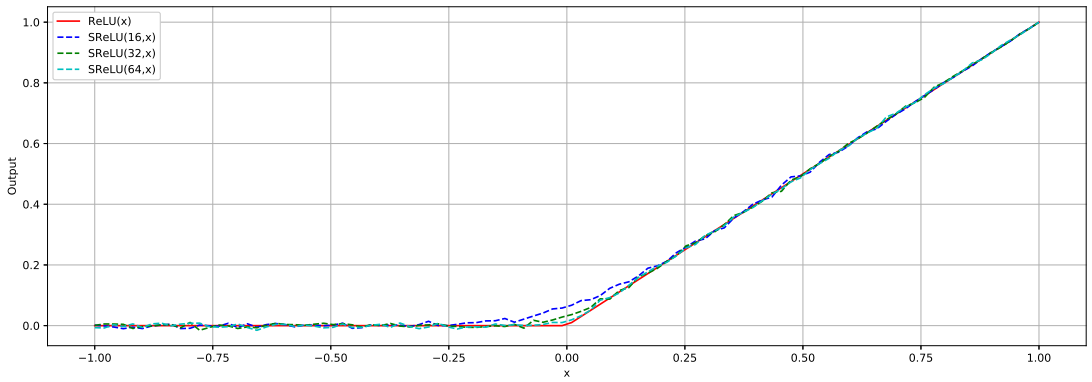


Figure 4.17: Simulation result of the stochastic implementation of the ReLU, $SReLU(N, x)$

for the proposed $SReLU$ are shown in Figure 4.17, where 2^{15} samples were used in the bipolar representation. Clearly, the approximation is a function of N and not surprisingly is poorest around $x = 0$. Nevertheless, for a sufficient number of states the stochastic implementation well approximates the conventional rectified linear unit.

4.3.8 Stochastic Max Pooling

Although not presented in the background chapter, a typical convolutional network architecture consists of convolutional layers, pooling layers and fully connected layers. A convolutional layer is responsible for extracting features through convolution of receptive fields and a set of learnable filters [18]. Thereinafter, a down-sampling step is usually performed to aggregate statistics of the extracted features aiming to reduce the dimension of the data and mitigate overfitting issues. This down-sampling process is termed *pooling* and two types of pooling are typically used, *average pooling* and *max pooling*. The former involves calculating the average value of the set of input candidates whereas the later selects the max value from the input candidates.

Average pooling can be easily implemented in stochastic computing exploiting the inherent down-scaling property of a MUX-based inner product with equally weighted input signals. On the other hand, max pooling can be realised in stochastic computing using the proposed $Smax$ unit. A possible, although naive, implementation of max pooling in stochastic computing is to construct a chain of $Smax$ units as shown in Figure 4.18. Assuming no resources are shared, the proposed configuration requires $N - 1$ units to compute the maximum out of a set of candidate inputs $\{X_i, ..., X_N\}$. Clearly, as N becomes large the hardware area required to implement such a unit will be significant. Thus, to enable a low-cost hardware implementation of stochastic max pooling alternative configurations need to be considered, perhaps analysing the possibility to share resources or to handle the input signals in an optimal way. Nonetheless, such an aspect is beyond the scope of this project and will not be further considered.

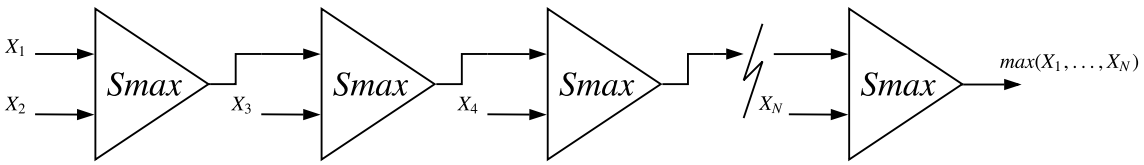


Figure 4.18: Stochastic max pooling

4.4 Saturation Arithmetic in Stochastic Computing

As analysed in Section 4.2, accumulation in stochastic arithmetic needs to be performed in a scaled manner as addition and subtraction are not closed operations on the interval $[-1, 1]$. The output of a two-input stochastic adder is therefore scaled by $1/2$. Cascading N such scaled adders, results in an output that is down-scaled by 2^N. Such a down-scaling phenomenon can cause severe accuracy loss in the overall computation especially when N is large, as a stochastic computing system using wordlengths of size 2^L can only represent values as low as $1/2^L$, which may be insufficient to represent the down-scaled output when N is large. This becomes even worst if the values involved in the computation are themselves small. Note that such imprecision cannot be compensated by post-processing (i.e. up-scaling) the output of the overall system as the information is already lost during the down-scaling procedure in the stochastic domain. Similarly, the corresponding output of the stochastic inner product is down-scaled by $\sum_{i=1}^{N} |w_i|$, which can be significantly large when either N or $|w_i|$ are large.

The aforementioned scaling factors are *worst-case* scalings and ensure that all internal signals are constrained within the representable by stochastic computing range, i.e. the range $[-1, 1]$. However, in some cases worst-case scaling can be overly pessimistic, catering for situations that are extremely rarely encountered in real world signals [7]. As briefly discussed above, overly pessimistic signal scaling in stochastic computing can lead to severe loss of precision which is at most undesired, given the inherent uncertainty in stochastic computations. Under these considerations, an alternative scaling scheme could be used to reduce the scaling factors and increase precision in the computations, however "overflow" becomes a distinct possibility[5] [7].

Similar principles apply to conventional binary arithmetic as described by Constantinides et al. in [7] and [8]. The authors present a detailed analysis of saturation arithmetic with particular emphasis on Digital Signal Processing (DSP) applications. When adding two numbers using the two's complement representation there exists a possibility of overflow, which results in a "wrap-around" phenomenon and in most applications is extremely undesired [7]. Hence, to alleviate such phenomena, signals are either scaled appropriately to avoid overflow or saturation arithmetic is used. Saturation arithmetic introduces extra hardware to avoid the wrap-around phenomenon, replacing it with saturation to either the largest positive number or the largest negative number representable at the adder output [7]. In a fixed-point representation, a saturation system is a nonlinear dynamical system containing two types of nonlinearities: saturations and truncations (roundoffs). Saturations are large-scale nonlinearities, whereas truncations are small-scale nonlinearities affecting a few of the least significant bits of a word [7].

In summary, the aim in the remaining of this section is to introduce possible architectures for the implementation of saturation arithmetic in stochastic computing. The objective while designing saturation arithmetic units in the stochastic domain is to mitigate the down-scaling effect by worst-case scalings at the output of a stochastic accumulator, thus increase precision while minimizing possible representation errors in the stochastic encoding of the result. Note that in contrast to the fixed-point realization of a saturation system in conventional binary computing, in stochastic computing truncation of low order bits is not possible as all bits in a stochastic sequence are equally weighted. Hence, only saturations can be considered as a possible way of realizing saturation arithmetic in SC. Finally, the overheads introduced for the realisation of saturation arithmetic in the context of stochastic arithmetic are analysed qualitatively.

[5] In this context, the term "overfolow" is used to denote the existence of signal values that once the scaling factors are reduced they will "overflow", i.e. exceed, the range $[-1, 1]$. Thus, they will not be representable, without a non-zero representation error, in the stochastic domain. A more clear definition of this kind of error is subsequently given.

4.4.1 Saturated Stochastic Adder

The output of the two input scaled adder in Figure 4.3b is given by $y = a \oplus b = (a+b)/2$, where a, b and $y \in [-1, 1]$. The desired sum $z = a + b$ lies, in general, within $[-2, 2]$. The output of a stochastic adder with saturation is given, in its most general form, by

$$\hat{y} = a\hat{\oplus}b = \begin{cases} M, & \frac{1}{2}(a+b) \geq M \\ \frac{1}{2}(a+b), & \frac{1}{2}|a+b| < M \\ -M, & \frac{1}{2}(a+b) \leq -M \end{cases} \tag{4.52}$$

where $M < 1$ denotes the saturation (clipping) level. Note that the output $\hat{y} \in [-M, M]$ and since $M < 1$, the full dynamic range of stochastic computing is not utilised. A linear mapping should therefore be applied to map the range $[-M, M]$ to $[-1, 1]$. This translates to the following specification

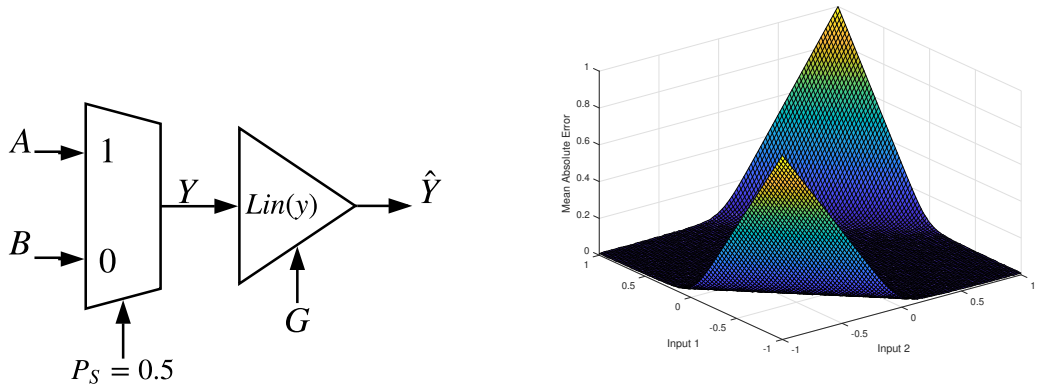
$$\hat{y} = a\hat{\oplus}b = \begin{cases} 1, & \frac{1}{2}(a+b) \geq M \\ \frac{1}{2M}(a+b), & \frac{1}{2}|a+b| < M \\ -1, & \frac{1}{2}(a+b) \leq -M \end{cases} \tag{4.53}$$

A possible way to realize such a nonlinearity is to feed the output of the scaled adder to the linear gain unit with saturation introduced in Section 4.3.4. This will apply a certain gain $G > 1$ to the output of the scaled adder while saturating the result to ± 1. In relation to (4.53), G and M are essentially related to each other as $G = 1/M$. Setting for example $G = 1/M = 2$, the output of the saturated stochastic adder is given by,

$$\hat{y} = a\hat{\oplus}b = \begin{cases} 1, & \frac{1}{2}(a+b) \geq \frac{1}{2} \\ a+b, & \frac{1}{2}|a+b| < \frac{1}{2} \\ -1, & \frac{1}{2}(a+b) \leq -\frac{1}{2} \end{cases} \tag{4.54}$$

The down-scaling due to the stochastic scaled adder is eliminated by the linear gain block, allowing the same number of stochastic bits to represent a smaller interval (i.e. $[-M, M]$) and thereby increasing precision. However, there exists the possibility of a non-zero *soft saturation error* in the representation of $a+b$ given by $\hat{y}$. The term non-linear *compression error* may also be used to denote this kind of error. Specifically, if the output $y = (a+b)/2$ of the scaled adder does not lie within $[-0.5, 0.5]$, then a soft saturation error will be introduced due to the saturation of the output. Otherwise, no error is introduced. In general, a scaled adder with saturation $M = 1/G$ will introduce a compression error if and only if the down-scaled output of the stochastic adder does not lie within $[-M, M]$. The selection of M and thereby G is a design choice and should be made with care. Possible ways to select appropriate values of M for the implementation of neural networks using stochastic computing are presented in subsequent chapters of the report.

The scaled adder with saturation is simulated in the target language and Figure 4.19b shows the mean absolute error, averaged over 100 results using 2^15 long bit-streams, of a stochastic adder with saturation level $M = 1/G = 1/2$. The errors are very small in the region where $|a+b| \leq 1$ and increase linearly in the region where $|a+b| > 1$. That is the region where soft saturation error occurs since the output of the adder exceeds the range $[-1, 1]$. Unsurprisingly, the largest absolute error occurs when $|a+b| = 2$ and is equal to one.



(a) (b)

Figure 4.19: Saturated stochastic adder. (a) Architecture (b) Mean absolute error

4.4.2 Saturated Inner Product

Similar analysis can be applied to the case of the stochastic inner product unit. Consider, without loss of generality, the inner product $y = \boldsymbol{w}^T\boldsymbol{x} = \sum_{i=1}^{N} w_i x_i$, where $x_i \in [-1, 1]$ and $w_i \in \mathbb{R}$ $\forall i = 1, ..., N$. The inner product computed in stochastic arithmetic by Algorithm 2 is downscaled by $s_o = \sum_{i=1}^{N} |w_i|$. This is the worst-case scaling, required to ensure that the result will always lie within the representable by stochastic computing range for any possible input combination. This can sometimes be overly pessimistic, catering for situations that are very unlikely to occur. Following a similar approach as in the case of the saturated stochastic adder, the output of the stochastic inner product with saturation is given, in its most general form, by

$$\hat{y} = \boldsymbol{w}\hat{\odot}\boldsymbol{x} = \begin{cases} M, & \frac{1}{s_o}\boldsymbol{w}^T\boldsymbol{x} \geq M \\ \frac{1}{s_o}\boldsymbol{w}^T\boldsymbol{x}, & \frac{1}{s_o}|\boldsymbol{w}^T\boldsymbol{x}| < M \\ -M, & \frac{1}{s_o}\boldsymbol{w}^T\boldsymbol{x} \leq -M \end{cases} \tag{4.55}$$

where $M < 1$ denotes the saturation level. Again, a linear mapping should be applied to map the output $\hat{y} \in [-M, M]$ to $\hat{y} \in [-1, 1]$ in order to take advantage of the full dynamic range of stochastic computing. The resulting specification

$$\hat{y} = \boldsymbol{w}\hat{\odot}\boldsymbol{x} = \begin{cases} 1, & \frac{1}{s_o}\boldsymbol{w}^T\boldsymbol{x} \geq M \\ \frac{1}{s_o M}\boldsymbol{w}^T\boldsymbol{x}, & \frac{1}{s_o}|\boldsymbol{w}^T\boldsymbol{x}| < M \\ -1, & \frac{1}{s_o}\boldsymbol{w}^T\boldsymbol{x} \leq -M \end{cases} \tag{4.56}$$

can be realised by cascading the MUX-based inner product with a linear gain block with saturation. The gain $G = 1/M$ is a design parameter and dictates the range of values that will remain intact by the nonlinear compression process. Soft saturation error is introduced if and only if the output of the scaled inner product, $y = \boldsymbol{w} \odot \boldsymbol{x}$, does not lie within the preserved, by compression, range $[-M, M]$. Otherwise, no error is introduced and the output y is up-scaled by G to yield $\hat{y}$.

4.4.3 Saturated Inner Product with Decomposition

The saturated inner product architecture proposed in Section 4.4.2 may still suffer from limited precision, especially if the number of inputs N is large. This is due to the fact that the MUX architecture used to realise the inner product only uses one of its inputs at a time and the remaining inputs are discarded. Hence, for a large N there might be a significant imprecision in the computation which may not be compensated by the post-processing block (i.e. linear gain) as the information has already been lost during the sampling procedure. A possible way to mitigate such behaviour is to consider the decomposition of the original inner product into several, smaller in size[6], inner products[7]. Consider the general inner product

$$y = \boldsymbol{w}^T \boldsymbol{x} = \sum_{i=0}^{N-1} w_i x_i \tag{4.57}$$

where $x_i \in [-1, 1]$ and $w_i \in \mathbb{R}$ for all $i = 0, ..., N-1$. This can also be computed as follows,

$$y = \sum_{k=0}^{P-1} \sum_{i=k\frac{N}{P}}^{(k+1)\frac{N}{P}-1} w_i x_i \tag{4.58}$$

where $P < N$. Essentially, the re-arrangement of (4.57) given by (4.58) involves calculating P inner products each of size N/P. This gives rise to the architecture illustrated in Figure 4.20. Each MUX unit in the first layer is responsible for computing the term $\sum_{i=k\frac{N}{P}}^{(k+1)\frac{N}{P}-1} w_i x_i$ for some $k = 0, ..., P-1$, whereas the output MUX unit accumulates the equally weighted intermediate results.

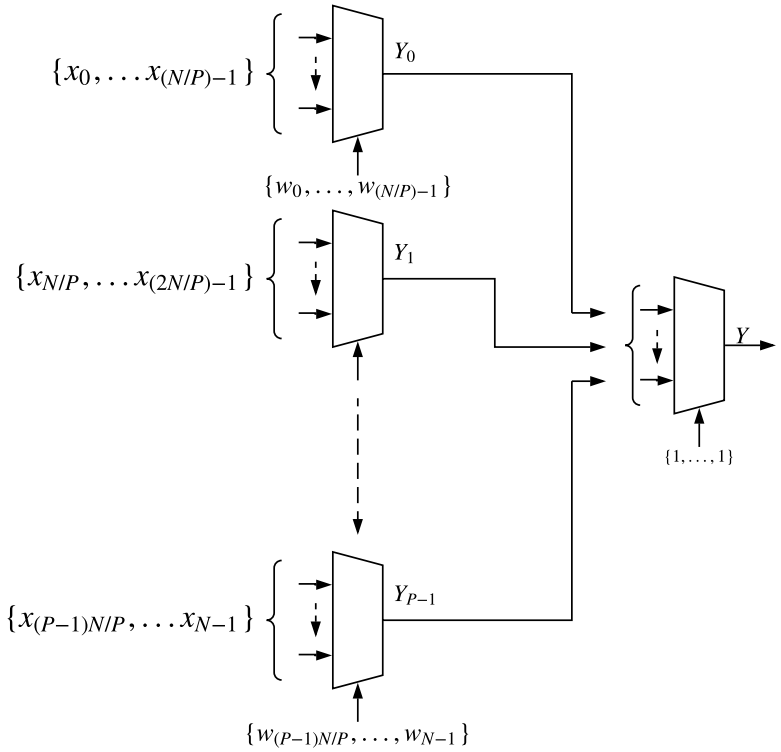


Figure 4.20: Inner product with decomposition

[6] In this context, the size of the inner product refers to the number of inputs.

[7] Obviously, another solution to improve precision in the computation is to increase the bit-stream length.

The motivation for considering such a decomposition is to reduce the number of inputs in each MUX in the case where N is large, and thereby mitigate the imprecision introduced by the MUX down-sampling process. Extending the saturated inner product introduced in Section 4.4.2, a linear gain factor G_k can be applied to each of the intermediate results but also an independent gain factor G_o can be applied to the output of the overall unit. Effectively, the proposed decomposition introduces additional design variables that may be used to reduce the overall compression error introduced by this architecture compared to the single MUX-based inner product unit with saturation presented in the preceded section. This is due to the fact that smaller saturation gain parameters may need to be applied to each signal throughout the inner product with decomposition, so the overall compression error could potentially be smaller.

4.4.4 Saturation Arithmetic Overheads

Saturation arithmetic is not a cost-free design methodology. The implementation of a saturation arithmetic component, both in conventional binary computing but also in stochastic computing, has some overheads, usually in terms of area and delay, associated with it when compared to the corresponding non-saturation component [8].

The key component to realise saturation arithmetic in the context of stochastic computing is the linear gain (with saturation) unit introduced presented in Section 4.3.4. The implementation of such a unit obviously incurs hardware area overheads. This is because each linear gain transformation needs one more FSM, hence both hardware area need to be sacrificed to realise saturation arithmetic. Depending on the approach used to implement the FSM in hardware, the complexity of the FSM would in the best-case be $\mathcal{O}(\lceil log_2(N) \rceil)$ and in the worst-case $\mathcal{O}(N)$, where N is the number of states[8]. In general, state assignment for area and power minimization is not trivial. Nonetheless, at least qualitatively, increasing the number of states, N, increases the complexity of the FSM implementation, and thereby area and power overheads.

In contrast to conventional binary computing [8], saturation arithmetic in stochastic computing does not incur delay overheads to the overall computation. This is due to the fact that the FSM used to realise saturation arithmetic in SC does not involve any kind of accumulation. Instead it computes an output bit for every input bit. It only introduces an additional delay cycle in the initialization of the overall pipelined architecture, which is insignificant. Once that is initialized, the throughput of the overall SC system is unaffected by the introduction of the linear gain FSM.

Assuming no resources are shared, then the saturated adder and inner product introduced in Sections 4.4.1 and 4.4.2 respectively require a single linear gain FSM for each computational unit. That is, each arithmetic unit with saturation employs is own linear gain component. Depending on the specific application, sharing of linear gain FSMs may be possible without trading performance. For example, artificial neural networks are massively parallel systems and activations in a single layer can be computed in parallel to each other without giving rise to race conditions. On the other hand, there is clearly a dependency between activations in different layers. Depending on the implementation methodology, it may be possible for saturation arithmetic units in different layers to share the same linear gain FSMs without disturbing the parallel computation in each layer of the network. Furthermore, it is always possible to employ a single linear gain FSM for all activations in a layer, however such an implementation trades off computation time for hardware area which may be undesired. Clearly such an optimisation problem needs to be further analysed during the design of a hardware system in stochastic computing. Nevertheless, apart from the

[8]This depends on whether binary encoding or one-hot encoding is used. The former requires $\lceil log_2(N) \rceil$ registers whereas the latter requires N registers for N state values. However, one-hot encoding usually tends to simplify the combinatorial circuit associated with the FSM implementation.

qualitative analysis carried out above the particular optimisation problem will not be further considered in the context of this project.

For comparison purposes, the saturated inner product with decomposition introduced in Section 4.4.3 requires $P + 1$ linear gain FSMs assuming that saturation is applied to all the nodes (intermediate and output) and that no sharing is applied. Clearly, this is a significant hardware overhead that needs to be taken into account during hardware design. Note that the smaller intermediate inner products require smaller in size MUX units, however the FSM size is invariant to the number of inputs to the MUX unit. Again, sharing linear gain FSMs between the intermediate results trades off computation time for hardware area.

In summary, saturation arithmetic provides advantages in terms of allowing controlled soft saturation errors that may not significantly affect the overall objective function of the system [7]. However, these advantages are provided at the cost of a somewhat larger circuit compared to the corresponding non-saturated system. In any design involving saturation arithmetic, it is not necessary to use saturation components for all operators, thus care must be taken to select the most appropriate places to saturate signals and also to appropriately select the saturation levels.

Chapter 5

Neural Network Inference in Stochastic Computing

The previous chapter analysed the design, in stochastic computing, of isolated processing elements employed in artificial neural networks. This chapter addresses the design and implementation of neural network inference in stochastic computing. Without loss of generality, emphasis is given on feed forward neural networks, i.e. multi-layer perceptrons.

5.1 Overview

Neural network inference consists of running a forward propagation of a trained neural network to classify, recognise and process unknown input data. Effectively, the network *infers* information about the new data based on the hypothesis function learned during the training phase. Thus, inference cannot occur prior training. Nonetheless, this section assumes that an arbitrary trained feedforward network exists and its parameters (i.e. weights and biases) are known.

Training will most likely occur using floating-point computations, hence the trained model will in general have floating-point inputs, floating-point coefficients and floating-point operators. SC neural network inference consists of a process of converting an existing neural network to a SC compatible model which will have L-bit inputs, compatible coefficient values and stochastic computing operators. This is equivalent to converting the directed acyclic graph of a conventional neural network (like the one in Figure 2.1) to a stochastic computing compatible graph. Note that during inference all of the model's parameters are fixed and coefficient values are determined, hence the conversion process only needs to occur once. After the conversion is finished, the resulting structure can be used to process, in stochastic computing, different data points without altering the hardware configuration. Briefly, given a trained network, SC-based inference requires to

1. Convert floating-point input data to stochastic bit-streams.
2. Convert floating-point network coefficients to stochastic bit-streams.
3. Construct the SC equivalent computational graph for the original neural network.
4. Convert the output of the SC equivalent network to its floating-point representation and evaluate the loss function.

The aim of this chapter is to therefore analyse this conversion process in detail. While emphasis is given on the implementation of multi-layer perceptrons, similar principles apply to other network

architectures such as convolutional networks. Furthermore, although the procedure described here is implemented and evaluated in software through simulation, effort is made to ensure that the proposed neural network architecture can be efficiently implemented in hardware. In general, the goal while constructing the stochastic computing network structure would be to minimize hardware area and power dissipation without affecting the network's recognition accuracy. Nonetheless, as there is no direct way to quantitatively access the incurred hardware resources and power consumption by means of a software simulation, the goal will be to ensure that the implementation of a neural network using stochastic computing maintains adequate performance in terms of network accuracy.

5.2 Overall Network Design

The aforementioned steps involved in the conversion of a conventional network graph to one that is compatible in the context of stochastic computing are analysed in detail in the subsequent sections. While the analysis is heavily based on the individual processing elements presented in the preceded chapter, it is important to consider the entire network as a whole throughout the conversion process.

5.2.1 Input Data Conversion

Conversion from floating-point arithmetic to stochastic arithmetic and vice versa needs to occur at the input and output of the SC computational graph respectively. In the preceded chapters, it was assumed that the inputs to a stochastic computing system lie within $[-1, 1]$. In practice however, it is very unlikely that raw input data values will lie within that range. The MNIST database for example, contains images whose pixels values lie within $[0, 256]$ [28]. Hence the primary inputs of the network need to be proactively down-scaled to constraint them within the representable by stochastic computing range.

In general, given a dataset whose inputs lie in the range $[-l, u]$, where $l \geq 0$ and $u \geq 0$, a down-scale factor of $\max\{l, u\}$ can be applied to ensure that any down-scaled input will lie within $[-1, 1]$. Effectively, this is the *worst-case* scaling, catering for all possible input values so that no saturation error is incurred at the input data conversion. As it is common to use scaling factors that are integer powers of 2, the down-scale coefficient at the network input is selected as follows,

$$s_{in} = 2^{\lceil \log_2(\max\{l,u\}) \rceil} \tag{5.1}$$

Furthermore, this selection is such that any primary input is down-scaled by the same factor. Although it will become more apparent towards the end of this chapter, having a common scaling parameter for all input data points ensures that the scaling coefficients throughout the entire network structure remain constant during inference time, a consequence which can significantly simplify the hardware requirements.

Finally, during neural network inference, the loss function of the model is only computed once at the output of the entire network. In the context of stochastic computing based inference, it will be assumed that the loss function will be computed in floating-point arithmetic. Hence, the labels of the data points do not need to be converted into stochastic bit-streams as they will only be used at the output to evaluate the model's predictions.

5.2.2 Network Coefficients Conversion

Similar to input data values, encoding the model's parameters by means of stochastic bit-streams without any saturation error requires that the trained coefficients lie inside the range $[-1, 1]$, either inherently or after appropriate processing. In contrast however to the primary input values, weights and biases can be scaled individually so that each coefficient can be represented without compression error in stochastic computing. This is because these are trained coefficients and are fixed during inference. Hence, once the scaling of each coefficient is determined it will not change for any input data point.

Following the design and analysis of stochastic processing elements in Section 4.2, the implementation of an inner product in stochastic computing according to Algorithm 2 does not require to convert the weight values into stochastic bit-streams. Instead, their absolute values are used to define a selecting probability distribution over the inputs of the MUX unit. On the other hand, the stochastic adder requires that both inputs are encoded as stochastic bit-streams. Therefore, in terms of simulating the neural network inference within a software environment, only the trained biases need to be converted into stochastic bit-streams, whereas the learned weights can be kept in their floating-point representation. To convert bias coefficients to stochastic bit-streams, each bias term b can be down-scaled by

$$s_{bias} = 2^{\lceil \log_2 |b| \rceil} \tag{5.2}$$

Then the down-scaled coefficient, $b' = \frac{b}{s_{bias}}$, can be represented without any compression error in stochastic computing.

5.2.3 Network Scaling Scheme

In the context of inference, once the scaling factor at the input is fixed as described in Section 5.2.1, the scaling coefficient of every node in the SC equivalent graph can be determined. This is exactly due to the fact that during inference the model's parameters are fixed and known. The process of specifying the scaling of every node in the network is termed *scaling scheme* and is based on the scaling parameter at the output of every individual processing element that is employed in the SC network graph. The scheme proposed in this project is based on forward propagation of known information on data ranges through the network data flow graph. The process is described in the remaining of this section considering feedforward network data flow graphs.

As an illustrative example, consider the network graph shown in Figure 5.1. The network has four inputs, $x_1, ..., x_4$, a single hidden layer with two units and a single output y. Next, consider a single data point $\boldsymbol{x} \in \mathbb{R}^4$ and assume the weight matrices in the hidden and output layer are given by $\boldsymbol{W}^{(1)} \in \mathbb{R}^{4\times 2}$ and $\boldsymbol{W}^{(2)} \in \mathbb{R}^{2\times 1}$ respectively. Furthermore, the biases are given by $\boldsymbol{b}^{(1)} \in \mathbb{R}^2$ and $b^{(2)} \in \mathbb{R}$ for the hidden and output layers respectively. The activations in the hidden layer are therefore calculated as,

$$\boldsymbol{h}^{(1)} = \phi(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$

and the output of the network is given by

$$y = \phi(\boldsymbol{W}^{(2)T}\boldsymbol{h}^{(1)} + b^{(2)})$$

where ϕ is the activation function.

To start with, assuming that the input data values x_i lie in some interval $[-l, u]$, the input scaling factor s_{in} is selected according to (5.1). Thus, every input x_i is scaled by s_{in} and the down-scaled inputs $x'_i = x_i/s_{in}$ are converted into stochastic bit-streams.

Next, consider the first activation $h_1^{(1)}$ in the hidden layer. This is computed as follows,

$$h_1^{(1)} = \phi\Big(\underbrace{\sum_{i=0}^{3} W_{i,1}^{(1)} \cdot x_i}_{\text{Inner Product}} \underbrace{+b_1}_{\text{Addition}} \Big) \tag{5.3}$$

The inner product between the weights and the down-scaled input values can be calculated in stochastic computing using the implementation given in Algorithm 2. The output will be a stochastic bit-stream representing the down-scaled weighted sum. Recall that the output of Algorithm 2 is associated to a scaling coefficient $w_{sum} = \sum_{i=0}^{3} |W_{i,1}^{(1)}|$. Hence, the inner product between $\{W_{i,1}^{(1)}\}$ and $\{x_i\}$ in (5.3), when computed in stochastic computing will be down-scaled by a factor of $s_{in} \times w_{sum}$. Note that $s_{in} \times w_{sum}$ is fixed for all data points and can be computed in advance as the matrix $\boldsymbol{W}^{(1)}$ is known. Finally, to maintain consistency throughout the network structure, the output of the stochastic inner product is *re-scaled* accordingly so that the scaling factor associated with it is given by

$$s_{dot} = 2^{\lceil \log_2(s_{in} \times w_{sum}) \rceil} \tag{5.4}$$

that is, the next integer power of 2 of $s_{in} \times w_{sum}$. Since $s_{dot} \geq s_{in} \times w_{sum}$, the aforementioned re-scaling of the inner product output can be realised by means of a XNOR multiplier with inputs the inner product output and a bit-stream representing $\frac{s_{in} \times w_{sum}}{s_{dot}} \leq 1$. This is easy to implement in hardware and does not incur significant hardware or delay overheads.

The addition of the bias term must be handled with care. This is because the two inputs to the stochastic adder may not have a common scaling. In this example, the two bit-streams to be added are the output of the stochastic inner product and the bit-stream representing the down-scaled bias term b_1'. The former is down-scaled by s_{dot} whereas the latter is down-scaled by s_{bias} as given by (5.2). The addition is meaningful if and only if the two bit-streams to be added are down-scaled by the same factor. To illustrate this concept consider the following example. Let $x \in [-M, M]$ where $M > 0$ and $y \in [-L, L]$ where $L > 0$. The down-scaled numbers $x' = x/M$ and $y' = y/L$ both lie within $[-1, 1]$. The addition of x' and y' involves computing the least common multiple (LCM) and appropriately re-scaling each number so that both operands have a common divisor. In the case where both M and L are integer powers of 2, the LCM is equal to $\max\{M, L\}$. Hence,

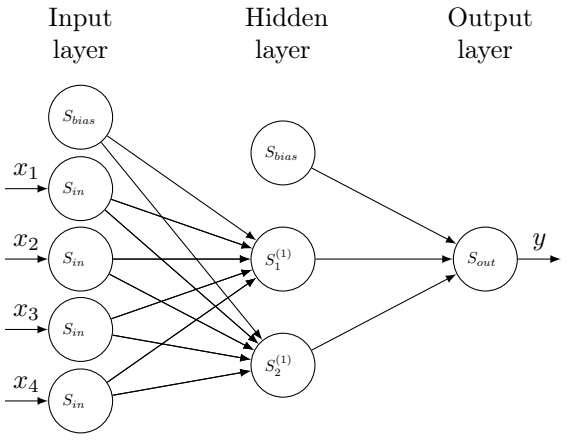


Figure 5.1: A neural network data flow graph

x' and y' are re-scaled as follows,

$$x' \leftarrow x' \times \frac{M}{\max\{M, L\}} \quad \text{and} \quad y' \leftarrow y' \times \frac{L}{\max\{M, L\}}$$

After re-scaling, both x' and y' still lie within $[-1, 1]$ and have a common scaling factor. Thus, their addition is meaningful and can be realised in stochastic computing.

Going back to the bias addition, the two bit-streams at the input of the stochastic adder are re-scaled accordingly so that both have a common scaling factor, namely $m_1^{(1)} = \max\{s_{dot}, s_{bias}\}$. In fact, only one of the two inputs needs to be re-scaled and this can be determined in advance as both s_{dot} and s_{bias} can be computed once the network's parameters are given. Thus, only one XNOR multiplier needs to be placed at one of the inputs of the scaled adder. This will be the input associated to the scaling factor that is not equal to $\max\{s_{dot}, s_{bias}\}$ and the re-scaling factor will be equal to $s_{dot}/m_1^{(1)}$ or $s_{bias}/m_1^{(1)}$. As a side remark, the most likely scenario is that $s_{dot} > s_{bias}$, thus the bit-stream that needs to be re-scaled will the one representing the down-scaled bias coefficient b'. Finally, the output of the scaled adder representing the quantity $\sum_{i=0}^{3} W_{i,1}^{(1)} \cdot x_i + b_1$ will be down-scaled by $s_1^{(1)} = 2 \times m_1^{(1)}$.

The final thing to note is that the activation function[1] does not affect the scaling coefficient of the bit-stream. Therefore, given that no saturation arithmetic is used, the output of the MUX-based neuron is down-scaled by a factor of $s_1^{(1)}$. Despite considering $h_1^{(1)}$ in isolation, exactly the same analysis can be applied for $h_2^{(1)}$ to obtain $s_2^{(1)}$. Note that in general $s_1^{(1)} \neq s_2^{(1)}$.

As already mentioned, the proposed scaling scheme is based on forward propagation of data ranges throughout the network graph. So when it comes to compute the output of the network in Figure 5.1 the scaling coefficients $s_1^{(1)}$ and $s_2^{(1)}$ are now considered as the scaling factors of the (transformed) input data and the preceded analysis is repeated using the corresponding parameters of the output layer. That is, $h_1^{(1)} \in [-s_1^{(1)}, s_1^{(1)}]$ and $h_2^{(1)} \in [-s_2^{(1)}, s_2^{(1)}]$ can be considered to be the inputs to the output layer with coefficients given by $\boldsymbol{W}^{(2)} \in \mathbb{R}^{2\times 1}$ and $b^{(2)} \in \mathbb{R}$. The output of the network y is given by

$$y = \phi\Big(\underbrace{\sum_{i=0}^{1} W_i^{(2)} \cdot h_i^{(1)}}_{\text{Inner Product}} \underbrace{+b_2}_{\text{Addition}} \Big) \tag{5.5}$$

The main difference is that now the inputs to the stochastic inner product $\{h_i^{(1)}\}$ may not in general have the same scaling factor. The weighted sum of the activations is meaningful if and only if all the scaling coefficients at the input of the MUX-based inner product are the same. This was not an issue at the input layer as the selection of s_{in} according to (5.1) was such that s_{in} is common for all input data points. However, in any subsequent layer it is not guaranteed that the inputs to the inner product will be associated to a common scaling parameter. As analysed above, the LCM can be selected as the common scaling factor and all the input bit-streams that are down-scaled by a factor different than that should be re-scaled accordingly so that their new scaling coefficient is the LCM. Since all scaling coefficients are integer powers of 2, the LCM is given by the maximum of $\{s_i^{(1)}\}$ and all input bit-streams should be re-scaled[2] such that

$$h_i^{(1)} \leftarrow h_i^{(1)} \times \frac{s_i^{(1)}}{\text{LCM}} \quad \forall i \tag{5.6}$$

Going back to the current example, the LCM is given by $\max\{s_1^{(1)}, s_2^{(1)}\}$ and all the input

[1] Assuming one of the sigmoid, hyperbolic tangent or ReLU is used

[2] As $s_i^{(1)} \leq \text{LCM}$ for all i, the re-scaling can be realised by means of a XNOR multiplier

bit-streams should be re-scaled according to (5.6). The output of the MUX-based inner product will be down-scaled by $\max\{s_1^{(1)}, s_2^{(1)}\} \times w_{sum}$ where $w_{sum} = \sum_{i=0}^{1} |W_i^{(2)}|$. Once again, to ensure consistency throughout the network graph, the output bit-stream is re-scaled accordingly so that its scaling parameter is given by

$$s_{dot} = 2^{\lceil \log_2(\max\{s_1^{(1)}, s_2^{(1)}\} \times w_{sum}) \rceil} \tag{5.7}$$

Thereinafter, the addition of the bias term $b^{(2)}$ is done by finding the $\max\{s_{dot}, s_{bias}\}$ and re-scaling one of the two bit-streams in the same way as in the previous layer. Finally, the scaling at the output of the entire network is given by $s_{out} = 2 \times \max\{s_{dot}, s_{bias}\}$, implying that the output is such that $y \in [-s_{out}, s_{out}]$.

Although the proposed scaling scheme is introduced in terms of a small-scale example, it can be easily extended to feedforward networks of arbitrary depth and width. In summary, the proposed scheme consists of two main procedures. The one associated with the MUX-based inner product between weights and features, and the one associated with the addition of the bias term. These are summarised, in their most general form, in Algorithms 3 and 4. The re-scaling coefficients are used at the input and output of stochastic accumulators to appropriately re-scale the corresponding bit-streams using XNOR multipliers[3].

Effectively, the scaling coefficients at every node of the network specify the range of values that the corresponding signal can take and are the same for all data points. This is because both the input scaling parameter selected according to (5.1) but also the scaling at the output of each individual processing unit are worst-case scalings. Hence, the procedures given in Algorithms 3 and 4 need to be executed only during the construction of the SC network graph and the re-scaling multipliers need to be inserted wherever is needed. Once these actions are done, the SC-based neural network graph is completed and remains fixed during inference run time. As briefly discussed in Section 5.2.1 this is a desired consequence as it allows the hardware infrastructure to remain fixed during run time.

Some simplifying modifications could be made to the proposed scaling scheme. One of them could be to enforce the scaling coefficients of all the activations in a specific layer to be the same[4]. This could be done both at the output of the MUX-based inner product but also after the bias addition. A viable option would be to choose the largest scaling along the layer and appropriately

Algorithm 3: Inner product scaling scheme

Input : Number of inputs $m \in \mathbb{R}$
Weight coefficients $\boldsymbol{w} \in \mathbb{R}^m$
Input scaling factors $\boldsymbol{s} \in \mathbb{R}^m$
Output: Input re-scaling coefficients $\boldsymbol{r}_{in} \in \mathbb{R}^m$
Output re-scaling coefficient $r_{out} \in \mathbb{R}$
Output scaling factor $s_{dot} \in \mathbb{R}$

1 $s_{max} = \max\{s_0, s_1, ..., s_{m-1}\}$
2 **for** $i = 0$ **to** $m - 1$ **do**
3 $\quad r_{in_i} = \frac{s_i}{s_{max}}$
4 $w_{sum} = \sum_{i=0}^{m-1} |w_i|$
5 $s_{dot} = 2^{\lceil \log_2(s_{max} \times w_{sum}) \rceil}$
6 $r_{out} = \frac{s_{max} \times w_{sum}}{s_{dot}}$

[3]Re-scaling components should be omitted for the nodes where $r = 1$

[4]Depending on the trained coefficients and network architecture this may in fact be true without any modification

Algorithm 4: Bias addition scaling scheme

Input : Bias scaling factor $s_{bias} \in \mathbb{R}$
Input scaling factor $s_{dot} \in \mathbb{R}$
Output: Input re-scaling coefficients r_{bias}, $r_{dot} \in \mathbb{R}$
Output scaling factor $s_{out} \in \mathbb{R}$

1 $s_{max} = \max\{s_{bias}, s_{dot}\}$
2 $r_{dot} = \frac{s_{dot}}{s_{max}} \quad r_{bias} = \frac{s_{bias}}{s_{max}}$
3 $s_{out} = 2 \times s_{max}$

re-scale the bit-streams that have a different scaling coefficient. The advantage of this modification is that the inputs to any MUX-based inner product will have a common scaling coefficient associated with them. Thus, no re-scaling multipliers will be needed prior the accumulation unit. As the re-scaling multiplier at the output of the inner product is needed anyway (to increase the scaling to the next power of 2), this modification reduces the number of XNOR multipliers that are needed. Finally, note that this amendment does not eliminate the need to re-scale the inputs of the stochastic adder that performs the addition of the bias term.

The final issue to consider would be to analyse how the aforementioned scaling scheme changes if saturation arithmetic is employed. Recall that saturation arithmetic, employed in the context of stochastic computing, introduces a linear gain block with saturation at the output of a stochastic accumulator to up-scale the resulting bit-stream and reduce the scaling coefficient associated with it. Starting from the input layer, the scaling coefficients of the primary inputs will not be affected by saturation as they are dictated by the dataset range. Thereinafter, the designer has the option to introduce linear gain blocks at the output of any MUX-based inner product or MUX-based adder to up-scale (and saturate) their outputs and thereby reduce the corresponding scaling by an amount equal to the gain that is applied. Thus, the analysis illustrated above still holds and the only modification that one needs to make is to decrease the scaling factor of a signal to which a linear gain of G is applied exactly by an amount equal to G. Beyond this modification, the same scheme applies by propagating known information on data ranges through the network graph.

Finally, in the event where saturation arithmetic is employed, appropriate saturation levels need to be determined. A standard approach when creating a saturation arithmetic implementation, is to determine saturation levels through simulation. For the purpose of neural network inference, this is described as follows. Test data are applied to the network, and the peak value reached by each signal is recorded. Internal scalings, and thereby saturation levels, are then selected to ensure that the full dynamic range afforded by the signal representation would be used under excitation with the given input vectors [7].

5.2.4 Output Data Conversion

Once the output signals are computed, conversion from stochastic arithmetic to floating-point arithmetic can be achieved following the procedure described in Section 4.1.2. Each conversion outcome will be intrinsically down-scaled by the scaling associated to the corresponding bit-stream. Thus, the outcome of the conversion must be up-scaled, in floating-point, by an amount equal to the scaling coefficient of the signal.

5.2.5 Remarks

Although viable, data range propagation, even with saturation arithmetic, can sometimes be overly pessimistic, accommodating the full dynamic range of possibilities thus loosing precision. To

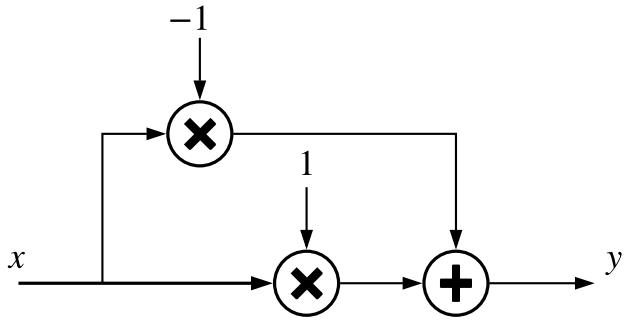


Figure 5.2: A trivial computational graph illustrating deficiencies of data range propagation

illustrate the concept, consider the data flow graph in Figure 5.2. Let $x \in [-1, 1]$, then the full dynamic range of the output, determined by data range propagation through the graph is $[-2, 2]$. However, the output is always going to be equal to zero. Although trivial, the example illustrates the aforementioned argument which depending on the application may cause issues to the design. Nonetheless, the proposed scaling scheme is functional, relatively simple and can be realised in hardware without incurring significant hardware overheads in terms of area, delay and power dissipation.

Chapter 6

Training Stochastic Computing Neural Networks

The last chapter described a procedure for transforming the computational graph of a trained network into a new graph that can be efficiently implemented using stochastic computing hardware. The major issue of the approach presented in Chapter 5 is that the network is trained using floating-point computations, assuming no limitations regarding the representation of numbers in stochastic computing. These are then imposed by the designer, as described in Chapter 5, to ensure the network graph is compatible in the context of stochastic computing, and SC hardware is used only for inference. This approach can sometimes be problematic as the dynamic range of stochastic computing is relatively small, compelling the designer to impose large down-scale coefficients throughout the network graph. For deep networks, the scaling coefficients can increase significantly leading to poor precision. This chapter addresses this issue by taking into consideration the limitations of stochastic arithmetic during the training phase of the network. The modified training procedure is termed *stochastic computing compatible training*.

Section 6.1 gives a brief overview and further sets the scene. A neuron architecture that models the features of stochastic computing during training is proposed in Section 6.2. Thereinafter, methods for constraining trainable parameters during the stochastic computing compatible training process are presented in Section 6.3. Finally, Section 6.4 proposes an optimization technique for learning optimal saturation levels during neural network training.

6.1 Overview

Supervised machine learning algorithms are usually trained by means of minimizing the loss function that is associated with the model. In the context of an optimisation problem[1], the loss function $\mathcal{J}(\boldsymbol{w}, \boldsymbol{b})$ is the objective function of the problem and the model's parameters $\boldsymbol{w}$ and $\boldsymbol{b}$ are the decision variables. These are essentially the parameters modified by the training algorithm in order to minimize the loss $\mathcal{J}$. Therefore training, in its simplest form, can be formulated as

$$\underset{\boldsymbol{w},\boldsymbol{b}}{\text{minimize}} \quad \mathcal{J}(\boldsymbol{w}, \boldsymbol{b}) \tag{6.1}$$

As briefly discussed in Section 2.2, the problem in (6.1) is usually solved by means of an optimization algorithm such as the gradient method. The selection of the optimization algorithm is a significant

[1] Without loss of generality optimisation is associated to minimization

design choice as it can notably dictate the performance of the model. Typical considerations that are encountered in the selection are the convergence of the algorithm, the speed of convergence and the computational complexity of the method. For example, Newton's method is known to have a very high speed of convergence. However, it requires to compute the inverse of the Hessian of $\mathcal{J}$ in every iteration of the algorithm, a highly intensive task especially in the context of training neural networks where the number of decision variables is very large. Thus, first-order gradient-based optimization algorithms are usually employed. Nevertheless, a detailed investigation on the design and analysis of optimization algorithms for training neural networks is beyond the scope of this project.

Following the implementation of neural network inference in stochastic computing, the main objective of this chapter is to capture the limitations of stochastic arithmetic during the training phase of the network, by means of a modified training process. Limitations include the small dynamic range of stochastic arithmetic and thereby the necessity to impose large scalings throughout the network graph to ensure numbers are constrained within $[-1, 1]$. These limitations were neglected in the preceded chapter where the network was trained with floating-point computations while being in ignorance of the subsequent implementation of inference in stochastic computing. Thus, the goal of stochastic computing compatible training is to train a network that is aware of the implementation of inference using SC hardware and appropriately modifies the model's parameters to mitigate the limitations of the stochastic representation.

Two of the main implications of using stochastic arithmetic in the context of a neural network implementation are:

- No number outside the range $[-1, 1]$ can be represented without proactive down-scaling
- Stochastic computing implements scaled addition and scaled inner product

To train a SC compatible network, these implications should be incorporated into the training process. Model's parameters should be restricted in a way so that they can be represented in the stochastic domain without proactive down-scaling, i.e. they lie inside the range $[-1, 1]$. Furthermore, scaled operations imply that the gradient of both the addition and inner product when computed in stochastic computing, will be different compared to the gradient of the conventional operations. from the gradient of the corresponding conventional operation. Hence, the gradients need to be suitably modified during the training process. The remaining of this chapter is therefore taken up with the problem of incorporating the aforementioned limitations of stochastic computing within the training phase of a network.

Although implementation details are given in Chapter 7, the training of neural networks is in general a computationally intensive task. At the same time, simulating the operation of stochastic computing in a software environment can itself incur long computational times especially if a large number of computations need to be performed. Therefore, to avoid excessively long computational periods while simulating in software the training of a neural network using stochastic computing, the proposed modified training procedure will be performed using floating-point computations in a software framework and an appropriate model that captures the features and limitations of stochastic computing will be used instead.

6.2 Stochastic Computing Training Model

The objective of this section is to develop a modified neuron architecture that models the main features of stochastic computing while performing computations using floating-point arithmetic.

That is, the dynamic range of the model should be restricted to that of stochastic computing and scaled addition should be used. Additionally, the error and uncertainty introduced by stochastic arithmetic need to be modelled as well. The aforementioned features are somewhat independent to each other, thus they are studied separately starting from the uncertainty in the computations introduced by stochastic arithmetic.

6.2.1 Modelling the Uncertainty of Stochastic Computing

By its nature, the paradigm of computing on stochastic bit-streams introduces error and uncertainty. To make stochastic computing compatible training as realistic as possible, these errors should be incorporated into the model. In stochastic computing, a number is represented by a bit-stream and its value is determined by the probability of an arbitrary bit in the sequence being one. At the output of the SC neural network, the conversion from stochastic arithmetic to floating-point arithmetic involves accumulating the bits in the stochastic sequence to essentially compute the ratio of 1s in the bit-stream to the length of the bit-stream.

In general, a stochastic bit-stream X is assumed to be a Bernoulli process. Hence, the random bits X_i are independent and identically distributed (i.i.d) binary random variables. That is, $P(X_i = 1) = p$ for all $i = 1, ..., L$ and $p \in [0, 1]$ the probability of an arbitrary bit being one. The stochastic to floating-point conversion involves computing the number of 1s in the sequence, i.e. $S_L = X_1 + X_2 + ... + X_L$. The random variable S_L follows a Binomial distribution with parameters L and p. Of particular interest is the value of S_L as the length of the bit-stream tends to infinity (i.e. the bit-stream is sufficiently long). By the central limit theorem (CLT), the sum of i.i.d random variables, $S_L = X_1 + X_2 + ... + X_L$, tends to a Normal random variable as $L \to \infty$. Thus, for sufficiently long stochastic bit-streams, the uncertainty due to random fluctuations in the output signals can be modelled by adding *Gaussian noise* to the floating-point representation of the result. A Gaussian distribution, is completely characterised by its mean and variance. The remaining of this section is therefore taken up with the problem of determining appropriate values for the mean and variance of the Gaussian distribution.

Starting with the mean of the distribution, as the representation of a number using stochastic arithmetic provides an unbiased estimate of the number itself, the additive Gaussian noise should have mean value equal to zero. That is, the addition of Gaussian noise should not alter the property of the stochastic representation being unbiased. On the other hand, the variance of the additive Gaussian noise has a physical interpretation in the context of stochastic computing. Recall from Sections 2.4.2 and 4.1.1 that both the error due to random fluctuations in the bit-streams as well as the quantization error due to rounding to digital probability values are inversely proportional to the bit-stream length L. Since the representation of a number in stochastic arithmetic is not biased, the error incurred by stochastic computing is directly related to the variance of the estimate provided by the stochastic representation. Increasing the bit-stream length L, reduces both sources of error, i.e. random fluctuations and quantization errors. Therefore, the variance of the zero-mean Gaussian noise used to model the uncertainty of stochastic computing is inversely proportional to the bit-stream length. Intuitively, one can reinforce this statement by considering the following argument. If the bit-stream length L becomes infinite, the error incurred by encoding any number as a stochastic bit-stream is effectively zero thus no perturbation, in terms of additive Gaussian noise, should be imposed to the result.

In conclusion, to model the uncertainty of stochastic computing while performing computations in floating-point arithmetic, zero-mean Gaussian noise can be added to the floating-point output of the entire neural network. The variance of the noise is inversely proportional to the bit-stream

length that would have been used in an actual SC neural network implementation. Thus, small variance models long stochastic bit-streams whereas high variance corresponds to small, in length, bit-streams.

6.2.2 Modelling Scaled Computations

Another major implication of stochastic computing in the context of training a neural network is that its dynamic range is limited to $[-1, 1]$, thus addition and inner product are performed in a scaled fashion. Addition of two numbers $x, b \in [-1, 1]$ in stochastic computing is down-scaled by 2, whereas the weighted sum given by $\sum_i w_i x_i$ where $x_i \in [-1, 1]$ for all i, when implemented in stochastic computing is down-scaled by[2] $\sum_i |w_i|$. Calculating the addition and/or inner product result using floating-point computations and post processing (i.e. down-scaling) the result is rather easy to implement in software and does indeed capture the down-scaling effect of stochastic operations during the forward propagation of the network. Furthermore, note that if the inputs of the network lie within $[-1, 1]$, down-scaling the result of each floating-point addition and inner product computation by the aforementioned scaling coefficients also ensures that every activation in the network will lie inside the representable by stochastic computing range.

Training a neural network using first-order gradient-based optimization algorithms, requires to compute the partial derivative of the loss function with respect to every decision variable. As explained in Section 2.2.6, the backpropagation algorithm provides an efficient way for calculating the gradient of the loss with respect to every variable in the network. However, the gradient of a scaled computation is not the same as the gradient of the corresponding conventional operation. Consider, for example, the scaled addition of two numbers $x, b \in [-1, 1]$. That is, $y = x \oplus b = (x + b)/2$. The partial derivative of y with respect to b is given by

$$\frac{\partial y}{\partial b} = \frac{1}{2}$$

whereas the partial derivative of the conventional addition $y = x + b$ with respect to b is equal to 1. Similarly, the gradient of the scaled inner product, as computed by stochastic computing, is different from the gradient of the corresponding conventional operation. Consider the scaled inner product given by

$$y = \boldsymbol{x} \odot \boldsymbol{w} = \frac{1}{\sum_i |w_i|} \sum_i x_i w_i$$

where $x_i \in [-1, 1]$ for all i. The partial derivative of y with respect to w_i is given by[3]

$$\frac{\partial y}{\partial w_i} = \frac{x_i \sum_i |w_i| - \partial |w_i| \sum_i x_i w_i}{\left(\sum_i |w_i|\right)^2}$$

On the other hand, the partial derivative of the conventional weighted sum, $y = \sum_i x_i w_i$, with respect to w_i is equal to x_i. Clearly, there is a difference between the partial derivatives of the scaled operations implemented in stochastic computing and the conventional operations implemented in floating-point arithmetic.

[2] Further post processing the result, as described in Chapter 5, to ensure all scaling coefficients are integer powers of 2 of can be neglected for training purposes.

[3] $\partial |w_i|$ denotes the subdifferential of the function $f(w_i) = |w_i|$ and is defined as $\partial |w_i| = \begin{cases} 1, & w_i > 0 \\ [-1, 1], & w_i = 0 \\ -1, & w_i < 0 \end{cases}$

In summary, post processing the floating-point computation with the appropriate scaling coefficient does indeed model the down-scaling effect of stochastic computing during the forward propagation of the network. However, this does not take into account the discrepancy between the gradients of conventional operations implemented in floating-point arithmetic and the scaled operations implemented in stochastic arithmetic. Although it may seem unclear at this stage, depending on the software framework that is used to deploy and train the neural network, it may indeed be possible to model the effect of gradient-discrepancy due to stochastic computing by simply post processing the result computed using floating-point computations. Further clarification and implementation details are provided in Chapter 7.

The amended neuron architecture used to model SC-based computations is shown in Figure 6.1. Note that operation symbols in the figure, refer to conventional (i.e. non-scaled) computations and numbers are represented in floating-point arithmetic. Furthermore, following the scaling scheme analysis for the forward propagation of a network graph in Chapter 5, it is important to note that for the addition operations in Figure 6.1 to be meaningful, the inputs to any adder must have a common scaling factor. The scaling scheme proposed in the preceded chapter can be employed for training purposes as well. Thus, the inputs to the inner product should be re-scaled appropriately so that they have a common scaling coefficient associated with them[4]. In addition, the bias term b should be re-scaled as well to ensure the addition is meaningful[5].

6.2.3 Modelling Saturation Arithmetic

The proposed SC neuron architecture can be further modified to model the effect of saturation arithmetic in stochastic computing. Recall that saturation arithmetic in stochastic computing employs linear gain blocks with saturation at the output of a MUX-based inner product and/or MUX-based adder to up-scale the result and reduce the associated scaling coefficient. Such a behaviour can be incorporated in the model proposed above by simply applying a gain factor after the down-scaling nodes and saturating the result to ± 1. The modified neuron architecture is illustrated in Figure 6.2. Once again, the operation symbols in Figure 6.2 correspond to conventional, non-scaled, arithmetic operations. Finally, the aforementioned scaling considerations still apply and the scaling scheme proposed in Chapter 5 can be employed to ensure addition operations are meaningful.

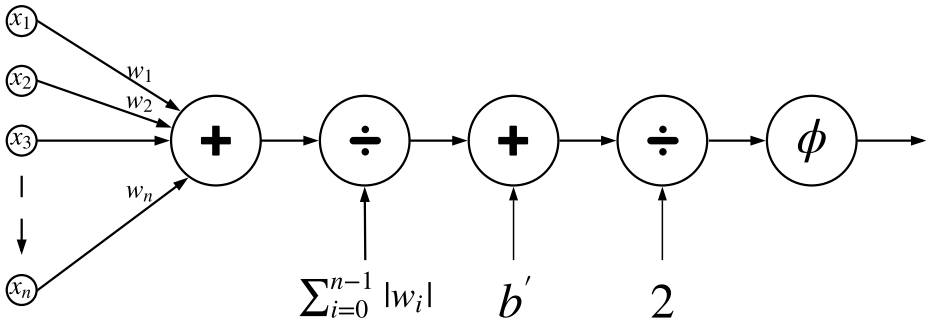


Figure 6.1: Stochastic computing compatible neuron architecture

[4]The maximum scaling can be selected as the common scaling coefficient

[5]b' denotes the re-scaled bias coefficient

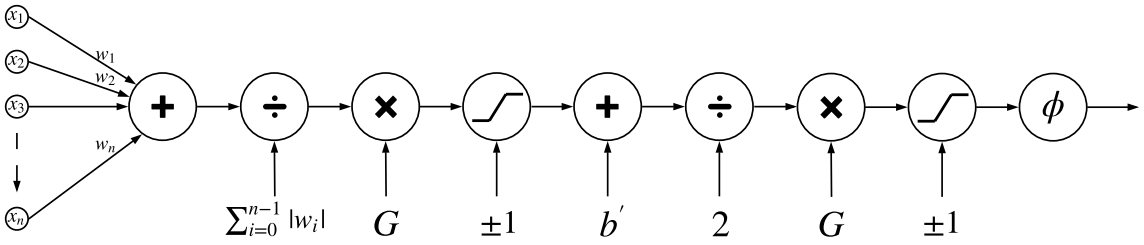


Figure 6.2: Stochastic computing compatible neuron architecture employing saturation arithmetic

6.3 Learning Stochastic Computing Compatible Coefficients

By its nature, stochastic computing can only represent numbers inside the range $[-1, 1]$. Thus, any number with absolute value larger than one, needs to be proactively down-scaled and the down-scaled quantity is then represented in terms of a stochastic bit-stream. In the context of neural network training, the inputs to the network are not controlled by the designer and are instead dictated by the nature of the data. On the other hand, network coefficients are trainable parameters and their value is determined by the optimization algorithm used to minimize the loss function. Stochastic computing compatible training aims to modify the training procedure in such a way, so that the optimal network coefficients can be represented in stochastic arithmetic without proactive down-scaling. Thereby, optimal coefficients should have absolute value less than one. The objective can be formulated as a constrained optimization problem of the form[6]

$$\begin{aligned} \underset{\boldsymbol{w},\boldsymbol{b}}{\text{minimize}} \quad & \mathcal{J}(\boldsymbol{w},\boldsymbol{b}) \\ \text{subject to} \quad & -1 \leq \boldsymbol{w} \leq 1 \\ & -1 \leq \boldsymbol{b} \leq 1 \end{aligned} \tag{6.2}$$

Solving in closed form problems of the form (6.2) for thousands of decision variables is clearly impossible. In practice, such problems are usually solved by transforming the constraint optimization problem into one or more unconstrained problems. The main idea in these kind of transformations is to augment the objective function $\mathcal{J}(\boldsymbol{w},\boldsymbol{b})$ with a penalty function, $\mathcal{P}(\boldsymbol{w},\boldsymbol{b})$, that penalizes non-admissible values for $\boldsymbol{w}$ and $\boldsymbol{b}$. Consider the problem (6.2), the penalty function,

$$\mathcal{P}(\boldsymbol{w},\boldsymbol{b}) = \begin{cases} 0, & \text{if } \boldsymbol{w} \text{ and } \boldsymbol{b} \text{ are admissible} \\ +\infty, & \text{otherwise} \end{cases} \tag{6.3}$$

and the function

$$\mathcal{J}_\alpha = \mathcal{J}(\boldsymbol{w},\boldsymbol{b}) + \mathcal{P}(\boldsymbol{w},\boldsymbol{b}) \tag{6.4}$$

Clearly, the unconstrained optimization of $\mathcal{J}_\alpha$ yields a solution of the problem (6.2). However, because of its discontinuous nature, the minimization of $\mathcal{J}_\alpha$ is rather impractical. Nevertheless, it is possible to construct penalty functions that approximate the behaviour of the function in (6.3). The remaining of this section analyses and proposes different penalty functions that can be used to approximately solve (6.2).

As a side remark, the methods that will be proposed do not actually realise (6.3) in an exact way. In fact some of them are not even designed for the purpose of solving (6.2). Hence, there is no guarantee that the solution obtained by employing such penalty functions will indeed be optimal.

[6]The conditions $-1 \leq \boldsymbol{w} \leq 1$ and $-1 \leq \boldsymbol{b} \leq 1$ have to be understood element-wise, i.e. $-1 \leq w_i \leq 1$ and $-1 \leq b_i \leq 1$ for all i.

In some situations, it may not even be admissible, i.e. some coefficients may have absolute value larger than one. Nevertheless, training of neural networks is never performed in an exact manner. The non-linearity of the model causes most loss functions to become non-convex, thus a globally optimal solution is usually never achieved and an approximate solution to the optimization problem may indeed yield satisfactory performance.

6.3.1 L^2 Regularization

A possible way to penalize large coefficient values is to consider the penalty function given by

$$\mathcal{P}(\boldsymbol{w}, \boldsymbol{b}) = \lambda(\boldsymbol{w}^T\boldsymbol{w} + \boldsymbol{b}^T\boldsymbol{b}) = \lambda\big(\sum_i w_i^2 + \sum_i b_i^2\big) \tag{6.5}$$

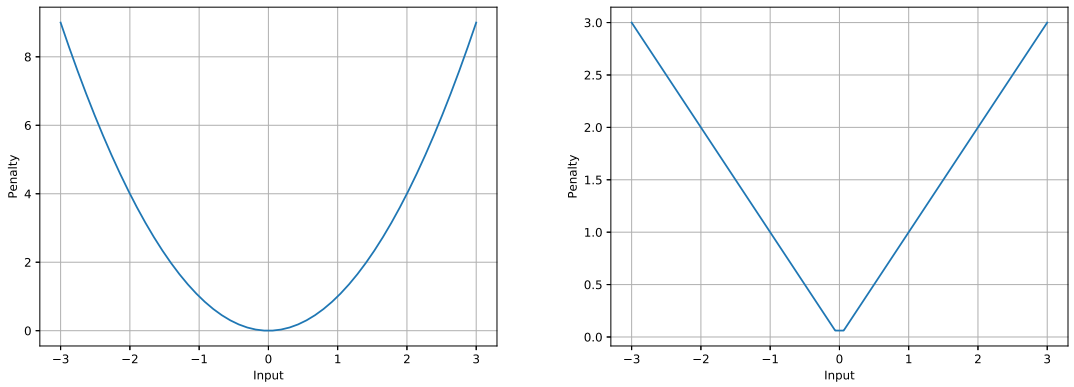
for which the augmented loss function is defined as,

$$\mathcal{J}_\alpha(\boldsymbol{w}, \boldsymbol{b}) = \mathcal{J}(\boldsymbol{w}, \boldsymbol{b}) + \lambda(\boldsymbol{w}^T\boldsymbol{w} + \boldsymbol{b}^T\boldsymbol{b}) \tag{6.6}$$

The penalty function given in (6.5) is equivalent to the L^2 norm[7] of the parameters $\boldsymbol{w}$ and $\boldsymbol{b}$, scaled by $\lambda \in [0, \infty)$. In machine learning literature [18], the L^2 norm penalty function is widely known as L^2 *regularization* or *weight decay*. In general, regularization is a collection of strategies used to reduce the test error of a learning algorithm and thereby avoid overfitting [18]. In fact, they have been used for decades prior the advent of deep learning. The majority of regularization techniques are based on limiting the capacity of models, by adding a parameter penalty $\mathcal{P}(\boldsymbol{w}, \boldsymbol{b})$ to the objective function $\mathcal{J}$. Hence, the augmented loss function is also referred to as the regularized objective function $\mathcal{J}_\alpha$.

The parameter λ is a hyperparameter that weights the relative contribution of the norm penalty, $\mathcal{P}$, relative to the standard objective function $\mathcal{J}$. Setting $\lambda = 0$ results in no regularization, whereas larger values of λ emphasise the regularization term. Therefore, when the training algorithm minimizes the regularized objective $\mathcal{J}_\alpha$ it will decrease the original objective $\mathcal{J}$ on the training data but also a measure of the size of the parameters $\boldsymbol{w}$ and $\boldsymbol{b}$.

As shown in Figure 6.3a, L^2 regularization drives the weights closer to the origin by penalizing



(a) L^2 Penalty function

(b) L^1 Penalty function

Figure 6.3: Regularization penalty functions for scalar quantities

[7]For a real number $p \geq 1$, the L^p norm of $\boldsymbol{x} \in \mathbb{R}^n$ is given by $\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{\frac{1}{p}}$

large weight values, and hence termed weight decay. Thus, for a suitable value of λ[8], L^2 regularization will constraint the parameters $\boldsymbol{w}$ and $\boldsymbol{b}$ in the representable by stochastic computing range $[-1, 1]$, which is one of the objectives of stochastic computing compatible training process.

Goodfellow et al. [18], suggest to use a penalty function $\mathcal{P}$ that penalizes only the weights of the affine transformation in each layer and leaves the biases unregularized. The motivation for this is twofold. Firstly, the biases typically require less data to fit accurately than the weights. Each weight specifies how two variables interact, thus fitting the weight well requires observing both variables in a variety of conditions. On the other hand, each bias controls only a single variable, hence unregularized biases do not induce too much variance. Secondly, in some cases, regularizing the bias parameters can introduce underfitting. In the context of stochastic compouting compatible training, the objective is to constraint both the weights and biases of the network within $[-1, 1]$, so that no proactive down-scaling is required. Nevertheless, in the event where bias regularization degrades the recognition capacity of the model, it should be omitted to rehabilitate performance. It is entirely up to the designer to decide whether it would be more advantageous to regularize or not the biases, but also to determine an appropriate value for the hyperparameter λ.

6.3.2 L^1 Regularization

While L^2 weight decay is one of the most common ways of regularizing model coefficients, it can be seen from Figure 6.3a that the penalty value follows a quadratic growth. Although this type of penalty function may still restrict the coefficients to have absolute value less than one, in principle there is no reason why such a quadratic growth in the penalty value should be used. A different type of norm penalty function for which the penalty value increases in a linear fashion as the absolute value of the coefficient deviates from zero is the L^1 penalty function, also known as L^1 regularization.

Formally, L^1 regularization on the model's parameters $\boldsymbol{w}$ and $\boldsymbol{b}$ is defined as

$$\mathcal{P}(\boldsymbol{w}, \boldsymbol{b}) = \|\boldsymbol{w}\|_1 + \|\boldsymbol{b}\|_1 = \sum_i |w_i| + \sum_i |b_i| \tag{6.7}$$

and the corresponding regularized objective is given by

$$\mathcal{J}_\alpha(\boldsymbol{w}, \boldsymbol{b}) = \mathcal{J}(\boldsymbol{w}, \boldsymbol{b}) + \lambda(\|\boldsymbol{w}\|_1 + \|\boldsymbol{b}\|_1) \tag{6.8}$$

Although L^1 regularization can be rigorously analysed to determine its effect on model coefficients [18], by inspecting the penalty function in Figure 6.3b, it can be seen that no penalty is imposed to coefficients that are equal to zero, whereas the penalty value increases linearly as the absolute value of coefficient deviates from zero. In general, L^1 regularization results in a solution which is *sparse*, that is many of the parameters have an optimal value of zero. Although L^2 regularization drives the coefficients closer to zero, it can be shown that it does not cause the parameters to become sparse [18], whereas L^1 regularization may do so for large enough λ.

6.3.3 Custom Penalty Function

L^1 and L^2 regularization methods can indeed, for suitably selected λ, restrict the weights and biases to have absolute value less than one, satisfying the inequality constraints of (6.2). However, they do penalize admissible coefficient values that are different from zero, which in principle should

[8] Appropriate values for the hyperparameter λ are usually determined experimentally

not happen. The aim in this section, is to therefore devise a penalty function that can be used to solve (6.2) without penalizing admissible coefficients. Consider a function $\mathcal{P}(\boldsymbol{w}, \boldsymbol{b})$ such that

$$\mathcal{P}(\boldsymbol{w}, \boldsymbol{b}) = \begin{cases} 0, & \text{if } \boldsymbol{w} \text{ and } \boldsymbol{b} \text{ are admissible} \\ > 0, & \text{otherwise} \end{cases} \tag{6.9}$$

and the function

$$\mathcal{J}_\epsilon(\boldsymbol{w}, \boldsymbol{b}) = \mathcal{J}(\boldsymbol{w}, \boldsymbol{b}) + \frac{1}{\epsilon}\mathcal{P}(\boldsymbol{w}, \boldsymbol{b}) \tag{6.10}$$

with $\epsilon > 0$. $\mathcal{J}_\epsilon$ is such that[9]

$$\lim_{\epsilon \to 0} \mathcal{J}_\epsilon = \mathcal{J}_\alpha$$

The function $\mathcal{J}_\epsilon$ is known as *external* penalty function. The attribute external is due to the fact that, if $\bar{\boldsymbol{w}}$ and $\bar{\boldsymbol{b}}$ are points that minimize $\mathcal{J}_\epsilon$ then, in general, $\mathcal{P}(\bar{\boldsymbol{w}}, \bar{\boldsymbol{b}}) \neq 0$, i.e. $\bar{\boldsymbol{w}}$ and $\bar{\boldsymbol{b}}$ are not admissible [39]. Based on (6.9), a possible penalty function for solving (6.2) has the form

$$\mathcal{P}(\boldsymbol{w}, \boldsymbol{b}) = \sum_i \big(\max(0, w_i - 1) + \max(0, -w_i - 1)\big) + \sum_i \big(\max(0, b_i - 1) + \max(0, -b_i - 1)\big) \tag{6.11}$$

The penalty imposed by (6.11) for a scalar quantity is shown in Figure 6.4. Clearly no penalty is imposed for admissible coefficient values and the penalty increases linearly as the absolute value of the variable exceeds one.

Although external penalty functions are usually employed in sequential optimization methods, where a sequence of unconstrained optimization problems is solved with the property that ϵ is decreased in each iteration such that $\epsilon \to 0$, following such an approach for training neural network models would be extremely impractical and it may not even improve the performance of the model. As already discussed, in the context of a machine learning algorithm, the optimization objective is to sufficiently minimize the loss function so that adequate performance is achieved rather than seeking to find the global minimum of a non-convex function. In addition, the purpose of imposing coefficient constraints is to encourage the network's coefficients to have absolute value less than one, so that they can be represented without down-scaling in SC. Hence, for the purpose of training a stochastic computing compatible network, the modified objective function given by (6.10) with $\mathcal{P}(\boldsymbol{w}, \boldsymbol{b})$ as defined in (6.11) can be minimized once. In the event where some coefficient values are

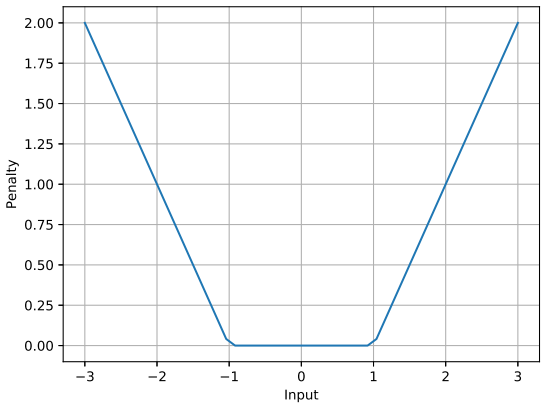


Figure 6.4: Custom penalty function for scalar quantities

[9]Because of the discontinuity of $\mathcal{J}_\alpha$, the limit has to be considered with proper *care*

not admissible, increasing the penalty on non-admissible values, by appropriately modifying ϵ, can ensure that all variables will indeed have absolute value less than one.

6.4 Training Scaling Scheme

For the purpose of neural network inference, the scaling coefficients of the SC network graph could be determined in advance and would remain fixed during run-time. On the other hand, during training time the weights and biases are continuously updated by the learning algorithm thus the scaling coefficients of the computational graph will, in general, very during run-time. The purpose of this section is to therefore present possible ways in which the scaling parameters of the SC network graph can be specified in the context of training a SC compatible neural network.

Since during training time the scaling parameters are not fixed, design choices regarding the scaling scheme must be made in advance. First, the designer should decide whether or not saturation arithmetic would be applied, i.e. whether the neuron architecture in Figure 6.1 or the one in Figure 6.2 will be used. In the event where no saturation arithmetic is used, the intrinsic scaling parameters, shown in Figure 6.1, can be used. Although viable, this choice may not in fact be optimal as these are worst-case scalings and could potentially lead to poor precision.

On the other hand, if saturation arithmetic is used, appropriate saturation levels need to be determined. In the context of inference this is relatively simple to do through simulation, where input data are applied to the network and the peak value reached by each internal signal is recorded. However, unlike inference, during the execution of the training algorithm the saturation levels may change as the network coefficients get updated. To determine appropriate saturation levels simulation based analysis could still be applied, by means of training the network in a conventional way and recording the peak value reached by each internal signal throughout the entire training process. Nevertheless, such an approach is laborious and requires to first conventionally train the network to determine saturation levels and then repeat the training process applying the regulations proposed by SC compatible training. For large scale neural networks, where training on high performance clusters may take several days to complete the aforementioned approach is clearly not viable. Furthermore, if inappropriate saturation levels are imposed, the performance of the network on the recognition task can severely degrade.

6.4.1 Optimization-Based Scaling Scheme

For the purpose of training SC compatible networks, this project proposes an optimization-based technique to determine optimal saturation levels during training time with minimal prior knowledge. Based on saturation arithmetic analysis in Section 4.4 and the proposed SC neuron architecture in Figure 6.2, it is clear that saturation levels are determined by the gain parameter of the linear gain block with saturation. In the proposed optimization-based scaling scheme, the gain coefficients associated with saturation arithmetic components are considered to be model parameters and thereby trainable parameters. Such a scheme, aims to *learn* optimal saturation levels during the training of a stochastic computing compatible network instead of manually devising their values.

Before delving into further details of the scaling scheme, a simplifying assumption could be made. Without loss of generality, only the gain coefficients at the output of a scaled inner product will be assumed to be trainable whereas the gain parameter associated with the addition of the bias term will be taken as a constant. The motivation for this simplification is rather empirical and it will become more apparent in Chapter 8 when experimental results are presented. It is also

expected that gain coefficients associated with up-scaling the inner product result will have greater impact on the overall network since the down-scale factor imposed by the scaled inner product is usually much larger than two, i.e. the down-scale factor of the scaled adder.

Let $\boldsymbol{g}$ denote collectively the gain coefficients of linear gain blocks (with saturation) associated with scaled inner products. The optimization-based scaling scheme can be formulated as follows,

$$\begin{aligned} \underset{\boldsymbol{w},\boldsymbol{b},\boldsymbol{g}}{\text{minimize}} \quad & \mathcal{J}(\boldsymbol{w},\boldsymbol{b},\boldsymbol{g}) \\ \text{subject to} \quad & -1 \leq \boldsymbol{w} \leq 1 \\ & -1 \leq \boldsymbol{b} \leq 1 \end{aligned} \tag{6.12}$$

Although in practice scalings that are integer powers of 2 are usually employed, the problem definition in (6.12) does not impose any constraints on the gain coefficients, i.e. $g_i \in \mathbb{R}$ for all i. The main reason is that optimization with respect to both discrete and continuous random variables is not straightforward. Nonetheless, allowing gain coefficients to take real values does not necessarily limit a SC-based hardware implementation.

Two distinct approaches are presented for constructing SC compatible network graphs:

- Case 1: A common gain coefficient is applied to all the SC-based neurons in a layer
- Case 2: A unique gain coefficient is applied to every SC-based neuron in a layer

While Case 2 may seem a more natural choice than Case 1 as it resembles the concept of bias addition, where a unique bias term is added to every neuron, there is no clear evidence, at least at this point, why it should always perform better than the option in Case 1. Both are possible design choices and is again the up to the designer's discretion to decide which one to use.

Another difference that should be noted, is that in Case 1 the number of additional decision variables that is introduced is significantly smaller compared to that in Case 2. Notwithstanding the fact that software based optimization tools are nowadays able to handle with relative success large scale optimization problems, increasing the number of decision variables always increases the complexity of the problem and of the model itself. Although it is not straightforward to perform generalization analysis of neural networks, in general as the model complexity increases a larger number of data points is needed to avoid overfitting of the training set. At the same time, a more complex model can achieve, under certain circumstances and for a sufficient number of data points, higher recognition accuracy, both in and out of sample.

Finally, while from a high level modelling perspective it may seem that additional decision variables or degrees of freedom are introduced to the network, these have a direct interpretation in terms of stochastic computing hardware. They dictate the saturation levels that could be used in a subsequent hardware implementation of neural network inference using SC hardware. The main advantage of using this approach is that, in contrast to the neural network inference approach presented in Chapter 5, training of the network is now aware of the subsequent SC-based inference implementation and optimally learns appropriate saturation levels based on the recognition task.

Chapter 7

Implementation

In this chapter, details regarding the environment used for development and simulation purposes are presented. In addition, implementation details for a certain number of SC processing elements analysed in Chapter 4 are given. As specified in the requirements capture section, in the context of this project a software implementation of SC is considered by simulating the functionality of SC processing elements. Development is undertaken in Python, a high level programming language which supports a large number of open-source libraries for scientific computing which are essential for the development and evaluation of deep learning models.

7.1 Implementation of SC Processing Elements

The SC processing elements presented and analysed in Chapter 4 are implemented and simulated in Python. A one-dimensional NumPy[1] array of integers is used to represent a stochastic bit-stream whereas multi-dimensional arrays are used to represent numerical vectors and matrices as stochastic bit-streams.

7.1.1 Combinational Logic-based Processing Elements

Combinational logic-based processing elements can be easily simulated in Python using NumPy's logical operations. What perhaps worth's mentioning is that neural network computational graphs are usually specified in terms of matrix computations, i.e. matrix multiplication and matrix addition. Specifically, the central computation in a feedforward network is given by the affine transformation $\boldsymbol{h}^{(i)} = \boldsymbol{W}^{(i)T}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}$. In software frameworks, it is common to use *broadcasting* to handle multiple data points in a single code statement[2]. Thus the previous statement is usually extended to $\boldsymbol{H}^{(i)} = \boldsymbol{H}^{(i-1)}\boldsymbol{W}^{(i)} + \boldsymbol{b}^{(i)}$, where each row in the matrices $\boldsymbol{H}^{(i)}$ and $\boldsymbol{H}^{(i-1)}$ corresponds to a distinct data point. To avoid misconceptions, it is important to note that the last statement is meaningless in mathematical terms as addition is only defined for objects of the same dimension. The statement implies the bias vector needs to be added to every row of the matrix product. This is exactly what is done by broadcasting where the smaller, in dimension, array is "broadcast" across the larger array so that they have compatible sizes.

[1] NumPy is a scientific computing library for the Python programming language.

[2] NumPy documentation defines broadcasting as "the process of making arrays with different shapes have compatible shapes for arithmetic operations".

To allow ease of use, these operations, namely matrix multiplication and matrix to vector "addition"[3], are implemented in SC in terms of the basic arithmetic operations introduced in Chapter 4. Although it may not be the most optimal approach, the aforementioned matrix operations are implemented by iteratively employing the basic SC arithmetic units of addition and inner product. Hence, *scaled* multiplication between two matrices of dimension $m \times n$ and $n \times p$ is computed by performing $m \times p$ scaled inner products, one for each entry of the resulting matrix. Similarly, the addition of the bias vector is handled by iteratively using the scaled adder in SC. Finally, Python's multiprocessing package is used to realise parallel computation of different rows in the matrix product. That is, every row of $\boldsymbol{HW}$ is computed in parallel. This is possible as there are no data dependencies between different inner product computations[4].

An additional implementation remark is that neural network coefficients, i.e. weights and biases, are stored in their floating-point representation. The motivation for this is twofold. Firstly, the SC inner product implementation given in Algorithm 2 does not require to convert floating-point weights into stochastic bit-streams. On the other hand, for the scaled adder used to realise bias addition this is not the case. That is, both inputs of the adder need to be stochastic bit-streams. Nevertheless, to maintain consistency with the weight coefficients, bias terms are also kept in their floating-point representation and the adder function used to carry out the addition of the bias vector converts "internally" the bias coefficients into stochastic bit-streams and thereafter performs MUX-based addition. Secondly, in a potential implementation of neural network training using SC hardware, the primary concern of the learning algorithm would be to learn the weight values and not their bit-stream representation. In fact, the bit-stream representation is not unique neither. As the implementation of SC arithmetic units was carried out while being aware of these facts, the aforementioned design choices regarding the representation of network parameters were made.

Furthermore, saturation arithmetic is incorporated within the functions implementing scaled addition and scaled inner product. Hence, additional input arguments are provided to the two functions to control whether or not saturation arithmetic will be used, that is whether the output of scaled addition or inner product will be further processed by the linear gain FSM. As briefly mentioned in the preceded chapters, for the scaled adder with saturation the linear gain parameter is fixed to two. On the other hand, for the scaled inner product with saturation the linear gain parameter can be controlled by the designer. Thus, an additional input argument is provided to the corresponding Python function that specifies the gain to be imposed by the FSM.

A final implementation remark is that the re-scaling procedure at the input and output of addition elements described in Chapter 5 is also incorporated within the functions implementing scaled addition and inner product. For the inner product, re-scaling XNOR multipliers are needed in both the input and the output of the MUX unit[5]. On the other hand, for the stochastic adder re-scaling is needed only at the input of the MUX unit. However, following the modification specified in Chapter 5, all the activations in a neural network layer are enforced (if needed) to have a common scaling factor. Thus, there is no need to re-scale the inputs of the MUX-based inner product. Recall that this amendment does not eliminate the need to re-scale the inputs of the stochastic adder that performs the addition of the bias term. Therefore, a re-scaling unit is needed at the input of the scaled adder and at the output of the scaled inner product. Therefrom, the two functions are extended as follows:

[3]For the remaining of this chapter, the phrase matrix to vector addition will be used to denote the operation employing broadcasting

[4]In fact, every inner product could be computed in parallel

[5]The multiplier at the output is needed to increase the scaling of the signal to the next integer power of two

- Scaled Adder: The scalings of the two operands are provided as input arguments to the function which re-scales the input signals as necessary.
- Scaled Inner Product: The (common) input scaling is provided as well as the desired output scaling[6].

The interface of the two functions is subsequently given,

```
def add(x,y,s_in,upscale=False):
    """
    Scaled addition of two stochastic bit-strems

    Parameters
    ----------
    x,y: 1D NumPy arrays holding the stochastic bit-streams to be added
    s_in: 1D NumPy array holding the scalings of x and y respectively
    upscale: Boolean indicating whether a linear gain of two with saturation
    should be applied or not

    Returns
    -------
    1D NumPy array holding the output bit-stream
    """

def dot(x,w,s_in,s_out,upscale=False,gain=None):
    """
    Scaled inner product in stochastic computing

    Parameters
    ----------
    x: 2D NumPy array holding input bit-streams
    w: 1D NumPy array holding floating-point weights
    s_in: Scalar specifying the input scaling parameter
    s_out: Scalar specifying the output scaling parameter
    upscale: Boolean indicating whether linear gain with saturation
    should be applied or not
    gain: Scalar specifying the linear gain to be applied if upscale is True

    Returns
    -------
    1D NumPy array holding the output bit-stream
    """
```

7.1.2 FSM-based Processing Elements

The implementation of FSM-based computational elements in Python is also relatively straightforward. Algorithm 5 illustrates a possible implementation of the *Stanh* FSM architecture in Figure

[6]Although the output scaling can be computed during run-time (using equation (5.4)), since the SC units are used only for inference purposes where all the scaling parameters can be computed in advance, it is more convenient to provide the desired output scaling as input argument to the function

4.6. A similar approach is used to implement the remaining FSMs. As a side remark, all the FSMs are initialised at the centre of the state sequence. For this type of FSM, initialisation should not be a major issue as long as the bit-stream length L is sufficiently large, i.e. $L \gg N$. Then, the probability encoded by the bit-stream will be inferred over time and the FSM will converge to an appropriate state value.

7.2 Neural Network Inference

Implementation of neural network inference in stochastic computing requires to construct the equivalent SC network graph and then execute that graph for a number of data points. To do so, the scaling scheme proposed in Chapter 5 needs to be applied. That is, given the trained weights and biases, the scaling of every signal in the network needs to be computed as analysed in the aforementioned chapter. To aid this process, *scaling operations* are developed. Essentially, these operations use the worst-case scaling of each individual SC arithmetic operation (i.e. scaled addition and scaled inner product) to automate the process of finding the scaling of every signal in the network graph.

As mentioned above, in software frameworks, neural network graphs are usually expressed in terms of matrix operations, i.e. matrix multiplication and matrix to vector addition. To allow ease of use, the scaling operations are vectorised to eventually develop a scaling operation for matrix multiplication and matrix to vector addition. So given the trained weights and biases, the user just needs to specify a two-dimensional array in which every entry is equal to the scaling of

Algorithm 5: Stochastic approximation of the hyperbolic tangent function

Input : Bit-stream X
Number of states N
Number of stochastic samples L
Output: Bit-stream Y

Initialise FSM parameters
$S_{min} = 0$
$S_{max} = N - 1$
$S_{bound} = \frac{N}{2}$
Initialise state sequence
$S = S_{bound}$
for $i = 0$ *to* $L - 1$ **do**
  State Transition
  if $X[i] == 0$ **then**
    $S = S - 1$
  else
    $S = S + 1$
  Saturate the counter
  if $S < S_{min}$ **then**
    $S = S_{min}$
  else if $S > S_{max}$ **then**
    $S = S_{max}$
  Output Logic
  if $S \geq S_{bound}$ **then**
    $Y[i] = 1$
  else
    $Y[i] = 0$

the corresponding input feature and thereafter construct a dataflow, using the provided scaling operations, that resembles the neural network graph.

The interface of the matrix multiplication scaling operation is given by

```
def sc_matmul_scaling(S_in,W):
```

The function computes the scaling coefficients of a SC-based matrix multiplication of the form $\boldsymbol{XW}$. The argument S_in is a two-dimensional NumPy array holding the scaling of each input signal in $\boldsymbol{X}$ and the argument W is a two-dimensional array holding the trained weights $\boldsymbol{W}$.

Similarly, for the case of matrix to vector addition the function

```
def sc_matvec_add_scaling(S_in,b)
```

computes the scaling coefficients of a SC-based matrix to vector addition of the form $\boldsymbol{X} + \boldsymbol{b}$. The argument S_in is a two-dimensional NumPy array holding the scaling of each input signal in $\boldsymbol{X}$ and the argument b is a one-dimensional array holding the bias coefficients $\boldsymbol{b}$.

As a side remark, based on the scaling scheme presented in Chapter 5, it is not in fact necessary to compute the scaling factors for every test data point, as they will actually be the same[7]. Nonetheless, the main reason for choosing to develop the scaling operations in terms of matrix multiplication and matrix to vector addition is to resemble the way in which neural network graphs are expressed in programming terms.

7.3 The TensorFlow Environment

Following the discussion in Chapter 3 of the Interim Report, TensorFlow is selected as the software framework for the development and evaluation of neural network models. TensorFlow is an open-source software library for dataflow programming across a range of tasks. The central unit of data in TensorFlow is the tensor, which consists of a set of primitive values shaped into a multi-dimensional array [1]. A dataflow computation in TensorFlow is expressed in terms of a directed graph, composed by a set of nodes and the graph can be constructed using Python. Note that in contrast to conventional sequential programs, a TensorFlow description consists of two distinct stages: 1) Part of the description specifies the computational graph 2) Part of the description executes the computational graph.

In a TensorFlow graph, each node has zero or more inputs, zero or more outputs and represents the instantiation of an operation [1]. In general, an operation represents an abstract computation like matrix multiplication or addition. In many iterative processes, like neural network training, the computation graph is executed multiple times and most tensors do not survive past a single execution of the graph [1]. However, a *variable* is a special kind of operation that is appropriately handled to survive across multiple executions of a graph. In contrast to tensors that are defined as *constants*, variables can be modified along the execution of the computational graph. Thus, in the context of neural network training, the trainable parameters of the model are typically stored in tensors held in variables. A graph can also be parameterized to accept external inputs such as input data to a neural network. In the TensorFlow framework such external inputs are known as *placeholders*[8].

TensorFlow also provides implementations of several optimization algorithms that can be used to train a network by means of minimizing the loss function. As many optimization algorithms

[7] Rows in S_in will be the same

[8] For neural network inference it is not necessarily important to distinguish placeholders from variables

use the information on the gradient of the objective function with respect to a set of inputs in order to update model parameters, TensorFlow provides built-in support for automatic gradient computation [1]. If for example a tensor C in a TensorFlow graph depends, through a subgraph of operations, on a set of tensors $\{X_k\}$, then there exists a built-in function that will return the gradient tensors of C with respect to each of X_k, i.e. the tensors $\{\partial C/\partial X_k\}$.

As described in [1], TensorFlow computes gradient tensors by extending the computational graph in the following way. When the gradient of a tensor C with respect to some tensor X_k needs to be computed, TensorFlow first finds the path in the graph from X_k to C. It then backtracks from C to X_k and for each operation in the backward path it adds a node to the TensorFlow graph which computes the partial derivatives along the backward path using the chain rule.

7.4 Stochastic Computing Training Process

As analysed in Chapter 6, neural networks that are "aware" of a subsequent inference implementation using SC hardware can be trained by following the so called stochastic computing compatible training procedure. To realise such a procedure it is essential to efficiently implement the modified neuron architectures, shown in Figures 6.1 and 6.2, in the TensorFlow environment.

As described in the preceded chapter, scaled operations in SC could be modelled, for the purpose of SC compatible training, by simply post processing (i.e. down-scaling) the result calculated using floating-point computations. However, scaled computations also have an impact on the gradient of each operation. In Chapter 6, it was argued that depending on the software framework used to deploy and train the neural network, it may indeed be possible to model the effect of gradient-discrepancy due to SC by simply post processing the result computed using floating-point computations. Following the preceded description on how gradients are computed in TensorFlow, it is now clearer why post processing the floating-point result will indeed have an impact on the gradient of the overall operation. The additional operations introduced to the proposed neuron model in Figure 6.2, when expressed as a composition of existing TensorFlow operations, will introduce additional nodes in the computational graph that will obviously affect the forward computation of graph but will also have the appropriate impact on the backward propagation of the graph, i.e. on the gradient computation.

For implementation purposes, the neuron architecture in Figure 6.2 is split into two parts: 1) A part implementing scaled inner product with saturation and 2) A part implementing bias addition with saturation. Following similar considerations as in Section 7.1, the inner product operation is extended to a matrix multiplication of the form $\boldsymbol{XW}$ in order to handle multiple data points on a single code statement. Similarly the addition of the bias term is extended to matrix addition. Thereby, functions that implement modified matrix multiplication and addition, according to the neuron model in Figure 6.2, are implemented as a composition of existing TensorFlow operations to allow propagation of gradients during the execution of the training algorithm. Code listings for the implementation of the modified operations in TensorFlow are given in Appendix A.2.

The final topic to comment on would be the implementation of the penalty functions proposed in Chapter 6 for constraining trainable parameters during SC compatible training. TensorFlow provides built-in regularizer operations that can be used to apply L^1 and L^2 regularization to the trainable parameters, i.e. TensorFlow variables. On the other hand, the custom penalty function presented in Section 6.3.3 is a non-standard regularizer and needs to be implemented. Once again, the penalty function is implemented as a composition of existing TensorFlow operations in order to leverage the auto differentiation capabilities of TensorFlow, so that training using the particular penalty function can be performed without any issues regarding gradient computation.

Chapter 8

Experiments and Results

This chapter presents experimental results from the implementation of both neural network inference in SC as well as results obtained by following the proposed stochastic computing compatible training procedure. The experiments are based on the MNIST handwritten digit image database [28], a dataset describing the set of 10 classes, i.e. digits 0 to 9. The dataset consists of $60,000$ training samples and $10,000$ testing samples with 28×28 grayscale images. Emphasis is given on feedforward networks without necessarily following any particular network architecture. Neural networks are deployed and trained in TensorFlow and *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions [25], is, used for minimizing the loss function of a network in the context of training.

8.1 Neural Network Inference in Stochastic Computing

As inference cannot occur prior training, a network is devised and trained to be further used for the purpose of SC-based inference. A network based on the configuration listed in Table 8.1 is trained on the MNIST dataset, following a conventional training process. Hence, the trained network is unaware of the subsequent implementation of the inference algorithm in stochastic computing. The trained model, namely Network I, achieves classification accuracy[1] of 87.05% on the training set and 87.03% on the testing set.

As a side remark, the objective of this chapter, and project in general, is not to develop models that will achieve state of the art recognition accuracy on a certain task, like the recognition of handwritten digits. Instead, the objective of the majority of the models developed in this section

Table 8.1: Network I Characteristics

Network Architecture		
Input Layer	Number of Units	784
Hidden Layer	Number of Units Hidden Unit	128 Linear
Output Layer	Number of Units Output Unit	10 Softmax

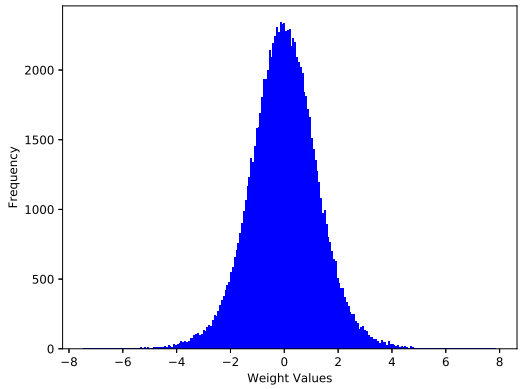
Training Parameters	
Optimizer	Adam
Loss Function	Cross Entropy Error
Initial Learning Rate	0.1
Epochs	500
Batch Size	128
SC Compatible Training	No
Penalty Function	N/A

[1] Classification accuracy is defined as: $\text{Accuracy}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}[y_i = \hat{y}_i]$, where y_i is the true label and $\hat{y}_i$ the predicted label corresponding to the data sample $\boldsymbol{x}_i$, for all $i = 1, ..., N$.

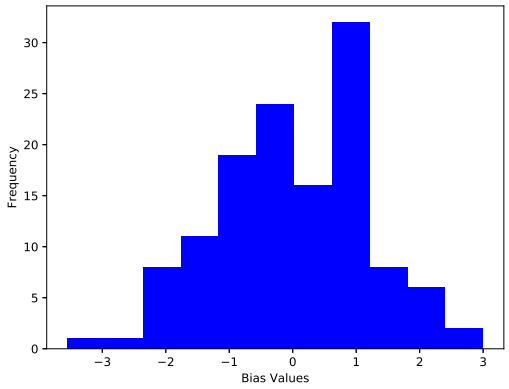
is to provide test case networks for which the techniques presented in the preceded chapters can be applied and tested. Yet, in every situation the "best" recognition accuracy is tried to be achieved[2].

Once trained coefficients for Network I are obtained, histogram plots are computed for the weights and biases in each layer. The objective of these plots is to obtain an empirical estimate for the distribution of the network's parameters in every layer but also to observe the range of values that these coefficients span. For network I, these are shown in Figure 8.1. Interestingly, the inherent empirical distribution of weights in both the hidden and output layer is given by a bell curve centered around zero but with different standard deviation for each layer. In particular, the vast majority of the weights in the first layer spans the range $[-4, 4]$, whereas the majority of the weights in the output layer spans the range $[-1, 1]$. On the other hand, the bias coefficients do not seem follow a similar distribution, or at least is not obvious from the results in Figure 8.1, mainly because the number of bias parameters is small thus the empirical estimate obtained by means of the histogram is not representative. Clearly, the above conclusion is not general and the distribution of the trainable parameters depends on many factors, including the specific dataset, the network architecture as well as the type of regularizer used.

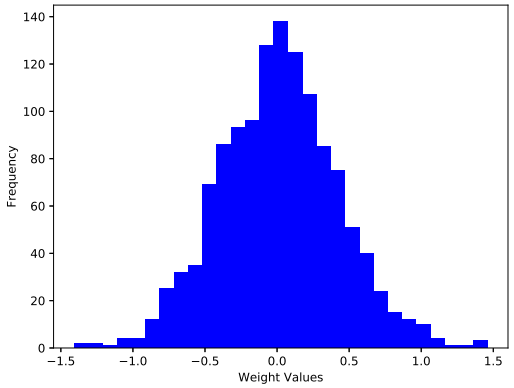
Given the trained coefficients, implementation of neural network inference in stochastic computing requires to apply the scaling scheme presented in Chapter 5. For the particular network, worst-case scalings are computed as follows. First of all, a normalized version of the MNIST



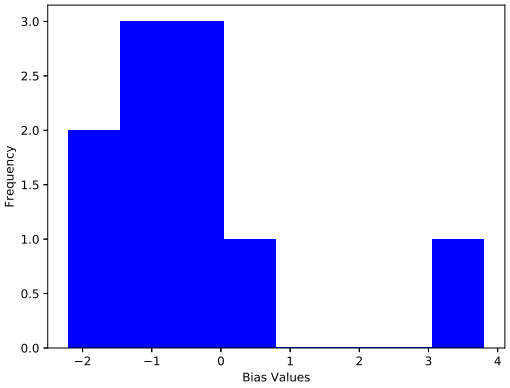
(a) Weights in the hidden layer



(b) Biases in the hidden layer



(c) Weights in the output layer



(d) Biases in the output layer

Figure 8.1: Histogram plots of the trainable parameters of Network I

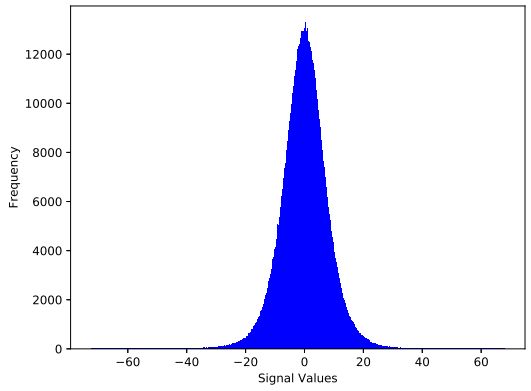
[2]Indeed, several machine learning methods, including neural network classifiers, have already been tested on the MNIST dataset and some of them have achieved close to perfect classification accuracy (99.77 %) [28]

Table 8.2: Worst-case scaling parameters for Network I

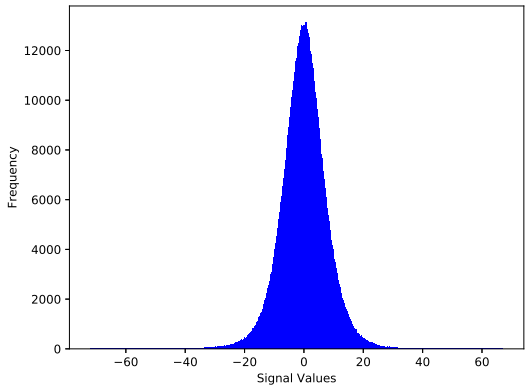
Node	Input	Weight Product 1^{st} Layer	Bias Addition 1^{st} Layer	Weight Product Output Layer	Output
Scaling	1	2^{10}	2^{11}	2^{17}	2^{18}

dataset, provided by the TensorFlow framework, is used. Hence, input values lie within $[0, 1]$ and the input scaling computed by (5.1) is equal to one[3]. Thereby, input data need not be down-scaled. Afterwards, using the scaling operations developed in Chapter 7, the scaling of every activation in the SC network graph is automatically computed. In fact, for this network, it happens that all signals in each layer have a common scaling parameter. These are listed in Table 8.2.

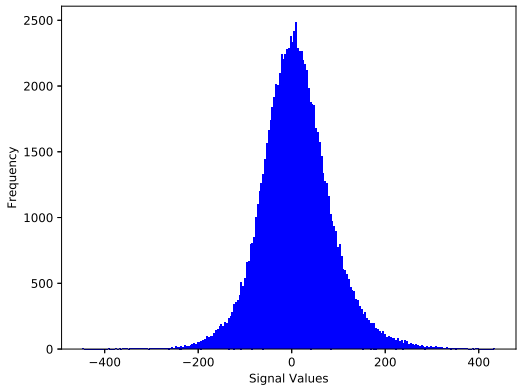
An output scaling of 2^{18} implies that the network's outputs lie within $[-2^{18}, 2^{18}]$. This is clearly a huge scaling parameter and is expected to significantly degrade the precision of the overall system. The main reason to argue for this is the following. SC computations are always performed within the interval $[-1, 1]$. Thereafter, the output of the entire SC system is converted back to the corresponding floating-point representation and finally up-scaled by the output scaling parameter. Since computations in stochastic computing are associated with a certain degree of variance, the output of the SC system is essentially a random variable. Clearly, scaling a random variable by 2^{18} significantly increases its variance, and thereby the uncertainty in the result.



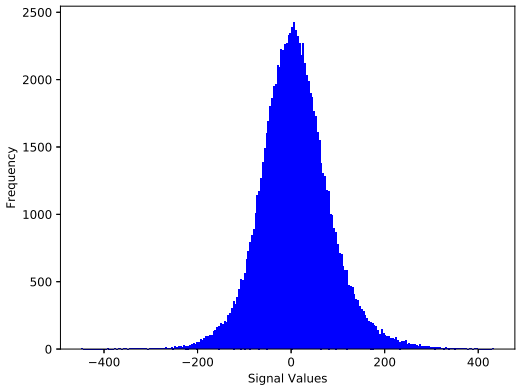
(a) Weight product in the hidden layer



(b) Bias addition in the hidden layer



(c) Weight product in the output layer



(d) Bias addition in the output layer

Figure 8.2: Signal values, due to test data, at the output of every operation in network I

[3]Pixel values in the original MNIST dataset lie within $[0, 255]$

To mitigate this issue, saturation arithmetic can be applied throughout the SC network graph. As discussed in Chapter 5, a possible approach is to determine the saturation levels through simulation. That is, given the trained weights and biases, test data are provided to the network graph and the value reached by each internal signal is recorded. For this network, signal values obtained by applying test data to the model are shown in Figure 8.2. Clearly, the signal values are significantly smaller than the worst-case scaling coefficients listed in Table 8.2. This is a great illustration of how worst-case scalings can be overly pessimistic, catering for situations that are extremely rarely encountered with practical input data. For instance, at the output of the weight product operation in the hidden layer, 99.92% of the signals have absolute value less than 32. Thus, a scaling parameter equal to 2^5 would be sufficient to represent the vast majority of the signals in stochastic computing without any saturation error. Compared to the worst-case scaling computed above, there is indeed a dramatic reduction. Finally, another interesting observation is that after the addition of the bias term, the range of values spanned by the signals remains almost intact. Hence, the down-scaling effect of the stochastic adder can be removed by applying a linear gain of 2 without introducing any saturation errors.

The objective is to then implement the inference algorithm of the trained network in stochastic computing. Three possible scenarios, as listed below, are considered.

1. No saturation arithmetic is used. That is, worst-case scalings are applied.
2. Saturation arithmetic is used and saturation levels are determined through simulation.
3. Saturation arithmetic is used along with the decomposed version of the MUX-based inner product, introduced in Section 4.4.3.

Effectively, each scenario results in a different SC network graph. The main reason to consider such scenarios is to investigate the effect of saturation arithmetic to the overall network graph, but also to explore if there are any benefits, at least in this example, from using the decomposed version of the MUX-based inner product. For each scenario, the parameters of the constructed SC network graph, including saturation levels, are listed in Tables 8.3 - 8.5. Saturation levels in both scenario 2 and 3 are selected so that the vast majority of the signal values in each layer, shown in Figure 8.2, can be represented in SC without compression error.

Table 8.3: Network I SC Inference Parameters - Scenario 1

Hidden Layer Parameters	
Input scaling	1
Inner product output scaling	1024
Bias addition input scaling	1024
Bias addition output scaling	2048

Output Layer Parameters	
Input scaling	2048
Inner product output scaling	131072
Bias addition input scaling	131072
Bias addition output scaling	262144

Table 8.4: Network I SC Inference Parameters - Scenario 2

Hidden Layer Parameters	
Input scaling	1
Inner product output scaling	1024
Inner product gain factor	32
Inner product saturation level	**32**
Bias addition input scaling	32
Bias addition output scaling	64
Linear gain factor	2
Saturation level	**32**

Output Layer Parameters	
Input scaling	32
Inner product output scaling	2048
Inner product gain factor	8
Inner product saturation level	**256**
Bias addition input scaling	256
Bias addition output scaling	512
Linear gain factor	2
Saturation level	**256**

Table 8.5: Network I SC Inference Parameters - Scenario 3

Hidden Layer Parameters	
Input scaling	1
Number of nodes in the decomposed inner product	8
Inner product intermediate scaling	128
Inner product intermediate gain factor	4
Inner product intermediate saturation level	32
Inner product output scaling	256
Inner product output gain factor	8
Inner product output saturation level	**32**
Bias addition input scaling	32
Bias addition output scaling	64
Linear gain factor	2
Saturation level	**32**

Output Layer Parameters	
Input scaling	32
Number of nodes in the decomposed inner product	4
Inner product intermediate scaling	512
Inner product intermediate gain factor	4
Inner product intermediate saturation level	128
Inner product output scaling	512
Inner product output gain factor	2
Inner product output saturation level	**256**
Bias addition input scaling	256
Bias addition output scaling	512
Linear gain factor	2
Saturation level	**256**

The SC network graph for each scenario is constructed and neural network inference in stochastic computing is performed. As a side comment, inference is performed only for a small subset of data points since the simulation of SC inference in software takes a significant amount of run-time especially long stochastic bit-streams [4]. In each scenario, SC inference is executed for an increasing bit-stream length and the SC network accuracy is recorded. It is, in general, expected that for a sufficiently long bit-stream, the accuracy of the SC network will converge to the accuracy of the network implemented in floating-point arithmetic. Figure 8.3 shows the obtained results. A logarithmic scale (base 2) is used for the horizontal axis.

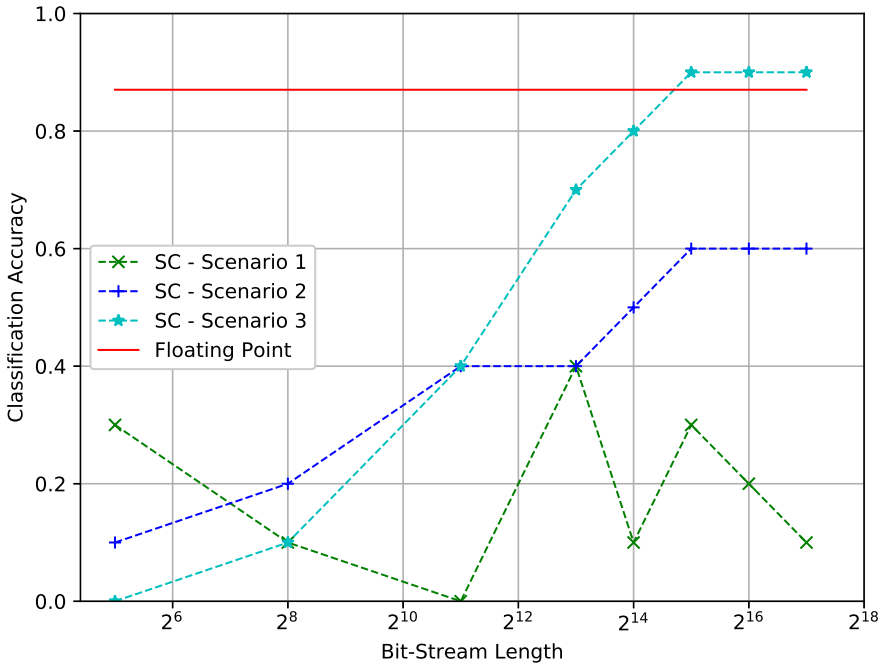


Figure 8.3: Test accuracy of Network I implemented in SC versus the bit-stream length

[4]In fact this is not a major issue as potentially such a neural network could be implemented on an embedded system where testing (i.e. inference) would be performed in real-time, thus on a very small number of incoming test data. Hence, testing the SC implementation on a small number of data points also provides an indication how the SC network deals with those scenarios.

Perhaps unsurprisingly, when worst-case scalings are employed, i.e. green trace, the SC network accuracy does not converge to the floating-point network accuracy, at least for a bit-stream of length 2^{17}. In fact, for the first scenario, there doesn't seem to be a clear trend in the accuracy trace of SC network, which seems to rather fluctuate randomly around low accuracy points. Hence, if worst-case scalings are applied, a significantly longer bit-stream would be needed for the SC network to reliably emulate the functionality of the conventional network.

On the other hand, accuracy traces for scenarios 2 and 3 seem to follow an increasing trend. That is, as the bit-stream length increases, the accuracy of the overall SC network increases. Specifically, for a bit-stream length equal to 2^{15}, the SC network accuracy for scenario 3 reaches, and in fact it exceeds, the floating-point network accuracy. The recognition accuracy of the SC network has effectively converged at the horizontal level of 0.9, as further increasing the bit-stream length does not alter the predictions of the SC network. Although the accuracy trace for scenario 2 is increasing, it does not reach that of the floating-point implementation, at least for a bit-stream length of 2^{17}. It was found that a wordlength of 2^{19} is needed for the SC network accuracy in the second scenario to reach the level of accuracy of the floating-point implementation.

From background analysis, it is known that an increase in the precision of a stochastic computation requires an exponential increase in the bit-stream length. Hence, the expectation is that the convergence of the SC network accuracy would be of the form $\sqrt{L}$, where L is the number of bits in the stochastic sequence. Recall that the horizontal axis of the plot in Figure 8.3 is based on a base-2 logarithmic scale. Thus, the linear growth followed by convergence at the horizontal level of 0.9 that is observed in Figure 8.3, is in fact a convergence of the form $\sqrt{L}$ given the logarithmic scale of the plot.

At least for this network, an interesting design trade-off arises out of the results in Figure 8.3. As qualitatively discussed in Section 4.4.4, saturation arithmetic is not a cost-free design methodology, both in stochastic as well as in conventional binary computing. In stochastic computing, every saturation arithmetic element is realized by means of a linear FSM, i.e. a saturating counter, which obviously incurs area overheads. Furthermore, as argued in the aforementioned section of the report, the saturated inner product with decomposition, utilised in scenario 3, incurs additional area overheads if a single linear gain FSM is used in every intermediate node, i.e. if no resources are shared. However, based on the results in Figure 8.3, the SC network utilizing the inner product with decomposition achieves the floating-point network accuracy with a bit-stream length 16 times smaller than that needed for the SC network in scenario 2. Recall that in stochastic computing, for the same hardware, longer bit-streams incur longer execution times. Thus, there seems to be, at least qualitatively, an area overhead versus delay trade-off between scenarios 2 and 3. Scenario 2 requires a bit-stream 16 times longer than that of scenario 3 to emulate the functionality of the floating-point network implementation. On the other hand, scenario 3 incurs larger area overheads due to the saturated inner product with decomposition.

Obviously, the preceded analysis was purely qualitative and in fact additional considerations need to be taken into account. For example, the inner product with decomposition will approximately introduce a two clock cycle delay in the initialization of the SC hardware architecture[5]. In general, it is not straightforward to qualitatively claim how large will be the impact of each of the aforementioned overheads. Such a conclusion can only be done by means of a hardware implementation where saturation area overheads as well as the delay impact due to the additional units can be quantitatively assessed.

[5]One clock cycle due to the additional linear gain FSM at each of the intermediate nodes and one clock cycle due to the additional MUX-based accumulator at the output that is used to aggregate the intermediate results

8.2 Regularized Network Inference in Stochastic Computing

In the last experiment, the network was trained in ignorance of the subsequent implementation of the inference algorithm in stochastic computing. In that case, weights and biases will not necessarily lie within $[-1, 1]$. Based on the implementation of the inner product computation in stochastic computing, large in absolute value weights will give rise to significantly large scaling parameters throughout the SC network graph, which in turn has several consequences on the implementation of inference in stochastic computing.

The aim in this section is to therefore modify the training process so that the trained network is aware of the subsequent inference implementation in stochastic computing. In Chapter 6, penalty functions were proposed for constraining the weights inside the interval $[-1, 1]$. Furthermore, an optimization-based scaling scheme was presented for learning optimal saturation levels, i.e. optimal gain parameters. In this experiment, only regularization of weights and biases is considered and saturation levels, if needed, are determined through simulation. The main reason is that the optimal gain parameters obtained from the optimization-based scaling scheme will not, in general, be integer powers of 2. In principle, the linear gain block with saturation, presented in Section 4.3.4, can be used to realise any gain parameter. However, this requires to determine appropriate values of the control parameter K, a task which is currently solved manually through trial and error. Thereby, for the purpose of this experiment, the neuron architecture in Figure 6.1 is employed (i.e. no saturation is applied during training) and if necessary, saturation levels for SC inference are determined through simulation.

Three penalty functions were proposed for regularizing the weights and biases during SC compatible training, namely: 1) L^1 Regularization 2) L^2 Regularization and 3) The custom penalty function. For each penalty function, a network with the same characteristics as those listed in Table 8.1 is trained[6] and recognition accuracy on both the training and testing sets, is reported in Table 8.6. Histogram plots for the parameters of each network are shown in Figures 8.4-8.6.

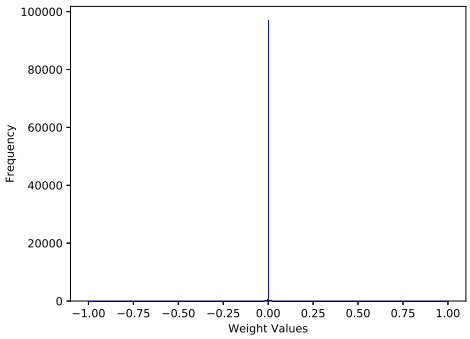
The effect of L^1 and L^2 regularization on the trainable parameters is rather expected. L^1 regularization promotes sparsity in the coefficients, whereas L^2 regularization preserves the bell-shaped distribution of weights that was obtained prior applying regularization. At least for this task, L^1 regularization does not seem optimal as it disrupts the inherent distribution of weights as well as degrading the recognition accuracy of the model. On the other hand, L^2 regularization does indeed restrict the weights and biases to have absolute value less than one and at the same time it improves the classification accuracy of the model, a consequence which is obviously desired. Finally, the custom penalty function restricts the weights and biases to lie within $[-1, 1]$ without preserving the inherent bell-shaped distribution of the weights.

As a side remark, for the purpose of constraining the weights and biases to have absolute value less than one, the parameters obtained through L^2 regularization may seem to be over-regularized, i.e. all of the weights and biases have absolute value way smaller than one. However, reducing the L^2 regularization scale λ, to mitigate the importance of the regularizer during training, was found to degrade the performance of the model on both the training and testing datasets.

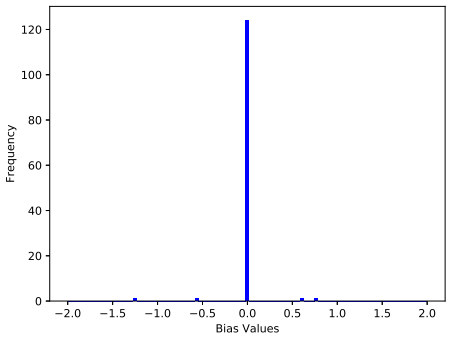
Table 8.6: Classification accuracy on the MNIST dataset with different regularization functions

	L^1 Regularization	**L^2 Regularization**	**Custom Penalty Function**
Training Accuracy	89.74 %	92.6 %	92.73 %
Testing Accuracy	89.88 %	92.34 %	92.1 %

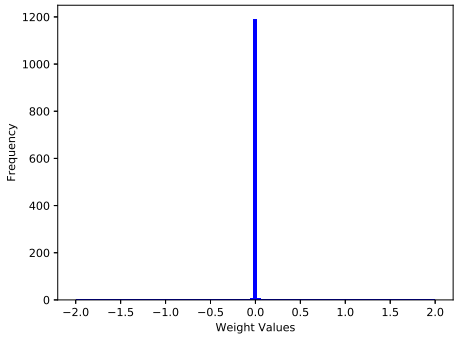
[6]Appropriate hyperparameter values, i.e. regularization scale, for each network are determined experimentally



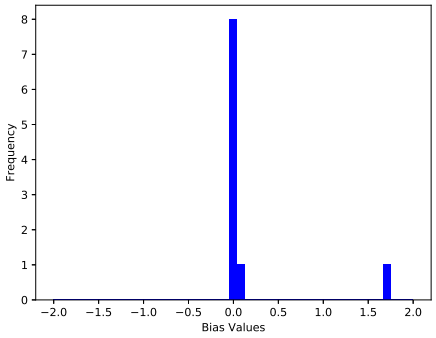
(a) Weights in the hidden layer



(b) Biases in the hidden layer

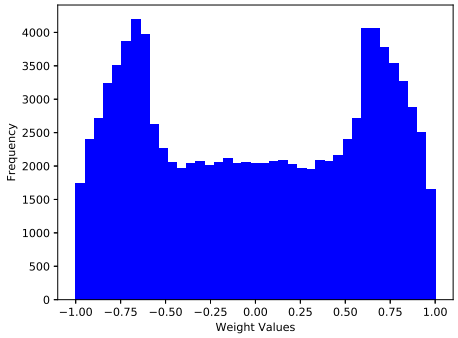


(c) Weights in the output layer

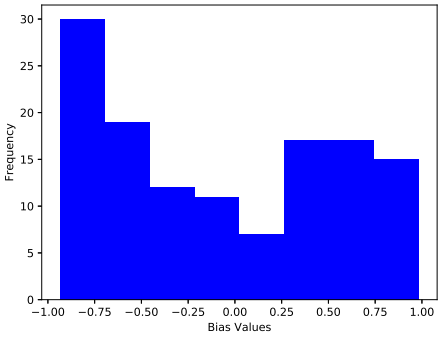


(d) Biases in the hidden layer

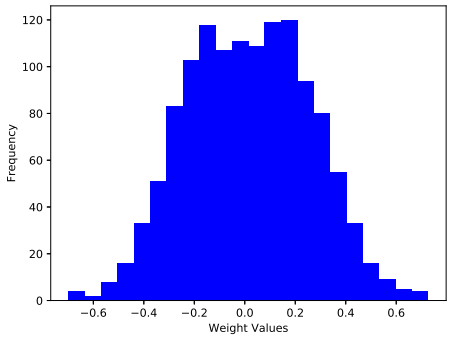
Figure 8.4: Histogram plots of the trainable parameters of Network II with L^1 regularization



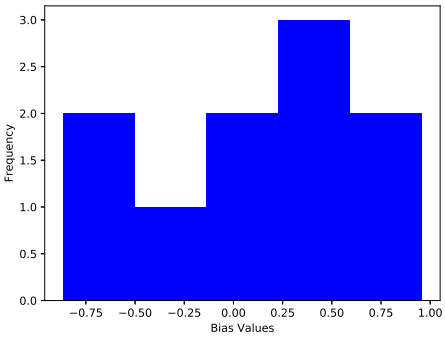
(a) Weights in hidden layer



(b) Biases in the hidden layer

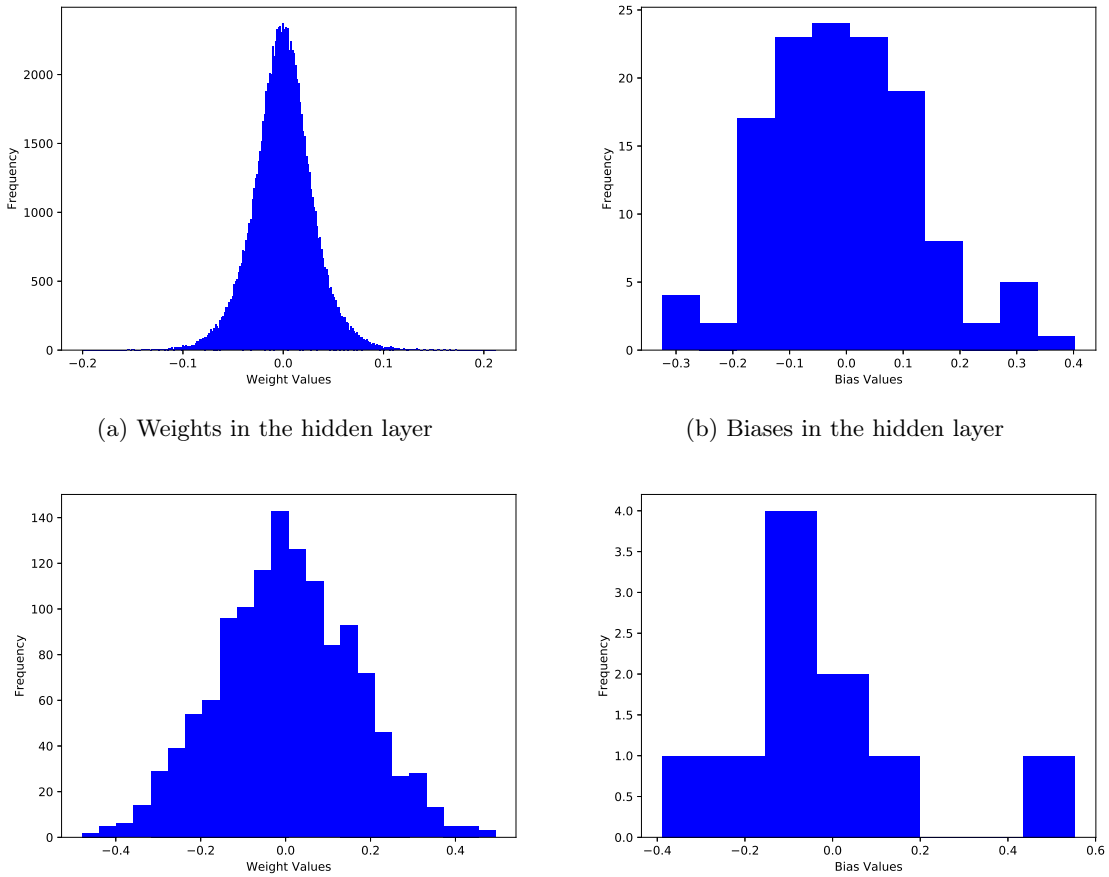


(c) Weights in the output layer



(d) Biases in the output layer

Figure 8.5: Histogram plots of the trainable parameters of Network II with the custom penalty function



(a) Weights in the hidden layer

(b) Biases in the hidden layer

(c) Weights in the output layer

(d) Biases in the output layer

Figure 8.6: Histogram plots of the trainable parameters of Network II with L^2 Regularization

Without loss of generality, SC inference is carried out only for the network trained with L^2 regularization. The main reason is that L^2 regularization gave rise to the model with the highest recognition accuracy while preserving the inherent distribution of parameters. Thus, it seems natural to select as the "best" model the one trained with L^2 regularization. We denote this network as Network II.

For Network II, worst-case scalings throughout the SC network graph are computed using the developed scaling operations and listed in Table 8.7. As expected, smaller, in absolute value, weights give rise to significantly smaller worst-case scaling parameters. In particular, the output scaling of Network II is 64 times smaller than that of Network I. This is a desired consequence, as smaller scaling parameters are, in general, easier to deal with.

As a side comment, it was found that in every layer, some signals had a scaling parameter half of that listed in Table 8.7, i.e. some signals were scaled by $\{1, 2^4, 2^5, 2^{10}, 2^{11}\}$ throughout the structure. Nonetheless, it is possible to appropriately re-scale signals at the output of every operation so that all of them have a common scaling parameter (the largest one). As discussed in Section 5.2.3, having a common scaling amongst signals in the same layer does not require to re-scale

Table 8.7: Worst-case scaling parameters for Network II

Node	Input	Weight Product 1^{st} Layer	Bias Addition 1^{st} Layer	Weight Product Output Layer	Output
Scaling	1	2^5	2^6	2^{11}	2^{12}

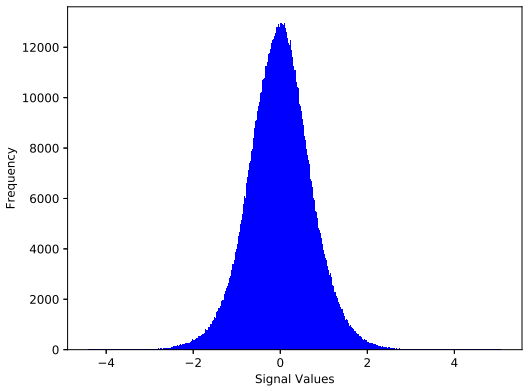
signals at the input of every inner product which, to some extend, simplifies the implementation of SC inference. Thus, for the remaining of this section, it is assumed that all signals in the network graph are scaled by the worst-case scaling parameters listed in Table 8.7.

The scalings listed in Table 8.7, although smaller than those of Network I, are still catering for all possible computational outcomes, some of which may never actually occur during inference. As empirically justified in the preceded section, worst-case scalings are indeed overly pessimistic. Thus, saturation arithmetic can be applied throughout the network graph to reduce the scaling parameters and increase precision in the computations, without introducing significant compression error to the results. Signal values obtained by applying test data to Network II are shown in Figure 8.7, and unsurprisingly peak signal values are way smaller than the worst-case scaling parameters. Similar to the previous experiment, the signal values follow a zero-mean normal distribution and the addition of the bias term leaves the range of values spanned by the signals intact.

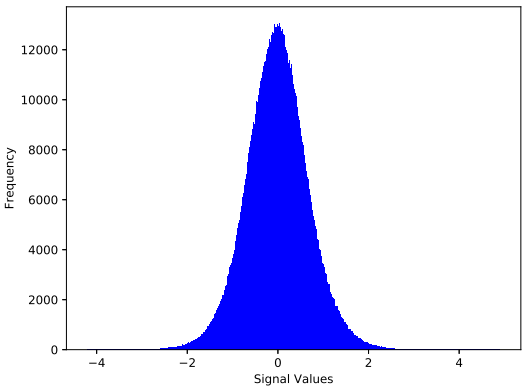
As in the previous section, three possible scenarios are considered while constructing the network graph. These are:

1. Saturation arithmetic with a gain of 2 is applied after bias addition only.
2. Same as in Section 8.1.
3. Same as in Section 8.1.

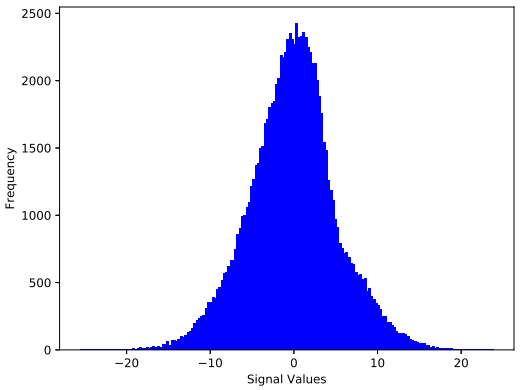
Three SC network graphs, one for each scenario, are constructed according to the parameters listed in Tables 8.8 - 8.10.



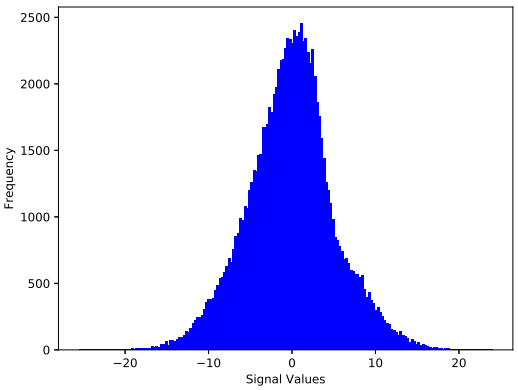
(a) Weight product in the hidden layer



(b) Bias addition in the hidden layer



(c) Weight product in the output layer



(d) Bias addition in the output layer

Figure 8.7: Signal values, due to test data, at the output of every operation in network II

Table 8.8: Network I SC Inference Parameters - Scenario 1

Hidden Layer Parameters	
Input scaling	1
Inner product output scaling	32
Bias addition input scaling	32
Bias addition output scaling	32

Output Layer Parameters	
Input scaling	32
Inner product output scaling	1024
Bias addition input scaling	1024
Bias addition output scaling	1024

Table 8.9: Network I SC Inference Parameters - Scenario 2

Hidden Layer Parameters	
Input scaling	1
Inner product output scaling	32
Inner product gain factor	8
Inner product saturation level	**4**
Bias addition input scaling	4
Bias addition output scaling	8
Linear gain factor	2
Saturation level	**4**

Output Layer Parameters	
Input scaling	4
Inner product output scaling	128
Inner product gain factor	8
Inner product saturation level	**16**
Bias addition input scaling	16
Bias addition output scaling	32
Linear gain factor	2
Saturation level	**16**

Table 8.10: Network I SC Inference Parameters - Scenario 3

Hidden Layer Parameters	
Input scaling	1
Number of nodes in the decomposed inner product	8
Inner product intermediate scaling	4
Inner product intermediate gain factor	2
Inner product intermediate saturation level	2
Inner product output scaling	16
Inner product output gain factor	4
Inner product output saturation level	**4**
Bias addition input scaling	4
Bias addition output scaling	8
Linear gain factor	2
Saturation level	**4**

Output Layer Parameters	
Input scaling	4
Number of nodes in the decomposed inner product	4
Inner product intermediate scaling	32
Inner product intermediate gain factor	2
Inner product intermediate saturation level	16
Inner product output scaling	64
Inner product output gain factor	4
Inner product output saturation level	**16**
Bias addition input scaling	16
Bias addition output scaling	32
Linear gain factor	2
Saturation level	**16**

Once again, saturation levels are determined from Figure 8.7 so that the vast majority of values can be represented in stochastic computing without saturation error. Similar to Section 8.1, for each scenario, SC inference is executed for an increasing bit-stream length and the network accuracy on the test set is recorded and shown in Figure 8.8[7].

Similar to the previous experiment, the SC implementation of Network II based on scenario 3 achieves, within an error of 2.34% and for a sufficiently long bit-stream, the floating-point network accuracy. The main difference, is that for Network II this is achieved with a bit-stream length of 2^{13} instead of 2^{15} that was previously needed, that is 4 times shorter. What is notable though, is that for the same bit-stream length the SC network based on scenario 2 also achieves the reference level of accuracy, whereas in the case of Network I the corresponding scenario required a 64× longer bit-stream. Finally, in this experiment, the accuracy of the SC network based on worst-case scalings also follows an increasing trend, although it does not achieve the reference recognition level, at least for a bit-stream length of 2^{16}.

[7]A base-2 logarithmic scale is used on the horizontal axis

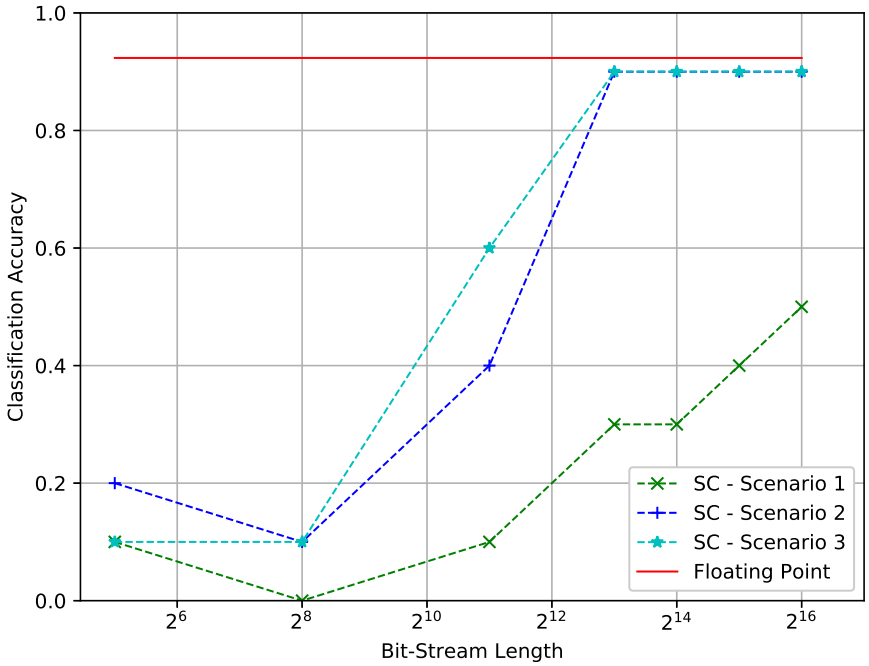


Figure 8.8: Test accuracy of Network II implemented in SC versus the bit-stream length

The observations in the preceded paragraph illustrate that is somewhat easier to implement Network II in stochastic computing rather than Network I. Recall that both networks have the same architecture and are trained on the same dataset using the same optimizer and training parameters. The major difference is that the coefficients of Network II are appropriately regularized for the purpose of SC-based inference. This has a direct impact on the peak signal values in every layer and thereby on the worst-case scaling parameters of the SC network graph. Specifically, Network II has significantly smaller scaling parameters than Network I, thus smaller linear gain parameters need to be imposed throughout the network graph. In general, as the gain parameter increases, the range of input values that is preserved (without compression) by the saturating block decreases and the transfer function in the preserved range becomes more steep. Thus, a larger linear gain block is more likely to incur higher error in stochastic computing, especially if many of such blocks are cascaded with each other.

In summary, the implementation of Network II in stochastic computing can be achieved by means of a shorter bit-stream and without employing the inner product with decomposition which incurs higher saturation overheads than the conventional MUX-based inner product. This experiment greatly illustrates how neural network inference in stochastic computing can benefit by appropriately regularizing the weights to have an absolute value less than one, i.e. following (even partially) the SC training process.

8.3 Training Stochastic Computing Compatible Networks

Following the implementation of neural network inference in stochastic computing, the main objective of this section is to conduct experiments with the training procedure proposed in Chapter 6 and investigate its effects on the overall performance of the neural network on the recognition task. Experiments are still based on the MNIST dataset and as a starting point a network based on the configuration listed in Table 8.11 is deployed and trained.

Table 8.11: Network III Characteristics

Network Architecture		
Input Layer	Number of Units	784
Hidden Layer	Number of Units Hidden Unit	128 Linear
Output Layer	Number of Units Output Unit	10 Softmax

Training Parameters	
Optimizer	Adam
Loss Function	Cross Entropy Error
Initial Learning Rate	0.1
Epochs	5000
Batch Size	500
SC Compatible Training	Yes
Penalty Function	L^2 Regularizer

This network, namely Network III, has the same architecture as the previously developed models, i.e. Networks I and II, but is trained according to the proposed SC training process. That is, the neuron architecture in Figure 6.2 is utilized along with the optimization-based scaling scheme to allow the network to learn optimal saturation levels during training. Furthermore, L^2 regularization is used to constraint the weights and biases to have absolute value less than one. Generally, it was observed that a larger number of training epochs is required for the modified network architecture to achieve an acceptable recognition accuracy on the training and testing sets. Thus, the number of epochs is increased to $5,000$ and a batch size of 500 samples is used in each epoch to compute the gradients and update the model's parameters. At this stage, it is assumed that the bit-stream length that would have been used in a stochastic computing implementation is sufficiently long so that the variance of the additive Gaussian noise is effectively zero, i.e. no Gaussian noise is added on the floating-point network output.

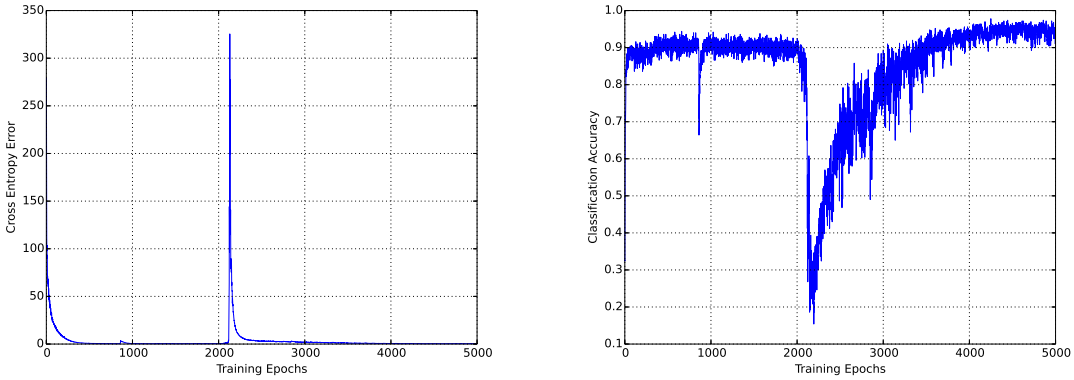
In Section 6.4.1 two distinct design approaches were presented regarding the optimization-based scaling scheme: 1) A common gain coefficient is applied to all the SC-based neurons in a layer and 2) A unique gain coefficient is applied to every SC-based neuron in a layer. For the purpose of this experiment, both cases are considered and results are presented and analysed for both. Furthermore, in Chapter 6, it was stated that only the gain coefficients at the output of a scaled inner product will be considered as trainable parameters, whereas the gain coefficients associated with the addition of the bias term will be fixed to two. Throughout the preceded experiments, it was generally observed that the range of values spanned by the signals before and after the addition of the bias term is effectively the same. Thus, setting the gain coefficient associated with that operation to two, i.e. remove the down-scaling effect of the stochastic adder, is indeed a reasonable design choice.

8.3.1 Case 1 - Common Gain Coefficients

First, experiments concerned with the common gain design choice were conducted and results are presented and analysed below[8]. The trained network achieves a classification accuracy of 93.39% on the training set and 93.55% on the testing set[9]. Furthermore, the optimal gain factor in the hidden layer was found to be 22.28 and in the output layer 24.33. Figure 8.9 shows the loss function and classification accuracy on the training set as a function of the training epochs. Similarly, Figure 8.10 shows the evolution of the gain parameters in each layer. Furthermore, the maximum signal value, out of all data points, prior clipping to ± 1 in every saturated operation is recorded and

[8] Generally, it was observed that initialization of the gain parameters may affect the overall optimization procedure. Throughout these experiments gain parameters were randomly initialized from a uniform distribution. Empirically, it was found that a uniform distribution with parameters $a = 0$ and $b = 16$, i.e. $\mathcal{U}(0, 16)$, provides a reasonable initialization range.

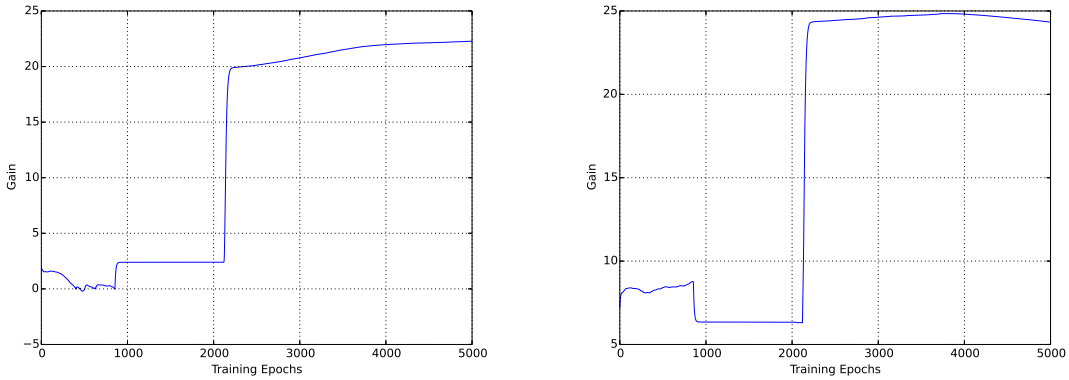
[9] A value of $\lambda = 0.0012$ was empirically found to be optimal, where λ is the regularization scale.



(a) Training loss versus training epochs

(b) Training accuracy versus training epochs

Figure 8.9: Evolution of the loss function and classification accuracy of Network III during training



(a) Gain parameter in the hidden layer

(b) Gain parameter in the output layer

Figure 8.10: Evolution of the gain parameters in each layer of Network III during SC compatible training

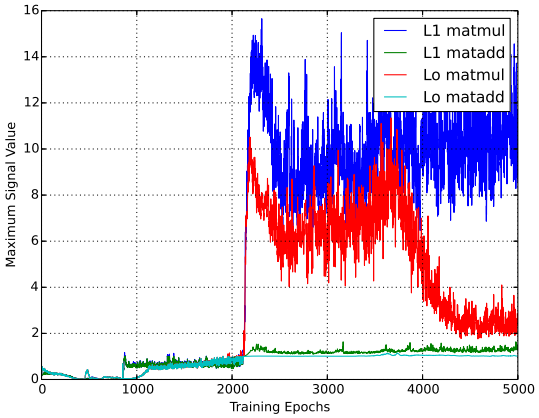


Figure 8.11: Maximum signal value for each saturated operation prior clipping to ± 1

shown in Figure 8.11[10].

[10]The term "L1" is used to denote the first (i.e. hidden) layer and the term "Lo" to denote the output layer. Similarly, "matmul" denotes the operation $\boldsymbol{XW}$, i.e. weight multiplication, and "matadd" denotes the operation $\boldsymbol{X} + \boldsymbol{b}$, i.e. the addition of the bias term

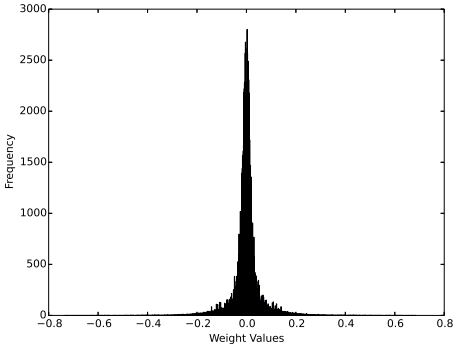
Throughout the experiments, it was generally observed that a sharp increase in the minimization of the loss function, like the one in Figure 8.9a, is related to a sudden increase in the gain parameters. By observing Figures 8.9a and 8.10, it is obvious that the spike in the minimization of the cross entropy error occurs at the same epoch were the two gain parameters increase suddenly. A reasonable interpretation is that the increase in the gain parameters causes many signal values to exceed the range $[-1, 1]$. Once they are clipped to ± 1 a large saturation error in the computations is incurred which in turn causes the sudden increase in the loss function as well as a deep notch in the training accuracy. What is more interesting though, is that despite this disturbance to the minimization of the loss function, the remaining trainable parameters (i.e. weights and biases) are appropriately modified by the training algorithm to counterbalance the disturbance as the epochs increase and make sure that the loss is indeed minimized and classification accuracy is increased. This is indeed a desirable outcome. The fact that the loss reduces again after the spike and the accuracy increases, and in fact exceeds the previously achieved recognition level, is indicative of the fact that it is reasonable to re-train the weights and biases in the neural network to adapt to the new gain parameters (i.e. saturation levels) that network would like to introduce.

Furthermore, as shown in Figure 8.11, prior the sudden gain increase, signal values are constraint to lie within the interval $[-1, 1]$[11]. Clearly the sharp increase in the maximum signal value after weight multiplication in both the hidden and output layer, i.e. blue and red plot, occurs at the same point where the gain parameters of the network rise suddenly. Interestingly, although the network appropriately adapts its weights and biases to the new gain coefficients in order to ensure the loss function is minimized, it does not compensate for the signal values that initially exceeded the range $[-1, 1]$ due to the increased gain. That is, there still exist some data that will cause a large saturation error when the signal values are clipped to ± 1, but the network does not compensate for those. In fact, to accommodate such data, one would have to be very conservative with the scaling parameters (and thereby the gain coefficients), consequently loosing quite a lot of precision in the computations. In other words, the scaling parameters should be kept large enough so that the resulting computations can be accommodated without saturation error. Remarkably, the network takes such a decision on its own during training and decides to reduce the scaling parameters, by means of increasing the gain factors, to avoid loosing precision. Finally, Figure 8.11 only shows the maximum activation value, so it only indicates that there exist some data that will cause saturation error. However, it does not indicate how frequently such data occur. Clearly, they must not occur so often otherwise the network would not have selected to modify in such a way the gain parameters and such phenomena would not have been observed in the first place.

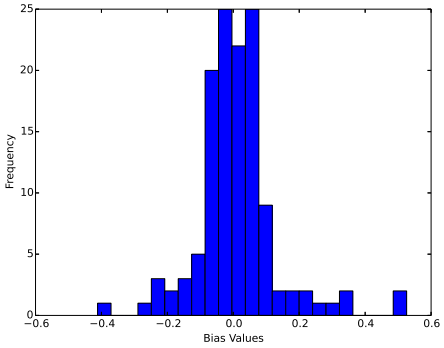
Desirably, the recognition accuracy of the network trained by means of the SC compatible training process is higher than the classification accuracy of both Network I and Network II. In a sense one might expect such an increase as additional design variables are introduced to the network. As analysed though, these design variables have a direct interpretation in stochastic computing hardware which makes this result particularly appealing.

Finally, histogram plots for the weights and biases of the trained network are shown in Figure 8.12. L^2 regularization was selected for the purpose of constraining the network's coefficients, as for the particular problem it preserves the inherent bell-shaped distribution of the weights. Alternatively, L^1 regularization or the custom penalty function could be used. Generally L^1 regularization was found to perform poorly on this task. On the other hand, the custom regularizer, which is subsequently considered, is still a viable option.

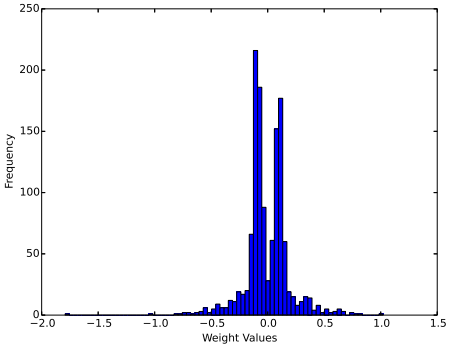
[11] Since the maximum signal value, shown in Figure 8.11, is less than one then all signal values are less than one



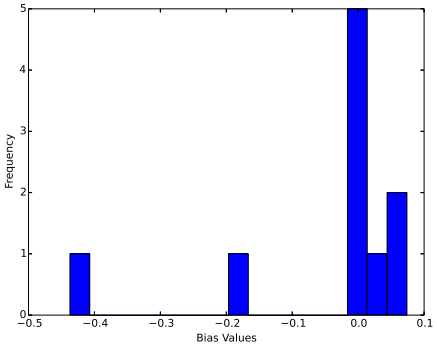
(a) Weights in the hidden layer



(b) Biases in the hidden layer



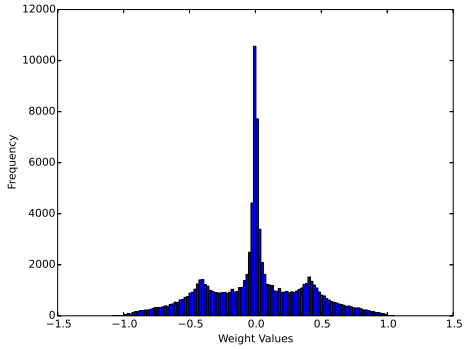
(c) Weights in the output layer



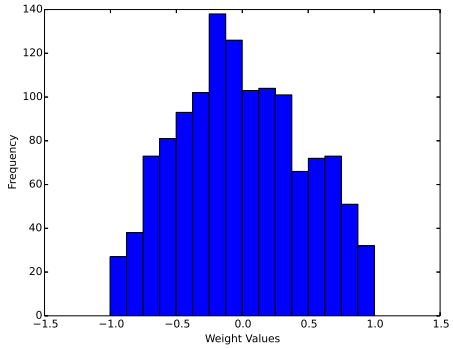
(d) Biases in the output layer

Figure 8.12: Histogram plots of the trainable parameters of Network III with L^2 Regularization

The training procedure is now repeated using the custom penalty function. Desirably, the network achieves a recognition accuracy of 94.29% on the training set and 93.67% on the test set, which is slightly higher than that obtained when L^2 regularization was applied. In contrast to the preceded experiment, no spikes were observed throughout the loss minimization of this network. In fact, such a spike-free behaviour was observed multiple times when the training procedure was repeated. Occasionally, spikes in the loss minimization were observed for this type of regularizer as well, and in those cases the network had the same behaviour as the one seen above. Histogram plots for the network weights and results from the SC compatible training process of this network are shown in Figures 8.13 and 8.14 respectively.

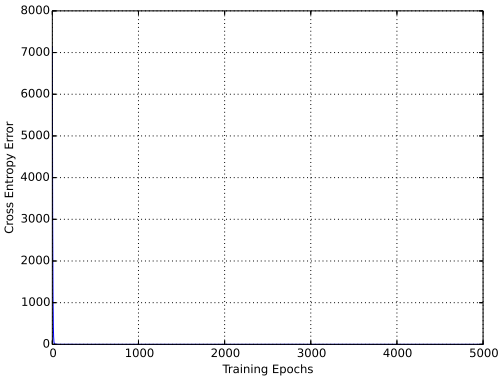


(a) Weights in the hidden layer

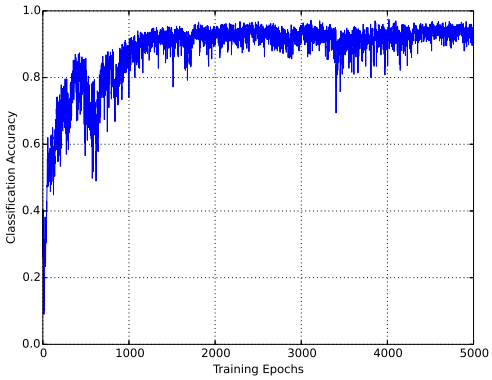


(b) Weights in the output layer

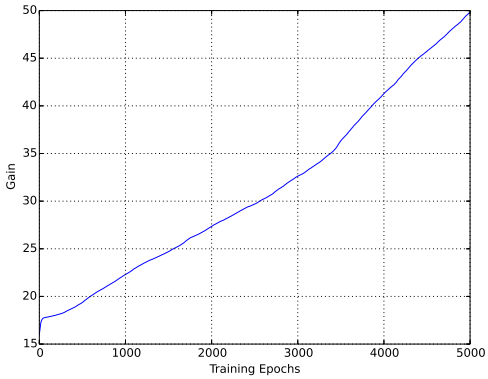
Figure 8.13: Histogram plots of the weights of Network III when the custom regularizer is employed



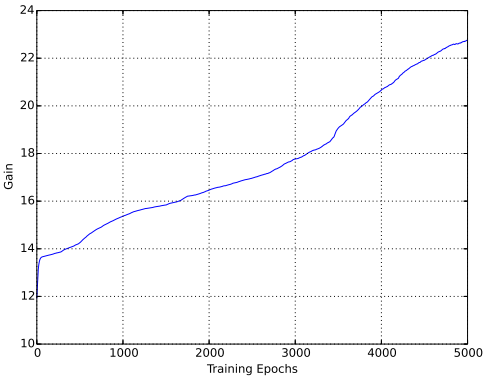
(a) Training loss versus epochs



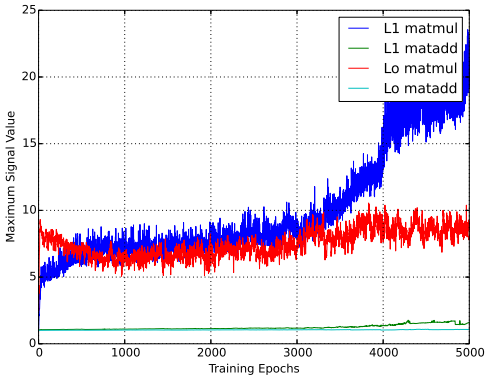
(b) Training accuracy versus epochs



(c) Gain parameters in the hidden layer



(d) Gain parameters in the output layer



(e) Max signal value prior clipping to ± 1

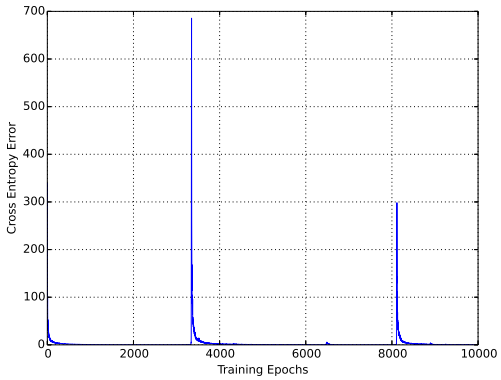
Figure 8.14: Results from the training process of Network III using the custom penalty function

A reasonable interpretation for the absence of spikes in the loss minimization could be the fact that there are no abrupt changes in the gain coefficients. Instead, both gain parameters increase smoothly during training. Consequently, the down-scale factors throughout the SC network graph decrease and so does the range of values that is represented without compression error. Nevertheless, this does not seem to affect the cross entropy error, neither the recognition accuracy of the model which remains approximately constant. As already argued, this is indicative of the fact that the network identifies on its own that worst-case scalings are overly pessimistic and appropriately increases the gain coefficients to avoid loosing precision. Once again, there exist some data that will cause a non-zero saturation error when signal values are clipped to ± 1, however the network does not seem to compensate for those.

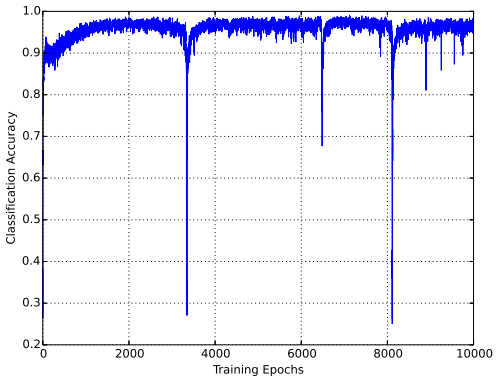
8.3.2 Case 2 - Distinct Gain Coefficients

The experiment is now repeated adopting the second design choice regarding the optimization-based scaling scheme, i.e. a distinct gain coefficient is applied to each neuron in the network. The trained network achieves a classification accuracy of 96.52% on the training set and 95.76% on the test set[12]. Results for this network are collectively shown in Figure 8.15.

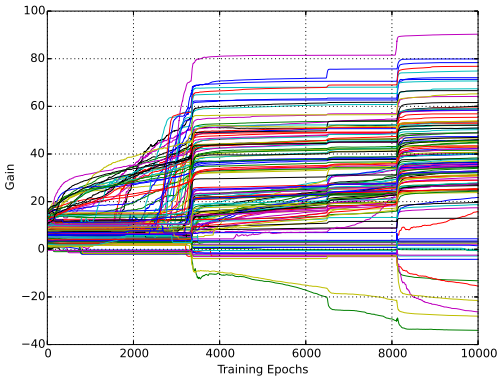
Perhaps unsurprisingly, the behaviour observed in the preceded experiment is noticed in this one as well. That is, sudden increments in the gain coefficients, which control the saturation levels of the SC neuron, cause a sharp increase in the loss function during training. Once again, the network seems to appropriately adapt the remaining trainable parameters to the updated gain coefficients in order to ensure that the loss is minimized. As Figure 8.15e suggests, there exist some data that will cause a large saturation error once signal values are clipped to ± 1. Nonetheless, as previously argued, it seems that the network consciously takes the decision to increase the gain coefficients (and thereby decrease the scaling factors), to avoid loss of precision due to large down-scale parameters in the network graph. In a sense, the network prefers to increase precision in the computations at the cost of a non-zero saturation error for some data points. A reasonable interpretation is that such data points occur rarely, thus it is preferable not to precisely accommodate computations related to those points and instead reduce the scaling coefficients to facilitate computations for the remaining data points with higher precision.



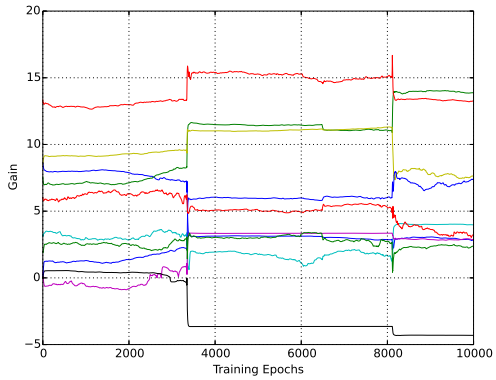
(a) Training loss versus epochs



(b) Training accuracy versus epochs

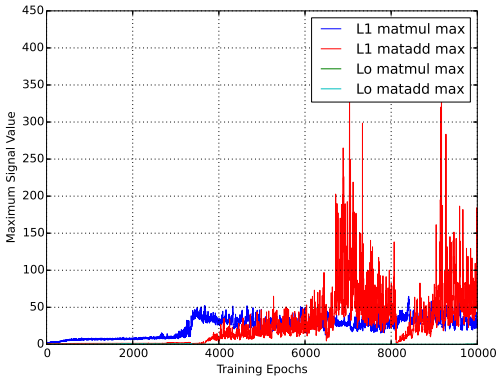


(c) Gain parameters in the hidden layer



(d) Gain parameters in the output layer

[12] A value of $\lambda = 0.0001$ was empirically found to be optimal.

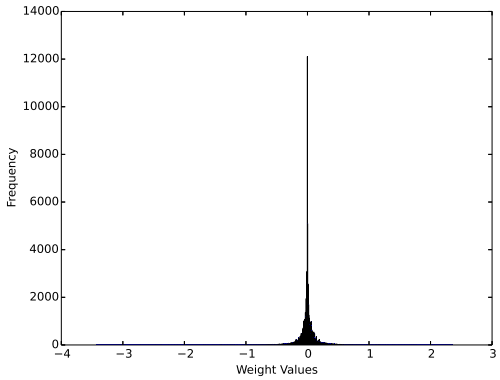


(e) Max signal value from each saturated operation prior clipping to ± 1

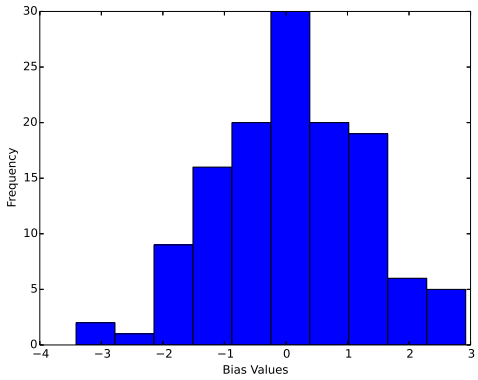
Figure 8.15: Results from the training process of Network III with a distinct gain coefficient applied to every neuron

Histogram plots of the weights and biases in each layer are shown in Figure 8.16. Although the majority of the weights has absolute value less than one, there exist some weight coefficients that do not lie inside the range $[-1, 1]$. It may indeed be possible to appropriately increase the regularization scale λ so that all weights have absolute value less than one, without degrading the recognition accuracy of the model. However, this is not the primary concern of this experiment. Furthermore, depending on the neural network architecture and the way that the inner product computation is implemented in SC, it may not be catastrophic to have weights outside the range $[-1, 1]$. Especially if for that selections of coefficients the performance[13] of the model on the recognition task has improved, as it happens in this situation. For instance, the stochastic inner product implementation given in Algorithm 2 does not require weight coefficients to be converted into stochastic bit-streams. Thus, the only consequence of having larger weight values is that the down-scale parameter of the inner product will be larger.

For comparison purposes, the training procedure is repeated using the custom penalty function introduced in Section 6.3.3. For this type of regularizer, the trained network achieves an accuracy of 95.99% on the training set and 94.82% on the testing set. The results are very similar to the ones obtained in previous experiments, where sudden increments in the gain coefficients gave rise to spikes in the loss minimization. For that reason they are included in Appendix B.1.

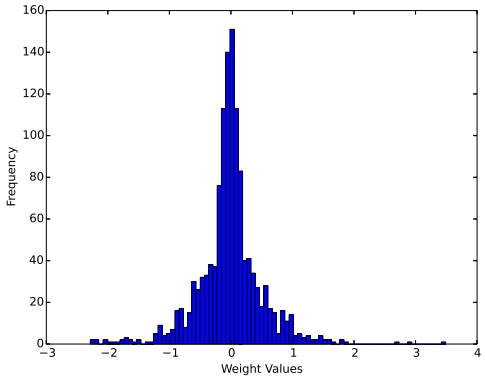


(a) Weights in the hidden layer

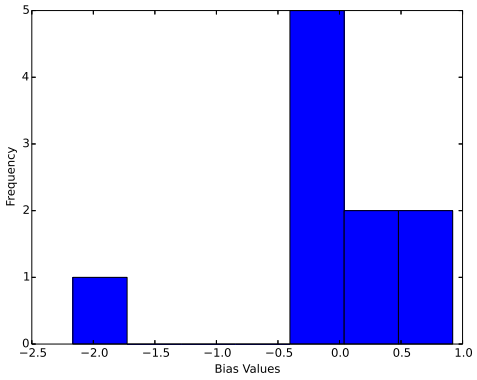


(b) Biases in the hidden layer

[13] In terms of the metric function used to quantify the quality of the model's predictions



(c) Weights in the output layer



(d) Biases in the output layer

Figure 8.16: Histogram plots of the trainable parameters of Network III - Case 2 with L^2 Regularization

8.4 Summary and Conclusions

This chapter presented experimental results from the implementation of neural network inference in SC as well results related to the training of a SC compatible neural network.

Implementing neural network inference in stochastic computing was the first major milestone of this project. It was found, that subject to certain conditions, this can indeed be achieved. A major challenge throughout this process is to effectively handle the scalings introduced by the operations of addition and inner product in stochastic computing. Especially for the later, the down-scaling parameter can be significantly large, undermining precision in the computations. Indeed none of the attempts based on the inherent worst-case scalings was successful. Thus, it is essential to apply saturation arithmetic to reduce the scalings and increase precision. In that case, it was found that the convergence of the SC network accuracy is of the form $\sqrt{L}$, where L is the bit-stream length.

Furthermore, it was found that neural network inference in stochastic computing can significantly benefit by appropriately regularizing the weights to have absolute value less than one. That is, when the trained network is aware of the subsequent implementation of inference using stochastic computing hardware. Constraining the weights within the interval $[-1, 1]$ can significantly reduce the scalings throughout the network graph. This in turn, implies that smaller gain parameters need to be imposed by the designer in the event where saturation arithmetic is used. As it was found in Section 8.2, a major benefit of following the proposed regularization technique, is that the implementation of neural network inference in stochastic computing may be realised by means of a shorter bit-stream. Such a consequence is of course desired as it reduces the computational time.

Extending the implementation of inference in stochastic computing, the limitations of stochastic arithmetic were taken into account during the training phase by means of the proposed SC compatible training process. Apart from constraining the weights within $[-1, 1]$, this process employs an optimization-based scheme to learn optimal scaling parameters during training. As argued several times, worst-case scalings tend to be overly pessimistic, catering for situations that are extremely rarely encountered with practical input data. This is in fact not true just for a stochastic computing system but also for a conventional binary system using a fixed-point representation.

Humans are indeed able to understand that concept and perhaps manually develop techniques to alleviate its effects. Of course those techniques will depend on the number representation used by the system. In stochastic computing, worst-case scalings can be mitigated by saturation arith-

metic where a linear gain with saturation is applied at the output of a stochastic adder. What is remarkable though, is that allowing the gain parameters to be trainable, permits the network to develop its own knowledge regarding both the recognition task as well as the alternative representation that we are trying to impose. During the training process, the network seems to understand the limitations of the stochastic representation and appropriately modifies its parameters. The fact that gain coefficients increase during training, either abruptly or smoothly, is indicative of the fact that the network identifies on its own that worst-case scalings are overly pessimistic and learns optimal scaling values to avoid loosing precision. This is done without degrading the performance of the model on the recognition task, which is of course the primal objective. Instead, it was found that this training approach can even improve the recognition accuracy of the model, both in and out of sample testing.

Chapter 9

Evaluation

This chapter aims to conduct a critical appraisal of the work done in this project compared to the original objectives and present relative merits but also deficiencies. Starting from the objectives set out in the requirements capture, the primal purpose of this project was to investigate under which circumstances stochastic computing can be incorporated within deep neural network models. Threinafter, this task was broken into two major objectives

1. Neural network inference in stochastic computing
2. Neural network training in stochastic computing

9.1 Neural Network Inference

The majority of this work was taken up with the analysis and implementation of stochastic processing units employed in ANNs as well as how scalings need to be handled for the purpose of implementing neural network inference in stochastic computing. The basic operations employed in both feedforward but also convolutional neural networks were implemented in stochastic computing. These are,

- Addition
- Multiplication
- Inner Product
- Max and Average Pooling
- Hyperbolic Tangent
- Rectified Linear Unit

To a large extend, the implementation of these units, in exception to the ReLU, was based on the prior work by Gaines [17] and Brown and Card [6]. Still, extensions were made to some of these blocks. For example, the inner product architecture proposed by Gaines was extended to facilitate a weighted sum with arbitrary input weights. On the contrary, there was no prior work regarding the implementation of the ReLU in stochastic computing. Previous attempts to design SC hardware targeting deep neural networks [30, 41] only considered the use of the hyperbolic function that was proposed by Brown and Card.

In terms of evaluating the aforementioned units, their functionality was verified empirically through simulation and for the majority of these units a mathematical justification was given as well. An inherent weakness of the MUX-based inner product is the large down-scale coefficient introduced to the output signal which may potentially lead to accuracy issues. The $SReLU$, although functional, does not utilise the full dynamic range of SC to represent the output signal.

Since the output of the proposed $SReLU$ is encoded as a bipolar stochastic signal, only half of the dynamic range of SC is used. This is due to the fact that for an input signal $x \in [-1, 1]$ the output of the $ReLU$ y will always lie within $[0, 1]$. Thus to employ the full dynamic range, the output of the $SReLU$ must be encoded as a unipolar signal, as in the case of the $Sexp$ unit. This would of course require to devise a new architecture for the $SReLU$ satisfying the aforementioned characteristics, if such a unit exists. However, in the event where the output of the $SReLU$ is encoded in the unipolar format, to further process this signal in a bipolar system (e.g. a bipolar SC network) one must take into account the difference in the representation.

Perhaps a possible approach would be to convert the unipolar signal into a bipolar one through the mapping defined by $y = 2x - 1$, where $x \in [0, 1]$ the unipolar $SRELU$ output. Yet, to perform such a transformation within the SC domain, the linear gain block with saturation will need to be employed to realise the $2\times$ operation. If that is the case then there exists the possibility of a non-zero compression error if $x \in (0.5, 1]$, so this may not be in fact an optimal approach to follow. Neither the option to convert the unipolar signal to its floating-point representation, perform the mapping and then convert the result back to a bipolar signal seems to be an optimal approach to follow as the two conversion processes will introduce a significant delay overhead in the overall computation.

Following the design and analysis of individual processing elements in SC, a significant part of the work on the implementation of neural network inference in stochastic computing was devoted to the analysis of the scalings of the individual processing units. The outcome of this study was the scaling scheme presented and analysed in Chapter 5. In fact, none of the previous attempts presents in a detailed manner how the scalings are handled throughout the SC network graph. The proposed scaling scheme is based on forward propagation of known information on data ranges through the network graph. These data ranges are derived from the worst-case scalings of the individual operations. While in principle this approach is functional, empirically it was found that worst-case scalings tend to be overly pessimistic, significantly reducing the precision in the computations. Following the results presented in Chapter 8, if no saturation arithmetic is applied a very long bit-stream will be needed for the SC network to reliably emulate the functionality of the conventional network. This could in fact exceed the break-even point, implying that it is advantageous to simply use an alternative representation.

Nevertheless, subject to saturation arithmetic, the results in Chapter 8 show that is indeed possible to implement neural network inference in stochastic computing. This is of course a desirable and promising outcome as the implementation of SC-based inference was a major objective of this project. Still, additional experiments need to be done. For example, deeper network architectures need to be considered as well as testing on alternative datasets. Furthermore, convolutional network architectures should be considered as well, as they are nowadays the dominant approach to many recognition tasks. The main reason why these extensions were not carried out is lack of time. As the entire project started from scratch, a significant amount of time was devoted to the analysis and development of the techniques used, thus only some base cases could be considered within the available timeframe.

In fact the work presented in this project presents all the processing units that needed to implement a convolutional neural network in stochastic computing. That is, inner product, pooling and activation functions. Although not analysed in detail, it may in fact be somewhat "easier" to implement convolutional neural networks in SC. The reason to argue for this is as follows. A major challenge in the implementation of inference in SC comes from the large scalings that are imposed throughout the graph. These scalings, mainly arise from the inner product unit and increase as the number of inputs to the inner product increases. In a fully connected multilayer perceptron,

each neuron will be connected to all neurons in the previous layer. Thus, the number of inputs to a certain neuron may be significantly large leading to a large scaling factor. For example, if the input volume has size $32 \times 32 \times 3$ then a neuron in the first layer will have 1024 inputs. On the other hand, in a convolutional network it is common to use a smaller *receptive field*, i.e. connect each neuron to only a local region of the input volume. In such a case, the scalings introduced by each inner product in a CNN may be significantly smaller compared to a MLP depending on the size of the receptive field.

In summary, the objectives set out in the requirements capture regarding neural network inference have been successfully achieved. Some of them both qualitatively as well as quantitatively. In the absence of a hardware implementation, the last two objectives were qualitatively analysed throughout the report.

9.2 Neural Network Training

The major objective of this task was to investigate an alternative training process that will take into account the limitations of stochastic computing. In fact, such an extension has not been considered in any aspect of related work, as most attempts to apply SC hardware to DNNs only consider the inference stage of a network.

It was found that stochastic arithmetic affects both the forward as well as the backward propagation of a neural network. Although training in SC was effectively *modelled* using floating-point computations instead of simulating the implementation of neural network training in SC, very insightful conclusions were made. In particular, it was found that the implementation of neural network inference in SC can indeed benefit if the network is trained by means of the proposed modified procedure. It turns out that it can also improve the overall performance of the network on the recognition task, i.e. increase recognition accuracy on both the training and testing datasets.

Perhaps a possible drawback of this approach is that it may introduce overfitting while training the neural network. This is due to the fact that additional constraints but also decision variables are introduced to the model. In general, it is not straightforward to perform generalization analysis for DNNs, to predict in advance what will be the impact of these changes on the performance of the network. Most likely, this will depend on the particular dataset, network architecture as well as the regularization techniques used to avoid overfitting.

While the original intention was to simulate the training using the developed SC processing units, this would be impracticable within the allowed timeframe mainly because simulating the training on a CPU platform would incur exceedingly long computational times. Thus, it was decided to model the effects of SC by means of the neuron architectures proposed in Chapter 6. Nevertheless, given the implementation of the necessary computational elements in stochastic computing, a subsequent work could consider to appropriately modify those targeting high performance computing devices like multi-core CPUs or GPUs. This is in fact feasible as ANNs are massively parallel systems, thus the code written for the implementation of such systems is highly parallelizable.

Such an extension will allow to simulate, within reasonable execution times, the training of a neural network using stochastic computing, and perhaps obtain a more accurate indication regarding the effects of stochastic computing during the training of a neural network. Furthermore, it will allow to conduct experiments regarding the dynamic selection of the bit-stream length L in each training epoch. A significant amount of experimentation could be made based on that line, and interesting results and observations could arise, perhaps changing the way we nowadays train neural network models.

9.3 Summary of Contributions

Finally, a summary of the main contributions of this project is listed below.

- The extension of the inner product architecture proposed by Gaines to facilitate computations with arbitrary input weights.
- The introduction of a stochastic rectified linear unit.
- The formulation and implementation of saturation arithmetic in stochastic computing based on the linear gain FSM proposed by Brown and Card.
- A thorough analysis of the scaling scheme used to implement neural network inference in stochastic computing.
- The formulation of the SC compatible training process and development of neuron architectures to realise such a procedure.
- An evaluation of the proposed approach for both neural network inference as well as training in stochastic computing.

Chapter 10

Conclusion and Further Work

This project considers whether stochastic computing, a low-cost alternative to conventional binary computing, can be used to implement modern deep neural networks. It was found that the worst-case scaling parameters that are inherently introduced by stochastic arithmetic tend to be overly pessimistic, undermining the implementation of neural network inference in SC. Nevertheless, it was shown that by appropriately applying saturation arithmetic, the SC network can achieve the same level of accuracy as the conventional floating-point network. Throughout this process, the project has presented some of the major challenges posed by stochastic computing as well as possible solutions to address them.

Extending the implementation of neural network inference in stochastic computing, a modified training procedure was proposed aiming to capture the limitations of the stochastic representation within the training phase of the model. Interestingly, it was found that this allows the network to develop its own knowledge regarding both the recognition task as well as the alternative representation that we are trying to impose. The network seems to identify the limitations of stochastic computing and appropriately modifies its parameters to address them. As a consequence, a subsequent implementation of the inference algorithm using SC hardware could benefit significantly by this training procedure. Finally, it was found that the proposed training approach can even improve the network's predictions, both in and out of sample.

Although a significant amount of work has been presented in this project, further research can be conducted. First of all, it seems natural to consider a hardware implementation of SC neural networks on an FPGA device. Extending the current work, a hardware implementation will allow to obtain quantitative results regarding the area consumed by the SC network as well as power dissipation and throughput and compare those with existing hardware platforms. It will also allow to quantitatively access the overheads introduced by saturation arithmetic in SC as well as to identify if there exists a break-even point. That is, a point where the execution time incurred by the SC network in order to achieve the required level of accuracy is so long that a conventional implementation would be preferred.

Furthermore, as discussed in Chapter 9, additional experiments need to be conducted. Deeper network architectures need to be considered as well as testing with alternative datasets. Moreover, the implementation of convolutional neural networks should be addressed. These extensions could be studied either by means of a software or a hardware implementation. Finally, another major aspect of future work would be to simulate the training of a neural network using SC processing units instead of modelling this process using floating-point computations. As argued in the evaluation chapter, such an extension will allow to conduct experiments regarding the dynamic selection of the bit-stream length during training which could potentially give rise to very interesting results and observations.

Bibliography

[1] Martín Abadi, Ashish Agarwal, and Paul Barham et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Ossama Abdel-Hamid, Abdel-Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 22(10):1533–1545, October 2014.

[3] Armin Alaghi and John P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013.

[4] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. *CoRR*, abs/1606.05487, 2016.

[5] Chris M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Comput.*, 7(1):108–116, January 1995.

[6] Bradley D. Brown and Howard C. Card. Stochastic neural computation i: Computational elements. *IEEE Trans. Comput.*, 50(9):891–905, September 2001.

[7] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Synthesis of saturation arithmetic architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):334–354, July 2003.

[8] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. *Synthesis And Optimization Of DSP Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[9] Florent de Dinechin, Milos Ercegovac, Jean-Michel Muller, and Nathalie Revol. Digital Arithmetic. In Benjamin Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*, pages 935–948. Wiley, 2009.

[10] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[11] Miloš D. Ercegovac and Tomás Lang. Chapter 1 - review of basic number representations and arithmetic algorithms. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 3 – 49. Morgan Kaufmann, San Francisco, 2004.

[12] Miloš D. Ercegovac and Tomás Lang. Chapter 2 - two-operand addition. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 50 – 135. Morgan Kaufmann, San Francisco, 2004.

[13] Miloš D. Ercegovac and Tomás Lang. Chapter 3 - multioperand addition. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 136 – 179. Morgan Kaufmann, San Francisco, 2004.

[14] Miloš D. Ercegovac and Tomás Lang. Chapter 4 - multiplication. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 180 – 245. Morgan Kaufmann, San Francisco, 2004.

[15] Miloš D. Ercegovac and Tomás Lang. Chapter 5 - division by digit recurrence. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 246 – 329. Morgan Kaufmann, San Francisco, 2004.

[16] Miloš D. Ercegovac and Tomás Lang. Chapter 8 - floating-point representation, algorithms, and implementations. In Miloš D. Ercegovac and Tomás Lang, editors, *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 396 – 487. Morgan Kaufmann, San Francisco, 2004.

[17] B. R. Gaines. *Stochastic Computing Systems*, pages 37–172. Springer US, Boston, MA, 1969.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of FPGA based neural network accelerator. *CoRR*, abs/1712.08934, 2017.

[20] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[21] P. Jeavons, D. A. Cohen, and J. Shawe-Taylor. Generating binary sequences for stochastic computing. *IEEE Transactions on Information Theory*, 40(3):716–720, May 1994.

[22] Yuan Ji, Feng Ran, Cong Ma, and David J. Lilja. *A hardware implementation of a radial basis function neural network using stochastic logic*, volume 2015-April, pages 880–883. Institute of Electrical and Electronics Engineers Inc., 4 2015.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[24] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 124:1–124:6, New York, NY, USA, 2016. ACM.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[27] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[28] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, Accessed:[29 May 2018].

[29] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 54:1–54:12, Piscataway, NJ, USA, 2016. IEEE Press.

[30] J. Li, A. Ren, Z. Li, C. Ding, B. Yuan, Q. Qiu, and Y. Wang. Towards acceleration of deep convolutional neural networks using stochastic computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 115–120, Jan 2017.

[31] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan. Logical computation on stochastic bit streams with linear finite-state machines. *IEEE Transactions on Computers*, 63(6):1474–1486, June 2014.

[32] E. László, P. Szolgay, and Z. Nagy. Analysis of a gpu based cnn implementation. In *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, pages 1–5, Aug 2012.

[33] Bohdan Macukow. Neural Networks – State of Art, Brief History, Basic Models and Architecture. In Khalid Saeed and Wladyslaw Homenda, editors, *15th IFIP International Conference on Computer Information Systems and Industrial Management (CISIM)*, volume LNCS-9842 of *Computer Information Systems and Industrial Management*, pages 3–14, Vilnius, Lithuania, September 2016. Springer International Publishing. Part 1: Invited Paper.

[34] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580, Jan 2016.

[35] A. F. Murray and P. J. Edwards. Enhanced mlp performance and fault tolerance resulting from synaptic weight noise during training. *Trans. Neur. Netw.*, 5(5):792–802, September 1994.

[36] W. J. Poppelbaum, C. Afuso, and J. W. Esch. Stochastic computing elements and systems. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 635–644, New York, NY, USA, 1967. ACM.

[37] S. Potluri, A. Fasih, L. K. Vutukuru, F. A. Machot, and K. Kyamakya. Cnn based high performance computing for real time image processing on gpu. In *Proceedings of the Joint INDS'11 ISTET'11*, pages 1–7, July 2011.

[38] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, Jan 2011.

[39] Fletcher R. *Nonlinear Programming*, chapter 12, pages 277–330. Wiley-Blackwell, 2013.

[40] A. Ren, Z. Li, Y. Wang, Q. Qiu, and B. Yuan. Designing reconfigurable large-scale deep learning systems using stochastic computing. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–7, Oct 2016.

[41] Ao Ren, Ji Li, Zhe Li, Caiwen Ding, Xuehai Qian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. SC-DCNN: highly-scalable deep convolutional neural network using stochastic computing. *CoRR*, abs/1611.05939, 2016.

[42] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing – Explorations in the Microstructure of Cognition*, chapter 8, pages 318–362. MIT Press, 1986.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[45] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A cgra-based approach for accelerating convolutional neural networks. In *Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, MCSOC '15, pages 73–80, Washington, DC, USA, 2015. IEEE Computer Society.

[46] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. Institute of Electrical and Electronics Engineers (IEEE), May 2016.

[47] Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. Multi-gpu training of convnets. *CoRR*, abs/1312.5853, 2013.

[48] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[49] Wenlai Zhao, Haohuan Fu, W. Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-cnn: An fpga-based framework for training convolutional neural networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, July 2016.

Appendix A

Code Listing

A.1 Source Code

The source files for this project can be found at `https://github.com/adamosSol/SC-DNN`

A.2 SC TensorFlow Operations

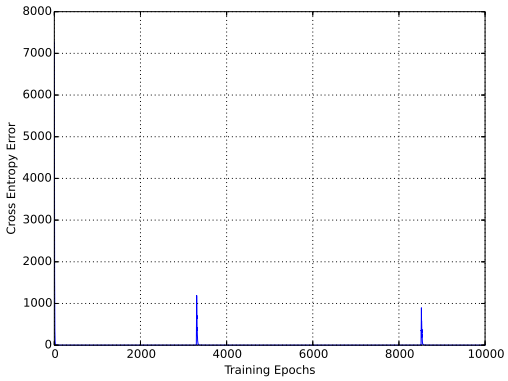
```
import tensorflow as tf
def sc_matmul(x,w,s_in,upscale=False,gain=None):
    """
    Parameters
    ----------
    x: Input Data 2D Tensor
    w: Weight 2D Tensor
    s_in: 1D Tensor with scaling factors of the input activations

    Returns
    -------
    2D Tensor with output activations
    1D Tensor with output scaling factors
    """
    # Bring all the activations under a common scaling factor
    max_scale = tf.reduce_max(s_in)
    rescale_ratio = tf.div(s_in,max_scale)
    x_rescaled = tf.multiply(x,rescale_ratio)

    # Calculate the down-scale coefficients in terms of the weights
    s = tf.reduce_sum(tf.abs(w),axis=0)
    S = tf.reshape(tf.tile(s, [tf.shape(x)[0]]),
                   shape=[tf.shape(x)[0], tf.shape(w)[1]])

    # Matrix multiplication using re-scaled feature data
    y = tf.matmul(x_rescaled,w)
    z = tf.div(y,S)
```

```
    curr_scale = tf.multiply(max_scale,S)

    # Apply gain followed by saturation
    if(upscale):
        z_upscaled = tf.multiply(z,gain) # Supports broadcasting
        new_scale = tf.div(curr_scale,gain)
    else:
        z_upscaled = z
        new_scale = curr_scale

    out_scale = new_scale[0,:]
    z_clipped = tf.clip_by_value(z_upscaled,-1,+1)

    return z_clipped, out_scale

def sc_add(x,b,s_in,upscale=False):
    """
    Parameters
    ----------
    x: Input Data 2D Tensor
    b: Bias 1D Tensor
    s_in: 1D Tensor with scaling factors of the input activations

    Returns
    -------
    2D Tensor with output activations
    1D Tensor with output scaling factors
    """
    # Down-scale bias terms by s_in
    b_scaled = tf.div(b,s_in)

    # Scaled addition using down-scaled bias terms
    y = tf.div(tf.add(x,b_scaled),2)

    curr_scale = tf.multiply(s_in,2)

    # Apply gain followed by saturation
    if(upscale):
        z_upscaled = tf.multiply(y,2)
        out_scale = tf.div(curr_scale,2)
    else:
        z_upscaled = y
        out_scale = curr_scale

    z_clipped = tf.clip_by_value(z_upscaled,-1,+1)

    return z_clipped, out_scale
```

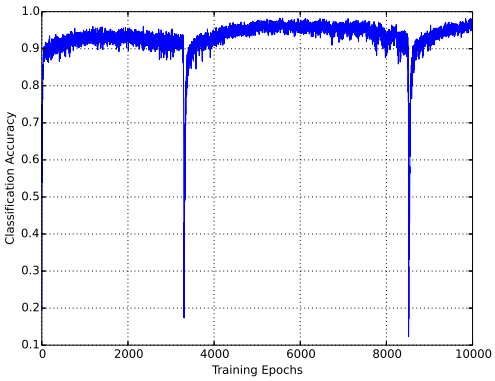
Appendix B

Results

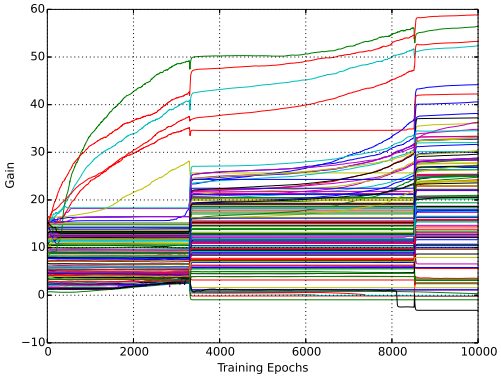
B.1 Network III - Case 2: Custom Penalty Function



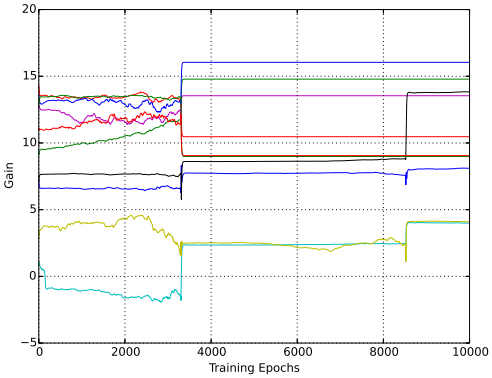
(a) Training loss versus epochs



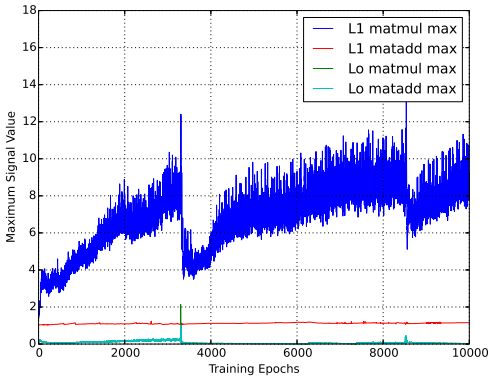
(b) Training accuracy versus epochs



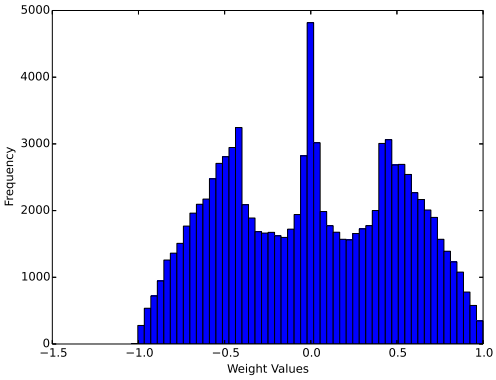
(c) Gain parameters in the hidden layer



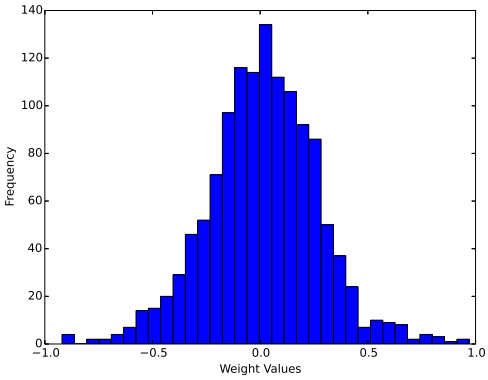
(d) Gain parameters in the output layer



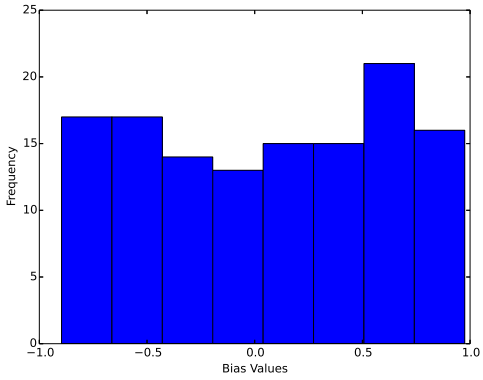
(e) Max signal value from each saturated operation prior clipping to ± 1



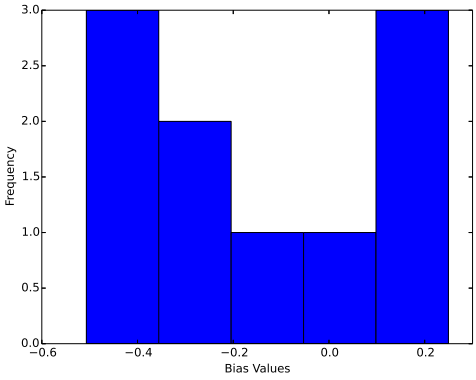
(f) Weights in the hidden layer



(g) Weights in the output layer



(h) Biases in the hidden layer



(i) Biases in the output layer

Figure B.1: Results from the training procedure of Network III when the custom penalty function is employed