REPUBLIC OF TURKEY

YILDIZ TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

LIGHTWEIGHT NEURAL NETWORK ARCHITECTURE USING STOCHASTIC COMPUTING

16011138 — Muhammad Arslan

17011908 — Mecdeddin Harrad

COMPUTER PROJECT

Advisor

Res. Asst. Sercan Aygün

July 2020

ACKNOWLEDGEMENTS

We would like to show our gratitude to our project supervisor, Mr. Sercan Aygün, for sharing his pearls of wisdom with us, from our first encounter with SC and BNNs, through to the final draft of this report. His enthusiasm, knowledge and exacting attention to detail kept our work on track, and helped solve a number of what seemed unsolvable problems at the time. We would also like to thank the university for the opportunity to work at such an exciting and challenging problem.

Muhammad Arslan
Mecdeddin Harrad

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
BN	BatchNorm
BNN	Binarized Neural Network
BPE	Bipolar Encoding
CV	Computer Vision
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
FP	Full-precision
FPNN	Full-precision Neural Network
MAE	Mean Absolute Error
ML	Machine Learning
MSE	Mean Squared Error
MSLE	Mean Squared Logarithmic Error
NN	Neural Network
OOP	Object-Oriented Programming
SC	Stochastic Computing
SCBNN	Binarized Neural Network with Stochastic Computing
SCNN	Stochastic Neural Network
SCFPNN	Full Precision Neural Network using SC
UPE	Unipolar Encoding

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Lightweight Neural Network Architecture using Stochastic Computing

Muhammad Arslan
Mecdeddin Harrad

Department of Computer Engineering
Computer Project

Advisor: Res. Asst. Sercan Aygün

This project aims to experimentally investigate the dynamics of Binarized Neural Networks and SC driven Neural Networks, their fault tolerance and error robustness, how well they interplay together, and effective strategies to train them. To this end, we perform a number of tests, running inference on messy and noisy data with models trained on clean data, and another suite of tests is performed on models trained with corrupted data. Furthur testing is done to specifically analyse the fault tolerance of these approaches by flipping the bits in the network weights or input image data, and then recording the accuracy on test datasets with these faulty parameters / inputs. Lastly, we build a software framework to serve as the foundation for our experiment pipeline.

Keywords: neural networks, stochastic , deep learning, bnn, binarized neural network, binary neural network, mnist-c, stochastic computing, mnist, fashion mnist

1 Introduction

1.1 Overview

One of the reasons for the allure of Machine Learning is due to the power it grants us: the ability to learn "rules" (more specifically, probability distributions) from data instead of having to hard code those rules by hand. This enables us to not just use those models to predict the future (so to speak) but also to interpret the present; to gain insights from the otherwise messy data. And yet, the same algorithms and models that allow us to effectively "see" far across the temporal dimension of the stock market and let us learn the optimal marketing strategy for a given product can't even properly differentiate between a cat and a dog, and are quite bad at playing the sort of games won regularly by secondary / high school students with a 'D' in Mathematics.

That was the situation of AI and ML for quite some time, until work by Hinton et al.[1] enabled us to train "deep" networks (that's where the term *'Deep Learning'* comes from). A simple neural network is composed of multiple layers of neurons stacked together, each neuron acting as a mini ML model. Another way to get some intuition about a fully connected neural network is to think of each layer as a mapping function which changes the representation of the m-dimensional input data to n dimensions, where m in the number of neurons in the previous layer / input data and n is the number of neurons in the current layer. And training the whole network is, conceptually, learning the probability distributions at each layer that produce the optimal representation mapping (a bit more technical explanation is given in section 2.1.2).

Theoretically, neural networks can compute any function [2]. But with this great power comes the need for a lot of computational power, which is one of the reasons deep learning took so long to take off even after we knew how to utilize it. Deep Neural Networks, even at inference time, require quite a lot of memory and CPU/GPU resources, which makes it difficult if not impossible to use them with low power devices (mobile phones, cameras, car sensors, microcontrollers etc). So, some smart

people got together to think of a possible solution, and came up with **Binarized Neural Networks** [3]. In BNNs, each parameter is constrained to only two possible values (+1 or -1), allowing us to represent them solely using bits. This not only decreases the memory size and accesses, but also allows us to replace most arithmetic operations with bit-wise operations, giving us a boost in terms of computational efficiency. More details on BNNs are provided in section 2.1.
Another problem neural networks suffer from is messy and erroneous data, which can be solved during training, but requires us to put a lot of redundant code in our application during run-time (inference time) to ensure the quality of incoming data. This, coupled with the possibility of memory errors in small device, led us to investigate the use of **Stochastic Computing** (SC) as an alternative method for numerical representation and operations.

In our project, we consider using SC as a low-cost alternative to conventional deterministic computing. In this computing paradigm, numbers are represented by bit-streams that can be processed using simple circuits. For example, a bit-stream containing 50% 1s and 50% 0s denotes the number $p = 0.5$ [4]. Arithmetic operations such as multiplication and addition can be performed using an XNOR gate and a multiplexer (MUX) respectively. This helps us reduce the number of gates, leading to reduced hardware cost in terms of computation.
Another attractive feature of SC is its high degree of error tolerance and robustness against noise when compared to conventional binary computing, For example, in binary computing, a single bit-flip could cause a huge error if it affects a high-order bit. However, this is not the case in SC, as the place of a bit does not affect its significance. On the other hand, SC suffers from major drawbacks including accuracy issues in the arithmetic operations due to the randomness of its bit-streams. Furthermore, an increase in the precision of a calculation requires an exponential increase in the length of the bit-stream [5].

1.2 Motivation

Our core motivation for this project was to empirically ascertain the truth value of the claims made in the literature regarding the performance and robustness of both BNNs and SC, and to introduce our own tests into the mix to better understand and explain the dynamics of both the approaches, when independent from each other or used together.
To this purpose, we sought out first to design a platform and framework which was to serve as the basis for our simulations and experiments, and then test different model architectures and hyper-parameters upon it to understand which things work, which

things don't, and why something works or doesn't work (which required tests of its own in order for us to be confident of our hypotheses regarding those dynamics). Lastly, we wanted to introduce errors into our models and input data, and then test them to experimentally verify their robustness to such errors.

2
Preview

2.1 Binarized Neural Networks

2.1.1 Overview

Binary Neural Networks, as the name suggests, are deep neural networks with binary weights and activations (constrained to -1 and +1). They were introduced by Courbariaux et al. [3] as a low cost alternative to conventional neural networks, requiring far less computing power and memory. This does come with a decrease in accuracy, but with proper architecture and hyper-parameter search, one can easily achieve near state-of-the-art results, as is shown both by our work [7.3 & 7.4] and that in the literature.

2.1.2 DNN Training and Inference

Instead of going directly towards BNN training cycle, it would be best to first consider the training of a conventional neural network, and then move towards the necessary modifications for training a BNN.
The training cycle of a DNN (Deep Neural Network) consists of three major steps:

1. The forward pass, where a prediction is calculated by the neural network. Each neuron in a given layer calculates a weighted sum of the input vector from the previous layer. This sum is then passed through an activation function to introduce non-linearity into the network. These outputs then become the inputs for the next layer, until we reach the last layer. Here, we make the actual prediction by passing the pre-activation sum through the output activation function, which may be the same as the hidden layer activation function or different from it. This output (prediction) is then compared to the actual label/target using a loss function (the choice of which depends on the task for which the neural network is being trained, and the type of input data). This loss

is important not only to get an idea of how good the current prediction is, but also to train the network.

2. The backward pass; Here, the loss calculated in the last step is propagated throughout the network by a method known as backpropagation [1]. The crux of this method lies in calculating the gradients of the network parameters (weights, biases, beta and gamma in case of batch norm, and so on) with respect to the loss, and propagating the loss to the previous layer proportional to the input of each neuron and how much it contributed to the loss in the next layer.

3. Weight (/parameter) update: In this step, we apply the previously calculated gradients to the network parameters, using the rule:

$$w_n^{k\,\prime} = w_n^k - \eta \cdot \frac{\partial Cost}{\partial w_n^k} \tag{2.1}$$

   where w_n^k is the nth weight in the kth layer.

2.1.3 Contrast with BNN training

In BNN training (not inference/run-time), two copies of weights are maintained: the full-precision weights as in conventional NNs, and the binarized weights.
At the start of each forward pass, the binarized weights are calculated by binarizing the FP weights, and are used to calculate the preactivations. The preactivation sums are then passed through the sign function:

$$sign\,(x) = \begin{cases} 1 & if\, x \geq 0 \\ 0 & otherwise \end{cases} \tag{2.2}$$

During the backward pass, the gradients are calculated using the binarized weights and activations. These gradients are then applied (in step 3) to the **full-precision weights**, not the binarized weights.

2.2 Stochastic Computing

Stochastic Computing (SC), was proposed as a novel computing paradigm in the 1960's by Brian Gaines [6] and Poppelbaum et al. This section presents an overview of the fundamentals of this computing paradigm.

2.2.1 *Fundamental Principles*

Stochastic computing (SC) is based on probability theory, where a number denoting a given probability is represented by a bit-stream of a chosen length and its value is determined by the probability of a random bit in the bit-stream being 1. For example, a bit-stream containing 25% of ones and 75% of zeros represents the number $p = 0.25$, which means that the probability of observing a 1 in a random bit position is 0.25. This property of SC leads to very low-complexity arithmetic units. For example, multiplication in stochastic arithmetic could be performed using a single AND gate as illustrated in Figure 2.1. However, AND gate can only be used in unsigned stochastic numbers multiplication. To perform signed multiplication, XNOR gate is used, as we'll see later.

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, (0,0,1,1),(1,1,0,0) and (0,1,0, 1) are all valid representations of $p = 0.5$ (using a bit stream of length 4).
This representation leads to 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 2.1. This would change the value represented from 6/8 to 5/8 or 7/8, which is an error of 1/8. On the other hand, a single bit-flip in a conventional binary representation can lead to a significant error if it affects a high-order bit [4].

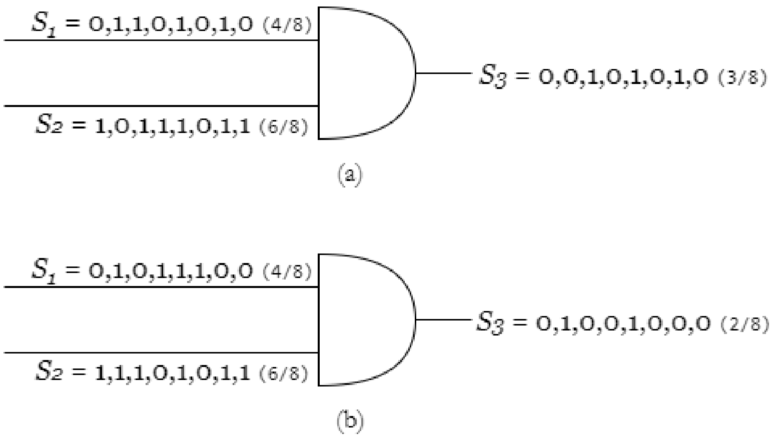

S_1 = 0,1,1,0,1,0,1,0 (4/8)
S_2 = 1,0,1,1,1,0,1,1 (6/8)
S_3 = 0,0,1,0,1,0,1,0 (3/8)
(a)
S_1 = 0,1,0,1,1,1,0,0 (4/8)
S_2 = 1,1,1,0,1,0,1,1 (6/8)
S_3 = 0,1,0,0,1,0,0,0 (2/8)
(b)

Figure 2.1 Multiplication using AND for Unipolar representation: (a) Exact and (b) Approximate computation

Despite these attractive features, SC has some disadvantages that have limited its practical applications. Specifically, the inherent variance in the computations of stochastic arithmetic [7], which comes from the fact that the representation of a stochastic number is not unique. For example, 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 [4]. This can lead to inaccuracy in computations as illustrated in Figure 2.1.
Another drawback that SC suffers from is that more accurate computations require an exponential increase in the number of bits, causing in turn an exponential increase in the number of clock cycles needed to perform a computation [7]. However, an increase in precision does not require an increase in computational resources, the same logic gates can be used to perform more precise arithmetic operations, the only downside is that the latency of computation will get larger.

2.2.2 Encoding Schemes

This section presents the two most popular representations of stochastic numbers. These are the unipolar and bipolar encoding, which represent numbers in the range $[0,1]$ and $[-1,1]$ respectively [6].

Unipolar Encoding

The encoding probability of any positive fractional number x in range $[0,1]$ is given by $P_{UPE} = \frac{x}{N}$, where N is the length of bit-stream.
Decoding on the other hand is represented simply by the ratio of the number of 1s in the bit-stream, to the overall length of the stream. Formally, this is given as $x = \frac{1}{n}\sum_{m=1}^{N} \boldsymbol{X}_m$, where $\boldsymbol{X}$ is a stream of bits and $\boldsymbol{X}_m$ is the m^{th} bit in $\boldsymbol{X}$.

Bipolar Encoding

The encoding probability of any fractional number x in range $[-1,1]$ is : $P_{BPE} = \frac{x+1}{2}$.
Whereas decoding is represented by the formula $x = [(\sum_{m=1}^{N} \boldsymbol{X}_m) - (N - \sum_{m=1}^{N} \boldsymbol{X}_m)]/N$, where $\boldsymbol{X}$ and N are the bit-stream and its length respectively.

2.2.3 Arithmetic Operations

As discussed above, arithmetic operations in SC can be performed using simple logic gates.

Multiplication

Previously, we demonstrated the multiplication for UPE representation in Figure 2.1. However, since the weights and biases can be negative, BPE is going to be used.

Multiplication for BPE representation is done using XNOR gate as illustrated in 2.2

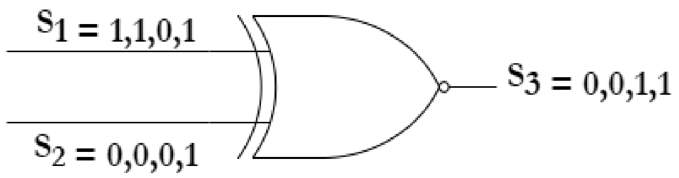


Figure 2.2 Multiplication for BPE using XNOR gate

Scaled Adder

Scaled addition in SC can be performed using a multiplexer (MUX) as illustrated in Figure 2.3, with a random bit-stream being at the selection pin representing the scale, and the two add numbers being at the input pins.
Despite its attractive simplicity, scaled addition in SC suffers from major accuracy loss, making it impractical for applications that require high number of additions.

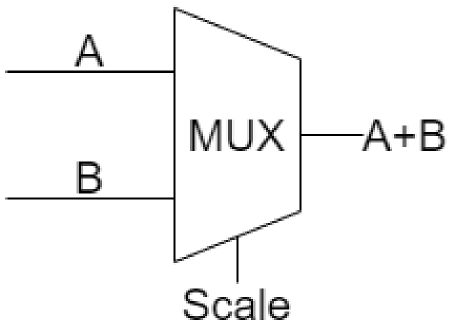


Figure 2.3 Scaled adder using MUX : addition of stream A and B

Integral SC Based Adder

Due to the nature of neural networks which require many additions during both inference and training, we had to seek a more accurate addition approach as an alternative to the faulty scaled adder.

The sum of a sequence of stochastic bit-streams can effectively be the total number of 1s, which denotes the encoding probability of the result's bit-stream.
Suppose that **X** is a sequence of real numbers in range $[-1, 1]$ and **S** is a sequence of stochastic bit-streams representing the sequence **X**, the encoding probability of the

stochastic bit-stream representing $\sum_{i=1}^{n} \boldsymbol{X}_i$ can be calculated by $\sum_{i=1}^{n}\sum_{j=1}^{l} \boldsymbol{S}_{ij}$, where $\boldsymbol{n}$ and $\boldsymbol{l}$ are the number of elements in $\mathbf{X}$ and the length of each bit-stream respectively. However, there is no guarantee that the result lies in BPE range $[-1, 1]$.
In order to represent numbers outside the bipolar format's range, an implicit scaling factor $\boldsymbol{m}$ is used in a way that $\boldsymbol{m}$ stochastic bit-streams are concatenated to form a single bit-stream with of length $\boldsymbol{m} \times \boldsymbol{l}$, which allows the bit-stream to represent real numbers in range $[-\boldsymbol{m}, \boldsymbol{m}]$. For example, the bit-streams representing $x_1 = 0.75$ and $x_2 = 0.5$, can be $s_1 = \{1,1,1,0,1,1,1,1\}$ and $s_2 = \{1,1,1,0,1,1,0,1\}$ respectively, where $\boldsymbol{l} = 8$. With $\boldsymbol{m} = 2$, the bit-stream representing $x_1 + x_2 = 1.25$ is $s_1 + s_2 = \{1,1,1,1,1,1,1,1,0,0,1,1,1,0,1,1\}$, note that the first 8 bits denote 1, and the second 8 bits denote 0.25.
Taking what's said above into consideration, the sum of a sequence of $\mathbf{n}$ bit-streams is given by :

$$\frac{2 \times \sum_{i=1}^{n}\sum_{j=1}^{l} \boldsymbol{S}_{ij} - l \times n}{l} \tag{2.3}$$

Where l is the length of a single bit-stream in the sequence. This method provides a flawless sum result, and is derived from the decoding formula of BPE discussed in 2.2, and was inspired by [8].

2.3 Error Injection

One of the ways we chose to investigate the comparative error robustness of different approaches under analysis is to use error injections, where bit flip errors are introduced into ("injected" into) the network parameters or inputs. More details are given in the next section and in section 6.3.

2.3.1 Error Modes

There are three main modes of injecting errors we made use of:

- Mode 0: Stuck-at 1 ($0 \rightarrow 1$)
- Mode 1: Stuck-at 0 ($1 \rightarrow 0$)
- Mode 2(Hybrid Mode): $0 \rightarrow 1$ &$1 \ \rightarrow 0$

Due to time and resource constraints, our focus was majorly on testing the hybrid mode, as it's more 'natural' and intuitive, and therefore have decided to include only Mode 2 results in Chapter 7.

3
Feasibility

In this section, we describe the feasibility study carried out during the development of the project.

3.1 Technical feasibility

Technical feasibility study is the complete study of the project in terms of input, processes, output, fields, programs and procedures.

3.1.1 *Software feasibility*

The operating systems, and development environments are explained in 3.1.

Table 3.1 Software options

	Option 1	Option 2
OS	Windows 10	Linux
Language	Python	Python
IDE	Pycharm	Visual Studio Code

This project is developed using Python due to its wide variety of "simple to use" and high performance libraries capable of performing complex mathematical operations. As for the operating systems, Windows and Linux are two operating systems we are familiar with.

3.1.2 *Hardware feasibility*

Since training a DNN is a compute-intensive task, a machine with a dedicated GPU is recommended for a better performance. However, once the trained model is acquired, the system is expected to run smoothly on any mid-range computer. 3.2.

Table 3.2 Hardware requirements for a healthy system operation

	Option 1	Option 2	Option 3	Option 4
Processor	$\geq$ up to 2.80 GHz	$\geq$ up to 3.60 GHz	i7-8750h	i7-4510u
RAM	8 GB	8 GB	16 GB	12 GB
Storage	30 GB	30 GB	30 GB	30 GB
GPU	Dedicated, VRAM $\geq$ 2GB	Integrated	GTX 1050	GT 840m

3.2 Communication feasibility

Git along with GitHub is used for project collaboration, version control and source control. Also, online virtual meetings are conducted using Zoom software.

3.3 Legal feasibility

The project complies with existing laws and regulations, any patents and so on, and does not violate any protected rights.

3.4 Economic feasibility

Expenditures made on behalf of the project are shown in Table 3.3.

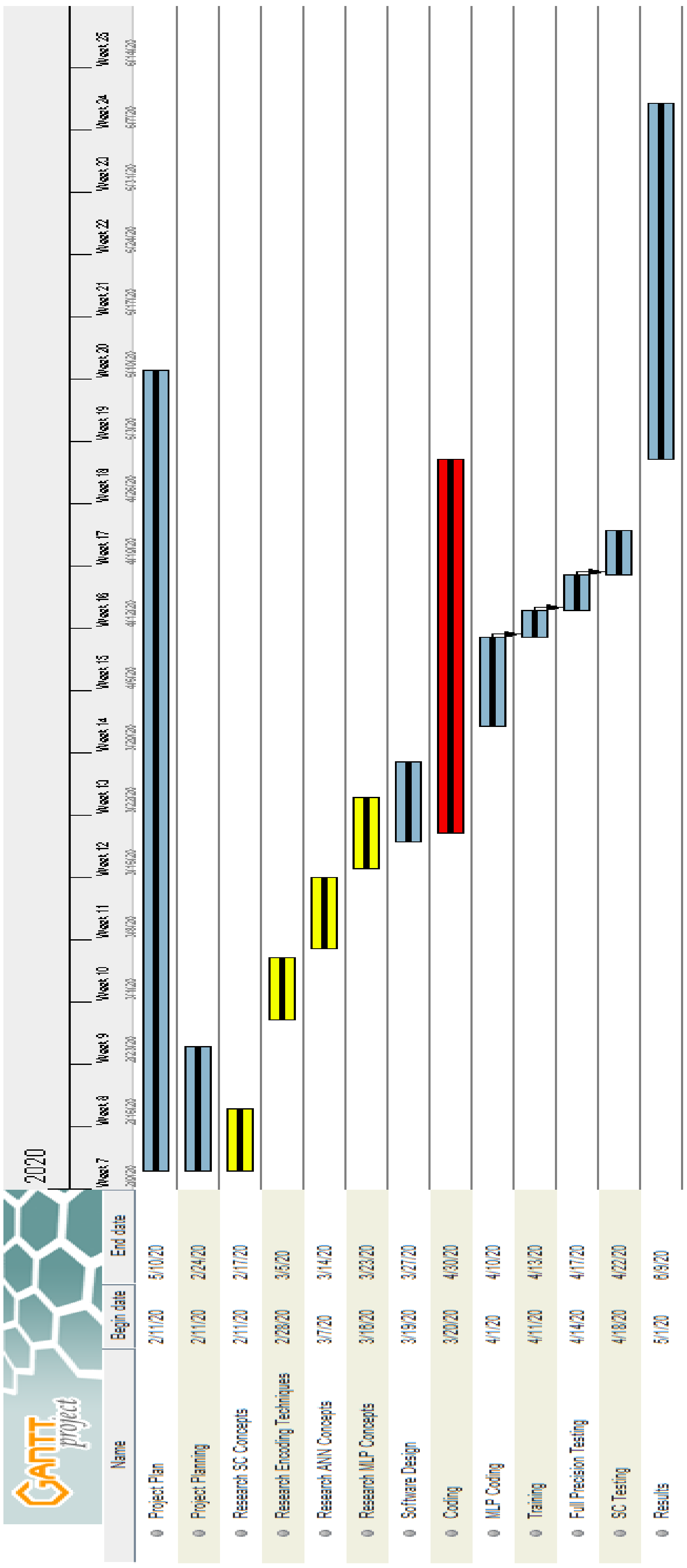
Table 3.3 Expenditures table

	amount	Price per unit	Total price	Brand
Computer 1	1	5000 TL	5000 TL	Casper Excalibur G650
Computer 2	1	2700 TL	2700 TL	Lenovo Z50-70
Software Developer Expense	2	120 TL (daily)	28800 TL	-
Total	-	-	36500 TL	-

Note: Since some development software and platforms are either open-source or provided through student licenses, there has been no expenditure on behalf of software.

3.5 Time feasibility

The design and implementation process of the project is as specified in the Gantt chart in Figure 3.1.



Name	Begin date	End date
Project Plan	2/11/20	5/10/20
Project Planning	2/11/20	2/24/20
Research SC Concepts	2/11/20	2/17/20
Research Encoding Techniques	2/28/20	3/5/20
Research ANN Concepts	3/7/20	3/14/20
Research MLP Concepts	3/16/20	3/23/20
Software Design	3/19/20	3/27/20
Coding	3/20/20	4/30/20
MLP Coding	4/1/20	4/10/20
Training	4/11/20	4/13/20
Full Precision Testing	4/14/20	4/17/20
SC Testing	4/18/20	4/22/20
Results	5/1/20	6/9/20

Figure 3.1 Gantt chart

4 System Analysis

Systems Analysis is the process of collecting and interpreting facts, identifying the problems, and decomposition of a system into its components. We use this chapter to present the requirements for the different components of our system and pipeline, to make it easier to understand and justify our design choices and implementation tradeoffs in the subsequent chapters.

4.1 Dataset Requirements

The dataset(s) we wanted to use was to possess mainly these features:

- Simple enough to achieve good results with small models, as the hyper-parameter search, training, and testing requires abundant computational, time and monetary resources.
- At the same time, it should be complex enough to make it a challenge for our model to achieve good results on it.
- Work related to the dataset should be present in the literature to enable us to judge our models with respect to the benchmarks.
- The dataset should either be easy to modify, or some variations of it should be available for us to test the error robustness of our models.

4.2 Neural Network Requirements

We were to perform a large number of tests on a variety of datasets and models with a huge set of possible settings. Therefore, the requirements for the Neural Network were related not only to the computational efficiency of each network component, but also to ease of use and powerful customization options. The user should be able to define a simple network with minimal code required, with the default settings good enough

for the network to reach high accuracy, and at the same time, the user should have the option to customize each part of the network and its training cycle, to precisely test the performance difference brought upon by the changes to any component / parameters / hyper-parameter.
The requirement for computational efficiency and GPU interoperability was a given, considering the time complexity for training a neural network.

4.2.1 Additional notes for BNN

As explained in section 2.1.3, a BNN requires two copies of weights, one of which is generated by binarizing the other at the start of each forward pass. This requires additional memory and computational resources. This additional strain the system resources should be kept in mind during the design of the system, to ensure minimal resource utilization, so that we can have more freedom regarding the architectural choices for the BNN testing part.

4.3 SC Based Neural Network Inference

The primary objective of this project is to examine the performance of stochastic computing based neural networks during inference phase. Therefore, we had to :

- Investigate and identify the arithmetic units required to implement a feedforward of a deep neural network.
- Implement the necessary computational units in stochastic computing, evaluate their performance experimentally and define their weak points.
- Investigate architectures and hyper-parameters in which SC based NN performs the best.
- Given that an increase in the precision of stochastic bit streams leads to an exponential increase in both space and time complexity, the effect of precision (bit-stream length) on the performance of both SCBNN and SCFPNN should also be investigated.

4.4 Error Injection Requirements

As our error injection procedure is to rely on probabilistic bit flips of the network parameters, our system design had to incorporate methods for selecting bits randomly among the parameters, and flip them according to the error injection mode, with minimal changes to the functional API, all the while keeping the whole inference run time fast enough to perform a large number of tests effectively.

5
System Design

System design is the process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements.

5.1 Datasets

Keeping the dataset requirements from section 4.1 in mind, we checked a number of datasets, and decided to work with MNIST, a set of its "corrupt" variants MNIST-C, and Fashion-MNIST.

5.1.1 MNIST

MNIST [9] is somewhat of a canonical dataset in literature, being simple enough to not require too sophisticated models, yet challenging enough to try out the effects of a variety of approaches and models in comparison to each other.

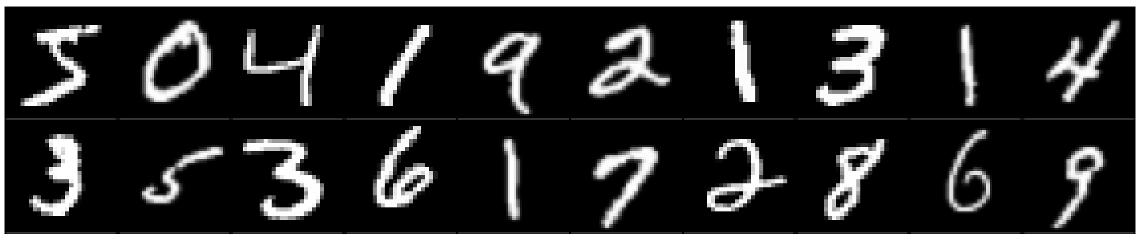


Figure 5.1 MNIST Image Grid

5.1.2 MNIST-C

MNIST-C [10] consists of 31 different variations / corruptions of MNIST (15 in the condensed version), 27 of which are used by us to test the error robustness of different models (FPNN, SCFPNN, BNN, SCBNN) trained on MNIST.

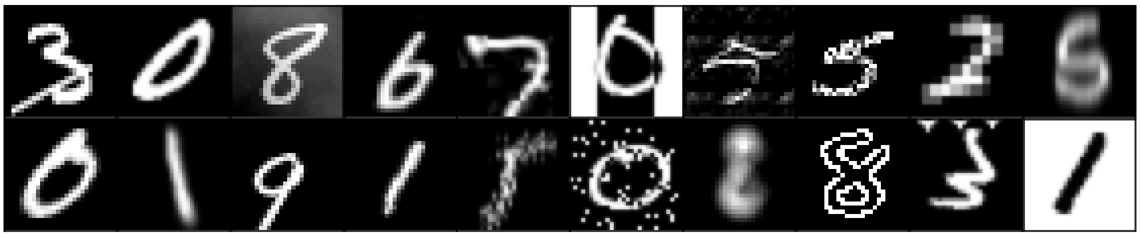


Figure 5.2 Assortment of MNIST-C Images

5.1.3 Fashion MNIST

Fashion MNIST [11] is a dataset with 70,000 28x28 grayscale images, intended to serve as a replacement for MNIST, offering more of a challenge in terms of getting good results, and more closely reflecting the state of the field of Computer Vision.

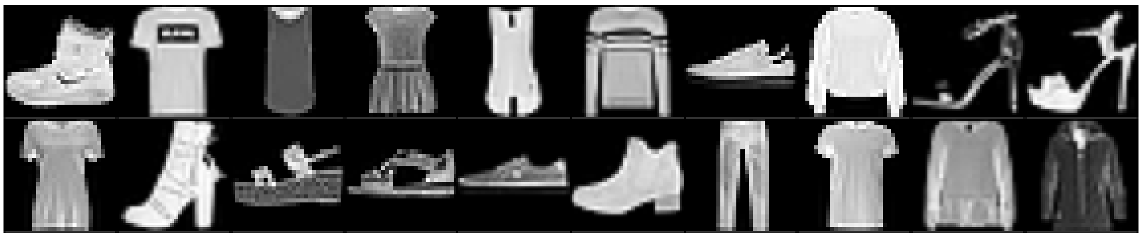


Figure 5.3 Fashion MNIST Images

5.2 Neural Network Design

The Neural Network is designed as a class, containing methods for training the model, calculating its accuracy on given data, saving and loading a model (for inference or finetuning) and some helper functions for changing network parameters and turning data into a batch generator.

In a simple training pipeline, the user instantiates a network object, compiles it with the required activation and loss functions, trains it on the dataset, calculates the accuracy and then saves the model for furthur training or run-time inference. As we provide the ability to train using the GPU as well, all the components and methods are written in a device agnostic way, with the same interface for the user regardless of the device ('CPU' or 'GPU') used. Along with that, all the methods relating to numerical computation are vectorized using C libraries with python wrappers to gain a major boost in speed.

Details for each of the major components are given below:

5.2.1 Network Object

The network object is internally composed of a list of neuron layers, created using a list of layer sizes in the constructor. It can also be instantiated using solely a file containing a previously saved model. The constructor also takes care of checking the type of network ('BNN' or 'DNN'), and the device on which the network is to be trained. The network is then *compiled* to actually instantiate all parameters and make some sanitary checks regarding the activation functions and the learning rate object, to keep things consistent.

A main method to train the model is present in the design, with an option to add a validation dataset to test different combinations of hyper-parameters and one to change the batch size. The only required parameters for this method should be the training data and the number of epochs to train (an epoch is one training pass over the whole dataset).

Lastly, there are methods to check the accuracy of the model on a given piece of data and to save the model for furthur use.

5.2.2 Layer modularity

As mentioned in the previous section, the network object is composed of multiple layer objects. This design choice mirrors that of major deep learning frameworks like Pytorch, Tensorflow with Keras and Theano with Lasagne. This allows us to keep the layer-level details like parameter instantiation, modification, and gradient updates separate from the training method in the Network class, enabling us to mix and match the layers as required, and to fully utilize the power of OOP (or as much of it as is available in Python).

This makes it easier to maintain, test and update the code, following the DRY and encapsulation principles. Another benefit of this approach is the ability to add hooks into the training method, allowing us to modify the inputs, preactivations and network parameters using user-defined functions, increasing the power of user level customization, and making it easier for us to perform error injection tests.

5.2.3 Data Loaders

Data is loaded using, you guessed it, a data loader. A data loader in the context of our framework is intended to be a simple method that takes in the name of the dataset (using an enum of available datasets), loads it from disk, and preprocesses it to ensure a consistent data format, with options to quantize and normalize the data, move it to CPU or GPU and one-hot encode the labels if required.

5.2.4 Model File Format

The model file should have all the information required to create the model from scratch. Therefore, the following information is saved in the model file:

- A list of layer sizes
- A boolean indicating whether bias is used in the model
- A boolean indicating whether batchnorm is used in the model
- A list of weights for each layer
- A list of biases for each layer (if it's used)
- Four lists containing the parameters for batchnorm (beta, gamma, mu and sigma), if it is used

5.3 Design of Stochastic Elements

5.3.1 Quantization

As explained in 2.2, SC is incapable of representing fully precise numbers. Thus, before a decimal number is converted to stochastic bit-stream, it has to be quantized according to ***p***, the precision/size of the stream.

$$quantize(x,p) = \begin{cases} 1 & if\, x \geq 1 \\ -1 & if\, x \leq -1 \\ \frac{2}{p} \cdot round(\frac{x \cdot p}{2}) & otherwise \end{cases} \tag{5.1}$$

5.3.2 Binary to Stochastic Conversion

Algorithm 1: Stochastic Number Generator

```
Input  : Float $x \in [-1, 1]$
         Integer $L$ : Bit-stream length
Output: $X \in I\!R^L$ : Generated stochastic bit-stream
$q \leftarrow quantize(x, L)$
$p \leftarrow \frac{x+1}{2}$ // Bipolar encoding probability
for $i \leftarrow 0$ to $L-1$ do
    $X[i] \leftarrow= 0$
$n \leftarrow round(p \times L)$ // Number of ones in the stream
for $i \leftarrow 0$ to $n-1$ do
    $X[i] \leftarrow= 1$
$X \leftarrow permute(X)$
```

5.4 Design of Stochastic Neural Network

The SC Neural Network is designed as class that concerns only with the inference of a pre-trained model. Therefore, although it doesn't have the same internal layer design as in 5.2, it is indirectly dependent on it. With the dependency being the data and model loaders.

The class contains methods for calculating the accuracy of a given model on given data, with the help of other functions which take care of :

- Extracting the parameters of a pre-trained model and converting the weights and biases to stochastic bit-streams.
- Dividing the input data into multiple batches, and distributing each batch on a different vCPU worker to gain a faster runtime performance.
- Preprocessing the input data by converting it to stochastic bit-streams and injecting error into it if required.
- Performing a forward propagation pass of the preprocessed input data.

6 Implementation

Poetry is being used for managing project dependencies, virtual environments, and the build system. Numpy provides support for vectorized operations, and Cupy augments it with GPU acceleration, with minimal changes in the API. HDF5 is utilized for saving the trained models in a consistent and effective manner.
We also make use of Tensorflow along with Keras to more effectively train the bigger BNN models, utilizing the features not yet present in our framework (like optimizers, DAGs and momentum) to make the comparisons with other models more fair.

6.1 Network class

Following the requirements and design choices from section 4.2 and 5.2, a network class is implemented with the following main methods:

6.1.1 Model Creation

The model is instantiated using either a list of layer sizes (using the main constructor), or using the path to a saved model (using a static method, as we cannot have multiple constructors for an object in Python). These methods also take some optional inputs to determine the device ('CPU' or 'GPU') on which to store the parameters, and whether to use bias and batchnorm. If we want to use batchnorm, bias is not used automatically, as the batchnorm step removes the effect of bias, and therefore using bias with batchnorm would only lead to redundant computations and memory usage.
After the network object instantiation, the compile method needs to be run before we can train/test the network. The compile method takes the hidden and output activation functions, loss function and the learning rate object (which can either be a float, or a LRScheduler object if we want to use decaying learning rate during training).
Inside the compile method, we build the parameters for each layer, check whether the activation functions are consistent with the network type ('BNN' / 'FPNN'), and instantiate a LRScheduler object (if the learning rate argument is a float) to maintain

a consistent API during training.
The compile method was created to serve three main purposes:

1. Following the DRY principle by extracting the common functionality from both the constructors to a separate method
2. Avoid putting too many (redundant) arguments in the constructor objects
3. Maintaining separation of concerns by keeping only the required things in the network construction methods

6.1.2 DNN Training Pipeline

The basic concept of training a neural network is explained in section 2.1.2. Training in our framework is done using the *train* method of the *Network* class. The function takes in the training data and the number of epochs as the arguments, along with the optional parameters batchsize, validation data and a boolean to indicate if the model should load the best performing weights at the end of the training cycle, and one to set whether to shuffle the training data at the start of each epoch.
At the start, the variables are declared and the shape of the data matrices is checked.

Then begins the training loop, the pseudocode for which is given below:

Algorithm 2: Neural Network Training (DNN/BNN)

```
1  for $i \leftarrow 1$ to $maxEpochs$ do
2      for $batch$ in $getBatches(trainX, trainY)$ do
3          if $modelType == 'BNN'$ then
4              $Wb_l \leftarrow binarize(W_l)$
           // FORWARD PASS
5          for $l \leftarrow 1$ to $totalLayers$ do
6              $z \leftarrow X \cdot W_L + b_L$ // $Wb_L$ instead of $W_L$ if BNN
7              $y \leftarrow activation_L(z)$
               // where activation is the ‘sign’ function for ‘BNN’
               // y becomes the input X for the next layer
           // BACKWARD PASS (Gradient Calculation)
8          $cost \leftarrow costFunction(trueLabel, predictedLabel)$ // predicted
             label is y from the final layer
9          for $L \leftarrow totalLayers$ to 1 do
10             $\delta_L$ , $\frac{\partial Cost}{\partial W_L} \leftarrow Backprop(X_L, W_L, y_L, \delta_{L+1})$
           // WEIGHT UPDATE
11         for $l \leftarrow totalLayers$ to 1 do
12             $W_L' \leftarrow W_L - \eta \cdot \frac{\partial Cost}{\partial W_L}$
           // IMPORTANT: In case of ‘BNN’, the gradients are calulcated using
            $Wb_L$ but applied to $W_L$
```

6.2 SCNetwork class

Following the requirements and design choices from section 4.3 and 5.4, a network class is implemented with the following main methods:

6.2.1 Model Creation

The model can be instantiated using a path to a saved model previously trained by a conventional Network object. The constructor also takes other inputs, which include precision, two boolean parameter indicating whether to use bias and batchnorm, and an optional boolean parameter indicating whether the network is binarized (BNN) or not. The constructor also takes care of converting the weights and biases to stochastic bit streams, represented by boolean *NumPy* arrays. Before testing the network, a compilation method needs to be run. The compile method takes the activation functions and the implicit scaling coefficients for both the hidden and output layers.

6.2.2 SCNetwork Testing Pipeline

To test the accuracy of the model on given input data, the *evaluate* method of the *SCNetwork* class is used. The function takes in the input data along with their labels, in addition to optional parameters including :

- ***batch_size***: As high precision bit-streams require a large space in memory, converting all instances of input data to bit-streams at the same time could cause the memory to overflow. Therefore, the input data can be passed to the network in batches, instead of being passed all at the same time. The *batch_size* parameter indicates the number of instances which each batch contains.
- ***parallel***: This is a boolean parallel that is used to indicate whether the testing is to be conducted in parallel or not. In order to accelerate the runtime performance of the network, an optimal number of vCPU workers can operate on different batches of data at the same, with each batch being of size *batch_size*.
- ***pLimit***: In python, **multithreading** is not effective for CPU bound tasks, due to the global interpreter lock. Therefore, the parallel computing approach used in this project is **multiprocessing**. Multiprocessing in this context means that each worker represents a process by its own, so the workers do not share the same memory, which in turn means that a large number of workers could also lead to memory problems because a lot of data is repeated redundantly. This parameter takes care of it by limiting the number of processes/workers that can be initiated at the same time.

Once called, *evaluate* method:

1. Divides the input data to multiple batches of size *batch_size*.
2. Initiates a pool of workers, with the size of the pool being the minimum of: number of batches, vCPU count on the device and *pLimit*.
3. Gets the accuracy, basically pre-processing each batch (converting it to stochastic bit-streams) and performing a forward propagation pass of each batch and comparing the results to the labels.

Furthermore, the forward pass of *SCNetwork* is similar to the one presented in *Network* class, with the major difference being that it performs the inner dot product required for the weighted sum by utilizing arithmetic units discussed in 2.2. Also, since we did not implement custom activation functions that can deal with stochastic bit-streams, the pre-activations are decoded prior to being passed to the activation function, and the result is re-encoded back again.

6.3 Error Injection

Keeping the requirements from 4.4 in mind, errors are injected using the following ways:

6.3.1 NN Layer Hooks

For FPNNs, the errors are injected using the layer hooks mentioned in section 5.2.2. The elements to be flipped are selected randomly using the 'choice' method from Numpy. Then for FPNNs, these elements, which are of dtype float32, are unpacked to bit arrays and then bits are flipped (randomly again) according to the error mode. For BNNs, we only need to change the sign of the selected elements.

6.3.2 SCNetwork Error Injection

A special version of *evaluate* function takes care of testing the model on given data with error injection. The function takes the same input parameters of *evaluate* function explained in 6.2, in addition to :

- ***weight_rates*** : a list which contains the rate of bits to be flipped at each layer of weights.
- ***image_rate*** : the rate of bits to be flipped in each input image.
- ***activation_rate*** : a list which contains the rate of bits to be flipped at each layer of activations.

Moreover, as the bit-streams are represented by NumPy arrays, the bitflip operation is performed in a way similar to that in 6.3.1, without the need of unpacking and repacking.

7
Experimental Test Results

7.1 Methodology

The tests were done mainly in three different phases. In each phase, the DNN (FPNN) and BNN models were trained, with the best possible combinations of hyperparameters and networks architectures chosen using validation accuracy and loss, and then tested on the datasets with and without SC. The details for each phase are given below:

1. In the first phase, we trained the DNN and BNN models on MNIST [9], and then used MNIST-C (MNIST-Corrupted [10]) datasets in the test phase to see how well the errors are handled by BNNs vs DNNs, and SC-NNs vs deterministic NNs. The architecture details are present in section 7.2.1, whereas the results for this phase are presented in section 7.3.1.

2. In the second phase, separate models were trained and tested on MNIST-C datasets, and the different approaches were compared like in phase one. Phase one and two collectively make up the "Image Corruption" tests. More details in section 7.2.2 and 7.3.2.

3. In the third phase, bitflip errors were introduced into the images and the weights (separately), and the comparative drops in the performance are measured. You can find more in section 7.4.1 and 7.4.2.

7.2 Architecture and Hyperparameter Details

All models presented here were trained for 1000 epochs. For each epoch, the validation accuracy was compared against the best validation accuracy yet, and if it was higher, the weights were saved. This allowed us to retain the parameters with the best validation (not test) accuracy over 1000 epochs [1]. Those parameters were then

[1]This allows us to select the model which generalises the best, and doesn't overfit

loaded into the model and used to perform the tests. For the SC tests, a stream size of 1024 was used to perform SC-FPNN tests, and 32 for the SC-BNN tests.

7.2.1 Phase One

Details for the MNIST and Fashion MNIST models are presented here:

Parameter	DNN	BNN
Hidden Layers	{200, 100}	{1024, 1024, 512}
Batch Size	100	100
Hidden Activation	Sigmoid	Sign
Output Activation	Softmax	Identity
Loss Function	CrossEntropy	CrossEntropy
Learning Rate Range	$7e^{-2}$ to $7e^{-7}$	$3e^{-3}$ to $3e^{-7}$
BatchNorm Used	False	True

Table 7.1 MNIST Architecture

Parameter	DNN	BNN
Hidden Layers	{200, 100}	{1024, 1024}
Batch Size	100	32
Hidden Activation	Sigmoid	Sign
Output Activation	Softmax	Softmax
Loss Function	CrossEntropy	Squared Hinge
Learning Rate Range	$3e^{-2}$ to $3e^{-7}$	$1e^{-3}$ to $1e^{-7}$
BatchNorm Used	False	True

Table 7.2 Fashion MNIST Architecture

7.2.2 Phase Two

The architecture and hyperparameter choices for phase two models are mostly the same as that for phase one MNIST (in table 7.1).
The only two differences are present in DNN training for two datasets, and are given below:

Dataset Name	Learning Rate Range
MNIST-C Fog	$7e^{-2}$ to $7e^{-5}$
MNIST-C Line	$7e^{-2}$ to $7e^{-6}$

7.3 Results

The performance of different models presented in section 7.2 is presented below, along with the important notes and observations:

7.3.1 Phase One results

Phase one results include the accuracies for MNIST and Fashion MNIST Models, along with the test accuracy of MNIST model on a variety of MNIST-C datasets.

Dataset Name	FPNN(%)	SC-FPNN(%)	BNN(%)	SC-BNN(%)
MNIST	98.02	97.70	98.58	**98.63**
Fashion MNIST	88.45	86.43	89.33	**89.47**
Brightness	63.50	59.97	29.29	26.58
Canny Edges	58.92	**60.00**	68.66	**68.86**
Contrast	33.11	25.88	13.79	13.62
Defocus Blur	78.54	**81.25**	76.14	**77.16**
Dotted Line	95.12	94.23	94.76	**94.77**
Fog	46.28	41.23	17.45	15.78
Elastic Transform	69.90	69.28	75.31	**75.36**
Frost	59.63	57.00	28.08	25.76
Glass Blur	90.78	90.72	92.33	**92.52**
Impulse Noise	91.75	90.58	69.83	**69.95**
Gaussian Blur	91.26	**91.67**	88.23	**88.40**
Gaussian Noise	90.20	87.79	49.17	48.54
Motion Blur	81.07	**81.53**	83.30	**83.74**
Shot Noise	96.41	95.72	97.92	97.84
Inverse	0.10	0.07	1.48	1.29
JPEG Compression	97.78	97.36	97.98	**98.11**
Spatter	95.10	94.10	94.78	94.72
Stripe	21.44	20.93	19.59	19.08
Line	82.28	**81.70**	81.35	**81.76**
Pessimal Noise	91.65	89.55	86.19	86.06

Zigzag	77.28	76.15	73.81	**74.00**
Pixelate	95.42	94.96	96.69	**96.76**
Quantize	97.45	97.09	98.45	98.43
Saturate	95.78	94.24	51.84	52.76
Snow	80.94	77.60	35.91	34.78
Speckle Noise	96.74	96.04	97.92	**97.94**
Zoom Blur	93.06	91.64	94.64	**94.86**

Table 7.3 Test Accuracies for Phase One Models

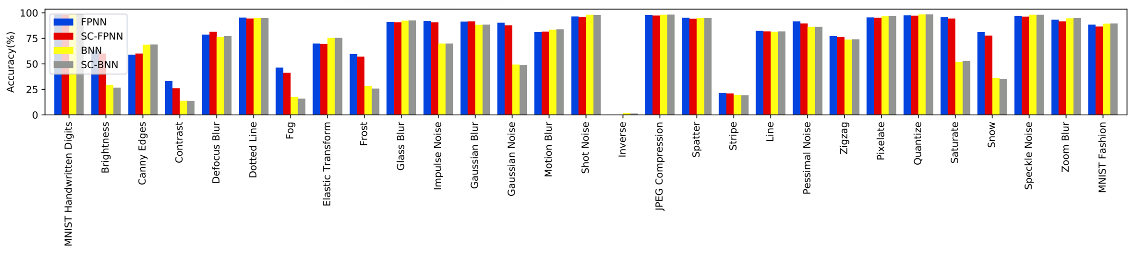


Figure 7.1 Test Accuracies for Phase One Models

7.3.2 Phase Two Results

Here, the models are tested on the datasets on which they are trained.

Dataset Name	FPNN(%)	SC-FPNN(%)	BNN(%)	SC-BNN(%)
Brightness	97.86	94.43	97.21	**97.22**
Canny Edges	94.99	93.91	96.27	**96.29**
Contrast	94.55	87.15	95.56	90.18
Defocus Blur	97.78	95.86	97.99	**98.02**
Dotted Line	97.15	96.96	98.11	**98.16**
Fog	96.31	87.72	94.15	94.13
Elastic Transform	94.37	90.67	95.24	**95.30**
Frost	96.45	90.13	96.18	96.14
Glass Blur	96.70	94.68	97.29	**97.30**
Impulse Noise	94.51	92.95	96.30	**96.30**
Gaussian Blur	97.69	96.63	98.12	98.03
Gaussian Noise	94.48	93.46	96.20	96.19
Motion Blur	95.58	**96.67**	98.17	**98.17**
Shot Noise	96.84	96.29	97.62	**97.65**

Inverse	97.87	96.95	97.93	**97.93**
JPEG Compression	97.79	97.19	98.59	98.53
Spatter	96.72	95.64	97.28	**97.40**
Stripe	98.11	97.48	98.22	**98.24**
Line	96.26	95.93	96.79	**96.84**
Pessimal Noise	97.95	96.98	98.35	98.25
Zigzag	96.97	94.95	97.77	**97.79**
Pixelate	98.02	97.42	98.29	**98.32**
Quantize	97.52	97.11	98.26	**98.26**
Saturate	97.46	96.50	97.92	**97.95**
Snow	96.60	93.77	97.07	**97.16**
Speckle Noise	97.29	96.67	97.97	97.93
Zoom Blur	98.10	97.74	98.48	98.44

Table 7.4 Test Accuracies for Phase Two Models

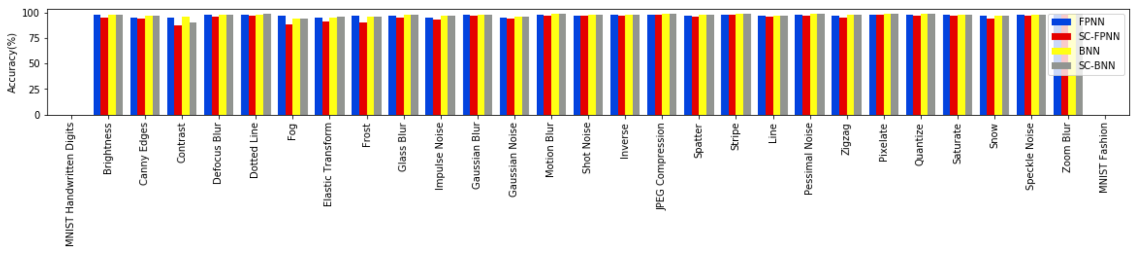


Figure 7.2 Test Accuracies for Phase Two Models

7.3.3 Extra information

The information regarding the best validation accuracies for all the models, and the epoch where that accuracy was obtained, is presented above is presented here.

Dataset Name	FPNN(%)	FPNN Epoch	BNN(%)	BNN Epoch
MNIST	97.985	615	98.467	784
Fashion MNIST	89.31	872	90.583	993
Brightness	97.68	524	96.983	978
Canny Edges	95.06	822	96.658	999
Contrast	94.97	954	96.267	942
Defocus Blur	97.64	892	98.508	854
Dotted Line	97.30	590	98.292	997

Fog	96.09	985	93.683	957
Elastic Transform	94.39	371	95.633	942
Frost	96.39	947	96.967	938
Glass Blur	96.80	978	97.75	660
Impulse Noise	94.37	461	96.408	884
Gaussian Blur	97.82	830	98.492	867
Gaussian Noise	94.36	461	95.942	847
Motion Blur	97.52	643	98.100	930
Shot Noise	96.83	872	97.733	911
Inverse	97.933	505	97.958	901
JPEG Compression	97.933	429	98.542	857
Spatter	96.467	626	97.242	957
Stripe	98.000	749	98.250	962
Line	96.125	283	97.100	990
Pessimal Noise	97.817	963	98.400	909
Zigzag	96.900	381	97.785	991
Pixelate	97.958	796	98.733	958
Quantize	97.458	824	98.467	771
Saturate	97.483	536	97.925	920
Snow	96.675	668	96.967	914
Speckle Noise	97.117	877	98.083	975
Zoom Blur	97.992	978	98.692	870

Table 7.5 Validation Accuracies and Best Epochs - All Models

7.4 Error Injection

The next two sections deal with the test results achieved by introducing errors into the network inputs or parameters. Only the results for mode 2 (hybrid) error injection are presented for reasons explained in section 2.3. A number of tests were performed by injecting the errors, and the results were then averaged to get a better idea of how the models deal with them with and without SC, as each error injections are probabilistic. The model used in both the experiments is a BNN model trained on MNIST images, the details for which are given below:

Parameter	Value
Hidden Layers	{1024, 1024}
Batch Size	100
Hidden Activation	Sign
Output Activation	Identity
Loss Function	Squared Hinge
Learning Rate Range	$3e^{-3}$ to $3e^{-7}$
BatchNorm Used	True
Best Epoch	968
Test Accuracy	98.27
Validation Accuracy	98.433

Table 7.6 Error Injection Model Architecture and Hyperparameters

7.4.1 Error Injection - Images

In image error injections, we flip the bits in the MNIST test dataset [9] and perform inference on it using the BNN model.

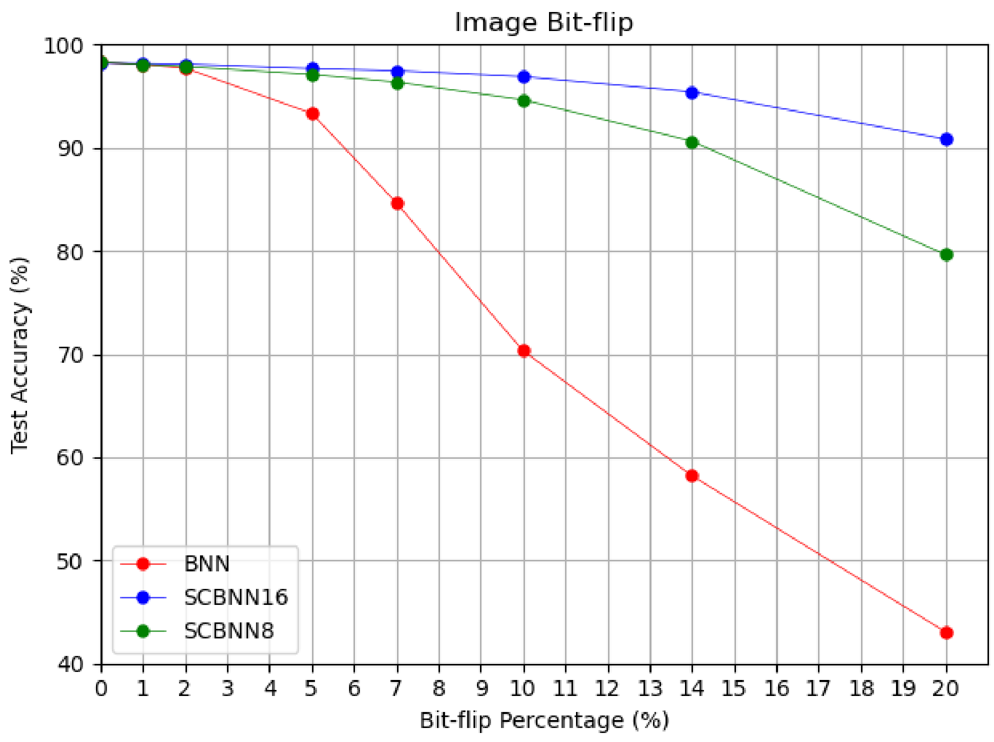


Figure 7.3 Performance Comparison - Image Error Injection

7.4.2 Error Injection - Weights

In these tests, we randomly choose the bits from the network weights and flip their sign (in the BNN model) or flip the bits (in SC model) according to the given probability.

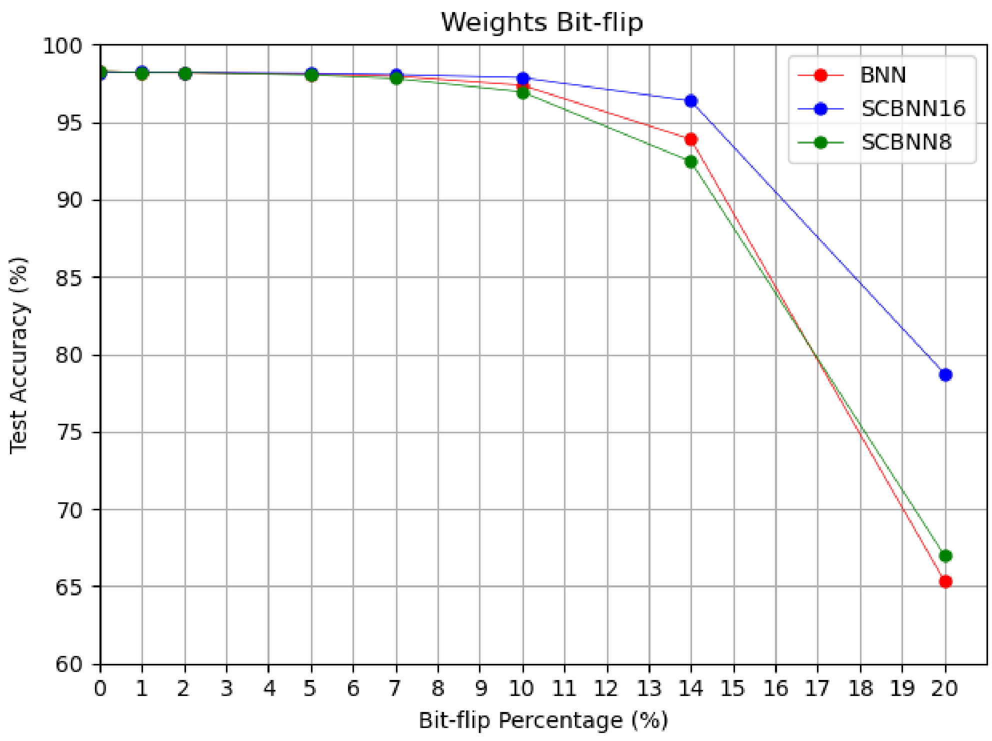


Figure 7.4 Performance Comparison - Weights Error Injection

8 Performance Analysis

In this chapter, we present the training and performance details of different activation and loss functions we checked during the course of our experiments. We'll be using MNIST dataset [9] for these tests.

8.1 Architecture and Hyperparameters

The table below presents the architecture and hyperparameter details used in the comparisons in this chapter (with the exception of section 8.2.2, where different output activation and loss functions are used for loss function comparison).

Parameter	Value
Hidden Layers	{200, 100}
Batch Size	100
Output Activation	Softmax
Loss Function	Cross Entropy
Learning Rate Range (FPNNs)	$1e^{-3}$ to $1e^{-7}$
Learning Rate Range (SCNNs)	$7e^{-2}$ to $7e^{-7}$
BatchNorm Used	False

Table 8.1 NN Architecture for Performance Analysis

8.2 FPNN Model - Performance Analysis

8.2.1 Activation Function Performance Comparison

This figure below shows the FPNN training curve, validation accuracy vs epoch, for different activation functions:

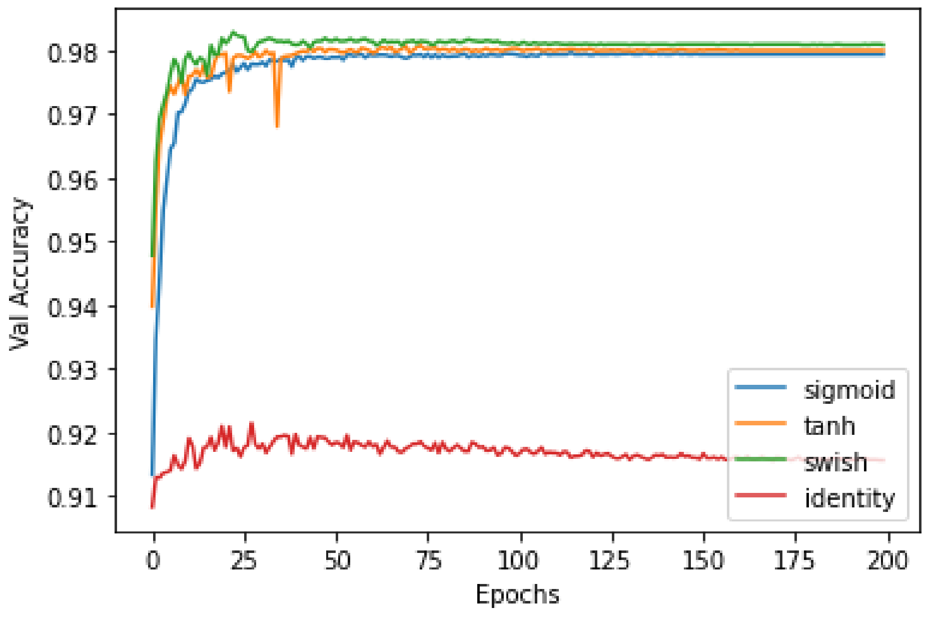


Figure 8.1 FPNN - Activation Function Comparison

8.2.2 Loss Function Performance Comparison

For testing the performance of different loss functions, we make use of the architecture in table 8.1, with swish for hidden layer activation. The comparison of different loss functions is presented below:

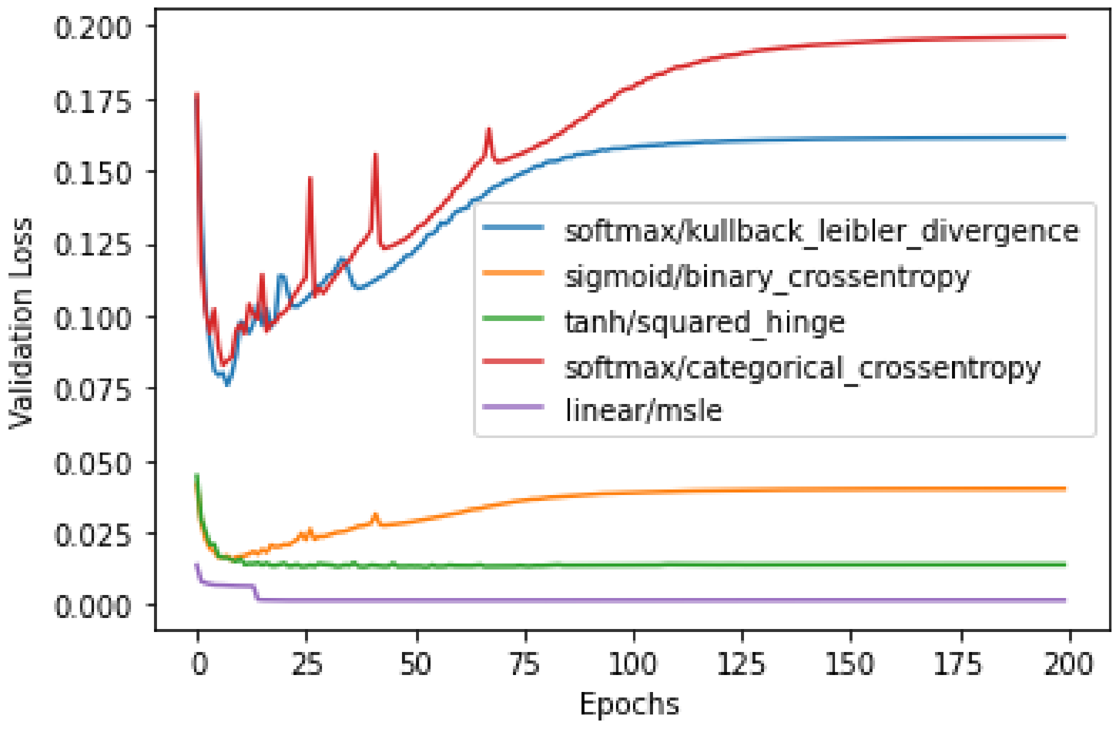


Figure 8.2 FPNN - Loss Function Comparison

8.3 SCNN Model - Performance Analysis

With the range of BPE being $[-1, 1]$, the usage of activation functions with an unlimited output range is not possible in SCNNs. Therefore our choice of activation functions was limited to $Sign$ -which is used in SCBNN-, $Sigmoid$, and $Tanh$. However, we observed in our experiments that $Tanh$ generally lead to larger gradients than $Sigmoid$, leading it to accelerate fast and drastically magnifying the results of small input errors on the function output. Fig 8.3 demonstrates this by showing how $Sigmoid$ performs better than $Tanh$ for short bit-streams.

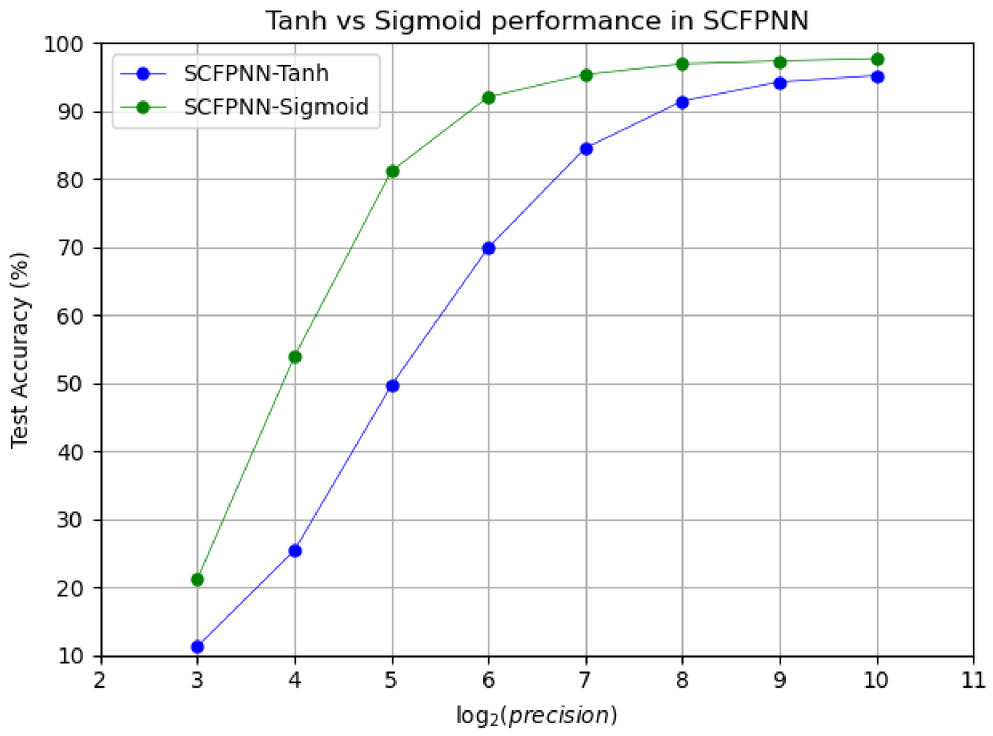


Figure 8.3 Tanh vs Sigmoid performance in SCFPNN

9 Conclusions

Running the whole suite of tests mentioned in chapter 7 lead us to draw some interesting conclusions in service of our project goals [1.2]. Some of these are concerned with the effective training and testing of the models, and others with the results of our experiments and how they relate to our initial hypotheses regarding the performance of BNNs and SC.

Some of our observations and conclusions drawn from our experiments are listed below:

9.1 BNN Training

Using only binary parameters and activations, BNNs can achieve results rivaling FPNNs, with proper choice of architecture and hyperparameters. This observation, combined with the fact that the hardware implementation of BNNs is resource effective makes them a very attractive option, especially for low power devices.

- For the training of BNNs, squared hinge loss along with no activation at the last layer performed much better than cross-entropy loss, MAE, MSE or Huber Loss.
- Learning rate decay allowed us to train for a much larger number of epochs, which enabled the trained model to conduct a thorough parameter search and achieve better performance overall.
- Generally, lower learning rates work well for BNN training, seeing that the parameter search space for BNNs is more constrained, and the gradient averaging more volatile, due to the huge decrease in the number of possible parameters (as compared to FPNNs). This effect can be partially alleviated using larger batches during the mini-batch training, but that leads to longer convergence time.

- Another thing that works well for BNN training is the use of BatchNorm, which reduces the covariance shift of the probability distibutions calculated at each layer during the training, leading to more independence between the layers, allowing the possibility to use higher learning rates, and faster convergence [1].

9.2 SCNN Inference

As multiplication in SC is probabilistic and not deterministic, the accuracy of multiplication result depends on the placement of bits in each the multiplier and multiplicand. With zero being represented as bit-stream consisting of 50% of 0s and 50% of 1s, the closer the multiplication operands are to zero, the higher is the probability of having faulty bit placement. This creates what's known as the near-zero problem, which results in a significant inaccuracy. As the images in the datasets used in our project contain a high number of black pixels which have a value of zero in grayscale, the near-zero problem was a major source of inaccuracy during the tests of SCFPNN. Mapping the input images to values in the interval $[-1, 1]$ instead of $[0, 1]$ was a perfect solution not only to overcome this problem, but at the same time reduce the stream length needed to achieve good performance [12].

Furthermore, section 8.3 shows how increasing the stream size generally leads to better performance due to the networks ability to represent more precise weights and biases, and due to the accuracy of multiplication being higher.

9.3 Error Robustness of SC

As shown in the previous two chapters, SC based Neural Networks deal with faulty data and erroneous parameters with a lower drop in accuracy than their FP counterparts, even with small stream sizes. This behaviour, we believe, stems from two reasons: the inherent probabilistic nature of SC, which helps to deal with noise in the data [7.3.1], and the lack of positional significance for bits, which, as one would expect, helps deal with parameter faults in hardware in a significant manner (as can be observed in section 7.4).

[1]It works well for DNNs too, but can have a much bigger impact on BNN training

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. DOI: 10.1038/323533a0. [Online]. Available: https://doi.org/10.1038/323533a0.

[2] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015, http://neuralnetworksanddeeplearning.com/chap4.html.

[3] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” *CoRR*, vol. abs/1602.02830, 2016. arXiv: 1602.02830 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1602.02830.

[4] A. Alaghi and J. Hayes, “Survey of stochastic computing,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, 92:1–92:19, May 2013. DOI: 10.1145/2465787.2465794.

[5] G. P. Nicola Bombieri Massimo Poncino, *Smart Systems Integration and Simulation*. Springer, 2016, ISBN: 9783319273921.

[6] B. R. Gaines, *Stochastic Computing Systems*. Springer, 1969, ISBN: 978-1-4899-5841-9.

[7] B. Brown and H. Card, “Stochastic neural computation. i. computational elements,” *Computers, IEEE Transactions on*, vol. 50, pp. 891–905, Oct. 2001. DOI: 10.1109/12.954505.

[8] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, “VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing,” *arXiv e-prints*, arXiv:1509.08972, arXiv:1509.08972, Sep. 2015. arXiv: 1509.08972 [cs.NE].

[9] Y. LeCun and C. Cortes, “MNIST handwritten digit database,” 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist.

[10] N. Mu and J. Gilmer, “MNIST-C:A Robustness Benchmark for Computer Vision,” *CoRR*, vol. abs/1906.02337, 2019. arXiv: 1906.02337 [cs.CV]. [Online]. Available: https://github.com/google-research/mnist-c.

[11] H. Xiao, K. Rasul, and R. Vollgraf. (Aug. 28, 2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. arXiv: cs.LG/1708.07747 [cs.LG], [Online]. Available: https://github.com/zalandoresearch/fashion-mnist.

[12] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, “Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks,” in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2016, pp. 1–6. DOI: 10.1145/2897937.2898011.

Curriculum Vitae

FIRST MEMBER

Name-Surname: Muhammad Arslan
Birthdate and Place of Birth: 11.11.1997, PAKISTAN
E-mail: rslnkrmt2552@gmail.com
Phone: +905050093313
Practical Training: N/A

SECOND MEMBER

Name-Surname: Mecdeddin Harrad
Birthdate and Place of Birth: 23.01.1999, ALEPPO
E-mail: majd.kharrat@gmail.com
Phone: +905511685495
Practical Training: N/A

Project System Informations

System and Software: Python, Numpy, Cupy, Numba, Tensorflow, Keras, Poetry, HDF5
Required RAM: 8GB
Required Disk: 12GB