A Benchmarking Tool for Evaluation of Approximate Arithmetic Circuits in Convolutional Neural Networks

Master's Thesis

Albert Berg Hansen & Simon Dahl Jepsen

Electronic Systems, ES10, 05/2024

Master's Thesis
Electronic Systems
Fredrik Bajers Vej 7A
9220 Aalborg Øst
http://www.tech.aau.dk

Title:

A Benchmarking Tool for Evaluation of Approximate Arithmetic Circuits in Convolutional Neural Networks

Project:

Master's Thesis

Project period:

Feb 2024 to May 2024

Group:

ES 1023

Participants:

Albert Berg Hansen
Simon Dahl Jepsen

Supervisors:

Jan Østergaard

Pages:	219
Appendices:	5
Date of Upload:	31-05-2024

Abstract/Summary:

The contents of this thesis describes the development of a *benchmarking tool* capable of taking an approximate, combinatorial arithmetic circuit and analysing the **power consumption**, **latency**, **error characteristics**, and implications of implementation on a given CNN. The purpose of the tool is to bridge the gap between approximate computing and machine learning, leveraging the reduction in power consumption and latency, in an attempt to make machine learning more sustainable. The *benchmarking tool* is structured in three steps. In **step I**, the arithmetic circuit is synthesised and the netlist is used to count the logic gates and their type, and by extension estimate the total amount of transistors required for the circuit. The netlist is also used to find the **latency** of the circuit by converting the netlist to a directed acyclic graph and searching for the path that results in the longest propagation delay. In **step II** the circuit is applied to a small-scale custom CNN implemented in C++. The error characteristics are generalised in easily scalable custom CNN layers, and the C++ model is utilised to evaluate the ''appropriateness'' of the generalisation. Lastly, in **step III** the custom layers are used in a full-scale CNN, whereby the implications on the accuracy can be weighed against the drop in **power consumption** and **latency**. The custom layers were applied in a full-scale example CNN, where the potential of approximate hardware was highlighted: The accuracy reduced form ~ 70 % to ~ 60 %, however, with an optimistic estimate the **power consumption** dropped to 10.3 % and the **latency** dropped to below half.

Preface

This Master's Thesis was written as a part of the *Electronic Systems* Master's programme at Aalborg University (AAU). The learning objectives for this module are summarised in the following paragraphs.

Knowledge: Have knowledge of the highest international level of research within a selected field of relevancy with *electronic systems* and comprehension w.r.t. research ethics.

Skills: Be able to argue the relevancy of the chosen problem and its context. Account for the scientific basis of the problem and methods to solve it. Analyse and describe the problem of choice, applying relevant theories, methods, and experimental data.

Competences: Be able to communicate scientific problems orally and in writing to specialists as well as non-specialists. Be capable of handling complex and unpredictable situation, which require new solutions, professional development and/or collaboration. Be able to independely initiate and take responsibility for collaboration and profession development, and specialisation.

Reading Guide

The thesis is written in chapters following a somewhat chronological order. In chapter 1 the chosen problem is described on a high-level basis, where the reader should be able to familiarise themselves with the chosen problem, and how the thesis will approach the solution. In chapter 2 the fundamental theories and methods relevant to the development of the solution are presented as a survey, and sections can be skipped if the reader is familiar with the subject; nothing is designed, implemented, or tested within this chapter. In chapter 4, 5, and 6 the solution is developed, important test results are presented, and reflections w.r.t. the solutions are described. chapter 7 seeks to discuss choices made during the development, the consequences, and possibly alternatives that might better the solution. chapter 8 summarises the solution as *a singular product* and contextualises the solution to the chosen problem. In chapter 9 topics and paths are highlighted, that would have been relevant to research or develop if more time was available. The appendices describe the most important tests performed throughout the development, however, the most important conclusions, data, and figures have been integrated into the relevant chapters.

In GitHub References a file tree can be found. The file tree refers to the Github page holding the files, scripts, data etc. used in the project. The GitHub directory tree is in an early stage and somewhat unmanageable, however, when relevant/important the path of scripts, data, etc. is explicitly stated and should be reachable from the appendix.

To improve the reading experience and provide navigation tools to the reader, references/ citations will (if possible) be clickable. Clicking on citations will move the reader to the Bibliography, clicking

on links will guide the reader to a browser, and clicking an internal reference will move the reader to the referenced item (table, figure, etc.).

This report includes a Table of Contents and a List of Figures. Every chapter, section, subsection, and figure will be referenced with a hyperlink ensuring proper overview.

Citations will follow the *Vancouver style*, and will thus be numbered (starting from 1 going up) in accordance with the order they are cited in the text. Clicking on the citation will take the reader to the Bibliography where the title, author, and other relevant information will be provided.

Due to the subject matter acronyms are frequently used and to avoid misconceptions a list of relevant acronyms (and their expanded forms) is present just after the bibliography (Acronyms).

The equations and formulas included in this report will follow the ISO 80000 standard whenever possible, however, to avoid ambiguity from figures to text the decimal separator will be ‘.’ and the thousands separator will be ‘,’. Typesetting will also follow the ruleset described in Table 1 in order to clearly distinguish between symbols and units.

Table 1: Typesetting for equations with examples.

Subject	Typeset	Examples
Numbers	Ordinary	$1,\ 2,\ 3, 10^{-24}$
Symbols	*Math cursive*	$V,\ A,\ F,\ c,\ l$
Units	Ordinary	volt, V, A, N, kg
Function calls	Ordinary	$\cos(x),\ \log(x),\ \exp(x),\ \arcsin(x)$
Words	Ordinary	$\lambda_{\mathrm{V}} = \frac{\text{final voltage}}{\text{starting voltage}}$
Greek letters	*Cursive*	τ, μ, β, Ω
Indices	Ordinary	$N_{\max},\ P_{\text{effective}},\ V_{\text{in}},$
Constants	Ordinary	e, c, i, j

When appropriate, the unit of the outcome of an equation will be placed in square brackets on the lefthand side of the formula numbers. All equations will be left aligned to the same margin and be followed by a description of all included variables.

Arithmetic for bit-sequences requires additional mathematical symbols to avoid ambiguity w.r.t. the performed operations:

$\oplus$	Addition with modulo 2
$\circ$	Concatenation
$\leftarrow$	Replace lefthand value with righthand value
$\wedge$	Bitwise AND
$\vee$	Bitwise OR

The footnotes will follow MLA style and be used for tangential information. Citations will never be presented as footnotes but will be displayed as described before.

Contents

List of Figures

1 Introduction

AI (Artificial Intelligence) has become an inherent part of modern society. Everybody interacts daily with systems that, to some extent, rely on the benefits of AI, e.g. recommendation algorithms on entertainment platforms and social media, image recognition in automated vehicles, and speech recognition in virtual assistants [5]. The concept of AI is present in the consciousness of non-experts, partially due to the portrayal in popular culture within the last century. The often antagonistic role of self-conscious AI-systems in science fiction has resulted in an extensive discussion within the public domain, about ethics and ramifications of AI. However, AI or more specifically ML (Machine Learning) algorithms have often proven very beneficial in data analysis, for clustering, classification, and segmentation of data sets [6][7]. AI and ML have seamlessly integrated into modern everyday life, despite the immense ethical implications of ''*granting*'' machines an extent of human intelligence. The cause is that the current confines of ML are manifested in more application-specific implementations, which have consistently proven a societal value.

ML is a general term for statistical algorithms that can make inferences about the distribution of datasets and is generally divided into three domains namely reinforcement- unsupervised-, supervised-learning [6][7][8]. Common for all approaches is the **learning** attribute, which is a description of the model's ability to adapt to an environment and thereby *learn* over time. Russell and Norvig [9] introduced the concept of **intelligent agents** as systems that *perceive* its environment and induce hypotheses about the external circumstances. By falsifying the hypotheses the intelligent agents can induce new hypotheses by exploiting previous metrics. This iteration scheme is known as **training** [8][9][10]. The three approaches differ in the manner in which they learn and train.

The *reinforcement learning* paradigm is based on the principle of carrot and stick, meaning a decision-making agent is performing a sequence of actions, where desired actions are rewarded and less attractive alternatives are punished [6][9][10]. Through trial and error, the model should iteratively approach the optimal program for the task, thereby maximising the accumulated reward [11].

The *unsupervised learning* paradigm is based on the principle of learning without confirmation of achievement; a common saying is that the models *"learn without a teacher"* [9][10]. Essentially the goal is to find patterns and regularities in unlabelled data, to make inferences about underlying distributions. One approach to unsupervised learning is known as **clustering**, which builds on the principle of density estimation in statistics [9][11][12]. The aim is to cluster or group the data based solely on dispersing the features within the set. The clustering can be performed without any assumptions of underlying distribution (i.e. non-parametric) or oppositely with *a priori* knowledge of the underlying distribution (i.e. parametric), which guides the clustering process based on assumptions about the data's distributional form. Procedures such as anomaly detection or feature space reduction can also be approached as unsupervised learning [10].

The *supervised learning* paradigm seeks a mapping from input to output variables. In other words, the aim is to find a function that matches given input features to an output label [10][13]. In simple terms, a supervised learning algorithm has two separate phases. First, a training phase where a set of annotated data is used to derive the aforementioned input/output mapping. Last, is an inference phase where the mapping is used to make predictions about unlabelled data [8][13]. Supervised learning applications can generally be divided into two branches; **classification** and **regression**. Classification tasks are to determine the label or class of an input feature, where some examples are e-mail spam detection or image classification. Regression differs from classification in that the inferences made in regression are concerned with predicting continuous outcomes, such as a numerical value or a range. In contrast, classification focuses on predicting discrete class labels or categories.

A multitude of statistical algorithms and models have been developed within the three approaches, however, ANN (Artificial Neural Networks) or simply NN (Neural Networks) have been at the pinnacle of machine learning algorithms for the past decades since the introduction of layer-wise deep network training in 2006 [14], which allows for the development of advanced high-performance models [11][15][16]. NNs are inspired by the processing abilities of the human nervous system and are both more versatile and complex than traditional statistical approaches [7][9][17]. Traditional algorithms rely on a sparse amount of intelligent agents to induce hypotheses about the data, why the algorithms usually are application specific [7]. The versatility of NNs comes from the scalability of intelligent agents used in the network. The NNs can scale to model large datasets with complex input/output relationships, and a great amount of features in the dataset [18].

The current frontiers within almost every field in science and technology rely on gathering and generating large quantities of data. The paradigm of **big data**-analysis is a recurrent issue for researchers and developers in the entire academic landscape; from healthcare and finance to biology and robotics [16][18]. The growing demand for data analysis has endowed research in computer engineering and statistics with a powerful tool, resulting in a variety of network architectures.

Associated with the progress of data analysis and machine learning is *the price to pay*. Financially and environmentally the burden of the seemingly ever-increasing size of the data models bears a toll; renting/purchasing the necessary hardware, electricity, compute time, and the environmental ramifications of producing the hardware and electricity [19]. Research from OpenAI uncovered a doubling of compute used to train new state-of-the-art models of around 3,4 months between 2012 and 2018 [20]. In a paper from 2019 regarding *energy and policy considerations for deep learning* it is estimated that training a big *transformer* with neural architecture search produces around five times the amount of CO_2 emissions than the lifetime of a car inclusive fuel.

Research and development of these data models are skewed toward increasing the accuracy of the final product rather than developing efficient implementations (see Figure 1.1). The outcome of the research is not to be understated, however, a balance must be found to ensure the sustainability of deep learning. Meeting these demands with traditional computing approaches that prioritise accuracy at all costs becomes increasingly challenging due to the infeasibility of Moore's Law and the breakdown of Dennard scaling [21]. Increasing computational load is therefore unsustainable in terms of energy consumption, processing time, and resource utilisation [22][23].

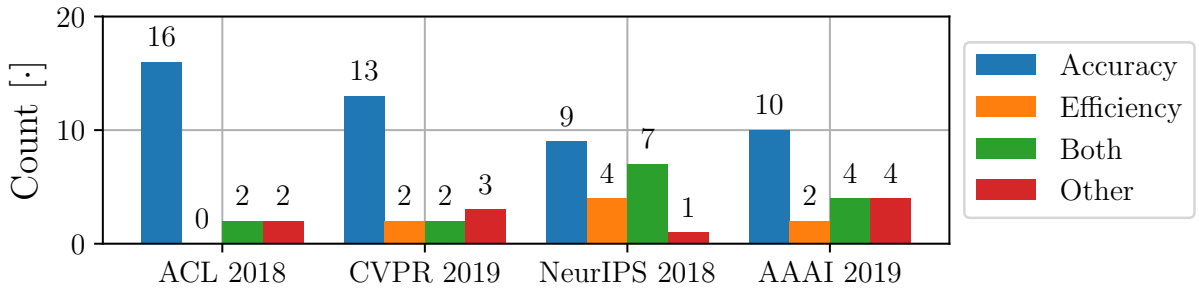

Figure 1.1: Distribution of foci from papers. A sample size of ~ 80 papers from four conferences is divided into four categories based on the main contribution claimed by the authors. Data w.r.t ACL, CVPR, and NeurIPS taken from [24] and w.r.t AAAI from [25].

The field of AC (Approximate Computing) has gained immense interest in recent years as an approach to comprehending the vast expansion of energy consumption and inference latency. The approach of AC is to reassess arithmetic problems in applications where the introduction/increase of error can be traded off for a disproportionate reduction in energy, resource, or time consumption. This, of course, excludes tasks that require the highest precision, e.g. tasks like diagnosing patients and control systems with human lives on the line. However, fault-tolerant applications like image- and speech-processing, data analysis, sensor data processing, and ML models could potentially benefit from utilising AC techniques. Based on the findings of *Understanding and mitigating noise in trained deep neural networks* by Semenova, Larger, and Brunner NNs are effective at avoiding the accumulation of noise and that the *signal-to-noise ratio* does not worsen when adding more layers [26], which indicates that introducing some error might not affect the accuracy of the models too much.

One of the techniques from *approximate computing* addresses arithmetic operations, whereby a decrease in area and/or latency can be exchanged for a drop in the accuracy of the operations. Methods exist wherewith circuits for addition and multiplication can be modified to automate this tradeoff, like CGP (Cartesian Genetic Programming) . However, the *ad hoc* approach is still relevant; say you have a multiplier design that is slightly too big for the implementation and you have to remove a couple of gates. It is difficult to know how the *removal* of gates will affect the circuit, i.e. how much *error* will be introduced. Furthermore, how will the ‘‘small’’ change affect the higher functionalities of a system? The ad hoc approach to developing an approximate ASIC (application-specific integrated circuit) , leaves developers with an unsystematic and ungovernable catalogue of options.

Leon et. al. [27] propose future points of investigation in the field of AC, where broad benchmarks are a critical point in making energy-efficient approximate systems a permanent staple in the development of *on the edge devices*. A foundation for fair comparison of AC approaches is desired for developers to determine the appropriate approximate arithmetic units for a given CNN. The ‘‘fair comparison’’ lies in developing general statistical evaluation methods that are inferable with the outcome of the CNN at hand. The statistical evaluation should ideally produce metrics related to the application’s performance.

It is desired to present a reproducible methodology for comparing a subset of approximation techniques applied to the netlist design of the hardware of a *CNN preliminary*: A *benchmarking tool* for AC techniques in the CNN application capable of using any AC design. This solution differs from the efforts in generalising approximate designs and rather embracing the ad hoc engineering approach previously described. Such a benchmarking tool would provide developers with an option to evaluate approximate circuit designs before implementation.

Frameworks for simulation of approximate computing techniques applied to CNNs already exist like: TypeCNN, AxDNN, and ProxSim [28][29][30]. However, these frameworks are aimed at the development of the CNNs and the demographic is the *machine learning developer* rather than the *circuit developer*. Furthermore, these three proposals are limited in the sense that the scalability and generalisation of the CNN models are of minor consideration. These methods contribute greatly to the methodology of designing cross-layer end-to-end simulations prior to hardware implementation but lack the *interpretation* of the actual outcome of a CNN model. In other words, *how are the approximations affecting the perceptual and recognisable features of the CNN?*

The purpose of this project is to achieve a method of **scaling** and **generalising** the cross-layer end-to-end approximate CNN simulations, to accommodate and interpret the raison d'être of CNN models prior to hardware implementation of the approximate circuits. This benchmarking system is summed up in the following problem statement:

"How can a benchmarking tool provide an ASIC developer with relevant metrics to evaluate an *approximate* arithmetic circuit as an integral part of a large scale system, i.e. a *neural network*, prior to implementation?"

Survey 2

Before tackling the development of the *benchmarking tool* conceived in chapter 1 a survey regarding the relevant concepts and methods is presented in this chapter. This chapter is divided into three parts: 2.1 Neural Networks, 2.2 Digital Design, and 2.3 Approximate Computing. In 2.1 Neural Networks the fundamental building blocks of *neural networks* are presented and the *state-of-the-art* neural networks are investigated. Before approaching the subject of *approximate computing*, specifically *approximate arithmetic circuits*, it is essential to understand *how* arithmetic logic circuits work and *why* approximating the functionality can lead to faster and/or more power-efficient circuits with leniency w.r.t. error. Lastly, 2.3 Approximate Computing presents the topic of *approximate computing* in broad terms, wherefrom relevant methods are drawn to be applied in later chapters.

2.1 Neural Networks

As described in chapter 1, NNs have been at the pinnacle of machine learning algorithms for the past decade, but come with a high computational cost and power consumption. This project aims to investigate the prospects of approximate- and stochastic computing in reducing the aforementioned metrics, without compromising the performance of the NNs proportionally.

The path from *problem identification* to the *model deployment* can be seen in Figure 2.1, it is the life of a NN in broad terms. Note, this is not unique for the development of an NN, however, it is meant as an overview. The developer *identifies a problem* that can be solved using a NN, data is collected, and the representation of the data is decided (i.e. what are the dimensions of the input to the NN). Next, the model is *designed*: A model is prepared by choosing architecture, optimisation algorithm, loss function, etc. The model is trained on the dataset, and the model is tested by inference. If the model yields satisfactory results it is ready to be *deployed* as a piece of software or implemented in hardware. If the model is unsatisfactory, the hyperparameters of the model must be adjusted and a new model will be prepared.

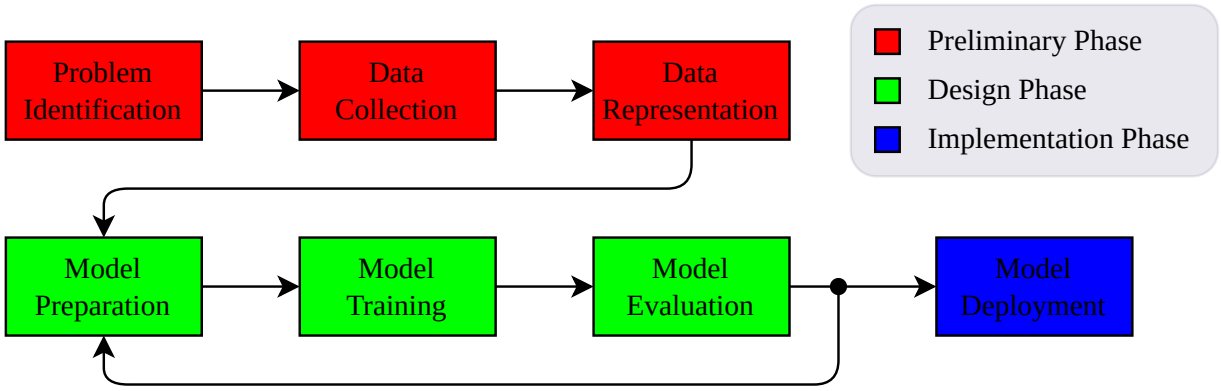

Figure 2.1: Neural networks from *problem identification* to *model deployment*. The seven steps are divided into three phases: *preliminary phase*, *design phase*, and *implementation phase*. The outcome of the first phase should be a clear blueprint of how the problem is formulated, i.e. what is the purpose, what is the relevant data, and how is it presented. In the second phase, a neural network is designed, trained, and evaluated. If the model is unsatisfactory, a new model may be designed based on the experience from the previous model; the output of the second phase should be a model that meets/exceeds the expectations w.r.t. speed, accuracy, and efficiency. The third phase takes the finalised neural network and implements it in the real world; examples could be as a software program that analyses data or image recognition in an embedded system.

The *preliminary phase* of Figure 2.1 is bound to the specifics of the chosen and immediate problem, however, the theory behind the building blocks of the NN models are generalised and worth investigating.

2.1.1 Perceptrons

The **perceptron** (also known as an *artificial neuron*), was initially presented by Rosenblatt [17] as a simple model of a biological neuron, designed to mimic some of the functions of the human brain's neural network, and is the building block of ANNs [31][32]. Concisely, the perceptron perceives an input, which can be from the environment or other perceptrons. Each input is assigned a scalar called **synaptic weights**. All scaled inputs are accumulated and processed through an **activation function**, which introduces non-linearity to the output [10][13]. A **bias** can be added as a synaptic connection before the activation function. An illustration of a perceptron is seen in Figure 2.2.

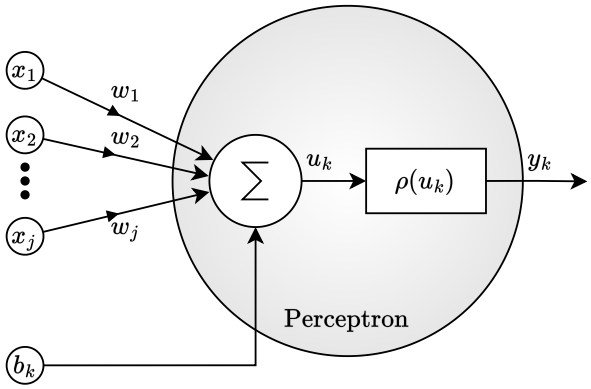

Figure 2.2: Illustration of a perceptron receiving j inputs, all scaled by an assigned synaptic weight w_j. All scaled inputs are accumulated in a summing junction along with a bias (b_k). The output of the summation u_k is used as input in an activation function $\rho(u_k)$. The scalar output y_k of the kth perceptron is obtained.

The perceptron is divided into a linear and non-linear part. The output of the linear (affine) contribution u_k is formally described as a MAC (multiply-accumulate) operation as shown in Eq. (2.1).

$$u_k = b_k + \sum_{n=1}^{j} w_n \cdot x_n \tag{2.1}$$

where:

u_k	The output of the MAC operation of the k^{th} perceptron
b_k	The bias contribution
x_n	The n^{th} input
w_n	The n^{th} synaptic weight
j	The total number of inputs

The MAC operation can also be written as the inner product of a vector of the inputs and synaptic weights:

$$u_k = \bar{w}\bar{x}^T \tag{2.2}$$

where:

$\bar{w}$	The vector $[1\ w_1\ w_2\ \dots\ w_j]$
$\bar{x}$	The vector $[b_k\ x_1\ x_2\ \dots\ x_j]$

The weights of the MAC-operation is a j-dimensional hyperplane, which can be used as a linear binary classifier [33]. The advantageous part of a NN is the non-linear contribution of the perceptron: **the activation function**. The activation function enables the perceptron to classify non-linearly separable sets and perform non-linear regression [34]. Multiple activation functions exist that can yield different results depending on the task of the network.

The most common realisations of the activation function are briefly presented with particular regard to the application that each function benefits. The following explanations are based on the surveys by Sharma et. al. [34], Dubey et. al. [35], and Nwankpa et. al. [36].

The sigmoid is one of the earliest activation functions known for being a bounded, differentiable function with positive real derivatives [36]. The main advantage of the sigmoid is that it yields smoothly transitioned output between 0 and 1, making it suitable for probabilistic interpretations. This bounded property also secures numerical stability [35]. In a historical context, the sigmoid is important, but compared to more modern approaches it is outperformed [35]. This is mainly due to a concept known as *vanishing gradient*. The issue is that the parameterised gradient becomes close to zero for large inputs, yielding undiminishable changes when training the NN.

The sigmoid function is given in Eq. (2.3).

$$\rho_{\text{sigmoid}}(u_k) = \frac{1}{1 + e^{-\alpha u_k}} \tag{2.3}$$

where:

$\rho_{\text{sigmoid}}(\cdot)$	The sigmoid activation function
α	The slope factor

The graph of Eq. (2.3) is shown in Figure 2.3.

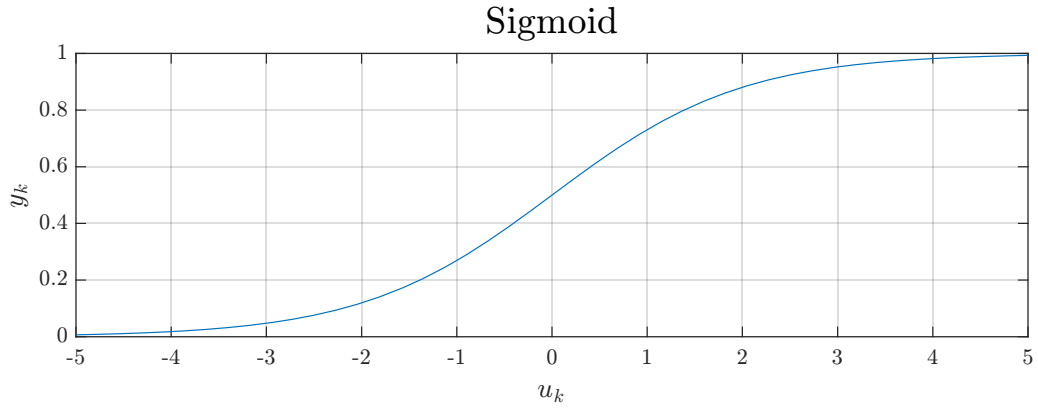

Figure 2.3: Plot of the sigmoid activation function, where the slope factor (α) is set to unity.

The hyperbolic tangent is similar in structure to the sigmoid because it is also a bounded, differentiable function with positive real derivatives. The difference is the hyperbolic tangent yields outputs between 1 and -1. The problem with a vanishing gradient is also present for the hyperbolic tangent, resulting in slow convergence while training the model [35]. Both the sigmoid and hyperbolic tangent activation functions rely on calculating the exponential function, which is computationally expensive.

The hyperbolic tangent activation function is shown in Eq. (2.4) and its graph is plotted in Figure 2.4.

$$\rho_{\tanh}(u_k) = \frac{e^{u_k} - e^{-u_k}}{e^{u_k} + e^{-u_k}} \tag{2.4}$$

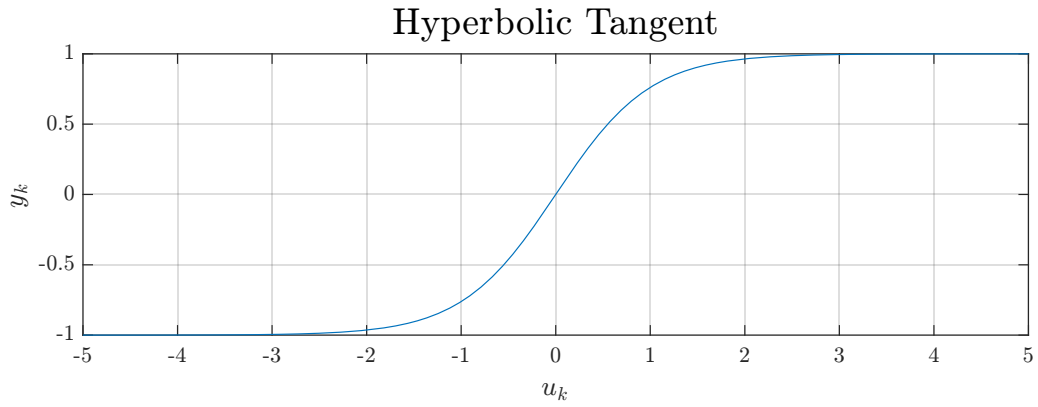

Figure 2.4: Plot of the hyperbolic tangent activation function.

The RELU (Rectified Linear Unit) is the most widely used activation function since its introduction (by Nair and Hinton in 2010 [37]) [35]. The RELU is the identity function if the input (u_k) is positive and zero otherwise. This is shown in Eq. (2.5).

$$\rho_{\text{ReLU}}(u_k) = \max(0, u_k) \tag{2.5}$$

The RELU function provides a simple computational structure for calculating its gradient; 0 for negative inputs, 1 for positive, and undefined at 0 (how it is handled is up to the implementation). A plot of the ReLU function is shown in Figure 2.5.

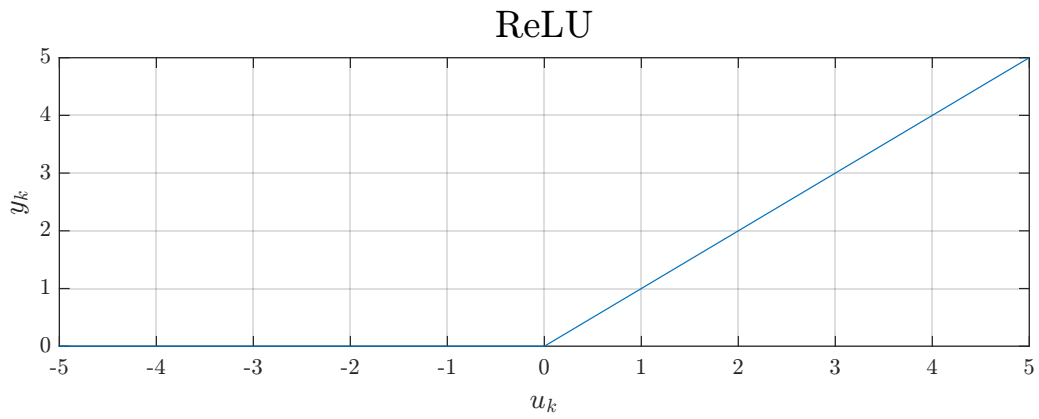

Figure 2.5: Plot of the ReLU activation function.

The problem of vanishing gradients from both the sigmoid and hyperbolic tangent functions also exists for the ReLU as the gradient for negative values is zero. For this reason, a broad selection of ReLU variations exists that tries to accommodate this issue, e.g. the *leaky ReLU* (a special case of the *parametric ReLU*), where instead of forcing negative values to zero they scaled by a factor of 0.01 [35]; the problem of the vanishing gradient is thus overcome since the gradient will be 0.01. However, neither the accuracy of the developed models nor the convergence time *significantly* improved by implementing these alternatives [35].

2.1.2 Multilayer Perceptrons

The perceptron is the building block of a DNN (Deep Neural Network) and especially MLP (Multilayer Perceptrons) , essentially a structure of layered perceptrons, where inputs to one perceptron are the output of a previous [8]. The perceptrons in an NN are also referred to as **neurons** or **computational nodes**. The following presentation focuses on fully connected NNs, which have multiple perceptrons asserted parallelly in **hidden layers**. The parallelism of the perceptrons in a fully connected NN allows the modelling of different features within the stimulus, as the same input is assigned multiple synaptic weights [32][33]. By adding hidden layers a hierarchical organisation of features allows the network to capture more abstract and context-dependent patterns in the data, leading to improved generalisation and robustness [11][15][31].

The notation of parameters within the NN used throughout this thesis is presented in Table 2.1.

Table 2.1: This table lists the symbols and notations used to represent parameters of a NN throughout the thesis. An understanding of these notations facilitates interpretation and comprehension of the following algorithms.

Parameter	Specification
l	The index describes the layers, with the input layer being $l = 0$ and the index incrementing by one as the network propagates forward.
j	The number of nodes in the previous adjacent layer.
n	The index describes the individual nodes of the previous adjacent layer.
k	The index describes the individual nodes of the current layer.

Continued on next page

Table 2.1: (Continued)

$w^l_{n,k}$	The synaptic weight between the output of the n^{th} node in layer $l-1$, and the input of node k in layer l.
b^l_k	The bias of node k in layer l.

To illustrate the hierarchical structure of an NN and the notation of synaptic weights as parameters an example is shown in Example 2.1.1.

Example 2.1.1: Two-Layer Feed-Forward Neural Network

A fully connected NN example is shown in Figure 2.6. The illustration proposes a NN structure that has two hidden layers, an input signal x, and three output nodes. To demonstrate the notation of synaptic weights presented in Table 2.1, two specific examples are provided, namely $w^2_{2,3}$ and $w^3_{3,1}$. To keep a simple and comprehensible illustration, the bias to every node is 0 and hence not displayed in the figure.

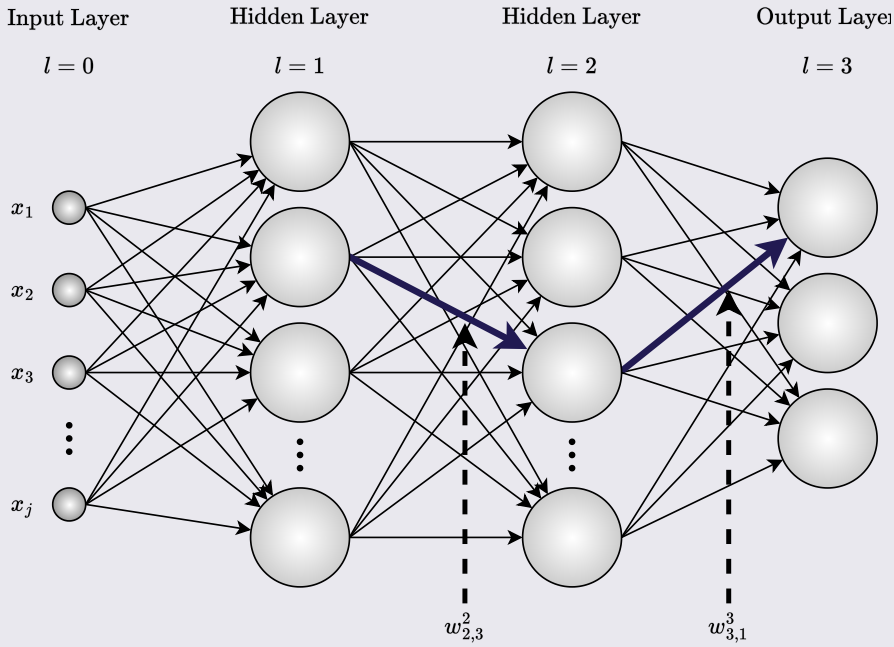

Figure 2.6: Illustration of a two-layer fully connected NN. The index of each layer is indicated as l. The synaptic weights are implicitly referenced as arrows and two specific synaptic weights illustrated as enlarged blue arrows are indexed as $w^2_{2,3}$ and $w^3_{1,1}$ respectively.

2.1.3 Training Perceptrons

Previously in this thesis, introductory remarks were made on training a NN, however, a more detailed presentation is provided, specifically regarding supervised learning tasks. As previously mentioned the training of the model refers to finding the synaptic weights (and biases) that yield a desired output. The training of a neural network is usually an **online learning** scheme, i.e. the network model is trained on a single input signal at a time [8][9]. Online learning aids the network's robustness to divergent inputs as the synaptic weights are stored between each iteration, and are updated from previously learned weights [38][39].

The training of a NN can be separated into steps, beginning with an **initialisation** of the synaptic

weights, which is assumed to be random for simplicity. An input is then **propagated** through the network and an output is obtained.

Loss Function

In a supervised learning application, the desired output is known in advance and a measure of the model's accuracy on a given input signal can be assessed. This measure is formally known as a **loss function**, which varies depending on the application. The loss function $L(\bar{y}, \hat{\bar{y}})$ is a function of the desired output ($\bar{y}$) and the output found by propagating the input through the network ($\bar{y}$) [9][38]. The MSE (mean-square error) is an example of a discernible and widely used loss function and is shown in Eq. (2.6).

$$L_{\text{MSE}}\left(\bar{y}, \hat{\bar{y}}\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2 \tag{2.6}$$

where:

$L_{\text{MSE}}(\cdot)$	The MSE loss function
$\bar{y}$	The desired output vector
$\hat{\bar{y}}$	The obtained output vector
N	The length of the output vector

Error Back-Propagation

With a mathematical description of the loss function, it is possible to compose a minimisation problem. It is desired to minimise the loss by adjusting the synaptic weights of the network. Provided that the problem is minimisable, minimisation will require the partial derivative of the loss function w.r.t. the synaptic weights. Intuitively the partial derivatives describe the relation between a perturbation in a synaptic weight and the change in the output of the loss function [6][8][33].

The desired derivatives relative to the synaptic weights are seen in Eq. (2.7).

$$\frac{\partial L}{\partial w^l_{n,k}} \tag{2.7}$$

Since the layered structure of perceptrons introduces non-linearity and parallelism, it can be challenging to attain these derivatives. To attain these partial derivatives, the BP (Back-Propagation) algorithm is introduced. The algorithm was originally presented by Rumelhart et. al. in [40] and employs **the chain rule** to propagate the metric evaluated by the loss function backwards through the network.

The first step is to find the gradient of the loss function (L), with respect to the output of the forward pass ($\hat{\bar{y}}$) of an input ($\bar{x}$). The loss is propagated backwards from the output layer. The gradient of the loss is found with respect to the pre-activation outputs ($\bar{u}^l$) (which will be denoted $\bar{\delta}^l$ for later convenience), using the chain rule as shown in Eq. (2.8) [15].

$$\bar{\delta}^l = \frac{\partial L}{\partial \bar{u}^l} = \frac{\partial L}{\partial \bar{y}^l} \odot \frac{\partial \bar{y}^l}{\partial \bar{u}^l} = \frac{\partial L}{\partial \bar{y}^l} \odot \rho'(\bar{u}^l) \tag{2.8}$$

where:

$\odot$	The Hadamard product

Further propagation of the loss, back in the layers of the MLP, is straight-forward. Firstly, the error needs to be propagated backward one layer. To obtain such an error metric the matrix-vector

product is found for the weight matrix ($\mathbf{W}^l$) and the gradient loss vector ($\bar{\delta}^l$). By applying Eq. (2.8) to the newly obtained vector, the gradient loss vector of the previous layer is obtained as seen in Eq. (2.9) [15].

$$\bar{\delta}^{l-1} = ((\mathbf{W}^l)^T \bar{\delta}^l) \odot \rho'(\bar{u}^{l-1}) \tag{2.9}$$

By repeating Eq. (2.9) the loss is propagated backwards through the entire MLP. To obtain the gradient of the loss function with respect to the synaptic weights Eq. (2.10) is used [15].

$$\frac{\partial L}{\partial w^l_{n,k}} = y^{l-1}_n \cdot \delta^l_k \tag{2.10}$$

The obtained gradients can now be used to minimise the loss function through **gradient descent** algorithms [6][33]. The algorithm for training a NN is presented in Algorithm 1.

Algorithm 1 Training of a Neural Network

Step 1: Initialise all synaptic weights ($w^l_{n,k}$)
Step 2: Forward pass one input and obtain an output (y^l)
Step 3: Calculate loss ($L(\bar{y}, \hat{\bar{y}}^l)$)
Step 4: Backwards pass errors and obtain the partial derivatives with respect to each synaptic weight ($\frac{\partial L}{\partial w^l_{n,k}}$)
Step 5: Minimise the loss using a gradient-based optimisation algorithm.
Repeat Step 2 to 5 until all inputs have been used

2.1.4 State-of-the-Art Neural Networks

Fully connected feed-forward NNs has traditionally been the pinnacle of deep learning architectures but modern research has provided multiple **state-of-the-art** architectures. The list includes CNN (Convolutional Neural Network) used extensively in computer vision applications [6][41], RNN (Recurrent Neural Networks) especially viable for applications with sequential data such as language processing and speech recognition [42], and GNN (Graph Neural Networks) suited for applications with graph-structured data, which could be social media recommendation algorithms or chemical drug discovery [43].

A CNN addresses an issue with the fully connected NNs: Recognition of features in the input is not necessarily invariant of the position in the signal. A CNN is a DNN that relies upon parameter-sharing to local connectivity to obtain **translation invariance** [41][44][45]. The parameter-sharing property is inherent in the name of the architecture, namely the **convolution** of the input signal with a **kernel filter**. A CNN is trained like the fully connected NN using the BP algorithm to obtain the partial derivatives that enable gradient-based optimisation [45][46][47]. The synaptic weights perturbed in the fully connected NN are replaced by the kernel filters convolved with the input signal [46]. A comprehensive study by LeCun et. al. [47] presents a specific CNN architecture (today known as *LeNet-5* [44]) which presents three architectural concepts, which are still used in a variety of configurations to this day [46]. The building blocks of the CNN are *convolutional layers*, *activation layers*, *subsampling/pooling layers*, and fully connected layers.

Convolutional Layers

The convolutional layer accounts for the equivalent of the linear classifier in the perceptron. An input signal X, a tensor of N dimensions is convolved with a series of parameterised kernels,

analogous to the synaptic weights in the MLP. The kernels are smaller matrices (or tensors) than the input matrix/tensor. For each shift, the inner product is found and all the inner products of one kernel are gathered in a FM (Feature Map). The kernel and corresponding subset of the input is *flattened* into vectors and their inner product is found as seen in Eq. (2.11).

$$\mathrm{FM}_{i,j,\ldots,k} = (I * K)_{i,j,\ldots,k} = \sum_{x=1}^{m_i} \sum_{y=1}^{m_j} \cdots \sum_{z=1}^{m_k} K_{x,y,\ldots,z} I_{i-x,j-y,\ldots,k-z} \tag{2.11}$$

where:

FM	The resulting FM-tensor
I	The input tensor
K	The kernel tensor

It is emphasised that the mathematical principle behind the convolutional layer is identical to that of the MLP, an inner product. The difference exists in the input/kernel configuration.

An example of a convolutional layer using two-dimensional inputs is provided in Example 2.1.2.

Example 2.1.2: Convolutional Layer

The following example of a convolutional layer in a CNN is from [48]. However, the concepts are congruent with those in [44][46][47].
The example has a 9×9 input matrix convolved with a 3×3 kernel, yielding a 7×7 FM, illustrated in Figure 2.7.

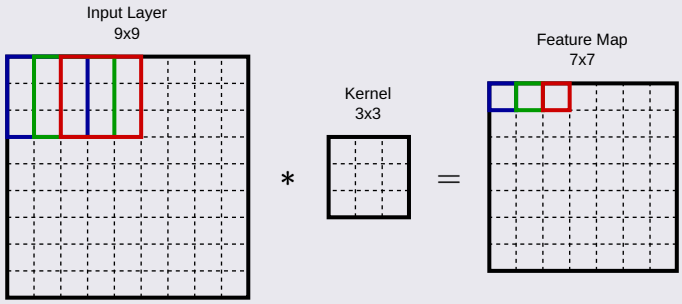

Figure 2.7: Illustration of a two-dimensional convolution layer, using a single kernel. The coloured squares provide an intuitive understanding of the relationship between the convolution with a kernel and the corresponding value in the FM. The figure is from [48].

Analogous to applying multiple nodes within a single layer in the MLP, a CNN can have multiple kernels, corresponding to an identical amount of FMs. An example using four kernels is illustrated in Figure 2.8.

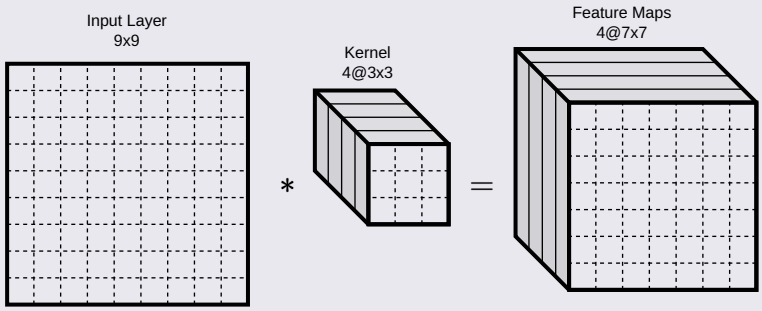

Figure 2.8: Illustration of the example convolutional layer, using four kernels resulting in four FMs. The figure is from [48].

Activation

The CNN needs a non-linear activation for the reasons explained in subsection 2.1.1. The activation functions used are usually the same as for MLPs [44]. As previously mentioned the most widely used activation function is the RELU function. The activation function is applied to each entry in each FM as shown in Eq. (2.12).

$$y_{i,j,\ldots,k} = \rho_{\text{ReLU}}(FM_{i,j,\ldots,k}) \tag{2.12}$$

Pooling/Subsampling

A crucial part of the CNNs is a layer known as both subsampling or pooling. The purpose of the pooling layer is to downsample the features to simplify calculations and reduce complexity as the network progresses [44]. The pooling layer can intuitively be interpreted as selecting only the most significant activations to investigate further [46].

The pooling layer is distinguished into two options; **max-pooling** and **average-pooling**. Max-pooling slides a window across each FM and selects the largest value of the window for each shift. This operation reduces the size of the FMs, but not the number of FMs. The average-pooling calculates the average of all activations within the window. An example of both max- and average-pooling is provided in Example 2.1.3.

Example 2.1.3: Pooling Layers

This example presents both max- and average-pooling, for a 4×4 input FM. The values in the FM are chosen arbitrarily, for the sole purpose of providing this example. The example is seen in Figure 2.9.

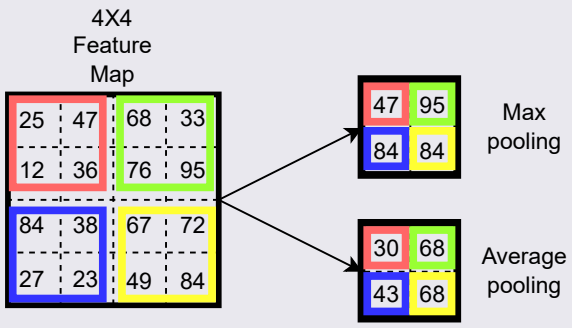

Figure 2.9: Illustration of both max- and average-pooling based on an FM with arbitrary values. The purpose of the coloured squares is to indicate the relationship between the FM and the pooled outputs.

As previously mentioned the architecture of a CNN is a varying configuration of convolutional and pooling layers. The network typically ends with a few fully connected layers configured based on application [46][47]. To this day the CNN paradigm is state-of-the-art in the field of pattern recognition and is widely used in computer-vision applications [45][44][49].

Full-Scale Convolutional Neural Networks

The first groundbreaking CNN architecture was the LeNet-5 [47] which provided the architecture presented in subsection 2.1.4. LeNet-5 was used for classifying handwritten digits (i.e. MNIST [50] dataset). In 2012 Krizhevsky et. al. presented a comprehensive CNN model for the classification of high-resolution images in the Image Classification on ImageNet [51]; today known as the *AlexNet* [52]. The AlexNet was at the time the pinnacle of CNN architectures and has therefore

been the baseline of comparison for architectures developed since. However, the basis for comparing architectures should be made considering the context and requirements of the particular research or application.

The impressiveness of the CNN in action can be seen in part by the number of papers with the focus of developing/benchmarking CNNs and in part by the results within these papers. A large collection of state-of-the-art benchmarking metrics w.r.t. CNNs is available on the website paperswithcode.com. *Papers with Code* states in their **about section**, that it is a ''free and open resource with Machine Learning papers, code, datasets, methods, and evaluation tables''. Furthermore, users can submit their results with an accommodating paper; to be included as a benchmark it is required that the paper has been published. In Figure 2.10 the progress of the accuracy on CIFAR-100 (without using extra training data) can be seen.

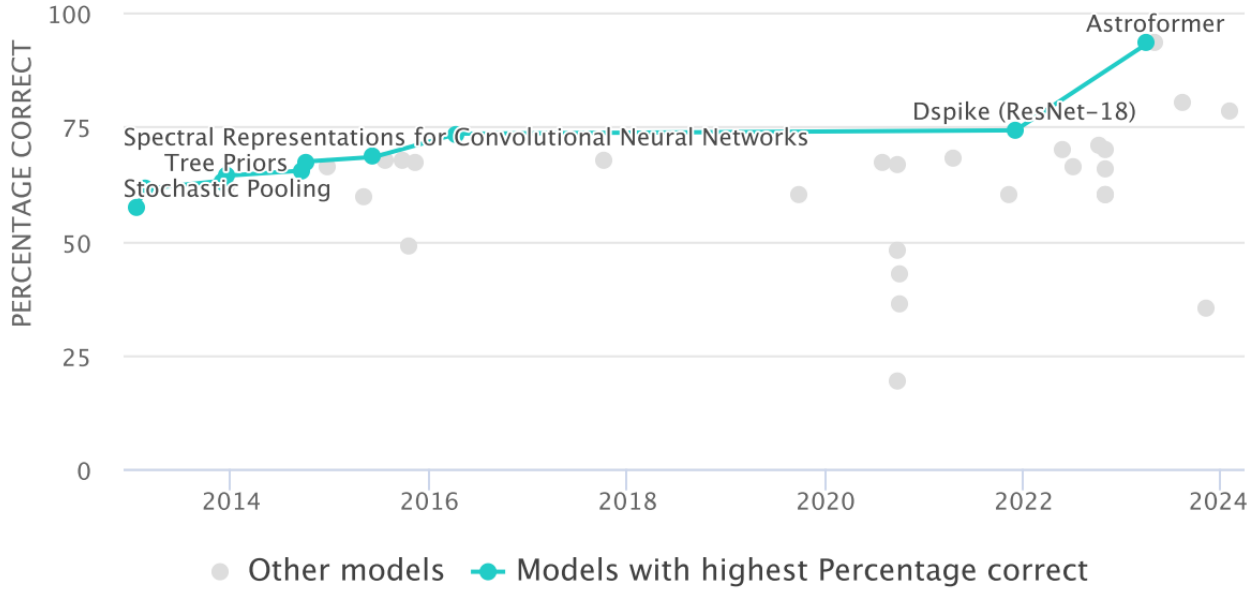

Figure 2.10: Overview of best-performing networks in terms of accuracy, performing image classification on CIFAR-100 without using extra data (image taken from paperwithcode.com).

The cyan line of *models with highest percentage correct* from Figure 2.10 are listed and summarised in Table 2.2. Interestingly, all the benchmarks presented in the articles utilize CNNs or the convolutional layers.

Table 2.2: Overview of the *models with highest percentage correct* from Figure 2.10. The right-most column indicates whether or not the benchmark, from which the data for Figure 2.10, is based on CNN architecture.

Name	Summary	Year	CNN (✗/✓)
Stochastic Pooling [53]	A method for regularising large CNNs by changing the pooling operations with stochastic variants	2013	✓
Maxout Network [54]	Using dropout and introducing maxout in tandem to improve optimization of deterministic feedforward architectures	2013	✓

Continued on next page

Table 2.2: (Continued)

Tree Priors [55]	A method for improving classification by introducing grouping and shared features within the groups	2013	✓
NiN [56]	Utilising multilayer perceptrons for convolution and global average pooling as a replacement for CNN fully connected layers	2014	✓†
DSN [57]	Focused on improvement of three aspects of CNNs: Transparency between intermediate layers and classification, discriminativeness and robustness of learned features, and effectiveness in training	2014	✓
HD-CNN [58]	Introduces hierarchical deep CNNs	2015	✓
Spectral Representations for CNNs [59]	Using discrete-time Fourier transform and spectral pooling, a speedup in computation can be gained	2015	✓
ResNet+ELU [60]	Exchanging the ReLU and Batch normalization for an *exponential linear unit* in residual networks	2016	✓‡
Dspike (ResNet-18) [61]	Introduces a family of ''DSpike'' functions that evolve during training, to overcome the problem of finding the gradient in spiking neural networks	2021	✓‡
Astroformer [62]	A method to learn more from less data.	2023	✓

† Similar to CNNs, discrepencies exist in the convolution performed by the new layers.
‡ The presented ResNet utilises convolutional layers.

From Table 2.2 it is clear, that CNNs are *state-of-the-art* when it comes to image classification and that the theory behind CNNs can be expanded, modified, introduced into other architectures, etc. with a positive influence.

2.2 Digital Design

The necessary computations to facilitate NNs are the MAC operations. In Figure 2.11 the basic MAC-unit can be seen; the unit consists of two mathematical operations-blocks and one memory cell: *Multiply*, *Accumulate*, and *Register*, respectively. A set of numbers $(x,\ y)$ are multiplied together. The product is then accumulated with the value from the register. The sum is then available at the output and is saved in the register to be utilised with the next multiplication and accumulation. This represents the following mathematical operation:

$$d = a + b \cdot c \tag{2.13}$$

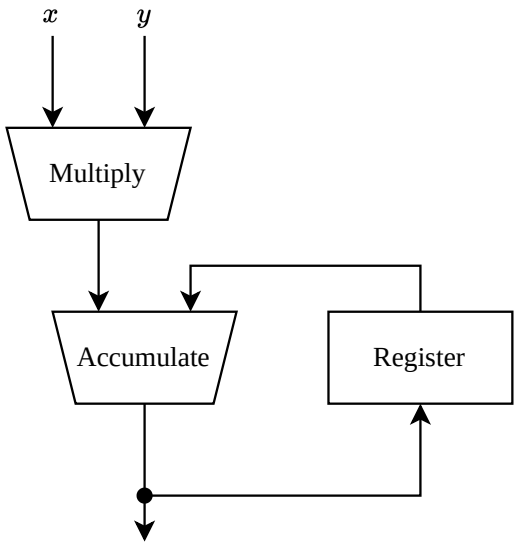

Figure 2.11: MAC unit.

This operation is extensively used in NNs to calculate all the u_ks in each perceptron, as seen in Eq. (2.1) and (2.2).

2.2.1 Number representation

The number representation of the operands is the deciding factor for the design of the architecture for the arithmetic operations. Two of the most common representations are FXP (fixed point) and FLP (floating point) .

Fixed-Point Number Representation

The decimal value of a sequence of N bits x_i, $0 < i < N-1$, represented in unsigned FXP can be calculated using Eq. (2.14) [63]:

$$x_{\text{unsigned}} = 2^{-b} \sum_{n=0}^{N-1} 2^n \cdot x_n \tag{2.14}$$

where:

x	The decimal representation of the number
x_i	The ith bit from the right, representing x
N	The total number of bits
b	The index where the weight of the bit should be unity

The terms *unsigned* and *signed* indicate whether or not it is possible to represent negative values, and for FXP the formula used to convert an unsigned bit-sequence to decimal values can be seen in Eq. (2.15) [63]:

$$x_{\text{signed}} = 2^{-b} \left(-\underbrace{2^{N-1} \cdot x_{N-1}}_{\text{MSB}} + \sum_{n=0}^{N-2} 2^n \cdot x_n \right) \tag{2.15}$$

Investigating the difference between Eq. (2.14) and (2.15) makes it clear, that the MSB (most significant bit) is the only term that can take a negative value. This discrepency effectively shifts the range of representable numbers from only positive to an almost even split of negative and positive numbers. A visualisation of the shift can be seen in Figure 2.12.

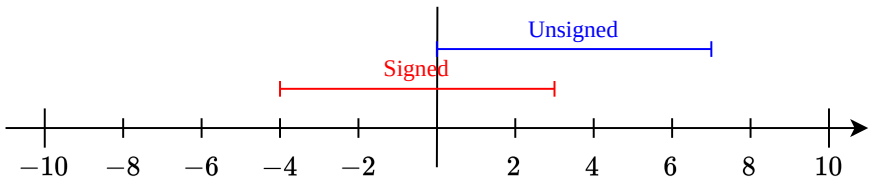

Figure 2.12: Intervals of representable values using 3 integer bits, no fractional bits, and signed/unsigned. Both intervals include 8 integers, however, the signed representation is slightly skewed towards negative numbers, being able to represent one more negative value than positive values.

The precision of signed/unsigned FXP numbers in the entire range is always the value of the LSB (least significant bit) , and can thus be changed by changing the value of b. Lowering b will lower the precision and increase the range and vice versa.

Floating-Point Number Representation

The other common number representation is FLP, with which a larger range is available. The larger range comes at a cost since the number of states possible with N bits does not change. The decimal value of a sequence of bits represented in FLP can be calculated using Eq. (2.16) [63]:

$$x = (-1)^{x_{N-1}} \cdot r^{E} \cdot M \tag{2.16}$$

where:

x	The decimal representation of the number
x_{N-1}	The MSB used as a sign bit
r	The radix of the exponent, being 2 for binary representation
E	The exponent, an integer value comprised of a sequence of bits
M	The mantissa, a fractional value between 1 and 2 (or 1/2 and 1)

Floating-point representation is standardised by IEEE (The Institute of Electrical and Electronics Engineers) to simplify interoperability of programs on devices that adhere to the standard [64]. An excerpt from the standard can be seen in Table 2.3. The naming scheme of the floating point representations is `binary{N}` because the radix is set to 2. From the information presented in the table, Eq. (2.16) can be elaborated; the exponent is represented using w bits and the mantissa is represented by t bits. Furthermore, based on the value of the *bias* the exponent can be negative (since the bias is subtracted from the value of the exponent), and since the trailing mantissa field widths contain one fewer bit than the precision, the mantissa includes an implicit 1.

Table 2.3: Binary interchange format parameters from IEEE std 754-2019 [64].

Parameter	`binary16`	`binary32`	`binary64`	`binary128`	`binary{N}`, ($N \geq 128$)
N, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$N - \text{round}(4 \cdot \log_2(N)) + 13$
$\max(E)$, maximum exponent E	15	127	1023	16383	$2^{N-p-1} - 1$
Encoding parameters					
bias	15	127	1023	16383	$\max(E)$
sign bit	1	1	1	1	1
w, exponent field width in bits	4	8	11	15	$\text{round}(4 \cdot \log_2(N)) - 13$
t, trailing mantissa field width in bits	10	23	52	112	$N - w - 1$

Note: Recreation of table 3.5 from the standard, however, the terminology and symbols have been changed to align with the rest of the section.

In Example 2.2.1 an FLP bit sequence is converted to its corresponding decimal value.

Example 2.2.1: `binary32` following IEEE std 754-2019

Figure 2.13 examplifies a floating point number (`binary32`). The MSB is 0 signaling that the float is positive. The following eight bits represent the exponent with a value of 170, however, the bias must be subtracted and the actual value is 43. The remaining bits are for the mantissa and with the implicit 1 it equals ~ 1.6666666 and the number is approximately equal to $2^{43} \cdot 1.6666666 \approx 1.466 \cdot 10^{13}$.

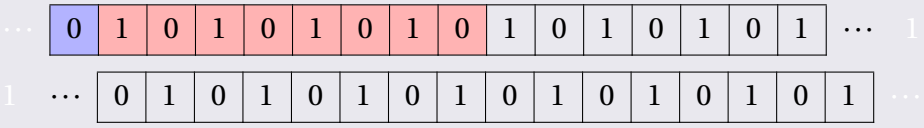

Figure 2.13: IEEE 754 `binary32` number representation (single-precision `float`) [64]. One sign-bit, seven bits for the exponent, and 24 bits for the mantissa.

Unlike FXP, the precision of FLP values is not invariant over the entire range. The cause of this can be found in Eq. (2.16) and specifically the multiplication of the radix to the power of the exponent and the mantissa; the LSB of the mantissa has a constant value, however, when multiplied with another term, the value of the LSB is suddenly dependent on the value of the radix and exponent. The benefit of the changing precision is that at small numerical values, the LSB represents a small value and the precision is high. However, when large numbers are represented, the precision is lowered to accommodate a larger range.

2.2.2 Arithmetic in Computer Systems

The MAC unit presented in Figure 2.11 represents the baseline for calculating a sum of multiplications in computer systems. To explore the individual blocks of the unit, it is essential to know the number representation; the architecture of the `Multiply` and `Accumulate` blocks are dependent on the chosen number representation. Exploring the arithmetics of FXP (fixed point) representation lays the foundation for the subroutines used to perform arithmetic operations with FLP (floating point) representation.

Fixed-point Addition

The addition of two sequences of bits performed in a computer system or any hardware is similar to regular pencil-and-paper addition of radix-10 numbers, the rightmost digits are added, and if the sum exceeds the value of the radix, the digit on the left side is incremented, move one position to the left and repeat, etc. However, computer systems are limited by the number representation (see subsection 2.2.1). Say two 1-bit numbers are to be summed and the sum must conform to the same form as the inputs; all states are written out in Eq. (2.17), (2.18), (2.19), and (2.20).

$$0 \oplus 0 = 0 \tag{2.17}$$
$$0 \oplus 1 = 1 \tag{2.18}$$
$$1 \oplus 0 = 1 \tag{2.19}$$
$$1 \oplus 1 = 0 \tag{2.20}$$

In Figure 2.14 the leftmost column contains the I/O diagram for the presented case, and the logic gates required to implement the summation following the set of equations above. This addition holds for three out of the four cases, however, in Eq. (2.20) it is clear that information is lost since $1+1=2$. This value cannot be represented by the single output bit and a carry bit can be introduced.

This will add another output to the I/0 diagram and add one logic gate to the implementation, both of which can be viewed in the 2 column of Figure 2.14. This is known as a `half-adder`.

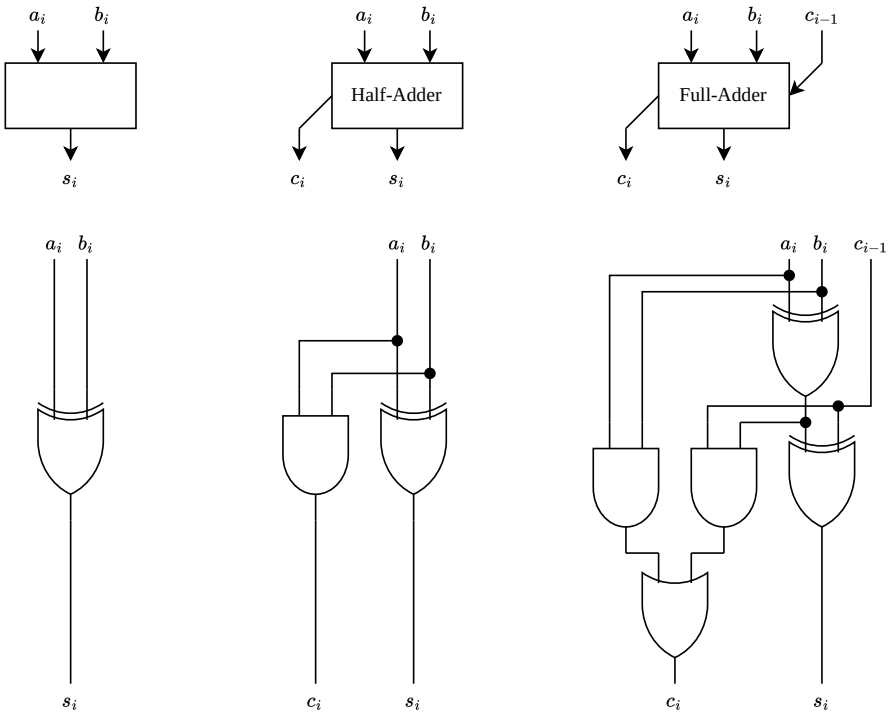

Figure 2.14: 2-bit adders. From left to right; a simple adder with inputs a and b producing the output sum as one bit, `Half-Adder` with the same inputs producing the sum as two bits (one carry-out), and `Full-Adder` with inputs a, b, and a carry-in from the previous adder c_{i-1} producing the sum as two bits (one carry-out).

Say the following two bit-sequences are to be added:

$$a = \{a_{n-1},\ a_{n-2},\ \ldots,\ a_0\}$$
$$b = \{b_{n-1},\ b_{n-2},\ \ldots,\ b_0\}$$

The addition can be viewed as n 1-bit additions, where the carry is propagated from right to left. A problem arises immediately, as the half-adders cannot receive the carry-bit. Adding an extra input to accommodate a carry bit from another operation produces the `full-adder` which can be seen in the rightmost column in Figure 2.14. As a consequence of adding the ability to receive a carry-bit as an input, three extra logic gates are added.

A full n-bit adder can thus be constructed from a series of full-adders connected as seen in Figure 2.15. This combination of half-adders comprises an RCA (ripple-carry adder) .

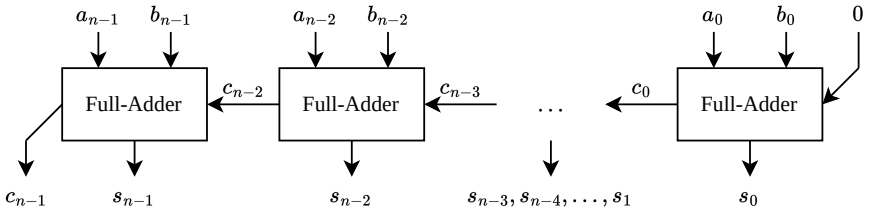

Figure 2.15: Adder consisting of a series of `full-adders`.

A problem that arises with the use of this kind of RCAs is the slowness associated with the RCA [65]. The carry has to propagate from right to left, requiring a long ‘‘critical path’’ from the addition of the LSB to the MSB. This problem is exacerbated when multiplying since it requires multiple summations. Many designs have been proposed to overcome the slowness of the RCA, some of which are mentioned and briefly described in the list below:

- CS (conditional-sum adders)
  - Two sets of circuits can perform additions based on assumptions w.r.t. the value of the carry bit. Both possible output states are prepared and using a multiplexer, the correct value is chosen when the carry bit is calculated [65].
- CCSA (carry-completion sensing adders)
  - Using dependent/independent carries, independent carries are detected, dependent carries are generated based on the independent carries, the sum-bit at each index can lastly be calculated as the sum of the generated carry bit and the inputs with modulo 2 (the calculation of the sum-bits are independent at the last step) [65].
- CLA (carry lookahead adders)
  - Using a *carry generation function* and a *carry propagation function* it can be shown, that the carries can be calculated in parallel with a *carry-lookahead unit* avoiding the linear delay scaling of the RCA [65].

Rather than designing a module specifically for **subtraction** of two sequences of bits, it is common to use *subtract-by-addition* utilising additional gates to enable the **complement** of the subtrahend, whereby addition between the minuend and complemented subtrahend will yield the difference between the two. The complement of a signed FXP number represented as in Eq. (2.15) can be found by negating all bits and adding 1 LSB.

Fixed-point Multiplication

The basic idea of **multiplication** of two sequences of bits is identical to **long-multiplication**; the multiplicand is multiplied with each digit (bit) of the multiplier, and the product is then shifted according to the placement of the digit of the multiplier. The partial products are then summed and the resulting value is the product of the multiplicand and the multiplier. This is exemplified in Figure 2.16 and 2.17 with decimal values and binary values, respectively.

Example 2.2.2: Long-multiplication with decimal values and binary values

Figure 2.16 and 2.17 illustrates long-multiplication in two number formats. The product of the multiplicand and the individual digits of the multiplier is calculated and shifted according to the placement of the digit and trailing zeros are added. This holds for both decimal and binary multiplication.

$$\begin{array}{rr} & 32 \\ \times & 132 \\ \hline & 64 \\ & 960 \\ + & 3200 \\ \hline & 4224 \end{array}$$

Figure 2.16: Long-multiplication of $32 \cdot 132$ in decimal.

$$\begin{array}{rr} & 1001 \\ \times & 111 \\ \hline & 1001 \\ & 10010 \\ + & 100100 \\ \hline & 111111 \end{array}$$

Figure 2.17: Long-multiplication of $9 \cdot 7$ in binary.

The approach in the example above is also known as *add-and-shift* and is the basis for some **sequential multiplication algorithms** in computer arithmetic [65]. At the gate level, the partial products of the binary multiplication can be found by ANDing the multiplicand with the digit of the multiplier and shifting the product. Instead of calculating all the partial products and summing them in the end, they are calculated one by one and added to a cumulative sum.

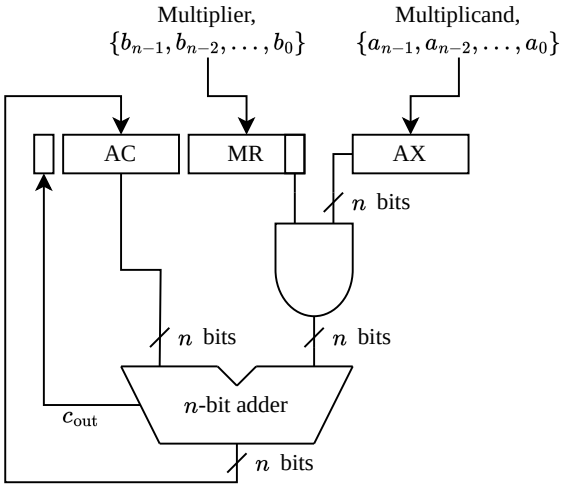

Table 2.4: Values of registers AC ∘ MR per operation. The product of the two n-bit is stored accross the two registers when the operation is finalised.

AC	MR
$\{0, 0, \ldots, 0\}$	$\{b_{n-1}, b_{n-2}, \ldots, b_0\}$
$\{c_{\text{out}}, s_{n-1}, \ldots, s_1\}$	$\{s_0, b_{n-1}, \ldots, b_1\}$
$\{c_{\text{out}}, s_n, \ldots, s_2\}$	$\{s_1, s_0, \ldots, b_2\}$
$\{c_{\text{out}}, s_{n+1}, \ldots, s_3\}$	$\{s_2, s_1, \ldots, b_3\}$
⋮	⋮
$\{c_{\text{out}}, s_{2n-2}, \ldots, s_n\}$	$\{s_{n-1}, s_{n-2}, \ldots, s_0\}$

Figure 2.18: Sequential multiplier. There are three registers to hold the input values and the cummulative sum; *accumulator* (AC) holds the result from the summation performed in the adder, *multiplier register* (MR) holds the value of the multiplier slowly shifting the least significant bits from the cummulative sum in from the left, and *auxiliary register* (AX) holds the multiplicand.

The circuit presented in Figure 2.18 follows five steps to perform a full multiplication of two n-bit numbers [65]:

I) Set initial register values:

$$\begin{aligned} \text{MR} &\leftarrow \{b_{n-1},\ b_{n-2},\ \ldots,\ b_0\} \\ \text{AX} &\leftarrow \{a_{n-1},\ a_{n-2},\ \ldots,\ a_0\} \\ \text{AC} &\leftarrow 0 \end{aligned} \tag{2.21}$$

II) Multiply the multiplicand and one digit from multiplier

$$\text{AX} \wedge \text{MR}_0 \tag{2.22}$$

III) Sum product of step II and the accumulated sum, save it in accumulated sum and the extra bit for a carry

$$c_{\text{out}} \circ \text{AC} \leftarrow \text{AC} + (\text{AX} \wedge \text{MR}_0) \tag{2.23}$$

IV) Shift one right

$$\text{AC} \circ \text{MR} \leftarrow c_{\text{out}} \circ \{\text{AC}_{n-1},\ \text{AC}_{n-2},\ \ldots,\ \text{AC}_0\} \circ \{\text{MR}_{n-1},\ \text{MR}_{n-2},\ \ldots,\ \text{MR}_1\} \tag{2.24}$$

This procedure runs n times, facilitated by an assigned counter, until the values of MR have been replaced (see progression of MR in Table 2.4). However, if the operands are signed it would only take $n-1$ loops since the sign is not a part of the magnitude The sign bits would have to be handled

separately and differently; the sign bit of the product can be found by applying an XOR operation on the sign-bits. Furthermore, if the operands are represented in complement form, the method above requires slight modifications. One method is to convert the operands to *sign-magnitude form* before multiplication. The resulting value, if negative, should be converted back into complemented form. Another method, *Robertson's Signed number multiplication* makes use of *arithmetic shifts* during step IV (see Eq. (2.24)) to avoid the extra overhead from a pre- and post-complement circuit. [65]

The process of multipliation by *add-shift* is slow and scales with n, and has therefore been the subject of a lot of optimisation and a short list of some of the methods can be seen below:

- **Recoding Techniques:**
  - *Booth's Algorithm [66]:* Using multiple bits and performing additions/shifts based on the value of the subset of bits, multiplication of signed numbers can be performed with fewer operations.
  - *Non-Overlapped Multiple-Bit Scanning Multiplications [65]:* Using multiple bits from the multiplier for every loop, the number of loops required per multiplication can be reduced.
  - *Overlapped Multiple-Bit Scanning Multiplications [67]:* Using the *string*-property, a zero followed by a set of 1s can be turned into 1 followed by a set of zeros. This offsets the value by 1 LSB (with relation to the placement of the string), which can be accounted for later in the operation.
- **Parallelisation Techniques:**
  - *Wallace Trees [1]:* Produces the product of two numbers using only combinatorial logic, i.e. no intermediate registers for partial sums needed. Utilises a tree of ''pseudo-adders'' to reduce time scaling to logarithmic instead of linear.
  - *Baugh-Wooley Algorithm [68]:* Useful for multiplication of 2's complement numbers. The negative partial products are segregated and summed and all partial product bits are treated equally.
  - *Pezaris [69]:* Using appropriate circuits the partial products are calculated with their respective signs.

Addition in floating-point arithmetic: Given two nonzero binary floating-point numbers, $x = (-1)^{s_x} \cdot |x|$ and $y = (-1)^{s_y} \cdot |y|$, addition is based on the following identity [2]:

$$x + y = (-1)^{s_x} \cdot \left(|x| + (-1)^{s_z} \cdot |y|\right) \tag{2.25}$$

where:

s_x	The sign-bit of x, $s_x \in \{1,\ 0\}$
s_y	The sign-bit of y, $s_y \in \{1,\ 0\}$
$s_z = s_x \oplus s_y$	The XOR'ed sign-bits, $s_z \in \{1,\ 0\}$

From Eq. (2.16), $|x|$ and $|y|$ can be expanded:

$$|x| = r^{E_x} \cdot M_x \quad \Rightarrow \quad |x| = 2^{e_x - \text{bias}} \cdot M_x \tag{2.26}$$

$$|y| = r^{E_y} \cdot M_y \quad \Rightarrow \quad |y| = 2^{e_y - \text{bias}} \cdot M_y \tag{2.27}$$

The addition of these two values can be performed, and traditionally the addition follows the following steps [2]:

I) **Swap** x and y to ensure that $e_x \geq e_y$

II) **Align** mantissa by shifting M_y right by $e_x - e_y$, the first shift should push in a 1 from the left due to the implicit 1 in floating-point numbers [64]. The exponent of the result, e_r is set to e^x for now

III) **Compute** the resulting mantissa as $M_r = M_x + (-1)^{s_z} \cdot m_y \cdot 2^{-(e_x - e_y)}$. If M_r is negative, it is negated. The sign-bit of the resulting sum is chosen based on this calculation and the values of s_x and s_y

IV) **Normalise** the sum if it falls into one of the two cases:
- $(M_r \geq r)$, this will result in a carry-out. M_r must be shifted 1 right and e_r must be incremented
- $(M_r < 1)$, this will result in leading zeros. M_r must be shifted left until a 1 has been shifted out of the mantissa (restoring the implicit 1). Accordingly, the exponent e_r must be decremented once per shift

V) **Round** the normalised sum

Multiplication in floating-point arithmetic: Given two nonzero binary floating-point numbers, $x = (-1)^{s_x} \cdot |x|$ and $y = (-1)^{s_y} \cdot |y|$, multiplication is based on the following identity [2]:

$$x \cdot y = (-1)^{s_r} \left(|x| \cdot |y| \right) \tag{2.28}$$

where:

s_x	The sign-bit of x, $s_x \in \{1,\ 0\}$
s_y	The sign-bit of y, $s_y \in \{1,\ 0\}$
$s_r = s_x \oplus s_y$	The XOR'ed sign-bits, $s_z \in \{1,\ 0\}$

Likewise, $|x|$ and $|y|$ can be expanded as seen in Eq. (2.26) and (2.27). The product of the two positive finite floating-point numbers is given by [2]:

$$|x| \cdot |y| = 2^{E_x + E_y} \cdot M_x M_y \tag{2.29}$$

From Eq. (2.29) it is clear, that multiplication requires:

- **Addition** of the multiplier's and multiplicand's exponents
  - Can be viewed as fixed-point addition
- **Multiplication** of the multiplier's and multiplicand's mantissas
  - Can be viewed as fixed-point multiplication

The resulting number may require normalisation identical to step IV of *addition in floating-point arithmetic*.

2.2.3 The Multiply-Accumulate Unit (MAC)

A commonly used technique to speed up calculations is to combine addition and multiplication in one module: The MAC-unit (see Figure 2.11). With the knowledge of how *addition* and *multiplication* of FXP and FLP are done (separately), the MAC unit can be designed. Firstly, the format of the multiplicand, multiplier, and result is chosen. Based on the format, an adder can be chosen (e.g. choosing FXP gives CSA (carry-save adder) , CCSA, and CLA as possible choices, among others). Then a suitable multiplier can be chosen (e.g. in the case of FXP a booth multiplier or a Wallace Tree multiplier are possible choices, among others).

In an article called *Design and Performance Analysis of Multiply-Accumulate (MAC) Unit* a few models are investigated [70]:

(A) MRBM (Modified Radix-2 Booth Multiplier) + CSA (carry-save adder) + Accumulator
(B) Array Multiplier + CSA (carry-save adder) + Accumulator
(C) Ripple Carry Multiplier + CSA (carry-save adder) + Accumulator
(D) Wallace Tree Multiplier + CSA (carry-save adder) + Accumulator

(E) DADDA Multiplier + CSA (carry-save adder) + Accumulator

The five units are simulated at gate-level to compare the designs w.r.t. the following 3 metris: **Area** [LUT′s][1], **Delay** [ns], and **power** [W].

Table 2.5: Comparison of performance metrics of MAC unit models [70].

Model	Area [LUT′s]	Delay [ns]	Power [W]
A	137	6,712	1,010
B	97	8,175	1,067
C	92	6,712	1,261
D	96	6,102	1,061
E	103	3,6592	2,142

Note: Data taken from [70], however, ‘‘S.No’’ and ‘‘Design’’ columns have been replaced by the ‘‘Model’’ column.

The performance metrics in Table 2.5 display an important property of the design of a MAC unit; *a compromise between area, delay, and power must be made*. Model A has the largest area, however, the delay is relatively small and its power consumption is the smallest of the five. On the other hand, model C is equal to model A in delay, however, the area is smaller, and the price to pay is an increase in power consumption. This compromise is essential when designing modules/units that include addition and/or multiplication.

2.2.4 State-of-the-Art Arithmetic Units

As mentioned in subsection 2.2.3 a compromise must be made with area, delay, and power, which means that there are more than one state-of-the-art MAC units. Finding reputable sources for which architecture is used for addition and multiplication in newer CPUs and GPUs is difficult (if not impossible) and in the technical blog on NVIDIAs own website, it was mentioned in 2022 that the latest NVIDIA Hopper GPU architecture uses AI-designed circuits [72]. From the available information, it is not possible to look at the *state-of-the-art* GPU/CPU architectures, however, looking at benchmarking tools instead, yields different results. ‘‘AxBench’’ is a *multi-platform benchmark suite for approximate computing* and in [73] the tools benchmarks are listed. In their list of benchmarks, three are under the domain *arithmetic computation*:

- Brent-Kung (FXP parallel prefix form of a CLA)
- Kogge-Stone (FXP parallel prefix form of a CLA)
- Wallace-Tree (FXP multiplier)

These three *benchmarks* are explained in the following paragraphs.

Brent-Kung Carry Chain Prefix

The *Brent-Kung Carry Chain Prefix* reformulates the full-adder design from serial to parallel. For the full-adder, the carry-out and sum can be calculated as [74]:

[1] A LUT (Look-up Table) is a configurable logic-block that is used in the most popular FPGAs [71].

$$
\begin{aligned}
c_{-1} &= 0 \\
c_i &= (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1}) \\
s_i &= a_i \oplus b_i \oplus c_{i-1}, \qquad i = 0, \ldots, n-1, \\
s_n &= c_{n-1}
\end{aligned}
\tag{2.30}
$$

where:

c_i	The carry-out at bit i
a_i	The ith bit of one summand
b_i	The ith bit of the other summand
s_i	The sum at bit i
n	The number of bits

The calculation of the carry-bit can be rewritten [74]:

$$
\begin{aligned}
g_i &= a_i \wedge b_i \\
p_i &= a_i \oplus b_i \\
c_i &= g_i \vee (p_i \wedge c_{i-1})
\end{aligned}
\tag{2.31}
$$

where:

g_i	The *carry generate*
p_i	The *carry propagate*

The reformulation of the carry chain computation is as follows; Brent and Kung define an operator "o"[2] to be:

$$
(g,\ p) o (g',\ p') = (g \vee (p \wedge g'),\ p \wedge p')
\tag{2.32}
$$

Brent and Kung's article provides and proves the following two lemmas [74]:

Lemma 2.2.1. Let

$$
(G_i,\ P_i) = \begin{cases} (g_i,\ p_i) & \text{if } i = 0 \\ (g_i,\ p_i) o (G_{i-1},\ P_{i-1}) & \text{if } 1 \le i \le n-1 \end{cases}
\tag{2.33}
$$

Then

$$
c_i = G_i \qquad \text{for } i = 0,\ 2,\ \ldots,\ n-1
\tag{2.34}
$$

Lemma 2.2.2. The operator "o" is associative

Given Lemma 2.2.1 and Lemma 2.2.2 all c_is can be computed in any order given all g_is and p_is. To that extent two nodes are defined in Figure 2.19:

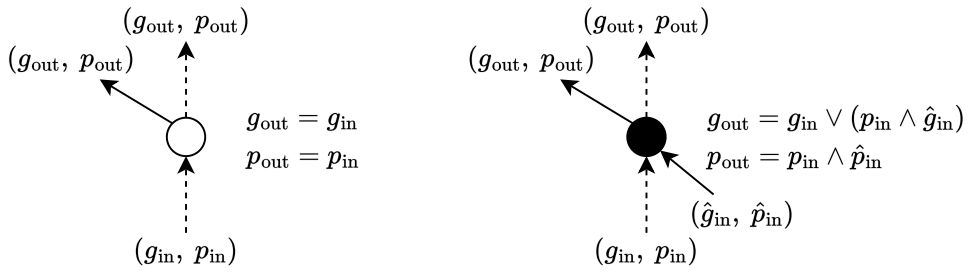

Figure 2.19: Pass-through node on the left and a computational node on the right (Recreation of Fig. 4 from [74]).

[2] Be aware that o is similar to $\circ$, the concatenation operator, however, they are different operations.

Using a lattice-like structure of the two nodes from Example 2.2.3 all carry-bits can be calculated at the same time in parallel. In Example 2.2.3 the carry-bits for a summation of two 8-bit numbers are calculated.

Example 2.2.3: Brent-Kung Parallel Prefix

Using the two nodes presented in Figure 2.19 the tree structures below can be created. On the lefthand side of Figure 2.20 the nodes required for calculating the 7th carry bit are highlighted. Although information about all bits 0, 1, ..., 6 is needed to calculate bit 7, this parallel method has a shorter critical path w.r.t. the carry propagation than an adder consisting of only full-adders (as seen in Figure 2.15).

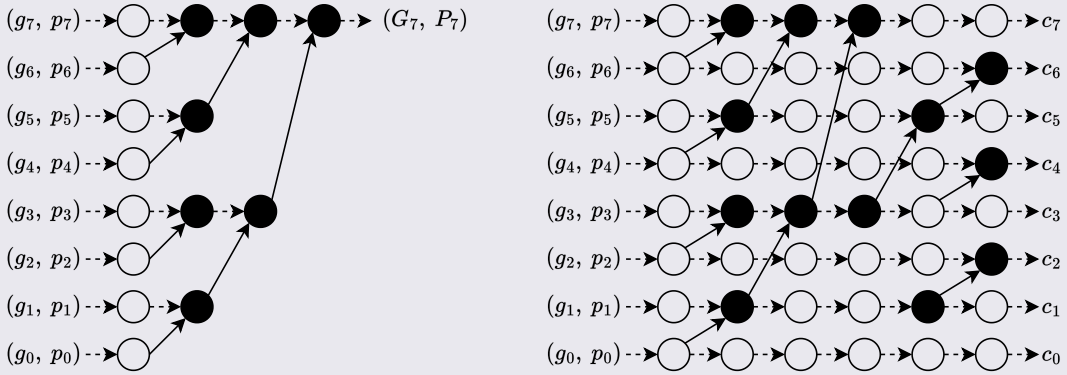

Figure 2.20: Brent-Kung adder carry tree. The lefthand side displays the nodes required to calculate the carry-out of the MSB, and the righthand side displays the full structure required to calculate all the carry-bits (Recreation with slight modifications of Fig. 3 and Fig. 5 from [74]).

The output of the lattice structure is all the carry-bits available at the same time.

To finalise the design of the adder using the Brent-Kung carry chain, the sum bits should still be calculated (using the g_{out}s and p_{out}s). However, all carry bits are available and the critical path is thus shortened. Assuming that $\vee$, $\wedge$, and $\oplus$ take unit time, the speedup of using the carry chain can be viewed in Table 2.6. For $n = 2^k$ the parallel and serial times can be generalised to $4k$ and $2n-1$, respectively [74].

Table 2.6: Comparison of parallel and serial times (Data for the time columns taken from Table 1 in [74]).

n	k	Time (parallel)	Time (serial)	Speedup [%]
8	3	12	15	25 %
16	4	16	31	~ 94 %
32	5	20	63	215 %
64	6	24	127	~ 429 %

Utilising the Brent-Kung adder significantly increases the speed. And the amount of computational nodes can be calculated as [75]:

$$\text{Comp. Nodes}_{\text{Brent-Kung}} = 2(n-1) - \log_2(n) \tag{2.35}$$

Kogge-Stone Adder

Another usage of the two nodes in Figure 2.19 is in the Kogge-Stone adder [76]. Revisiting Lemma 2.2.1 it is clear, that as long as there is a path from the 0th bit to the ith bit through the computational nodes (see Figure 2.19), $(G_i,\ P_i)$ is solved. The Kogge-Stone adder prefix takes advantage of this, and creates a dense network of nodes.

Example 2.2.4: Kogge-Stone Parallel Prefix

Using the two nodes presented in Figure 2.19 the tree structures below can be created. The lefthand side of Figure 2.21 displays the nodes and connection necessary for processing $(G_7,\ P_7)$ and thereby the carry-out of the MSB. The lefthand side is identical to that of the Brent-Kung prefix displayed in Figure 2.20, however, the difference between the two parallel prefixes can be seen on the right hand side. The Kogge-Stone prefix requires fewer layers at the cost of an increase to the amount of computational nodes; the result of which is a faster parallel prefix that uses more energy. Indeed, doubling the amount of bits would only increase the depth by 1 layer.

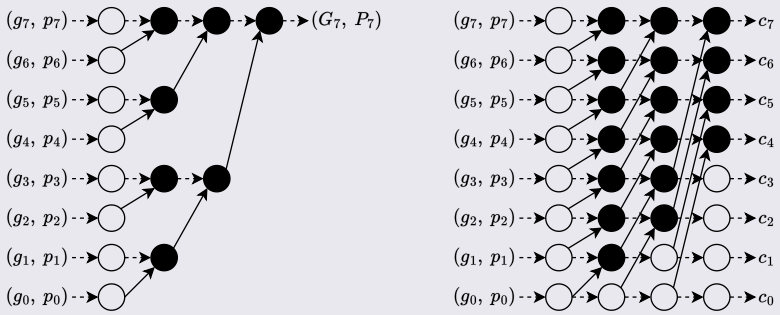

Figure 2.21: Kogge-Stone adder carry tree. The lefthand side displays the nodes required to calculate the carry-out of the MSB, and the righthand side displays the full structure required to calculate all the carry-bits. Unlike the Brent-Kung adder prefix, no carry-out requires more layers than for the MSB.

The output of the Kogge-Stone parallel prefix is also all the carry-bits and not the sum of the two input numbers.

Like for the Brent-Kung parallel prefix, the sum must still be calculated. The circuitry required for calculating the sums can be simplified. Making the same assumptions w.r.t. unit time operations another table for speedup can be created (see Table 2.7).

Table 2.7: Comparison of parallel and serial times. Data for the ''Time (serial)''-column taken from Table 1 in [74] and data for the ''Time (parallel)'' is calculated by assuming each layer adds 2 unit time (this assumption is based on the longest path in the computational node being a 2, `AND` followed by `OR`, and that the time of a layerwise-operation is decided by the longest computation of that layer).

n	k	Time (parallel)	Time (serial)	Speedup [%]
8	3	8	15	~ 88 %
16	4	10	31	210 %
32	5	12	63	425 %
64	6	14	127	~ 807 %

Utilising the Kogge-Stone parallel prefix significantly increases the speed. And the amount of computational nodes can be calculated as [75]:

$$\text{Comp. Nodes}_{\text{Kogge-Stone}} = n\left(\log_2(n) - 1\right) + 1 \tag{2.36}$$

Wallace-Tree Multiplier

The *Wallace tree multiplier* is a combinatorial alternative to the sequential multiplier presented in Figure 2.18 [1]. The sequential multiplier generates a summand from the product of 1 bit of the multiplier and all bits of the multiplicand and iteratively accumulates the summands, finally giving the product of the multiplier and the multiplicand. This sequentiality costs time since the sums have to be saved to a register. The proposed alternative uses a combination of *pseudo-adders*, which instead of producing a single sum, produces two partial sums that collectively add up to the single sum [1]. However, this requires that the summands are generated beforehand, and if they are not generated in parallel, nothing is gained from this approach. There are multiple ways to generate the summand, but the simplest (in terms of understanding) is the same way it was done in the sequential multiplier: one bit from the multiplier `AND`ed with the multiplicand, however, done in parallel.

Example 2.2.5: 3:2 Pseudo-Adder Using Full-Adders

An example of a pseudo-adder from [1] is a set of full-adders where the carry-in bits are used for the third input number and the carry-out bits are used as the second output. In Figure 2.22 a 3:2 pseudo-adder using this method is visualised. The three 4-bit values are piped to the full-adders: The LSBs are added in a full-adder, the second LSBs are added in the next, etc. The outputs of the 3:2 pseudo adder are two 4-bit values comprised of the carries and the sums, respectively. It is important to note, that the value of the carry-bits is double that of the sums, and an implicit 0 is added for disambiguity. On the righthand side of Figure 2.22, the sum of the inputs is calculated, and the sum of the outputs is calculated; the values are equal and thus no information was lost.

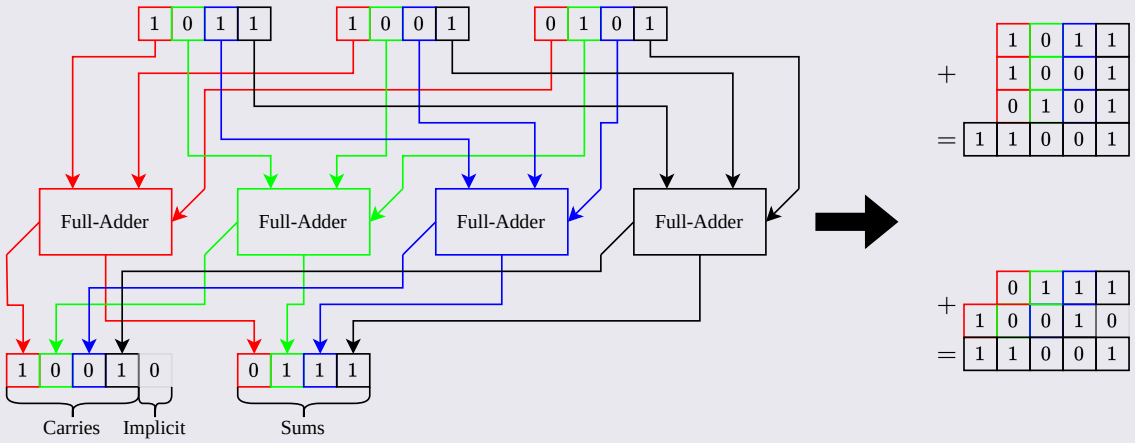

Figure 2.22: Example of a 3:2 pseudo-adder consisting of 4 full-adders. The lefthand ''circuit'' utilises four full-adders two reduce the sum of three values to two values. The righthand side shows that the sum of the inputs is equal to the sum of the outputs.

The pseudo-adder presented in this example is one of many designs. 4:2 pseudo-adders are also relevant in this context.

Example 2.2.6: Wallace Tree Multiplication

An example of the Wallace tree can be seen in Figure 2.23 where 20 summands (W_1, W_3, …, W_{39}) are inputs to a tree of pseudo-adders. The last step requires a regular adder and will produce the *final product*.

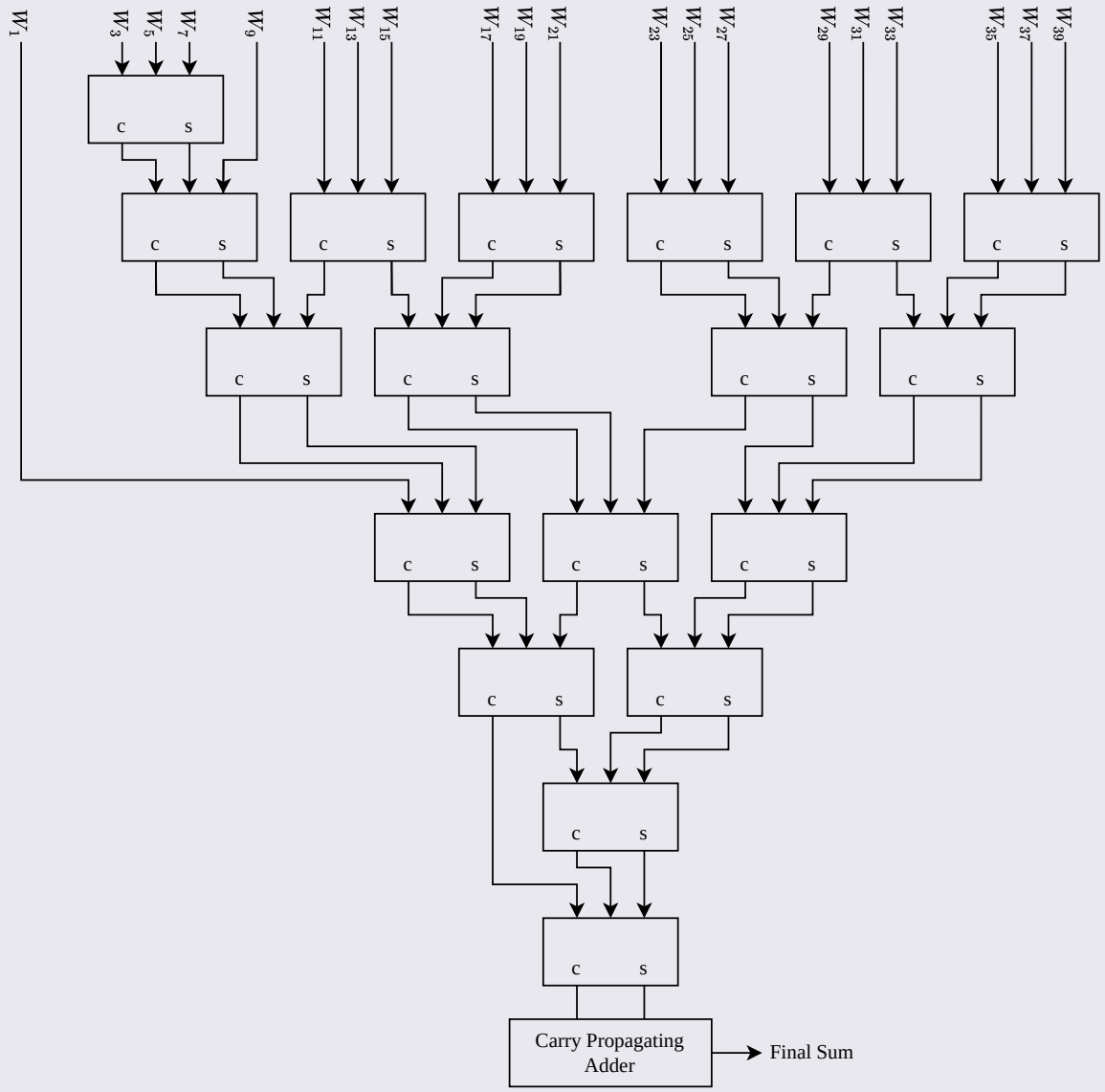

Figure 2.23: Example Wallace tree for 20 summands (Recreation of Fig. 1 from [1]).

From Example 2.2.6 it is clear, that the delay and gatecount are dependent on the choice of pseudo-adders, the carry propagating adder, how many summands are needed, and how the summands are generated. Instead of having a formula to estimate the delay and gatecount, some examples are presented in Table 2.8:

Table 2.8: Examples of Wallace tree delay and gate count (information taken from [77]).

Multiplier	Delay [ns]	Full Adders [·]	Half Adders [·]	Total Gate Count [·]
(16 × 16) Conventional WT	14,252	223	56	701
(16 × 16) Hybrid WT	15,510	221	47	675
(32 × 32) Conventional WT	21,5519	960	191	2849
(32 × 32) Hybrid WT	22,165	956	76	2750

Floating Point Arithmetics

As previously mentioned, addition and multiplication are the main building blocks of a MAC-unit. MAC-units are an integral part of the Graphical Processing Unit and in state-of-the-art NVIDIA GPUs the number representation is floating point [78]. Furthermore, in *Floating Point and IEEE 754 Compliance for NVIDIA GPUs* it is mentioned, that they utilise the faster and more accurate fused multiply-add instead of separate multiply and add operations (can be disabled) [78]. However, the specific architecture of the FMA in the NVIDIA GPUs is seemingly not available. Instead, the ''classic architecture'' of an FMA is researched and summarised here.

In Figure 2.24 the architecture of the first widely available FMA [2] can be viewed. Note the inputs at the top of the figure, which are split into 3 groups: sign-bits s, exponents E, and mantissa m. Two of the mantissas of the operands are piped into a *partial product generation*, through a *compression tree*, the outcome of which are two numbers for piped into an adder. This process is almost identical to the **parallel multiplication in FXP** presented in Example 2.2.6; partial products are generated and piped into a tree-structure, whereby the partial products are ''reduced'' to two summands, which are piped into an adder.

The last mantissa is processed in the *invert* block, wherein its value is inverted based on n_c, the ''is normal''-bit and if it is subtraction. Afterward, the mantissa is shifted before it is also piped into the same **adder** as the two multiplied mantissas. The *3:2 carry-save adder* is a **pseudo adder** (two outputs, that add up to the sum), wherefrom the output is added in a *fast adder* and complemented if necessary. The adders are **FXP adders** and could be replaced with something using Brent-Kung or Kogge-Stone prefix. Lastly the mantissa and exponents are normalised (see step IV of **addition in floating-point arithmetic** in section 2.2.2). Some of the blocks and connections have been left out of this description, however, they are explained in the source material [2, pp. 303-304].

Floating-point multiplication requires small portions of fixed-point addition and fixed-point multiplication.

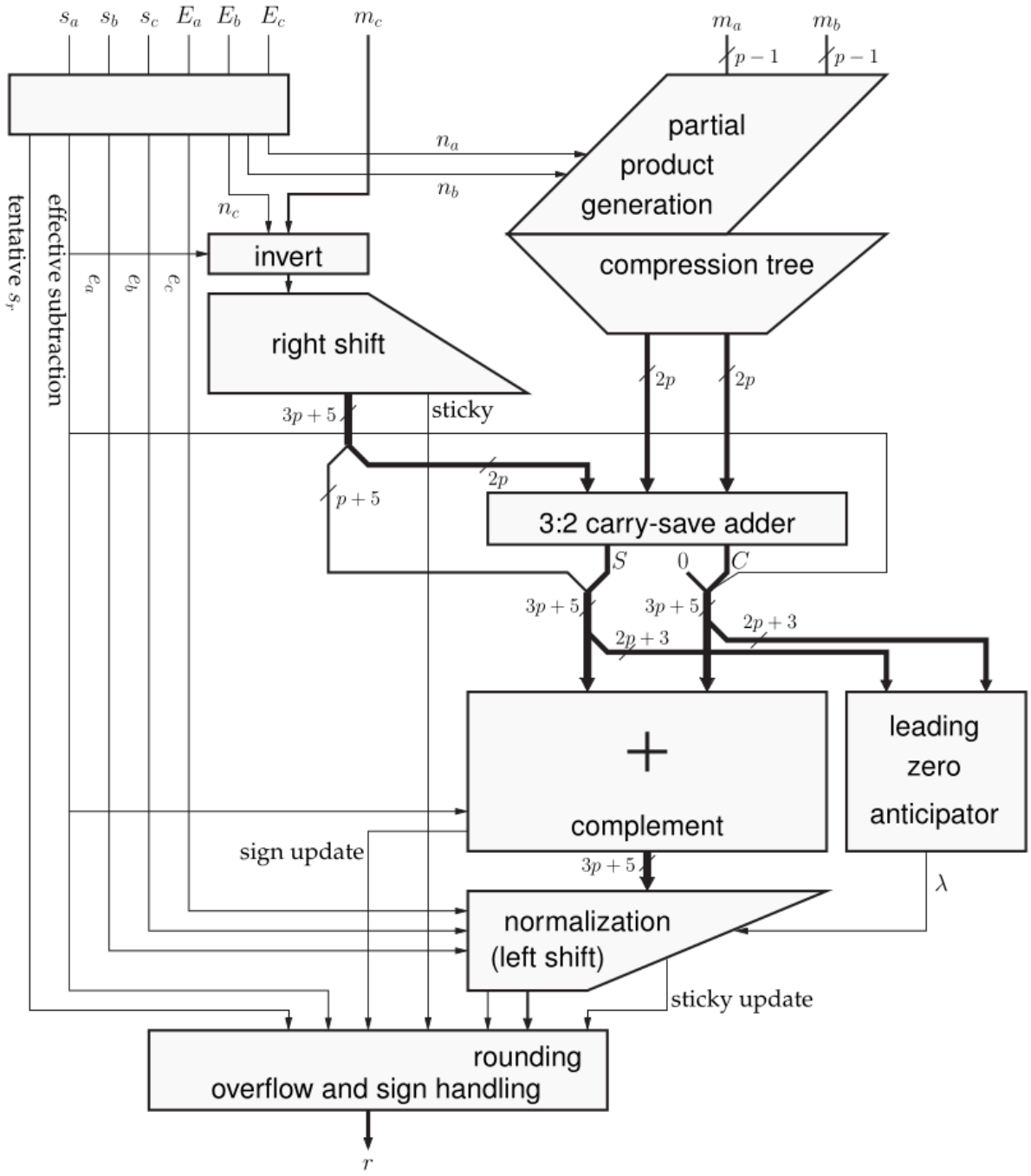

Figure 2.24: The classical single-path FMA architecture from the IBM RS/6000 [2].

2.3 Approximate Computing

As described in chapter 1 the rapid growth of data generated in computing systems and academic respects, there is a pressing demand for computational ability [22].

AC (Approximate Computing) is an umbrella term encapsulating any concept that simplifies system computations at the expense of the accuraty of the calculations. Generally, the field is distinguished into two domains; HW (Hardware) and SW (Software) [22][27]. Each domain possesses an extensive catalogue of subdomains and techniques that are of varying relevance to the CNN application [22][79]. The following sections will focus solely on the AC techniques applicable to CNNs. A survey by Leon et. al. split in two parts, [22] and [27], presents a variety of methods and techniques within both SW and HW domains. [22] offsets in a general-purpose review of AC and will for this project be used as a guide to relevant literature. [27] presents (among other things) techniques specifically viable in NN applications. The following sections are based on [27] and the literature presented in the survey.

2.3.1 Approximate Software

The SW domain is explored first and is generally divided into four categories that apply to NN; pruning, precision scaling, early determination, and input dimensionality reduction.

Pruning is a technique that deliberately exploits computation skipping to lower the necessary memory and computational load of the trained NN model. The principle behind pruning is to remove synaptic weights, biases, or entire perceptrons deemed unnecessary or irrelevant. Relevance in this context could refer to low activation of the perceptron or synaptic weights close to zero [80][81][82][83]. Pruning is further distinguished into structured and unstructured pruning.

Unstructured pruning refers to techniques that remove individual synaptic weights, with the simplest technique being an assessment of the weight's magnitude and any weight below a threshold is removed. This technique is known as **fine-grain pruning**. Unstructured pruning may lead to irregular memory access patterns, which could impact computational efficiency and might disallow simple reconfiguration of the NN [82]. However, the unstructured pruning schemes usually provide a greater **compression ratio** [27].

Structured pruning refers to strategies where entire kernels (i.e. perceptrons) or filters (i.e. collections of kernels) are removed based on activation. In opposition to the unstructured counterpart, structured pruning preserves the overall structure of the layers in a CNN, eventually limiting the compression ratio [81][82].

Precision Scaling has been thoroughly introduced from a conceptual perspective in subsection 2.2.1, and refers to the approximations introduced by representing numbers in low precision FXP rather than the 32-bit FLP IEEE-standard [84].

A straightforward approach to precision scaling in a NN application is the PTQ (Post-Training Quantisation) , in which a CNN is trained on general-purpose CPU (Central Processing Unit) s or Graphical Processing Units using a standard FLP quantisation. The trained model is then re-optimised using a quantised number format (e.g. FXP number representation) before deployment [80][85].

Several efforts have been devised to use QAT (Quantisation-Aware Training) strategies which address the apparent loss of information due to precision scaling. As the name suggests, QAT is incorporating the effects of the quantisation in the backpropagation algorithm, allowing the

optimiser to *learn* these relations [80]. It has been shown that the loss of accuracy (due to the quantisation errors) in the inference of a NN can be regained by using QAT [86][87].

Precision scaling is most effective when deployed on hardware that supports quantised operations. Some hardware accelerators are designed to work optimally with quantised models, further improving efficiency during inference [27].

Early Determination is an approach that aspires to decide whether it is necessary to finish a comprehensive computation, or if an intermediate result will suffice. The aim is to relax requirements for computation, by initially estimating the result's significance [88]. A sign-predictor is a specific example deeply related to the CNN application. The principle is to estimate if the output of a MAC-operation is negative. If the RELU function is used as activation; negative values will return 0 and therefore a complete MAC-operation is a waste of computation.

Input Dimensionality Reduction reduces the number of input features with an effort to preserve the most characteristic ones. The efficiency gain is that a smaller network is needed to model the input/output relationship [6].

2.3.2 Approximate Hardware

The HW domain relies on circuit-level approximations for a reduction in energy consumption or circuit latency. An early stage deep CNN such as AlexNet has 650,000 perceptrons and 60 million tunable parameters, hence at least that amount of MAC-operations is necessary for each forward-pass in the network. Furthermore, the AlexNet was trained on the ImageNet dataset containing 3.2 million images [52][51], resulting in an unfathomable amount of computations in the training of the AlexNet or state-of-the-artNNs in general. Significant energy reduction can therefore be obtained by introducing AC to the MAC units. Two approaches are presented in this section namely VOS (Voltage Over Scaling) and approximate arithmetic circuits.

Voltage Over Scaling is a specific technique that relies on operating digital systems, with a voltage below the critical voltage where the critical path delay is met for a given circuit. Since the consumed power has a quadratic relationship with the voltage of a circuit VOS is assuring a reduction in energy over time [89][90]. The VOS introduces timing errors as the charging time of transistors in the circuit increases as the voltage decreases, requiring a lower clock frequency [90].

An approach similar to VOS is *overclocking*. Opposite to VOS overclocking is the concept of operating a circuit with a higher clock frequency to enhance the latency performance, trading off a higher power consumption [89].

Modern processors and GPUs are utilising the concept of DVFS (Dynamic Voltage and Frequency Scaling) , which is a technique that scales voltage and clock frequency dynamically based on intermediate requirements to either power consumption or throughput. DVFS operates between the critical path delay and maximum power requirements, where VOS and overclocking introduces errors by exceeding these thresholds [89][90].

Approximate Arithmetic Circuits are circuits designed to perform arithmetic operations such as additions or multiplications. The approximate circuits differ from the adders and multipliers presented in section 2.2, in that circuits are designed using fewer logic gates to deduce the output of an arithmetic operation [21][23]. The methodology for developing such circuits has historically been rather *ad hoc* and application-specific, as arithmetic designs can be fine-tuned to reduce area or latency based on the problem or application at hand [91][92].

Approximate arithmetic circuit design includes various strategies of differing analytical properties. The main focus is on the fundamental building blocks of digital processors (i.e. adders, multipliers etc.). To approximate the arithmetic units only the requirements for the magnitude of errors limit the possibilities in design. The point to notice is that any outcome approximates any arithmetic operation, with a varying degree of accuracy. To illustrate this fact three open-source libraries of approximate adders and multipliers are provided by Shafique et. al. [93], Ullah et. al. [94], and Mrazek et. al. [4]. [93] presents a configurable strategy to synthesising approximate adders. [94] provides a library of approximate multipliers with varying degrees of accuracy and circuit complexity. [4] presents a library of 430 approximate adders and 471 approximate multipliers. This also serves as an illustration of the point that anything can approximate an arithmetic circuit. However, an effort has been made to categorise the design of approximate arithmetic units.

Approximate adders generally be divided into two approaches:

I) **Approximate Full-Adders:** The technique attempts to approximate the fundamental building blocks of an adder circuit (i.e. the `full-adders`) [22][21]. As described in subsection 2.2.2 the `full-adders` are 1-bit adders, used as building blocks in broader addition architectures (e.g. RCA or CLA). The approximation of `full-adders` is realised by configuring the logic-gate circuits using fewer transistors. Yang et. al. [95] present three designs of an XOR/XNOR-based AFA (Approximate Full-Adder) s reducing the transistor count relative to the accurate equivalent by up to 40 %. Gupta et. al. [96] presents approximate architectures of the *mirror-adder* reducing transistor count of a `mirror-adder` by up to 52 %

II) **Segmentation and Carry Prediction:** These are strategies that aim to reduce transistor count and/or critical path delay by dividing the architecture of an n-bit adder into smaller segments that can be evaluated in parallel and approximated individually [22]. This approach differs from the AFA as the configuration of `full-adders` are approximated rather than the `full-adders` themselves. Mahdiani et. al. [97] presented an architecture known as the LOA (Lower-part-OR-Adder) . The principle of the LOA is to divide an p-bit adder into two segments; an m-bit accurate adder (e.g. RCA or CLA) for the m MSBs and an n-bit segment of LSBs where the bit-wise addition is approximated as OR-gates. Dalloo et. al. systematises a LOA design in [98]. The segmented approaches also allow parallel computing of each segment, lowering the critical path delay [99]. In [100] The authors present an ACA (Accuracy Configurable Adder) which uses approximate sub-adders as was the case for the LOA but implements an error-correction scheme, which can be configured throughout the pipeline of the calculations in a given system. When accurate results are necessary the error correction is applied to segments handling the MSBs and when an accurate result is not critical, the error correction is applied to fewer segments (if any) of the MSBs. [101] builds upon the ACA and utilise carry-prediction techniques to anticipate carry bits before they are fully propagated through segments of the adder. These adders can reduce the critical path delay by predicting carries early in the computation in each segment.

Approximate multipliers presented in the literature are primarily based on the Wallace tree from subsection 2.2.4 and can be generalised into three main approaches:

I) **Truncation, Rounding, and Perforation:** The Wallace tree uses a tree of *pseudo-adders* to accumulate the summands generated using the long multiplication scheme presented in Example 2.2.2. Approximations can be introduced by truncating, rounding, or perforating the summands before accumulation which yields pseudo-adders with simpler circuits, as the inputs are of shorter bit-length [21]. Hashemi et. al. [102] presents a truncation scheme known as the DRUM (Dynamic Range Unbiased Multiplier) . The DRUM first detects the most significant 1 in each multiplicand and discards all more significant 0s. Next, a fixed bit-width is selected and the bits of less significance than this are truncated into a 1 as the LSB of the

fixed bit-width and every bit of less significance is discarded. The length of the fixed bit-width determines the precision of the approximation. The multiplication is then performed on the unbiased truncated multiplicands and the result is shifted into the appropriate bit-width (i.e. the length if no truncation was performed). The length of the bit-width bounds the errors of this approach and the distribution has a Gaussian distribution while obtaining 58% power savings [102]. Zervakis et. al. [103] present a perforation technique where the generation of summands is limited to a selection of MSBs where the developer chooses the exact amount. The amount of discarded LSBs yields an equivalent reduction in the number of summands in the Wallace tree.

II) **Approximate Pseudo-Adders:** An alternative approach to the approximate summands is to approximate the accumulation of summands (i.e. the pseudo-adders in the Wallace tree). Four approaches with varying degrees of approximation are presented in [104], all focusing on applying approximations to the 4:2 compressor (i.e. a realisation of a pseudo-adder implemented using two full-adders). The specific design presented in [104] provides the ability to switch between an exact implementation and the proposed designs dynamically. Two of the approximate 4:2 compressors are implemented as simple wirings (i.e. no logic gates in the compressors). Strollo et. al. [105] surveys proposed 4:2 compressors from the literature and compares the performances for developers to choose from. In the context of approximate pseudo adders, developers are once again left with a rather ad hoc approach.

III) **Logarithmic Binary Multiplication:** Mitchell's logarithmic binary multiplier was presented in 1962 by John N. Mitchell [106]. The principle is to encode the multiplicands as LN (Logarithmic Number) s, to yield a simple multiplier implementation. The conversion to LNs is an approximate operation, resulting in inaccuracies in the final result. The gain in computing efficiency is that the logarithm of multiplication is an addition, meaning that the far more complex multiplier HW is substituted with the much simpler adder HW. Liu et. al. [107] extend the concept of approximate logarithmic multipliers, by using approximate adders in the design. One of the approaches uses the LOA that was described earlier.

The antecedent survey of approximate adders and multipliers is not exhaustive of the approximate arithmetic circuits proposed in the literature, nor the possible design space. However, a brief introduction to multiple categories within the field, along with relevant references to the literature should provide a foundation for developers to explore approximate designs further.

Automated approaches for approximate circuit design have also been proposed in the form of approximate logic synthesis. This is the concept of approximating the boolean function in the synthesis of a logic circuit. Two approaches are considered for approximate synthesis:

I) **Structural Netlist Transformation:** A netlist is a structural representation of a digital circuit specifying the interconnections between logic gates and different components. The netlist realises a boolean function (described by a truth table) and by making simplifications to the netlist, approximations are introduced to the boolean function [23]. Schlachter et. al. [108] presents a simple algorithm for pruning netlists. The general idea is to remove one gate at a time and simulate until a certain error threshold is reached in the boolean functions truth table. A similar approach is presented in [109], known as *Circuit Carving*. Rather than the iterative technique just described, the circuit carving algorithm exhausts the entire design space finding the smallest possible circuit (i.e. lowest gate count) that does not cross the design-specified error threshold. This is done by assigning a weight to each operation in a netlist and using a binary tree to locate the minimum [109]. Both the iterative and circuit carving methods have a neat property of not being application invariant, as no prior information about the functionality is needed.

II) **Cartesian Genetic Programming:** This approach utilises genetic algorithms and graph-based

representation of accurate circuits to derive approximate solutions. The concept was presented by Hrbacek et. al. in [110]. The design space is defined in a graph representation which is a grid of nodes and connections. The nodes represent logic circuits and the connections represent wires. Evolutionary algorithms are then applied to reach approximate circuits by *''mutating''* the accurate circuit. The authors of [110] used their proposed algorithm to develop the EvoApproxLib described earlier [4].

It is noticed that the field of possible approximate computing design is wide-reaching and can be difficult to navigate in the approximate circuit's impact on the CNN. Efforts have been made to find practices for determining approximate solutions to CNN-based problems with user-defined constraints. Tools for benchmarking approximate computing techniques in NNs have already been developed, three of which are presented in the list below:

- **TypeCNN [28]:** A software framework for the development of (approximate) CNNs. With TypeCNN the user can modify the datatypes and multipliers of a CNN and infer the impact on the classification accuracy. In the paper *TypeCNN: CNN Development Framework With Flexible Data Types*, the tool is presented and evaluated specifically on altering data types (on either weights or the data); the accuracy of the CNN on MNIST in three cases are placed in tables: 10 epochs in floats and conversion to FXP, 10 epochs in floats and fine-tuning for 5 epochs in FXP and 10 epochs with FXP used for inference and weights and the backward propagation in floats.
- **AxDNN [29]:** A systematical framework towards the cross-layer design of approximation DNNs with a pre-RTL simulation environment to evaluate the power consumption. As per the authors of the paper *AxDNN: Towards The Cross-layer Design of Approximate DNNs* their contributions are threefold: Exploration of approximation techniques presenting an analysis of accuracy energy trade-offs offered by the techniques, the pre-RTL simulation environment for accurate power performance evaluation to figure out the accuracy energy trade-off at the design stage, and the detailed experiments for validation of the effectiveness of the pre-RTL simulator.
- **ProxSim [30]:** A fast simulation framework for approximate hardware supporting approximate DNN inference and retraining. The main contributions of the paper *ProxSim: GPU-based Simulation Framework for Cross-Layer Approximate DNN Optimization* are as follows: The simulation framework itself, accelerated simulation of approximate hardware in DNN computation, formulation of a novel hardware-aware regularisation for retraining approximate DNNs, and exploration and analysis of several optimisation and retraining techniques for approximate DNNs. The retraining techniques include a so-called STE (Straight-Through Estimator) and a proposal for an approximation-aware backpropagation. The STE is a technique where the loss-function evaluation is obtained through forward-passing in an approximate circuit and backward-passing and optimising using the accurate equivalent of the circuit.

2.4 Summary of the Survey

In 2.1 Neural Networks the *perceptron* has been presented as the basic building block of NN (Neural Networks)s. The perceptron is connected to a set of inputs and each input is associated with a *weight*. Internally, the inputs are weighted and summed, possibly with a *bias*, and the sum is processed through an *activation function*, which introduces *non-linearity*. The activation function enables the perceptron to classify non-linearly separable sets and perform non-linear regression. A lattice structure of *perceptrons* (like the one presented in Example 2.1.1) constitute a *neural network*. In order to set the *weights* and *bias* the concept of *training* has been presented: A *loss function* is defined, which is used to calculate how ''far'' the output of the neural network is from the wanted output. Based on the loss *error back-propagation* is utilised to find the *gradient* of from each weight to the loss, i.e. the contribution of the weight on the evaluated loss. The gradients are used to *update the weights and biases* using some *gradient descent algorithm* (optimisation algorithm). CNN (Convolutional Neural Network)s are the product of a *state-of-the-art* architectural approach, that addresses a problem with densely connected neural networks: *recognition of features in the input is not invariant of the position in the signal*. This is done by *parameter-sharing to local connectivity*; local convolution of the input signal with a kernel filter. Unlike a single perceptron, the outputs of a convolutional layer are *feature maps*. The size of the feature maps can be reduced by *pooling/subsampling*, where another form of kernel filter is applied, e.g. max pooling.

The efficacy of utilising CNNs on a task like *image classification* on the CIFAR-100 dataset is reflected in Table 2.2 where it is shown, that in the period between 2013 and 2024, the best-performing networks utilised *convolutional layers*.

In 2.2 Digital design the inner workings of the MAC (multiply-accumulate) unit were investigated. The MAC unit is an essential block to facilitate the arithmetic operations required in an NN. The MAC unit consists of a *multiplication*-block, an *accumulation*-block, and a *register*-block. The register stores the output of the accumulation and is thus a ''memory''-block and has not been explored further. However, multiplication and addition require an understanding of the *number representation* of computer systems. Unsigned/signed fixed point and floating point have been presented; unsigned/signed FXP is essentially the *integer* interpretation of a sequence of bits scaled by a factor 2^{-b}, where the most significant bit is negative in the case of *signed* fixed-point. The floating-point number representation is divided into three parts: sign-bit, exponent, and mantissa. The sign-bit indicates if the decimal interpretation is positive or negative, the mantissa represents a number between 1 and 2, and the exponent scales the value of the mantissa. The resolution of FXP is static, i.e. the LSB (least significant bit) defines the step size in the entire range of representable values. The resolution of FLP is dynamic since the product of the exponent and the LSB of the mantissa dictates the step size; for small values of the exponents, the step size is also small, and vice versa. Next, the principles for performing addition and multiplication with the different number representations have been presented. The most basic (and exact) addition in FXP is utilising a sequence of *full-adders* where the carry-out bits are sent to the next full-adder. This approach is slow, due to the ''critical path'' where the carries are *rippled* from the LSB to the MSB (most significant bit) and alternatives that with a shorter ''critical path'' and larger power consumption have been presented. *Multiplication* in FXP is a matter of generating partial products and summing them (not unlike long multiplication). This process can be *parallelised* and the partial products can be produced at the same time, which shortens the latency and removes the dependency on registers, i.e. the multiplications can be performed by *combinatorial logic*. FXP addition and multiplication are more involved, since alignment, normalisation, and rounding have to be performed. However, the actual *computations* are performed similarly: During addition the mantissas are summed as two FXP values, and during multiplication the exponents are added as two FXP values, and the mantissas are multiplied as two FXP values.

State-of-the-art MAC units are the product of a compromise between *power consumption* and *latency*; lowering one increases the other.

In 2.3 Approximate computing the term *approximate computing* has been presented; an umbrella term encapsulating any concept that simplifies system computations at the expense of the accuracy of the calculations. Two branches of this development strategy have been presented: *approximate software* and *approximate hardware*, with a focus on their relevance to NNs. Through **software** it is possible to *prune* ''unnecessary'' computations, by removing elements with small significance. The number representation can be revised and the precision can be scaled, e.g. instead of using `float32` the values/weights may be represented by 16 bits in FXP, simplifying arithmetics and reducing required memory. Using *early determination* an intermediate result may be utilised to avoid fully performing an operation. ''Simplifying'' input by *reducing the dimensionality* may also simplify/reduce the required operations. Through **hardware** the circuit-level designs may be approximated for a reduction in energy consumption or circuit latency. *Voltage over scaling* can be utilised, where the supply voltage can be reduced at the cost of a lower clock-rate. *Approximate arithmetic circuits* utilises fewer gates to produce an output, where some error is allowed. Many methods exist that guide/perform the development of the approximate circuits like *approximating full-adders*, *truncation*, *rounding*, *perforation*, etc.

Three tools for benchmarking approximate computing techniques have been briefly mentioned in section 2.3: TypeCNN, AxDNN, and ProxSim. These three approaches serve as a beneficial commencement for developing efficient CNNs, however, it is clear that the tools are mostly aimed at the *machine learning* developer demographic. The tools do not solve the ad hoc problems that are present when designing the approximate circuits for implementation. In this thesis, a contribution towards an easy validation process of approximate circuits will be made. Rather than avoid the ad hoc problems, the solution will lean into it and allow relatively fast validation of an ad hoc circuit. However, the knowledge gathered in the papers will be taken into account during the design and implementation of the solution. In the following chapter, the solution will be presented on a functional level.

A Benchmarking Tool for Approximate Arithmetics 3

The preceding chapter has presented fundamental concepts as well as state-of-the-art methods for NNs and the principal arithmetic building blocks of such systems. The purpose is to provide a basic foundation for a brief survey of AC methods relevant to the field of NN design. The vast amount of MAC-operations performed in state-of-the-art models are the main issue regarding computational complexity and energy consumption, driving the need for efficient approximate computing techniques to strike a balance between accuracy and resource utilisation. AC has shown great potential in reducing the power consumption of large CNN models, both in the HW and SW domains.

Frameworks for simulation of approximate computing techniques applied to CNNs already exist and three were presented in subsection 2.3.2: TypeCNN, AxCNN, and ProxSim. However, the frameworks are aimed at the development of the CNNs and the demographic is the *machine learning developer* rather than the *circuit developer*. Furthermore, these three proposals are limited in the sense that the scalability and generalisation of the CNN models are of minor consideration. They contribute greatly to the methodology of designing cross-layer end-to-end simulations prior to RTL design but lack the interpretation of the actual outcome of a CNN model.

The purpose of this thesis is to provide a *benchmarking tool*, that in a broad sense evaluates a supplied *approximate arithmetic design* in the context of CNNs. The evaluation will seek to inform the user about the potential savings in **latency**, **power consumption**, and the **impact on accuracy** for the context in which the user wants to apply the supplied design. This benchmarking system is summed up in the problem statement from the introduction:

"How can a benchmarking tool provide an ASIC developer with relevant metrics to evaluate an *approximate* arithmetic circuit as an integral part of a large scale system, i.e. a *neural network*, prior to implementation?"

In this chapter the solution to the problem statement will be presented on a functional level. Preliminary questions will be asked at the end of the chapter, which will be used to guide the development of the tool.

3.1 Functional Overview of the Tool

Before implementing the *benchmarking tool*, an overview at a functional level is presented. In Figure 3.1 a functional diagram is presented subdivided into three steps:

I) **Circuit analysis:** A user-supplied AC circuit is investigated in isolation. The purpose of this step is to evaluate relevant metrics, this allows the user to balanc the *power consumption*, *latency*, and *error distribution* characterstics of the circuit. The *error distribution* is also essential for the other two steps.

II) **Small-scale Implementation:** Three networks are created: An *approximate model*, an *exact model*, and a *probabilistic model*. The three models are structurally the same and will share the same weights. The *exact model* takes a random input, processes it, and the output is used as a reference for the other two models. The *probabilistic model* takes the same random input, processes it multiple times, compares the output to that of the *exact model*, and generates an *error distribution*. The *approximate model* takes the same random input, processes the input with the supplied ''approximate arithmetic design'' in-place, the output of which will be compared with the output of the *exact model*, giving an *error*. The likelihood of seeing that error, given the distribution from the output of the *probabilistic model* is evaluated, and by extension the *probabilistic model* is evaluated, i.e. if there is a high likelihood of seeing the *error* the *probabilistic model* and the *approximate model* are similar, and thus the *probabilistic model* can be scaled and utilised in **step III**.

III) **Full-scale Simulation:** A user-defined CNN is then introduced in **step III** and the statistical model found in **step II** is used to infer the influence of a given approximate arithmetic circuit in the user-defined problem. The purpose of this step is to provide the user of the benchmarking system with a qualified impression of how AC affects the performance of their application-specific full-scale CNN, prior to netlist design.

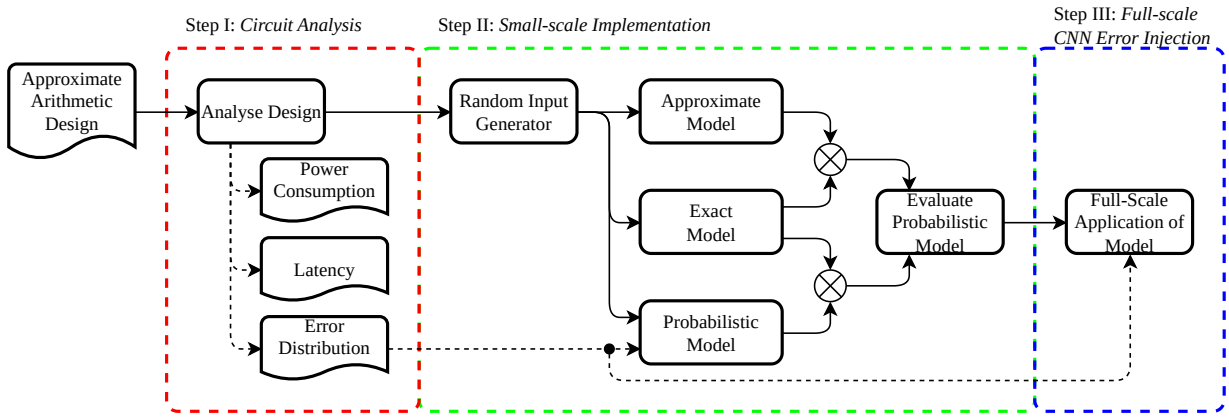

Figure 3.1: Functional diagram of the *benchmarking tool*. An *approximate arithmetic design* is supplied to the tool and is processed through three steps. In **step I** the design is analysed, producing three metrics to evaluate the design with: *Latency*, *power consumption*, and *error distribution*. The evaluated error distribution is then utilised in **step II**, where three NNs are created: An *approximate model*, an *exact model*, and a *probabilistic model*; an input is generated and propagated through each model and the error is computed with respect to the *exact model*. Based on the error at the output, the *probabilistic model* is evaluated based on ''how well it represents the *approximate model*''. Lastly, in **step III** the *probabilistic model* is scaled and applied to a full-scale neural network.

The idea behind creating a *probabilistic model* as opposed to a generally applicable *approximate model* is the potential of the speedup. To implement the *approximate model* it will be necessary to fundamentally rework how convolutional sums are computed since it is not possible to rely on

fast implementations of vector/matrix multiplication, if the innermost MAC operation is modified. However, given a probabilistic model that can emulate the errors of the *approximate model* and add the error as ''noise'', the fast implementations of vector/matrix multiplication can be utilised. Furthermore, the generalisation and scalability of the error modelled as ''noise'' is likely simpler than implementing the *approximate arithmetic circuits* in a full-scale model, where number representation, overflow, etc. will play an important role.

Although, the three models in **step II** should share the weights, a question arises: *how should the weights be trained?* Training the *exact model* and sharing its weights with the other two models would be simple, as a standard machine learning library can be utilised. However, the weights from training *exact model*, may not yield good results in the *approximate model*. Training the *approximate model* and sharing its weights may ensure that the optimisation is performed in the real ''cost-landscape'', however, it is not guaranteed that a NN with approximate can be trained; the derivatives required for back-propagation may not be well-defined. It is seen as essential to research, test, and implement a method for training the weights. Furthermore, it is seen as beneficial for scalability and generalisation if the *probabilistic model* could be trained in place of the *approximate model* if the behaviours of the models are almost identical. The layers added in the *probabilistic model* to simulate the error, should be easy to integrate in full-scale applications.

From a user's perspective, an *approximate arithmetic design* should be supplied to the *benchmarking tool*, and in return, the user should get insight into the **reduction in latency**, the **reduction in power consumption**, and get a **noise model** based on the error distribution, that can be applied to a CNN of the user's choosing. The **noise model** should also include some evaluation of *how well it fits*, which is the purpose of **step II**.

3.2 Delimitation and Research Questions

The solution presented in section 3.1 is broadly applicable, however, it is deemed infeasible to implement a solution capable of evaluating any *approximate arithmetic circuit* in any CNN, why a delimitation is performed in this section.

Multipliers will be the main focus w.r.t. arithmetic circuits. The logic circuits required to perform multiplication are comprised of more gates and by extension are often slower and consume more power. Approximating addition would also lead to some savings in latency and power consumption, however, implementation, debugging, and testing is not prioritised.

No Sequential Logic will be implemented or modelled. As mentioned in section 2.2 sequential multiplication algorithms are slow, but not because of complex logic structures, rather they are tied to a clock. As shown in Figure 2.18 it is possible to construct using a single addition circuit, a register, and some `AND`-gates. Reducing the amount of gates may lead to power saving, however, the latency is tied to the clock, and little is to gain in comparison with combinatorial logic.

Only CNNs will be discussed/implemented due to time constraints. The number of different NN architectures is essentially endless, and the number of different methods is too great to be extensively tested and implemented. However, the resulting methods and implementation may be easily modifiable to accommodate a greater number of methods and architectures.

3.2.1 Research Questions

To further help guide the development and implementation of the *benchmarking tool*, a set of *research questions* have been formulated, and presented below:

I) How can the *error distribution*, *latency*, and *power consumption* of an approximate arithmetic circuit be evaluated and presented to the user of the benchmark system?
II) What is the effect of approximate arithmetic on the training of a CNN?
III) How can the deterministic error distribution of an approximate arithmetic circuit be modelled probabilistically and how can this model be applied to a CNN?
IV) How well does the model fit a deterministic simulation, and can the model be scaled and generalised to the application provided by the user?

These questions will be investigated and answered in chapter 4, 5, and 6, which are structured in the same three steps presented in this chapter.

Step I: Circuit Analysis 4

This chapter walks through the design and implementation of **step I** presented in chapter 3 and highlighted in Figure 4.1. **Step I** of the benchmarking tool revolves around choosing metrics and developing models to evaluate approximate arithmetic circuits such as adders and multipliers independently from any application.

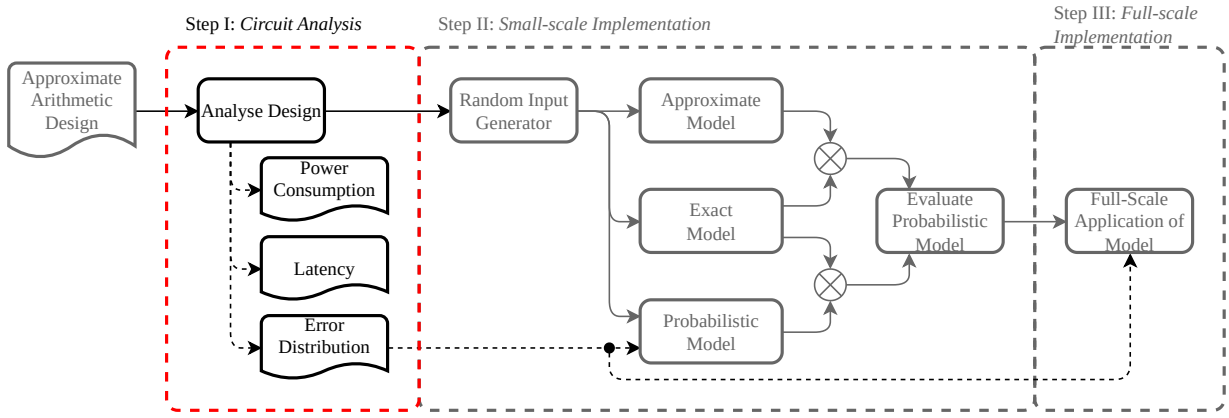

Figure 4.1: Functional diagram of the *benchmarking tool*. An *approximate arithmetic design* is supplied to the tool and is processed through three steps. In **step I** the design is analysed, producing three metrics to evaluate the design with: *Latency*, *power consumption*, and *error distribution*. The evaluated error distribution will then be utilised in **step II**.

Terminologies like *power consumption*, *latency*, and *inaccuracy* are often used when making compromises with a design, but *how are they defined?*

The Power Consumption of a circuit is usually defined in Watts, i.e. *how much energy is used per second*. However, this definition is influenced by the circumstances of the implementation, i.e. the technology of the implementation, leakage, etc. To generalise the *power consumption*, the *transistorcount* is utilised. An optimistic guess w.r.t. the reduction in power consumption would be to say that the percentage-wise reduction in transistors translates into the same percentage-wise reduction in power consumption. This estimate exchanges the precision of simulation tools with a simple and more general metric.

The Latency of a digital circuit can be defined by how long it takes to perform the intended operation [111]. *Throughput* (rate at which a digital circuit comptes outputs from new inputs) of a digital circuit can also be used for the same purpose, but with combinatorial logic the throughput is exactly the reciprocal of latency [111]. It takes a small amount of time for a gate to switch its value, and thus the latency is tied to the path from input to output, which has the largest cumulative

gate-delay: **The critical path**. Given this path, the latency of a circuit can be estimated by summing the delays from each gate in series [112].

The inaccuracy will be defined by *errors metrics* induced by adjusting the **power consumption** and/or **latency** with relation to an exact counterpart. An important error metric is the *difference* defined as [23]:

$$\text{diff}(f(x),\ \tilde{f}(x)) \triangleq ||f(x) - \tilde{f}(x)|| \tag{4.1}$$

where:

x	input to functions f and $\tilde{f}$
$f(x)$	exact function performed on input x
$\tilde{f}(x)$	approximate function performed on input x

There are several error metrics, which utilise the difference, some of them are defined below [23]:

WCD (Worst Case Distance):

$$\max_{x \in X} \big(\text{diff}(f(x),\ \tilde{f}(x))\big) \tag{4.2}$$

MAE (Mean Absolute Error):

$$\mathbb{E}\big\{\text{diff}(f(x),\ \tilde{f}(x))\big\} \tag{4.3}$$

MSE (Mean Squared Error):

$$\frac{1}{|X|} \sum_{x \in X} \big(\text{diff}(f(x),\ \tilde{f}(x))\big)^2 \tag{4.4}$$

Eq. (4.2), (4.3), and (4.4) are good tools to indicate the precision of the approximation performed, however, they are tied to the numerical values of the evaluated input. In some contexts it would be more appropriate to examine how often errors happen, the ER (Error Rate) is defined as [22]:

$$\frac{|W|}{|X|}, \qquad W = \{x \in X | f(x) \neq \tilde{f}(x)\} \tag{4.5}$$

In digital circuit design, it is also relevant to view how much the evaluated input differs from approximate to exact on the bit level. For this purpose, the HD (Hamming Distance) can be utilised; the HD between two binary values, x and y, is defined as the count of bits that differ:

$$\sum_{i=0}^{n} \mathbb{1}[x_i \neq y_i] \tag{4.6}$$

In circuits with multiple input-output relations, it can be beneficial to evaluate the MHD (Mean Hamming Distance) , defined as:

$$\frac{1}{|X|} \sum_{x \in X} \sum_{i=0}^{n} \mathbb{1}[x_i \neq y_i] \tag{4.7}$$

4.1 Gates, Transistors, and Delay

In the benchmarking system, no assumptions are made on the user choice of hardware technology realising the approximate arithmetic circuits. The user is expected to provide a functional approximate RTL circuit. Based on the user input, the appropriate abstraction level for **power consumption** is deemed to be **gate-count**. Furthermore, the **latency** has been defined as the sum of the gate-delays on the *critical-path*.

The design of the analysis tool that constitutes **step I** of the benchmarking system is implemented such that a user can change assumptions about the number of transistors in each gate type, and the latency of the individual logic gates. Furthermore, the user can modify which gates are available in the synthesis. The gates that are available in the *benchmarking tool* to choose from and their assumed `CMOS` layout are congruent with those presented in [113]. The transistor count is presented in Table 4.1.

To get an estimate of the critical path delay of each available logic gate, an estimate is derived based on research in [114], that finds the propagation delay of a *conventional* inverter and `NAND`-gates to be about 150 ps. The estimate used for the remainder of the logic gates is 300 ps for `AND` and `OR` gates and 450 ps for `XOR` and `XNOR`. It is **emphasised** that these delays are assumed to be provided by the user of the benchmarking system and that these estimates are only placeholder values used for exemplification.

Table 4.1: Assumptions on transistor count and latency for a selection of logic gates. These estimates are used as examples through the entirety of the step one analysis examples in this project.

	AND	XOR	NAND	OR	XNOR	NOT	Reference
Transistor Count	6	10	4	6	10	2	[113]
Propagation Delay	300 ps	450 ps	150 ps	300 ps	450 ps	150 ps	Inspired by [114]

Before descending into the substance of *how* to evaluate approximate arithmetic circuits, a short emphasis is made on what circuits to assess. As previously presented multiple libraries of approximate arithmetic circuits exist, where the EvoApproxLib span a great variety. In Appendix C a selection of adders and multipliers from the EvoApproxLib is presented and they are summed up in Table 4.2. In the appendix, **step I** is utilised to evaluate the **power consumption**, **latency**, and **error distributions**, and relevant figures and values from the appendix will appear as examples during this chapter.

Table 4.2: The chosen approximate circuits from EvoApproxLib [4]. All circuits perform signed 8-bit arithmetic operations.

mul8s_1L12	mul8s_1KV9	mul8s_1KV8	mul8s_1KVM	mul8s_1KVA	mul8s_1L2J	mul8s_1KV6	add8se839	add8se8VQ	add8se8NH	add8se8CL
Approx	Approx	Approx	Approx	Approx	Approx	Exact	Approx	Approx	Approx	Exact
Multipl	Multipl	Multipl	Multipl	Multipl	Multipl	Multipl	Adder	Adder	Adder	Adder

4.2 RTL Synthesis: Gatecount and Critical Path Delay

Designing a digital circuit can be viewed at multiple abstraction levels and in Figure 4.2 an example of how the abstraction levels can be itemized is presented (abstraction levels taken from Fig 1.2 in [3] and the explanation of the levels are taken from the surrounding sections and subsections). *Functional design* covers the fundamental idea behind the development. From the specifications of the *functional design* both the *architecture* and *micro-architecture* can be elaborated. The *architecture* is a translation from the *functional design* to a block-level representation, whereas the *micro-architecture* describes hierarchical design details, such as blocks and sub-blocks, interfaces, pin connections, reset capabilities, synchronous/asynchronous designs, etc. In the *RTL design* the *micro-architecture* is used as a reference to develop the design in an HDL (Hardware Description Language) such as Verilog and VHDL, furthermore, the design is simulated and synthesised to a *gate-level* netlist. *Switch level design* represents the design in standard cells for ASICs or architecture resources in FPGAs.

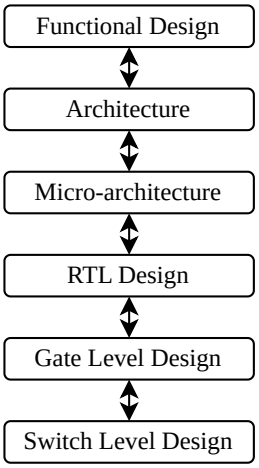

Figure 4.2: Different levels of abstraction in digital design, inspired by Fig 1.2 from [3].

Power consumption has been equated to the *transistor-count* of the designed approximate arithmetic circuit and **latency** has been equated to the accumulated gate-delay on the *critical path*. Both *transistor-count* and the *critical path* are innately tied to the hardware implementation of the circuit, why it is relevant to *synthesise* the circuit: The relevant abstraction layers will be limited to the *RTL design* and the *gate level design*. The end product of these two layers is a gate-level netlist, which is a ''representation of the functional design in the form of combinational and sequential logic cells'' [3]. Where *sequential logic* is defined as digital logic with which the output is a function of present input and past output, e.g. memory cells. *Combinational/combinatorial logic* is only dependent on present input, e.g. logic gates.

Given the gate-level netlist of a design, it should be possible to count the number of gates required for the design, find the critical path delay, and by extension have two out of the three metrics metrics: **Power consumption** and **latency**. However, going from an RTL design to a corresponding gate-level netlist requires multiple intermediate steps. In Figure 4.3 a generic process from RTL design to the physical design is depicted: A user creates a design written in an HDL, the functionality of the design is verified, and synthesis can be commenced. *Synthesis* of a design requires the RTL design and libraries, whereas *design constraints* can be optionally defined and passed to the synthesis tool. Some common constraints are *area*, *speed*, and *power*. The product of a successful synthesis is the gate-level netlist with which the *physical design* can be produced.

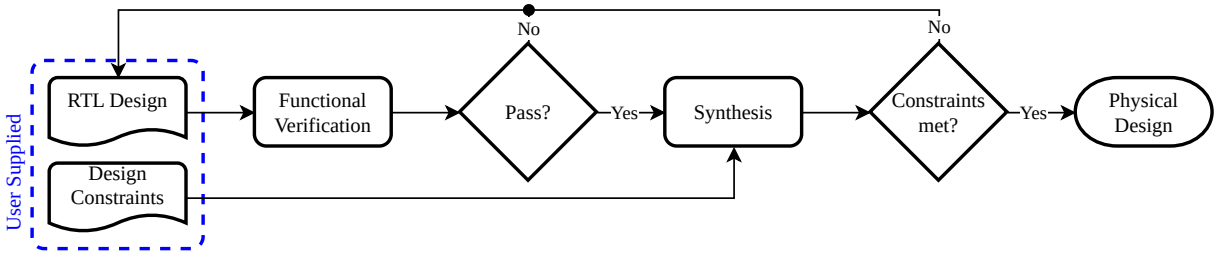

Figure 4.3: Generic simulation and synthesis flow. The *user* supplies design constraints that their design should meet. An RTL design is created, its functionality is verified, and the design is synthesised based on the constraints. If the design does not meet the constraints, the RTL will have to be revisited. The end product of the *simulation and synthesis flow* is a netlist (inspired by Fig 1.3 from [3]).

In section 2.3 some of the approximate computing methods for arithmetic circuit design were presented. Since the nature of approximate computing allows errors, designing a globally applicable functional verification is not feasible, and this task is off-loaded to the user. Furthermore, the purpose of analysing the design is to report on the metrics **power consumption** and **latency**, why applying design constraints is redundant. This greatly reduces the scope of the RTL synthesis in Figure 4.3 and the only block left is the *Synthesis*-block.

4.2.1 Synthesis Flow

The *synthesis*-block from Figure 4.3 is elaborated in Figure 4.4, where the *desired flowchart* is represented. From a user's perspective, the synthesis process should be a blackbox with minimum of two interfaces: Input the RTL design and output a netlist. To facilitate this process it is essential, that the tool can *read the RTL design*, *synthesise the design*, and write the netlist to a Verilog file (marked in Figure 4.4 with black lines/text). Furthermore, some *nice-to-have* features would be the capability to flatten the hierarchy of the design, map the netlist to a specific set of gates, and write the netlist to a JSON formatted file (marked in Figure 4.4 with grey lines/text). Flattening the hierarchy would simplify further processing of the netlist by removing hierarchical boundaries and superfluous references; keeping the hierarchy intact is not essential when the purpose is to find the **critical path delay** and **gate-count**. Mapping the netlist to a specific set of gates would allow for customisability on the user's part, and add another way to adjust the **power consumption** and **latency** of the designed arithmetic circuit. Writing the netlist to a JSON file will simplify the analysis of the netlists, e.g. in Python a JSON-file can be decoded with the package `json` to a dictionary of dictionaries.

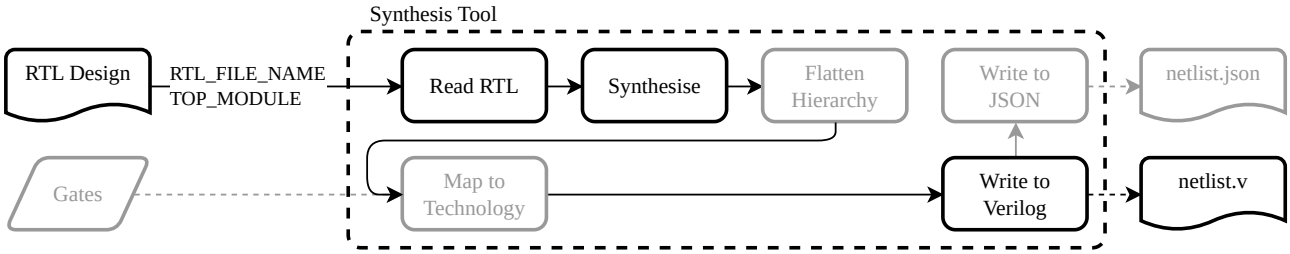

Figure 4.4: Synthesis flowchart. An RTL design is supplied by passing its path with the name of the *top module* to a synthesis tool. The synthesis tool should *read the RTL* and *synthesise the design* resulting in a netlist. This netlist may contain submodules, and to simplify the data processing of the netlist, the hierarchy should be flattened. The netlist should then be exported to a Verilog file. Furthermore, two nice nice-to-have features are the possibility to force the synthesis tool to use a specific set of gates and write the netlist formatted in a JSON file, *Map to Technology* and *Write to JSON*, respectively. Black lines/text indicate *need-to-have* functions/features, whereas the grey lines/text indicate *nice-to-have* functions/features.

The choice of the *synthesis tool* is yosys [115]. Yosys describes itself as a ''framework for Verilog RTL synthesis'' [115] and is capable of accommodating both the *nice-to-have* and *need-to-have* features previously described. Many alternatives could have been utilised for this purpose, e.g. Cadence Synthesis, Synopsys Design Compiler, Intel Quartus, etc. however, yosys is more lightweight than the full-scale EDA (Electronic Design Automation) tools. Furthermore, it is free of charge and open-source. Yosys can be interfaced through a CLI (Command Line Interface) and the blocks in the flowchart in Figure 4.4 can be mapped one-to-one with available commands:

- *Read RTL:* `read_verilog [options] [filename]`
  - Load modules from a Verilog file to the current design. A large subset of Verilog-2005 is supported [115]
- *Synthesise:* `synth [options]`
  - This command runs the default synthesis script. This command does not operate on partly selected designs [115]
  - The top module will be provided using `-top $::env(TOP_MODULE)`
- *Flatten Hierarchy:* `flatten [options] [selection]`
  - This pass flattens the design by replacing cells by their implementation. This pass is very similar to the 'techmap' pass. The only difference is that this pass is using the current design as mapping library [115]
- *Map to Technology:* `abc [options] [selection]`
  - This pass uses the ABC tool [1] for technology mapping of yosys's internal gate library to a target architecture [115]
  - The gates will be provided using `-g $::env(GATES)`
- *Write to Verilog:* `write_verilog [options] [filename]`
  - Writes the current design to a Verilog file [115]
- *Write to JSON:* `write_json [options] [filename]`
  - Writes a JSON netlist of the current design [115]

The flow presented in Figure 4.4, using yosys as the synthesis tool and the functions presented in the list above, is implemented as a Makefile function, and the source code can be found in Appendix A under `/rtl_analysis/Makefile` and `/rtl_analysis/netlist.tcl`.

After performing the synthesis of a circuit, the generated netlist can be visualised using a tool like `netlistsvg`, with which an SVG file of the netlist can be created. In Figure 4.5 an example of this can be seen. The design `add8s_8VQ` from EvoApproxLib [4] has been synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR`, and `XNOR` gates. Furthermore, in Figure 4.6 and Figure 4.7 examples of the customisable *map to technology* are visualised, by mapping to `AND` gates and `OR` gates, respectively. Notice, that both circuits contain `NOT` gates, which is required for *functional completeness* and is therefore unavoidable when using the `abc` function from yosys.

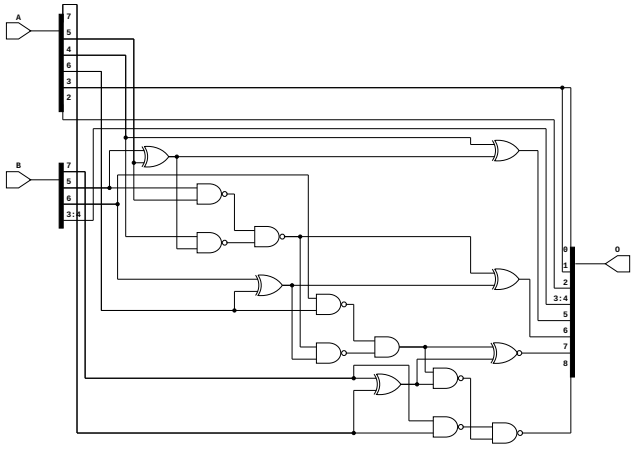

Figure 4.5: `add8se_8VQ` from EvoApproxLib [4] synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

Although the complexity of these circuits is low enough, that the gatecount and critical path (in number of gates) can be found in a matter of minutes by hand, in larger and more complex designs, it would be tedious and very difficult to keep track of. For this reason the processes of *counting the gates* and *finding the critical path* are automated in subsection 4.2.2 and subsection 4.2.3.

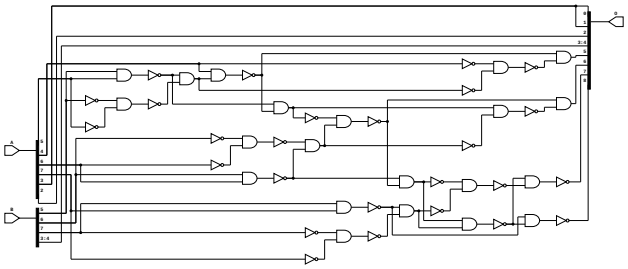

Figure 4.6: `add8s_8VQ` from EvoApproxLib [4] synthesised to `AND` and `NOT` gates and visualised using `netlistsvg`.

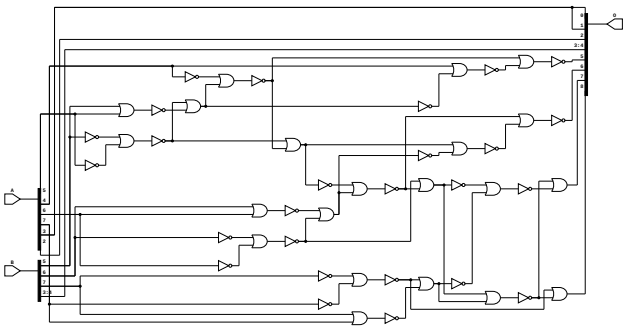

Figure 4.7: `add8s_8VQ` from EvoApproxLib [4] synthesised to `OR` and `NOT` gates and visualised using `netlistsvg`.

4.2.2 Counting the Gates

Assuming that the synthesis described in subsection 4.2.1 has been performed, both a netlist in Verilog and a netlist in JSON should be available. The JSON file of the netlist is used to count the gates and the structure can be seen in Figure 4.8. Since the hierarchy of the design is flattened during the synthesis, the only module will be the `TOP_MODULE`. From here, the gates can be counted by looping over all the `cells` and incrementing counters based on their `type`.

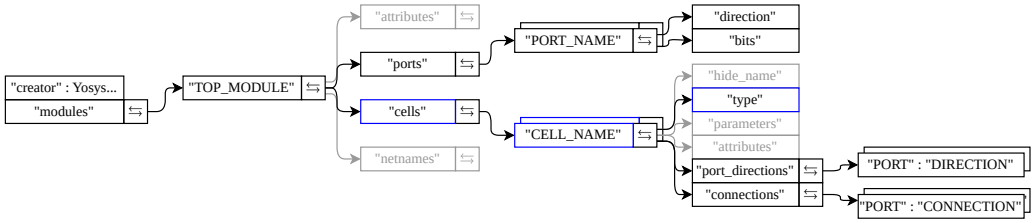

Figure 4.8: `netlist.json` data-structure, ⇆ indicates the key's value is a dictionary. The relevant keys for **counting gates** are highlighted with blue.

To index the JSON structure within `netlist.json` the Python script needs the name of the `TOP_MODULE`, which can be provided as an environment variable by exporting its name from the Makefile function (see `/rtl-analysis/Makefile` in Appendix A). Given `netlist.json` and the name of the top module, the gates and their types can be counted, following the flow in Figure 4.9. The `cells` dictionary is indexed following the JSON structure in Figure 4.8 and the script loops over every cell incrementing typecounters. The typecounters are used to estimate the total amount of transistors required for the circuit by summing the products of the *typecounts* and the *transistors per type*. The typecounts are summed to calculate the total **gatecount**. The typecounts, the total amount of transistors, and the gatecount are saved in a csv file.

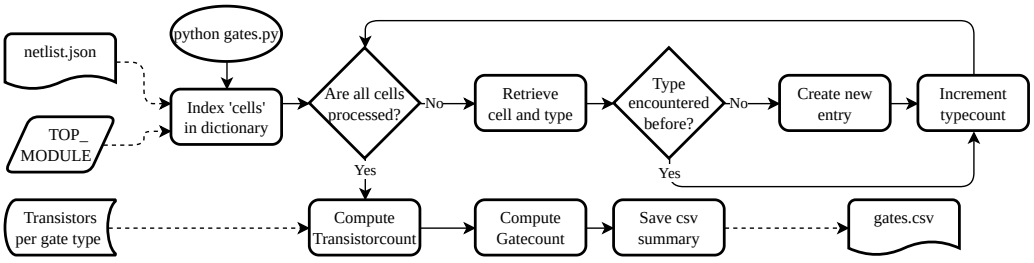

Figure 4.9: Gatecount flowchart. The python script `gates.py` is called with an available `netlist.json`, the `TOP_MODULE` available in the environment, and the transistors per gate type set. The script loops over all cells in the design, incrementing counters based on the cells' types. The transistor count required for the design is estimated based on the type counters, the type counters are summed to give the gatecount, and the information is exported to a csv file.

Performing the gatecount on the netlists visualised in Figure 4.5, Figure 4.6, and Figure 4.7 yields three csv files, with contents tabularised in Table 4.3. In the third and fourth row, there are no `XOR`, `NAND`, or `XNOR`, which matches with Figure 4.6 and Figure 4.7 since they have been mapped to `AND` and `OR` gates, respectively; indicating that the count is correct (which can also be verified by counting the gates by hand).

Table 4.3: Gatecounts of the netlists visualised in Figure 4.5, Figure 4.6, and Figure 4.7.

Netlist	AND	XOR	NAND	OR	XNOR	NOT	gatecount	transistorcount†
Figure 4.5	1	5	8	0	15	0	15	98
Figure 4.6	21	0	0	0	0	26	47	178
Figure 4.7	0	0	0	21	0	26	47	178

† Calculated based on the assumptions presented in Table 4.1.

A note on transistors: The ‘‘gate technology’’ of an implementation of a gate can be classified in one of four ways [116]: Resistor-Transistor Logic, Diode-Transistor Logic, TTL (Transistor-Transistor Logic) , and CMOS (same as TTL, using FETs rather than transistors). The choice of ‘‘gate technology’’ also affects the amount of transistors required for implementing a specific gate, i.e. a CMOS NOT gate architecture may use 2 transistors [113] whereas a TTL NOT gate may use 6 [117]. Due to discrepancies in the amount of transistors required for a gate, configuration hereof is possible within the `gates.py` script.

4.2.3 Critical Path

Again, it is assumed that the synthesis described in subsection 4.2.1 has been performed and the JSON netlist is available. The JSON netlist is the starting point for finding the critical path and the structure can be seen in Figure 4.10, where the relevant keys and values are highlighted in blue. The hierarchy of the design is still flat, why there is only one module. In order to figure out the critical path of a design, it is necessary to keep track of all paths from input to output and perform a search in the paths to determine the longest path.

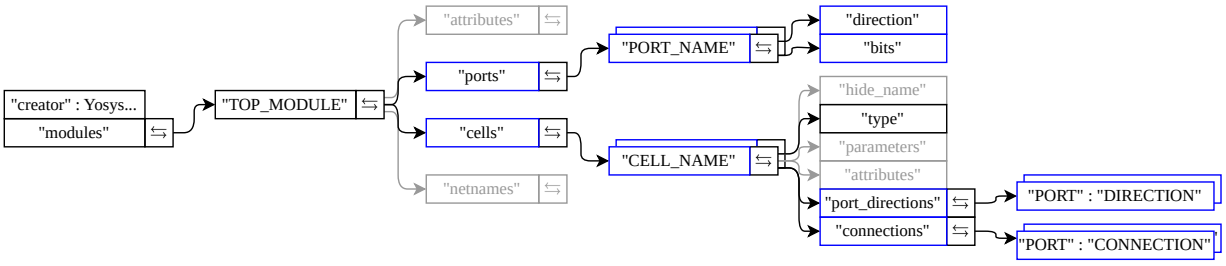

Figure 4.10: `netlist.json` data-structure, ⇆ indicates the key’s value is a dictionary. The relevant keys for finding the **critical path** are highlighted with blue.

In section 3.2 the benchmarking tool was delimited to only analysing combinatorial logic. From this delimitation it must follow, that there are no ‘‘cyclic’’ connections in the designs. This observation simplifies the process of finding the longest path since the combinatorial circuit can be translated into a DAG (Directed Acyclic Graph). Analysis of DAGs is well documented, and tools exist to perform the analysis. A tool fit to process DAGs is the `networkx` package for Python, which describes itself as ‘‘a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks’’ [118]. Using `networkx` to analyse the DAG also requires the

creation of the DAG; all ports and gates are converted to *nodes*, the nets interconnecting the ports and gates are the drawn connections, where the direction of the connections are from one gate's/port's output to another's input. Furthermore, it is possible to define a ''weight'' for the connections, and the ''weights'' are chosen to be the gate-delay of the driving gate. This choice will affect the search for the **critical path** since it is now not just the number of gates that define the longest path, but the propagation-delay. The scripts used to perform this analysis can be found in Appendix A: `/rtl-analysis/Makefile` and `/rtl_analysis/paths.py`.

Example 4.2.1: Netlist to DAG

In Figure 4.11 the translation from an example circuit to a DAG can be seen. The ports are marked in green and the gates are marked in fuchsia, and the translated DAG follows the same colour scheme. The connections on the right-hand diagram are weighted based on the delay values shown on each of the gates on the left-hand side. The wires are modeled as ideal, why there are no delays from port to gate. In this example, the **critical path** would be `AND-OR-XOR` with the propagation delay of 0.3 ns + 0.45 ns + 0.15 ns = 0.9 ns.

Figure 4.11: Example translation of a *combinatorial logic circuit* into a DAG. The ports are marked in green and the gates are marked in fuchsia to clarify how they are placed as nodes. The connections on the right-hand diagram are weighted based on the delay values shown on each of the gates on the left-hand side. The wires are modeled as ideal, why there are no delays from port to gate. The gate-delay values are example values and are not necessarily correct.

In Figure 4.12 and Figure 4.13 an example of a netlist and its corresponding DAG can be seen. Given the DAG has been set up properly with `networkx`, it is possible to find the longest/critical path by calling `networkx.dag_longest_path(graph, weight='weight')`. The **critical path** is exported to a textfile wherein the estimated propagation-delay for circuit is noted as well as the number of gates passed and the names of the gates passed.

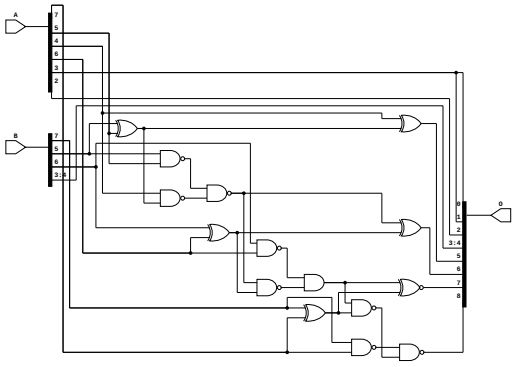

Figure 4.12: Visualisation of an example netlist (Copy of Figure 4.5).

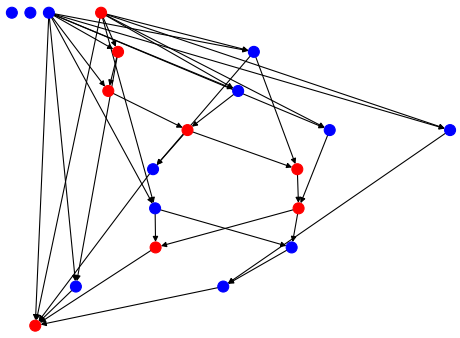

Figure 4.13: Visualisation of the DAG representation of Figure 4.12. The red nodes indicate the calculated **critical path**.

Given the `critical path` two important metrics are readily available: *The number of gates in the path* and the *propagation delay*. Returning to the example netlists visualised in Figure 4.5, 4.6, and 4.7 and performing the **critical path** analysis yields the values presented in Table 4.4.

Table 4.4: Number of gates and propagation delay for the critical paths of the netlists visualised in Figure 4.5, 4.6, and 4.7.

Netlist	Number of gates	Propagation delay†[ns]
Figure 4.5	6	1.65
Figure 4.6	16	3.6
Figure 4.7	15	3.45

† Calculated based on assumptions presented in Table 4.1.

A note on gate-delay: The gate-delay of a gate is dependent on the ''gate technology'', why there can be discrepancies between the estimation from the implementation to real-life. To accommodate the discrepancies it is possible to configure the gate-delay associated with each logic gate in `/rtl_analysis/paths.py`.

A note on sequential circuits: The implementation assumes that the supplied circuit is *combinatorial logic*, why the netlist can be represented as a DAG. This assumption will be wrong given a *sequential logic* circuit, however, for sequential logic the propagation-delay is not a metric of how fast the circuit *performs* its function, which is defined by the clock, however, it defines a ceiling for the clock frequency [119]. For this reason, the corresponding metric would be proportional to 1/clock frequency depending on how many times the sequential blocks have to be used, before reaching the result.

4.3 Error Simulation

To analyse the inaccuracy of the approximate arithmetic circuits, the metrics from the beginning of the chapter are computed. As all circuits from the EvoApproxLib [4] are provided with both a Verilog and C implementations, a C++ class is implemented that evaluates the **mean square error**, **mean absolute error**, **worst case distance**, **error rate**, and **mean hamming distance**. Since the approximate circuits are deterministic it is possible to record every input/output relation in a LUT and calculate the metrics from this distribution. The functional circuit corresponding with the netlist from Figure 4.5 is analysed to provide an example.

The circuit is an 8-bit signed adder meaning that both addends can take any integer value between -128 and 127, i.e. $x \in \{-128, -127, -126, ..., 127\}$. It is assumed that the addends take any value with equal probability, i.e. $A, B \sim U\{-128, 127\}$. This means that the $P(A = x) = P(B = x) = \frac{1}{256}$. The joint distribution can be represented as a LUT where each entry is the product of the marginal entries, i.e. $f_{A,B} = P(A = x, B = x) = P(A = x) \cdot P(B = x) = \frac{1}{512}$. It is then possible to describe the error difference as another discrete RV (random variable) (E) with a conditional PMF $f_{E|A,B}$. This PMF is obtained by evaluating every output of the approximate circuits exactly once. The single-value metrics for this distribution are shown in Table 4.5 and The PMF is plotted in Figure 4.14.

Table 4.5: Error-metrics of the distribution presented in Figure 4.14.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
8	77	99.22 %	16	2.75

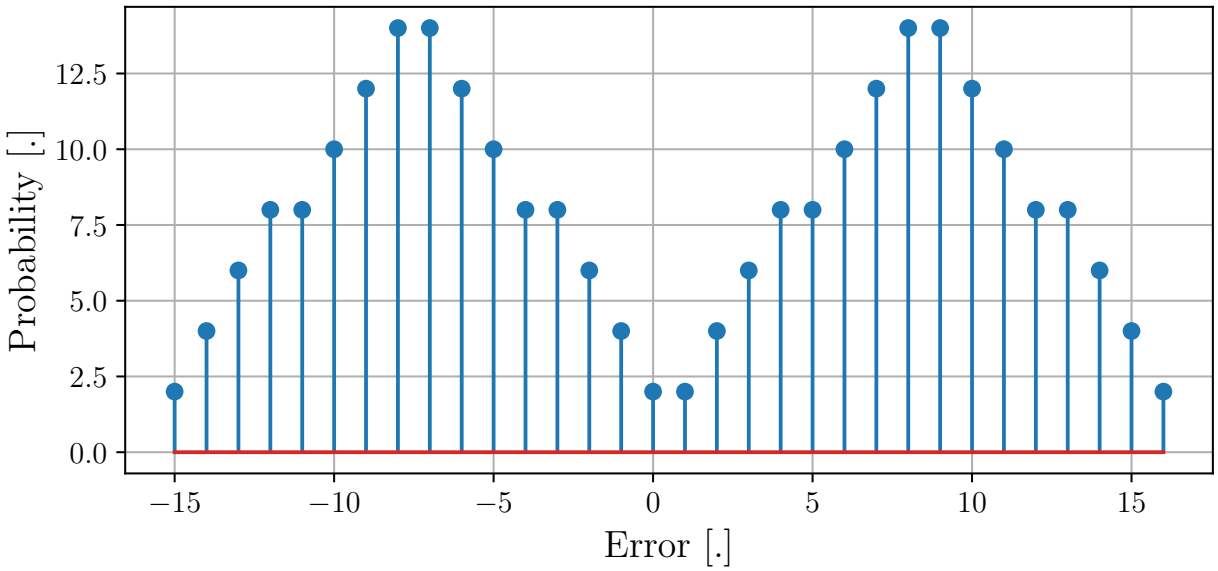

Figure 4.14: PMF of the error distribution of the approximate circuit in Figure 4.5. The distribution is plotted where each bar represents a discrete error distance.

4.4 Circuit Comparison and Summary

To compare the chosen approximate arithmetic circuit is a balance not only between the **power**, **latency** and **inaccuracy**, but also the application-relevant sub-metrics. In Appendix C a thorough analysis is performed for all chosen circuits. The following comparison is a constituent of the appendix, specifically the comparison of the multipliers `mul8s_1KV9`, `mul8s_1KVM`, `mul8s_1L12`, and `mul8s_1KV6`.

For some applications, there could be a boundary to the WCD (Worst-Case Distance) and for different applications, it is desired to trade off a high *gate count* for a short *critical path*. An illustration that aids the comparability of the trade-offs for different approximate circuits is also provided using the `MakeFile` (see `/rtl-analysis/Makefile` in Appendix A), which takes approximate circuit in Verilog and C++ and finds the metrics presented in all previous sections. It is further possible to specify comparison metrics, i.e. one for *power*, *latency* and *inaccuracy* each. These three constitute a vector that can be plotted in three dimensions, where the vector with the lower magnitude is considered more satisfactory compared to the one with a higher magnitude. Firstly an example is shown for the multipliers `mul8s_1KV9`, `mul8s_1KVM`, `mul8s_1L12`, and `mul8s_1KV6`, comparing the gate-count for *power*, critical path gate-count for *latency*, and error rate for *inaccuracy*. This is shown in Figure 4.15.

A few points are highlighted from Figure 4.15. Firstly it is noticed that `mul8s_1L12` has the shortest critical path and lowest gate count, but at the expense of a large error rate. Secondly, the `mul8s_1KV9` has a shorter critical path compared to `mul8s_1KVM`, but trades both total gate count and error rate. Lastly, it is noticed that `mul8s_1KV6` is accurate, as it has 0% error rate and it is noticed that the total gate count is also the highest of the multipliers. However, the critical is even longer for `mul8s_1KVM`. If the user's application values low latency more than a low error rate, the user should choose `mul8s_1KV9` and maybe even `mul8s_1L12` over `mul8s_1KVM` and if the application can only *"afford"* an error rate of 50% only `mul8s_1KVM` and `mul8s_1KV6` is feasible for this selection of multipliers.

Secondly, a comparison is made for the same multipliers using gate-count for *power*, critical path gate-count for *latency*, and WCD for *inaccuracy*. This is shown in Figure 4.16.

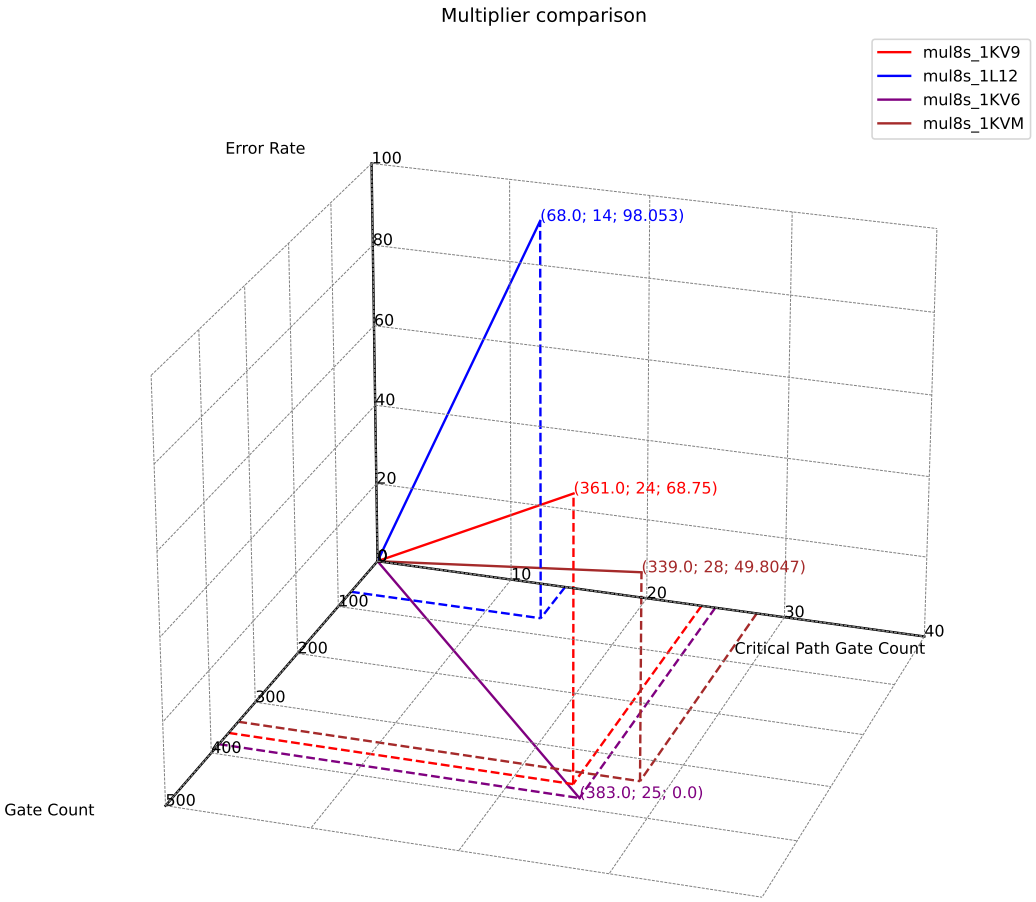

Figure 4.15: 3D-plot of the metrics of `mul8s_1KV9`, `mul8s_1KVM`, `mul8s_1L12`, and `mul8s_1KV6`.

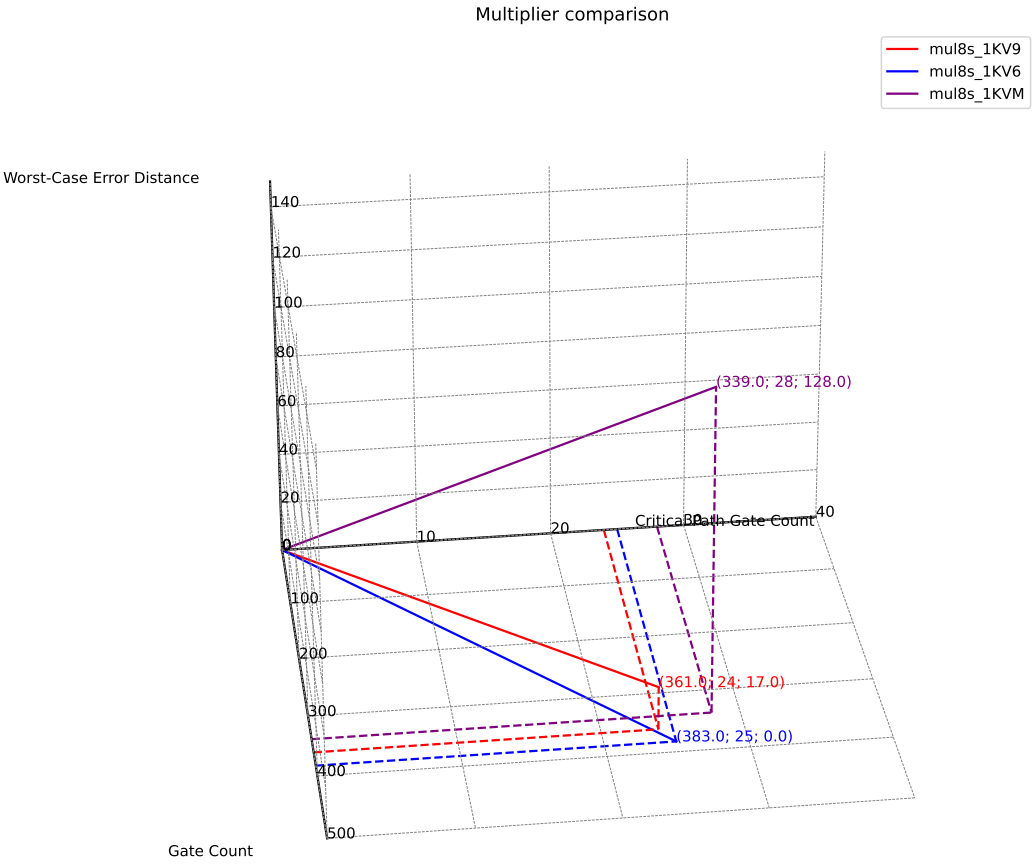

Figure 4.16: 3D-plot of the metrics of `mul8s_1KV9`, `mul8s_1KVM`, and `mul8s_1KV6`.

The takeaway from this comparison is that even though `mul8s_1KV9` had a higher error rate compared to `mul8s_1KVM`, it has a significantly lower WCD. This constitutes the fact that comparing the approximate circuits using single parameter metrics is a challenging task and the ability to visualise these metrics benefits the interpretability.

This concludes the development and implementation of **step I** of the benchmarking system. The abilities of this subsystem can be summarised in four main points:

I) The system can estimate the **power consumption** of a user-provided RTL functional circuit, by synthesising the circuit (also using the user-specified available logic gates) and counting the type and amount of gates used in the circuit, and by extension counting the transistors. This estimate was chosen as the actual power consumption is specific to the technology used in the transistors of the circuit, which is not within the scope of this project to identify. The gate count is viewed as a more general estimate.

II) To estimate the **latency** of the provided approximate circuit, a method is developed to deduce the critical path of the synthesised circuit. The circuits are represented as acyclic directed graphs, to obtain the path with the most nodes (i.e. logic gates). The connections between nodes is weighted by the latency of the specific logic gate to obtain an estimate of the critical path delay. However, these weights are also technology-specific and have to be provided by the user.

III) Error metrics are found by empirically obtaining the degenerate PMF of each circuit, by evaluating the output of approximate circuits exactly once for each of the possible inputs (this is finite as the inputs are quantised by e.g. 8-bits). Single-value metrics can then be calculated from this distribution of errors.

IV) The obtained metrics are compared through a representation of the metrics as a three-dimensional vector, where low magnitudes are desired. These vectors are plotted for different approximate arithmetic circuits to provide a possibility to visually inspect the trade-offs that can be made from the compared circuits.

Step one provides an analysis and comparison tool for approximate arithmetic circuits enabling the user to estimate the circuit's performance for their application. However, this can be a challenging task to extrapolate the metrics presented in step one to the performance of a CNN, which is the case chosen for this project. The goal of the second step in the benchmarking system is to provide a model that can do exactly this.

Step II: Small-Scale Approximate Neural Network

5

This chapter describes the design and implementation of **step II** presented in chapter 3. **Step II** of the benchmarking tool revolves around the implementation and evaluation of a *probabilistic model* in relation with a hierarchically identical *approximate model*. The *probabilistic model* will utilise the *error distribution* from **step I** to generate ‘‘noise’’ emulating the error of the approximate arithmetic circuit, thereby generalising the error. If the outcome of the *probabilistic model* is similar to the outcome of the *approximate model*, the *probabilistic model* can easily be scaled and reused.

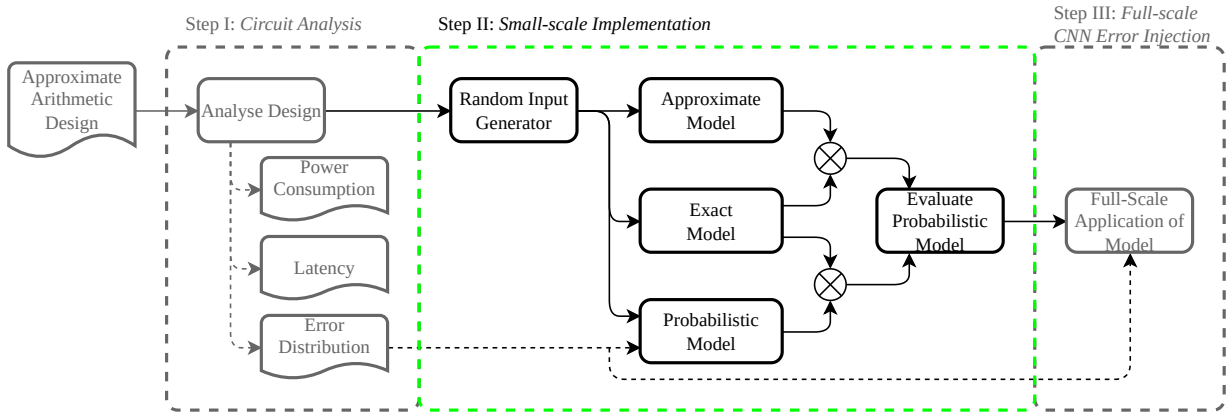

Figure 5.1: Functional diagram of the *benchmarking tool*. An *approximate arithmetic design* is supplied to the tool and is processed through three steps. The evaluated error distribution from **step I** is utilised in **step II**, where three NNs are created: An *approximate model*, an *exact model*, and a *probabilistic model*; an input is generated and propagated through each model and the error is computed with respect to the *exact model*. Based on the error at the output, the *probabilistic model* is evaluated based on ‘‘how well it represents the *approximate model*’’.

Step II requires three CNNs, identical in hierarchy, why the *exact model* will be developed and implemented in the first section of this chapter. In section 5.1 the *exact model* is defined as well as the ‘‘machine learning problem’’ for the models. The structure of the model will be copied in the *approximate model* and the *statistical model*.

The *approximate model* is designed and implemented in section 5.2 using the structure of the *exact model* as a starting point. The outcome of the section will be a model capable of performing forward passes with approximate arithmetic in-place.

The *probabilistic model* is designed in section 5.3 as a modification that can be applied to the *exact model*. The outcome of the section will be general and scalable methods, that can be applied to any CNN model.

Lastly, the training of the models is investigated in section 5.4 to expose a strategy to effectively train the weights of the *approximate model*. Furthermore, the similarity between the training of the *approximate model* and the *statistical model* will be investigated, which may lead to a robust strategy to fit the weights to the *approximate model* of full-scale model in **step III**.

5.1 *Exact Model* - Reference System and Application

The design of the reference system is performed using the TensorFlow framework, and the classes native to the framework are investigated concerning the accuracy of the network. The entire investigation is documented in Appendix B and central takeaways are presented in this section. In addition to the information regarding the structure and functionality of a CNN from section 2.1; the CNN is designed using methodologies native to TensorFlow. This CNN will act as a bridge between state-of-the-art machine learning algorithms, and the use of approximate arithmetic units.

In section 2.1 a block diagram was presented in Figure 2.1, describing the procedure of designing NNs and subsection 5.1.1, 5.1.2, and 5.1.3 follow the three phases of that procedure.

5.1.1 Preliminary Phase

The main purpose of the reference system is to provide a comparison for another implementation of an almost identical system, and to that extent any machine learning problem would suffice. For visualisation purposes, the **identified problem** is chosen to be *image classification*.

Due to the arbitrary choice of image classification, the **data collection** can be reduced to downloading a premade dataset. The choice of the dataset will be based on the following parameters:

Table 5.1: Three important parameters for the choice of a dataset

Size	The dataset should be large enough to fully train a CNN, without requiring gathering more data than is already available
Difficulty	The difficulty should be high enough to make it a challenge for state-of-the-art image classification networks
Configurability	Combined with the difficulty there should be a way to configure the difficulty of the problem, to accommodate for a small-scale network as well as a large-scale network

MNIST [50] is a classical example of image classification, the size is sufficient and visualization is simple. However, it is not a difficult task to classify the dataset. Using the 92 submissions on Image Classification on MNIST [120] as an indication of the difficulty and noting the lowest accuracy is 92,47 % it is clear to see, that it is not a difficult classification problem. Another well-known image classification dataset is the CIFAR-10 [121] dataset. Again, the size is sufficient, visualisation is simple, and the difficulty is increased compared to MNIST. Based on 240 submissions on Image Classification on CIFAR-10 [120] the lowest score is an accuracy of 80,45 %. The configurability of the dataset is decent with 10 classes, however, CIFAR-10 has a sibling dataset: CIFAR-100 with 100 classes. The increase in difficulty from CIFAR-10 to CIFAR-100 is notable, however, not overwhelming for state-of-the-art models. The span of accuracies for models *not using extra training data* on CIFAR-10 from Image Classification on CIFAR-10 is 60,6 % to 99,5 %, whereas the models *not using extra training data* on CIFAR-100 from Image Classification on CIFAR-100 [120] is 19,49 % to 93,36 %. CIFAR-100 is chosen to be the dataset on with which the models will be trained on.

The **data representation** of the CIFAR-100 is sets of images and their corresponding labels. The images are of the shape $(32, 32, 3)$, with each value formatted as an unsigned 8-bit integer. In Figure 5.2 five examples are depicted: a *raccoon*, a *cloud*, a *lamp*, a *keyboard*, and a *beetle*, respectively.

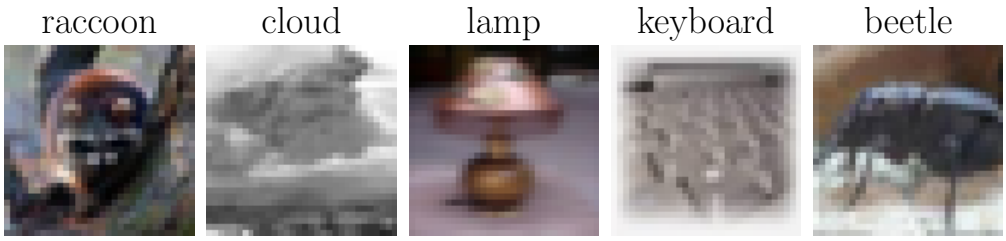

Figure 5.2: 5 example images from CIFAR-100 represented as 32×32 colour images.

In the creation of a small-scale neural network, the complexity of the model should be proportionate with the complexity of the dataset. For the small-scale network, the data should be ''simpler'', whereby the first layer of the CNN can be reduced. The dataset can be reduced to 1/3 of its size by converting to **grayscale**. Multiple ways of converting images to grayscale exists, however, most of the models are concerned with the perception of the human-eye and is just a weighted average over the colours for each pixel, the grayscale conversion will be performed as the mean of the RGB-values.

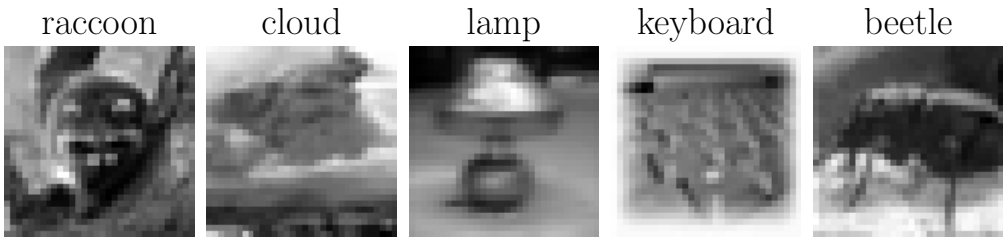

Figure 5.3: 5 example images from CIFAR-100 represented as 32×32 grayscale images.

Each datapoint is still comprised of $32 \cdot 32 = 1024$ 8-bit integers, which is incongruent with a small-scale network. The dataset can be reduced further to 1/12 of its original size, by compressing the images to 16×16 grayscale. The resizing method was chosen based on the results of section B.3 in Appendix B, and a plot of the accuracy using different resizing methods are visualised in Figure 5.4.

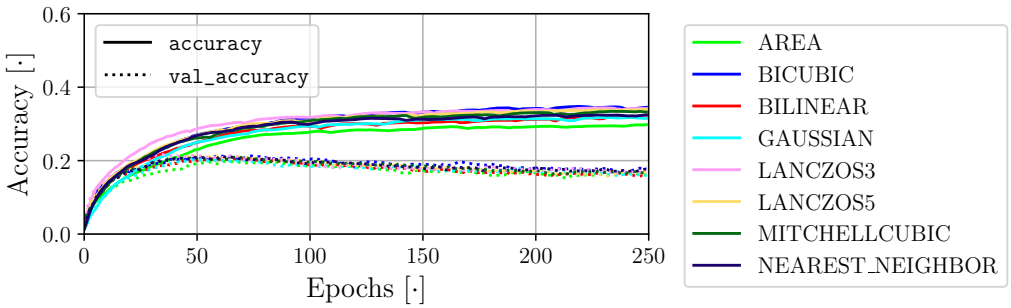

Figure 5.4: (Copy of Figure B.4) The accuracy as a function of epoch training the same model on differently resized datasets. The loss function is `MeanSquaredError` and the optimisation algorithm is `adam`. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

Using *TensorFlow datasets* available resizing methods, LANCZOS3 was chosen to be the resizing method, since it reaches the highest peak accuracy in Figure 5.4.

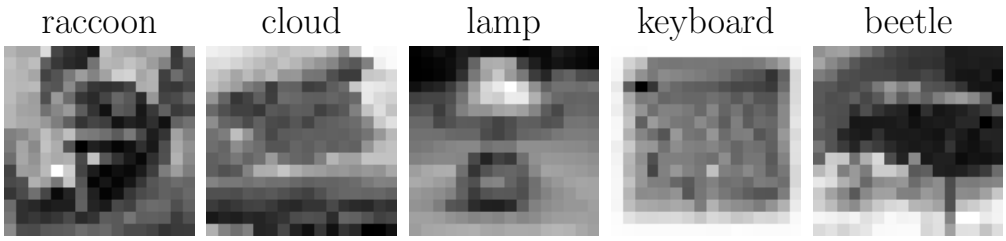

Figure 5.5: 5 example images from CIFAR-100 represented as 16 × 16 grayscale images after being resized using LANCZOS3.

The outcome of the *preliminary phase* is thus a complexity reduced CIFAR-100 dataset consisting of 50.000 training examples, 10.000 test examples, in the shape (16, 16) of 8-bit integers. The labels are converted to one-hot encoding.

5.1.2 Design Phase

There is no analytical process with which an optimal neural network can be designed, so trial-and-error is somewhat necessary. From TensorFlow the basic CNN model from example Convolutional Neural Network (CNN) is taken and utilised as the base network. The example CNN performs image classification on CIFAR-10; changes have to be made to accommodate the chosen (and modified) dataset:

- **Adjust kernel size:** Since the images have been resized to (16, 16), the kernel size produced an error: `ValueError: Exception encountered when calling Conv2D.call(). Negative dimension size caused by subtracting 3 from 2`, which was solved by changing the kernel size to (2, 2).
- **Adjust output layer:** The output of the network should be directly comparable to the labels, which have been one-hot encoded; the output vector should be (100, 1) corresponding with the 100 classes.

This model, `base_model`, is a seven layer CNN with three convolutional layers, two pooling layers, a flattening layer, and two dense layers (not in that order), where all kernels are 2 × 2. The model could potentially be implemented at this stage, however, as depicted in Figure 5.4 this basic implementation using the modified dataset only reaches a test accuracy of ~ 20 % before it is overtrained. To optimise the model, the parameters of the `optimizer` are heuristically chosen. Firstly, the optimisation algorithm was tested. Although `adamax` takes relatively many epochs to reach peak test accuracy, it reaches the highest test accuracy and the effect of overtraining is seemingly negligible compared to the others, and was therefore picked.

Another parameter that can be changed in the `optimizer` is the loss-function, with `adamax` as optimisation algorithm, all the readily available loss functions from TensorFlow were used to train the same network. The results are depicted in Figure 5.6, where `BinaryFocalCrossentropy` and `SquaredHinge` reach a notably higher test accuracy. `BinaryFocalCrossentropy` reaches a higher accuracy in fewer epochs than `SquaredHinge`, and is chosen to be the loss function for the small-scale networks.

Comparing the `val_accuracy` using `BinaryFocalCrossentropy` in Figure 5.6 with the accuracies reached in Figure 5.4 the choice of `adamax` and `BinaryFocalCrossentropy` has improved the accuracy by around 5 percentage points, furthermore, the accuracy in Figure 5.6 does not show signs of being overtrained.

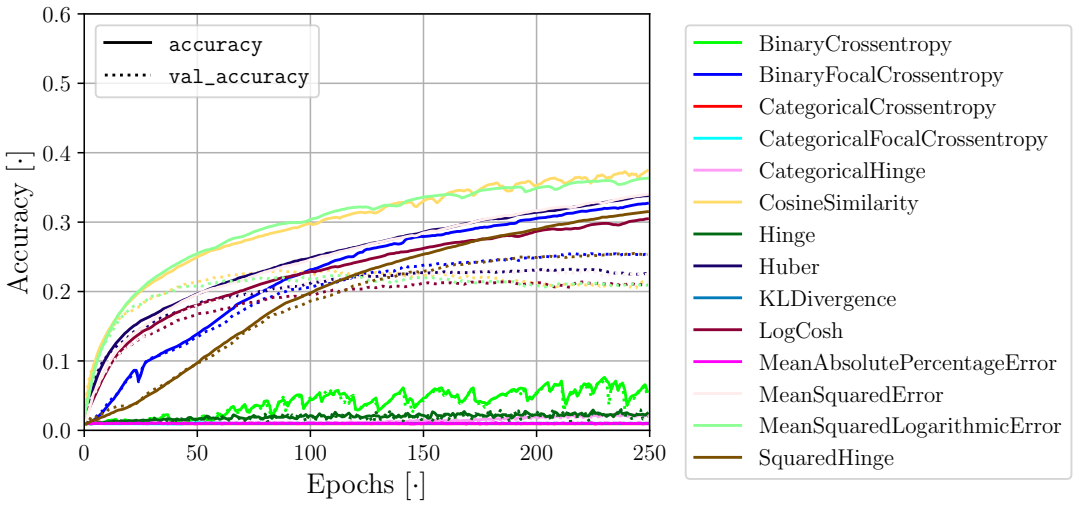

Figure 5.6: (Copy of Figure B.8) The accuracy as a function of epochs, training the same model on the same dataset using different loss functions. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

The *optimisation algorithm* and *loss function* do not change the architecture of the model, but rather how it is *trained*. It should be possible to tune the architecture of the model to reduce the amount of trainable parameters, with a disproportionate affect on its accuracy, and for that purpose, the model architecture was also investigated.

The **depth** of the was tested by adding/removing layers from the base model. Out of the five models, the base model had the highest accuracy and the second-most accurate model had an increase in total number of parameters. The depth of the model was chosen to remain unchanged. The **width** of the model, however, was also tested and an excerpt of the tested models can be seen in Figure 5.7. The `base_model` is the starting point, however, all the other models have fewer parameters, which can be seen in Table 5.2.

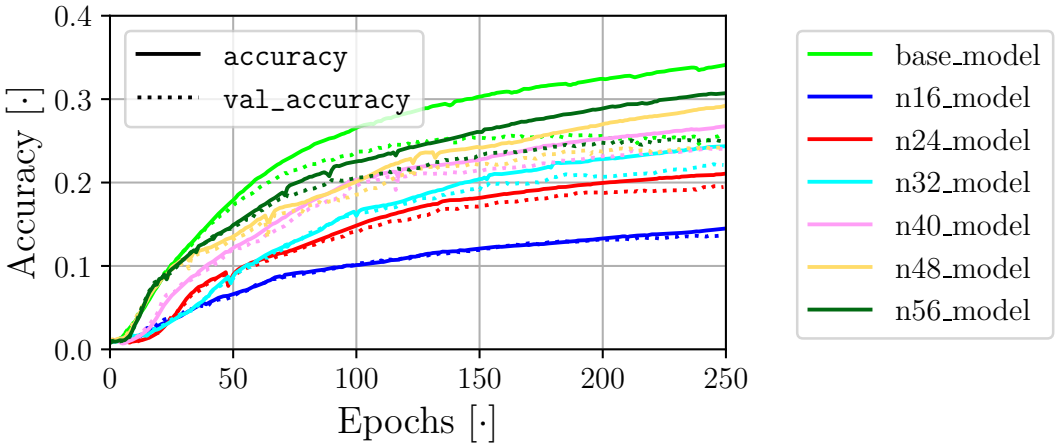

Figure 5.7: (Copy of Figure B.11) Accuracy as a funciton of epochs, training models with varying *depth*. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

From Table 5.2 and Figure 5.7 it can be seen that `n40_model` has nearly the same testing accuracy as the `base_model`, `n48_model`, and `n56_model`, however, the total amount of parameters are half of the `Base Model`. Furthermore, the `n40_model` is just as fast per epoch as the `base_model`, and significantly faster than both `n48_model` and `n56_model`.

Table 5.2: (Copy of Table B.10) Base model and the model with fewer parameters. For each model the number of parameters and the mean time per epoch are listed.

Name	Total Params #	Time [s]
`base_model`	47812	1,61
`n16_model`	4900	1,05
`n24_model`	9604	1,16
`n32_model`	15844	1,40
`n40_model`	23620	1,60
`n48_model`	32932	1,84
`n56_model`	43780	2,15

Increasing the width yielded no increase in accuracy, however, a smaller model retained a good testing accuracy with a significant decrease in the number of parameters: `n40_model` with only 23620 parameters.

5.1.3 Implementation Phase

The purpose of the developed small-scale CNN is to have a reference system of low complexity, that can be *implemented* with approximate arithmetic and used to test/research the influence of approximate computing on neural networks. In subsection 5.1.2 the ''final model'' (`n40_model`) was chosen, however, the accuracy of this model only reaches around 25 %. Approximate computing techniques are appropriate to implement in error-tolerant scenarios, however, with this relatively low accuracy it may be fallacious to say that the model is error-tolerant given the problem of image classification on CIFAR-100. However, as previously mentioned, CIFAR-100 was chosen so that the number of classes could be adjusted to accommodate different levels of difficulty. In Appendix B under subsection B.6.2 the number of classes was adjusted to ensure a ~ 50 % accuracy for the model to attain more leverage. Furthermore, in the same test, the effect of having bias was tested and an excerpt of the accuracies can be seen in Figure 5.8.

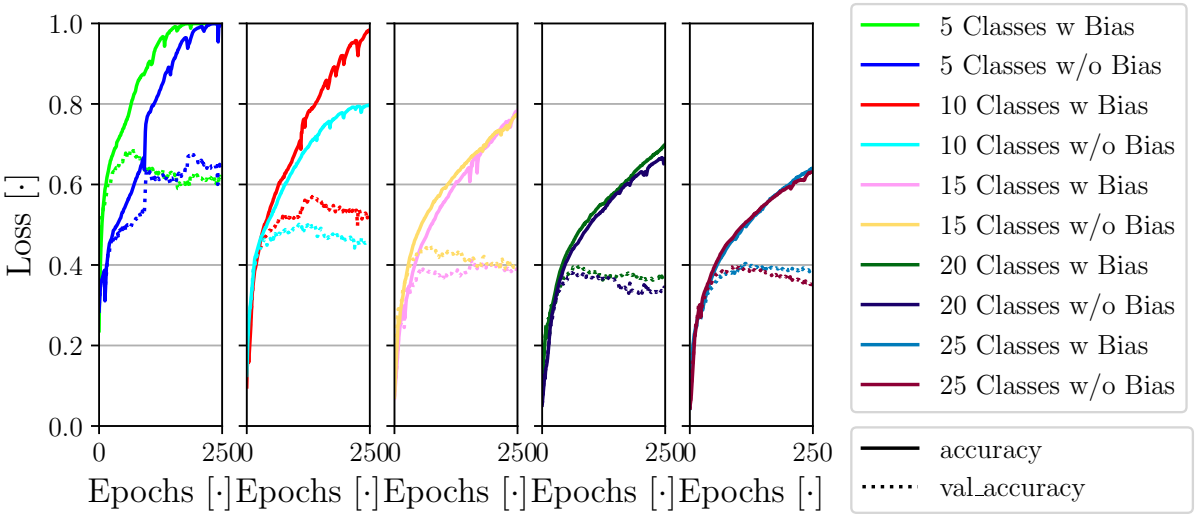

Figure 5.8: (Copy of Figure B.12) Accuracy and validation accuracy plotted as a function of number of epochs in the interval [0, 250]. From left to right, the number of classes to perform classification on is incremented by 5, and each combination is tried with and without bias.

The difference between using bias and not using bias is noted as mostly insignificant in the appendix, however, for 5, 10, 15, and 60 classes there are differences (the accuracy with 60 classes can be found in Figure B.12). For 5 classes the model is trained faster with bias, but they reach around the same maximum validation accuracy. For 10 classes using bias outperforms not using bias. For 15 and 60 classes not using bias outperforms using bias. To simplify the implementation of the manual implementation of the small-scale network, the bias weights are discarded going further. This also reduces the total number of parameters.

Due to the goal of 50 % accuracy the model will perform classification on **10 classes** (see Figure 5.8). It is noticed, that the accuracies are very ragged and there are clear signs of overtraining, both of which are unwanted in the model. *Regularisation* is seen as a method to minimise the effects of overtraining and potentially ''smooth'' out the accuracies. L1, L2, and L1L2 regularisation with a set of λ-values have been tested in subsection B.6.2, where L2 regularisation was deemed effective. To optimise the λ-value for L2 regularisation another search was performed, and the results can be seen in Figure 5.9

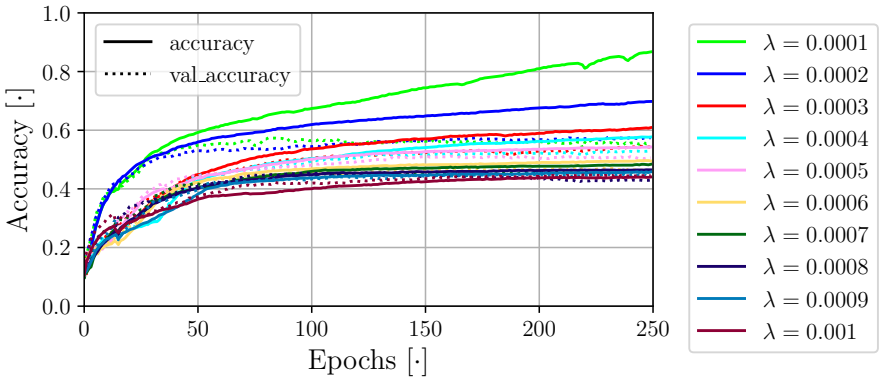

Figure 5.9: (Copy of Figure B.19) **L2** regularisation with adjusted λ-values.

From Figure 5.9 it is clear that the values 0.0001 and 0.0002 are the best performing of the tested values. After 25 epochs, $\lambda = 0.0001$ reaches a higher accuracy than $\lambda = 0.0002$, however, $\lambda = 0.0002$ overtakes at around 125 epochs. Furthermore, both `accuracy` and `val_accuracy` are notably less ragged for $\lambda = 0.0002$, why this value is chosen.

The final model is summarised. The task of the model is to perform *image classification* on 10 classes of the *CIFAR-100 dataset*. The dataset has been ''simplified'' by converting the images to *gray-scale* by calculating the mean over the RGB-channels. Furthermore, the (32×32) images have been resized to 16×16 using the *LANCZOS3* method. Optimisation of the model is performed using the `adamax` *algorithm* and the loss is calculated as `BinaryFocalCrossentropy` with $\lambda = 0.0002$ L2 regularisation. The structure of the model is shown in Table 5.3, the kernels of the convolutional and pooling layers are all 2×2, and there are no bias weights. In section B.9 the methods are briefly described.

Table 5.3: (Identical to Table B.14) Summary of the final small-scale model (`n40 Model`) found by calling `model.Summary()`.

Layer Type	Output Shape	Params #†
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 40)	6400
MaxPooling 2D	(None, 3, 3, 40)	0
Conv2D	(None, 2, 2, 40)	6400
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

† Total parameters 19800 and trainable parameters 19800.

5.2 *Approximate Model* - Approximate Forwardpass in a Convolutional Neural Network

The reference system provided a model (i.e. Table 5.3) which solves the image classification on a reduced CIFAR-100 data set. The purpose of designing the reference system was threefold; to provide the architecture and design of the CNNs, to provide an accurate network to employ the statistical models derived from the approximate arithmetic circuits presented in section 4.3, and to potentially provide the trained kernels and biases to an approximate twin of the accurate network. This section revolves around designing a simulation of the approximate twin. The approximate twin should provide an option to compare the statistical model with the actual approximate distribution in the network.

As mentioned in chapter 3, the benchmarking system should investigate the effects of approximation in the inference stage of the designed CNN through simulation, i.e. before hardware implementation. The design presented in the previous section is reused and methods to simulate approximate computing techniques is implemented, specifically precision scaling and approximate arithmetic circuits.

In short, the simulation implementation should allow the user to provide an approximate arithmetic circuit and a FXP Q-format compatible with the design of the approximate circuits. It is chosen to implement the *approximate model* in C++ since the authors are more familiar with memory and datatype management compared to `python`. A block diagram illustrating the relationship/ interactions of the reference TensorFlow model and the approximate C++ model is shown in Figure 5.10.

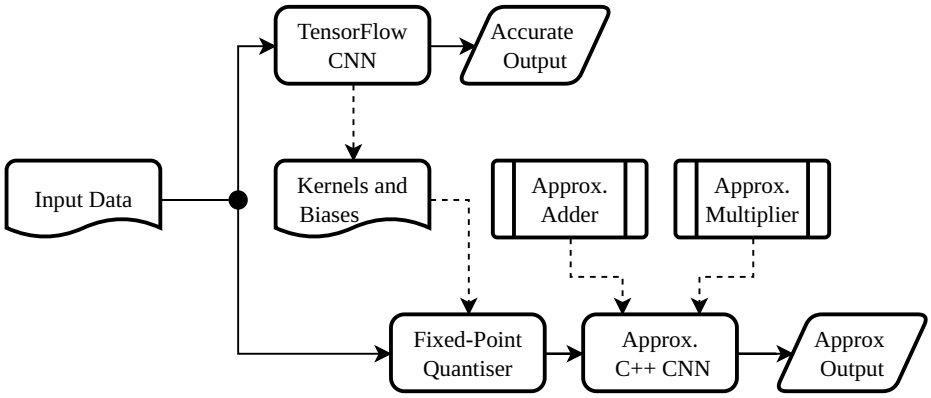

Figure 5.10: Illustration of the relationship between the reference TensorFlow model and the approximate CNN. The two networks have identical architecture and are presented with the same inputs. The reference system writes its kernels to a document, which the approximate model then adopts. Two predefined processes are adopted, emulating approximate arithmetic units (i.e. approximate adder/multiplier). The dotted lines represent the data flow that constitutes the interactions of the two CNNs.

The approach for implementing the CNN model in C++ is to develop classes for the convolutional, pooling, and dense layers in the model, and then create objects of these classes corresponding to the model from Table 5.3. The perceptron is also designed as a class since these are used both for convolutional and dense layers. This property also enables evaluating a model with **different** adders or multipliers for the various layers. A class diagram of the C++ implementation is seen in Figure 5.11.

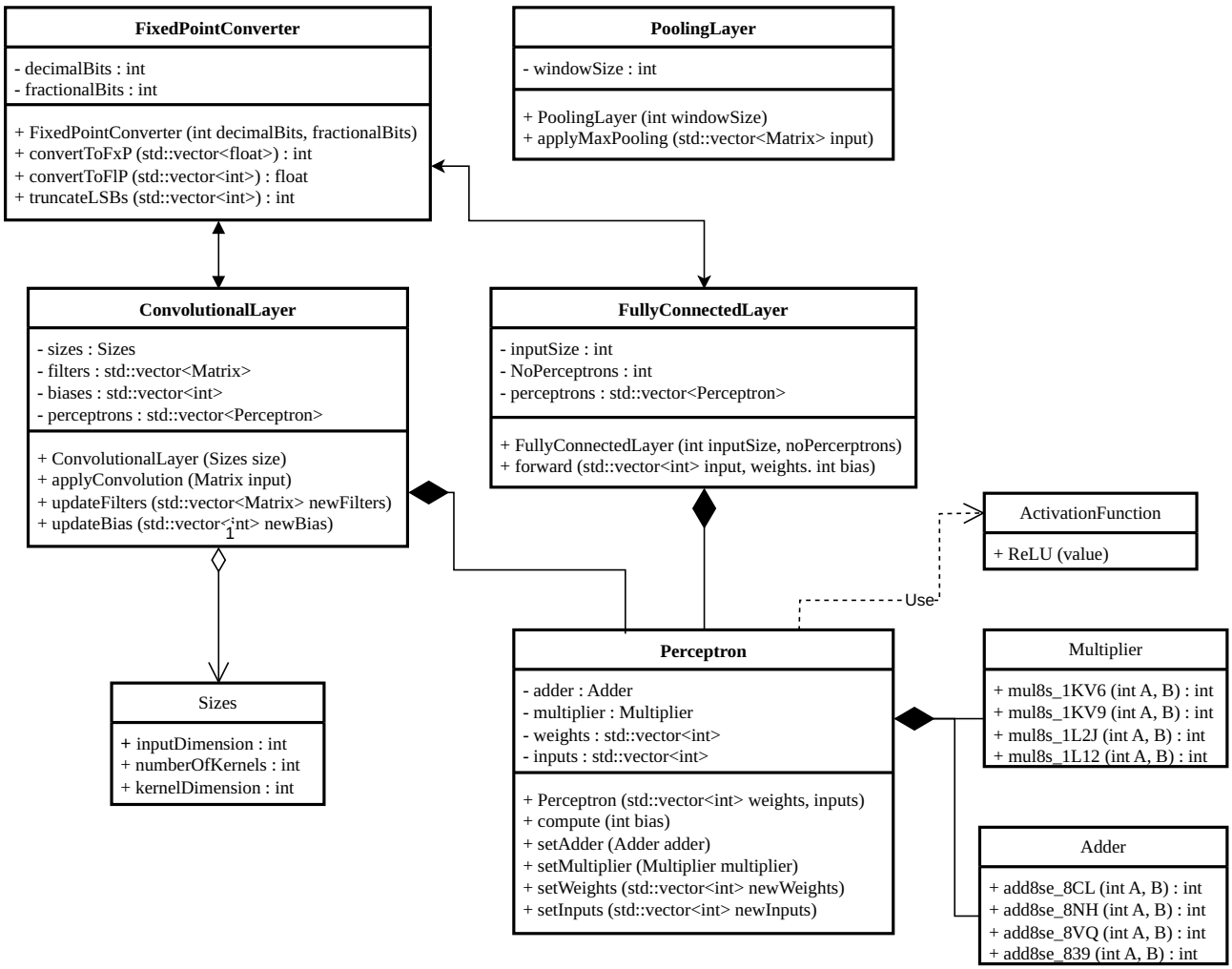

Figure 5.11: Class diagram of the implementation of the reference CNN in C++. The structured representation shows the modular configuration of the different layers in the implementation. The connections with a filled diamond denote a composition relationship, dotted arrows denote dependency and solid arrows denote association. Each class's methods and attributes are denoted, and the naming scheme should describe functionality.

The class diagram shows the fundamental building blocks of the CNN, represented as classes for simple modification. The convolutional and fully connected layers are the *''computational''* classes and are composed of objects of the perceptron class. The fact that both layers are constructed using objects of the same class is due to a requirement for modularity of the arithmetic operations, hence the perceptron class is composed of approximate adders and multipliers. The perceptron class depend on the activation function class, which in this implementation only holds the ReLU method.

The FXP-converter class is auxiliary to the remaining classes which all utilise FXP number representation, hence the conversion methods native to objects of this class is necessary, for FLP inputs and outputs.

The pooling layers are only implemented with a method for max-pooling and are associated with any of the remaining classes.

The specific implementation of the CNN from Table 5.3 is presented in the object diagram from Figure 5.12.

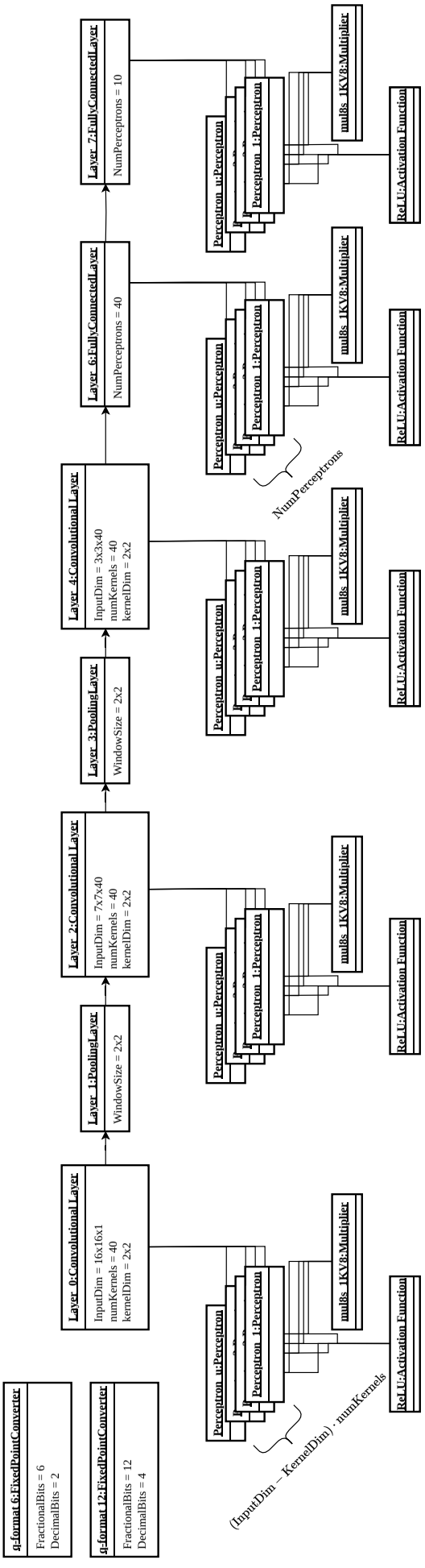

Figure 5.12: Object diagram of the CNN model presented in Table 5.3. The objects are instantiations of the classes presented in Figure 5.11. The connections in this object diagram illustrate a composite relationship between objects. The arrows indicate the input-output relationship between objects.

From this object diagram, it is noticed that each convolutional and fully connected layers contain objects for each resulting perceptron. The constructors of the classes are responsible for assigning the appropriate weights and inputs to the perceptron objects native to the class. This means that from a user's perspective, it is only necessary to specify the architecture of a desired CNN, the weights of each layer, and which approximate multiplier to use to comprise an approximate simulation. It is further noticed that the choice of approximate arithmetic can be different from layer to layer. This is advantageous if one layer contains deep perceptrons of many MAC-operations and more accurate approximations are desired, compared to simple layers. However, for all implementations in this project, the multipliers are kept congruent for all perceptrons.

A thorough description of each method and attribute is omitted from this presentation as this is considered beyond the scope of this introductory overview, which aims to provide a high-level understanding of the key concepts and functionalities. However, the description of the convolutional layers and the FXP-converter are presented in the following subsections as these are considered non-trivial. The scripts, files, and classes for the C++ implementation can be found in GitHub through Appendix A under the directory `/Perceptron/`.

5.2.1 Convolutional Layers

The convolutional layer gets the weights from the TensorFlow model. The weights in TensorFlow are represented in a 4-dimensional structure of the size (filter_height × filter_width × input_channels × filters). The implementation loops over the `input_channels` when convolving the input and weights as presented in Figure 5.13. Given an input consisting of j channels to a layer of k filters, j structures of k kernels are constructed. From the first structure, the first kernel is convolved with the first input channel, the first kernel from the second structure is convolved with the second input channel, etc. the outcome of which is a 3-dimensional structure, with the height and width of (input heigth − kernel heigth + 1) and (input width − kernel width + 1), respectively, and the depth is j. This structure is summed over the 3rd axis, shrinking the depth to 1, and resulting in the first FM (Feature Map). This process is repeated k times utilising different kernels, resulting in k FMs that are put together as the output.

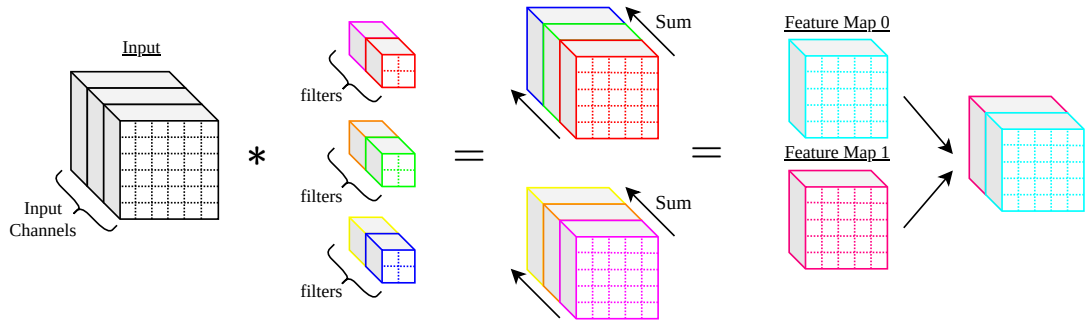

Figure 5.13: Implementation of convolution in the *approximate model*. An input is convolved with as many kernel-structures as input channels.

This implementation is based on purely the structure of the weights and the visualisation is not intuitive. However, another equally valid method would be to loop over the filters, rather than the input channels. The visualisation is presented in Figure 5.14. Each filter has the size filter heigth × filter width × input dimensions, and the convolution can be seen as can be seen as taking the first filter and a slice of the input of equal size, compute the entry-wise multiplication and sum the products, yielding one pixel of the feature map. This is repeated for all ''slices'' of the input, with overlap, and 1 filter results in 1 feature map. After this has been done using all filters, the resulting feature maps can be stacked, comprising the output of the layer.

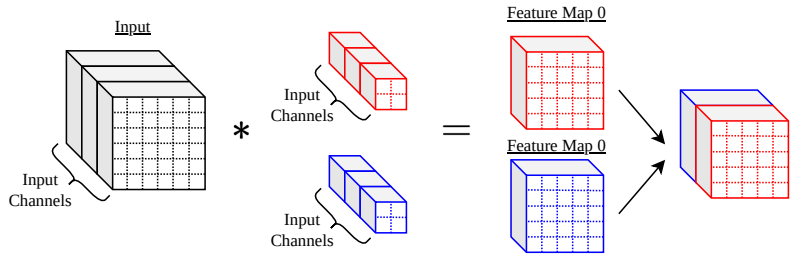

Figure 5.14: Alternative convolution method. A filter is entry-wise multiplied with an equally sized slice of the input and the products are summed to get the value of **one pixel**. This is performed for all possible slices, resulting in a feature map. One feature is thus computed per filter, and the feature maps are stacked, comprising the output of the convolutional layer.

Although the two methods are equally valid, the method in Figure 5.14 is presented, as it is more intuitive, and will be used again during the development of the *probabilistic model*.

5.2.2 Fixed-Point Precision Scaling

The kernels and input images of the CNN are provided by the TensorFlow implementation as illustrated in Figure 5.10. However, there is a clear mismatch between the `float32` representation of the weights and the FXP (fixed point) representation required for the approximate circuits. A class is implemented, where converters can be instantiated to process values from/to `float32`. The class must be aware of which FXP format the value should be converted to, i.e. the converter is constructed with a Q-format, reserving a specific number of bits to represent the integer-value of the number-to-be-converted, and reserving a specific number of bits to represent the fractional-values.

Converting from `float32` to `Qn.m` requires utilisation of the `int`-types in C++. There is no type for interpreting Q-format fixed-point values, however, as seen in Eq. (2.14) and (2.15) Q-formats are essentially a scaled version of `uints` and `ints`, respectively. This scaling factor, 2^{-m}, can be used to go between the `int`-types and `float32`, by scaling the `float32` value with the reciprocal of the scaling factor. The product of a `float32` and the reciprocal scaling factor is not guarenteed to be an integer value, why truncation is performed, and the operation is thus non-reversible. In Example 5.2.1 an example of this conversion is presented.

Example 5.2.1: Converting a `float32` to an `int` in Q2.6 format

Say a weight, $a = 1.6453$ has to be represented in signed Q2.6 format. First, the value is scaled:

$$a_{\text{scaled}} = a \cdot 2^m = 1.6453 \cdot 2^6 = 105.2992$$

This value can not be represented as an integer, however, by casting this value to an `int` in C++, truncation is automatically applied, and $a_{\text{scaled}} = 105$. In two's complement, this value as an 8-bit signed integer is 0110 1001:

0	1	1	0	1	0	0	1

Examining this 8-bit signed integer as a signed Q2.6, the decimal value represented can be found: One sign-bit ▫, one bit for integer values ▫, and 6 bits for the fractional value ▫, yields:

$$a_{\text{Q2.6}} = -\boxed{0} \cdot 2^1 + \boxed{1} \cdot 2^0 + \boxed{1} \cdot 2^{-1} + \boxed{0} \cdot 2^{-2} + \boxed{1} \cdot 2^{-3} + \boxed{0} \cdot 2^{-4} + \boxed{0} \cdot 2^{-5} + \boxed{1} \cdot 2^{-6} = 1.640625$$

$a_{\text{Q2.6}}$ is a bit smaller than the original value for a, however, that is expected due to the truncation.

Converting from `Qn.m` to `float32`, can be performed by scaling the `int`s from C++ with the factor 2^{-m}. However, it is not possible to multiply an `int` with a fractional value, the `int` is cast to a `float32` just before the scaling is applied.

Truncation of the Product is another method defined for the converter class. The product of two values in `Qn.m` is represented in `Q2n.2m` format. Say the product also has to be passed through an adder, which expects a value in the format `Qn.m`. These formats are incongruent and some of the bits must be removed. Although the implementation allows for defining the Q-formats for each layer/perceptron, one Q-format is applied globally. This means, that the product of two values in `Qn.m`, which should be represented in `Q2n.2m`, will also be represented in `Qn.m`, i.e. n integer bits are removed (from the left) and m fractional bits are removed (from the right). There is an innate risk of overflow associated with this approach, however, in the `exact model` the weights are found using L2 regularisation, and seemingly not too large.

5.3 *Probabilistic Model* - Modelling Errors in Forward Propagation

The goal of **step II** of the benchmarking system is to evaluate the user-provided approximate arithmetic circuits in a CNN which have been developed in the previous sections. The purpose is to develop a probabilistic model for the errors introduced by the approximate arithmetic circuits applied to a copy of the TensorFlow model derived in section 5.1. The *probabilistic model* will then be evaluated using a statistical test using the C++ simulation as samples of observations.

Step II of the benchmarking system is essential as the probabilistic model of the approximate arithmetic circuits can be tested on the reference system ensuring *generalisation*, before applying the same modeling principles in a **scaled** model in step three.

In regards to modeling the effects of employing approximate arithmetic circuits in a user-specified CNN, a decision is to be made for the interface between modeling and simulation. section 5.2 provides an option for simulating the effect on the CNN derived in section 5.1. However, this simulated error in prediction accuracy on the CIFAR-100 reduced data set is, doubtless not generaliseable to other CNNs with different applications. Oppositely, modeling the input/output relationship of the CNN derived in section 5.1 given an approximate circuit will neither suffice, as the model once again becomes application specific. The strategy chosen in this project is to derive a probabilistic model for the output of a perceptron, as a function of the chosen circuit and the length of the vector of weights. This solution should provide a model that can be sampled and added as noise for the perceptrons used in step 3, regardless of the network architecture and application.

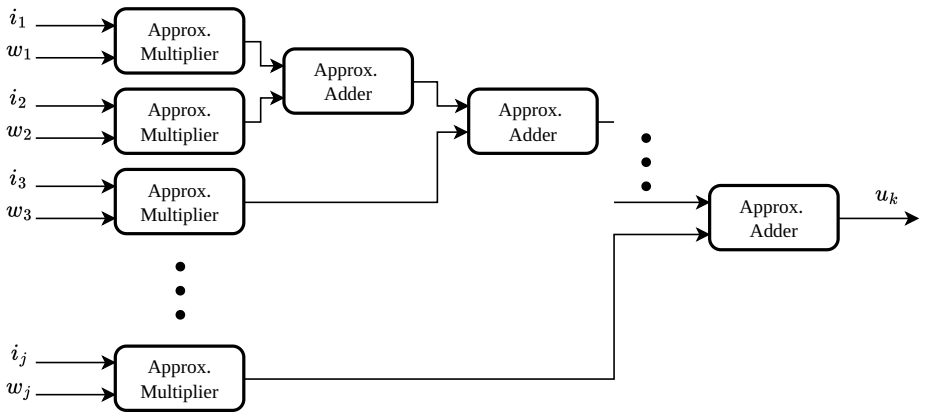

Figure 5.15: Signal flow diagram of a perceptron with j inputs and weights using approximate arithmetic circuits. This figure should illustrate that an additional input/weight combination only appends one approximate adder and multiplier.

A perceptron can be modeled as a series of MAC-operations. The length of this series is determined solely by the amount of weights within a perceptron/filter in the CNN. This point is illustrated in Figure 5.15. This perceptron design is congruent for both fully connected and convolutional layers, further assuring the scaling and generalising ability of the probabilistic model in step 3.

5.3.1 Modelling Errors of Approximate MAC-operations

The principle of the probabilistic modeling is to describe the **error** of the result of the MAC-operations as an RV, with a PDF (Probability Density Function) that can be sampled to emulate the actual error for the approximate circuits. It is noticed that in addition to the error introduced by the approximate circuits, a *truncation* error is introduced by the necessary quantisation and FXP-conversion to the appropriate word length for the given circuit. Initially, it is assumed that all weights and inputs are congruent with the appropriate FXP values, to simplify the analysis.

In section 4.3 the approximate adders and multipliers were simulated assuming the operation's inputs were uniformly distributed throughout the input space. The distribution of error conditioned by the joint distribution of the inputs was found by evaluating the circuits using every element in the input space exactly once.

For a trained CNN the kernels are known in advance, meaning that the error of the approximate **multiplier** is a distribution conditioned on the first input, which is an RV, and the second input which is a constant. The error can then be described as:

$$E = \tilde{P}(I, w) - I \cdot w \tag{5.1}$$

where:

E	The error modelled as an RV
$\tilde{P}(\cdot,\cdot)$	The mapping performed by the approximate circuit
I	The input modelled as a uniform RV distributed over A = Precision range of a B-bit 2's complement fixed point number.
w	The weight of a certain kernel

The distribution of the RV E is described as a conditional PMF, i.e. $p_{E|I,w}(e|i,w)$. This is modelled as a degenerate PMF, as the error is deterministic for a given sample of I, i.e. $P(E = e_{i,w}|I = i, w) = 1$. The error is marginalised over A as the weighted sum of the degenerate PMFs $\forall\, i \in A, I = i$ as shown in Eq. (5.2).

$$p_{E|w}(e|w) = \sum_i \delta(e - e_{i,w}) \cdot P(I = i) \tag{5.2}$$

Eq. (5.2) is obtained empirically by evaluating Eq. (5.1) exactly once for each value of I given a certain w. If an 8-bit multiplier is used this results in a LUT with 256 PMFs, one for each possible weight w.

The scenario is slightly different for the approximate adders as both inputs should be treated as RVs because these are the outputs of the approximate multipliers. The goal is once more to model the error of the output of the adder as an RV which is done for the case where the output of two approximate multipliers is the input to an approximate adder (i.e. a perceptron with two inputs and weights), shown in Eq. (5.3).

$$E_u = \tilde{S}(\bar{I}, \bar{w}) - S \tag{5.3}$$

where:

E_u	The error modelled as an RV
$\tilde{S}$	The mapping from input to output for the perceptron realised by the approximate circuit.
S	The mapping from input to output using accurate MAC operations

A derivation of the error of the perceptron as a random variable is shown in Eq. (5.4).

$$\begin{aligned} \tilde{S} &= \tilde{P}_1 + \tilde{P}_2 + E_{add} \\ S &= P_1 + P_2 = I_1 \cdot w_1 + I_2 \cdot w_2 \\ \tilde{P}_1 &= P_1 + E_{m1} \\ \tilde{P}_2 &= P_2 + E_{m2} \\ &\Downarrow \\ E_u &= E_{m1} + E_{m2} + E_{add} \end{aligned} \tag{5.4}$$

(5.5)

This means that the error of the perceptron can simply be modelled as the sum of the RVs of each arithmetic circuit! The PMF of the error in the approximate multipliers was conditioned on the known weights of the perceptron. For the approximate adders, the inputs are the outputs of the approximate multipliers. Therefore the PMF is modelled as the conditional probability $p_{E_{add}|\tilde{P}_1=p_1,\tilde{P}_2=p_2}(e_{add}|p_1,p_2)$. This is also a degenerate PMF as for the approximate multipliers and the PMF can be marginalised over the inputs $\tilde{P}_1$ and $\tilde{P}_2$ as:

$$p_{E_{add}}(e) = \sum_{p_1}\sum_{p_2} \delta(e - e_{add}) \cdot P(\tilde{P}_1 = p_1) P(\tilde{P}_2 = p_2) \tag{5.6}$$

For the approximate multipliers, the input RV, I, was assumed to be uniformly distributed, making empirically obtaining the PMF a simple task. However, for the approximate adders, the inputs are the approximate products which add complexity to the model, as $P(\tilde{P}_1 = p_1), P(\tilde{P}_2 = p_2)$ are no longer assumed uniform. To empirically obtain the output of the last adder in the series of MAC-operations, requires knowledge of the output distribution of j$^{\text{th}}$ multiplier and the second to last MAC-operation, the latter of which requires the (j-1)$^{\text{th}}$ multiplier and the third to last MAC-operation and so on. Therefore the chronology of the empirical distribution is essential.

Another approach would be to empirically obtain the output of an inner product with a certain approximate multiplier and adder combination, as a function of the number of inputs. This arrives with the implication that the input-output relationship has to be observed ideally for any number of inputs to a perceptron. This approach presents a computational-labour-intensive approach as the expansion of the input and output spaces now needs to be simulated.

It has previously been discussed that the approximation efforts are most effective in multiplication, as these are far more complicated circuits than adders. Therefore it is assumed that multipliers have the greatest interest in being implemented as approximate circuits in the CNN, why approximate adders will be disregarded.

By only considering the multipliers of the accumulation of errors described in Eq. (5.5), the problem becomes a sum of independent RVs. By the CLT (Central Limit Theorem) it is known that the sum (or mean) of a *large* number of I.I.D. (independent and identically distributed) RVs is approximately normally distributed [122][123]. Unfortunately, the outputs are not identically

distributed as different weights result in a specific distribution and the number of independent RVs that are summed in the CNN model is 4 (i.e. in the first Conv2D-layer in Table 5.3). These properties imply that the CLT in the classical sense does not apply. The PMF of the accumulated error (i.e. $p_{E_u}(e_u)$) is calculated by convolving the individual error PMFs of each multiplier since the addition of RVs is the convolution of their PMFs.

The next step is to fit a Gaussian distribution to the obtained PMF. The purpose of making a parametric fit of continuous RV is to accommodate for the truncation errors introduced by the FXP-conversion described in subsection 5.2.2. Truncation introduces an error by removing the LSBs after the desired word length. As both weights and inputs are drawn from continuous distributions, it requires an infinite word length to represent their numbers accurately. The error from the truncation is however bounded by the chosen word length (B) as:

$$e_Q < 2^{-B}$$

The truncation error is assumed to be accounted for by fitting a continuous distribution to the PMF. The chosen continuous distribution is Gaussian as it is assumed to be the best parametric option due to the CLT. The Gaussian distribution is fitted using a MLE (Maximum Likelihood Estimate) . The MLE of a Gaussian is the mean (μ) and standard deviation (σ):

$$\hat{\mu}_u = \sum_{e_u} e_u \cdot p_{E_u}(e_u)$$
$$\hat{\sigma^2}_u = \sum_{e_u} (e_u - \hat{\mu})^2 \cdot p_{E_u}(e_u)$$

The final model is therefore given in Eq. (5.7)

$$\tilde{E_u} \sim \mathcal{N}(\hat{\mu}_u, \hat{\sigma^2}_u) \tag{5.7}$$

Since the focus of this project is delimited to investigating the approximate multiplier's effect on the CNN a probability model is derived for the trained CNN model.

5.3.2 Adding Error to the CNN

The modeled error from Eq. (5.7) is implemented in `python`, to be compatible with the CNN implemented using TensorFlow. The flow of obtaining the model is illustrated in Figure 5.16.

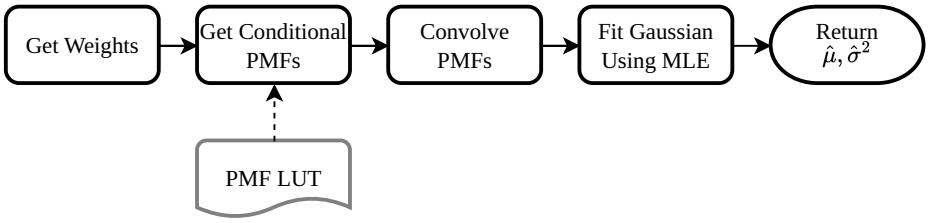

Figure 5.16: Conceptual flow diagram illustrating the methods utilised for obtaining a probabilistic model for each perceptron in the CNN.

The implementation of the probabilistic error modeling is provided in Appendix A under `/statistic_test_3_models/NoisyLayers.py`. The error PMFs are provided in a `.csv` file used as a LUT developed in step one (i.e. section 4.3). The weights are read from the `tf.layer.kernel`, to use as indexing in the LUT. Since the multipliers are multiplying in FXP-format, the FLP-represented

weights are firstly converted by scaling by 2^B where B is the word length of the inputs to the approximate multipliers.

The python library `scipy.stats` is used to provide the MLE using the method scipy.stats.fit(norm,pmf) which provides the MLE parameters given the Gaussian distribution and the error PMF.

To verify the fitted normal distribution a test case is specified using the weights from `./statistic_test_3_models/1KV9_weights/save_45/layer_6/weights.csv` found in Appendix A. This is the weights of the 6^{th} layer in Table 5.3, which is a fully connected layer with 40 perceptrons, meaning that 40 sets of 160 weights are provided. The LUT of the multiplier `mul8s_1KV9` (see Appendix C) is used for this example. The convolved PMF given the weights specific to the first perceptron is plotted along the fitted Gaussian distribution. This is shown in Figure 5.17.

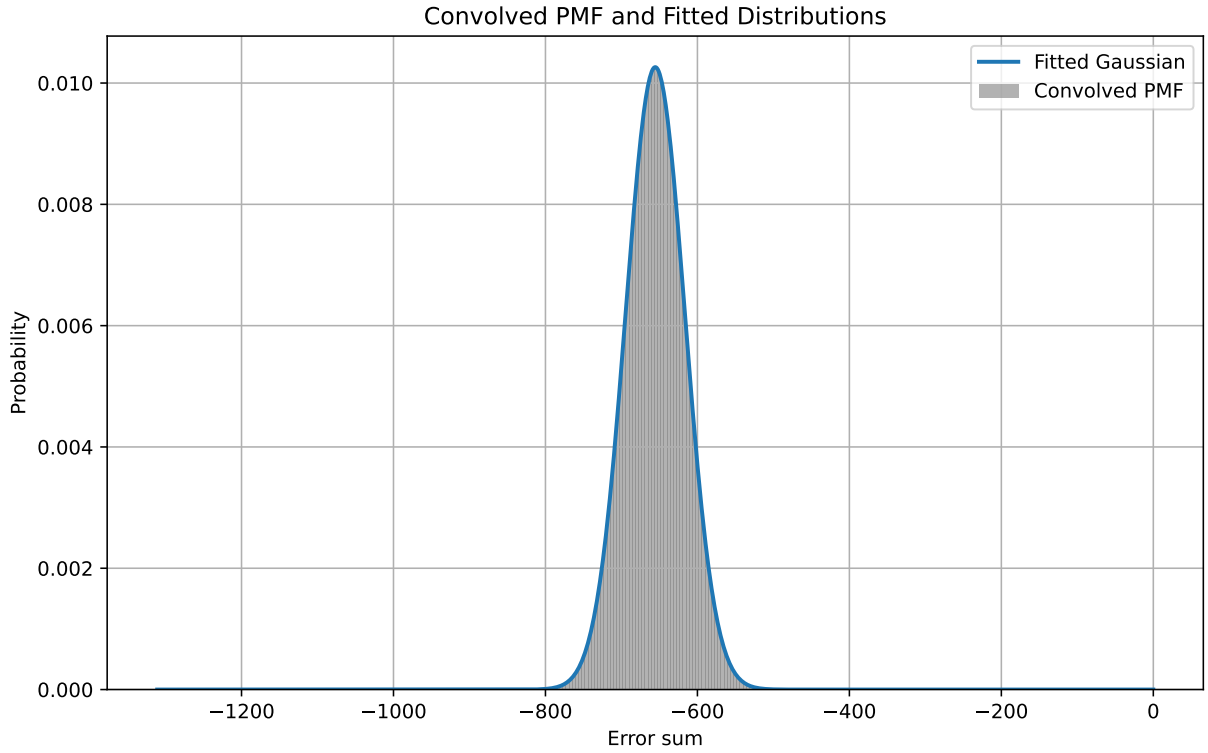

Figure 5.17: The accumulated PMF plotted along the fitted Gaussian distribution. By visual inspection, it is noticed that the fitted Gaussian seems to model the calculated PMF accurately.

Firstly it is noticed that the calculated PMF seems to follow a Gaussian distribution accurately. This is to be expected due to the CLT as previously described. The mean of the fitted Gaussian distribution for these specific weights is −655. As presented in Appendix C the MAE of the `mul8s_1KV9` is 4.25. A quick estimate of the mean of the distribution for a perceptron with 160 weights is $4.25 \cdot 160 = 680$. The cause of the calculated mean of the fit not matching this estimate exactly, is that the individual PMFs given a weight, differ from the total MAE. By inspection of the fitted distribution for the remaining weights shows that the fitted means are distributed around the estimate, hence it is reasoned that the MLE fit of the accumulated PMFs works as it was intended in Eq. (5.7).

Given the ability to generate an *error fit* based on the weights used in a series of MAC-operations, adding the modelled error is possible, it is desired to implement custom classes that can be used as layers in the TensorFlow framework. These noise layers should be general and scalable since it should be possible to reuse them in **step III**, where a user should be able to apply the layers to their models. The noise should be added just after convolution/multiplication, however, usually the

activation function is applied as the last step of a layer. It is possible to apply `None` as the *activation function* and create a separate layer only performing the activation function, however, this would force the user to change their model and add more layers. Instead, a set of dense- and convolutional layers are created based on the TensorFlow implementation of `tf.keras.layers.Dense()` and `tf.keras.layers.Conv2D()`. The custom layers will perform the same computations, include an activation function, and have the possibility of adding noise from a distribution created from the weights of the layer and the error characteristics from the supplied approximate arithmetic circuit. The process of generating the distributions is simplified since the layers will have the weights readily available and be aware of how the output is calculated, i.e. be aware of which weights should be included when generating the distributions.

Dense Layers with Noise - `NoisyDense()`

The dense layer is comprised of a set of **perceptrons**. Each perceptron has a set of weights equal in length to the number of inputs. The inner product of the input and the weights are computed, yielding one value per perceptron. The implementation of the custom noisy dense layer will follow these steps when computing the outputs:

I) Perform the inner product of the inputs and the weights using `tf.matmul()`
II) Generate a tensor of the same size as the output filled with zeros
III) Fit the ''error PMFs'' of the weights to one normal distribution per perceptron
IV) Sample from the error distribution of each perceptron and add the error to the tensor of zeros
V) Sum the result of the inner product with the error
VI) Pass the result through the activation function

Furthermore, to accommodate batches another dimension is added to the output, which now has the shape (batch × perceptrons). This does not change the behaviour of `tf.matmul()`, however, when sampling from the error distribution the number of samples is now equal to the number of inputs.

Convolutional Layers with Noise - `NoisyConv2D()`

The convolutional layer is comprised of **filters**. Each filter has a set of weights, with the shape of input channels × kernel width × kernel height. However, the method of applying error is very similar to the case of dense layers. In Figure 5.18 the method to compute the value of the first ''pixel'' of two feature maps can be seen: A 3-dimensional filter of weights is entry-wise multiplied with a slice of equal size of the input, the result of which is summed to one value. The filter is then reused by taking another slice of the input, multiplying, and summing, resulting in the value of another ''pixel'' in the same output feature map. When adding errors to the computed feature maps, it is worth noting that this means that all entries in a feature map are calculated based on the same filter, i.e. the same set of weights are utilised to generate an entire feature map. All error values for a feature map can thus be sampled from the same distribution.

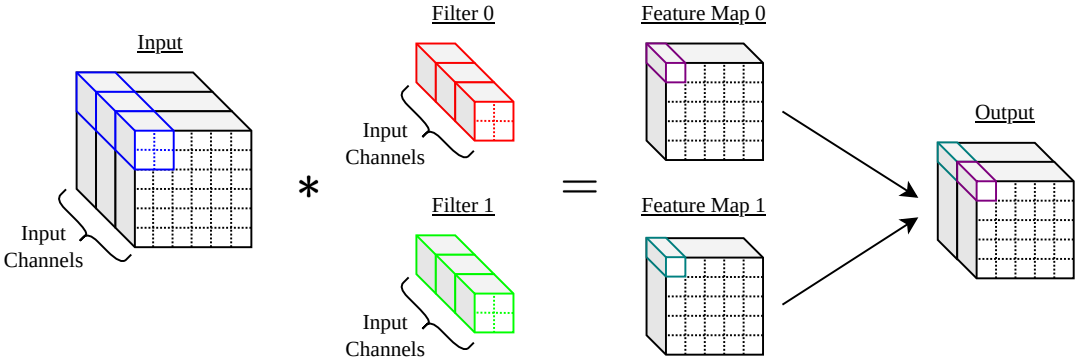

Figure 5.18: Convolution of a 3-dimensional input and two filters.

The implementation of the custom noisy convolutional layer will follow these steps when computing the outputs:

I) Perform the inner product of the inputs and the filters using `tf.matmul()`
II) Generate a tensor of the same size as the output filled with zeros
III) Fetch the weights of a filter and convert them to a list
IV) Fit the ''error PMFs'' of the weights to one normal distribution per filter
V) Sample from the error distribution of each filter and generate a ''feature map of errors''
VI) Sum the feature maps from `tf.matmul()` with the ''feature maps of errors''
VII) Pass the result through the activation function

Furthermore, to accommodate batches another dimension is added to the output, which now has the shape (batch × input width - filter width + 1 × input height - filter height + 1 × filters). This does not change the behaviour of `tf.matmul()`, however, when sampling from the error distributions the number of ''error feature maps'' is now equal to the number of inputs.

Verification of Custom Noise Layers

To ensure that the custom layers are correctly designed, two small tests are performed on each of the layers. The first test will use ''error PMFs'' consisting of pure zeros, thereby sampling noise with 0 mean and 0 variance, and the output of the layer will be compared to an identically configured `tf.keras.layers.Dense()` and `tf.keras.layers.Conv2D()`, respectively. A randomly generated input signal is processed by a custom layer and its TensorFlow counterpart, the output of the custom layer is subtracted from the TensorFlow layer, and all nonzero-values are counted. If there are any nonzero-values, the custom layer has not been implemented correctly. The script for performing this can be found in Appendix A under `/statistic_test_3_models/test_custom_layers.py`. The results show no indications of nonzero-values, i.e. the custom layers produce the same output as the TensorFlow layers, given error distributions with $\mu = 0$ and $\sigma^2 = 0$.

The ''error PMFs'' are processed using the functions and methods from section 5.3. Given they work as intended, it is verified, that ''noise'' is applied in the custom noise layers: Performing the same test as before, but using ''error PMFs'' from an approximate multiplier, `mul8s_1KV8`, (almost) all values should differ and the non-zero count should be high. Modifying the function `test_noisy_layers()` from Appendix A under `/statistic_test_3_models/test_custom_layers.py` to read the ''error PMFs'' from `mul8s_1KV8` yields a high non-zeros count, which is interpreted as *the noise is added.*

5.4 Training the CNN with Approximate Arithmetic

To avoid ambiguity the meaning of *training the CNN with approximate arithmetic* is clarified: In this section the problem regarding training/optimisation of a CNN, which utilises approximate arithmetic operations to perform the MAC operations required for a neural network. The alternative training approach would be a training of a CNN using the probabilistically modelled error, which is not the focus of this section.

The *exact model* is a CNN using standard layers, optimisation algorithm, etc. from the TensorFlow API (configured in section 5.1). The *approximate model* is architecturally identical to the TensorFlow model, however, it is developed in C++ and it is possible to insert the bitwise calculations that would comprise an approximate arithmetic adder/multiplier (developed in section 5.2). The weights are shared between the two models, however, since the calculations in the C++ models can be approximations, it is necessary to utilise the STE as opposed to developing, testing, and implementing a general method for performing gradient descent on the C++ model with approximate

arithmetic in place. The STE has been effective for AxDNN [29] and ProxSim [30]: The gradient of a non-differentiable function $\tilde{f}(x)$ is substituted with the gradient of a related differentiable function $f(x)$:

$$\frac{\partial \tilde{f}(x)}{\partial x} \rightarrow \frac{\partial f(x)}{\partial x} \tag{5.8}$$

With this estimator, training the two networks should be identical, since they share the weights and the architecture. This means that it should be possible to train the models using all the built-in tools for gradient descent from the TensorFlow API.

In Appendix D the process of defining a method for training a network with approximate arithmetic operations was undertaken. The following section will highlight the most relevant points from the appendix. Firstly, some terms that are going to be used are defined:

- **Exact Epoch:** An epoch of training the models only using the TensorFlow model. The flowchart can be seen in Figure D.1; a batch is retrieved from the training set; forward passed through the TensorFlow model; the predictions are compared to the labels and a loss is calculated; the gradients are evaluated based on the calculations performed in the model and the loss[3]; the weights are updated based on the gradients; another batch is processed. This goes on until the entire training set has been processed: *One epoch.*

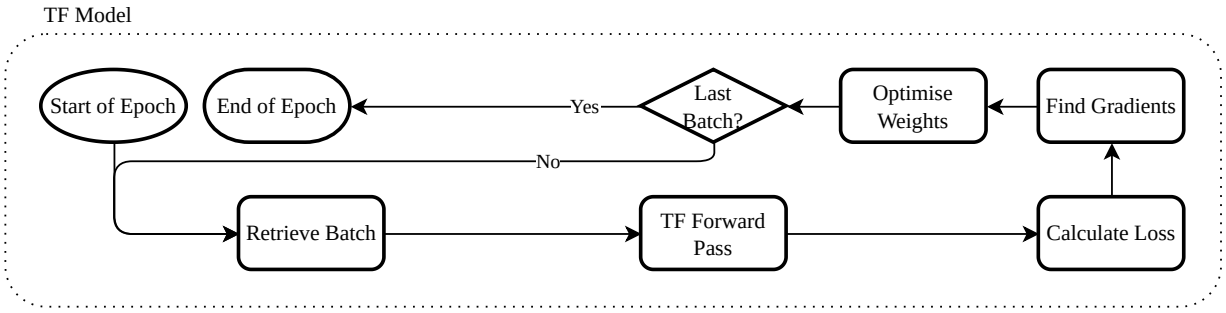

Figure 5.19: (Copy of Figure D.1) Flowchart of an *exact epoch* of training the model. A batch from the training set is propagated forward in the TensorFlow model and the loss is calculated. The gradients are evaluated and used to optimise the weights. This process is repeated until the entire training dataset has been processed.

- **Approximate Epoch:** An epoch of training the models using the C++ model's prediction combined with the optimisation tools available for the TensorFlow model. The flowchart can be seen in Figure D.2; the TensorFlow model's weights are exported to CSV files; a batch is retrieved from the training set and exported to another CSV file; the C++ model is called as an executable file, wherewith the weights and batch are read from the CSV files and the predictions are exported to another CSV file; the TensorFlow model performs also performs the forward pass, which is required for the automatic differentiation; the predictions from the TensorFlow model are changed to match the predictions from the C++ model; the gradients are evaluated based on the calculations performed in the model and the loss; the weights are updated based on the gradients; another batch is processed. This goes on until the entire training set has been processed: *One epoch.*

[3]TensorFlow has a tool for automatic differentiation called `GradientTape()`.

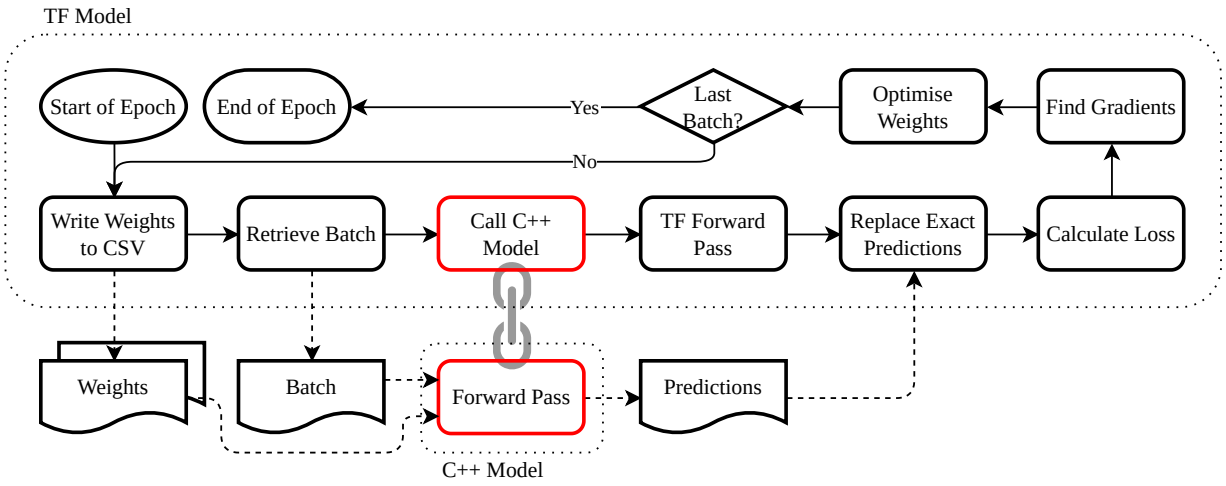

Figure 5.20: (Copy of Figure D.2) Flowchart of an *approximate epoch* of training the model. The weights of the TensorFlow are exported to a CSV file. A batch from the trainings dataset is retrieved and exported to a CSV file. The C++ model is called, wherein the weights and batch are read into the program and forward propagated. A forward propagation is also performed in the TensorFlow model. The predictions from the C++ model replaces the prediction from the TensorFlow model, wherewith the loss is calculated, the gradients are found, and the weights are optimised. This process is repeated until the entire training dataset has been processed.

An investigation of the three scenarios was proposed for the models developed in section 5.1 and section 5.2; training and evaluating the C++ model and the TensorFlow model three times:

I) 50 *approximate epochs*
II) 50 *exact epochs*
III) 45 *exact epochs* followed by 5 *approximate epochs*

To ensure that the integration of *approximate/exact epochs* performs optimisation as intended, the chosen approximation for the preliminary training runs is conversion to FXP (fixed point) with **20 bits for accuracy** and **no approximate multiplier**. In Figure 5.21 the result of each of these training processes can be seen. Note that the values `accuracy` and `val_accuracy` are found using the C++ network: `accuracy` is found by passing the `train dataset` through the C++ network and calculating the rate of correct predictions, `val_accuracy` is found by passing the `test dataset` through the C++ network and calculating the rate of correct predictions.

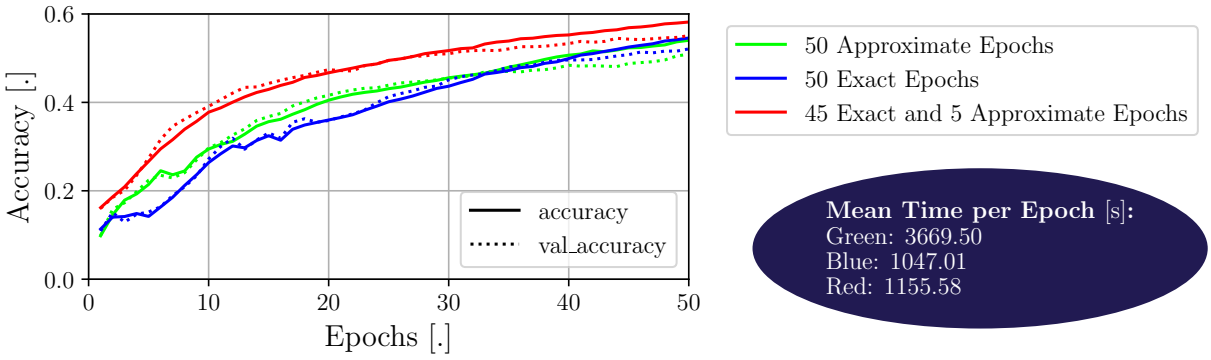

Figure 5.21: (Copy of Figure D.3) Accuracy of C++ network with quantisation noise, 20 bits for precision. Accuracy of C++ network on the train and test datasets: `accuracy` and `val_accuracy`, respectively. Furthermore, the *mean time per epoch* is noted for each of the three trainings.

The run using *45 exact and 5 approximate epochs* is performing better than the other two. For the

training with *50 exact epochs* it should be noted, that the low accuracy must be from an unlucky start (or a lucky start for the *45 exact and 5 approximate* training run) since the training process is identical to that of *45 exact and 5 approximate*; the accuracy of these two training runs should be close to identical in the first 45 epochs. Important for these three runs is that the optimisation method is working when the predictions from the TensorFlow model are replaced with the predictions from the C++ model (see Figure D.2). This suggests that the implemented method for making performing *approximate epochs* is **working as intended**.

The same three methods were tested after lowering the precision of the weights and values to only 8 bits. The results were similar to what can be seen in Figure 5.21, however, 50 *approximate epochs* performs better in this case. However, as presented in the dark-blue ellipsis, the time it takes to perform an *approximate epoch* is significantly longer than performing one *exact epoch*: The run with *50 approximate epochs* took around $3750,03\ \text{s/epoch} \cdot 50\ \text{epochs} \approx 2$ days 4 hours and 5 minutes, whereas the run with *50 exact epochs* only took $1089,44\ \text{s/epoch} \cdot 50\ \text{epochs} \approx 15$ hours and 8 minutes. For that reason, the compromise of using a $45:5$ split of *exact* and *approximate epochs* is chosen as the method for training the C++ network.

Now that the method of training the C++ network is defined, training a network that implements some approximate arithmetic operation is tested. For this purpose the 8-bit multiplier `mul8s_1KV9` from EvoApproxLib|[4] was chosen. The relevant error metrics are presented again in Table 5.4.

Table 5.4: (Copy of Table C.8) Error-metrics of the distribution presented in Figure C.6.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
4.25	34.25	68.75 %	17	1.4

Since the test is essentially creating training two models: An exact model in TensorFlow and a C++ model (first 45 epochs only inference), the accuracies for both are evaluated. The number of precision bits is chosen to be 7. In Figure 5.22 the results can be seen.

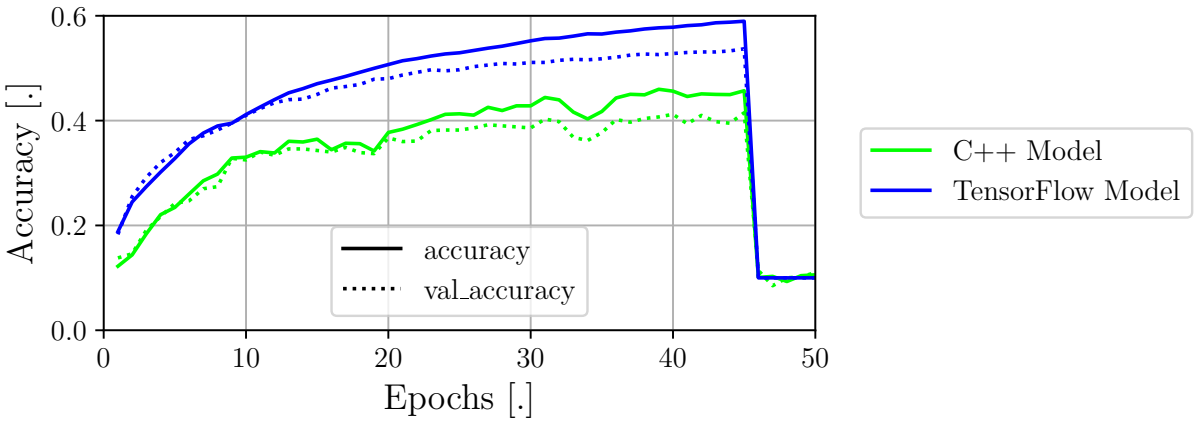

Figure 5.22: (Copy of Figure D.5) Accuracy of C++ network with an approximate multiplier, `mul8s_1KV9`. The first 45 epochs of the training of the C++ network are inference only, why the TensorFlow model is plotted for comparison. The last 5 epochs of the training are with the predictions from the C++ network.

During the 45 *exact epochs* it is clear that the accuracy of the C++ network is steadily increasing. Interestingly, after the 46th epoch (the first *approximate epoch*), the accuracies of the networks fall

to around 0.1, corresponding to random guesses. This puts some doubt into the training process previously devised. To overcome this drastic drop in accuracy three paths are tested:

I) **Training purely with *approximate epochs*:** As shown in Figure 5.21, training using the approximate arithmetic from the beginning may lead to high accuracy. Perhaps, using only *approximate epochs* would prevent the drastic drop in accuracy seen in Figure 5.22.
   - 15 *approximate epochs* with the approximate multiplier in-place were performed, however, the accuracy never rose significantly above 10 %, a random guess.

II) **Pre-training of the TensorFlow model:** In Figure 5.22 the accuracy of the C++ model is increasing alongside the TensorFlow model, and the inference at the 45th epoch is pretty decent. Since the objective of the network is not subject to change, it may be beneficial to have the weights of a pre-trained TensorFlow model, and use the weights as a springboard from which the C++ model can be finetuned. This would reset the training just as the finetuning were to commence since the optimisation algorithm `adamax` would not know the previous gradients.
   - Another 15 *approximate epochs* were performed on a set of pre-trained networks. Again, the results showed no sign of optimisation, as the accuracy stayed at 10 %.

III) **Changing the optimisation algorithm:** A possible explanation for the drastic drop in accuracy in Figure 5.22 is the optimisation algorithm: `adamax`, which has an adaptive learning rate that is based on the first and second moments of previous gradients (see section B.9). The difference in the evaluated loss may be so large it is irreconcilable for the optimiser, and the updated values for the weights are off. At the shift from *exact epochs* to *approximate epochs* the optimiser could be replaced by another, whose learning rate is not adaptive or based on the previous gradients.
   - Unlike the other two methods, the accuracy did not immediately drop to 10 %, rather, the accuracy hovered around 40 %, before dropping off slowly. This might suggest that this is the right path to train the model, but the pre-trained model may be overtrained, the weights may have been a local minimum, or perhaps the learning rate is off.

The optimisation using SGD was further investigated and the resulting accuracies from using the same set of pre-trained weights and adjusting the SGD learning rate can be viewed in Figure 5.23:

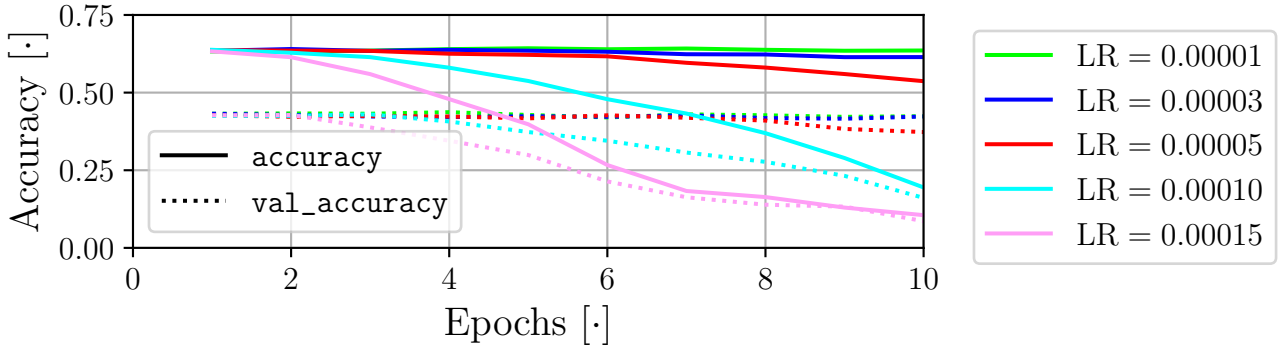

Figure 5.23: (Copy of Figure D.9) Finetuning with different SGD learning rates. 10 *approximate epochs* performed on pre-trained network with varying learning rates.

Every value for the learning rate results in the model ''optimising'' toward a lower accuracy. This is a surprising trend since the optimisation should lower the loss and by extension increase the `accuracy`. This may suggest a fundamental flaw in the implementation of the training. However, the training worked with high precision quantisation and low precision quantisation. No clear path to rectifying the problem of training the approximate network has presented itself.

In order to investigate this problem, it is seen as beneficial to reduce the amount of computations required for a forward pass, in order to speed up the evaluation process, because debugging the implemented CNN is infeasible due to the long duration of each epoch. The slowest layer is the 2nd convolutional layer and the ''complexity'' is reduced to effectivise the debugging process. This is done by reducing the amount of filters from 40 to 2, reducing the amount of parameters from 19.800 to 7.600. This modification is applied to the TensorFlow model as well as the C++ model.

Table 5.5: (Copy of Table D.3) Summary of the smaller network found by calling `model.Summary()`.

Layer Type	Output Shape	Params #[†]
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 2)	320
MaxPooling 2D	(None, 3, 3, 2)	0
Conv2D	(None, 2, 2, 40)	320
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

[†] Total parameters 7600 and trainable parameters 7600.

Firstly, a preliminary test with adamax and SGD was performed with 20 bits precision and an *accurate* multiplier, whereby it was concluded, that *this smaller model is trainable using either optimisation algorithm*. Secondly, two of the SGD learning rates were tested on with an 8×8 accurate multiplier, whereby it was concluded, that *it is possible to finetune with SGD on accurate* 8×8 *multipliers, quantisation with 6 precision bits does not add enough noise to make training impossible*. Perhaps 6 precision bits may also work with adamax; 5 different multipliers were tested and the approximate multipliers with relatively low error characteristics showed positive results, i.e. the *approximate epochs* were effective on the accuracy. These results are interpreted as *the dropoff seen in Figure 5.22 may have been caused by the number of precision bits or the architecture*.

Testing Variations of the Number of Precision Bits on the multiplier used in Figure 5.22, `mul8s_1KV9`, gave an insight into the effects of the chosen bit precision. In Figure 5.24 the resulting accuracies given 45 epochs pre-training, 10 epochs of finetuning with adamax as the optimisation algorithm, and `mul8s_1KV9` with a varying number of precision bits is visualised.

Neither *2 precision bits*, *3 precision bits*, or *4 precision bits* are training and stay at 10 %, the same as a random guess. This is likely a consequence of too much quantisation, as the values of the LSBs are 1/4, 1/8, and 1/16, respectively, which are not matching effectively with the regularised weights. Furthermore, the small changes in the weights at each epoch/iteration may not be transferred to the C++ model, as the changes may not be large enough to ''overcome'' the quantisation. However, for *5 precision bits* and *6 precision bits*, the first epoch of finetuning significantly improves the accuracies, suggesting that the changes are transferred and that they are improving the accuracy of the models.

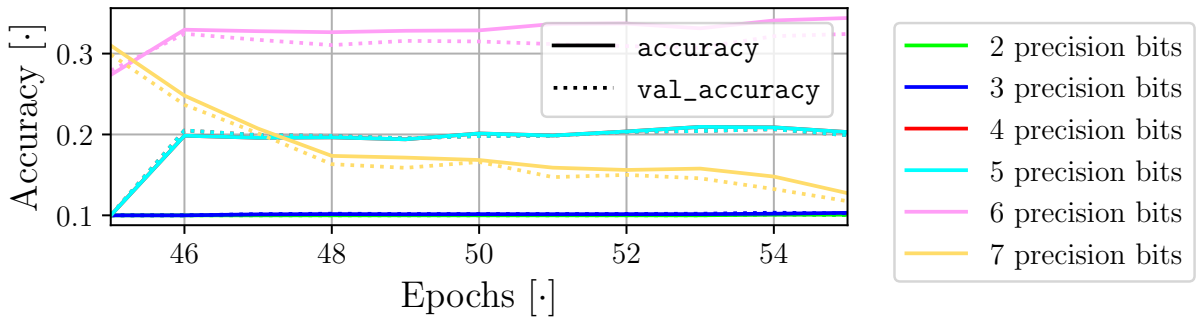

Figure 5.24: (Copy of Figure D.13) Adamax finetuning with varying number of precision bits on `mul8s_1KV9`.

For *7 precision bits* the accuracy worsens at each epoch, this is reminiscent of the training using SGD from Figure D.9, which also was tested with 7 precision bits. The implication of having 7 precision bits in a *signed* 8×8 multiplier, is that the representable values exist between $[-1;\ 1-2^{-7}]$; if the multiplicands are in the said interval no problem should arise. Using regularisation may force the weights into this range, however, the inputs are not taken into account. In the first layer, it is known that the input has been *normalised*, *quantised*, and *truncated*, ensuring they are in the interval. However, the outputs of each hidden layer (being the inputs to the following layer), are not ensured to be in the same interval. Say there are 40 input channels and the kernel size is 2×2, the each ''pixel'' of the resulting FM (Feature Map) is the sum of $2 \cdot 2 \cdot 40 = 160$ multiplications, which may cause overflow/underflow. From Figure 5.24 it is clear, that *it is possible to fintune using adamax* and *given `mul8s_1KV9` and only 2 filters in the second convolutional layer the best choice of precision bits is 6.*

Finetuning with 45 Approximate Epochs using adamax as the optimisation algorithm is performed to ensure that the conclusion from the previous test is correct. Weights from a pre-trained TensorFlow model are applied to the C++ model. 45 *approximate epochs* are then performed, and the inferred accuracy of the C++ network is saved and visualised in Figure 5.25. Furthermore, the accuracy of a TensorFlow model with the same architecture and starting weights trained for 45 epochs and averaged over 5 runs can be seen.

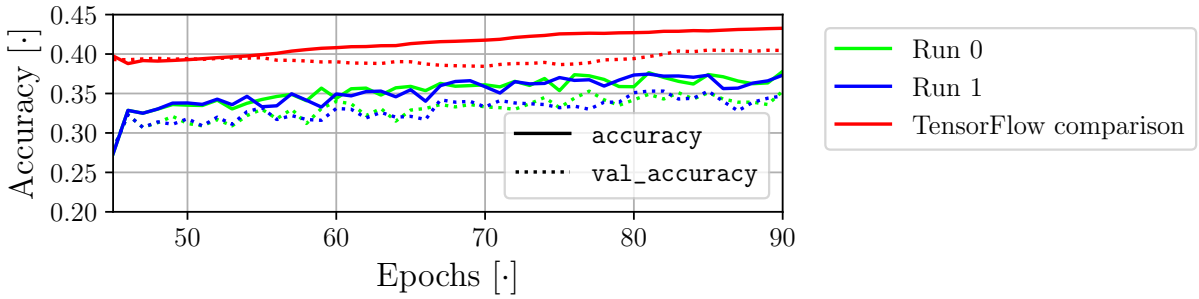

Figure 5.25: (Copy of Figure D.14) Finetuning with 45 *approximate epochs* with `mul8s_1KV9` using 6 bits for precision.

Both runs show that the C++ model with an approximate multiplier is training and the optimisation is working. Comparing the two runs with the TensorFlow model's accuracies the difference is only around 5 percentage points, which is impressive given the approximate multiplier and only 8-bits to represent the weights. The solution was changing the number of precision bits to match with the network. This suggests, that the problem in Figure D.5 may have been caused by overflow/underflow. The solution is to:

- Keep the ‘‘simplified’’ CNN with 2 filters in the second convolutional layer
- Use 6 bits for precision

5.4.1 Considerations/Reflections when Training the *Approximate Model*

The results from finetuning a set of weights from a pre-trained network in Figure 5.25 incidentally simplified the training process. Three things are noteworthy:

I) **Exact epochs** yielded positive results for the C++ model, wherewith the inferred accuracy rose.
II) **Restarting the learning rate of adamax** did not negatively affect the finetuning, i.e. the finetuning can be performed independent of information from previous epochs. Given a set of pre-trained weights, the required amount of epochs could potentially be lowered.
III) **The problem is static** and consecutive uses of the same weights should yield the same results (without training).

These three remarks in combination should allow the simplification of the training process, i.e. given the weights of a trained CNN from TensorFlow, the only training necessary is *finetuning* with *approximate epochs* using adamax as the optimisation algorithm; a TensorFlow model has been trained 45 epochs and the weights have been saved for this purpose. Training the C++ model given some approximate arithmetic circuit, should be possible by taking the pre-trained weights and finetuning for some epochs, using the defined *approximate epochs*.

Although it has been stated that overflow/underflow is the culprit for the dropoff in Figure 5.22 and that it was solved by lowering the number of precision bits, the chosen amount of precision bits is not guaranteed to work for every architecture nor every approximate multiplier. Let’s reexamine the example of potential overflow: *40 input channels and the kernel size is* 2×2*, each “pixel” of the resulting FM (Feature Map) is the sum of* $2 \cdot 2 \cdot 40 = 160$ *multiplications*. The multiplicands are represented with 8 bits, 6 of which are precision bits, i.e. formatted as Q2.6. To ensure no information is lost, the product of the multiplications should be Q4.12, and 160 of these products should be calculated and summed, requiring the format Q11.12 to avoid overflow/underflow. This sum is then processed by the *activation function*, in this case, ReLU, and the format Q11.12 is still required to avoid overflow. This value is then the input of another 8-bit multiplier. The format is incongruent with the multipliers, and a lot of information may be lost. In the implementation, the multiplicands are always transformed to Q2.6 just before the 8×8 approximate multipliers. Overflow is thus still a possibility, however, based on the results of Figure 5.25 the simplified architecture is not as affected by overflow as Figure 5.22 seemingly is.

5.5 Investigation of Congruency Between Probabilistic and Deterministic Modelling

The purpose of this section is to statistically evaluate the fit of the probabilistic model designed and implemented in section 5.3. As described in chapter 3 it is desired to compare the output of a trained CNN (i.e. the CNN architecture from Table 5.3), with the output of the same network using approximate arithmetic units, and using the probabilistic model (i.e. the models designed and implemented in section 5.2, and section 5.3, respectively). The concept is then to apply appropriate statistical evaluations, to determine how well the probabilistic model’s effect on a CNN’s output matches the effects observed from the deterministic simulation of approximate hardware.

A test is conducted in Appendix E wherein two experiments are performed, both associated with the evaluation of the *probabilistic model* in comparison with the *approximate deterministic system*,

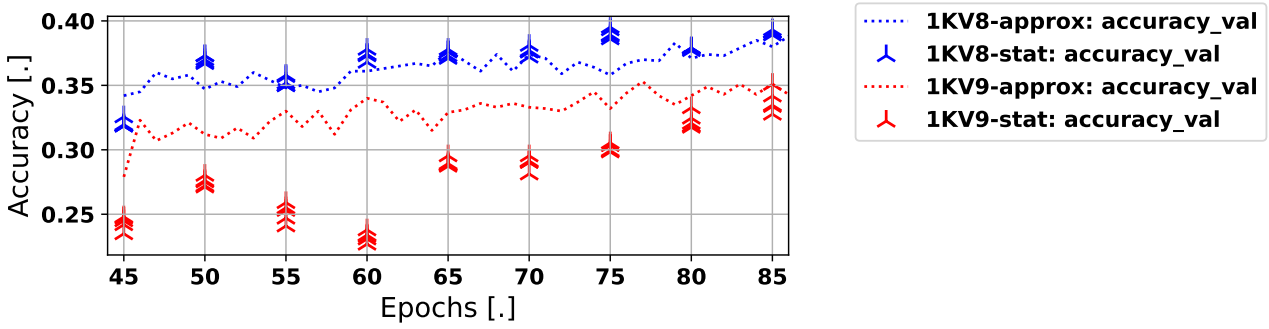

Figure 5.26: Results of the accuracy evaluation on the *test* data set for the `mul8s_1KV8` and `mul8s_1KV9`. The deterministic *approximate model* is plotted as dotted lines and are congruent with the ones in Figure E.3. The accuracy of the *probabilistic model* is plotted using three-pointed stars for every 5 epochs. 5 data points are obtained for the probabilistic model and are all shown in this figure.

which it is attempting to emulate. The purpose is to obtain a metric that can imply ''to which extent this is the case''. The experiment uses the *exact model* derived in section 5.1 as a reference for the *deterministic* and *probabilistic models*.

The first experiment evaluates the prediction accuracy for inference using the CNN presented in Table 5.3. The network trains 45 epochs using exact arithmetic (*exact epochs*), and continues training using STE on the *approximate model* for 40 epochs (*approximate epochs*). The weights are stored every 5 epochs to use for inference on the test data set for both the *approximate* and *probabilistic model*. The results are plotted in Figure 5.26.

From these results, it is noticed that the *probabilistic model* of the `mul8s_1KV8` follows to a considerable extent the accuracy of its deterministic equivalent. The case is different for the *probabilistic model* of `mul8s_1KV9` where the accuracy is considerably lower than its deterministic equivalent. However, it seems that the probabilistic model is ''*catching up*'' as the epochs progress for the STE. An investigation of the relationship between the output vectors is necessary to give an appropriate explanation for this result, which is the essence of the second experiment.

The second experiment uses 1000 input pictures in the CNN. The pixels of these are samples of a continuous RV with a uniform distribution between 0 and 1. A reference test-accuracy is obtained by forward passing in the *exact model*. The same procedure is done for the *approximate model*. The difference between these outputs yields 1000 vectors that are denoted *deterministic error*. The same is done for the outputs of the *probabilistic models*. Since the model is probabilistic the output will be different when applying the same input multiple times, therefore each input is applied 1000 times to mitigate the risk of unfortunate samples. This results in a distribution of 1000000 samples of error vectors. This experiment is conducted for both the `mul8s_1KV8` and `mul8s_1KV9` (with an MAE of 1.2 and 4.25, respectively).

The task is now to determine if the *probabilistic errors* are an appropriate model of the deterministic errors. The Kullback-Liebler divergence is introduced to perform this task. The Kullback-Liebler divergence is a measure of the ''*distance*'' or ''*divergence*'' of two distributions, based on how difficult it is to tell samples of the two distributions apart [124]. Given two distributions, P and Q the KL-divergence is denoted as $D_{\mathrm{KL}}(P||Q)$. P is here interpreted as an observation of a probability distribution and Q is seen as a model of the distribution P. The KL divergence is then intuitively construed as a measure of the information lost when using Q as a model for P. This intuition can directly be applied to the problem at hand, where the *probabilistic model* attempts to model the *approximate* outputs.

When both P and Q are discrete the KL divergence is given as:

$$D(P||Q) = \sum_{x \in \mathcal{X}} P(x) \cdot \ln\left(\frac{P(x)}{Q(x)}\right)$$

The KL-divergence has a closed formula for when both distributions are multivariate Gaussian. This is presented without a proof (see Soch et. al. [125]) in Eq. (5.9).

$$D(P||Q) = \frac{1}{2}\left((\mu_2 - \mu_1)^T \Sigma_2^{-1}(\mu_2 - \mu_1) + \text{tr}(\Sigma_2^{-1}\Sigma_1) - n - \ln\left(\frac{|\Sigma_1|}{|\Sigma_2|}\right)\right) \tag{5.9}$$

where:

P $\sim \mathcal{N}(\mu_1, \Sigma_1)$
Q $\sim \mathcal{N}(\mu_2, \Sigma_2)$
n Dimension of output vector

At first, the `mul8s_1KV8` results are considered. This is done for inference using the weights attained after training the CNN 45 epochs using *exact* arithmetic. The distributions of each of the variables in the output vector are plotted as histograms in Figure 5.27.

From these figures, it is seen that the probabilistic errors are normally distributed, and the deterministic errors can to some degree be approximated as Gaussian. It is chosen to use the closed-form solution to the KL divergence and it has a nice interpretation which can be used to interpret if the difference in mean or covariance is responsible for the divergence. The KL divergence is evaluated to 14.5 for distribution illustrated in Figure 5.27.

This measure by itself shows that the *probabilistic model* is not an exact match of the observed sample. However, this is to be expected from the plots shown in Figure E.6. It is emphasised that the KL measure by itself does not provide much useable information, as the measure should be used for comparisons of methods. For this reason, the same analysis is applied to the remaining inference stages for each additional 5 epochs of STE training, for both multipliers.

In the same manner as the `mul8s_1KV8` the `mul8s_1KV9` is investigated using the KL-divergence measure. From the results presented in Figure 5.26 it is expected to observe a generally higher KL-divergence than for the `mul8s_1KV8` however, it is hypothesised that the measure will decrease with the training on the *approximate model* as a convergence is seen in the classification accuracy.

The `mul8s_1KV8` results are considered. This is done for inference using the weights attained after training the CNN 45 epochs using *exact* arithmetic. The distributions of each of the variables in the output vector are plotted as histograms in Figure 5.28.

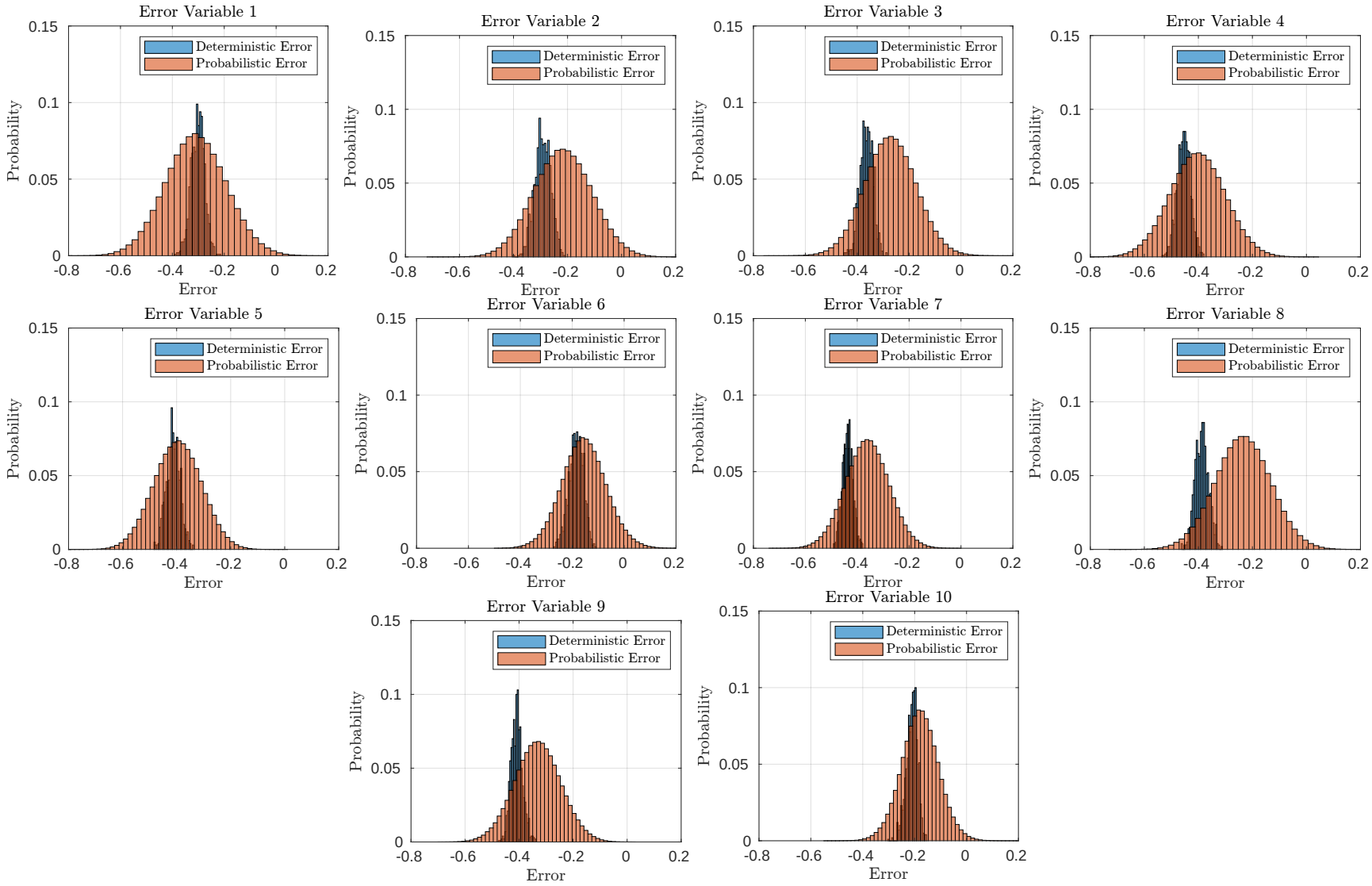

Figure 5.27: Histogram of the deterministic and probabilistic error distributions, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

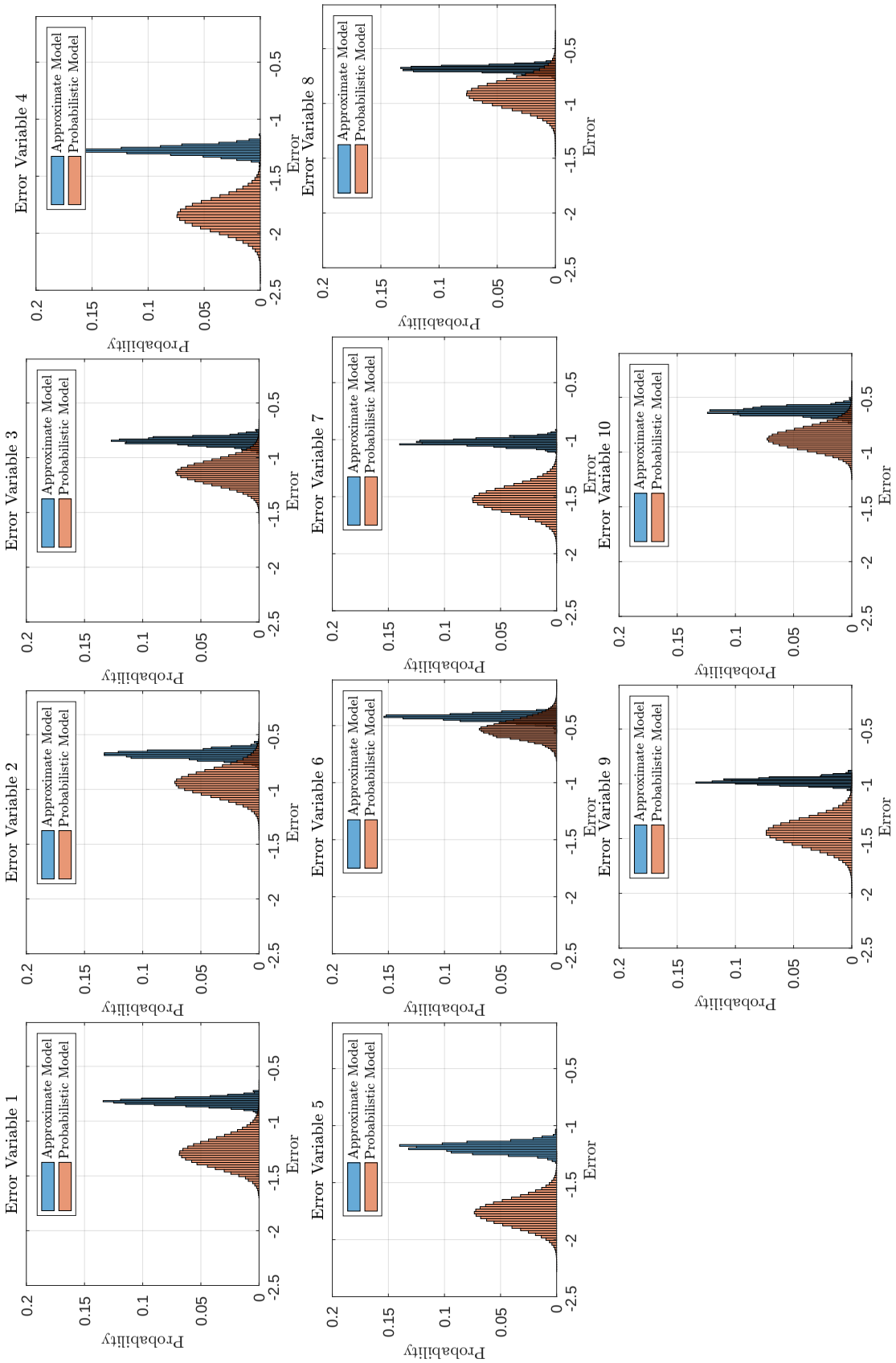

Figure 5.28: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

From these histograms, it is noticed that the negative magnitude of the errors is larger compared to the measurements performed using the `mul8s_1KV8`. The divergence between the two distributions also seems larger. The KL divergence is calculated to 37 for this multiplier, which is larger than the `muls8s_1KV8` as expected from the results in Figure 5.26.

The remaining measurements for every 5 epochs of training using STE on the *approximate model* are shown in Appendix E. The KL divergence was calculated for these measurements and is plotted in Figure 5.29.

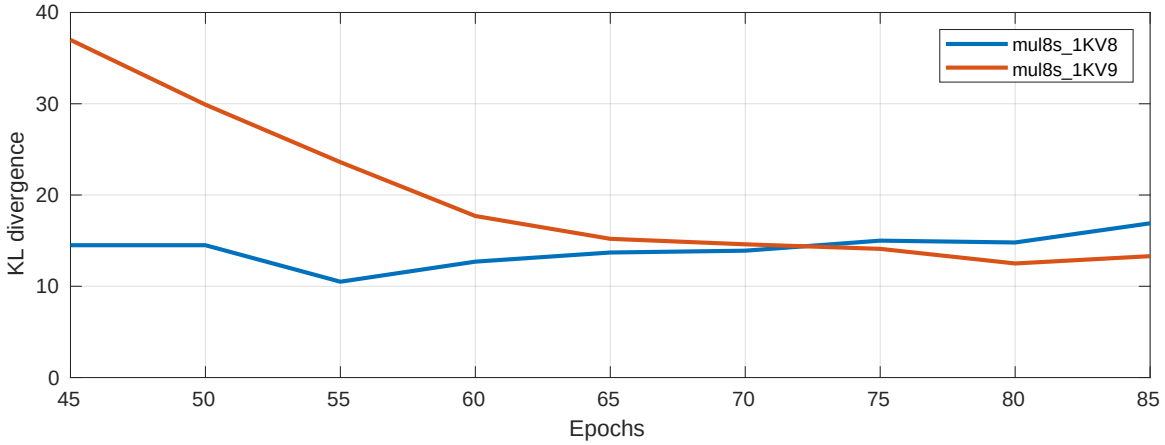

Figure 5.29: Plot of the evolution of the KL-divergence, for progressing epochs. Both chosen multipliers are plotted for comparison purposes. The continuous lines are linear interpolations as measurements are only taken for each 5 epochs.

The development seen in this comparison is congruent with the observations from Figure 5.26, as the divergence of `muls8s_1KV9` is largest before approximate training and decreases along with the training.

A feature worth noticing is the incongruency between the difference in means between the two multipliers. The `mul8s_1KV8` the mean is generally larger, while the opposite is true for the `mul8s_1KV9`.

From the two experiments conducted in Appendix E (summarised here), it is concluded that the KL divergence is a measure that can be used for comparison of *''how well''* a probabilistic model emulates an observation from the *approximate model*. It is observed that it is possible to improve the divergence of the models, by training on the approximate model using STE. However, there seems to be a limit to how well the probabilistic model in its current form can model the deterministic error. The primary source of divergence is observed to be caused by the difference in standard deviation of the two models, where the probabilistic is larger than the deterministic. The differences in means of the two distributions are also contributing to the overall convergence, but it seems to be possible to reduce this impact through approximate training.

The reason behind the difference in standard deviations is reasoned to be caused by the inherent difference between probabilistic and deterministic systems. Adding RVs corresponds to a convolution of their distributions. This means that the standard deviation will grow as more additions of approximate circuits' PMFs are convolved, which is not the case for Q-format limited deterministic circuits.

An invariance to consider is also that the KL divergence is evaluated using the closed-form solution for multivariate Gaussian distributions. However, this is not the case for the deterministic errors

as observed from the plotted histograms. It is regardless expected that the *''shape''* observed in Figure E.24 will be similar as this fits the observations in Figure 5.26.

Conclusively, the analysis indicates that inherent differences in deterministic and probabilistic modelling are limiting the potential of using the modelling strategy developed in section 5.3. Although this is a valid consideration, it is still observed that it is possible to obtain prediction accuracies using the probabilistic model, similar to the ones for the deterministic model. The possibility of evaluating the divergence between the modelling strategies applied to the reference CNN from Table 5.3, is deemed a valuable metric to provide the user of the benchmarking system.

5.6 Summary, Reflection, and Considerations

In this chapter, a CNN used for classification tasks on a reduced CIFAR-100 was designed and implemented in TensorFlow. This network served as a reference for the development of a simulation tool where it is possible to simulate the network using approximate arithmetic units in the multiplications and additions. This system is implemented in C++ where classes are provided such that the possibility of implementing different CNNs using approximate arithmetic is provided to users of the benchmarking system.

Efforts were put into deriving a model that should be able to emulate the effects of the deterministic approximate CNN using, simpler classes that are easily configurable with TensorFlow models. A metric for determining *''how well''* the probabilistic model, emulates the deterministic was introduced, specifically the Kullback-Liebler divergence.

Experiments showed that the concept of using a probabilistic model for emulating deterministic systems introduces unfortunate implications. However, the experiment showed that although the model is not perfect, it can still provide valuable insights into the feasibility of implementing approximate multipliers in a CNN. It is nonetheless suspected that this result is not scalable, for larger networks without further development.

Lastly, an investigation into the prospects of training a CNN using approximate arithmetic units was conducted. This investigation showed promising results using STE, for approximate backpropagation, yielding prominent improvements to the prediction accuracies, both during training and for the *test* data set.

Step III: Full-Scale CNN Error Injection 6

This chapter describes **step III** presented in chapter 3. **Step III** of the benchmarking tool revolves around the *application* and presentation of the metrics found in **step I** and **step II**. The context of **step III** can be seen in Figure 6.1.

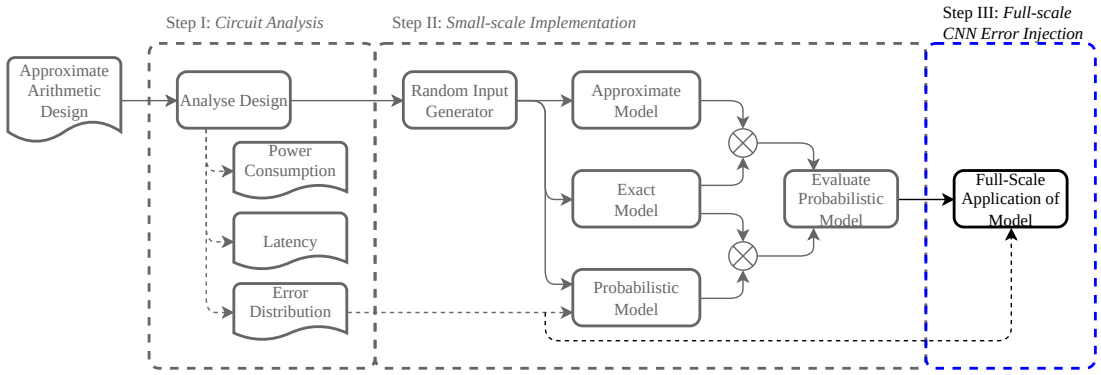

Figure 6.1: Functional diagram of the *benchmarking tool*. An *approximate arithmetic design* is supplied to the tool and is processed through three steps. Lastly, in **step III** the *probabilistic model* from **setp II** is scaled and applied to a full-scale neural network.

Step III requires the *error distributions* from an approximate multiplier found in **step I**, and the evaluation of the noise model from *step II*.

In the following *directory tree* an overview of important files for the interaction with/usage of the *benchmarking tool* can be found. The directory tree will be utilised as a reference to describe *how to use the benchmarking tool*.

```
/
└── STEP_III_walkthrough
    ├── Makefile
    ├── gates.py
    ├── paths.py
    ├── summary/
    ├── figures/
    ├── netlist/
    ├── Error/
    ├── statistic_test_3_models/
    │   ├── NoisyLayer.py
    │   └── statistical_test_3_models.py
```

The following sections will refer to this directory tree.

6.1 Interacting with the Benchmarking Tool

This section will walk through the steps to take, when interacting with the benchmarking tool. The section will take the perspective of the user and define what the user has to do, in order to use the tool to analyse their approximate circuit design.

6.1.1 Performing Step I

A user has defined an approximate circuit and wants to query w.r.t. its power consumption, latency, and the implication of implementing the circuit in their defined CNN. The user has to write two files:

- C/C++ file with the functional description of the circuit, i.e. how the circuit calculates the results on a *functional level*
- Verilog file with the RTL description of the circuit, i.e. how the circuit calculates the results on a *hardware level*

These files are dropped in the directory `STEP_III_walkthrough`. The user defines where to find the two files in the `Makefile` and the name of their circuit (the name of the top `module` in the Verilog file). In the `Makefile`, the user also has access to decide which gates to map their design to and select the appropriate error metrics.

Calling `make analysis` will perform the entirety of **step I** described in chapter 4. The outcome of which will be a set of figures and textfiles:

- **Figures (can be found in `figures/`):**
  - A visual representations of the netlist on gate-level and as a DAG (Directed Acyclic Graph)
  - A histogram of the error distribution
  - A 3D-plot of the metrics *gate count*, *critical path gate count*, and *error rate*.
- **Textfiles (can be found in `summary/`):**
  - A summary of the gatecount, types, and transistorcount
  - A summary of the critical path with the number of gates passed, the estimated latency, and the names of the gates,
  - A CSV file of the error metrics, MSE, MAE, WCD, ER, and MHD
  - A CSV file containing the error for each set of inputs will be available for the following ''steps''.

Given the CSV file with error for each set of inputs, **step II** can be performed.

6.1.2 Performing Step II

From the user's perspective, **step II** is a black box with little to no interaction. The user can run the script, thereby running the ''small-scale implementation''. The CSV containing the errors is utilised in this step, however, the user is not expected to contribute in this step. The script can be found in the directory tree under `/statistical_test_3_models/statistical_test_3_models.py`

Inside the black box 1000 random images of size (16×16) are generated and passed to three CNN implementations of the same architecture: the *approximate model*, the *exact model*, and the *probabilistic model*. The images are forward passed in the models, 1000 repetitions for the *probabilistic model*, and the predictions of the *approximate model* and the *probabilistic model* are compared according to the test defined in chapter 5. The outcome of the comparison is an evaluation of how well the *probabilistic model* represents the *approximate model*.

6.1.3 Performing Step III

Like **step II**, the third step utilises the CSV containing the errors. However, in this step, the user must integrate elements from the *benchmarking tool* in their own CNN. In subsection 5.3.2 two custom layers were defined, which can be found in the directory tree under `/statistical_test_3_models/NoisyLayers.py`:

- `NoisyConv2D()`
- `NoisyDense()`

The user can replace their layers with the custom layers, by importing the classes and changing their model instantiation:

- `tf.keras.layers.Conv2D()` → `NoisyConv2D(error_pmfs=pmfs, precision_bits=m)`
- `tf.keras.layers.Dense()` → `NoisyDense(error_pmfs=pmfs, precision_bits=m)`

The custom layers require two more arguments to be passed: the number of bits used for precision and the ''error PMFs''. The ''error PMFs'' can be found by calling the function `make_pmfs(filepath)` passing the path to the CSV containing the errors.

Although the custom layers are implemented and capable of adding noise based on the ''error PMFs'' of the approximate multipliers, the integration come with some caveats:

- **No Bias:** To simplify the implementation, no bias term has been implemented.
- **Problematic TensorFlow Interactions:** Although, the custom layers are defined using mainly TensorFlow functions and can be inserted into a `Sequential()` model, important functionalities and methods for training and evaluating the models do not work. TensorFlow has multiple Tensor-types, however, they do not share attributes nor methods. `model.evaluate()` and `model.fit()` will fail, as they use `SymbolicTensor`.

To overcome the problematic TensorFlow interactions, a set of functions has been defined in `/statistical_test_3_models/statistical_test_3_models.py` to perform the training and evaluation of the model using the custom layers.

6.2 Applying the Custom Layers to a Full-scale CNN

Step I and **step II** have already been explored with examples, and building on one of these, an approximate multiplier will be utilised in a full-scale CNN. Ideally, the *probabilistic model* would be evaluated for divergence as done in section 5.5. However, as discussed in the section, the model has flaws that make the comparison unsuitable. This is primarily due to the inherent difference between the probabilistic and deterministic domains. The addition of multiple independent RVs makes the PMF of the probabilistic error wider as more distributions are added. Since the CNN derived in section 5.1 was purposely a *small-scale* network it is expected that for larger models, such as the one to be investigated in this section, the models will fit poorly, hence the analysis is omitted. The following example is still viewed as a valuable guide, to showcase the interpretations to be drawn from the results of applying the custom layers on a full-scale CNN.

The small scale CNN developed in chapter 5 was a modified version of Convolutional Neural Network (CNN) from TensorFlows documentation. The model was ''simplified'' and the dataset was replaced with a version of the CIFAR-100 dataset with reduced size and classes. The original example network will now be utilised again, almost without modifications, to see the implications of using the `mul8s_1KV8` 8-bit approximate multiplier on a full-scale network. Two almost identical models are generated, one utilising the custom layers, and the other utilising the TensorFlow

counterparts. As mentioned in subsection 6.1.3 the custom layers do not have the capabilities for introducing a bias term, and for a fair comparison, the model will also not have the bias term.

Table 6.1: Summary of the base model found by calling `model.Summary()`.

Layer Type	Output Shape	Params #
Conv2D	(None, 30, 30, 32)	864
MaxPooling 2D	(None, 15, 15, 32)	0
Conv2D	(None, 13, 13, 64)	18432
MaxPooling 2D	(None, 6, 6, 64)	0
Conv2D	(None, 4, 4, 64)	36864
Flatten	(None, 1024)	0
Dense	(None, 64)	65536
Dense	(None, 10)	640

Note: The total amount of params is 102436.

The two models are trained for 20 epochs with 5 repetitions. When creating a TensorFlow model, the model is instantiated, compiled, and then built. During the ''build'', the error PMFs are convolved with the weights of a given filter/perceptron, the outcome of which is a distribution for each of the filters and perceptrons, wherefrom the noise is sampled. The duration of this process is substantial, and was noted to be around 210 s. Technically, these distributions should be updated for each iteration i.e. after processing each batch. Given the CIFAR-10 dataset in batches of 32 images, each epoch would take ~ 3 days and 19 hours, which is too long. To minimize the time required for an epoch, the noise distributions are not updated, however, this is not seen as a problem, since the distributions will not significantly change given the amount of errors that are summed. The results can be seen in Figure 6.2.

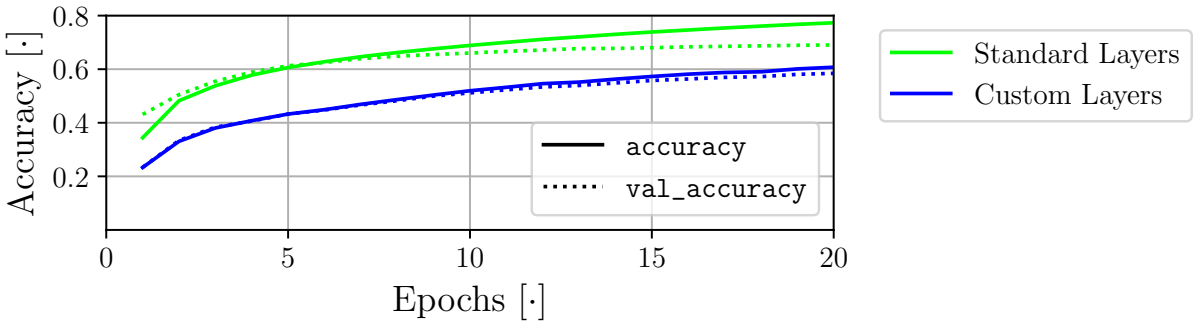

Figure 6.2: Full-scale CIFAR-10 CNN trained with and without noise. The added noise is from the custom layers, that emulate the effect of having the 8-bit approximate multiplier `mul8s_1KV8` as the multiplication circuit.

In Figure 6.2 there is a clear drop in accuracy from using the standard layers to the custom layers, which is expected since probabilistic noise is added in each layer of the network. However, it is worth noting that at the last epoch, the difference in accuracy on the validation set only differs with ~ 10 percentage points.

The questions remains, *how much is the power consumption reduced?* and *is the latency reduced?*. TensorFlow uses `float32` datatypes by default, whereas the custom layers emulate 8-bit FXP, i.e. the approximate multiplier `mul8s_1KV8` replaces a ‘‘single-precision’’ multiplier. It is not straightforward to figure out which single-precision multiplier is used by the computer that ran the simulation, nor is it easy to find the number of transistors in a single-precision multiplier, as the architecture differs from company to company. However, it was possible to find a Verilog file of a single-precision multiplier with truncation instead of rounding from `akilm` in GitHub [126]. Since the Verilog is synthesisable, **step I** is performed on the Verilog file, the outcome of whic is summarised in Table 6.2, alongside the same metrics for `mul8s_1KV8`.

Table 6.2: Summary of the two multipliers passed through **step I** of the *benchmarking tool*.

Multiplier	AND	XOR	NAND	OR	XNOR	NOR	Gate Count	Trans. Count	Gates in Path	Prop. Delay [ns]
single-precision	597	956	1581	71	145	3	3420	21890	70	17.1
`mul8s_1KV8`	91	19	153	50	61	1	375	2262	28	8.1

Note: Transistor count and gate-delays based on Table 4.1 from section 4.1.

With the optimistic estimate of the percentage-wise drop in transistor count reflecting in the same percentage-wise drop in **power consumption**, using `mul8s_1KV8` instead of a single-precision multiplier will reduce the power consumption to 2262/21890 = 10.3 % of the original power consumption. Furthermore, based purely on the propagation delay, the duration of each multiplication is less than half. These estimates do not take the addition circuit, writing to/reading from memory, etc. into account, however, the relatively low drop in accuracy is exchanged for a disproportionate drop in **power consumption** and **latency**.

As a final remark, it is noted that the relationship between STE training on the deterministic and probabilistic models needs verification in the same manner as was conducted in section 5.5. It is from this thesis still unclear if these two training results would yield congruent performances. However, from the analysis conducted it is deemed as a valid area of further research.

Discussion 7

This chapter will discuss some of the results, methods, and consequences of the choices made.

Although the *benchmarking tool* would require some refinement before the final implementation, the tool's potential is considered auspicious. Some of the problems associated with using approximate arithmetic circuits are the ad hoc nature of their design and the difficulty of noticing the implication of integrating these into a CNN before implementation in hardware. With the tool, it is possible to consider the implications w.r.t. **power consumption**, **latency**, **error metrics**, and have the ability to apply the error as noise on their CNN model, thereby providing the user with enough information to evaluate whether or not the approximate arithmetic design fulfils their requirements and compare different approximate arithmetic circuits. **Step I** of the *benchmarking tool* in isolation also serves as a powerful and fast tool for evaluating an approximate arithmetic design in isolation, which may also have uses for an ASIC designer.

In the survey (chapter 2) many other approximate computing techniques were presented, however, only the *precision scaling* and *approximate arithmetic circuits* have been implemented in the benchmarking tool. This choice was made to limit the scope of the project, however, the tool may have benefited from this limitation; applying pruning, early determination, etc. on the final model would more than likely have made the results more unclear. The final implementation focuses on *approximate arithmetic circuits* and by extension applies *precision scaling*, which ensures that no other factors play a role in the evaluation of the supplied approximate arithmetic design.

A fundamental question to ask when dealing with the development of a tool for benchmarking approximate computing techniques is: *How is it possible to verify a solution, that is supposed to be wrong?* The purpose of applying approximate computing techniques is relaxing requirements w.r.t. accuracy by allowing some level of *error* in order to lower parameters like **latency** and **power consumption**. This is problematic for the development of a tool that is supposed to be generalised and scalable, since there is no definitive method of verifying whether the implementation is causing the errors or the techniques. By proxy, it can be verified that the solution works if an exact version of the method is applied, however, as soon as the approximate computing technique is applied, evaluation is based mostly on intuition. This issue was specifically encountered in the verification of the *approximate model* developed in C++ in section 5.2 and the implementation of the *probabilistic model* in section 5.3

The problem statement and the delimitations from chapter 3 depicts a tool, capable of evaluating an *combinatorial approximate arithmetic multiplier* in the context of any CNN. The implementation of the tool is capable of analysing supplied arithmetic designs, apply the designs to a small-scale network, and generalising and scaling the results. There are limitations to the extent of the applicability of the model. Many CNN models use other layer-types, e.g. the ResNets briefly mentioned in subsection 2.1.4, which are not accommodated at this stage of implementation.

During the development of the *approximate model* and the ensuing training hereof, it became clear that many factors have to align to successfully implement a CNN in a *trainable* state; metrics like Q-format, choice of activation, limiting the size of the weights, complexity of the network, etc. The question of whether it makes sense to try and develop a general framework for all CNN applications arise. Even if these factors would culminate into a bad result for some application, this is still deemed as important information to give the user. Rather than interpreting negative results from a supplied approximate arithmetic design as ''not implementable'', it should be interpreted as ''will require configurations and further development'', which in itself is *nice-to-know* for the user.

During development and testing only 8-bit multipliers were tested, but other sizes like 16-bit multipliers were readily available from the same source. The 8-bit multipliers are more difficult to deal with because, alongside the errors introduced from the approximativeness of the circuits, a considerable quantisation error is also introduced. Furthermore, only having 8 bits to represent both integer values and fractional values is a finicky process that will easily result in overflow or too few bits for precision (as seen in section 5.4). Using the larger 16-bit multipliers might have made the implementation easier by avoiding these specific pitfalls, however, uncovering these ''problems'' is seen as important for the *benchmarking tool*, since it should be able to accommodate any multiplier design. In short, the choice of solely relying on 8-bit arithmetic units introduces only the most challenging problems, hence the range of the application was never fully explored.

In the early part of development, the machine learning API was chosen to be TensorFlow, a powerful tool capable of easing the burden of training, testing, and implementing neural networks of many different types. Designing custom layers using TensorFlow was accompanied by `AttributeErrors` from their datatypes. During the development and testing of the custom layers, the basic `Tensor` was used and no problems arose, however, when the layers were applied in a model, the methods utilised for the model became unusable since they use `SymbolicTensor` and some of the important methods did not exist, either causing an error and stopping the program or skipping essential lines of code. Thus it became necessary to create custom functions for performing epochs and evaluating the models, adding more slow/inefficient code. It is difficult to say with certainty that this problem could have been avoided by using another API, however, it is worth noting that further refinement of the code may lead to functioning custom layers, that would be able to handle these `SymbolicTensors` and thereby making it possible to use faster and better methods for training and evaluating the models.

7.1 Step I - Circuit Analysis

A simplification w.r.t. the **power consumption** and **latency** was performed, when equating them to the transistor-count and critical path delay, respectively. Many factors play into the **power consumption** such as the specific technology available, the ''activity'' of the transistors, the registers used for saving intermediate values, etc. **Latency** is innately tied to the system, i.e. unless an entire CNN was implemented without using any sequential elements, the clock-rate of the sequential elements will specify the latency, where the critical path will set the ceiling of the clock-rate. The simplifications may lack precision, however, this is traded for more general metrics. Furthermore, to accommodate a user with knowledge about the available technology, some of these metrics can be specified: Transistors and gate-delay per gate. Granting the user more customisability w.r.t. the power consumption of each gate-type, could potentially make the metrics more precise.

At the current state of implementation, the user is expected to supply two separate files, specifying the multiplication/addition process in both C/C++ and Verilog. This is redundant, and impractical, and ensuring the ''two'' designs are identical may take time or introduce errors. The benchmarking tool would benefit greatly if there was a ''translation layer'' for taking the Verilog design and

converting it to C/C++ and/or vice versa. This would also reduce the number of errors the user might cause and streamline the process of benchmarking an *approximate arithmetic design*. Multiple tools exist, that can potentially be the ''translation layer'' like Verilator and v2c. During development both C and Verilog files for all the modules used for testing and simulation were available through certain libraries, why this ''translation layer'' was not deemed essential for the development of the *benchmarking tool*.

The development of the *benchmarking tool* revolved mostly around 8×8 approximate multiplication circuits, however, there exists a lot of differently sized approximate multipliers. When performing the *error simulation* as presented in section 4.3, it is possible to check every combination of outputs and generate an LUT. However, if the *benchmarking tool* had to handle a larger 16×16 approximate multiplier an LUT would be infeasible to store in the RAM. The number of entries would be $2^{16}\cdot2^{16}$ each entry being 32 bits, meaning the LUT would be $\sim$ 17.18 GB, saving each entry as a 16-bit value would still require half of that. In this case, it would be appropriate to generalise the error distribution by performing Monte Carlo simulations instead, which would not affect the PMF plots, however, the current method of saving ''error PMFs'' as a lookup table would have to be refined to accommodate this method.

7.2 Step II - Small-scale approximate neural network

The chosen reference system performs image classifications on the CIFAR-100 dataset with reduced number of classes and image sizes. This choice works well as a subsystem, since there is little interpretation required, i.e. the accuracy is an evaluation of how well the model performs its task. The results can then be generalised, scaled, and applied to other machine learning tasks. However, approximate computing techniques are especially relevant in tasks, where human perception plays a role, since the errors induced by the techniques may be ''filtered/sorted'' out by the brain. The lateral jump from image classification to another CNN application like *denoising* has not been explored, and the implications are not entirely clear. Since the machine learning techniques do not differ, the ''error'' added should be similar (perhaps identical), however, perception is a complex topic, and the noise may be very clear for the brain, if it can see/recognize patterns in it.

The *approximate model* was implemented in C++ due to familiarity and the end product is slow. During the training of the C++ using the defined *approximate epochs* it was revealed that for the specific CNN architecture, the mean time per epoch was 3669.5 s $\approx$ 1 hour 1 minute 9.5 seconds. Since the implementation required modifying the MAC operation, libraries and tools for performing matrix computations could not be utilised, and speeding up the processes would have required extensive development. However, since the model was operational and seemingly produced the correct output, speeding up the process was not seen as essential for the development of the *benchmarking tool*. The effects of this choice has rippled down the subsequent development and tests since repetitions were deemed too costly w.r.t. time. A few simple ideas for speeding up the network were tried: Using python was too slow, since the convolutions would require seven nested for-loops, using parallel processing to unpack the outermost for loop sped this up by lowering the computation time to just below 400 s per iteration but still too slow since each epoch consisted of 157 iterations totaling just below 17 hours 27 minutes. One of the benefits associated with utilising Python would be to avoid the I/O process of saving and reading from CSV files. There are tools/libraries like Pybind11, wherewith C++ functions can be called from within a Python script, which should remove the I/O processes and speed up the for-loops, but with no familiarity with these types of tools and a working CNN written in C++, this was not prioritised.

Implementing fixed-point arithmetic consequentially adds the possibility of overflow and the need for being aware of the Q-format. The current implementation was capable of performing

inference, training, and more using an approximate arithmetic circuit in-place, however, stability and generalisation of this model would require a more fixed-point-aware design. The weights are encouraged to stay at small values using L2 regularisation, the images were normalised, but the values between the hidden layers are not entirely accounted for. An idea for ensuring the values between the layers remain relatively small would be to use ‘‘sigmoid’’ as the activation function, but this would have implications on the training, since the *vanishing gradients* may force the weights to take very large values, and by extension shift the burden to weights rather than solving the problem. It should be noted, however, that although there are improvements to be made, the implementation of a small-scale CNN with the capability to insert an *approximate arithemetic operation* works. The implementation is capable of training with the defined architecture, optimisation algorithm, and machine learning problem. Potentially, the design and architecture of this model could be applied to another problem and by extension act as the user’s implementation.

From the results in section 5.5 it is clear, that there is a significant difference between the model with approximate arithmetic in-place and the probabilistic model trying to emulate the effects of the approximate arithmetic design. A possible explanation would be the inherent difference between deterministic and probabilistic modeling; perhaps the assumption of this being representative of the error from an approximate circuit is fallacious. An extensive analysis would be necessary to derive a more suitable model, but based on the results the possibility of providing accurate estimates that are simple to implement for a user is not deemed infeasible.

The statistical model in isolation was supposed to be a simple and effective method for applying the implications of utilising a specific approximate arithmetic design in a user-specific CNN. However, due to the vast possibilities of approximating a circuit, it was seen as necessary to have a metric to *evaluate* how well the *probabilistic model* emulated the noise from the error distribution, and by extension how accurate the noise applied to the user’s CNN is. Unfortunately, there is a clear distinction between the error added in the *probabilistic model* and the error from the arithmetic circuits, however, this is clear because of the evaluation, and underlines the importance of the process.

The results of section 5.3 suggest that the *probabilistic model* requires some modifications and revision to be an appropriate stand-in for the *approximate model*, why training the *probabilistic model* and exploring whether the training mirrors the training of the *approximate model* has not been explored. Given the future development of a more concise modelling of the deterministic errors, justify an investigation of the congruency between training on the using probabilistic model using the same STE technique as developed methods from section 5.4.

The *approximate model* is implemented and functioning with the possibility of replacing the multiplication arithmetic. This might suggest that designing and implementing a *probabilistic model* might be superfluous, however, as displayed in section 5.4 there are some application specific problems like the number of precision bits, overflow, etc. that have to be taken into account, before training is possible. This is not seen as user friendly, as the purpose of the tool is to give a ‘‘quick’’ insight into the effect on a final implementation.

7.3 Step III - Full-scale CNN Error Injection

In Figure 6.2 the accuracy of a TensorFlow model using standard layers and a TensorFlow model using the custom ‘‘noise’’ layers based on the approximate multiplier `mul8s_1KV8` trained on the CIFAR-10 dataset are plotted against each other. The comparison seeks to show how the approximate multiplier would affect the accuracy of the CNN if implemented, without actually implementing it. Based on the results from section 5.5 it may be fallacious to say with certainty,

that the accuracy reached with the probabilistic model is congruent with the reachable accuracy with the approximative multiplier in-place. Furthermore, verifying the accuracy would be very difficult as shown in section 5.1, where a smaller network suffered from problems originating from the choice of precision bits, and likely overflow caused by the summation of a large set of products. This problem would likely be exacerbated in this example, and the verification would then require extensive development for the approximate full-scale model.

In Figure 6.2 the *probabilistic model* is trained, however, the implications of training the noise model have not been explored. It is unknown at this point if the accuracy of the *probabilistic model* and the *approximate model* will converge, diverge, or run in parallel if the weights are trained using the *probabilistic model*. The purpose of the *probabilistic model* is to give the user a ‘‘simple’’ method for applying the error of an approximate arithmetic circuit as ‘‘noise’’ to their model, with little to no configuration, however, training the *probabilistic model* is not guarenteed to yield the same accuracy of an actual implementation nor weights that could be applied if the model were to be implemented.

Conclusion 8

This thesis aimed to apply **approximate computing** techniques on **neural networks** by developing a tool that would be able to *benchmark* approximate arithmetic designs, and evaluate their **power consumption**, **latency**, and the implications of applying them to a full-scale scenario before implementation. In chapter 1 the financial and environmental burdens associated with large scale data models was highlighted and how the main focus of papers about machine learning focused on the *accuracy* of the models rather than *efficiency*. On the other hand the field of *approximate computing* is gaining traction, where an increase in ''errors'' can be exchanged for a disproportionate reduction in energy, resource, and/or time consumption. Machine learning algorithms and models have proven to be powerful at removing/disregarding noise, why this thesis grabbed the opportunity to combine elements of these two fields of study. There are existing tools for applying some approximate computing techniques on *neural networks*, however, these frameworks are mostly aimed at the *machine learning designer* rather than an *ASIC developer*. The ad hoc nature of developing approximate arithmetic designs and the difficulties of seeing the effects/implications of the implementation, is viewed as an opportunity to develop a *benchmarking tool* to alleviate these problems. The benchmarking tool was summarised in the problem statement:

''How can a benchmarking tool provide an ASIC developer with relevant metrics to evaluate an *approximate* arithmetic circuit as an integral part of a large scale system, i.e. a *neural network*, prior to implementation?''

To answer the *problem statement*, a survey about neural networks, digital design, and approximate computing was performed and documented. The neural networks survey highlighted and elaborated on the building block of NNs, the **perceptron**. The perceptron is a simplistic model of a biological neuron, designed to mimic functions of the human brain's neural network. With this model, a set of inputs are weighed by individual factors, summed, and processed through an *activation function*, which can introduce non-linearity. A set of perceptrons can be structured to form layers, and a set of layers can be structured to comprise a neural network. The effectiveness of neural networks is partly due to their *ability to learn*; the weights of the perceptrons can be updated using a *loss function* and the concept of *error back-propagation*. The loss function defines how well the model is performing the assigned task. The evaluated loss is used in the back-propagation step, where the weights' influence on the loss is used to find the individual gradients from weight to loss. With the gradients a *gradient descent algorithm* can be utilised to *minimise the loss*, thereby improving the model. The perceptrons are essentially MAC (multiply-accumulate) units with an activation function to process the output. In the ''digital design'' part of the survey, the MAC units were explored. Number representation is essential for the development of arithmetic circuits required for performing a MAC operation, and two number representations were highlighted: FXP (fixed point) and FLP (floating point). FXP has the same precision across the entire range, the LSB

(least significant bit); the format of a FXP value is a tradeoff between range and precision, increasing one will lower the other. With FLP a larger range is available, however, the precision is lowered at large values. Arithmetic circuits must be aware of the number representation, as it requires larger and more complex circuits to perform exact arithmetic operations with FLP values. Lastly, the topic of *approximate computing* computing was addressed; an umbrella term for any concept that simplifies system computations at the expense of accurate operations. W.r.t. the MAC units *precision scaling* and *approximate arithmetic circuits* are especially relevant. Precision scaling is approximations introduced by representing numbers with fewer bits than the `float32` standard from IEEE and *approximate arithmetic circuits* are modifications of existing arithmetic circuit to lower the *power consumption* and/or *latency*.

Based on the knowledge gained from the survey, it became clear that there is room for a contribution to the field of *approximate computing*; a *benchmarking tool* for evaluating *approximate arithmetic circuits* pre-implementation. The tool would consist of three steps:

I) **Circuit Analysis:** Investigation of a user-supplied arithmetic design in isolation, analysing and presenting metrics for *power consumption*, *latency*, and *error distribution*
II) **Small-scale Implementation:** Implementation of the arithmetic design in a CNN architecture mirrored by a *probabilistic model*. The purpose of which is to evaluate how well the *probabilistic model* emulates the errors of a real implementation, thereby giving a metric for the appropriateness of the generalisation of the error and by extension, the appropriateness of scaling the model
III) **Full-scale CNN Error Injection:** The *probabilistic model* is applied to the user's CNN, whereby the implications of implementing the approximate arithmetic circuit can be scoped pre-implementation

The development of these three steps comprises the entirety of the *benchmarking tool*.

Step I - Circuit Analysis requires two files describing the *approximate arithmetic circuit*: A Verilog file of the design describing the design at the RTL (Register Transfer Level) and a C/C++ file with the operation described at a functional level. The module defined in the Verilog file is synthesised and mapped to a set of logic gates, i.e. a netlist. This netlist is saved in two formats: Verilog and JSON. The JSON netlist is utilised for processing the design in Python. The **power consumption** of the design is simplified to the **transistor-count** of the synthesised netlist, to avoid some technology-specific factors, and is found by counting all gates, taking note of their type, and scaling the gate type-count with how many transistors are used for the different types of gates. The number of transistors per gate is also a technology-specific factor, however, this can be configured in the script. The **latency** of the approximate circuit is simplified to be *the sum of gate-delays along the critical path*. The *critical path* is found by translating the netlist to a DAG (Directed Acyclic Graph) and utilising the Python library `networkx` to find the longest path. Furthermore, the weights of the ''edges'' between the nodes are defined to be the gate-delay of the specific gates, thereby incorporating a method to find the *critical path* defined by the slowest path and not necessarily the path with most nodes. The C/C++ file is passed to a C++ script which evaluates the **mean square error**, **mean absolute error**, **worst case distance**, **error rate**, and **mean Hamming distance**. Furthermore, the error PMF of the approximate arithmetic circuit is found by evaluating every possible combination of inputs and outputs and comparing them with an exact version of the same circuit. This error PMF is essential for both **step II** and **step III**. All the metrics from **step I** inform the user about what to expect from an isolated implementation of the circuit, wherewith comparison with other approximate circuits can quickly be performed.

Step II - Small-scale Approximate Neural Network implements and compares three architecturally identical CNNs: The *exact model*, the *approximate model*, and the *probabilistic model*. The exact model should be viewed as a reference for the other two models. The development of the exact model through TensorFlow's API defined the machine learning problem, architecture, and design of the two other models, and was essential for the training in subsequent tests. The approximate model was implemented in C++ as a twin to the reference model, wherewith the multiplication operation could be exchanged with the user's C/C++ design of their approximate arithmetic circuits. The *approximate model* is thus capable of performing forward-passes using the custom arithmetic circuits in-place. Research and testing went into trying to train the *approximate model*, which uncovered some important but subtle complications a user may encounter during the implementation of an approximate arithmetic design in a large system, e.g. the importance of choosing the correct number of precision bits, STE (Straight-Through Estimator) is a powerful simplification, however, training is not guaranteed. Importantly, the *approximate model* is a fully functional CNN with approximate arithmetic operations. It is capable of training its weights using the STE and could potentially be reused on simple machine learning problems ''as is''. The generalisation and scalability of this model are limited, why the *probabilistic model* is introduced. The *probabilistic model* seeks to generalise and produce a scalable method for emulating the noise from the errors associated with the supplied circuit on any CNN model. This is done with custom-defined layers, within the TensorFlow framework, in which error distributions are generated based on the current weights per perceptron/filter, sampling noise from these distributions, and applying them just before the activation function. Efforts were made to evaluate the effects of using the *probabilistic model* for inference through the metric Kullback-Liebler divergence. This metric proved useful in the analysis of the *probabalistic model*. Experiments showed a significant difference in standard deviations, between the *probabilistic* and *deterministic model*. The difference in standard deviations is attributed to the inherent nature of probabilistic systems, where adding random variables (RVs) involves convolution of their distributions, resulting in a growing standard deviation, unlike the fixed standard deviation in Q-format limited deterministic circuits.

Step III - Full-scale CNN Error Injection is the culmination of the knowledge and models from **step I** and **step II**. The error PMFs from **step I** are reused in the custom layers defined in **step II** and the custom layers replaces their exact counterparts in the user's model. The user can then infer the implications of implementing the approximate arithmetic circuit in their model pre-implementation. This was trialed on an example CNN from TensorFlow, performing image classification on the CIFAR-10 dataset. The noise was based on the error from the approximate 8-bit multiplier, `mul8s_1KV8`, and the graph comparing the accuracy of the example CNN with and without noise shows that the accuracy dropped by ~ 10 percentage points, from ~ 70 % to ~ 60 %. The reduction in accuracy is traded for a disproportionate drop in **power consumption** and **latency**: An example `float32` multiplier was used as the ''baseline'' for the comparison which consists of 21890 transistors, whereas `mul8s_1KV8` only consists of 2262, i.e. only 10.3 % of the baseline multiplier. The latency also went from 17.1 ns to 8.1 ns, less than half of the baseline.

The combined *benchmarking tool* is capable of RTL synthesis and by extension evaluating combinatorial circuits in isolation. The tool can insert the circuit into a configurable CNN and train the network. It was seen in Figure 5.25 that training a network with approximate arithmetic operations in-place using the STE (Straight-Through Estimator) yields positive results, this result in itself opens up many possibilities for research: *To what extent is STE appropriate?*, *What happens with the error distribution at the output as well as between the layers during training?*, etc. Furthermore, two custom layers have been defined, that a user can import and utilise to emulate the error as ''noise'' in their CNNs. The layers are also evaluated by the tool, giving a metric of the *certainty* with which the modeled noise mirrors the errors of the approximate circuits. There are some caveats as presen-

ted in the discussion in chapter 7, but the potential of the tool is viewed as a ‘‘possibly significant contribution to help bridge the gap between *approximate circuit design* and the ever-increasing relevancy of *neural networks*, that may assist in making machine learning more sustainable’’.

Further Work 9

The current implementation mostly succeeds in solving the *problem statement*, however, there is room for improvements. As stated in the discussion, the speed of the current implementation is very slow. This would be a problem for a user, since ad hoc solutions may require frequent small changes, which would be impossible with the current speed. Furthermore, the development and testing of the solution would be positively impacted by speeding up the tool. The I/O problem associated with reading/writing weights from/to CSV files must be handled, possibly by using a tool like Pybind11 where C++ functions can be called with variables from Python; this would solve the I/O problem without losing the speed of C++ compiled code. The interactions between a user and the *benchmarking tool* are currently not very intuitive nor user-friendly, and something like a graphical user interface should be designed. The tool is configurable with transistors and propagation delay for each logic gate type, however, the **power consumption** and **latency** metrics are affected by more than just these two factors, and elaborating the metric enough to give an estimate in Watts or something else easily interpretable would give the user a more definitive view of their circuit. This would most likely require more focus and research into simulation of the supplied circuits. Another interesting path to explore the possibility of making the *probabilistic model* aware of implementation-specific factors, such as the pooling layers and activation functions. This may help the probabilistic model fit better with the user's application.

Approximate arithmetic circuits and precision scaling is a small subset of the available AC (Approximate Computing) techniques. Incorporating more techniques to be benchmarked or possibly creating a suite of benchmarking tools for the other techniques would significantly improve the contribution of the *benchmarking tool*, granting access and information to the user, and hopefully simplifying the process of analysing and implementing approximate techniques in large scale systems

During the development of the *benchmarking tool* many fascinating branches of research and possible research opportunities/questions were uncovered:

- What are the implications of training the probabilistic model? Could the weights potentially be transferred to the *approximate model*, thereby avoiding the problems accompanying the training of the *approximate model*? Would these weights perform better than the weights from the *exact model* on inference?
- If both approximate multipliers and adders were implemented in the same model, how would training and inference be affected? What if multipliers/adders were varied across the layers?
- Would *approximative aware back-propagation* yield better results than using the STE (Straight-Through Estimator)?
- Are the results presented in the thesis applicable in other CNN applications like denoising and regression?
- Would it be possible to generalise and define clear rules/methods w.r.t. precision scaling to avoid problems like overflow?

The answers could be integrated into the tool and improve the models.

Bibliography

[1] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14--17, 1964.

[2] Serge Torres, Claude-Pierre Jea, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Lefèvre. *Handbook of Floating-Point Arithmetic*. Boston, MA: Birkhäuser, 2nd ed. 2018 edition, 2009.

[3] Vaibbhav Taraate. *Digital logic design using Verilog : coding and RTL synthesis* . Springer, Singapore, 2nd ed. edition, 2022.

[4] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, 2017.

[5] IBM. What is Artificial Intelligence. https://www.ibm.com/topics/artificial-intelligence. (Accessed on 19/02/2024).

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1. edition, 2006.

[7] Nils J. Nilson. *The Quest for Artificial Intelligence: A History of Ideas and Achievements*. Cambridge University Press, 1. edition, 2010.

[8] Zhi-Hua Zhou. *Machine Learning*. Springer, 1. edition, 2016.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2. edition, 2003.

[10] Ethem Alpaydin. *Introduction to Machine Learning*. MIT University Press, 1. edition, 2004.

[11] Ke-Lin Du and M. N. S. Swamy. *Neural Networks and Statistical Learning*. Springer, 2014.

[12] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. London: Chapman and Hall, 26. edition, 1998.

[13] Matthieu Cord and Pádraig Cunningham. *Machine Learning Techniques for Multimedia*. Springer, 1. edition, 2006.

[14] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computing*, 7(18), 2006.

[15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[16] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An Introductory Review of Deep Learning for Prediction Models With Big Data. *Frontiers in Artificial Intelligence*, 3(28), 2020.

[17] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 1958.

[18] Yu chen Wu and Jun wen Feng. Development and Application of Artificial Neural Network. *Wireless Pers Commun*, 102, 2017.

[19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[20] AI and compute, March 2024. [Online; accessed 8. Mar. 2024].

[21] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proceedings of the IEEE*, 108(12), 2020.

[22] Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Muhammad Shafique Xun Jiao, Kiamal Pekmestzi, and Dimitrios Soudris. Approximate Computing Survey, Part I: Terminology and Software & Hardware Approximation Techniques. *https://doi.org/10.48550/arXiv.2307.11124*, 2023. (Under Review).

[23] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, Laura Pozzi, and Sherief Reda. Approximate Logic Synthesis: A Survey. *Proceedings of the IEEE*, 108(12), 2020.

[24] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597v3*, aug 2019.

[25] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20)*, 2020.

[26] Nadezhda Semenova, Laurent Larger, and Daniel Brunner. Understanding and mitigating noise in trained deep neural networks. *Neural networks*, 146:151--160, 2022.

[27] Vasileios Leon, Muhammad Abdullah Hanif, Giorgos Armeniakos, Muhammad Shafique Xun Jiao, Kiamal Pekmestzi, and Dimitrios Soudris. Approximate Computing Survey, Part II: Application-Specific and Architectural Approximation Techniques and Applications. *https://doi.org/10.48550/arXiv.2307.11128*, 2023. (Under Review).

[28] Petr Rek and Lukas Sekanina. TypeCNN: CNN Development Framework With Flexible Data Types. *IEEE Xplore*, 2019.

[29] Yinghui Fan, Xiaoxi Wu, Jiying Dong, and Zhi Qi. Axdnn: Towards the cross-layer design of approximate dnns. jan 2019.

[30] Cecilia De la Parra, Andre Guntoro, and Akash Kumar. Proxsim: Gpu-based simulation framework for cross-layer approximate dnn optimization. *IEEE Xplore*, 2020.

[31] Fionn Murtagh. Multilayer Perceptrons for Classification and Regression. *Neurocomputing*, 183(2), 1990.

[32] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer Perceptrons and Neural Networks. *WSEAS TRANSACTIONS ON CIRCUITS AND SYSTEMS*, 7(8), 2009.

[33] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Pearson, 2. edition, 2005.

[34] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation Functions in Neural Networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12), 2000.

[35] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503, 2022.

[36] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of Trends in Practice and Research for Deep Learning. *https://doi.org/10.48550/arXiv.1811.03378*, 2018.

[37] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[38] Leon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9), 1998.

[39] Doyen Sahoo, Quang Pham, Jing Lu, and Steven C.H. Hoi. Online Deep Learning: Learning Deep Neural Networks on the Fly. *https://doi.org/10.48550/arXiv.1711.03705*, 2017.

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-Propagating Errors. *Nature*, 323(9), 1986.

[41] Yeshwanth Valaboju. A Literature Review on Neural Network Architechtures. *Journal of Interdisciplinary Cycle Research*, 7(2), 2015.

[42] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *https://doi.org/10.48550/arXiv.1506.00019*, 2015.

[43] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *https://doi.org/10.48550/arXiv.1901.00596*, 2019.

[44] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Sawi. Understanding of a Convolutional Neural Networks. *ICET2017*, 2017.

[45] Claus Neubauer. Evaluation of Convolutional Neural Networks for Visual Recognition. *IEEE Transactions on Neural Networks*, 9(4), 1998.

[46] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12), 2021.

[47] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 1998.

[48] Bjørn Højmose Grevenkop-Castenskiold, Albert Berg Hansen, Simon Dahl Jepsen, Nikolaj Krægpøth, Kristoffer Martinsen, Jonas Emil Nielsen, and Morten Rønberg. Quantum Machine Learning. Technical report, 2022.

[49] IBM. What is Computer Vision. https://www.ibm.com/topics/computer-vision. (Accessed on 01/04/2024).

[50] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141--142, 2012.

[51] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, 1, 2012.

[53] Matthew D. Zeriler and Rob Fergus. Stochastic Pooling for Regularization of Deep Conovolutional Neural Networks. *https://arxiv.org/abs/1301.3557*, 2013.

[54] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout Networks. *https://arxiv.org/abs/1302.4389v4*, 2013.

[55] Nitish Srivastava and Ruslan Salakhutidinov. Discriminative Transfer Learning with Tree-based Priors. *Department of Computer Science, University of Toronto*, 2013.

[56] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network . *https://arxiv.org/abs/1312.4400v3*, 2014.

[57] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-Supervised Nets. *https://arxiv.org/abs/1409.5185*, 2014.

[58] Zhichen Yan, Hao Zhang, Robinson Piramuthu, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Yizhou Yu. HD-CNN: Hierarchical Deep Convolutional Neural Networks for Large Scale Visual Recognition. *https://arxiv.org/abs/1410.0736v4*, 2015.

[59] Oren Rippel, Jasper Snoek, and Ryan P. Adams. Spectral Representations for Convolutional Neural Networks. *https://arxiv.org/abs/1506.03767v1*, 2015.

[60] Anish Shah, Sameer Shinde, Eashan Kadam, Hena Shah, and Sandip Shingade. Deep Residual networks with Exponential Linear Unit. *https://arxiv.org/abs/1604.04112v4*, 2016.

[61] Yuhang Li, Yufei Guo, Shanghang Zhang, Shikuang Deng, Yongqing Hai, and Shi Gu. Differentiable Spike: Rethinking Gradient-Descent for Training Spiking Neural Networks. *NeurIPS 2021*, 2021.

[62] Rishit Dagli. ASTROFORMER: MORE DATA MIGHT NOT BE ALL YOU NEED FOR CLASSIFICATION. *https://arxiv.org/abs/2304.05350v2*, 2023. (Published as a conference paper at ICLR 2023).

[63] Randy Yates. Fixed-Point Arithmetic: An Introduction. http://www.digitalsignallabs.com/fp.pdf, Sep 2020. (Accessed on 20/02/24).

[64] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019*, 754, 2019. (Revision of IEEE std 754-2008).

[65] Mi Lu. *Arithmetic and Logic in Computer Systems*. Hoboken, NJ : Wiley-Interscience, 1. edition, 2004.

[66] Andrew D. Booth. A Signed Binary Multiplication Technique, aug 1950.

[67] Stamatis Vassiliadis, Eric M. Schwarz, and Don J. Hanrahan. A General Prood for Overlapped Multiple-Bit Scanning Multiplications. *IEEE Transactions on Computers*, 38(2), 1989.

[68] C.R. Baugh and B.A. Wooley. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Transactions on Computers*, C-22(12):1045--1047, 1973.

[69] S.D. Pezaris. A 40-ns 17-Bit by 17-Bit Array Multiplier. *IEEE Transactions on Computers*, C-20(4):442--447, 1971.

[70] Maroju Sai Kumar, D Ashok Kumar, and P Samundiswary. Design and performance analysis of Multiply-Accumulate (MAC) unit. In *The Institute of Electrical and Electronics Engineers, Inc. (IEEE) Conference Proceedings*, pages 1084--, Piscataway, 2014. The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

[71] Marcin Kubica, Adam Opara, and Dariusz Kania. *Technology mapping for LUT-based FPGA*. Lecture notes in electrical engineering ; Volume 713. Springer, Cham, Switzerland, 1st ed. 2021. edition, 2021.

[72] Designing Arithmetic Circuits with Deep Reinforcement Learning | NVIDIA Technical Blog, jul 2022. [Online; accessed 1. Apr. 2024].

[73] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test*, 34(2):60--68, 2017.

[74] Richard P. Brent and H. T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, C-31(3), 1982.

[75] Rahila K. C. and U. Sajesh Kumar. A Comphrehensive Compariative Analysis of Parallel Prefix Adders for ASIC Implementation. *ICSEE*, 2019.

[76] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, C-22(8):786--793, 1973.

[77] Sa'ed Abed, Yasser Khalil, Mahdi Modhaffar, and Imtiaz Ahmad. High-performance low-power approximate wallace tree multiplier. *International journal of circuit theory and applications*, 46(12):2334--2348, 2018.

[78] Floating Point and IEEE 754 Compliance for NVIDIA GPUs, March 2024. [Online; accessed 12. Apr. 2024].

[79] Adrian Sampson. *Hardware and Software for Approximate Computing*. PhD thesis, University of Washington, 2015.

[80] Tailin Lianga, John Glossnera, Lei Wanga, Shaobo Shia, and Xiaotong Zhanga. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *https://doi.org/10.48550/arXiv.2101.09671*, 2021.

[81] Yang He and Lingao Xiao. Structured Pruning for Deep Convolutional Neural Networks: A survey. *Journal of LaTeX Class Files*, 14(8), 2015.

[82] N. Manikandan, M. Priyanka, Sasikumar, and R. Muthaiah. Approximation Computing Techniques to Accelerate CNN Based Image Processing Applications – A Survey in Hardware/Software Perspective. *International Journal of Advanced Trends in Computer Science and Engineering*, 9(3), 2020.

[83] Sourav Sanyal, Shubham Negi, Anand Raghunathan, and Kaushik Roy. *Approximate Computing for Machine Learning Workloads: A Circuits and Systems Perspective*, chapter 15. Springer, 2022.

[84] Mukhammed Garifulla, Juncheol Shin, Chanho Kim, Won Hwa Kim, Hye Jung Kim, Jaeil Kim, and Seokin Hong. A Case Study of Quantizing Convolutional Neural Networks for Fast Disease Diagnosis on Portable Medical Devices. *Sensors*, 219(22), 2022.

[85] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A White Paper on Neural Network Quantization. *preprint arXiv:2106.08295v1*, 2021.

[86] Paul Merolla, Rathinakumar Appuswamy, John Arthur, Steve K. Esser, and Dharmendra Modha. Deep Neural Networks are Robust to Weight Binarization and Other Non-Linear Distortions. *preprint arXiv:1606.01981v1*, 2016.

[87] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks. *30th Conference on Neural Information Processing Systems*, 2016.

[88] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. Energy-Efficient and Improved Image Recognition with Conditional Deep Learning. *ACM Journal on Emerging Technologies in Computing Systems*, 13(3), 2017.

[89] Tanvir Arafin, Qian Xu, and Gang Qu. *Voltage Overscaling Techniques for Security Applications*, chapter 12. Springer, 2022.

[90] Armin Alaghi, Weikang Qian, and John P. Hayes. Voltage over-scaling: A cross-layer design perspective for energy efficient systems. *20th European Conference on Circuit Theory and Design*, 2011.

[91] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[92] Oliver Keszocze. Approximate Computing. *Information Technology*, 64(3), 2022.

[93] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. A low latency generic accuracy configurable adder. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[94] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. SMApproxLib: Library of FPGA-based Approximate Multipliers. *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, 2018.

[95] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. Approximate XOR/XNOR-based adders for inexact computing. *IEEE International Conference on Nanotechnology*, 2013.

[96] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. IMPACT: IMPrecise adders for low-power approximate computing. *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011.

[97] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4), 2010.

[98] Ayad Dalloo, Ardalan Najafi, and Alberto Garcia-Ortiz. Systematic Design of an Approximate Adder: The Optimized Lower Part Constant-OR Adder. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(8), 2018.

[99] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo. An enhanced low-power high-speed Adder For Error-Tolerant application. *Proceedings of the 2009 12th International Symposium on Integrated Circuits*, 2009.

[100] Andrew B. Kahng and Seokhyeong Kang. Accuracy-Configurable Adder for Approximate Arithmetic Designs. *DAC Design Automation Conference 2012*, 2012.

[101] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo. On reconfiguration-oriented approximate adder design and its application. *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013.

[102] S. Hashemi, R. I. Bahar, and S. Reda. DRUM: A Dynamic Range Unbiased Multiplier for approximate applications. *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[103] Georgios Zervakis, Kostas Tsoumanis, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10), 2016.

[104] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. Dual-Quality 4:2 Compressors for Utilizing in Dynamic Accuracy Configurable Multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(4), 2017.

[105] Antonio Giuseppe Maria Strollo, Ettore Napoli, Davide De Caro, Nicola Petra, and Gennaro Di Meo. Comparison and Extension of Approximate 4-2 Compressors for Low-Power Approximate Multipliers. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(9), 2020.

[106] John N. Mitchell. Computer Multiplication and Division Using Binary Logarithms. *IRE Transactions on Electronic Computers*, EC-11(4), 1962.

[107] Weiqiang Liu, Jiahua Xu, Danye Wang, Chenghua Wang, Paolo Montuschi, and Fabrizio Lombardi. Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(9), 2018.

[108] Jeremy Schlachter, Vincent Camus, Krishna V. Palem, and Christian Enz. Design and Applications of Approximate Circuits by Gate-Level Pruning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(5), 2017.

[109] Ilaria Scarabottolo, Giovanni Ansaloni, and Laura Pozzi. Circuit carving: A methodology for the design of approximate hardware. *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018.

[110] Radek Hrbacek, Vojtech Mrazek, and Zdenek Vasicek. Automatic design of approximate circuits by means of multi-objective evolutionary algorithms. *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2016.

[111] Steve Ward. Performance Measures. https://computationstructures.org/notes/performance/notes.html, 2017. (Accessed on 04/15/2024).

[112] John Wawrzynek. EECS150 - Digital Design Lecture 17 - Circuit Timing [2]. http://wla.berkeley.edu/~cs150/sp13/agenda/lec/lec17-timing2.pdf, mar 2013. (Accessed on 04/15/2024).

[113] Lorenzo Mari. Basic cmos logic gates - technical articles. https://eepower.com/technical-articles/basic-cmos-logic-gates/#, oct 2021. (Accessed on 04/29/2024).

[114] Vijay Sharma, Balwinder Raj, and Manisha Pattanaik. ONOFIC approach: low power high speed nanoscale VLSI circuits design. *International Journal of Electronics*, 2013.

[115] C. Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys.

[116] Hubert Henry. Ward. *Mastering Digital Electronics : An Ultimate Guide to Logic Circuits and Advanced Circuitry*. Maker Innovations Series. Apress L. P., Berkeley, CA, 1st ed. edition, 2023.

[117] ELPROCUS. Transistor transistor logic : History, types, working & its applications. https://www.elprocus.com/transistor-transistor-logic-ttl/, 2024. (Accessed on 29/04/2024).

[118] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11--15, aug 2008.

[119] NANDLAND. What is Propagation Delay. https://nandland.com/lesson-11-what-is-propagation-delay/. (Accessed on 05/01/2024).

[120] Papers with Code - The latest in Machine Learning, apr 2024. [Online; accessed 3. Apr. 2024].

[121] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[122] H. Pishro-Nik. *Introduction to Probability, Statistics, and Random Processes*. Kappa LLC, 2014.

[123] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric Statistical Inference*. Taylor and Francis Group LLC, 5. edition, 2011.

[124] R. A. Leibler S. Kullback. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 8(1), 1951.

[125] Joram Soch, Pietro Monticone, Thomas J. Faulkenberry, Alex Kipnis, Kenneth Petrykowski, Carsten Allefeld, Heiner Atze, Adam Knapp, and Ciarán D. McInerney. *The Book of Statistical Proofs*. Zenodo, 1. edition, 2023.

[126] akilm. akilm/fpu-ieee-754: Synthesizable floating point unit written using verilog. supports 32-bit (single-precision) multiplication, addition and division and square root operations based on the ieee-754 standard for floating point numbers. https://github.com/akilm/FPU-IEEE-754/tree/main, jan 2021. (Accessed on 05/28/2024).

[127] B. N. Madhukar and R. Narendra. Lanczos Resampling for the Digital Processing of Remotely Sensed Images. In *Lecture Notes in Electrical Engineering*, volume 258 of *Lecture Notes in Electrical Engineering*, pages 403--411. Springer India, India, 2013.

[128] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv.org*, 2017.

[129] Daniel Godoy. Understanding binary cross-entropy / log loss: a visual explanation | by Daniel Godoy | Towards Data Science. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a, nov 2018. (Accessed on 05/08/2024).

[130] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal Loss for Dense Object Detection. *IEEE transactions on pattern analysis and machine intelligence*, 42(2):318--327, 2020.

[131] Anuja Nagpal. L1 and l2 regularization methods, explained | built in. https://builtin.com/data-science/l2-regularization, jan 2022. (Accessed on 05/08/2024).

[132] P.C. Mahalanobis. On the Generalised Distance in Statistics. *The journal of the Asiatic Society of Bengal*, 2(1), 1936.

Acronyms

ACA Accuracy Configurable Adder. 35
AFA Approximate Full-Adder. 35
AI Artificial Intelligence. 1
ANN Artificial Neural Networks. 2, 6
ASIC application-specific integrated circuit. 3, 47
BP Back-Propagation. 11, 12
CCSA carry-completion sensing adders. 21, 24
CGP Cartesian Genetic Programming. 3
CLA carry lookahead adders. 21, 24, 25, 35
CLI Command Line Interface. 48
CLT Central Limit Theorem. 71--73, 191
CPU Central Processing Unit. 33
CSA carry-save adder. 24, 25
CS conditional-sum adders. 21
DNN Deep Neural Network. 9, 12, 37
DRUM Dynamic Range Unbiased Multiplier. 35
DVFS Dynamic Voltage and Frequency Scaling. 34
EDA Electronic Design Automation. 48
ER Error Rate. 45, 139
FLP floating point. 17--19, 24, 31, 33, 38, 65, 72, 99, 100
FXP fixed point. 17, 19, 21, 24, 25, 31, 33, 37--39, 64, 65, 67, 68, 70, 72, 77, 93, 99, 100, 175, 176
GNN Graph Neural Networks. 12
HDL Hardware Description Language. 47
HD Hamming Distance. 45, 139
HW Hardware. 33, 34, 36, 40
IEEE The Institute of Electrical and Electronics Engineers. 18, 33
LN Logarithmic Number. 36
LOA Lower-part-OR-Adder. 35, 36
LSB least significant bit. 17, 19, 21, 23, 29, 35, 38, 72, 80, 99, 183, 185
LUT Look-up Table. 25, 53, 70, 72, 96, 141
MAC multiply-accumulate. 7, 16, 19, 24, 25, 31, 34, 38--40, 42, 67, 70, 71, 73, 75, 96, 99, 100
MHD Mean Hamming Distance. 45, 139
MLE Maximum Likelihood Estimate. 72, 73
MLP Multilayer Perceptrons. 9, 11--14
ML Machine Learning. 1, 3
MRBM Modified Radix-2 Booth Multiplier. 24
MSB most significant bit. 17--19, 21, 27, 28, 35, 36, 38
MSE mean-square error. 11
NN Neural Networks. 2, 3, 5--7, 9, 10, 12, 16, 17, 33, 34, 37--42, 57, 58, 99, 187

PDF Probability Density Function. 70
PTQ Post-Training Quantisation. 33
QAT Quantisation-Aware Training. 33, 34
RCA ripple-carry adder. 20, 21, 35
RNN Recurrent Neural Networks. 12
RV random variable. 53, 70--72, 83, 87, 91, 141, 191
RELU Rectified Linear Unit. 8, 14, 34
STE Straight-Through Estimator. 37, 75, 76, 101, 103, 173, 175, 187
SW Software. 33, 40
TTL Transistor-Transistor Logic. 51
VOS Voltage Over Scaling. 34
WCD Worst-Case Distance. 54, 56, 159--161, 171
I.I.D. independent and identically distributed. 71

AC Approximate Computing. 3, 33, 34, 40, 41, 103
API Application Programming Interface. 116, 118, 189

CNN Convolutional Neural Network. vii, 3, 4, 12--16, 33, 34, 37--40, 42, 43, 57--60, 62, 64, 69--72, 75, 80, 82, 88, 90, 91, 94--97, 100, 101, 116, 119, 134, 173, 180, 184, 185, 190

DAG Directed Acyclic Graph. vii, viii, 51--53, 90, 100, 141, 144, 147, 150, 153, 156, 159, 164, 166, 169, 170

FM Feature Map. vi, 13, 14, 67, 81, 82, 184--186
FMA fused multiply-add. vi, 31, 32

GPU Graphical Processing Unit. 31, 33, 34

PMF Probability Mass Function. vii, viii, 53, 54, 56, 70--72, 141, 144, 147, 150, 153, 156, 165, 167, 169

RTL Register Transfer Level. 40, 46--48, 100, 138

Github References A

```
/
├── rtl_analysis/
│   ├── figures/
│   ├── Error/
│   ├── netlist/
│   ├── summary/
│   ├── Makefile
│   ├── gates.py
│   ├── netlist.tcl
│   └── paths.py
├── Perceptron
├── statistic_test_3_models/
│   ├── NoisyLayers.py
│   ├── statistical_test_3_models.py
│   └── test_custom_layers.py
├── STEP_III_walkthrough
│   ├── Makefile
│   ├── gates.py
│   ├── paths.py
│   ├── summary/
│   ├── figures/
│   ├── netlist/
│   ├── Error/
│   └── statistic_test_3_models/
│       ├── NoisyLayer.py
│       └── statistical_test_3_models.py
├── app-small_network/
│   ├── functions/
│   └── results/
└── convergence_of_stat_and_approx_model/
    ├── mul8s_1kv8_stats_and_approx.py
    ├── mul8s_1kv8_stats_and_approx.csv
    ├── mul8s_1kv9_stats_and_approx.py
    └── mul8s_1kv9_stats_and_approx.csv
```

Defining small CNN for benchmarking

B

This appendix walks through the design process of a CNN (Convolutional Neural Network). The small CNN will be implemented in C++ as the *small scale neural network using approximate arithmetic* and will be an integral part of the benchmarking tool. The methods, procedure, results, and analysis will be presented in aptly named sections and the outcome of this appendix will come in the form of a small-scale CNN.

In Appendix A every script, network summary, and resulting data can be found under `/app-small_-network/`.

B.1 Introduction

In this appendix a small-scale CNN will be created by heuristically ‘‘optimising’’ one element at a time. The chosen machine learning API (Application Programming Interface) is TensorFlow and the testing will be limited to readily available methods, objects, etc. From TensorFlow the basic CNN model from example Convolutional Neural Network (CNN) is taken and utilised as the base network. The example CNN performs **classification** on the *cifar10* dataset, which is a set of 60.000 (32×32)-images in colour divided in 10 classes. The proposed model has a total of 122570 total (and trainable) parameters and reaches a test accuracy of ~ 71%. It is deemed, that it would be beneficial to reduce the number of parameters (i.e. simplify the network) since the same network will be implemented in C++ from scratch.

The modifications will be divided into 4 categories with subentries of:

- **Dataset**
  - Dimensionality Reduction
  - Task simplification/expansion (e.g. adding/removing classes in the case of classification)
- **Optimizer**
  - Optimisation algorithm
  - Loss function
- **Model**
  - Layers (type, count, width, etc.)
  - Activation function
  - Kernel size
  - Pooling operation
- **Final adjustments**
  - Number of classes
  - Bias

- Regularization

Adjusting anything from the list above will change the cost landscape and thus an optimal model would require extensive research and tests. This would require a lot of time, however, the return from the time investment would be minimal as this model will only be compared to itself with slight modifications. Instead, the model is heuristically modified based on the list above; the dataset will be simplified first in section B.3, followed by the optimizer in section B.4, the model itself will be modified in section B.5, and lastly the final adjustments of the model are made in .

B.2 Workspace Setup, Software and Hardware

All tests are run on the same hardware, i.e. an available cloud-computer has been set up as seen in Table B.1:

Table B.1: Information about the cloud-computer used in this appendix. Information taken from `https://strato-new.claaudia.aau.dk/` under the *Instances* tab.

Flavor Name	Flavor ID	RAM	VCPUs	Disk
AAU.CPU.g.16-64	10a7313a-2e8c-421a-9cdc-8e1283ef905b	64GB	16 VCPU	50GB

As mentioned in section B.1 an eexample network is used as the baseline, whereon modifications will be performed. The specifics of the baseline network can be seen in Table B.2 and is updated when modifications are performed.

Table B.2: Summary of the base model found by calling `model.Summary()`.

Layer Type	Output Shape	Params #
Conv2D	(None, 30, 30, 32)	896
MaxPooling 2D	(None, 15, 15, 32)	0
Conv2D	(None, 13, 13, 64)	18496
MaxPooling 2D	(None, 6, 6, 64)	0
Conv2D	(None, 4, 4, 64)	36928
Flatten	(None, 1024)	0
Dense	(None, 64)	65600
Dense	(None, 100)	6500

Table B.3: The software packages used in the appendix. The names and versions were found using the `print-versions` package in python.

Package	Explanation	Version

Continued on next page

Table B.3: (Continued)

`numpy`	Math package	1.26.4
`pandas`	Math package used for storing resulting data in ‘‘.csv’’ files	2.2.1
`tensorflow`	The chosen machine learning API, utilised for data manipulation, loading datasets, etc.	4.9.4
`tensorflow_datasets`	Used for dedicated dataset manipulation	4.9.4
`keras`	Used to create and compile the models	3.1.1

B.3 Dataset Preparation

Although the example from TensorFlow already has a dataset, it is relevant to consider alternatives. Due to the arbitrary choice of image classification, the **data collection** can be reduced to downloading a premade dataset. The choice of the dataset will be based on the following parameters:

Table B.4: Three important parameters for the choice of a dataset

Size	The dataset should be large enough to fully train a CNN, without requiring gathering more data than is already available
Difficulty	The difficulty should be high enough to make it a challenge for state-of-the-art image classification networks
Configurability	Combined with the difficulty there should be a way to configure the difficulty of the problem, to accommodate for a small-scale network as well as a large-scale network

MNIST [50] is a classical example of image classification, the size is sufficient and visualization is simple. However, it is not a difficult task to classify the dataset. Using the 92 submissions on Image Classification on MNIST [120] as an indication of the difficulty and noting the lowest accuracy is 92,47 % it is clear to see, that it is not a difficult classification problem. Another well-known image classification dataset is the CIFAR-10 [121] dataset. Again, the size is sufficient, visualisation is simple, and the difficulty is increased compared to MNIST. Based on 240 submissions on Image Classification on CIFAR-10 [120] the lowest score is an accuracy of 80,45 %. The configurability of the dataset is decent with 10 classes, however, CIFAR-10 has a sibling dataset: CIFAR-100 with 100 classes. The increase in difficulty from CIFAR-10 to CIFAR-100 is notable, however, not overwhelming for state-of-the-art models. The span of accuracies for models *not using extra training data* on CIFAR-10 from Image Classification on CIFAR-10 is 60,6 % to 99,5 %, whereas the models *not using extra training data* on CIFAR-100 from Image Classification on CIFAR-100 [120] is 19,49 % to 93,36 %. CIFAR-100 is chosen to be the dataset on with which the models will be trained on.

The **data representation** of the CIFAR-100 is sets of images and their corresponding labels. The images are of the shape 32, 32, 3, with each value in formatted as an unsigned 8-bit integer. In Figure B.1 five examples are depicted: a *raccoon*, a *cloud*, a *lamp*, a *keyboard*, and a *beetle*, respectively.

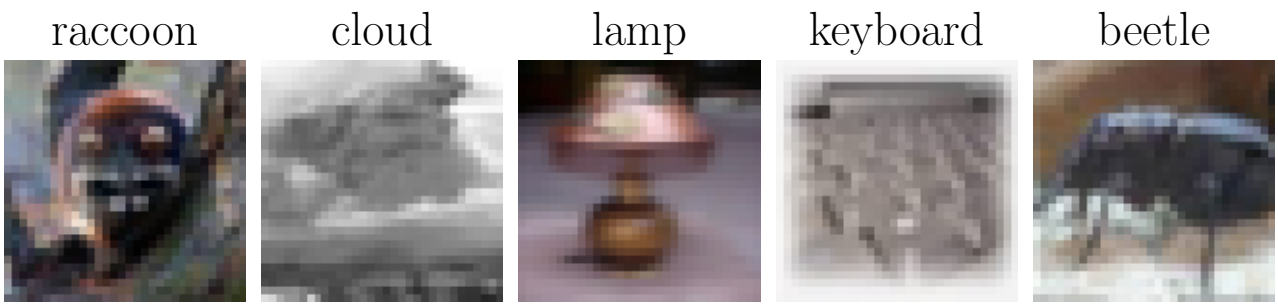

Figure B.1: 5 example images from CIFAR-100 represented as 32×32 colour images.

In the creation of a small-scale neural network, the complexity of the model should be proportionate with the complexity of the dataset. For the small-scale network, the data should be ''simpler'', why the first layer of the CNN can be reduced. The dataset can be reduced to 1/3 of its size by converting to **grayscale**. Multiple ways of converting images to grayscale exists, however, most of the models are concerned with the perception of the human-eye and is just a weighted average over the colours for each pixel, the grayscale conversion will be performed as the mean of the RGB-values.

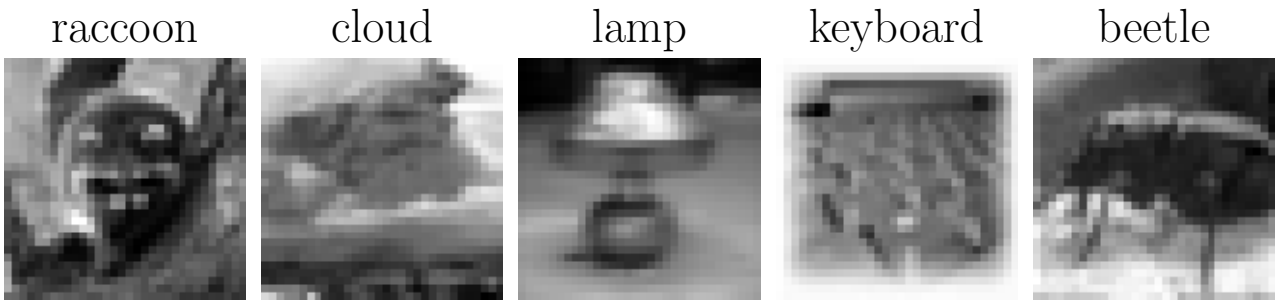

Figure B.2: 5 example images from CIFAR-100 represented as 32×32 grayscale images.

Each datapoint is still comprised of $32 \cdot 32 = 1024$ 8-bit integers, which is incongruent with a small-scale network. The dataset can be reduced further to 1/12 of its original size, by compressing the images to 16×16 grayscale. Multiple resizing methods exists, and is tested using the procedure presented in subsection B.3.1, with the results presented in subsection B.3.2.

Some changes have to be made on the model to accommodate the chosen (and modified) dataset:

- **Adjust kernel size:** Since the images have been resized to $(16,\ 16)$, the kernel size produced an error: `ValueError: Exception encountered when calling Conv2D.call(). Negative dimension size caused by subtracting 3 from 2`, which was solved by changing the kernel size to $(2,\ 2)$.
- **Adjust output layer:** The output of the network should be directly comparable to the labels, which have been one-hot encoded; the output vector should be $(100, 1)$ corresponding with the 100 classes.

The resulting model can be seen in Table B.5.

Table B.5: Summary of the base model found by calling `model.Summary()`.

Layer Type	Output Shape	Params #
Conv2D	(None, 15, 15, 32)	160
MaxPooling 2D	(None, 7, 7, 32)	0
Conv2D	(None, 6, 6, 64)	8256
MaxPooling 2D	(None, 3, 3, 64)	0
Conv2D	(None, 2, 2, 64)	16448
Flatten	(None, 256)	0
Dense	(None, 64)	16448
Dense	(None, 100)	6500

† Total parameters 47812 and trainable parameters 47812.

Note: Orange rows have been modified to accommodate the (modified) dataset.

The total amount of parameters have thus been reduced from 128420 to 47812 or reduced by 63 %.

B.3.1 Procedure

The procedure of the *dataset preparation* is split into two parts: `create_dataset` and `test_resize_method`.

In `create_dataset`, the dataset is downloaded and modified:

I) Import required packages (a subset of packages displayed in Table B.3)
II) Set settings:
- enable grayscale
- enable dimensionality
- enable normalisation
- choose resize method and the wanted size of the datapoints

III) Save summary of the settings
IV) Import dataset (CIFAR-100)
V) Convert to grayscale
VI) Resize
VII) Normalise
VIII) Save modified dataset

The above procedure is performed eight times; one for each resizing method that works out-of-the-box. With the saved (and modified) datasets, the `test_resize_method` is run:

I) Import packages (a subset of packages displayed in Table B.3)
II) Define paths for the datasets
III) Load dataset
IV) Format the dataset
- Change labeling to one-hot-encoding
- Create batches of the training data (512)

- Create batches of the test data (512)

V) Create the model
VI) Compile the model (choose optimizer algorithm, loss function, and metric)
VII) Fit the model over 250 epochs

Items III through VII are repeated for each resize method available from the *TensorFlow Dataset* package.

B.3.2 Results

The *loss* of the resizing methods can be viewed in Figure B.3 and the corresponding accuracy can be viewed in Figure B.4. Furthermore, the mean time per epoch for all eight resizing methods can be viewed in Table B.6.

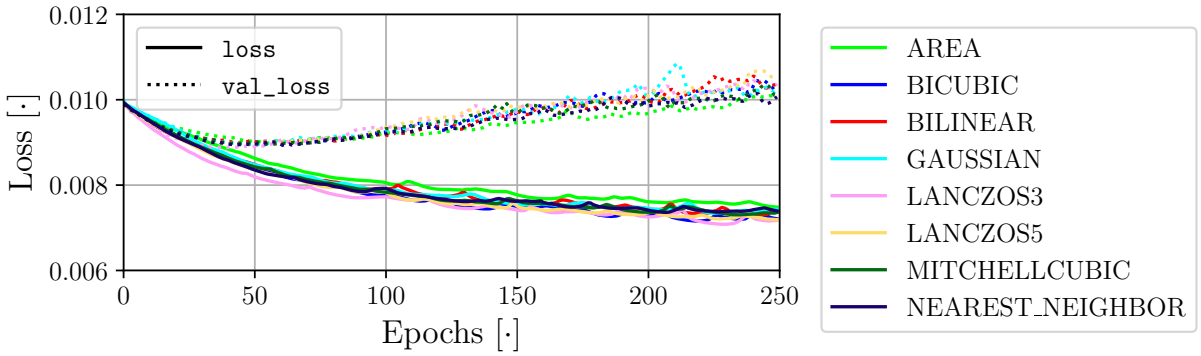

Figure B.3: The loss as a function of epoch training the same model on differently resized datasets. The loss function is `MeanSquaredError` and the optimisation algorithm is `adam`. The evaluated training losses are depicted as fully-drawn lines, whereas the evaluated testing losses are depicted as dotted-lines.

From Figure B.3 no single resizing method is objectively better than the rest, as all training losses follow a similar path, and the same is true for the testing losses. It is noted, that the all models are seemingly full trained at 50 epochs, and further training has resulted in overtraining. At 50 epochs all testing losses are almost identical, however, the training loss for `LANCZOS3` is a bit lower than the rest.

In Figure B.4 the accuracy of classification on the testing examples reaches just above 20 % using some of the resizing methods. It is noted, that up to around the 50 epoch point, `LANCZOS3` is performing slightly better in the testing accuracy, but significantly better in the training accuracy.

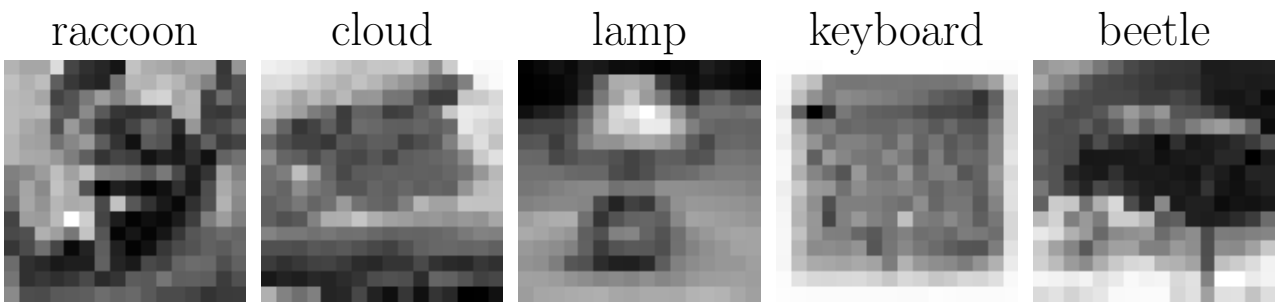

Figure B.5: 5 example images from CIFAR-100 represented as 16 × 16 grayscale images.

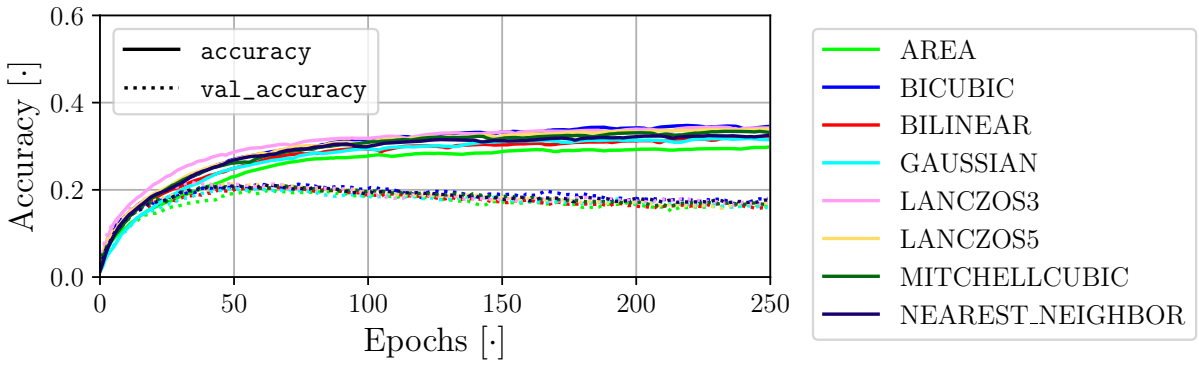

Figure B.4: The accuracy as a function of epoch training the same model on differently resized datasets. The loss function is `MeanSquaredError` and the optimisation algorithm is `adam`. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

In Table B.6 the mean time per epoch with different resizing methods applied to the CIFAR-100 dataset can be viewed. Since every datapoint is of the same size/shape (16, 16), the time required per epoch should be very similar if not identical, which is cooperated by the values in the table. The discrepancies are insignificant and there is no resizing method that results in the ‘‘fastest’’ times.

Table B.6: The mean epoch time in [s] for each tested resizing method. one epoch includes training, and testing a batch of 512.

Resize Method	Time [s]
AREA	1,49
BICUBIC	1,47
BILINEAR	1,47
GAUSSIAN	1,49
⋮	
LANCZOS3	1,50
LANCZOS5	1,49
MITCHELLCUBIC	1,47
NEAREST_NEIGHBOR	1,46

`LANCZOS3` is chosen as the preferred resizing method, due to the slightly better accuracy in the lower epochs. Using the same examples as Figure B.1 and Figure B.2, the images will be reduced to what can be viewed in Figure B.5:

The outcome of the *dataset preparation* is thus a complexity reduced CIFAR-100 dataset consisting of 50.000 training examples, 10.000 test examples, in the shape (16, 16) of 8-bit integers (using grayscale and `LANCZOS3`). The labels are converted to one-hot encoding.

Task Simplication was not tested, however, it is noted, that due to the dataset being comprised of 100 classes, the problem can be ''simplified'' by removing classes.

B.4 Optimizer Preparation

Opposite to the *dataset preparation* the outcome of the *optimizer preparation* will not reduce the amount of trainable parameters, since the model itself is not affected. Nonetheless, it is essential to optimise the performance, by choosing a fitting **optimisation algorithm** and **loss function**.

B.4.1 Procedure

The *optimizer preparation* will follow a heuristic approach, wherewith the technique resulting in the ''best performance'' is chosen and subsequent testing will not revise the choice. Firstly, the optimisation algorithm is found:

I) Import necessary packages
II) Load dataset (CIFAR-100, grayscale, 16×16)
III) Format the dataset
- Change labeling to one-hot-encoding
- Create batches of the training data (512)
- Create batches of the test data (512)

IV) Create the model
V) Compile model (Choose optimizer algorithm, loss function and metric are not changed)
VI) Fit the model over 250 epochs

Items IV through VI are repeated for each optimisation algorithm available from *TensorFlow*. The script performing the *optimisation algorithm* search is called `test_optimizer` and can also be found in Appendix A.

A similar procedure is performed to find the *loss function*, however, now the newfound *optimisation algorithm* will be integrated into the compilation of the model:

I) Import necessary packages
II) Load dataset (CIFAR-100, grayscale, 16×16)
III) Format the dataset
- Change labeling to one-hot-encoding
- Create batches of the training data (512)
- Create batches of the test data (512)

IV) Create the model
V) Compile model (Choose loss function, optimizer algorithm and metric are not changed)
VI) Fit the model over 250 epochs

Items IV through VI are repeated for each loss function available from *TensorFlow*. The script performing the *loss function* search is called `test_loss` and can also be found in Appendix A.

B.4.2 Results

The *loss* of the different optimisation algorithms can be viewed in Figure B.6 and the corresponding accuracy is available in Figure B.7. Furthermore, the mean time per epoch for all ten optimisation algorithms can be viewed in Table B.7.

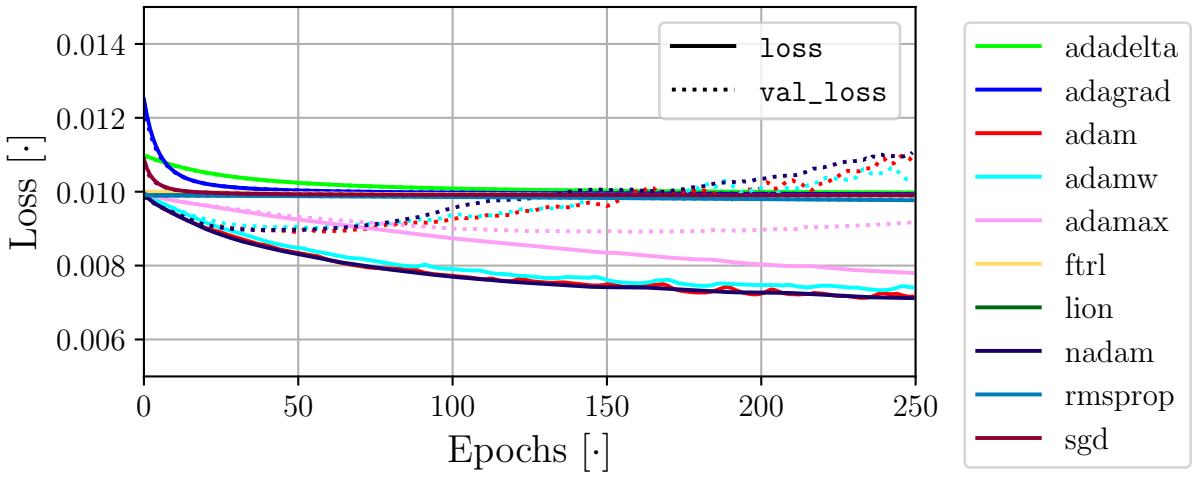

Figure B.6: The loss as a function of epochs, training the same model on the same dataset using different optimisation algorithms. The *loss function* is `MeanSquaredError`. The eavluated training losses are depicted as fully-drawn lines, whereas the evaluated testing losses are depicted as dotted-lines.

In Figure B.6 there is a clear difference between the optimisation algorithms. `adamw`, `nadam`, and `adam` reaches their lowest loss very fast (around 30 epochs), however, they all suffer greatly from overtraining after 50 epochs. `adamax` takes longer to reach its lowest loss, however, the effects of overtraining are diminished. All four reach almost the same lowest loss at some point in their training.

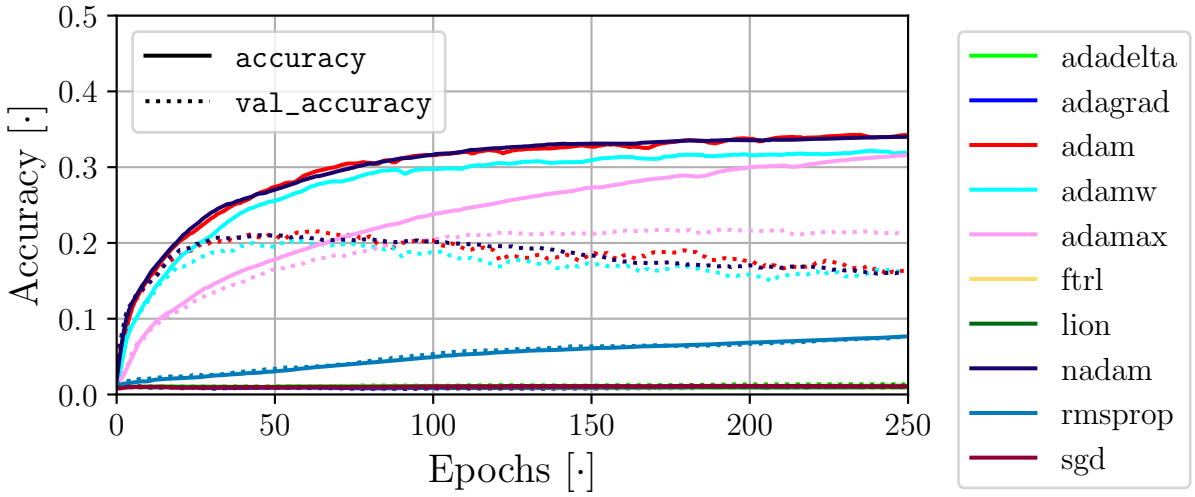

Figure B.7: The accuracy as a function of epoch, training the same model on the same dataset using different optimisation algorithms. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

Figure B.7 depicts the consequences of Figure B.6; `adam`, `adamw`, and `nadam` quickly reach their respective peaks, at around the same epoch as their lowest loss. However, the effect of overtraining are applied to the accuracies, and they each fall with around five percentage points. `adamax` is slower, but reaches around the same accuracy as the others, and retains the accuracy.

In Table B.7 the mean duration of an epoch using the different optimisation algorithms can be seen. Some of the times are notably lower, however, their losses and accuracies were not sufficient. W.r.t. the four ‘‘best’’ optimisation algorithm, based on the loss and accuracy in Figure B.6 and B.6, there exists no meanningful difference.

Table B.7: The mean epoch time in [s] for each tested resizing method. One epoch includes training, and testing a batch of 512.

Resize Method	Time [s]
adadelta	1,51
adagrad	1,50
adam	1,50
adamw	1,51
adamax	1,50
⋮	
ftrl	1,45
lion	1,44
nadam	1,52
rmsprop	1,48
sgd	1,46

Based on the stability of the test loss/accuracy, `nadam` is chosen out of the four optimisation algorithms. In the following test where the *loss function* is changed, the optimisation algorithm will thus be `nadam`.

Unlike the previous tests, comparing the losses of the different runs will not yield useful information, since the loss is differently defined. Instead, the accuracies can be viewed in Figure B.8

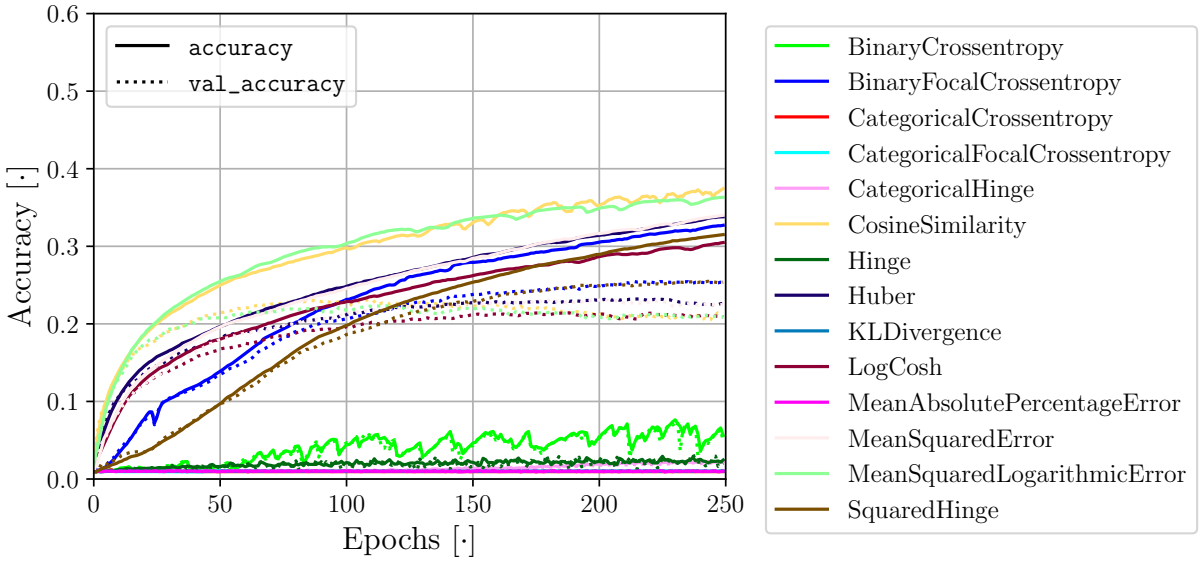

Figure B.8: The accuracy as a function of epochs, training the same model on the same dataset using different loss functions. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

In Figure B.8 using `MeanSquaredLogarithmicError` and `CosineSimilarity` is the fastest method to reach above 20 % accuracy. However, `BinaryFocalCrossentropy` and `SquaredHinge` reach a higher accuracy of around 25 % at their peaks. The accuracy using `BinaryFocalCrossentropy` is for the most epochs slightly higher than using `SquaredHinge`.

Comparing the mean duration of an epoch between using `BinaryFocalCrossentropy` and `SquaredHinge`, it is clear that using `SquaredHinge` is notably slower. However, it is deemed more important to have the higher accuracy.

Table B.8: The mean epoch time in [s] for each tested resizing method. one epoch includes training, and testing a batch of 512.

Resize Method	Time [s]
BinaryCrossentropy	1,59
BinaryFocalCrossentropy	1,63
CategoricalCrossentropy	1,56
CategoricalFocal-Crossentropy	1,59
CategoricalHinge	1,52
CosineSimilarity	1,54
Hinge	1,54
⋮	
Huber	1,57
KLDivergence	1,57
LogCosh	1,57
MeanAbsolutePercentage-Error	1,51
MeanSquaredError	1,54
MeanSquaredLogarithmic-Error	1,58
SquaredHinge	1,56

From comparison between using the different *loss functions*, a choice is made, and BinaryFocalCrossentro will be utilised going forward, due to the slightly higher accuracy.

B.5 Model Architecture Preparation

Before the architecture of the model is tuned, the dataset, optimisation algorithm, and loss function have been adjusted. The current goal is to create a model, with relatively few parameters. The model was modified to accommodate the new data, which reduced the amount of parameters from 128.420 to 47.812 and given adamax as the optimisation algorithm and BinaryFocalCrossentropy as the loss function, the model has reached an accuracy of around 25 %.

If possible the goal is to reduce the amount of parameters and increase the accuracy.

B.5.1 Procedure

The possible combination to produce a model is incredibly large, and some simplifications have to be made, to make the search for an efficient model be done in a timely manner. Firstly, it is decided, that the width and depth of the model will not be changed at the same time; i.e. if a layer is added/removed from the base model, none of the layers can be modified at the same time and vice versa. Furthermore, the layers will be restricted to be the same types as the baseline: Conv2D, MaxPooling2D, Flatten, and Dense. The kernel size and activation functions will not be changed. The Flatten layer and the output layer Dense(100) will not be touched, as they ensure the output is in the right format.

The models are based on the baseline. However, they each have an equal amount of the different layer types (disregarding the Flatten-layer) per model. A summary of the models can be seen in Table B.9:

Table B.9:

Name	Layers (in-order)	Total Params #
Base Model	`Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Conv2D`, `Flatten`, `Dense`, `Dense(100)`	47812
One-Of-Each Model	`Conv2D`, `MaxPooling2D`, `Flatten`, `Dense(100)`	157060
Two-Of-Each Model	`Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Flatten`, `Dense`, `Dense(100)`	51844
Three-Of-Each Model	`Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Flatten`, `Dense`, `Dense`, `Dense(100)`	39684
Four-Of-Each Model	`Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Conv2D`, `MaxPooling2D`, `Flatten`, `Dense`, `Dense`, `Dense`, `Dense(100)`	48004

For the modification of the *depth* of the model, the procedure will follow:

- Import necessary packages
- Load dataset (CIFAR-100, grayscale, 16×16)
- Format the dataset
  - Change labeling to on-hot-encoding
  - Create batches of the training data (512)
  - Create batches of the testing data (512)
- Choose a model
- Compile the model (`adamax` is used as the optimisation algorithm, `BinaryFocalCrossentropy` is used as the loss function)
- Fit the model over 250 epochs

Steps IV through VI is repeated for all the prepared models. Based on the accuracy and the amount of parameters, a model is chosen and the *width* will be examined.

After the *depth* of the model has been set, the width is the subject of change. To keep it simple, the chosen model will be compared to models where the ‘‘number of filters’’ in the `Conv2D`-layer and the amount of nodes in the `Dense`-layers are equal. To generate all the comparison models, the number of filters/nodes starts at 16 and are incremented by eight until the number has reached 128.

The procedure for the *width* will follow:

- Import necessary packages
- Load dataset (CIFAR-100, grayscale, 16×16)
- Format the dataset
  - Change labeling to on-hot-encoding
  - Create batches of the training data (512)
  - Create batches of the testing data (512)
- Choose a model
- Compile the model (`adamax` is used as the optimisation algorithm, `BinaryFocalCrossentropy` is used as the loss function)
- Fit the model over 250 epochs

B.5.2 Results

The resulting accuracies of the models presented in Table B.9 can be seen in Figure B.9. The base models is significantly outperforming the other models w.r.t. its testing accuracy being multiple percentage points higher than the second best: `twoofeach_model`. Furthermore, in Table B.9 it can be seen, that `twoofeach_model` has more parameters than the `Base Model`; 51844 parameters compared to the 47812. With the decrease in accuracy and increase amount of parameters, there is no reason to further investigate any other model than the `Base Model`.

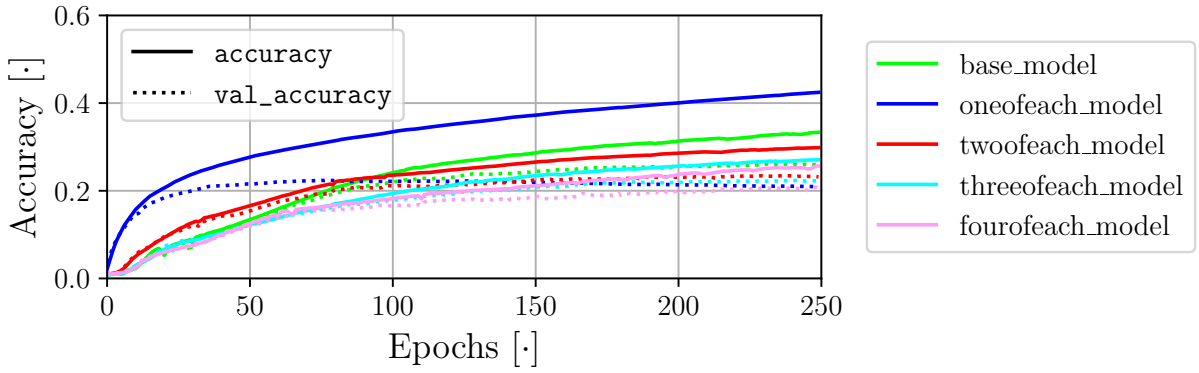

Figure B.9: Accuracy as a function of epochs, training models of different depth on the same dataset. The models can be seen in Table B.9. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

Testing the `Base Model` with varying width as described in subsection B.5.1, the resulting accuracies can be viewed in Figure B.10:

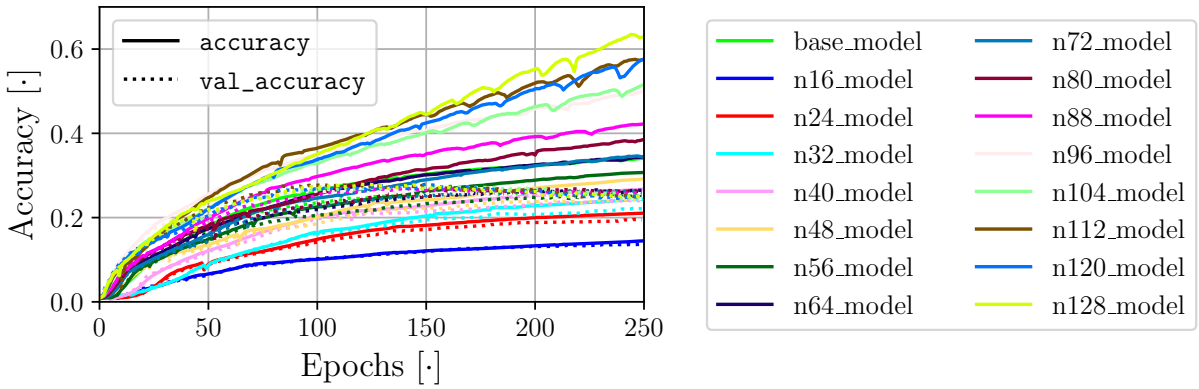

Figure B.10: The accuracy as a function of epochs, training modified versions of the base model using the same dataset. The evaluated training accuracies are depicted as fully-drawn lines, whereas the evaluated testing accuracies are depicted as dotted-lines.

There is no meaningful increase in accuracy from the `Base Model`, why all the models with more weights can be discarded. The remaining models and their respective amount of parameters can be found in Table B.10.

Table B.10: Base model and the model with fewer parameters. For each model the number of parameters and the mean time per epoch are listed.

Name	Total Params #	Time [s]
Base Model	47812	1,61
n16 Model (Base)	4900	1,05
n24 Model (Base)	9604	1,16
n32 Model (Base)	15844	1,40
n40 Model (Base)	23620	1,60
n48 Model (Base)	32932	1,84
n56 Model (Base)	43780	2,15

From Table B.10 and Figure B.11 it can be seen that `n40 Model` has nearly the same testing accuracy as the `Base Model`, `n48 Model`, and `n56 Model`, however, the total amount of parameters are half of the `Base Model`. Furthermore, the `n40 Model` is just as fast per epoch as the `Base Model`, and significantly faster both `n48 Model` and `n56 Model`.

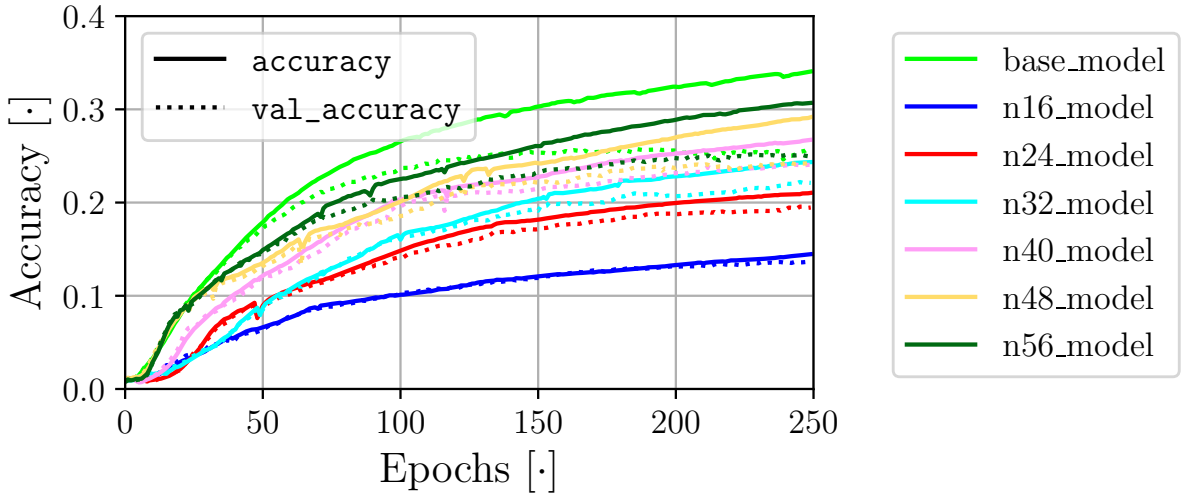

Figure B.11: Repeat of Figure B.10 with the discarded networks removed.

The small drop in testing accuracy and the large drop in the number of parameters from `Base Model` to `n40 Model` is deemed to be a good tradeoff.

B.6 Final Adjustments

For the *final adjustments* the current model-so-far is reviewed: The `n40 Model` chosen in section B.5 reaches around a 25 % accuracy on the test data. This is viewed as a potential problem, as approximate computing is relevant in scenarios where there is room for error. However, due to the choice of a dataset of 100 classes, some of the classes may be removed to increase the accuracy, which will be done in this section. Furthermore, the bias terms are temporarily removed to see the effect of the values since no biases would simplify the C++ implementation. Lastly, to smooth out the accuracy and avoid overtraining, regularisation is performed.

The goal of this section is to have a model with ~ 50 % test accuracy, with a smooth curve and no overfitting.

B.6.1 Procedure

The *final adjustments* will be performed over two tests: Number of classes with/without bias and regularisation.

I) Import necessary packages
II) Load dataset (CIFAR-100, grayscale 16 × 16)
III) format the dataset
- Change labeling to one-hot-encoding
- Create batches of the training data (32)
- create batches of the test data (32)

IV) Create the model
V) Compile model with optimiser found in section B.4
VI) Fit the model over 250 epochs
VII) Remove five classes and repeat III through VI (100 to 5 in decrements of 5)
VIII) Repeat III through VII, where in step IV the biases are removed

The script performing the *number of classes with/without bias* can be found in Appendix A under `/app-small_network/functions/` and are called `classes_with_bias.py` and `classes_without_bias.py`.

For the *regularisation*, three types are examined: L1, L2, and L1&L2. The procedure will follow

I) Import necessary packages
II) Load dataset (CIFAR-100, grayscale 16 × 16)
III) format the dataset
- Change labeling to one-hot-encoding
- Create batches of the training data (32)
- create batches of the test data (32)

IV) Create three models with regularisation (L1, L2, or L1&L2)
V) Compile models with optimiser found in section B.4
VI) Fit the models over 250 epochs
VII) Change lambda value for the regularisation and repeat III through V

The lambda values, that will be tried can be found in Table B.11.

Table B.11: λ for the regularisation.

λ	0,001	0,01	0,05	0,1	0,3	0,5

The script performing the *regularisation* can be found in Appendix A under `/app-small_network/functions/` and is called `regularization.py`.

B.6.2 Results

In Figure B.12, B.13, B.14, and B.15 the result of *number of classes with/without bias* can be viewed. The difference between using bias and not using bias is mostly insignificant, however, for 5, 10, 15, and 60 classes there are differences. For 5 classes the model is trained faster, but they reach around

the same maximum validation accuracy. For 10 classes using bias outperforms not using bias. For 15 and 60 classes not using bias outperforms using bias.

Given the descrepancies noted above, it is assumed that using bias for this specific scenario does not significantly affect the accuracy; the differences are probably caused by the start guess since the value of the weights are random at first. Bias will not be included in the model.

Due to the goal of 50 % accuracy the model will perform classification on **10 classes** (see Figure B.12).

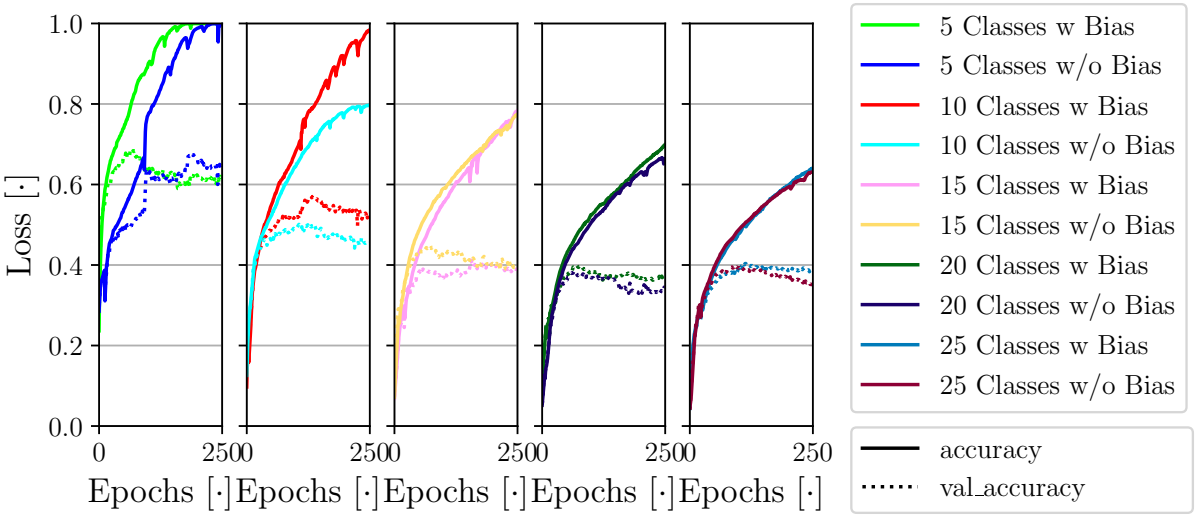

Figure B.12: Accuracy and validation accuracy plotted as a function of number of epochs in the interval [0, 250]. From left to right, the number of classes to perform classification on is incremented by 5, and each combination is tried with and without bias.

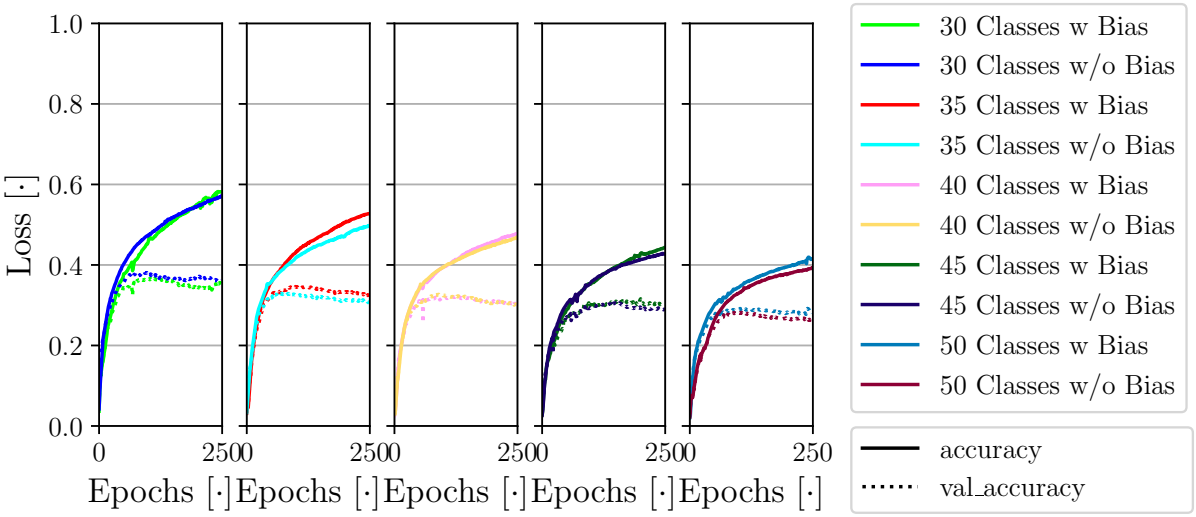

Figure B.13: Accuracy and validation accuracy plotted as a function of number of epochs in the interval [0, 250]. From left to right, the number of classes to perform classification on is incremented by 5, and each combination is tried with and without bias.

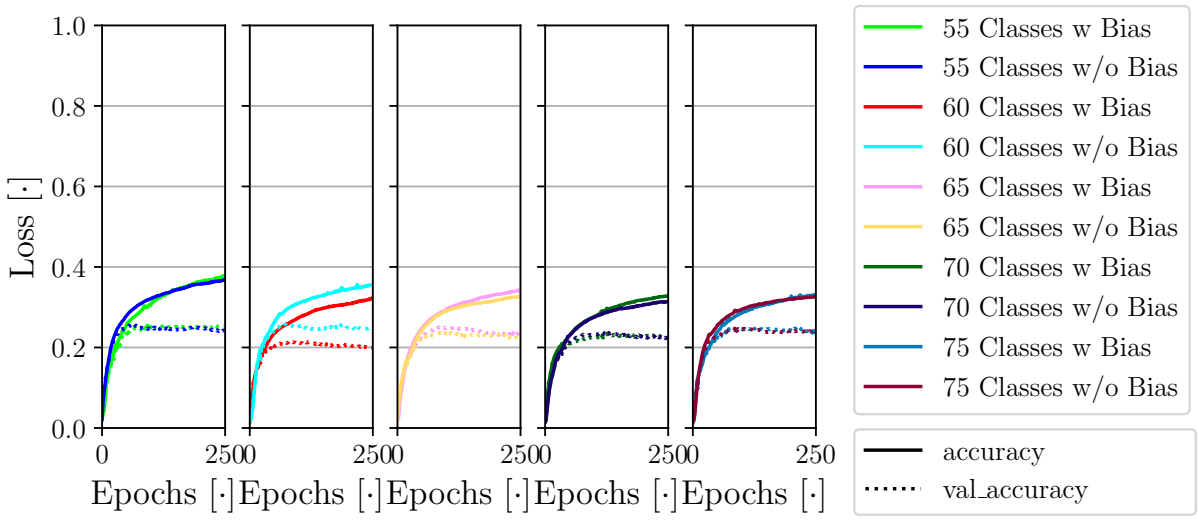

Figure B.14: Accuracy and validation accuracy plotted as a function of number of epochs in the interval [0, 250]. From left to right, the number of classes to perform classification on is incremented by 5, and each combination is tried with and without bias.

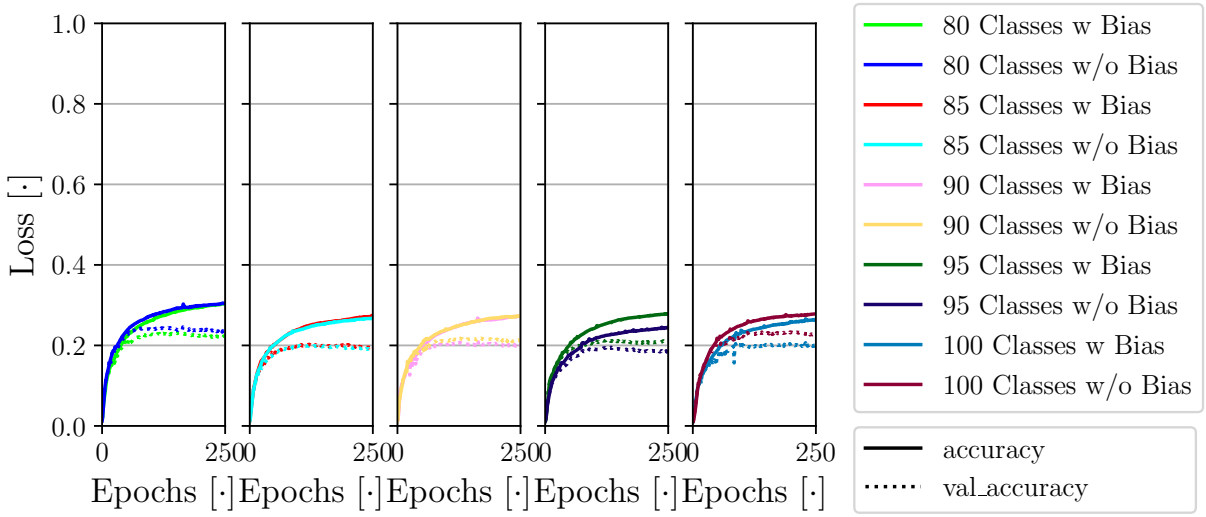

Figure B.15: Accuracy and validation accuracy plotted as a function of number of epochs in the interval [0, 250]. From left to right, the number of classes to perform classification on is incremented by 5, and each combination is tried with and without bias.

Table B.12: Calculated mean time per epoch in seconds for Figure B.12 and B.13.

Classes	5	10	15	20	25	30	35	40	45	50
w bias	0.51 s	1.11 s	1.75 s	2.23 s	2.76 s	3.32 s	3.94 s	4.37 s	4.84 s	5.71 s
w/o bias	0.51 s	1.10 s	1.56 s	1.94 s	2.52 s	3.18 s	3.56 s	4.04 s	4.56 s	5.03 s

Table B.13: Calculated mean time per epoch in seconds for Figure B.14 and B.15.

Classes	55	60	65	70	75	80	85	90	95	1000
w bias	6.19 s	6.51 s	6.97 s	7.52 s	8.00 s	8.55 s	8.97 s	9.62 s	10.58 s	11.31 s
w/o bias	5.77 s	6.10 s	6.67 s	6.91 s	7.37 s	7.92 s	8.59 s	9.32 s	9.82 s	11.43 s

In Figure B.16, B.17, and B.18 the accuracy is plotted against the epochs for L1, L2, and L1L2 regularisation for the different values from Table B.11.

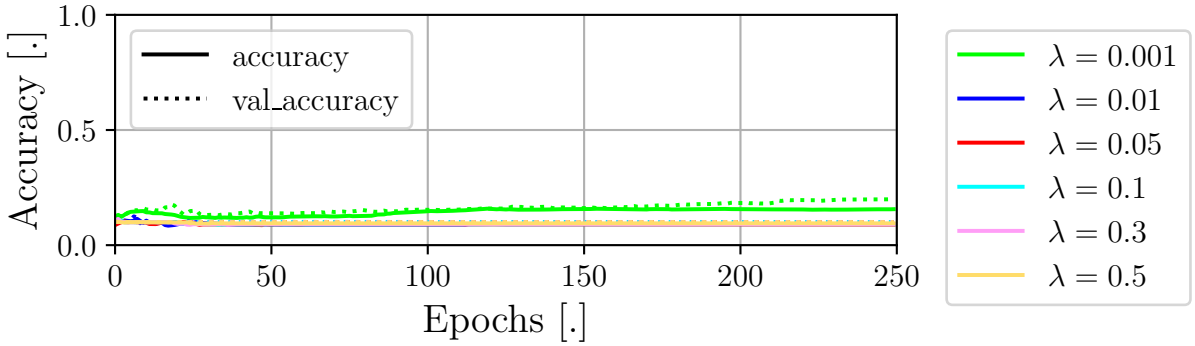

Figure B.16: L1 regularisation with different λ-values. Accuracy as a function of epochs.

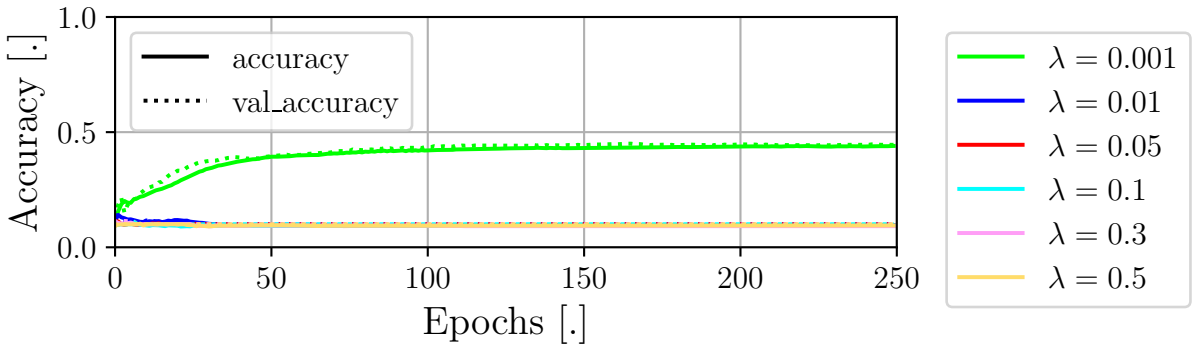

Figure B.17: L2 regularisation with different λ-values. Accuracy as a function of epochs.

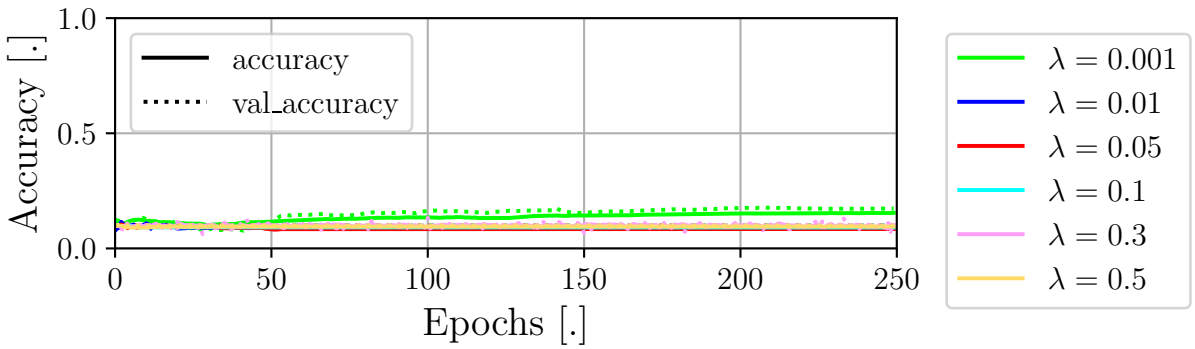

Figure B.18: L2 regularisation with different λ-values. Accuracy as a function of epochs

It is clear from Figure B.16, B.17, and B.18, that most of the chosen λ-values are affecting the accuracy of the model very negatively, and they fall to an accuracy of ~ 10 %, which is corresponding

to a random guess. However, it is noticed that using L2 regularisation with a λ-value of 0.001 is the best of the bunch, and increasing this value seems to cause a decrease in accuracy. In Figure B.19 an extra set of λ-values have been tested, all yielding better results than the previous.

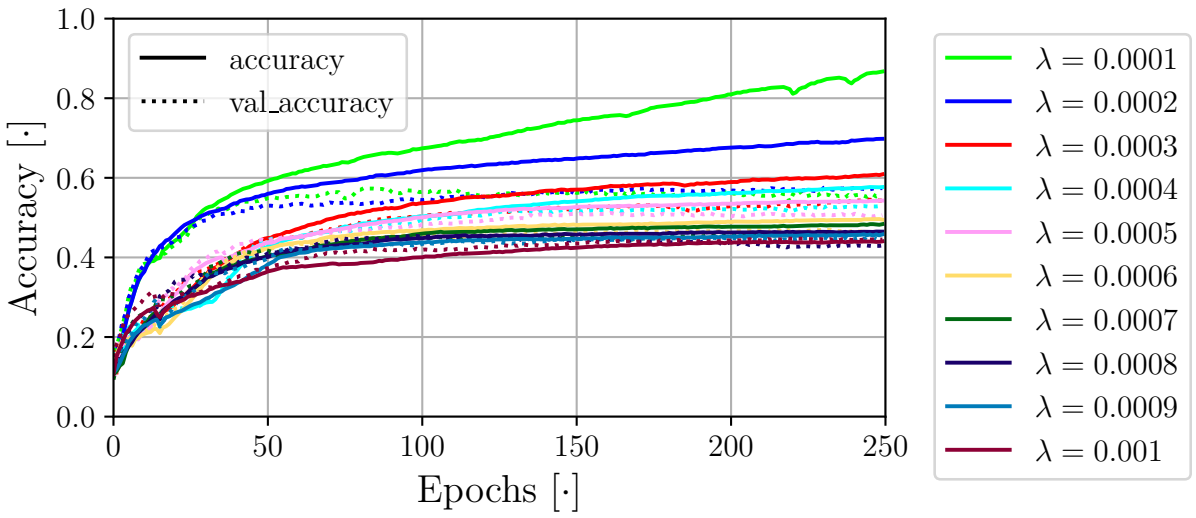

Figure B.19: L2 regularisation with adjusted λ-values.

From Figure B.19 it is clear that the values 0.0001 and 0.0002 are the best performing of the tested values. After 25 epochs, $\lambda = 0.0001$ reaches a higher accuracy than $\lambda = 0.0002$, however, $\lambda = 0.0002$ overtakes at around 125 epochs. Furthermore, both `accuracy` and `val_accuracy` are notably less ragged for $\lambda = 0.0002$, why this value is chosen.

B.7 Conclusion

The designed small-scale network is based on the example CNN presented in Convolutional Neural Network (CNN). The dataset was changed from CIFAR-10 to CIFAR-100 to make the machine learning problem more difficult, with the benefit of many classes. The amount of classes can be changed, and thus the difficulty of the problem faced by the network can be adjusted, if need be. The CIFAR-100 dataset was converted to grayscale and the size of each image was changed from (32, 32) to (16, 16) using `LANCZOS3`. Due to the changed (and modified) dataset, some modifications of the model followed: The kernel size of the convolutional and pooling layers were changed from (3, 3) to (2, 2) and the output layer had to be increased from 10 to 100 to accommodate the new number of classes.

By heuristically testing the parameters of the network's optimizer, the chosen optimisation algorithm was `adamax` and the loss function was chosen to be `BinaryFocalCrossentropy`. The combination of these two choices resulted in a network with an accuracy of around 25 %.

The model itself was also modified by heuristically choosing the best model from sets of models. The depth of the model remained unchanged as none of the other models retained the accuracy without negatively affecting the number of parameters. However, testing the model with different widths, a modified model emerged with slightly worse accuracy but a steep drop in number of parameters.

This model was tuned to around a 50 % accuracy by reducing the set of classes to 20. The effect of having bias in this scenario was also while changing the number of classes, and the difference was deemed to be caused by the ''random guess'' with which the weights are initialised. The bias

was removed from the model, further reducing the amount of parameters, and slightly simplifying implementation. To avoid overtraining and smooth out the training and testing accuracies, regularisation was explored. The initial λ-values were tested with L1, L2, and L1L2 regularisation, however, only $\lambda = 0.001$ L2 regularisation performed significantly better than a random guess. L2 regularisation was chosen, however, to improve performance other λ-values were tested and $\lambda = 0.0002$ was chosen.

The final model is presented in Table B.14:

Table B.14: Summary of the final small-scale model (`n40 Model`) found by calling `model.Summary()`.

Layer Type	Output Shape	Params #
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 40)	6400
MaxPooling 2D	(None, 3, 3, 40)	0
Conv2D	(None, 2, 2, 40)	6400
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

† Total parameters 19800 and trainable parameters 19800.

B.8 Discussion

Designing and optimising a neural network can take a lot of time, due to the sheer amount of hyperparameters. In this appendix, a lot of time has been saved by heuristically going forward with the best performing model; testing one hyperparameter per test. This, however, does not ensure the best model and small changes that would improve the performance of the model may have been missed. The purpose of the model is for comparison with another identical implementation, and for that purpose, an optimal model is unnecessary and the shortcomings of the heuristic approach are deemed unimportant.

Furthermore, some of the hyperparameters have settings that can be adjusted, however, this has been purposefully avoided with the same reasoning above.

Repetition of the tests has been purposefully avoided to save time, however, to support the validity of the choices and ensure the correct conclusion would have been drawn, this would be essential. Since the end goal is a model neural network, that will be compared with an almost identical copy, this is not seen as important.

B.9 Methods

This section will briefly describe the methods utilised by the final model.

LANCZOS3 is an *interpolation* function. The reader is recommended to read *section 48.1 Lanczos Resampling* from *Lanczos Resampling for the Digital Processing of Remotely Sensed Images* by Madhukar, B. N. and Narendra, R. [127], as this is the main source for the following brief explanation:

Each samples of the original signal (in this case pixels) are effectively mapped to a translated and scaled Lanczos kernel. The Lanczos kernel is defined as:

$$\mathrm{L}(x) = \begin{cases} 1, & x = 0 \\ \frac{a \sin(\pi x) \sin\left(\frac{\pi x}{a}\right)}{\pi^2 x^2}, & 0 < |x| < a \\ 0, & \text{otherwise} \end{cases} \tag{B.1}$$

where:

x	Arbitrary real argument, where the resampling is performed
a	The size of the kernel in integer-values (for the TensorFlow method `LANCZOS3` $a = 3$)

Since this method is performed on a 2D signal (grayscale CIFAR-100 images), the Lanczos kernel must be 2D, which is defined by the product of two 1D kernels:

$$\mathrm{L}(x,\ y) = \mathrm{L}(x) \cdot \mathrm{L}(y) \tag{B.2}$$

The reconstruction/regeneration of a 2D signal s_{ij} can be performed given:

$$\mathrm{S}(x,\ y) = \sum_{i=\lfloor x \rfloor - a + 1}^{\lfloor x \rfloor + a} \sum_{j=\lfloor x \rfloor - a + 1}^{\lfloor y \rfloor + a} s_{ij} \mathrm{L}(x - i) \cdot \mathrm{L}(y - j) \tag{B.3}$$

where:

$x,\ y$	Arbitrary real argument (coordinate), where the resampling is performed
a	The size of the kernel in integer-values (for the TensorFlow method `LANCZOS3` $a = 3$)
s_{ij}	Original signal indexed at $(i,\ j)$

adamax is an *optimisation algorithm*. The reader is recommended to read the article *Adam: A Method for Stochastic Optimization* by Kingma, Diederik P and Ba, Jimmy [128], as this is the main source for the following brief explanation.

Adamax is a modified version of *Adam*; a gradient descent method with individual adaptive learning rates for the parameters based on the first and second moments of the gradients. *Adam* is designed to work well with *sparse gradients* and work well in on-line and non-stationary settings, by combining the advantages of *AdaGrad* and *RMSProp*. *Adamax* differs from *Adam* when scaling the gradients: Using *Adam* the gradients are inversely proportional to a L^2 norm of their individual current and past gradients, whereas the norm used in *Adamax* is L^∞.

BinaryFocalCrossentropy is a *loss function*. The reader is recommended to read the article by Daniel Godoy [129] and to explore the Keras 3 API documentation about probabilistic losses, as these are the main sources for the following brief explanation.

Entropy is a measure for the uncertainty of a given distribution: In the case of image classification *high entropy* would mean that the distribution of different classes is uniform, whereas *low entropy* would be if the distribution of the classes is heavily skewed toward a class. The *entropy* may not be known, however, given a set of labels it can be approximated as the *crossentropy* [129]:

$$\begin{aligned} \mathrm{H}(q) &= -\sum_{c=1}^{C} q(y_c) \cdot \log(p(y_c)) \\ \mathrm{H_p}(q) &= -\sum_{c=1}^{C} q(y_c) \cdot \log(q(y_c)) \end{aligned} \tag{B.4}$$

where:

$H(q)$	The *entropy*
$\mathrm{H_p}(q)$	The *cross-entropy*
C	Number of classes
$q(y_c)$	Likelihood of class c given the **true distribution**
$p(y_c)$	Likelihood of class c given the **approximated distribution**

If the approximated distribution matches the true distribution perfectly, $\mathrm{H}(q)$ and $\mathrm{H_p}(q)$ would be equal, otherwise *cross-entropy* will have a bigger value than *entropy*. This inequality is called *KL divergence*, however, for brevity the explanation is omitted, but can be found in the article. The goal is to minimize this divergence, i.e. this can be seen as a loss function. Given that the training/test sets have a uniform distribution of classes and ''a little bit of manipulation'', any point from any class can be utilised in the same formula, *binary cross-entropy*:

$$\mathrm{H}_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1-y_i) \cdot \log(1-p(y_i)) \tag{B.5}$$

Binary cross-entropy becomes *binary focal cross-entropy* when another term is introduced: the *focal factor*. The focal factor scales the difference between the true label and predicted label, which should help to ''down-weights the ss assigned to well-classified examples [...] focuses training on a sparse set of hard examples and prevents the vast number of easy negatives from overwhelming the detector during training'' [130].

L2 Regularisation is a *penalty term for the loss function*. The reader is recommended to read the article *L1 and L2 Regularization Methods, Explained* by *Anuja Nagpal* [131] or watch Regularization Part 1: Ridge (L2) Regression, as they are the main sources for the following brief explanation.

The *L2 penalty term* is the addition of the squared magnitudes of the weights/coefficients on the loss function, scaled by a set constant λ:

$$\lambda \sum_{i=1}^{n} \beta_j^2 \tag{B.6}$$

where:

λ	Scalar constant
β_i	ith weight
n	Number of weights

Adding this extra loss term helps reduce variance by shrinking the weights and by extension making the predictions less sensitive to them. Reducing the variance may in turn help the model avoid overfitting.

Selection of Approximate Circuits for Comparing Metrics

C

This appendix presents a selection of approximate adders and multipliers from the EvoApproxLib [4]. The purpose is to attain different approximate arithmetic circuits with different characteristics, to be able to evaluate the benchmark system for varying use cases. The selection is made to achieve a catalogue with varying approximate parameters.

The approximate circuits alone constitute step one of the benchmarking system, and the procedure for evaluation is described in chapter 4. The purpose of step one is to determine the **power**, **latency** and **inaccuracy** of a functional approximate circuit on the RTL.

It is chosen to measure *power* as **gate-count** and *latency* as **critical path** as respectively power/area and latency are functions hereof. The designed analysis tool described in section 4.2 is implemented such that a user can change assumptions about the number of transistors in a certain gate, and the latency of the logic gate circuits. Furthermore, the user can modify which gates are available in the synthesis. The gates that are available to choose from and their assumed `CMOS` layout is congruent with those presented in [113]. The transistor count is presented in Table C.1.

To get an estimate of the critical path delay of each available logic gate, an estimate is derived based on research in [114], that finds the propagation delay of a *conventional* inverter and `NAND`-gates to be about 150 ps. The estimate used for the remainder of the logic gates is 300 ps for `AND` and `OR` gates and 450 ps for `XOR` and `XNOR`. It is **emphasised** that these delays are assumed to be provided by the user of the benchmarking system and that these estimates are only placeholder values used for exemplification.

Table C.1: Assumptions on transistor count and latency for a selection of logic gates. These estimates are used as examples through the entirety of the step one analysis examples in this project.

	AND	XOR	NAND	OR	XNOR	NOT	Reference
Transistor Count	6	10	4	6	10	2	[113]
Propagation Delay	300 ps	450 ps	150 ps	300 ps	450 ps	150 ps	Inspired by [114]

The inaccuracy will be defined by *errors metrics* induced by adjusting the **power consumption** and/or **latency** with relation to an exact counterpart. An important error metric is the *difference* defined as [23]:

$$\text{diff}(f(x),\ \tilde{f}(x)) \triangleq ||f(x) - \tilde{f}(x)|| \tag{C.1}$$

where:

x	input to functions f and $\tilde{f}$
$f(x)$	exact function performed on input x
$\tilde{f}(x)$	approximate function performed on input x

There are several error metrics, which utilise the difference, some of them are defined below [23]:

WCD (Worst Case Distance):

$$\max_{x \in X} \left(\text{diff}(f(x),\ \tilde{f}(x))\right) \tag{C.2}$$

MAE (Mean Absolute Error):

$$\mathbb{E}\left\{\text{diff}(f(x),\ \tilde{f}(x))\right\} \tag{C.3}$$

MSE (Mean Squared Error):

$$\frac{1}{|X|} \sum_{x \in X} \left(\text{diff}(f(x),\ \tilde{f}(x))\right)^2 \tag{C.4}$$

Eq. (C.2), (C.3), and (C.4) are good tools to indicate the precision of the approximation performed, however, they are tied to the numerical values of the evaluated input. In some contexts it would be more appropriate to examine how often errors happen, the ER is defined as [22]:

$$\frac{|W|}{|X|}, \qquad W = \{x \in X | f(x) \neq \tilde{f}(x)\} \tag{C.5}$$

In digital circuit design, it is also relevant to view how much the evaluated input differs from approximate to exact on the bit level. For this purpose, the HD can be utilised; the HD between two binary values, x and y, is defined as the count of bits that differ:

$$\sum_{i=0}^{n} \mathbb{1}[x_i \neq y_i] \tag{C.6}$$

In circuits with multiple input-output relations, it can be beneficial to evaluate the MHD which is simply:

$$\frac{1}{|X|} \sum_{x \in X} \sum_{i=0}^{n} \mathbb{1}[x_i \neq y_i] \tag{C.7}$$

C.1 Step One Analysis on Approximate Circuits

To obtain a selection of approximate arithmetic circuits to use as examples in the remainder of this project, multiple adders and multipliers from the EvoApproxLib are analysed using the methods described in chapter 4. The chosen circuits are summed up in Table C.2.

Table C.2: The chosen approximate circuits from EvoApproxLib [4]. All circuits perform signed 8-bit arithmetic operations.

mul8s_1L12	mul8s_1KV9	mul8s_1KV8	mul8s_1KVM	mul8s_1KVA	mul8s_1L2J	mul8s_1KV6	add8se_839	add8se_8VQ	add8se_8NH	add8se_8CL
Apx.	Apx.	Apx.	Apx.	Apx.	Apx.	Acc.	Apx.	Apx.	Apx.	Acc.
Multipl	Multipl	Multipl	Multipl	Multipl	Multipl	Multipl	Adder	Adder	Adder	Adder

Each circuit is provided with a `Verilog` and C file in the EvoApproxLib and can be evaluated provide these to the Makefile function (see `/rtl-analysis/Makefile` in Appendix A). The result provides metrics regarding cost, performance and inaccuracy for the input circuit and provides a synthesised RTL netlist, a graph illustrating the critical path in a directed graph, and a distribution of all possible errors for the approximate circuit.

C.1.1 mul8s_1L12

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.1.

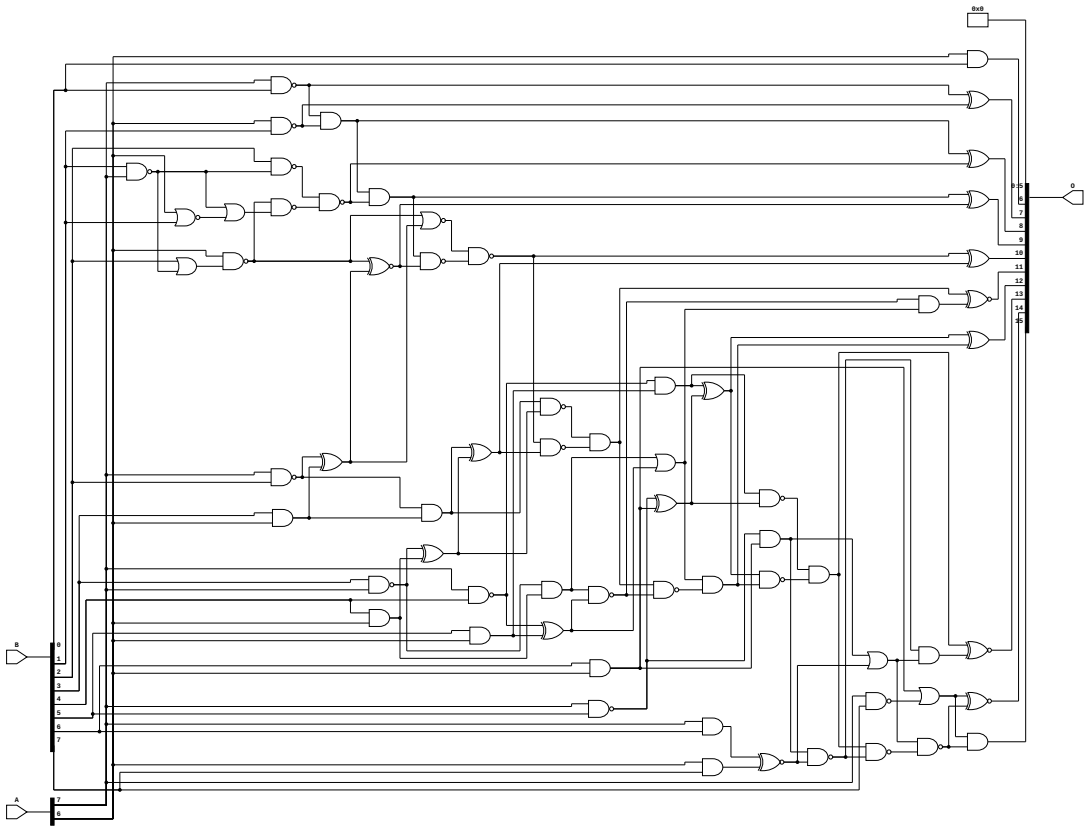

Figure C.1: `mul8s_1L12` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.3.

Table C.3: Gatecounts of the netlist visualised in Figure C.1 (i.e. `mul8s_1L12`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
20	11	24	5	5	0	3	68	412

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.2 and the corresponding critical path gate count and delay are shown in Table C.4.

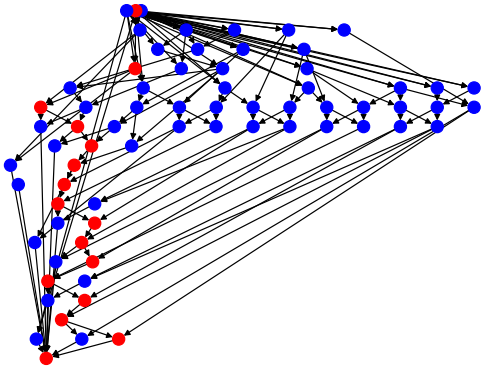

Figure C.2: Visualisation of the DAG representation of Figure C.1. The red nodes indicate the calculated **critical path**.

Table C.4: Critical path data of Figure C.2 (i.e. the directed graph of the `mul8s_1KV6` circuit).

Number of Gates	Propagation delay† [ns]
18	19.95

† Calculated based on assumptions presented in Table C.1.

The functional circuit is an 8-bit signed multiplier meaning that both the multiplicand and multiplier can take any integer value between -128 and 127, i.e. $x \in \{-128, -127, -126, ..., 127\}$. It is assumed that the multiplier and multiplicand take any value with equal probability, i.e. $A, B \sim U\{-128, 127\}$. This means that the $P(A = x) = P(B = x) = \frac{1}{256}$. The joint distribution can be represented as a LUT where each entry is the product of the marginal entries, i.e. $f_{A,B} = P(A = x, B = x) = P(A = x) \cdot P(B = x) = \frac{1}{512}$. It is then possible to describe the error distance as another discrete RV (E) with a conditional PMF $f_{E|A,B}$. This PMF is obtained by evaluating every output of the approximate circuits exactly once.

The PMF is plotted in a histogram using 100 bins for this particular circuit. The histogram is seen in Figure C.3.

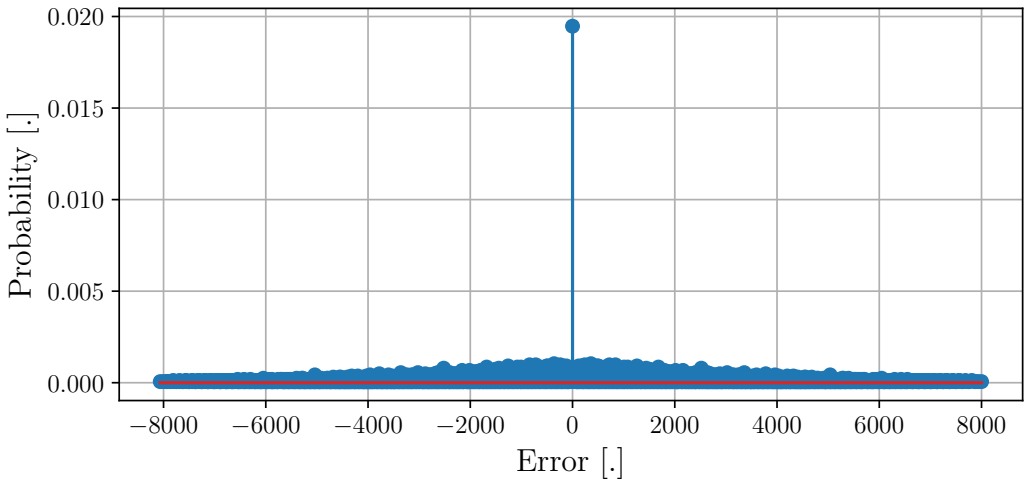

Figure C.3: Histogram of the error distribution of the approximate circuit in Figure C.1. The distribution is illustrated as a histogram of 100 distinct bins and normalised to show probability density.

The single-value metrics for this distribution are shown in Table C.5.

Table C.5: Error-metrics of the distribution presented in Figure C.3.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
2016	$7.283 \cdot 10^6$	98.053 %	8064	5.06

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

The purpose of providing these metrics is for the user to pick and choose between metrics for a certain application

C.1.2 mul8s_1KV9

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.4.

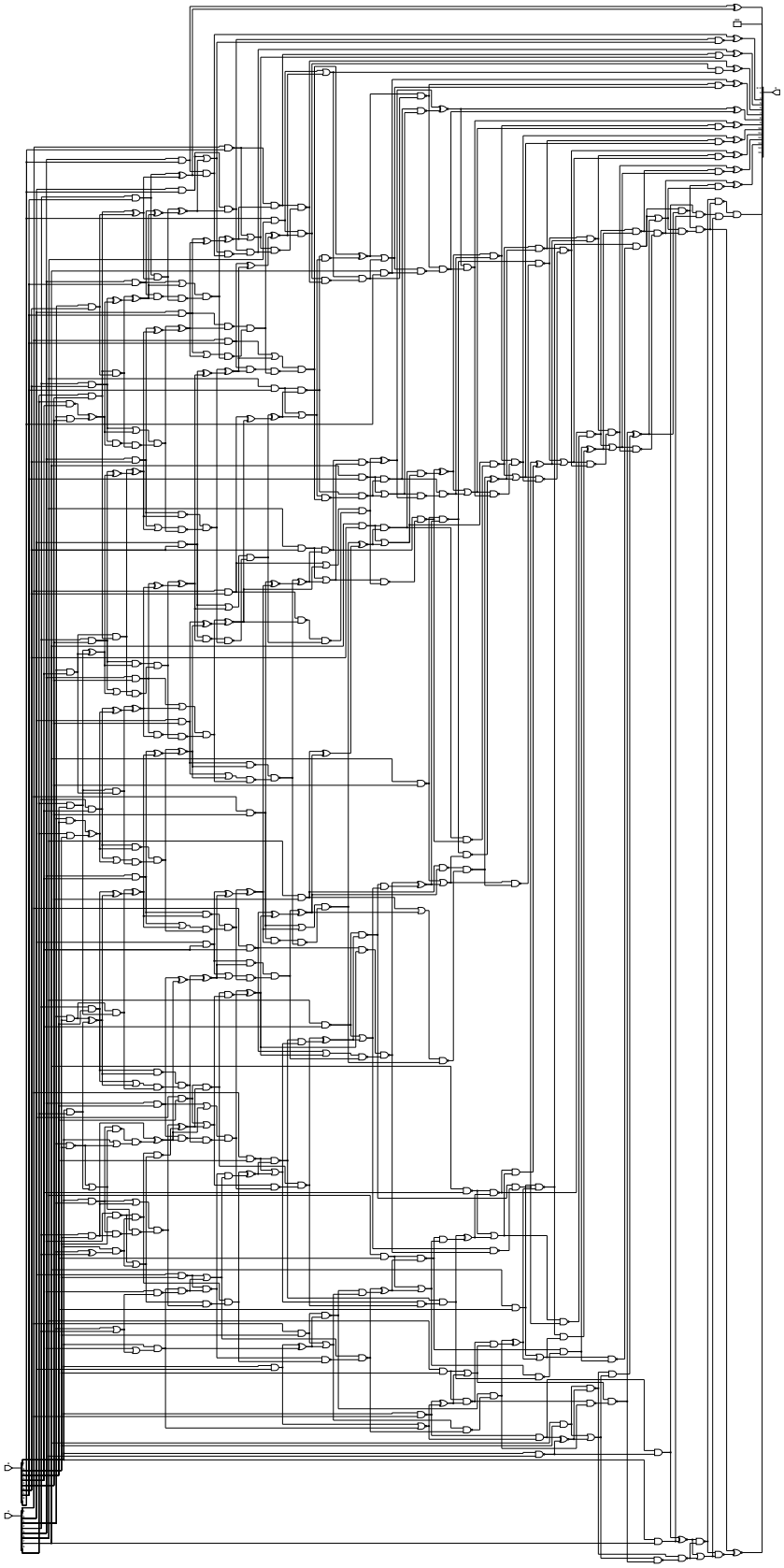

Figure C.4: `mul8s_1KV9` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.6.

Table C.6: Gatecounts of the netlist visualised in Figure C.4 (i.e. `mul8s_1KV9`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
76	19	159	49	57	1	0	361	2150

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.5 and the corresponding critical path gate count and delay are shown in Table C.7.

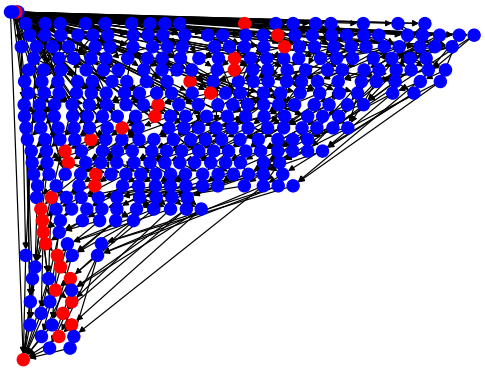

Figure C.5: Visualisation of the DAG representation of Figure C.4. The red nodes indicate the calculated **critical path**.

Table C.7: Critical path data of Figure C.5 (i.e. the directed graph of the `mul8s_1KV9` circuit).

Number of Gates	Propagation delay† [ns]
18	3.9

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.6.

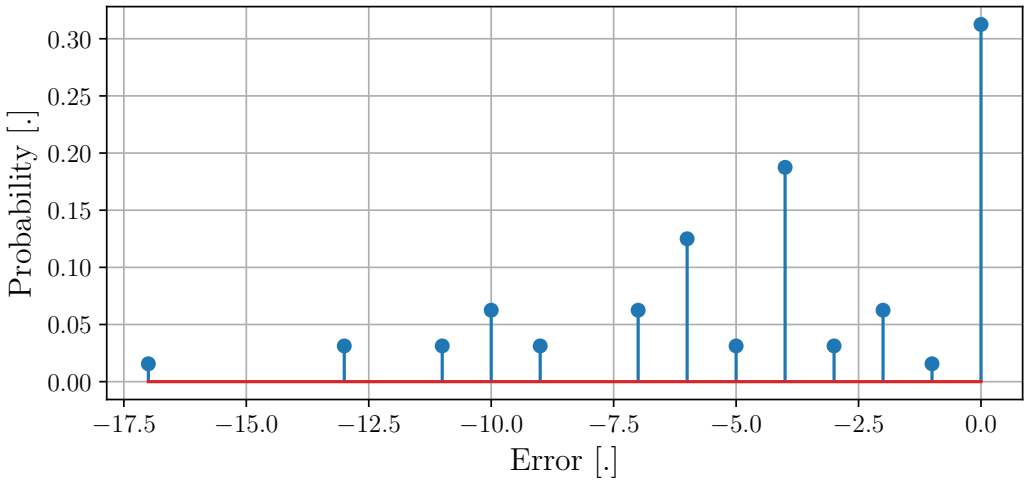

Figure C.6: PMF of the error distribution of the approximate circuit in Figure C.4. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.8.

Table C.8: Error-metrics of the distribution presented in Figure C.6.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
4.25	34.25	68.75 %	17	1.4

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.3 mul8s_1KV8

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.7.

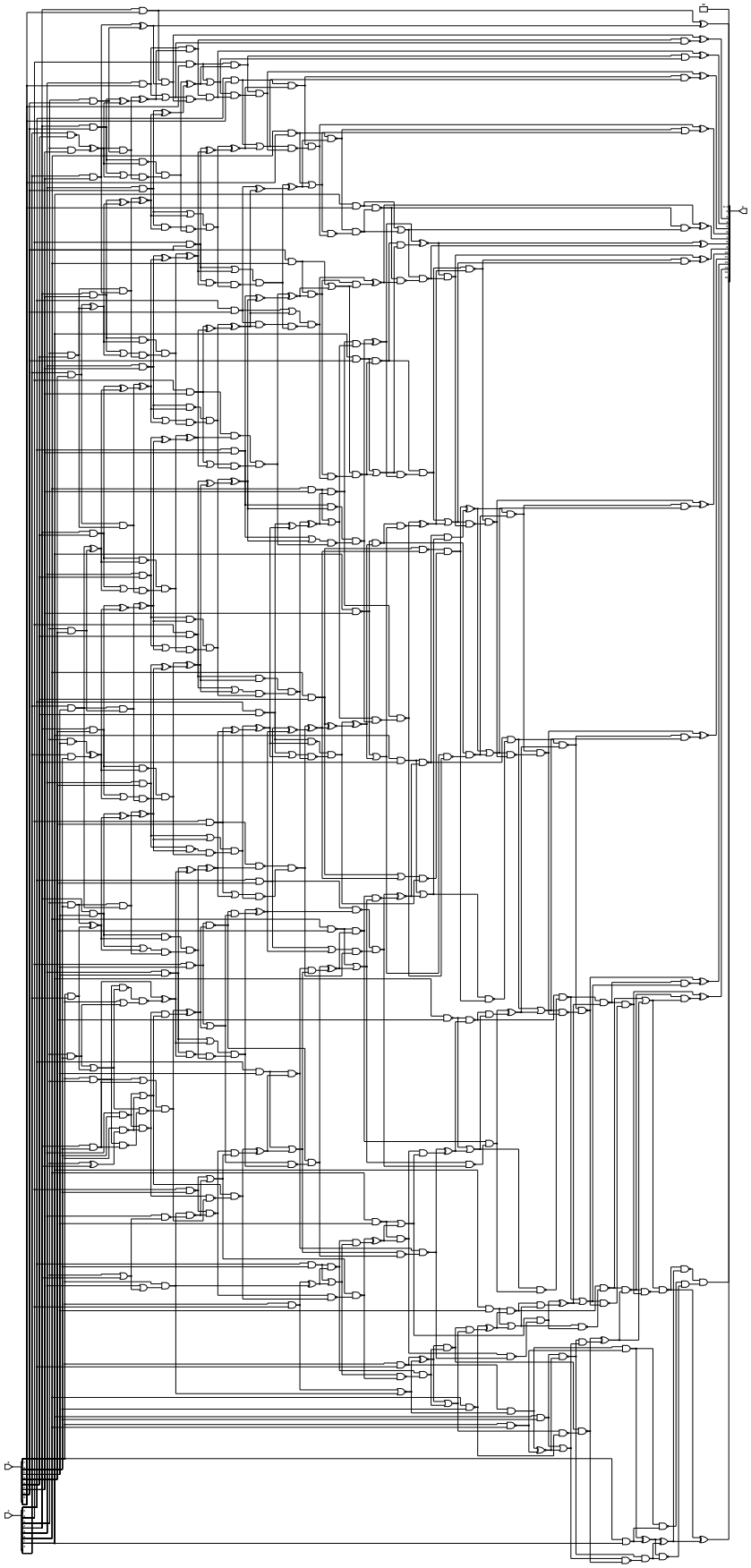

Figure C.7: `mul8s_1KV8` synthesised to `AND, XOR, NAND, OR, NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.9.

Table C.9: Gatecounts of the netlist visualised in Figure C.7 (i.e. `mul8s_1KV8`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
91	19	153	50	61	1	0	375	2262

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.8 and the corresponding critical path gate count and delay are shown in Table C.10.

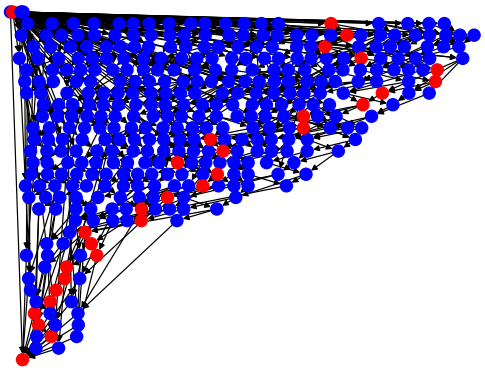

Figure C.8: Visualisation of the DAG representation of Figure C.7. The red nodes indicate the calculated **critical path**.

Table C.10: Critical path data of Figure C.8 (i.e. the directed graph of the `mul8s_1KV8` circuit).

Number of Gates	Propagation delay† [ns]
28	8.1

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.9.

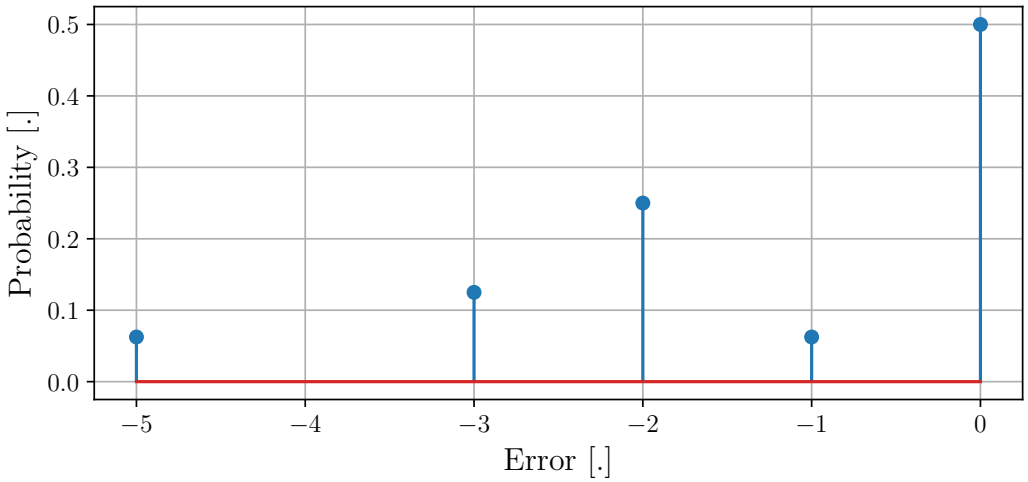

Figure C.9: PMF of the error distribution of the approximate circuit in Figure C.7. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.11.

Table C.11: Error-metrics of the distribution presented in Figure C.9.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
1.25	3.75	50 %	5	0.75

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.4 mul8s_1KVM

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.10.

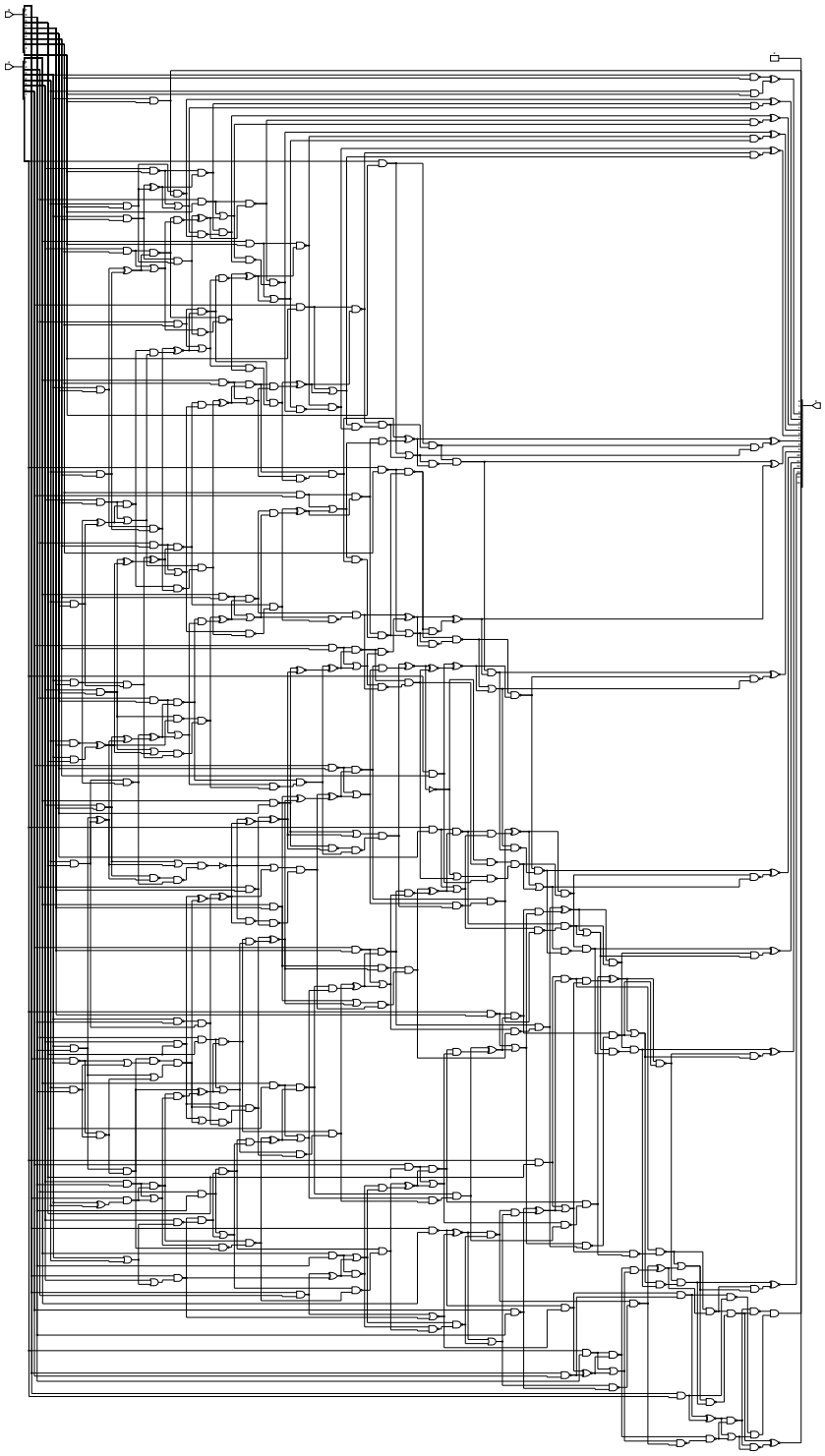

Figure C.10: `mul8s_1KVM` synthesised to `AND, XOR, NAND, OR, NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.12.

Table C.12: Gatecounts of the netlist visualised in Figure C.10 (i.e. `mul8s_1KVM`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
92	11	139	45	49	1	2	339	1986

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.11 and the corresponding critical path gate count and delay are shown in Table C.13.

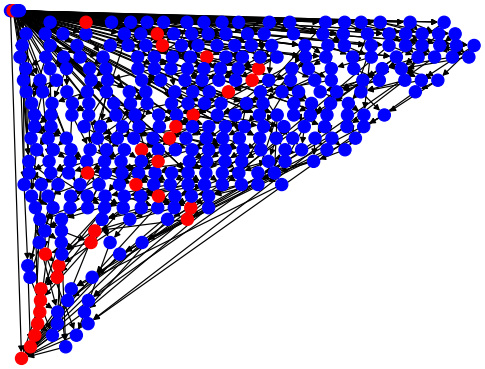

Figure C.11: Visualisation of the DAG representation of Figure C.10. The red nodes indicate the calculated **critical path**.

Table C.13: Critical path data of Figure C.11 (i.e. the directed graph of the `mul8s_1KVM` circuit).

Number of Gates	Propagation delay† [ns]
28	8.25

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.12.

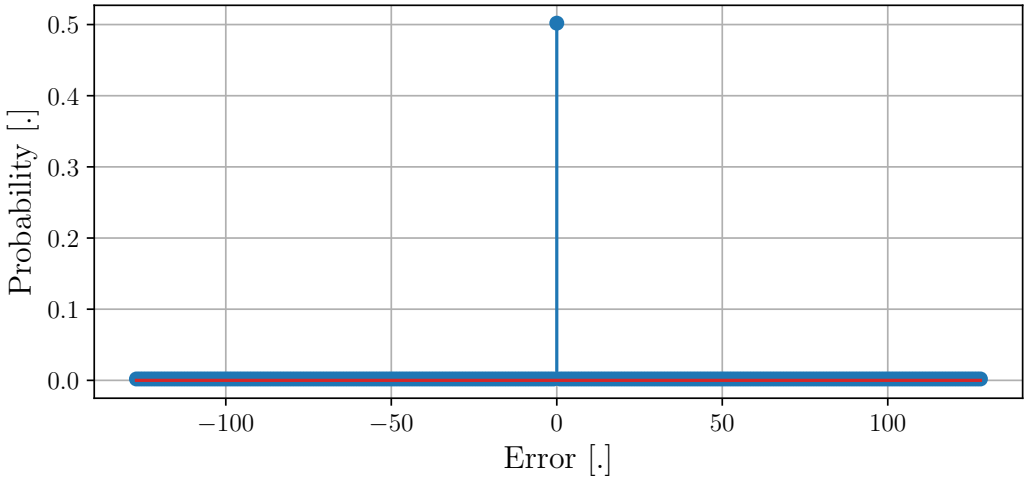

Figure C.12: PMF of the error distribution of the approximate circuit in Figure C.10. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.14.

Table C.14: Error-metrics of the distribution presented in Figure C.12.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
32	2730.75	49.8 %	128	2.23

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.5 mul8s_1KVA

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.13.

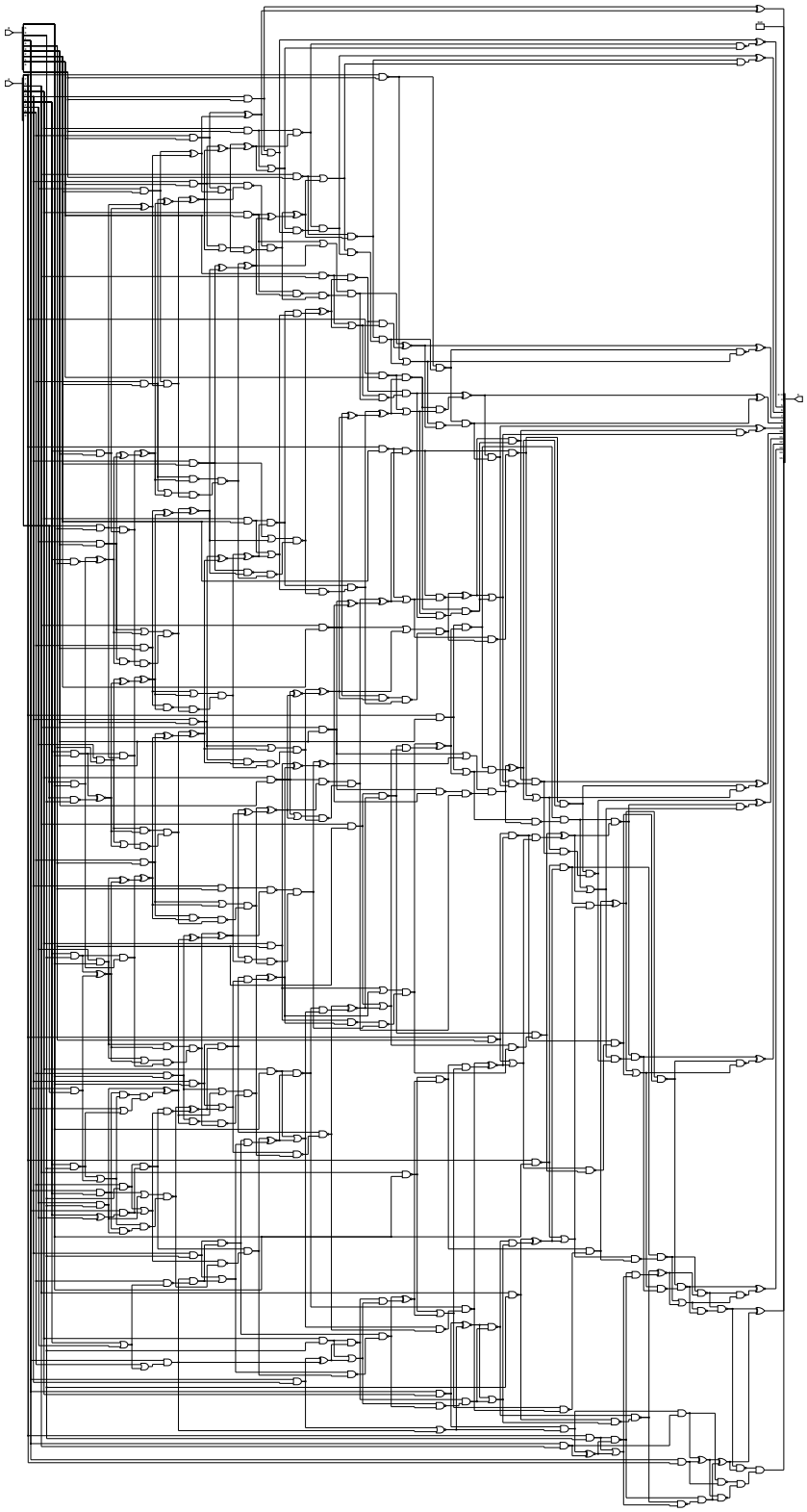

Figure C.13: `mul8s_1KVA` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.15.

Table C.15: Gatecounts of the netlist visualised in Figure C.13 (i.e. `mul8s_1KVA`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
73	16	147	45	56	1	0	338	2020

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.14 and the corresponding critical path gate count and delay are shown in Table C.16.

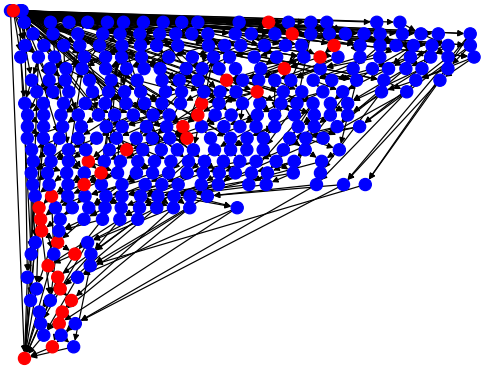

Figure C.14: Visualisation of the DAG representation of Figure C.13. The red nodes indicate the calculated **critical path**.

Table C.16: Critical path data of Figure C.14 (i.e. the directed graph of the `mul8s_1KVA` circuit).

Number of Gates	Propagation delay† [ns]
28	8.1

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.15.

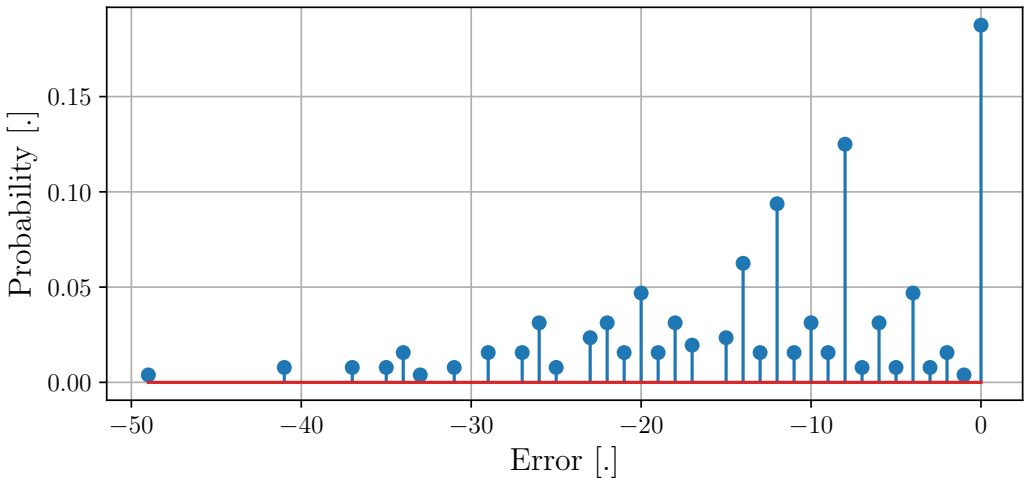

Figure C.15: PMF of the error distribution of the approximate circuit in Figure C.13. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.17.

Table C.17: Error-metrics of the distribution presented in Figure C.15.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
12.25	248.25	81.25 %	49	2.14

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.6 mul8s_1L2J

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.16.

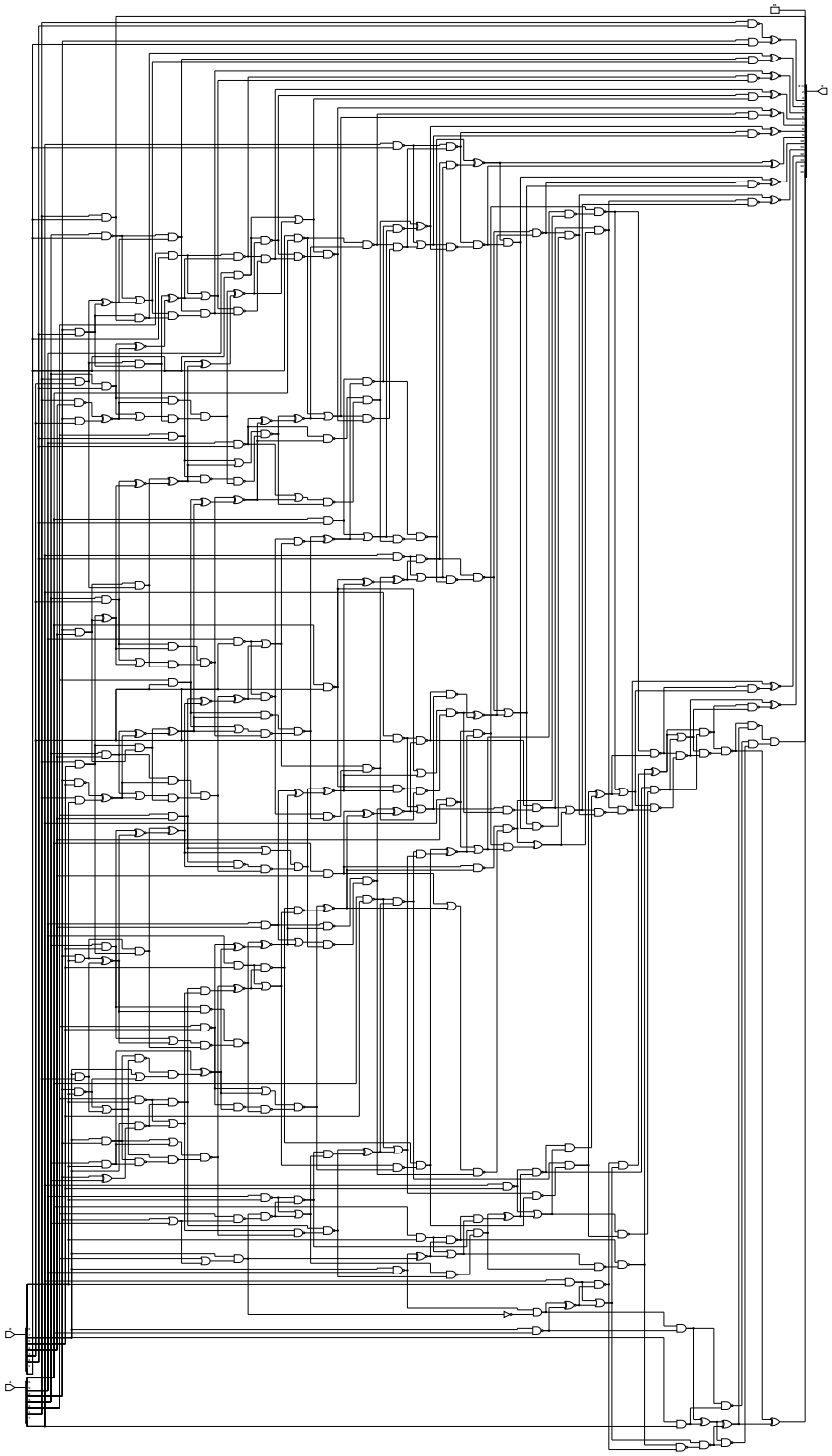

Figure C.16: `mul8s_1L2J` synthesised to `AND, XOR, NAND, OR, NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.18.

Table C.18: Gatecounts of the netlist visualised in Figure C.16 (i.e. `mul8s_1L2J`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
58	14	132	38	45	0	1	288	1696

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.17 and the corresponding critical path gate count and delay are shown in Table C.19.

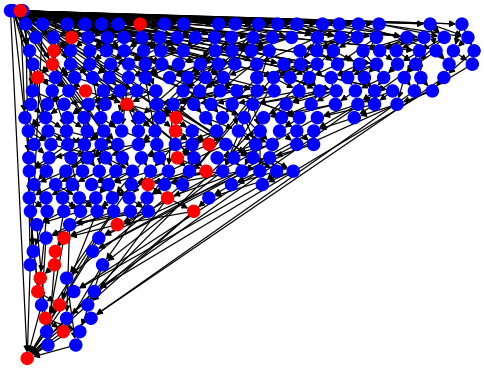

Figure C.17: Visualisation of the DAG representation of Figure C.16. The red nodes indicate the calculated **critical path**.

Table C.19: Critical path data of Figure C.17 (i.e. the directed graph of the `mul8s_1L2J` circuit).

Number of Gates	Propagation delay† [ns]
24	6.9

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in a histogram using 150 bins for this particular circuit. The histogram is seen in Figure C.18.

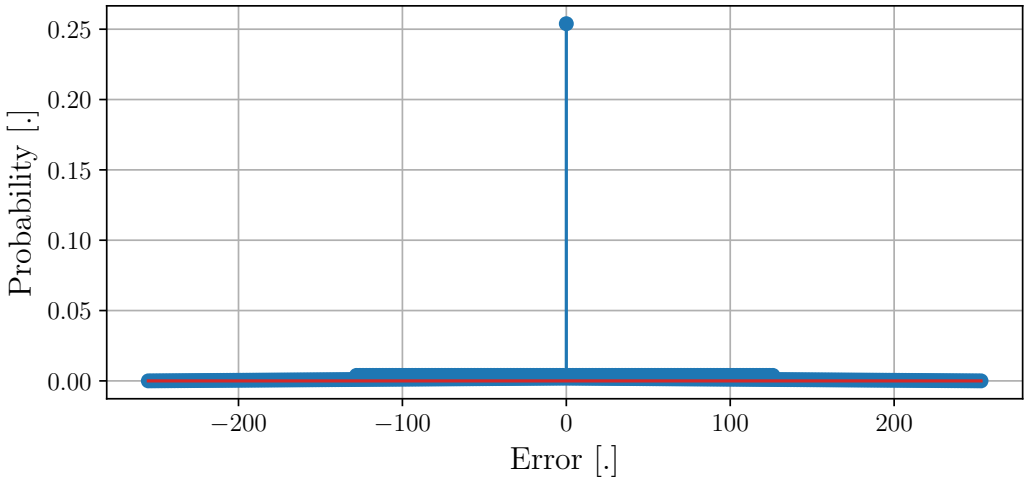

Figure C.18: PMF of the error distribution of the approximate circuit in Figure C.16. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.20.

Table C.20: Error-metrics of the distribution presented in Figure C.18.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
53.3	5462	74.6 %	255	3.25

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.7 mul8s_1KV6

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.19.

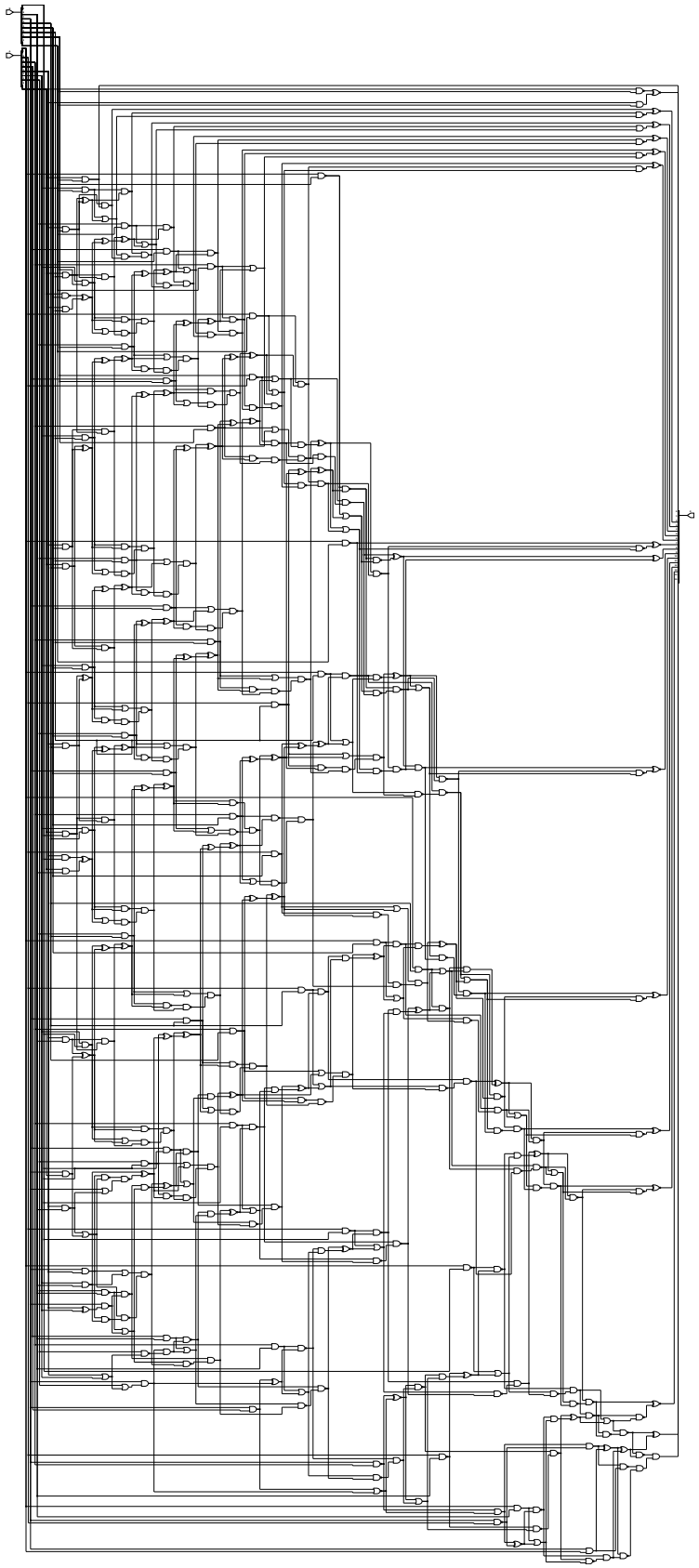

Figure C.19: `mul8s_1KV6` synthesised to `AND, XOR, NAND, OR, NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.21.

Table C.21: Gatecounts of the netlist visualised in Figure C.19 (i.e. `mul8s_1KV6`).

AND	XOR	NAND	OR	XNOR	NOR	gatecount	transistorcount†
79	19	171	51	63	1	384	2288

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.20 and the corresponding critical path gate count and delay are shown in Table C.22.

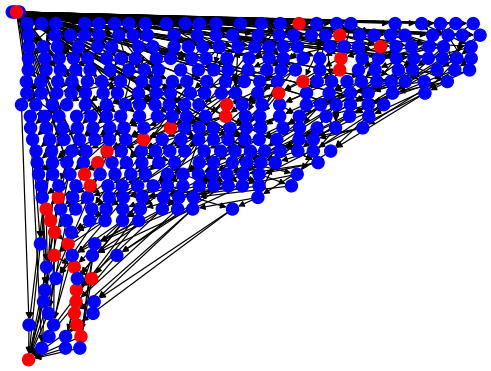

Figure C.20: Visualisation of the DAG representation of Figure C.19. The red nodes indicate the calculated **critical path**.

Table C.22: Critical path data of Figure C.20 (i.e. the directed graph of the `mul8s_1KV6` circuit).

Number of Gates	Propagation delay† [ns]
82	8.1

† Calculated based on assumptions presented in Table C.1.

Since the mul8s_1KV6 is an accurate multiplier, there are no error metrics to report.

C.1.8 Multiplier Comparison

To compare the chosen multipliers is a balance not only between the *power*, *latency* and *inaccuracy*, but also the application-relevant sub-metrics. For some applications, there could be a boundary to the WCD and for different applications, it is desired to trade off a high *gate count* for a short *critical path*. An illustration that aids the comparability of the trade-offs for different approximate circuits is also provided using the `MakeFile` (see `/rtl-analysis/Makefile` in Appendix A), which takes approximate circuit in Verilog and C++ and finds the metrics presented in all previous sections. It is further possible to specify comparison metrics, i.e. one for *power*, *latency* and *inaccuracy* each. These three constitute a vector that can be plotted in three dimensions, where the vector with the lower magnitude is considered more satisfactory compared to the one with a higher magnitude. Firstly an example is shown for the multipliers `mul8s_1KV9`, `mul8s_1KVM`, `mul8s_1L12`, and `mul8s_1KV6`, comparing the gate-count for *power*, critical path gate-count for *latency*, and error rate for *inaccuracy*. This is shown in Figure C.21.

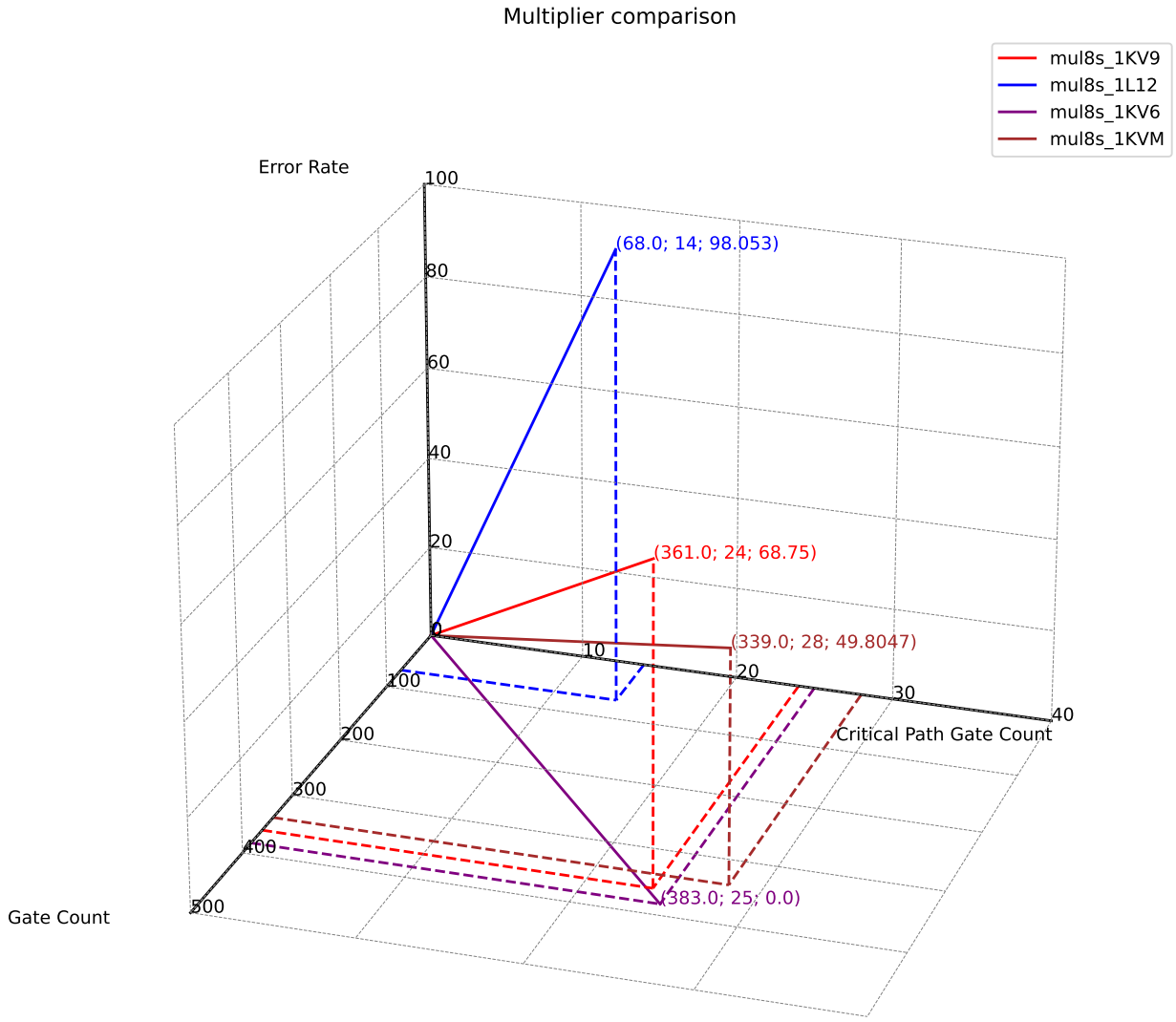

Figure C.21: 3D-plot of the metrics of `mul8s_1KV9`, `mul8s_1KVM`, `mul8s_1L12`, and `mul8s_1KV6`.

A few points are highlighted from Figure C.21. Firstly it is noticed that `mul8s_1L12` has the shortest critical path and lowest gate count, but at the expense of a large error rate. Secondly, the `mul8s_1KV9` has a shorter critical path compared to `mul8s_1KVM`, but trades both total gate count and error rate. Lastly, it is noticed that `mul8s_1KV6` is accurate, as it has 0% error rate and it is noticed that the total gate count is also the highest of the multipliers. However, the critical is even longer for `mul8s_1KVM`. If the user's application values low latency more than a low error rate, the user should choose `mul8s_1KV9` and maybe even `mul8s_1L12` over `mul8s_1KVM` and if the application can only *"afford"* an error rate of 50% only `mul8s_1KVM` and `mul8s_1KV6` is feasible for this selection of multipliers.

Another comparison is made for the same multipliers using gate-count for *power*, critical path gate-count for *latency*, and WCD for *inaccuracy*. This is shown in Figure C.22.

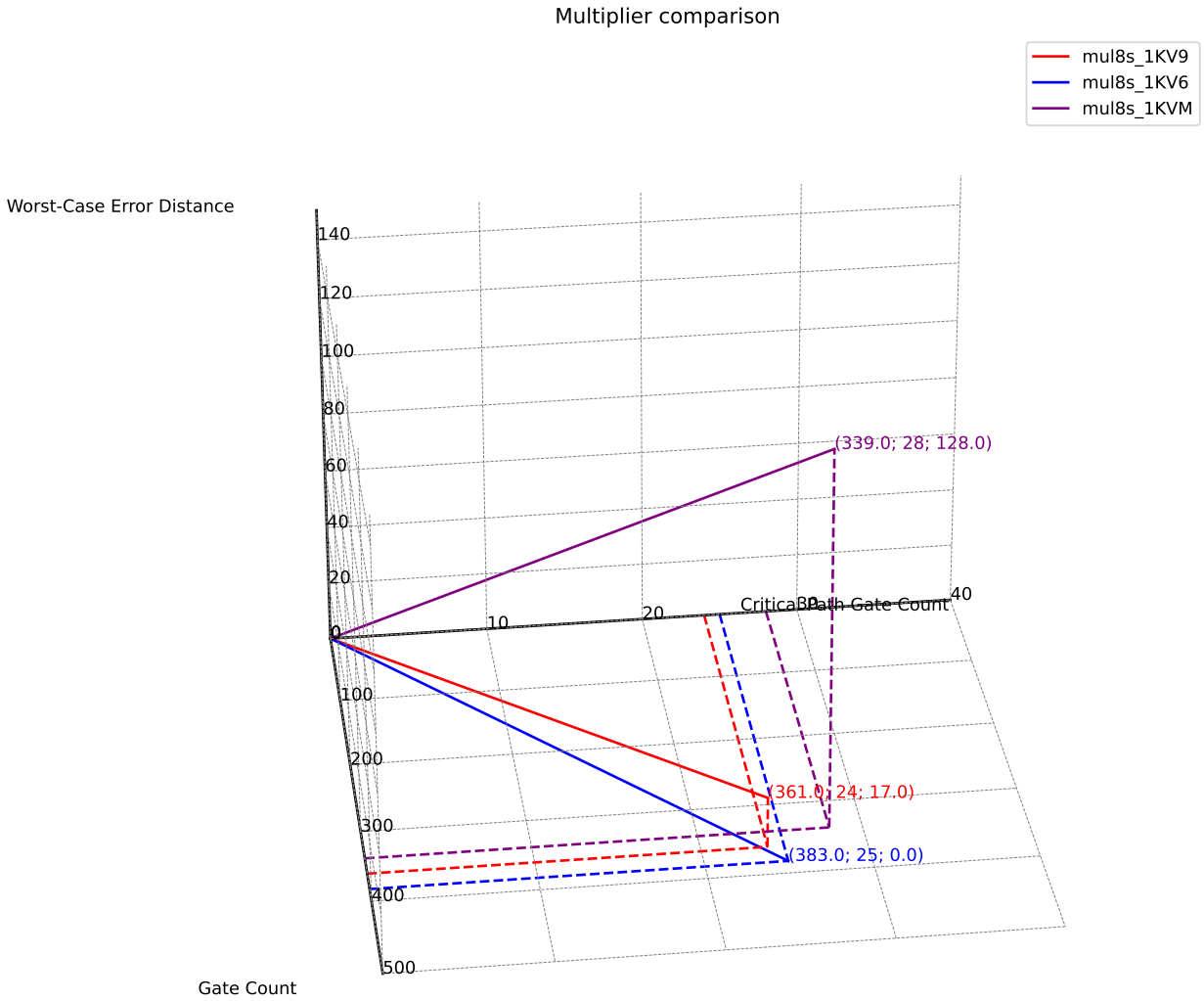

Figure C.22: 3D-plot of the metrics of `mul8s_1KV9`, `mul8s_1KVM`, and `mul8s_1KV6`.

The takeaway from this comparison is that even though `mul8s_1KV9` had a higher error rate compared to `mul8s_1KVM`, it has a significantly lower WCD. This constitutes the fact that comparing the approximate circuits using single parameter metrics is a challenging task and the ability to visualise these metrics benefits the interpretability.

For good measure another comparison is made for the remaining multipliers, i.e. `mul8s_1KV8`, `mul8s_1KVA` and `mul8s_1L2J` using gate-count for *power*, critical path gate-count for *latency*, and error rate for *inaccuracy*, this is shown in Figure C.23.

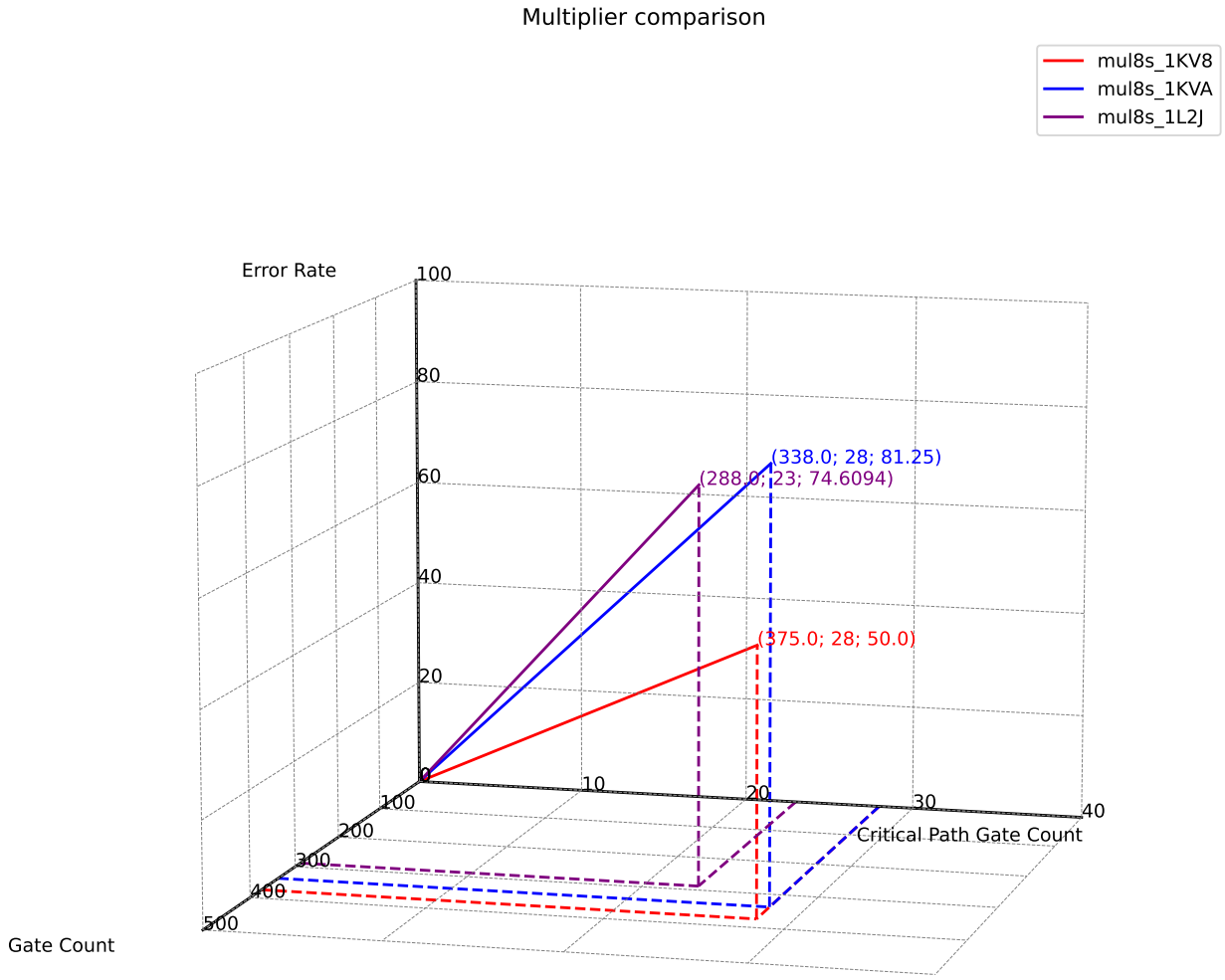

Figure C.23: 3D-plot of the metrics of `mul8s_1KV8`, `mul8s_1KVA` and `mul8s_1L2J`.

This plot shows that `mul8s_1L2J` is superior to `mul8s_1KVA` as its vector is shorter in all dimensions. However, the plots of the same multipliers are shown using WCD as the metric for inaccuracy is shown in Figure C.24. This shows that the trade-off still exists but between the different inaccuracy metrics as `mul8s_1L2J` has a significantly larger WCD compared to `mul8s_1KVA`.

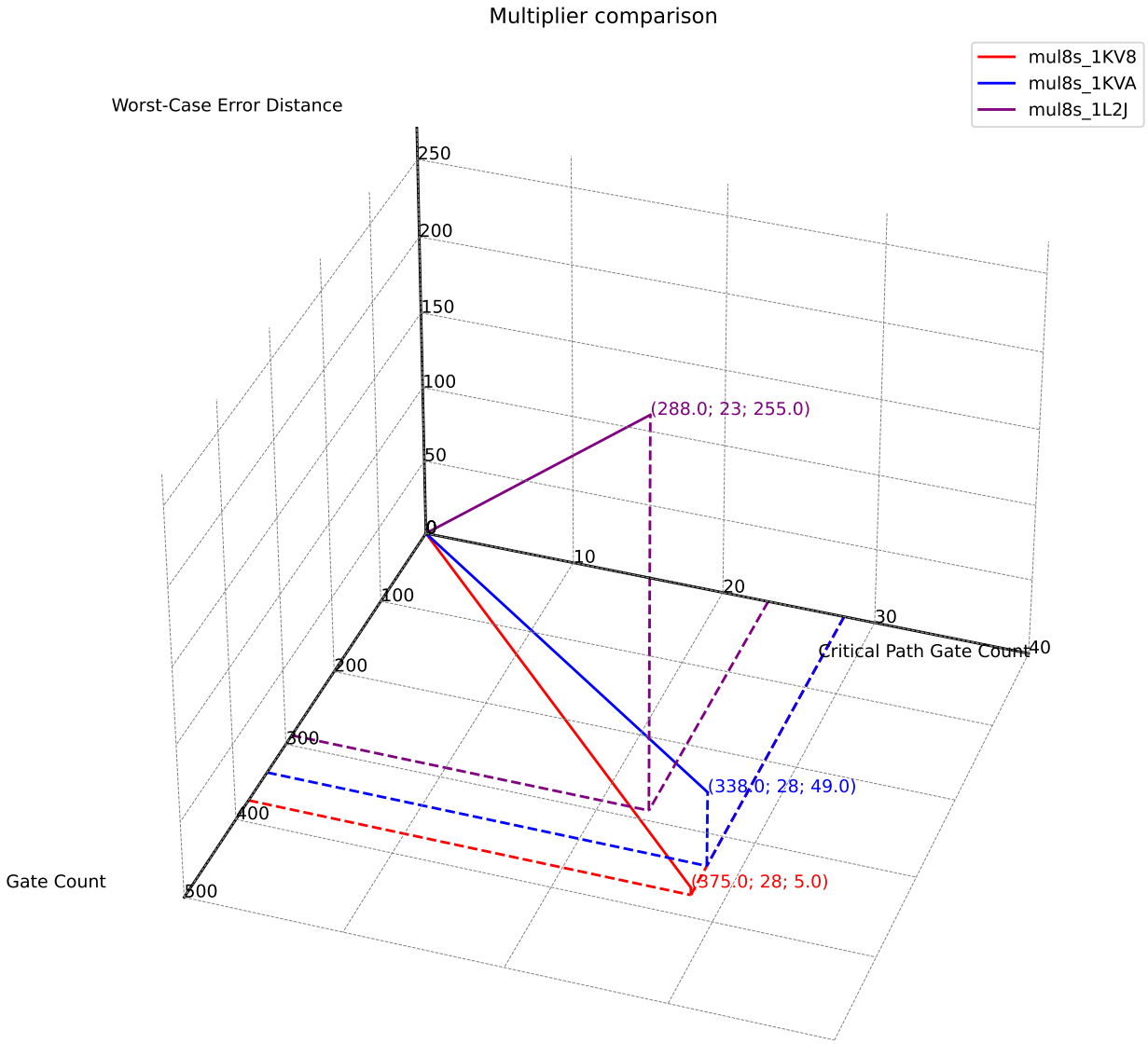

Figure C.24: 3D-plot of the metrics of `mul8s_1KV8`, `mul8s_1KVA` and `mul8s_1L2J`.

C.1.9 add8se_839

In a manner equal to the multiplier circuits the different adders is analysed. Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.25.

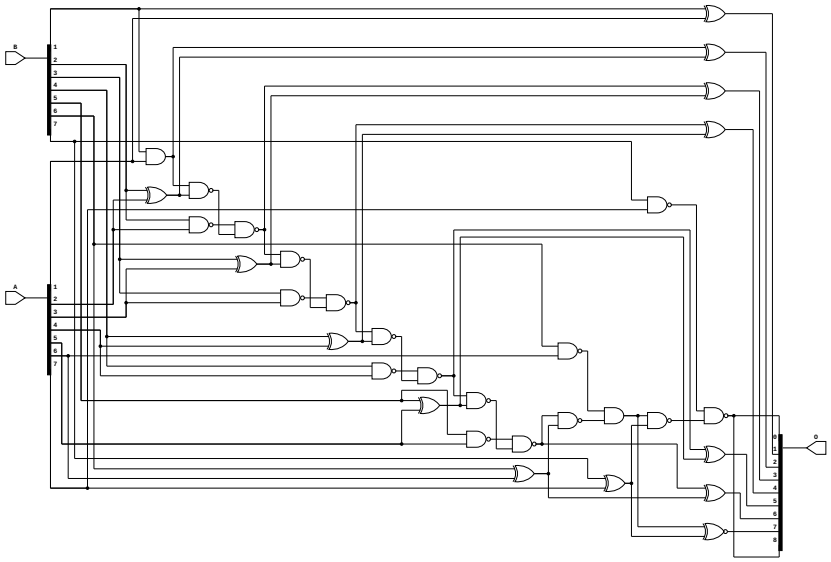

Figure C.25: `add8se_839` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.23.

Table C.23: Gatecounts of the netlist visualised in Figure C.25 (i.e. `add8se_839`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
2	12	17	0	1	0	0	32	210

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.26 and the corresponding critical path gate count and delay are shown in Table C.24.

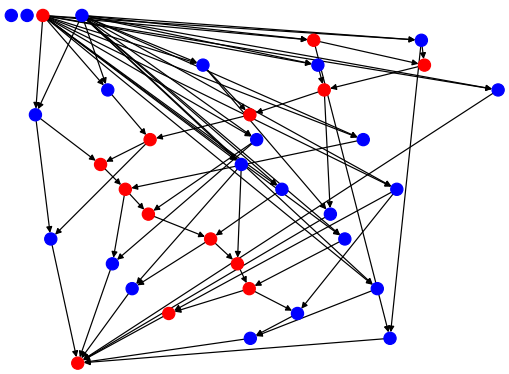

Figure C.26: Visualisation of the DAG representation of Figure C.25. The red nodes indicate the calculated **critical path**.

Table C.24: Critical path data of Figure C.26 (i.e. the directed graph of the `add8se_839` circuit).

Number of Gates	Propagation delay† [ns]
12	2.55

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.27.

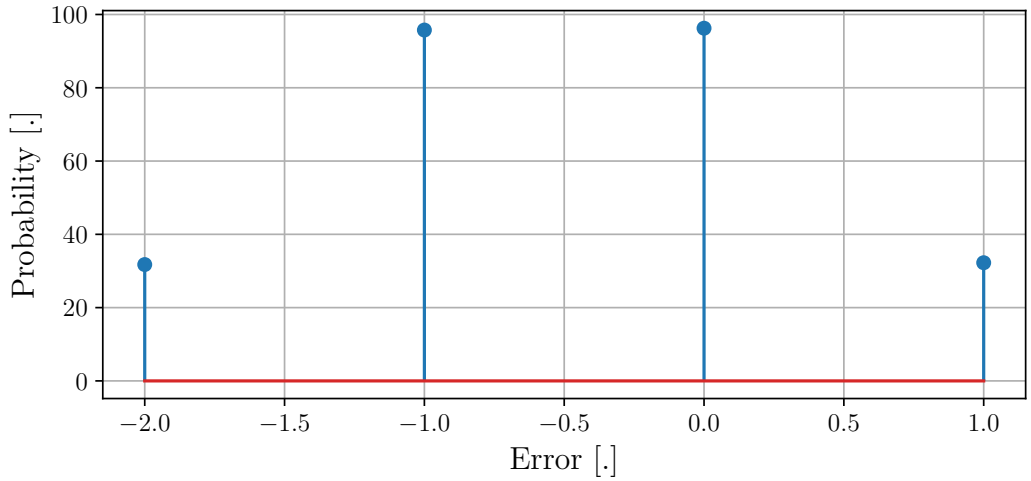

Figure C.27: PMF of the error distribution of the approximate circuit in Figure C.25. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.25.

Table C.25: Error-metrics of the distribution presented in Figure C.27.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
0.75	1	62.4 %	2	0.98

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.10 add8se_8VQ

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.28.

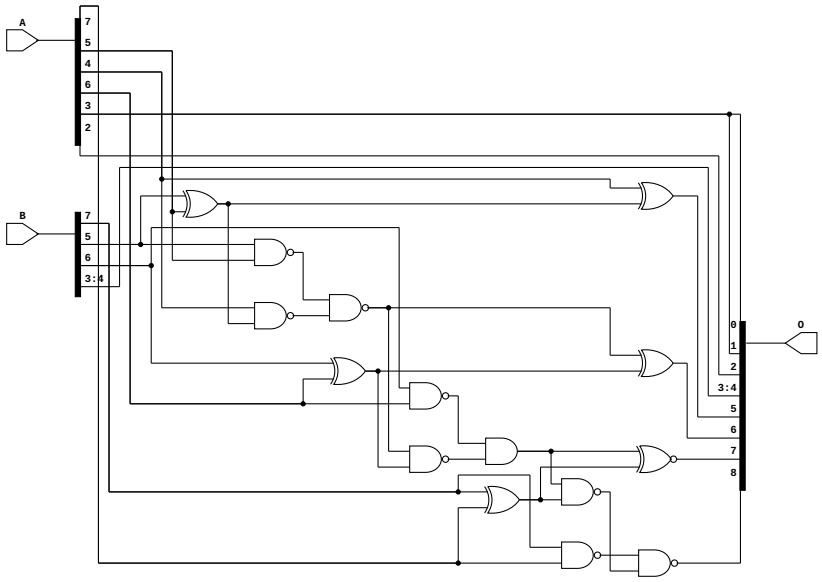

Figure C.28: `add8se_8VQ` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.26.

Table C.26: Gatecounts of the netlist visualised in Figure C.28 (i.e. `add8se_8VQ`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
1	5	8	0	1	0	0	32	210

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.29 and the corresponding critical path gate count and delay are shown in Table C.27.

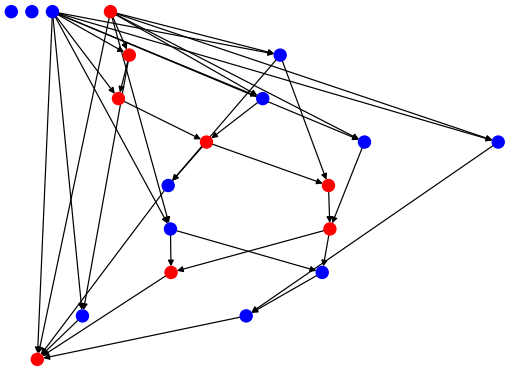

Figure C.29: Visualisation of the DAG representation of Figure C.28. The red nodes indicate the calculated **critical path**.

Table C.27: Critical path data of Figure C.29 (i.e. the directed graph of the `add8se_8VQ` circuit).

Number of Gates	Propagation delay† [ns]
6	1.65

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.30.

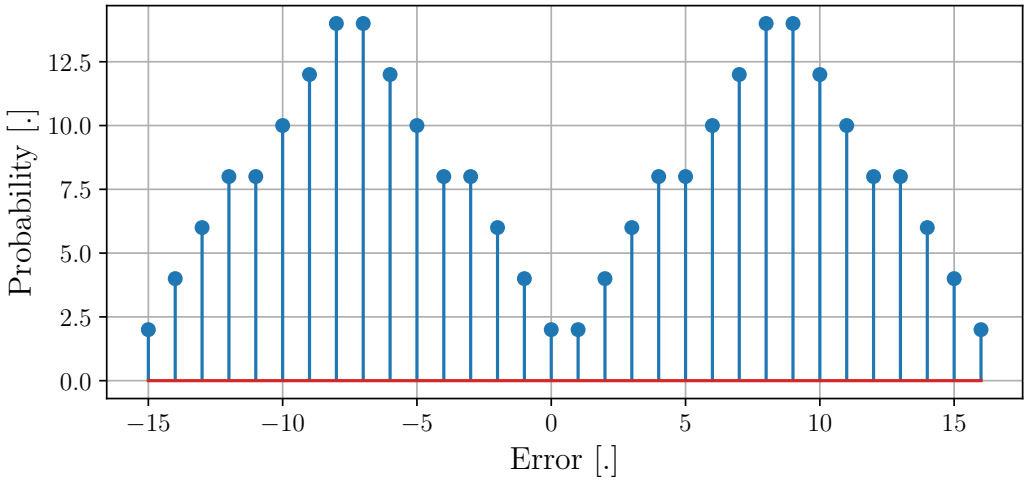

Figure C.30: PMF of the error distribution of the approximate circuit in Figure C.28. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.28.

Table C.28: Error-metrics of the distribution presented in Figure C.30.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
8	77	99.22 %	16	2.75

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.11 add8se_8NH

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.31.

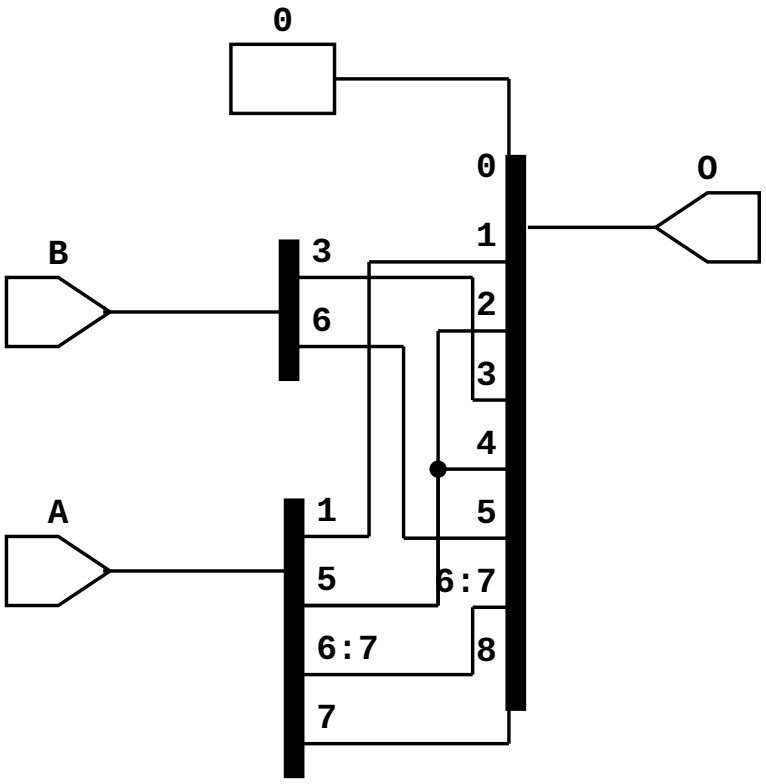

Figure C.31: `add8se_8NH` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.29.

Table C.29: Gatecounts of the netlist visualised in Figure C.31 (i.e. `add8se_8NH`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
0	0	0	0	0	0	0	0	0

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.32 and the corresponding critical path gate count and delay are shown in Table C.30.

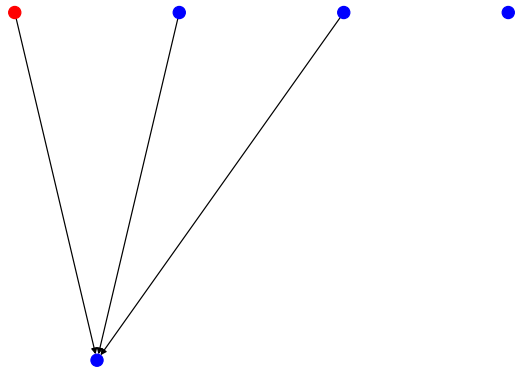

Figure C.32: Visualisation of the DAG representation of Figure C.31. The red nodes indicate the calculated **critical path**.

Table C.30: Critical path data of Figure C.32 (i.e. the directed graph of the `add8se_8NH` circuit).

Number of Gates	Propagation delay† [ns]
0	0

† Calculated based on assumptions presented in Table C.1.

The PMF is plotted in Figure C.33.

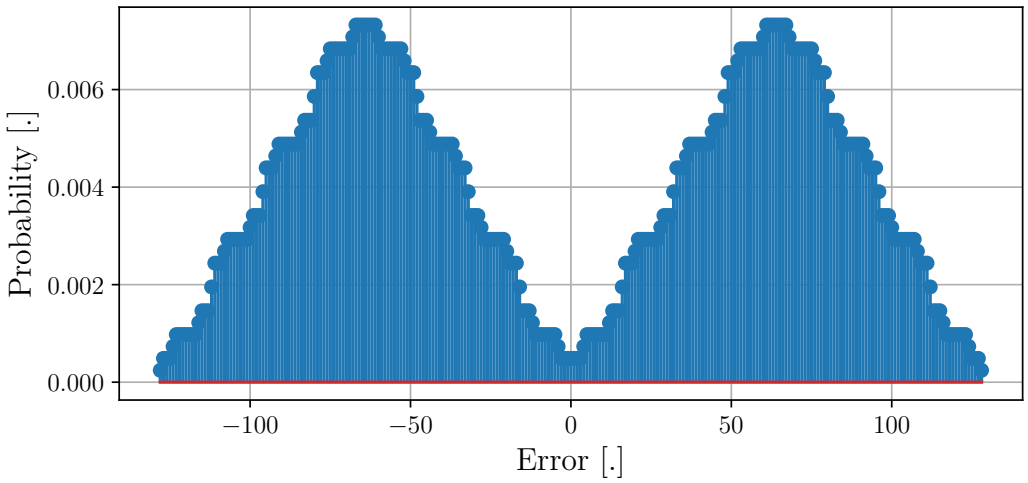

Figure C.33: PMF of the error distribution of the approximate circuit in Figure C.31. The distribution is plotted where each bar represents a discrete error distance.

The single-value metrics for this distribution are shown in Table C.31.

Table C.31: Error-metrics of the distribution presented in Figure C.33.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
64	4797	99.95 %	118	2.875

Note: The formulas for these metrics were presented in chapter 4 and repeated in the beginning of this chapter.

C.1.12 add8se_8CL

Firstly, the netlist is synthesised, providing the circuit diagram shown in Figure C.34.

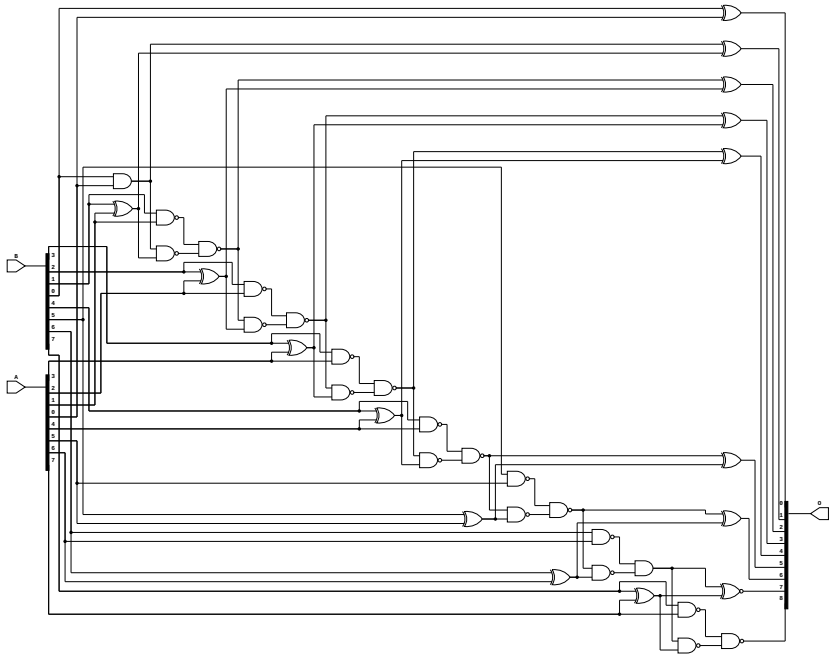

Figure C.34: `add8se_8CL` synthesised to `AND`, `XOR`, `NAND`, `OR`, `NOR` and `XNOR` gates using `netlistsvg`.

The gate-count of the circuit is shown in Table C.32.

Table C.32: Gatecounts of the netlist visualised in Figure C.34 (i.e. `add8se_8CL`).

AND	XOR	NAND	OR	XNOR	NOR	NOT	gate count	transistor count†
2	14	20	0	1	0	0	37	242

† Calculated based on values presented in Table C.1.

The directed graph illustration of the critical path is shown in Figure C.35 and the corresponding critical path gate count and delay are shown in Table C.33.

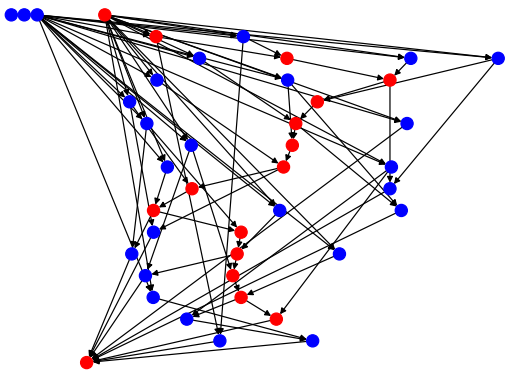

Figure C.35: Visualisation of the DAG representation of Figure C.34. The red nodes indicate the calculated **critical path**.

Table C.33: Critical path data of Figure C.35 (i.e. the directed graph of the `add8se_8CL` circuit).

Number of Gates	Propagation delay† [ns]
14	2.85

† Calculated based on assumptions presented in Table C.1.

The `add8se_8CL` is an accurate adder, meaning there are no errors to report.

C.1.13 Adder Comparison

As the case was for the multipliers, the presented adder circuits are analysed using the 3D illustration tool, to compare the metrics deduced in the previous sections. Firstly the adders are compared through the gate count for *power*, critical path gate count for *latency*, and error rate for *inaccuracy*. This is shown in Figure C.36

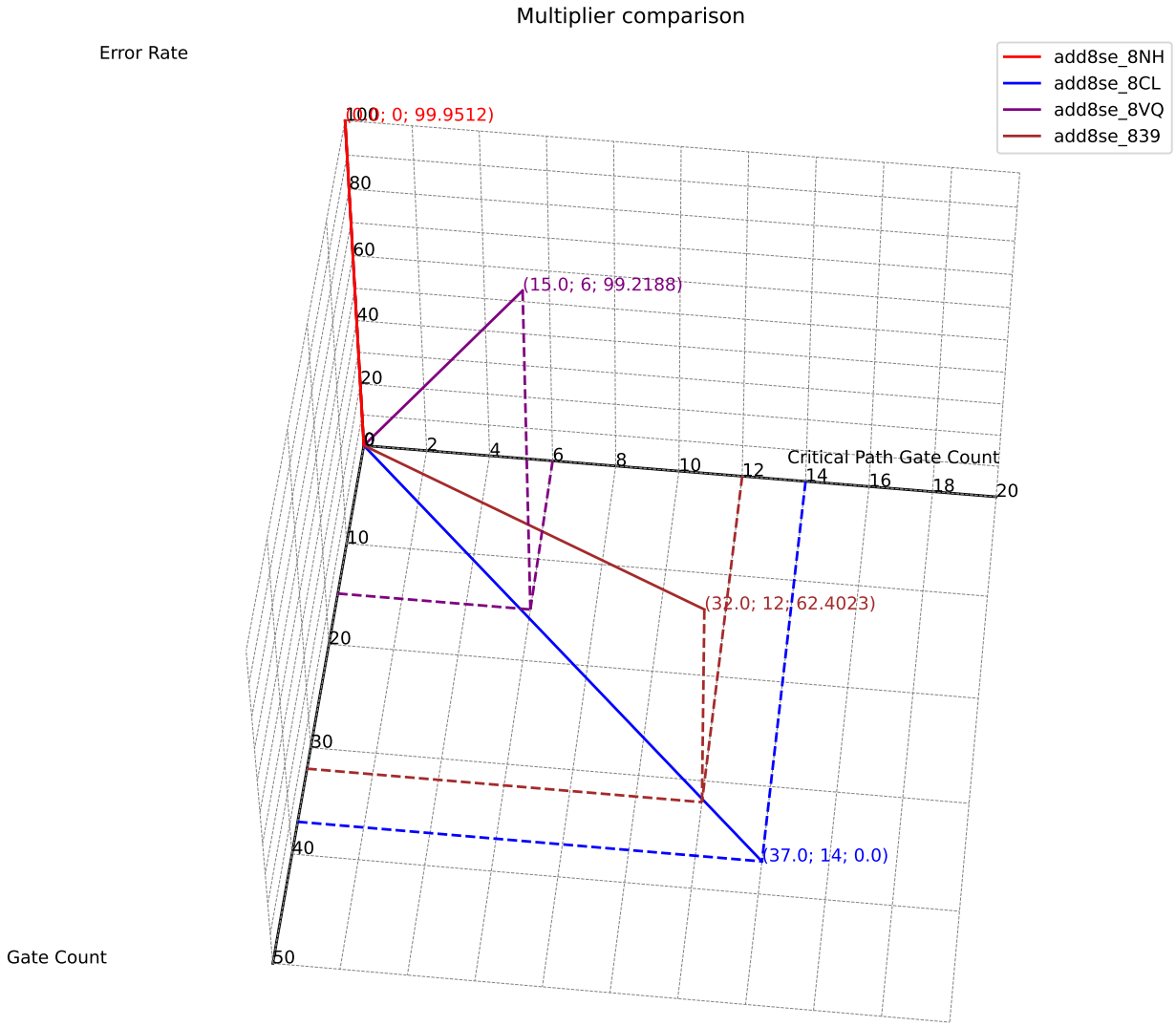

Figure C.36: 3D-plot of the metrics of `add8se_839`, `add8se_8VQ`, `add8se_8NH`, and `8CL`.

It is noticed that the adders are of significantly lower magnitude than the multipliers presented in subsection C.1.8, which makes sense since adders are simpler circuits than multipliers. From the plots shown in Figure C.36 the adders chosen do not have a pronounced trade-off between power and latency, meaning they are both reduced for an increase in error rate, across all multipliers. Note that the gate count is 0 for `add8se_8NH`, which is a peculiar case where the adder is approximated only by wires as shown in Figure C.31.

The same adders except for `add8se_8NH` are compared using WCD as the metric for inaccuracy, and shown in Figure C.37.

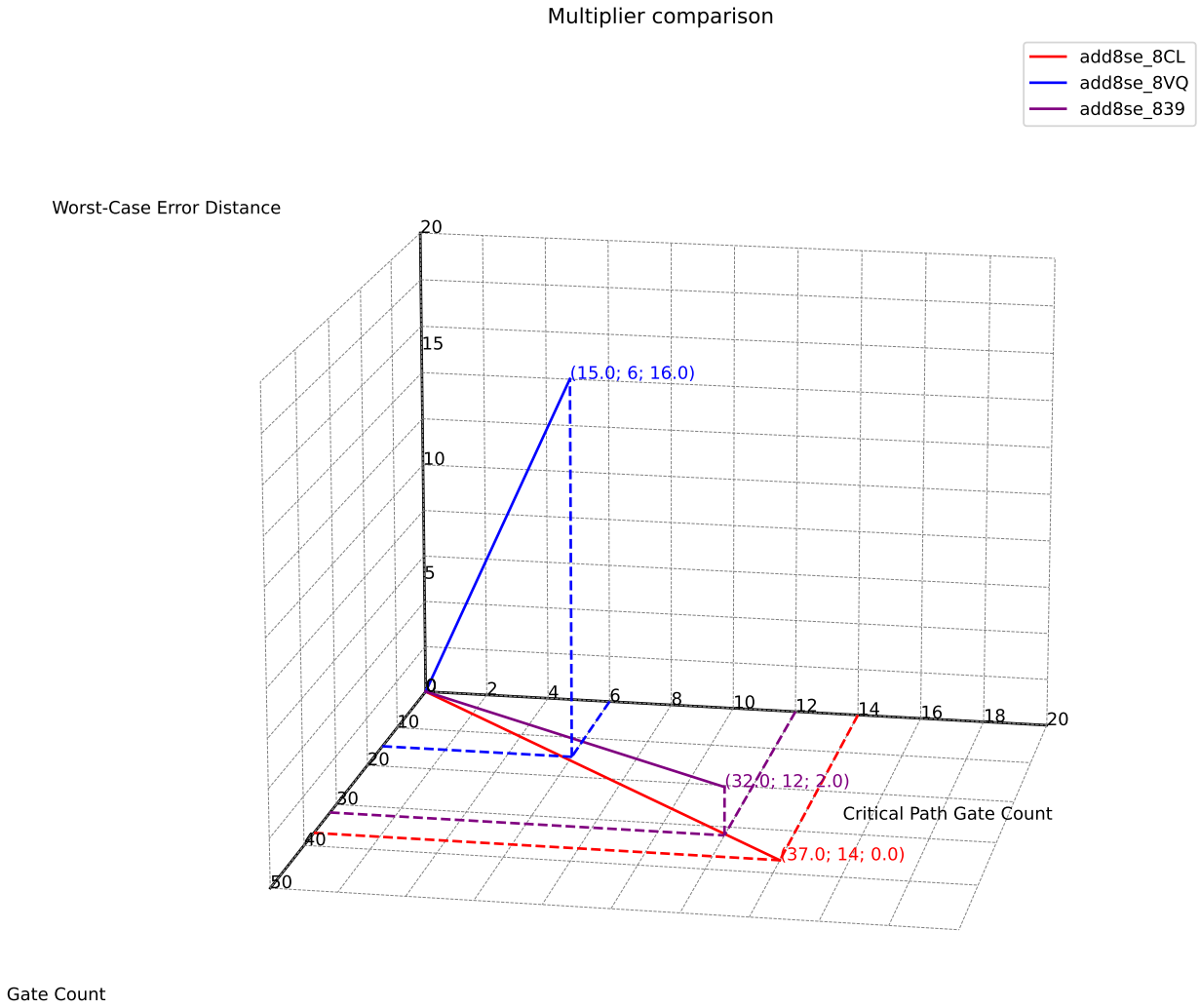

Figure C.37: 3D-plot of the metrics of `add8se_839`, `add8se_8VQ`, and `8CL`.

The tendency as the one noticed for Figure C.36 applies for Figure C.37, further implying the point pf multipliers being more complex to analyse.

Training an Approximate Arithemetic Network

D

This appendix walks through finding the best process with which the weights of the CNN with approximate arithmetics, aka. the C++ model, can be trained.

D.1 Introduction

Two networks have been designed: One with TensorFlow and one in C++. The TensorFlow model is a CNN using standard layers, optimisation algorithm, etc. from the TensorFlow API. The C++ model is architecturally identical to the TensorFlow model, however, it is possible to insert the bitwise calculations that would comprise an approximate arithmetic adder/multiplier. The weights are shared between the two models, however, since the calculations in the C++ models can be approximations, it is necessary to utilise the STE as opposed to developing, testing, and implementing a general method for performing gradient descent on the C++ model with approximate arithmetic in place. The STE has been effective for AxDNN [29] and ProxSim [30]: The gradient of a non-differentiable function $\tilde{f}(x)$ is substituted with the gradient of a related differentiable function $f(x)$:

$$\frac{\partial \tilde{f}(x)}{\partial x} \rightarrow \frac{\partial f(x)}{\partial x} \tag{D.1}$$

With this estimator, the training of the two networks should be identical, since they share the weights and the architecture. This means that it should be possible to train the models using all the built-in tools for gradient descent from the TensorFlow API.

This appendix seeks to research if the STE is warranted, how and if it is possible to train the C++ model with some approximate arithmetic, with the assumption that the STE holds.

Firstly, some terms that are going to be used in this appendix are defined:

- **Exact Epoch:** An epoch of training the models only using the TensorFlow model. The flowchart can be seen in Figure D.1; a batch is retrieved from the training set; forward passed through the TensorFlow model; the predictions are compared to the labels and a loss is calculated; the gradients are evaluated based on the calculations performed in the model and the loss[4]; the weights are updated based on the gradients; another batch is processed. This goes on until the entire training set has been processed: *One epoch.*

[4]TensorFlow has a tool for automatic differentiation called `GradientTape()`.

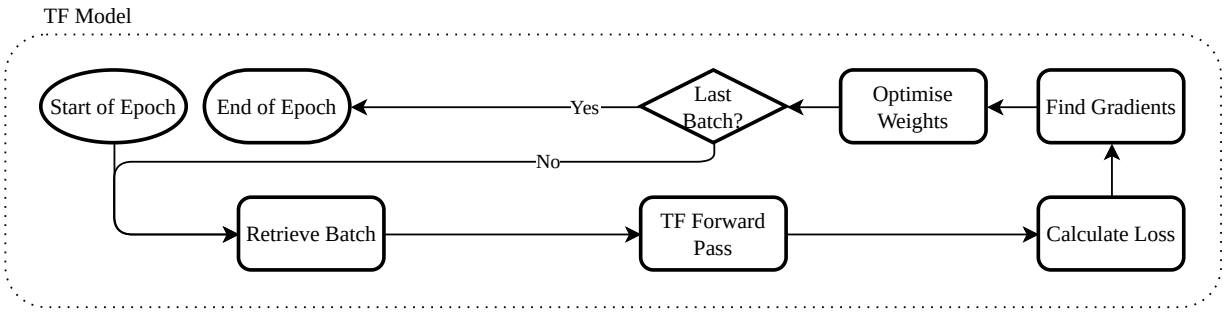

Figure D.1: Flowchart of an *exact epoch* of training the model. A batch from the training set is propagated forward in the TensorFlow model and the loss is calculated. The gradients are evaluated and used to optimise the weights. This process is repeated until the entire training dataset has been processed.

- **Approximate Epoch:** An epoch of training the models using the C++ model's prediction combined with the optimisation tools in the TensorFlow model. The flowchart can be seen in Figure D.2; the TensorFlow model's weights are exported to CSV files; a batch is retrieved from the training set and exported to another CSV file; the C++ model is called as an executable file, wherewith the weights and batch are read from the CSV files and the predictions are exported to another CSV file; the TensorFlow model performs also performs the forward pass, which is required for the automatic differentiation; the predictions from the TensorFlow model are changed to match the predictions from the C++ model; the gradients are evaluated based on the calculations performed in the model and the loss; the weights are updated based on the gradients; another batch is processed. This goes on until the entire training set has been processed: *One epoch*.

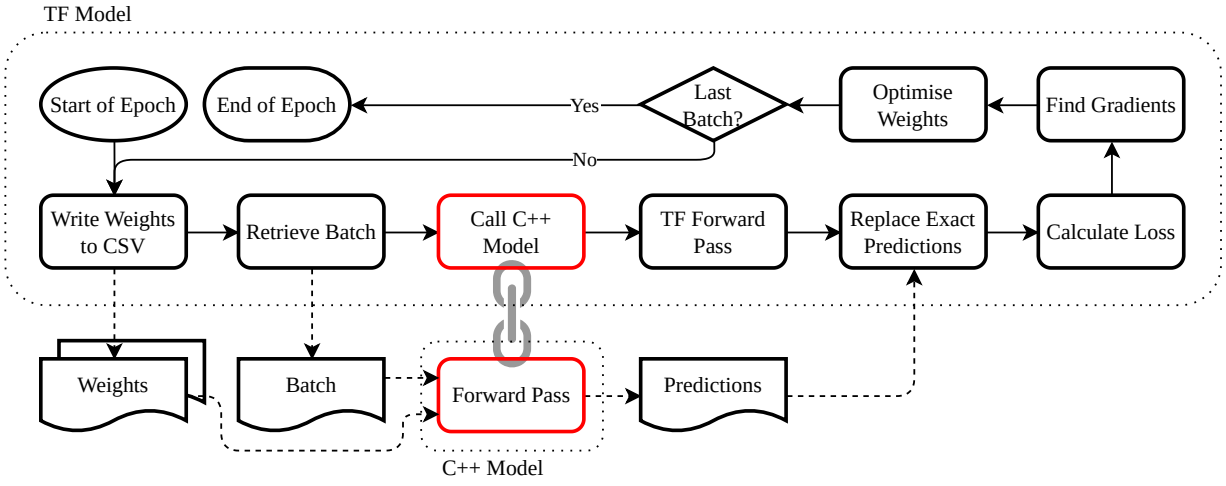

Figure D.2: Flowchart of an *approximate epoch* of training the model. The weights of the TensorFlow are exported to a CSV file. A batch from the trainings dataset is retrieved and exported to a CSV file. The C++ model is called, wherein the weights and batch are read into the program and forward propagated. A forward propagation is also performed in the TensorFlow model. The predictions from the C++ model replaces the prediction from the TensorFlow model, wherewith the loss is calculated, the gradients are found, and the weights are optimised. This process is repeated until the entire training dataset has been processed.

D.2 The Model

In this appendix the utilised *TensorFlow model* is defined as the final model of Appendix B: The task of the model is to perform *image classification* on 10 classes of the *CIFAR-100 dataset*. The dataset has been ''simplified'' by converting the images to *grayscale* by calculating the mean

over the RGB channels. Furthermore, the (32×32) images have been resized to 16×16 using the *LANCZOS3* method. Optimisation of the model is performed using the `adamax` *algorithm* and the loss is calculated as `BinaryFocalCrossentropy` with $\lambda = 0.0002$ L2 regularisation. The structure of the model is shown in Table D.1, the kernels of the convolutional and pooling layers are all 2×2, and there are no bias weights. In section B.9 the methods are briefly described.

Table D.1: (Identical to Table B.14) Summary of the final small-scale model (`n40 Model`) found by calling `model.Summary()`.

Layer Type	Output Shape	Params #[†]
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 40)	6400
MaxPooling 2D	(None, 3, 3, 40)	0
Conv2D	(None, 2, 2, 40)	6400
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

[†] Total parameters 19800 and trainable parameters 19800.

D.3 High Precision Quantisation

In *TypeCNN: CNN Development Framework With Flexible Data Types* [28] they use three different training methods: 10 epochs in floats and conversion to FXP, 10 epochs in floats and fine-tuning for 5 epochs in FXP, and 10 epochs with FXP representation used for inference and weights and the backward operation in floats. Modifications of these methods are approached in this appendix:

I) **Only approximate training:** Always updating the weights based on the predictions from the C++ model may work better, since the cost landscape would be closely related to the C++ model. However, an approximate arithmetic operation may be so distant from the STE, that the chain rule (and by extension the evaluated gradients) will not hold, and no optimisation will be performed.

II) **Only exact training:** Always updating the weights based on the TensorFlow predictions will ensure that the optimisation will be performed succesfully and a minimum will be approached. However, since the cost landscape associated with the approximate arithmetic operations may be very different, the approached minimum may not be close to a minimum for the approximate weights.

III) **Exact training, finetuning on approximate training:** Updating the weights based on the TensorFlow model and the C++ model (in that strict order) may alleviate the problems associated with the individual methods: Using the *exact training* to approach some minimum, and changing the cost landscape to be more related to the approximate circuit, may let the model approach a minimum in the cost landscape of the C++ model.

An investigation of the three scenarios is proposed; training and evaluating the C++ model and the TensorFlow model three times:

I) 50 *approximate epochs*
II) 50 *exact epochs*
III) 45 *exact epochs* followed by 5 *approximate epochs*

To ensure that the integration of *approximate/exact epochs* performs optimisation as intended, the chosen approximation for the preliminary training runs is conversion to FXP (fixed point) with **20 bits for accuracy** and **no approximate multiplier**.

In Figure D.3 the result of each of these training processes can be seen. Note that the values `accuracy` and `val_accuracy` are found using the C++ network: `accuracy` is found by passing the `train dataset` through the C++ network and calculating the rate of correct predictions, `val_accuracy` is found by passing the `test dataset` through the C++ network and calculating the rate of correct predictions.

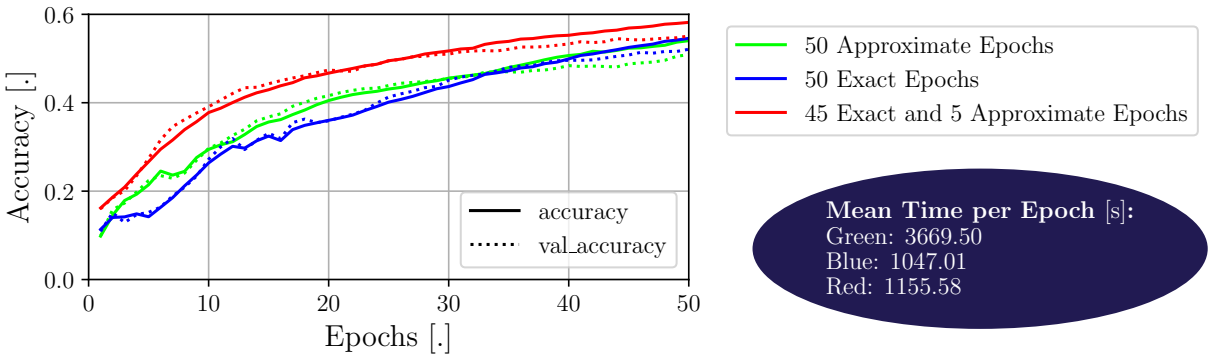

Figure D.3: Accuracy of C++ network with quantisation noise, 20 bits for precision. Accuracy of C++ network on the train and test datasets: `accuracy` and `val_accuracy`, respectively. Furthermore, the *mean time per epoch* is noted for each of the three trainings.

In Figure D.3 it is clear, that the run using *45 exact and 5 approximate epochs* is performing better than the other two. For the training with *50 exact epochs* it should be noted, that the low accuracy must be from an unlucky start (or a lucky start for the *45 exact and 5 approximate* training run) since the training process is identical to that of *45 exact and 5 approximate*; the accuracy of these two training runs should be close to identical in the first 45 epochs. This would likely not be seen, if the plot showed the mean of multiple identical runs.

Important for these three runs is that the optimisation method is working when the predictions from the TensorFlow model are replaced with the predictions from the C++ model (see Figure D.2). This suggests that the implemented method for performing *approximate epochs* is **working as intended**.

In all three of these training runs, the approximation was ‘‘soft’’, i.e. quantisation with 20 bits for precision. The next step is to test the same three training scenarios with something less ‘‘soft’’ and in section D.4 a test with only eight bits for precision is tested.

D.4 Low Precision Quantisation

The setup for the *low precision quantisation* test is almost identical to the *high precision quantisation* presented in section D.3, but the precision of the values in the C++ model has been lowered from 20 bits to 8 bits, still **not using an approximate multiplier**. In Figure D.4 the results can be seen.

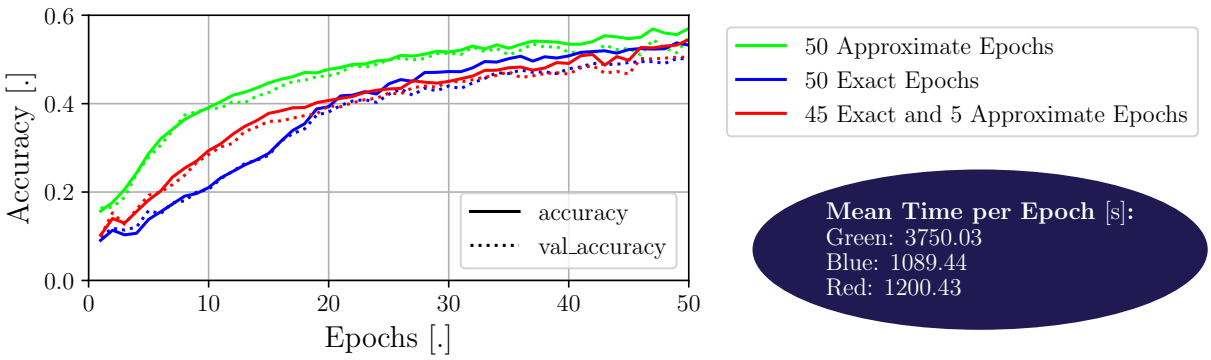

Figure D.4: Accuracy of C++ network with quantisation noise, 8 bits for precision. Accuracy of C++ network on the train and test datasets: `accuracy` and `val_accuracy`, respectively. Furthermore, the *mean time per epoch* is noted for each of the three trainings.

Unlike Figure D.3 the training run with *50 approximate epochs* outperforms the two runs with *exact epochs*. The results suggest that using the *approximate epochs* will yield a higher accuracy, however, the time is also worth investigating. As presented in the dark-blue ellipsis, the time it takes to perform an *approximate epoch* is significantly longer than performing one *exact epoch*: The run with *50 approximate epochs* took around $3750,03\ \text{s/epoch} \cdot 50\ \text{epochs} \approx 2$ days 4 hours and 5 minutes, whereas the run with *50 exact epochs* only took $1089,44\ \text{s/epoch} \cdot 50\ \text{epochs} \approx 15$ hours and 8 minutes. For that reason, the compromise of using a $45:5$ split of *exact* and *approximate epochs* is chosen as the method for training the C++ network.

Now that a method of training the C++ network is defined, the next step is testing the training with an **approximate multiplier** in the C++ network

D.5 Multiplier Approximation - `mul8s_1KV9`

The chosen multiplier, `mul8s_1KV9` was presented in subsection C.1.2. The relevant error metrics are presented again in Table D.2.

Table D.2: (Copy of Table C.8) Error-metrics of the distribution presented in Figure C.6.

Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
4.25	34.25	68.75 %	17	1.4

Since the test is essentially creating training two models: An exact model in TensorFlow and a C++ model (first 45 epochs only inference), the accuracies for both are evaluated. The number of precision bits is chosen to be 7. In Figure D.5 the results can be seen.

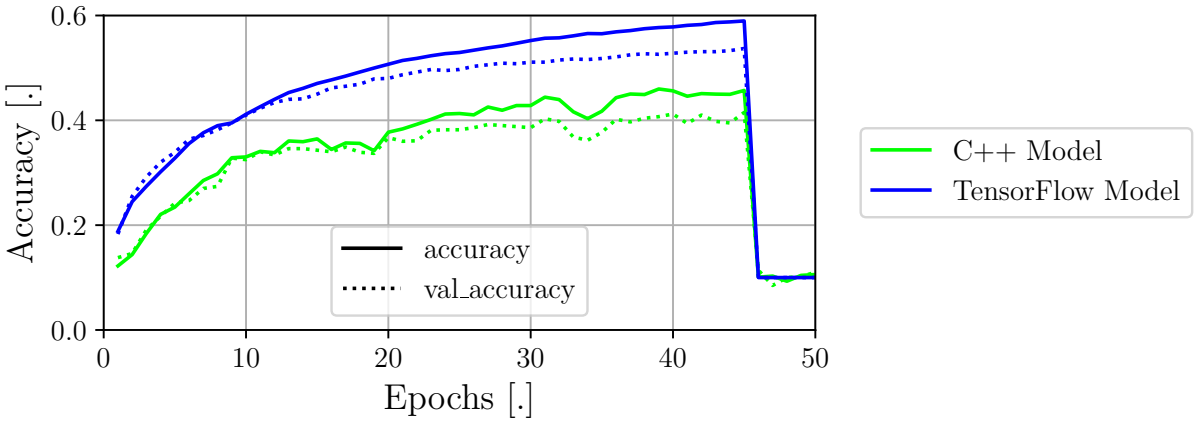

Figure D.5: Accuracy of C++ network with an approximate multiplier, `mul8s_1KV9`. The first 45 epochs of the training of the C++ network are inference only, why the TensorFlow model is plotted for comparison. The last 5 epochs of the training are with the predictions from the C++ network.

During the 45 *exact epochs* it is clear that the accuracy of the C++ network is steadily increasing. Interestingly, after the 46th epoch (the first *approximate epoch*), the accuracies of the networks fall to around 0.1, corresponding to random guesses. This puts some doubt into the training process previously devised. To overcome this drastic drop in accuracy three paths are tested:

I) **Training purely with *approximate epochs*:** As shown in Figure D.3 and D.4, training using the approximate arithmetic from the beginning may lead to high accuracy. Perhaps, using only *approximate epochs* would prevent the drastic drop in accuracy seen in Figure D.5.

II) **Pre-training of the TensorFlow model:** In Figure D.5 the accuracy of the C++ model is increasing alongside the TensorFlow model, and the inference at the 45th epoch is pretty decent. Since the objective of the network is not subject to change, it may be beneficial to have the weights of a pre-trained TensorFlow model, and use the weights as a springboard from which the C++ model can be finetuned. This would reset the training just as the finetuning were to commence since the optimisation algorithm `adamax` would not know the previous gradients.

III) **Changing the optimisation algorithm:** A possible explanation for the drastic drop in accuracy in Figure D.5 is the optimisation algorithm: `adamax`, which has an adaptive learning rate that is based on the first and second moments of previous gradients (see section B.9). The difference in the evaluated loss may be so large it is irreconcilable for the optimiser, and the updated values for the weights are off. At the shift from *exact epochs* to *approximate epochs* the optimiser could be replaced by another, whose learning rate is not adaptive or based on the previous gradients.

D.5.1 Training purely with *approximate epochs*

In Figure D.6 it is clear, that *training purely with approximate epochs* is not the solution, as the accuracy does not significantly differ from a random guess at any time during the training. This result might suggest that the cause of the drastic drop is to be found in the process of *approximate epochs*.

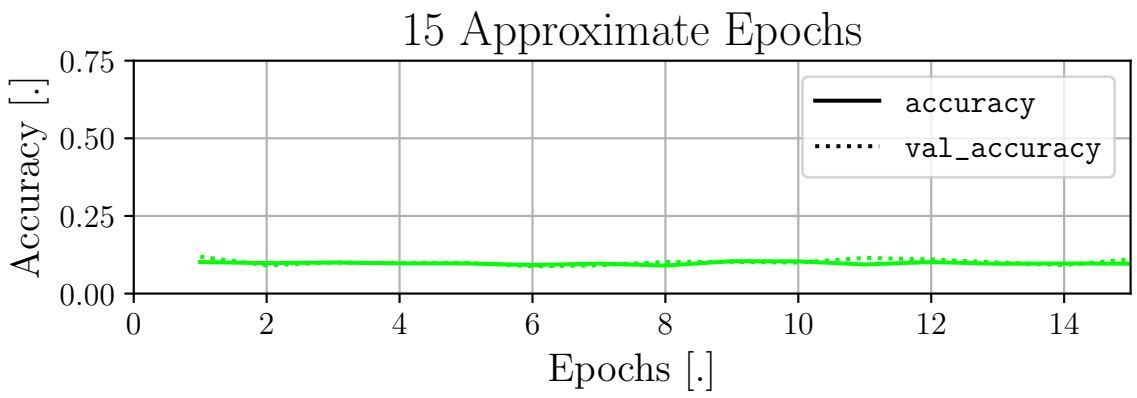

Figure D.6: `mul8s_1KV9` trained used *approximate epochs*. The accuracy is somewhat stable at ~ 10 %, indicating that the model is **not improving** during optimisation.

D.5.2 Pre-training of the TensorFlow model

The results of using the weights from a training run where the TensorFlow model has been trained on 250 *exact epochs*, and then performing 15 *approximate epochs* can be seen in Figure D.7. Since the weights are read in from CSV files just before the training, the `adamax` optimisation has not had time to update its learning rate. However, the same problem arises as in Figure D.5: the accuracy drastically drops, and the predictions from the C++ network are no better than a random guess. ''Restarting'' the learning rate of the adamax optimisation algorithm is not the solution.

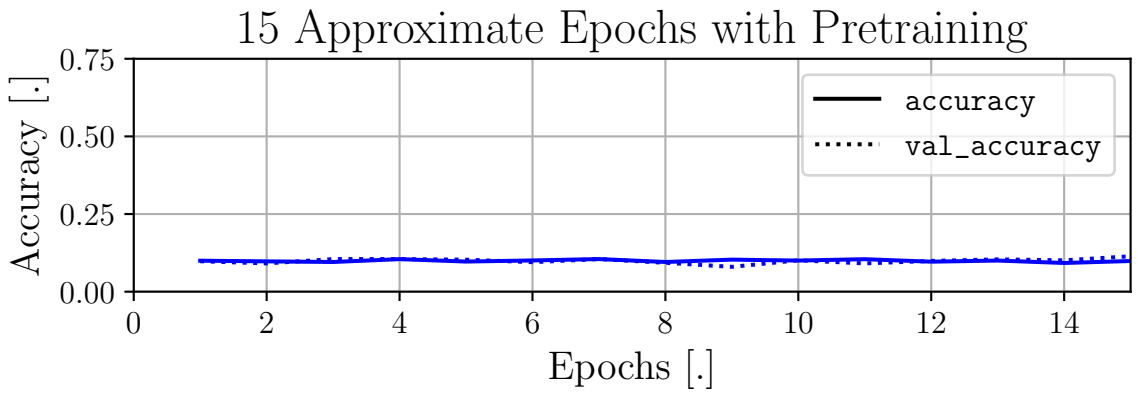

Figure D.7: `mul8s_1KV9` trained using `approximate epochs` on a pre-trained set of weights. The accuracy is somewhat stable at ~ 10 %, indicating that the model is **not improving** during optimisation.

D.5.3 Changing the optimisation algorithm

Again, the weights of the pre-trained TensorFlow model are read from CSV files, however, in this run the `adamax` optimisation algorithm has been exchanged for SGD with a learning rate of 0.00005. In Figure D.8 the results of this test can be seen. The drop-off seen in the tests above has disappeared, however, it has been replaced with a slow drop-off. This may be rectified using another learning rate.

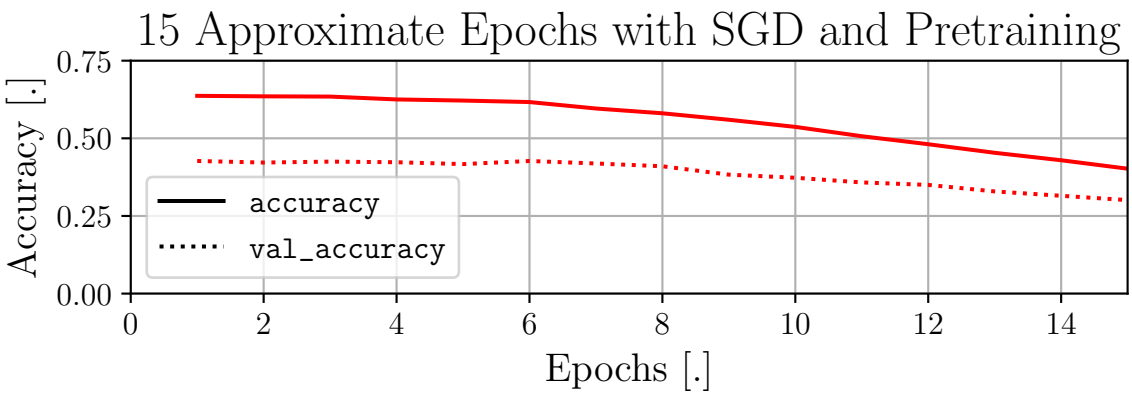

Figure D.8: `mul8s_1KV9` trained using `approximate epochs` on a pre-trained set of weights with SGD. Although the accuracy is not increasing, it is not decreasing in the first 6-7 epochs, indicating that **optimisation may be possible**.

The resulting accuracies from using the same set of weights as the starting point of Figure D.8 and adjusting the SGD learning rate can be viewed in Figure D.9:

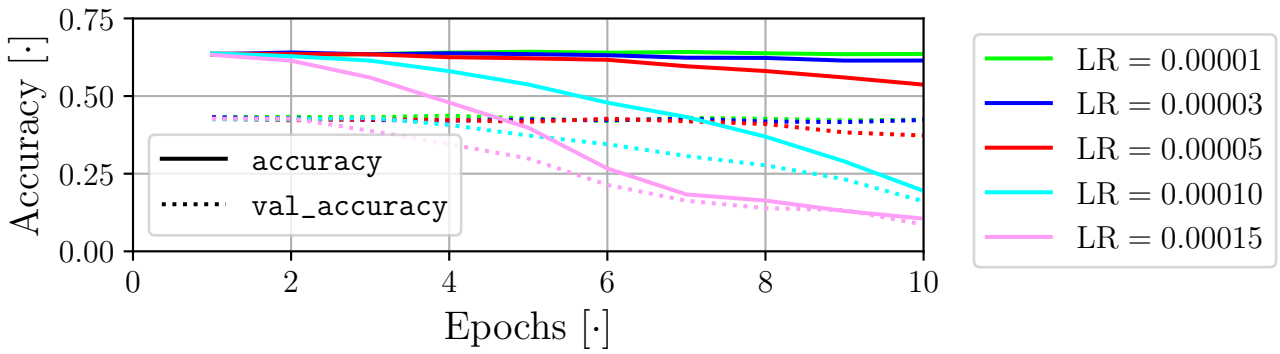

Figure D.9: Finetuning with different SGD learning rates. 10 *approximate epochs* performed on pre-trained network with varying learning rates.

Every value for the learning rate results in the model ‘‘optimising’’ toward a lower accuracy. No clear path to rectifying the problem of training the approximate network has presented itself. In order to investigate this problem, it is seen as beneficial to reduce the amount of computations required for a forward pass, in order to speed up the evaluation process.

D.5.4 Debugging with Smaller Network

The results in Figure D.9 seem strange, since the weights change, however, the optimisation process is seemingly worsening the outcome of the model. Debugging the implemented CNN is infeasible due to the long duration of each epoch; the slowest layer is the 2nd convolutional layer and the ‘‘complexity’’ is reduced to effectivise the debugging process. This is done by reducing the amount of filters from 40 to 2, effectively reducing the amount of parameters from 19.800 to 7.600. This change is also applied to the TensorFlow model as well as the C++ model.

Table D.3: ummary of the smaller network found by calling `model.Summary()`.

Layer Type	Output Shape	Params #[†]
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 2)	320
MaxPooling 2D	(None, 3, 3, 2)	0
Conv2D	(None, 2, 2, 40)	320
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

[†] Total parameters 7600 and trainable parameters 7600.

The TensorFlow model from Table D.3 is pre-trained 45 epochs and the resulting weights are saved to be utilised in the following small tests.

A Preliminary Test of the Optimisation Algorithms is performed to ensure, that the new small model is trainable and that it is possible to increase the accuracy using either of the optimisation algorithms. The result of training 10 epochs with the two optimisation algorithms from the weights of a pre-trained network can be seen in Figure D.10. The epochs are *approximate epochs* and the C++ model has 20 bits precision and an accurate multiplier.

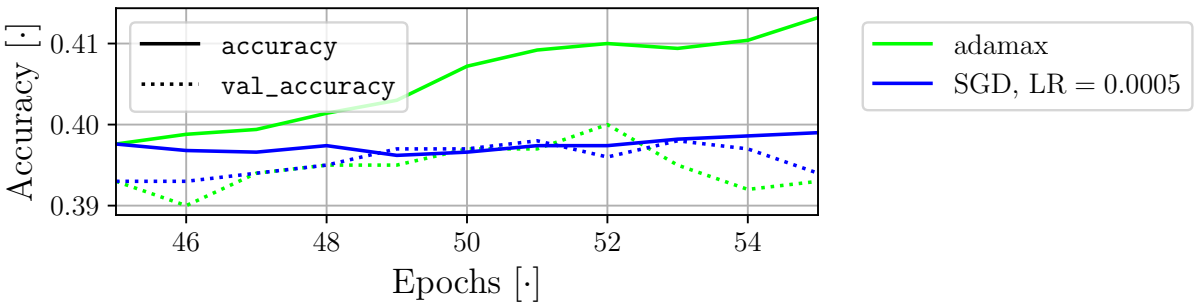

Figure D.10: Preliminary test with only 2 filters in the second layer. Two separate runs initialised with identical weights from a network with 45 epochs of pretraining.

Based on the results from Figure D.10, the accuracy of the network increases regardless of which optimisation algorithm is in use; the `accuracy` has significantly improved using adamax, and the `accuracy` has slightly improved using SGD. This is interpreted as *the reduced model is trainable* and *both adamax and SGD should be able to train the reduced network.*

Retrying Different SGD Learning Rates to see whether the quantisation is the cause of a downward trend seen in Figure D.9. Perhaps the results seen Figure D.9 are caused by the bit-precision rather than multiplication-approximation. In Figure D.11 an accurate 8×8 multiplier with 6 bits reserved for precision has been trained using SGD with two different values of *learning rate*.

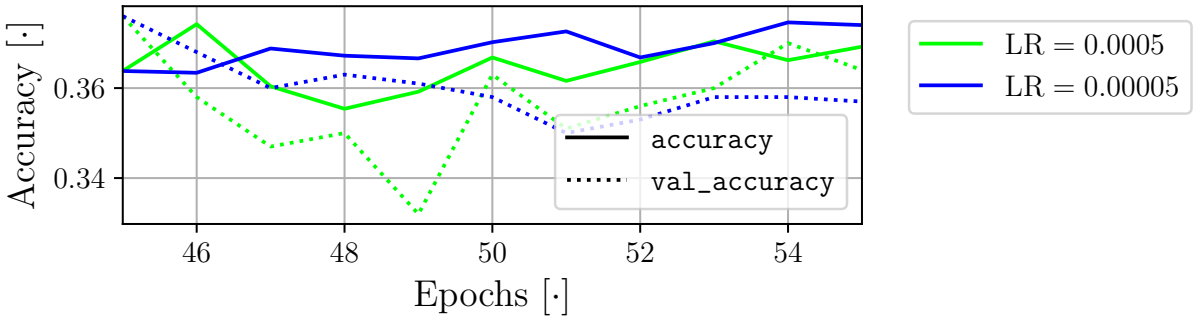

Figure D.11: SGD finetuning with 6 bits for precision on an 8 × 8 accurate multiplier. The 2nd axis is scaled to showcase the slight ''upwards trend''.

Both of the learning rates result in accuracies with a slight ''upwards trend'', which is interpreted as *it is possible to finetune with SGD on accurate* 8 × 8 *multipliers, quantisation with 6 precision bits does not add enough noise to make training impossible.*

Retrying adamax since the SGD learning curve has been somewhat rectified by lowering the amount of precision bits, perhaps this will also hold for adamax. For this reason, adamax is reinvestigated in Figure D.12, wherein the *mean* accuracies across 5 runs of 10 epochs with a set of approximate multipliers with different *error metrics* can be viewed. Note that ''accurate'' in this context should be interpreted as a ''*'' has been inserted as the multiplication instead of a function emulating a circuit, this will avoid overflow but not quantisation.

Table D.4: Error-metrics of the different multipliers used in Figure D.12.

Multiplier	Mean Absolute Error	Mean Square Error	Error Rate	Worst-Case Error Distance	Mean Hamming Distance
1KV6	0	0	0 %	0	0
1KV8	1.25	3.75	50 %	5	0.75
1KV9	4.25	34.25	68.75 %	17	1.4
1L12	2016	$7.283 \cdot 10^6$	98.053 %	8064	5.06
1L2J	53.3	5462	74.6 %	255	3.25

Note: The formulas for these metrics were presented in chapter 4

For all 1KVx and ''accurate'' there is no dropoff and all the models are training (improving their accuracy). For the 1Lxx multipliers there is no upward trend, however, it is not guaranteed that it is possible to train the multipliers, and the error metrics associated with these two approximate multipliers are very large compared to the other 3.

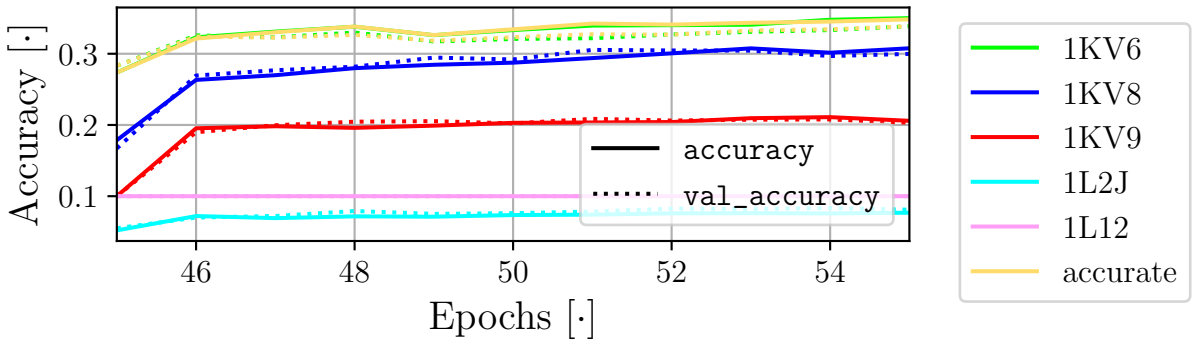

Figure D.12: Adamax finetuning with 5 bits for precision on various multipliers.

Seemingly, finetuning the weights by using adamax as optimisation algorithm and an approximate multiplier in the C++ model yields a positive result. This is the opposite of what what seen in Figure D.5, and the only parameters that have been changed are the number of precision bits and the number of filters in the second convolutional layer. These results are interpreted as *the dropoff seen in Figure D.5 may be caused by the choice of the number of precision bits or the architecture.*

Testing Variations of the Number of Precision Bits on the multiplier used in Figure D.5, `mul8s_1KV9`, might give an insight into the effects of the chosen bit precision. In Figure D.13 the resulting accuracies given 45 epochs pre-training, 10 epochs of finetuning with adamax as the optimisation algorithm, and `mul8s_1KV9` with a varying number of precision bits is visualised.

Neither *2 precision bits*, *3 precision bits*, or *4 precision bits* are training and stay at 10 %, the same as a random guess. This is likely a consequence of too much quantisation, as the values of the LSBs are 1/4, 1/8, and 1/16, respectively, which are not matching effectively with the regularised weights. Furthermore, the small changes in the weights at each epoch/iteration may not be transferred to the C++ model, as the changes may not be large enough to ''overcome'' the quantisation. However, for *5 precision bits* and *6 precision bits*, the first epoch of finetuning significantly improves the accuracies, suggesting that the changes are transferred and that they are improving the accuracy of the models.

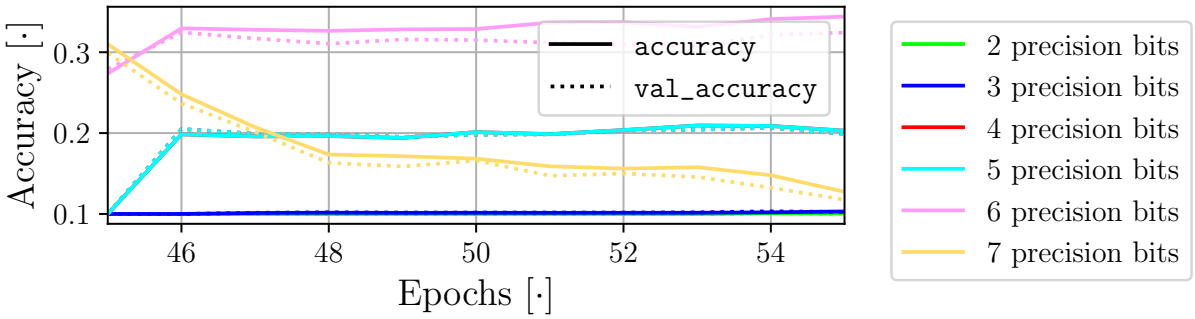

Figure D.13: Adamax finetuning with varying number of precision bits on `mul8s_1KV9`.

For *7 precision bits* the accuracy worsens at each epoch, this is reminiscent of the training using SGD from Figure D.9, which also was tested with 7 precision bits. The implication of having 7 precision bits in a *signed* 8×8 multiplier, is that the representable values exist between $[-1;\ 1-2^{-7}]$; if the multiplicands are in the said interval no problem should arise. Using regularisation may force the weights into this range, however, the inputs are not taken into account. In the first layer, it is known that the input has been *normalised*, *quantised*, and *truncated*, ensuring they are in the

interval. However, the outputs of each hidden layer (being the inputs to the following layer), are not ensured to be in the same interval. Say there are 40 input channels and the kernel size is 2×2, the each ''pixel'' of the resulting FM (Feature Map) is the sum of $2 \cdot 2 \cdot 40 = 160$ multiplications, which may cause overflow/underflow.

From Figure D.13 it is clear, that *it is possible to fintune using adamax* and *given `mul8s_1KV9` and only 2 filters in the second convolutional layer the best choice of precision bits is 6.*

Finetuning with 45 Approximate Epochs using adamax as the optimisation algorithm is performed. This test is performed to ensure that the conclusion from the previous test is correct. In Figure D.14 the results of two runs are seen. Weights from a pre-trained TensorFlow model are applied to the C++ model. 45 *approximate epochs* are then performed, and the inferred accuracy of the C++ network is saved and visualised in Figure D.14. Furthermore, the accuracy of a TensorFlow model with the same architecture and starting weights, trained for 45 epochs and averaged over 5 runs can be seen.

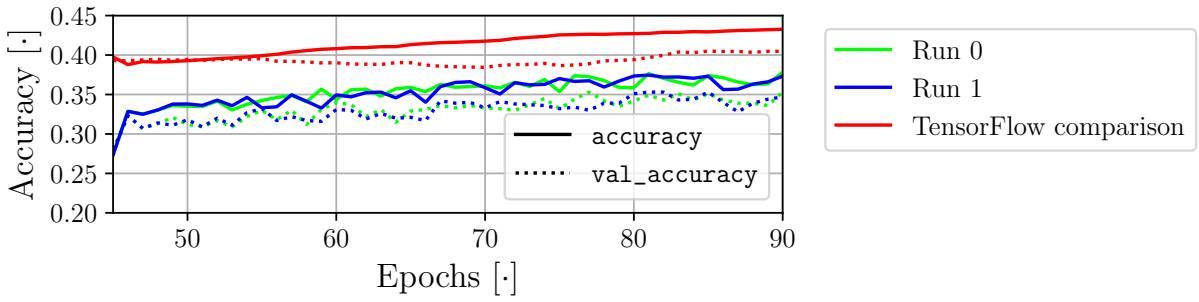

Figure D.14: Finetuning with 45 *approximate epochs* with `mul8s_1KV9` using 6 bits for precision.

Both runs show that the C++ model with an approximate multiplier is training and the optimisation is working. Comparing the two runs with the TensorFlow model's accuracies the difference is only around 5 percentage points, which is impressive given the approximate multiplier and only 8-bits to represent the weights. The solution was changing the number of precision bits to match with the network. This suggests, that the problem in Figure D.5 may have been caused by overflow/underflow.

D.6 Conclusion

In this appendix, different methods were approached and tested, to develop a method for training a CNN which uses *approximate arithmetic*. Performing *50 approximate epochs*, *50 exact epochs*, or *45 exact* and *5 approximate epochs* (see Figure D.2 and D.1), yielded no significantly different results in two cases of quantisation, 20 bits for precision and 8 bits for precision. However, in section D.5 it became clear, that given the approximate multiplier `mul8s_1KV9` using *45 exact* and *5 approximate epochs* did not yield good results; evaluating the accuracy of the C++ network during the *45 exact epochs* it was clear that by pure inference, the network was trained, however, at the shift from *exact epochs* to *approximate epochs* the accuracy fell to that of a random guess. A couple of tests with ideas to remedy the problem were performed, one of which worked decently: Replacing the *optimisation algorithm* at the shift from *exact* to *approximate epochs*. However, further testing debunked this solution, since no upward trend (optimisation) could be performed.

To perform a sequence of ad-hoc tests for debugging the training process, a significantly smaller CNN was created, by reducing the number of filters in the second convolutional layer in both

the TensorFlow and C++ models, reducing the number of parameters from 19800 to 7600. This significantly sped up the debugging process. Firstly, a preliminary test with adamax and SGD was performed with 20 bits precision and an *accurate* multiplier, whereby it was concluded, that *this smaller model is trainable using either optimisation algorithm*. Secondly, two of the SGD learning rates were tested on with an 8×8 accurate multiplier, whereby it was concluded, that *it is possible to finetune with SGD on accurate* 8×8 *multipliers, quantisation with 6 precision bits does not add enough noise to make training impossible*. Perhaps 6 precision bits may also work with adamax; 5 different multipliers were tested and the approximate multipliers with relatively low error characteristics showed positive results, i.e. the *approximate epochs* were effective on the accuracy. These results are interpreted as *the dropoff seen in Figure D.5 may have been caused by the number of precision bits or the architecture*. Another test was performed to see the effects of changing the number of precision bits, where `mul8s_1KV9` was utilised again. Neither *2 precision bits*, *3 precision bits*, or *4 precision bits* improved in accuracy during training and stayed at 10 %, the same as a random guess. This is likely a consequence of too much quantisation, as the values of the LSBs are $1/4$, $1/8$, and $1/16$, respectively, which are not matching effectively with the regularised weights. Furthermore, the small changes in the weights at each epoch/iteration may not be transferred to the C++ model, as the changes may not be large enough to ''overcome'' the quantisation. However, for *5 precision bits* and *6 precision bits*, the first epoch of finetuning significantly improves the accuracies, suggesting that the changes are transferred and that they are improving the accuracy of the models. For *7 precision bits* the accuracy worsens at each epoch. The implication of having 7 precision bits in a *signed* 8×8 multiplier, is that the representable values exist between $[-1;\ 1-2^{-7}]$; if the multiplicands are in the said interval no problem should arise. Using regularisation may force the weights into this range, however, the inputs are not taken into account. In the first layer, it is known that the input has been *normalised*, *quantised*, and *truncated*, ensuring they are in the interval. However, the outputs of each hidden layer (being the inputs to the following layer), are not ensured to be in the same interval. Say there are 40 input channels and the kernel size is 2×2, the each ''pixel'' of the resulting FM (Feature Map) is the sum of $2 \cdot 2 \cdot 40 = 160$ multiplications, which may cause overflow/underflow. The problem of underflow/overflow is thus dependent on the number of precision bits and the architecture, why the CNN model will be permanently replaced by the small version, since the training process works, furthermore, for any approximate multiplier only 6 precision bits will be used.

The results from finetuning a set of weights from a pre-trained network in Figure D.14 incidentally simplified the training process. Three things are noteworthy:

I) **Exact epochs** yielded positive for the C++ model, wherewith the inferred accuracy rose.
II) **Restarting the learning rate of adamax** did not negatively affect the finetuning, i.e. the finetuning can be performed independent of information from previous epochs. Given a set of pre-trained weights, the required amount of epochs could potentially be lowered.
III) **The problem is static** and consecutive uses of the same weights should yield the same results (without training).

These three remarks in combination should allow the simplification of the training process, i.e. given the weights of a trained CNN from TensorFlow, the only training necessary is *finetuning* with *approximate epochs* using adamax as the optimisation algorithm; a TensorFlow model has been trained 45 epochs and the weights have been saved. Training the C++ model given some approximate arithmetic circuit, should be possible by taking the pre-trained weights and finetuning for some epochs, using the defined *approximate epochs*.

D.7 Discussion

Some of the tests have only been run 1 time due to time restrictions. Even 50 *exact epochs* takes $50 \cdot 1089.44$ s $\approx$ 15 hours 8 minutes (based on Figure D.4) and repetitions were deemed too costly. The effect of this choice may have caused fallacious conclusions and missed opportunities for optimisation of the process of training the C++ model. Given apt time, the conclusion should be adjusted by performing an adequate amount of repetitions.

Although the conclusion states that overflow/underflow is the culprit for the dropoff in Figure D.5 and that it was solved by lowering the number of precision bits, the chosen amount of precision bits is not guaranteed to work for every architecture nor every approximate multiplier. Let's reexamine the example of potential overflow: *40 input channels and the kernel size is* 2×2*, the each "pixel" of the resulting FM (Feature Map) is the sum of* $2 \cdot 2 \cdot 40 = 160$ *multiplications*. The multiplicands are represented with 8 bits, 6 of which are precision bits, i.e. formatted as Q2.6. To ensure no information is lost, the product of the multiplications should be Q4.12, and 160 of these products should be calculated and summed, requiring the format Q11.12 to avoid overflow/underflow. This sum is then processed by the *activation function*, in this case, ReLU, and the format Q11.12 is still required to avoid overflow. This value is then the input of another 8-bit multiplier. The format is incongruent the multipliers, and a lot of information may be lost. In the implementation, the multiplicands are always transformed to Q2.6 just before the 8×8 approximate multipliers. Overflow is thus still a possibility.

Testing the Probabilistic Model E

This appendix walks through the testing of the probabilistic model. Two specific tests are conducted.

Firstly, a test of the prediction accuracies using the probabilistic error model developed and implemented in section 5.3. This result is compared to the prediction accuracies of the deterministic error model, simulated using approximate arithmetic circuits. The inference is made on the *test-data set* of the CIFAR 100, reduced to 10 classes. This test is performed on the CNN network using weights *trained accurately* and for each 5^{th} epoch when training with STE on the deterministic system.

Secondly, a test of the output vector of the probabilistic model, which is compared to the output vector of the deterministic model. For this test, each pixel for the input images is random, with a uniform distribution between 0 and 1. The purpose of this test is to be able to analyse the distributions of the deterministic and probabilistic outputs, to evaluate the statistical similarity of these.

In Figure E.1 the probabilistic model can be seen in the context of the rest of the benchmarking tool.

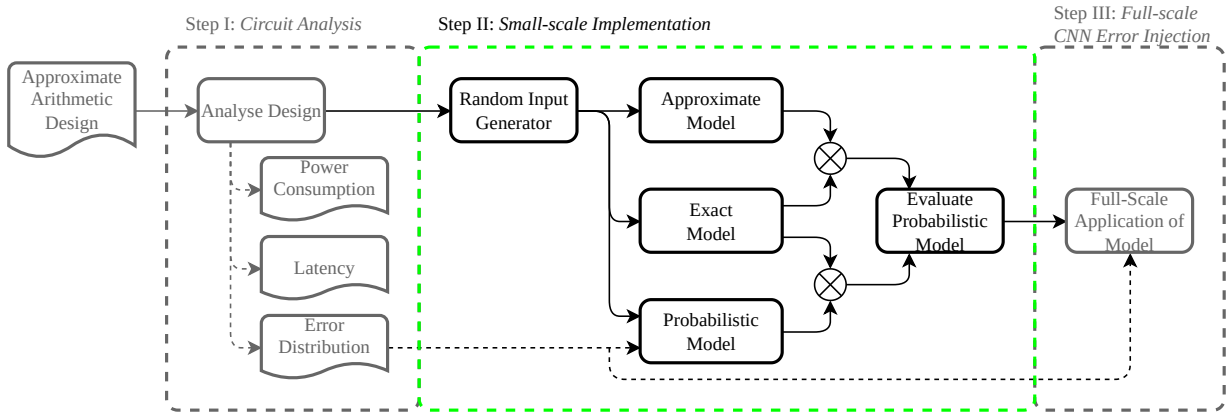

Figure E.1: (Copy of Figure 5.1) Functional diagram of the *benchmarking tool*. An *approximate arithmetic design* is supplied to the tool and is processed through three steps. The evaluated error distribution from **step I** is utilised in **step II**, where three NNs are created: An *approximate model*, an *exact model*, and a *probabilistic model*; an input is generated and propagated through each model and the error is computed with respect to the *exact model*. Based on the error at the output, the *probabilistic model* is evaluated based on ''how well it represents the *approximate model*''.

The Exact Model is a baseline implementation of the architecture seen in Table E.1. The model is generated using TensorFlows API, creating a sequential keras model with the layers from ''Layer Type''.

The Approximate Model is the C++ model developed in section 5.2 with an identical architecture to the *exact model*, i.e. the architecture seen in Table E.1.

The Probabilistic Model is a simplification of the *approximate model*, where error induced from the approximate arithemtic circuits is modelled as ''noise'', which is then applied using custom layers. The architecture is identical to the other two models, and can be viewed in Table E.1.

Table E.1: Summary of the network architecture found by calling `model.Summary()`.

Layer Type	Output Shape	Params #†
Conv2D	(None, 15, 15, 40)	160
MaxPooling 2D	(None, 7, 7, 40)	0
Conv2D	(None, 6, 6, 2)	320
MaxPooling 2D	(None, 3, 3, 2)	0
Conv2D	(None, 2, 2, 40)	320
Flatten	(None, 160)	0
Dense	(None, 40)	6400
Dense	(None, 10)	400

† Total parameters 7600 and trainable parameters 7600.

E.1 Workspace Setup, Software and Hardware

The tests performed in this appendix have been run on a cloud-computer set up in strato. The hardware overview can be seen in Table E.2. Using the `screen` command in Linux, a detachable screen was created, whereby the tests could run in the background and overnight.

Table E.2: Information about the cloud-computer used in this appendix. Information taken from `https://strato-new.claaudia.aau.dk/` under the *Instances* tab.

Flavor Name	Flavor ID	RAM	VCPUs	Disk
AAU.CPU.g.16-64	10a7313a-2e8c-421a-9cdc-8e1283ef905b	64GB	16 VCPU	50GB

To ensure that the tests are reproducable, important packages and their versions are tabularised in Table E.3. The versions are found using the `print-versions` package from Python, and the method `print_versions(globals())`.

Table E.3: The software packages used in the appendix. The names and versions were found using the `print-versions` package in python.

Package	Explanation	Version
`numpy`	Math package	1.26.4
`pandas`	Math package used for reading data from ''.csv'' files	2.2.2

Continued on next page

Table E.3: (Continued)

`tensorflow`	The chosen machine learning API, utilised for data manipulation, loading datasets, etc.	2.16.1
`tensorflow_datasets`	Used for dedicated dataset manipulation	4.9.4
`csv`	Used to export weights to CSV-files	1.0

E.2 Procedure

The procedure of **test I** follows the list below and the scripts and data can be found in Appendix A under `/convergence_of_stat_and_approx_model/`:

I) Perform the following steps for each multiplier: `mul8s_1KV8` and `mul8s_1KV9`
II) Fetch the pre-trained weights (45 epochs using the *exact model* from section 5.1), can be found under `/convergence_of_stat_and_approx_model/training_1KVx/2_kernels_-45_epochs_start`
III) Train the *approximate model* from section 5.2 with the multiplier in-place for 45 epochs, saving the weights with a 5 epoch interval, i.e. at 45, 50, 55, 60, 65, 70, 75, 80, and 85, can be found under `/convergence_of_stat_and_approx_model/1KVx_weights/`
IV) For each of the sets of weights perform the following steps
V) Instantiate the *probabilistic model* from section 5.3 and import the weights
VI) Evaluate the accuracy of the probabilistic model on the training- and test datasets five times given the set of weights. The accuracies are saved in:
    - `/convergence_of_stat_and_approx_model/mul8s_1kvx_stats_and_approx.csv`

The procedure of **test II** follows the flowchart in Figure E.2. Furthermore, the script and collected data can be found in Appendix A under `/statistic_test_3_models/`:

I) 1000 images are randomly generated, where each pixel is uniformly sampled from values between 0 and 1, using `tf.random.uniform()`. The size of the images is (16×16). The images are saved under `/statistic_test_3_models/input.csv`
II) Perform the following steps for each multiplier: `mul8s_1KV8` and `mul8s_1KV9`
III) Perform the following steps for each set of weights: 45, 50, 55, 60, 65, 70, 75, 80, and 85, pre-trained *exact epochs*, can be found under `/statistic_test_3_models/1KV8_weights` and `/statistic_test_3_models/1KV9_weights`
IV) Instantiate the three models:
    - The *exact model* from section 5.1 requires importing the weights
    - The *approximate model* from section 5.2 is an executable looking in a specific directory and the weights have to be copied into that directory
    - The *probabilistic model* from section 5.3 also needs the weights to be imported, however, it has to create error distributions based on the weights
V) The 1000 images are forward passed in the *exact model* and the *approximate model* and their predictions are exported to CSV files, which can be found in the following directories:
    - `/statistic_test_3_models/accurate_predictions/`
    - `/statistic_test_3_models/approximate_predictions/`
    - `/statistic_test_3_models/statistical_predictions/`
VI) The 1000 images are iteratively handled in the probabilistic model, where each image is copied 1000 times. Each ‘‘batch ’’ of 1000 identical images is forward passed in the *probabilistic model* and the predictions are exported to CSV files

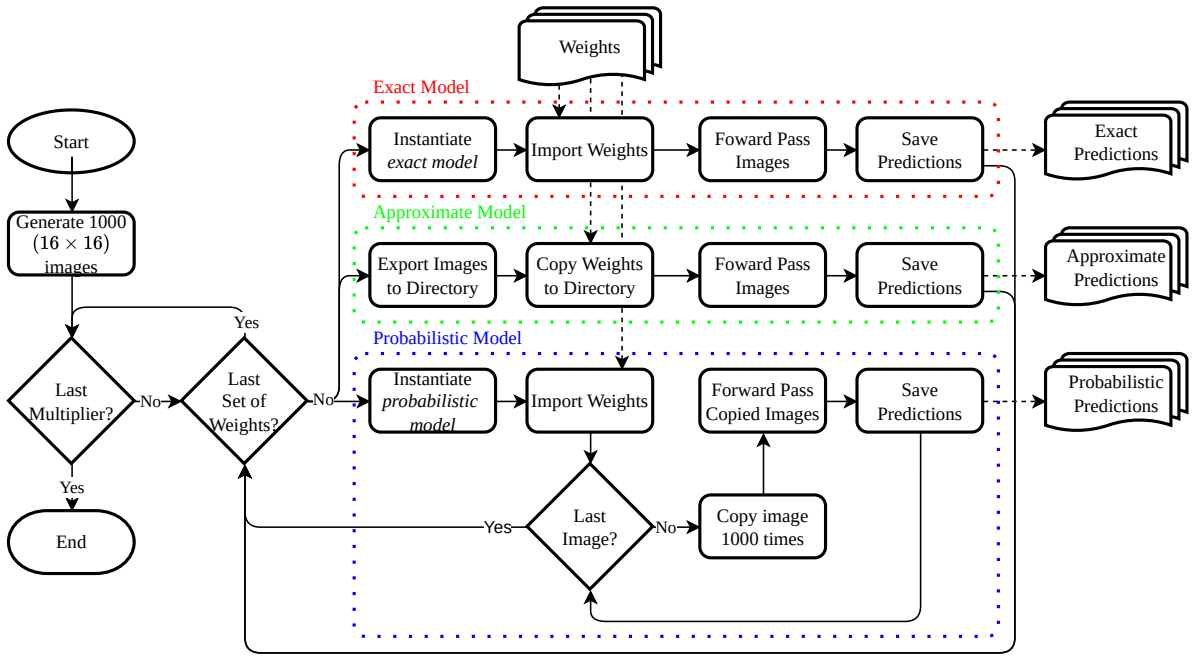

Figure E.2: Testing the probabilistic model flow. For each combination of approximate multiplier and set of pre-trained weights for that multiplier, the three models process the same input. The input consists of 1000 randomly generated images, where each pixel is sampled from a uniform distribution between 0 and 1. The predictions from each model are exported to CSV files for data processing. The outcome of the *probabilistic model* is probabilistic, why the same input is processed multiple times.

E.3 Results

In **test I** the CNN is trained using the *approximate model* for 45 epochs which will be used for reference against the inference *probabilistic model*. The accuracy of the *approximate model*, on both the training and test data set is seen plotted in Figure E.3.

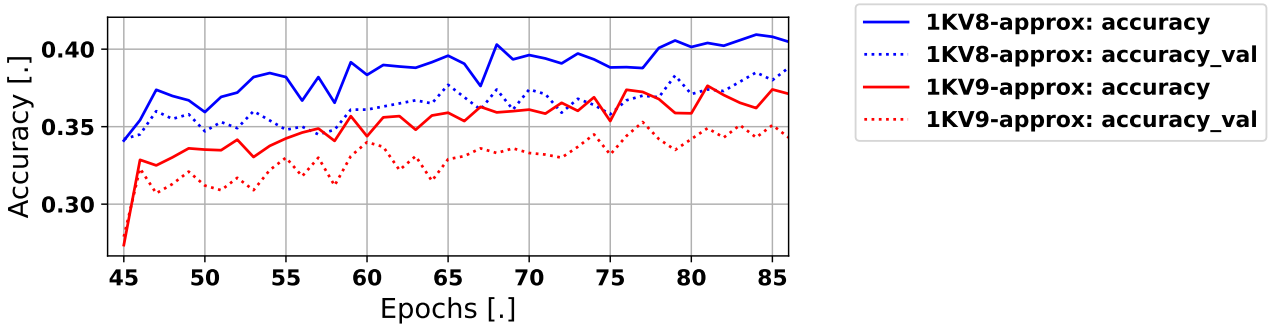

Figure E.3: The train and test accuracy for the approximate model using both the `mul8s_1KV8`and `mul8s_1KV9`. The evaluation is presented for the training using the STE.

These results are used as the basis for comparison for the evaluation of the *probabilistic model* using the weights trained using STE on the *approximate model*. The test is conducted using the test data for inference on both models and the results are plotted in Figure E.4.

From these results, it is noticed that the *probabilistic model* of the `mul8s_1KV8` follows to a considerable extent the accuracy of its deterministic equivalent. The case is different for the *probabilistic model* of `mul8s_1KV9` where the accuracy is considerably lower than its deterministic equivalent. However, it seems that the probabilistic model is ''*catching up*'' as the epochs progress for the STE.

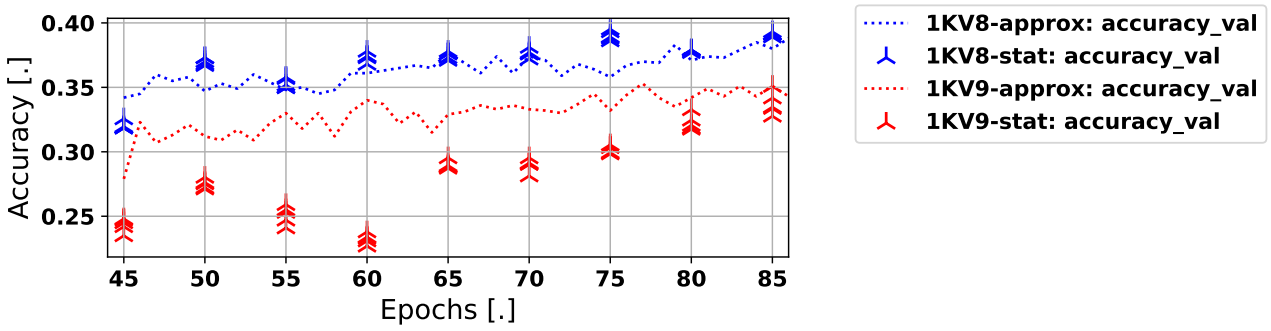

Figure E.4: Results of the accuracy evaluation on the *test* data set for the `mul8s_1KV8` and `mul8s_1KV9`. The deterministic *approximate model* is plotted as dotted lines and are congruent with the ones in Figure E.3. The accuracy of the *probabilistic model* is plotted using three-pointed stars for every 5 epochs.

An investigation of the relationship between the output vectors is necessary to give an appropriate explanation for this result.

Test II is conducted by applying uniformly distributed input pictures to the *exact*, *approximate* and *probabilistic models*. 1000 images are applied, once for the *exact* and *approximate* models and 1000 times for the probabilistic model. The reasoning behind this choice is that it is possible to get a quite accurate representation of the behaviour of all three systems, hence it is possible to perform a more detailed analysis on this basis.

E.3.1 mul8s_1KV8

As this experiment generates a large amount of data the analysis is first focused on the `mul8s_1KV8` for the inference stage, before STE training. The three samples are 10-variate random vectors and all RVs are assumed independent. To get an impression of the distribution of the output of the three models, these are plotted as histograms in Figure E.5

It is noticed that both the *approximate* and *probabilistic* models have lower means than the exact model, as the `mul8s_1KV8` introduces a negative bias to all multiplications in the network, hence an underestimate is to be expected.

As the model is developed to model the error of the *approximate model* the deterministic and probabilistic error distributions are obtained by subtracting the output vectors from the *exact model*, from both the *approximate* and *probabilistic* outputs. The distribution for each of the indices in the error vectors is plotted in Figure E.6.

It is noticed that, by visual inspection, the probabilistic errors are close to normally distributed, which makes sense as the individual samples of error distribution (i.e. the 1000 pr. image) are all close to normally distributed, due to the CLT as described in section 4.3. The deterministic errors also seem normally distributed, however clearly with a different variance than the probabilistic distribution. The means are rather similar for some of the variables in the output vector. Pondering on the purpose of the *probabilistic model*, it is known that this should model the deterministic errors such that ideally this distribution could be mistaken for a sample of the probabilistic error.

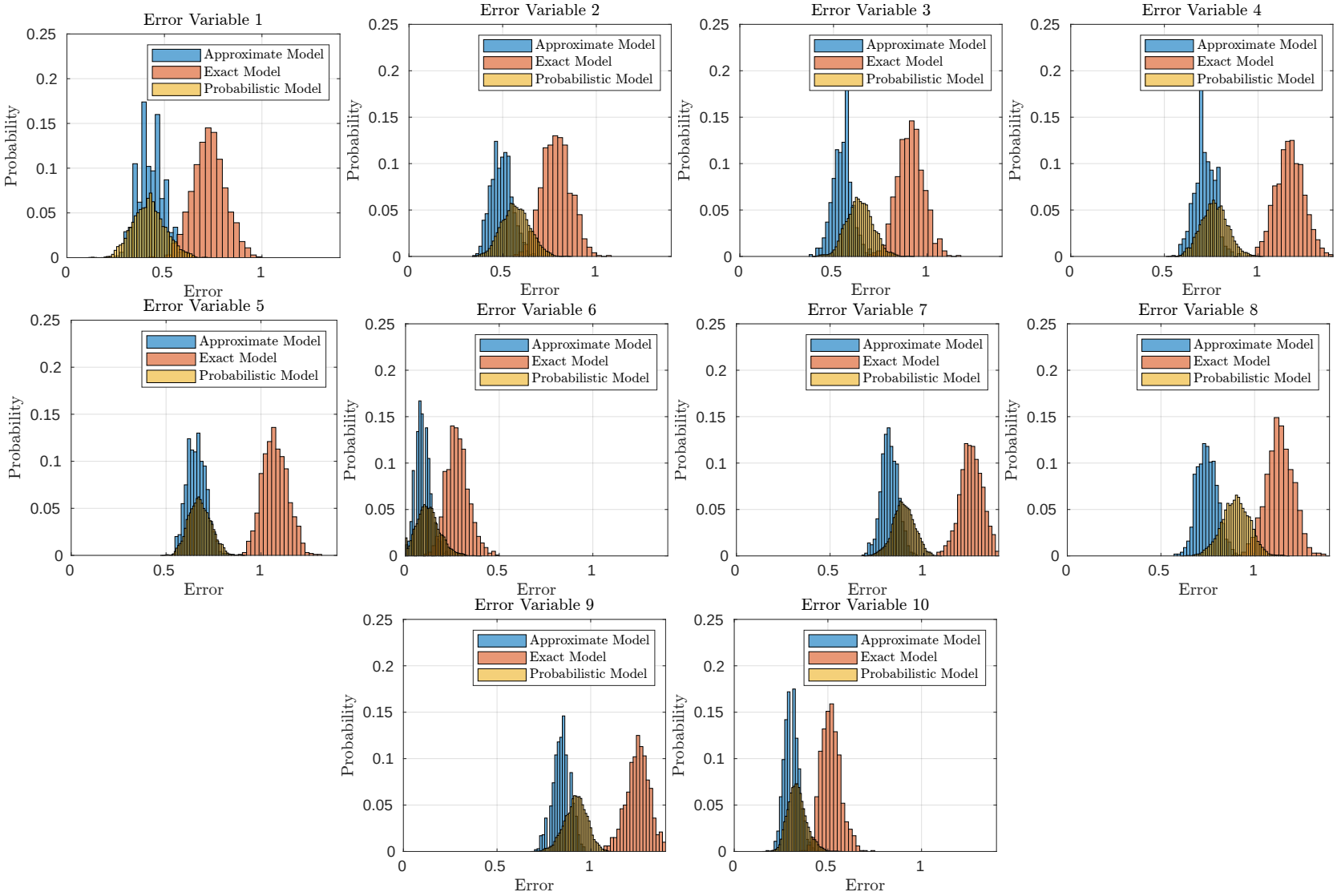

Figure E.5: Histogram of the output of the *approximate, exact* and *probabilistic* models, given 1000 input images. The deterministic histograms are partitioned into 20 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the output vectors.

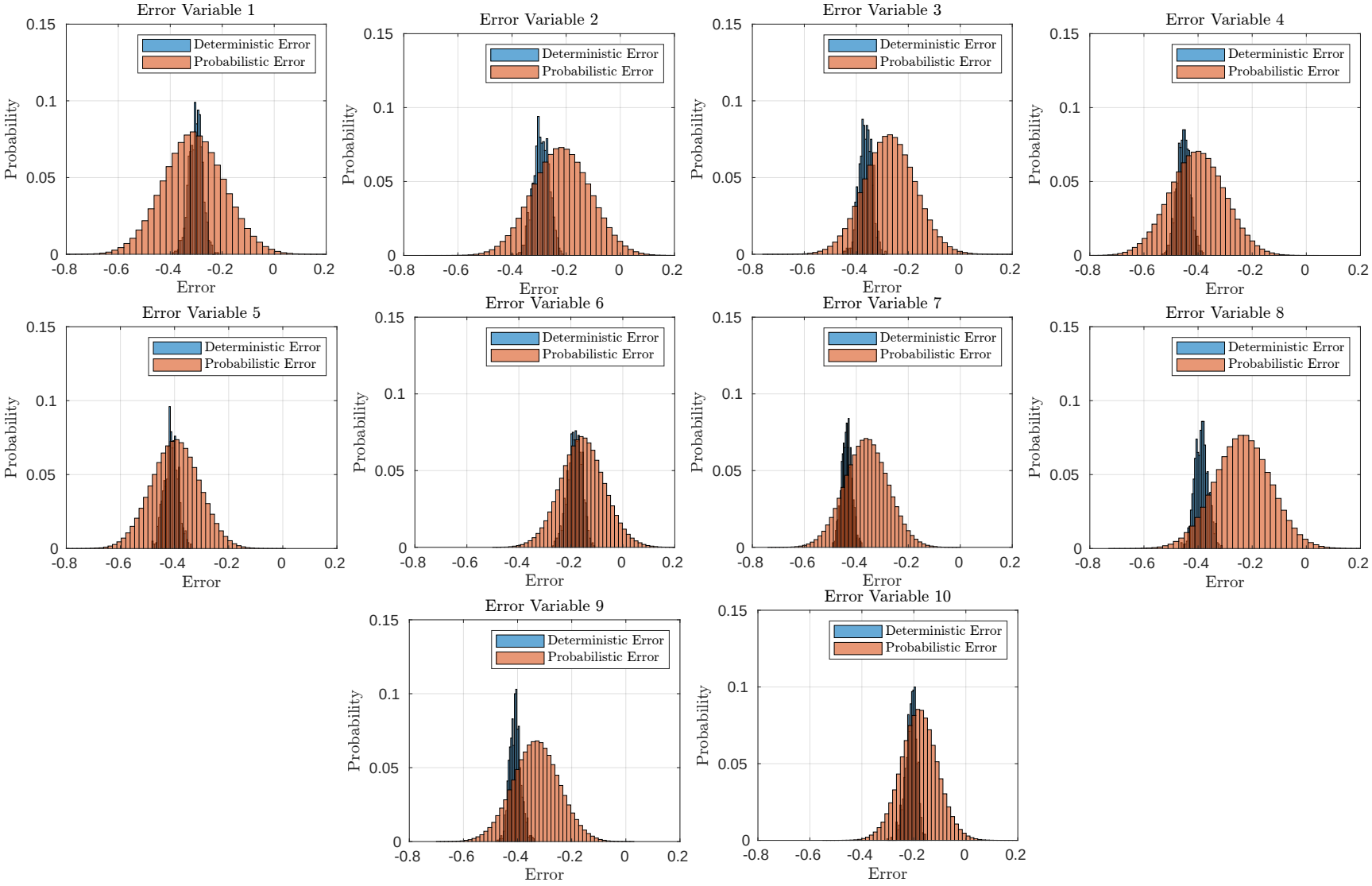

Figure E.6: Histogram of the deterministic and probabilistic error distributions, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

The Kullback-Liebler divergence is a measure of the ''*distance*'' or ''*divergence* of two distributions, based on how difficult it is to tell samples of the two distributions apart [124]. Given two distributions, P and Q the KL-divergence is denoted as $D_{KL}(P||Q)$. P is here interpreted as an observation of a probability distribution and Q is seen as a model of the distribution P. The KL-divergence is then intuitively construed as a measure of the information lost when using Q as a model for P. This intuition can directly be applied to the problem at hand, where the *probabilistic model* attempts to model the *approximate* outputs. The KL-divergence has a neat formula for when both distributions are multivariate Gaussian. This is presented without a proof (see Soch et. al. [125]) in Eq. (E.1).

$$D(P||Q) = \frac{1}{2}\left((\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1) + \text{tr}(\Sigma_2^{-1}\Sigma_1) - n - \ln\left(\frac{|\Sigma_1|}{|\Sigma_2|}\right)\right) \tag{E.1}$$

where:

P $\sim \mathcal{N}(\mu_1, \Sigma_1)$
Q $\sim \mathcal{N}(\mu_2, \Sigma_2)$
n Dimension of output vector

The KL-divergence is a simple scalar measure, which can be used to quickly assess the magnitude of the loss of information. However, the cause of this magnitude is unclear by just evaluating Eq. (E.1). The magnitude of the addends within the formula reveals some of the causes for the total magnitude.

The first term:

$$(\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)$$

is the squared Mahalanobis distance between the mean vectors μ_1 and μ_2, with Σ_2 as the covariance matrix. The Mahalanobis distance is a measure between a point and a probability distribution, the use of the inverse covariance matrix accounts for correlations between the variables in the distribution, hence this is a beneficial measure for multivariate probabilistic data set [132]. The squared Mahalanobis distance is used in the KL-divergence as a measure from the mean of P to the distribution of Q. This term is therefore interpreted as a measure of the differences in means.

The second term:

$$\text{tr}(\Sigma_2^{-1}\Sigma_1) - n$$

is simple to show that if the covariance matrices are equal their product will be the identity matrix of size nxn, hence the trace subtracted by n will be zero. The second term is therefore interpreted as a measure of differences in covariance.

The third term:

$$\ln\left(\frac{|\Sigma_1|}{|\Sigma_2|}\right)$$

is the logarithm of the difference in determinants are used to compensate for differences in the scaling of the distributions. This ratio will also be zero if the distributions are equal.

Since this KL-divergence assumes Gaussian distributions, it is decided to measure the error distributions, since these seem to be more consistently Gaussianly distribution. The KL-divergence between the deterministic and probabilistic models is found for the `mul8s_1KV8` where the CNN is not trained using STE (i.e. inference on the CNN trained on the exact model). This is shown in Table E.4

Table E.4: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
14.5	14	−8.9	23.9

These measures by themselves show that the *probabilistic model* is not an exact match of the observed sample. However, this is to be expected from the plots shown in Figure E.6. Furthermore, it is seen that the difference in means (measured by the Mahalanobis distance) seems to contribute significantly to the divergence compared to the difference in covariances, although the largest contributor to the overall divergence. These observations coincide with the histograms in Figure E.6. It is emphasised that the KL measure by itself does not provide much useable information, as the measures should be used for comparisons of methods. For this reason, the same analysis is applied to the remaining inference stage for each additional 5 epochs of STE training.

45/5 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.7.

The KL-divergence is calculated and the results are shown in Table E.5.

Table E.5: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 5 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
14.5	12.2	−8.8	25.7

The KL-divergence is the same as the one where the CNN did not consist of weights trained with STE. However, the contribution from the Mahalanobis distance decreased, at the expense of a larger contribution from the scaling term.

45/10 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.8.

It is noticed that the sixth variable has lost its Gaussian shape, and the probabilistic model seems to fit the deterministic better than previously. This result is due to a shift in the weights during training resulting in a negative value on this output for some inputs. The nonlinear ReLU function is equating this to zero.

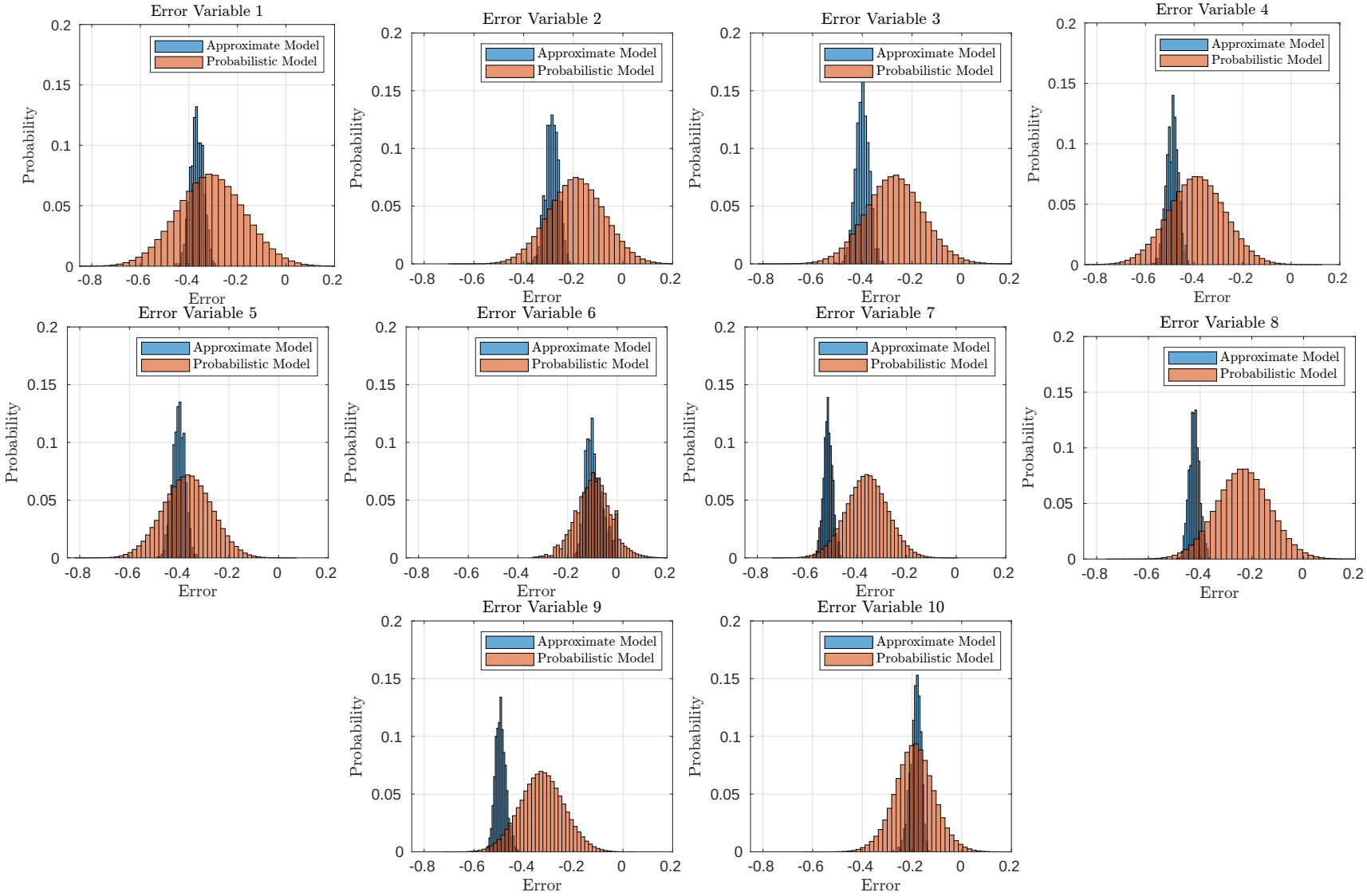

Figure E.7: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

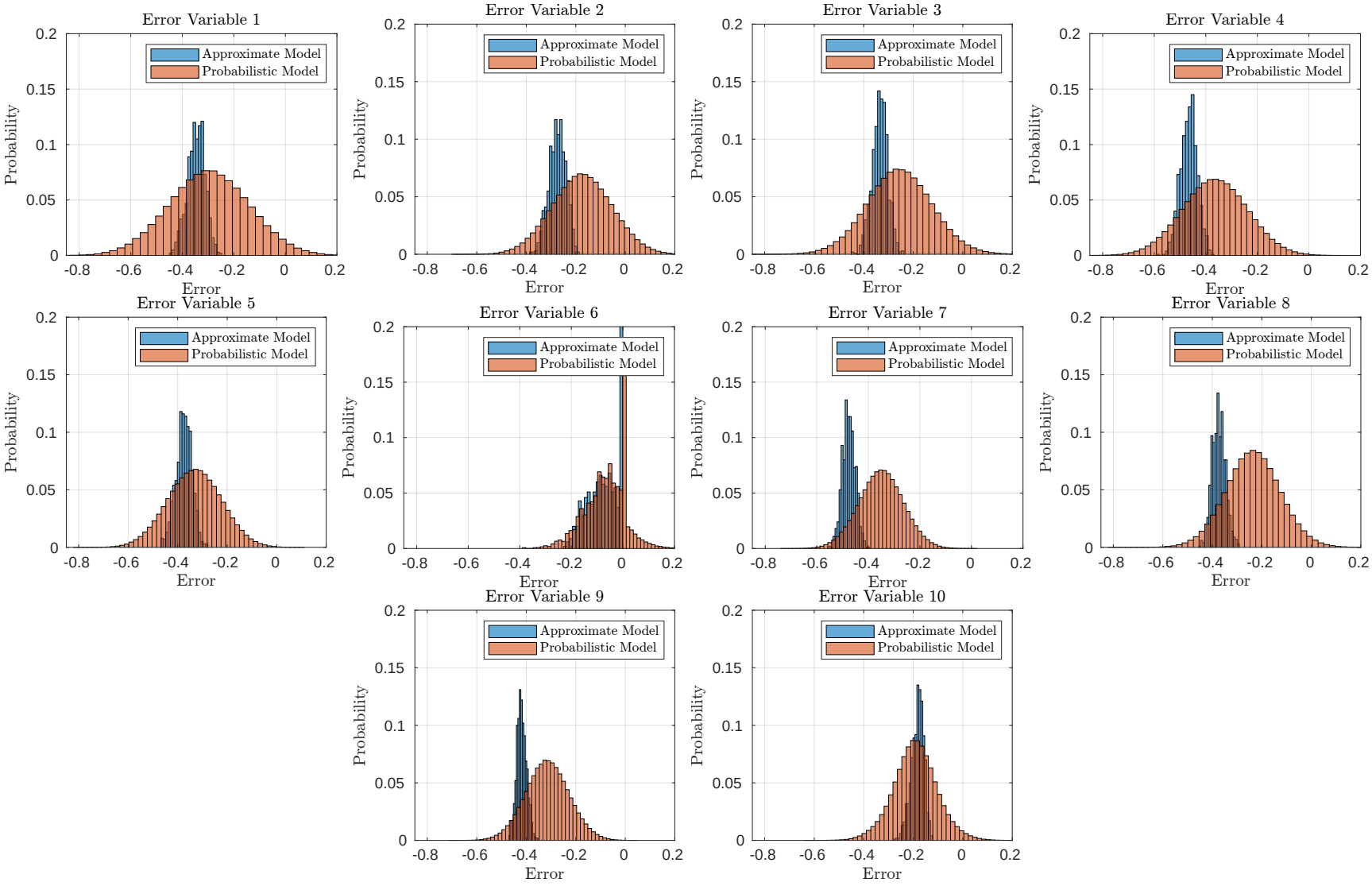

Figure E.8: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

The KL-divergence is calculated and the results are shown in Table E.6.

Table E.6: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 10 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
10.5	5.4	−8.3	28.8

The KL-divergence reduces from the previous measure, primarily because of a reduction in the Mahalanobis distance and thereby the difference in means. It is suspected that this is due to the development regarding the deactivation of the 6^{th}.

45/15 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.9.

The development of the deactivation of the 6^{th} output persists, and even more samples seem to have fallen under the effect of ReLU.

The KL-divergence is calculated and the results are shown in Table E.7.

Table E.7: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 15 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
12.7	10.6	−8.4	23.2

45/20 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.10.

The development of the deactivation of the 6^{th} output persists, and even more samples seem to have fallen under the effect of ReLU.

The KL-divergence is calculated and the results are shown in Table E.8.

Table E.8: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 20 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
13.7	11.9	−8.8	24.5

45/25 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.11.

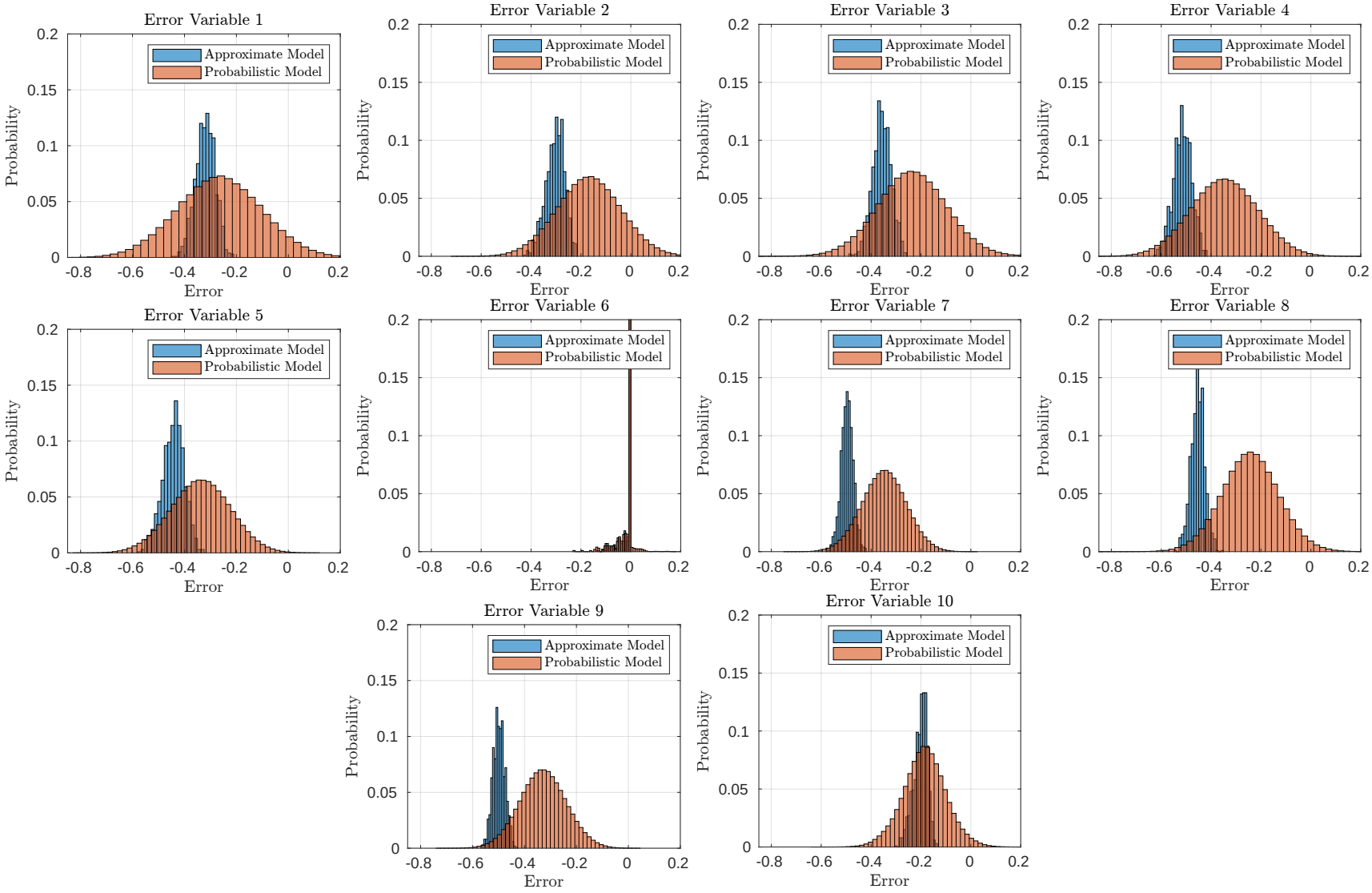

Figure E.9: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

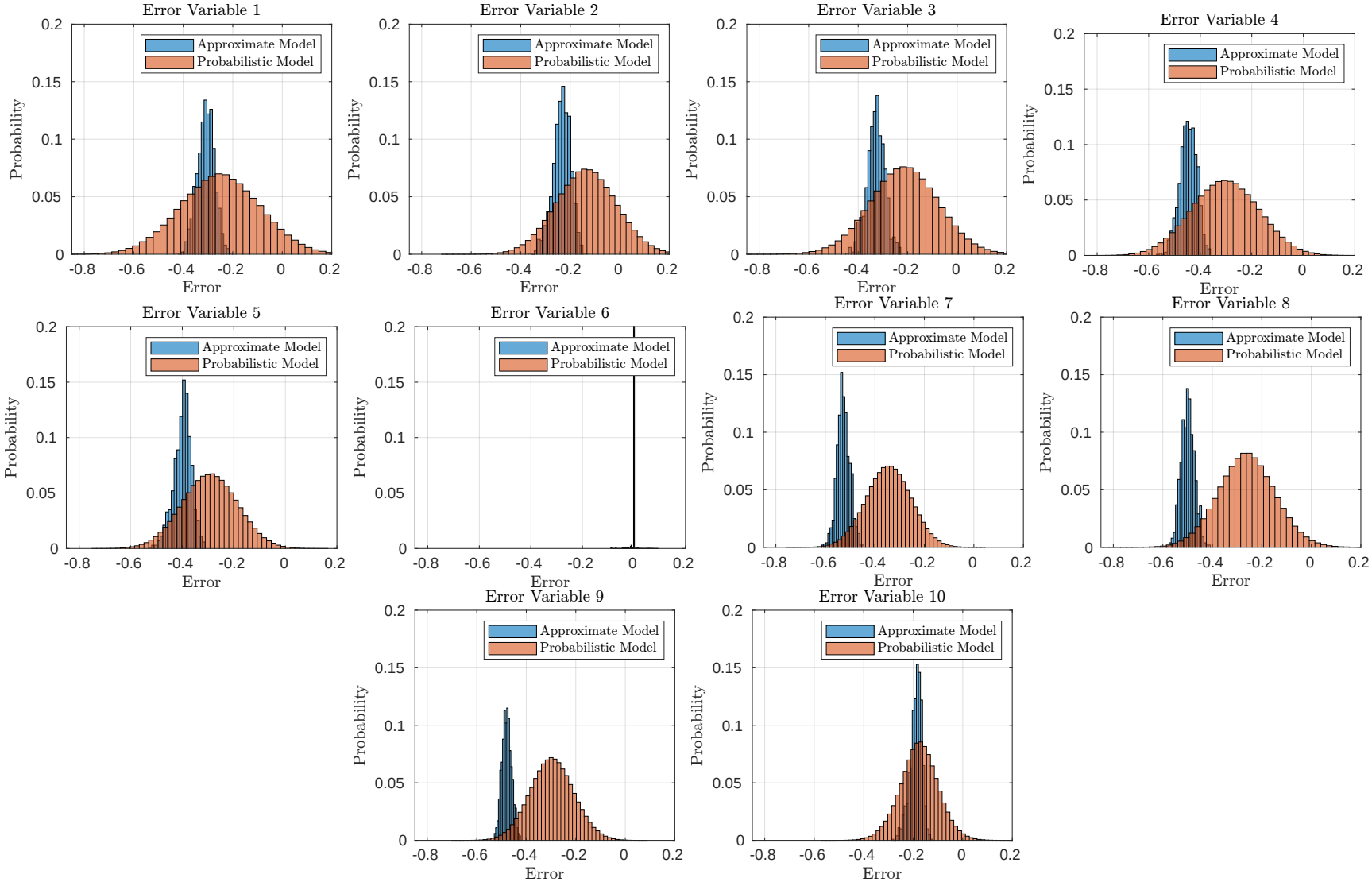

Figure E.10: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

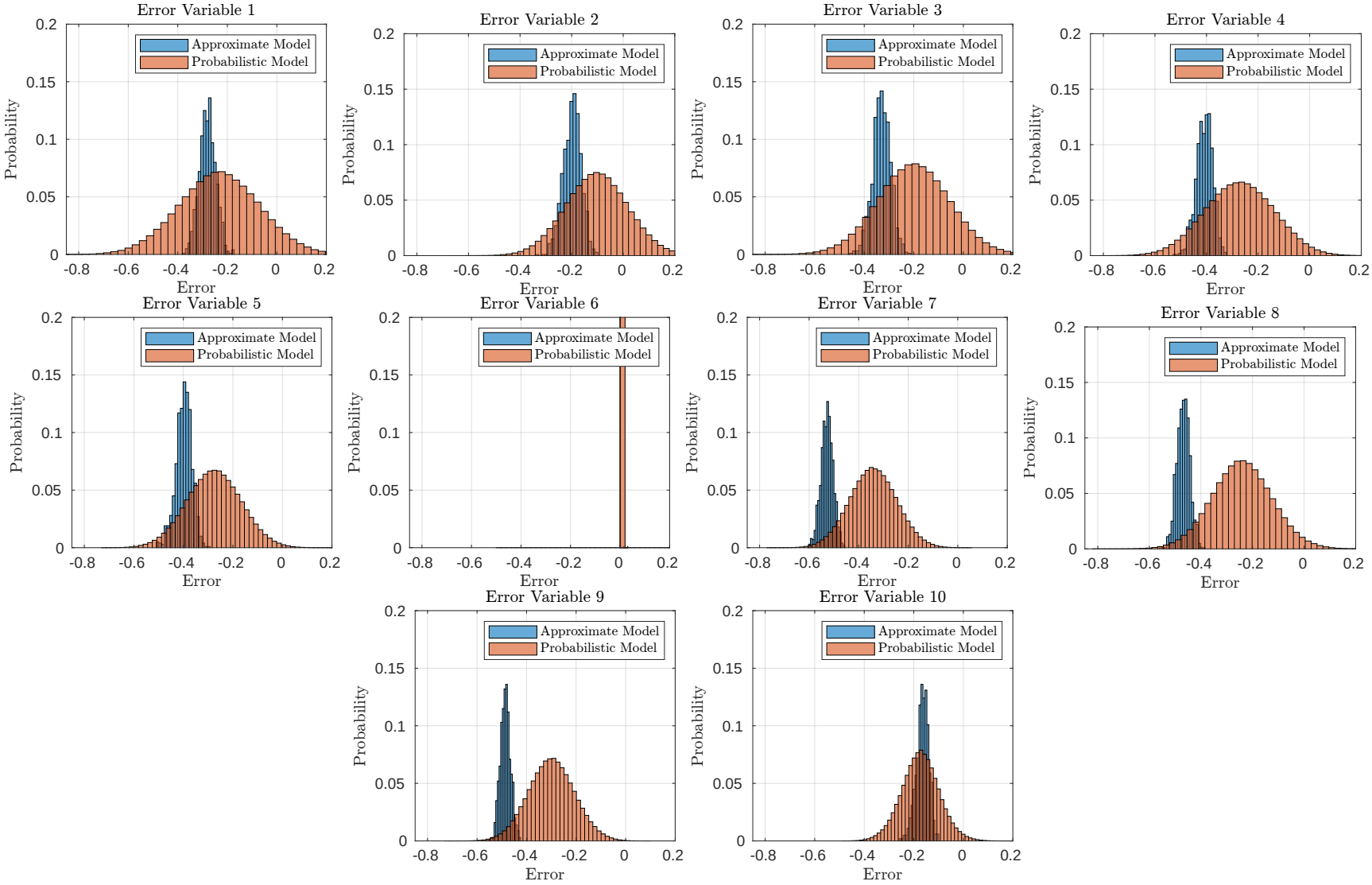

Figure E.11: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

The development of the deactivation of the 6^{th} output persists, and all samples seem to have fallen under the effect of ReLU.

The KL-divergence is calculated and the results are shown in Table E.9.

Table E.9: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 15 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
13.9	10.6	−7.9	22.4

45/30 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.12.

The KL-divergence is calculated and the results are shown in Table E.10.

Table E.10: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 30 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
15	16.1	−8	21.9

The KL-divergence is increasing and is at the highest value yet. This is mainly due to a significant rise in the Mahalanobis term, which is supported by the development in the histograms.

45/35 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.13.

The KL-divergence is calculated and the results are shown in Table E.11.

Table E.11: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 15 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
14.8	14.7	−8.1	23.1

45/40 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.14.

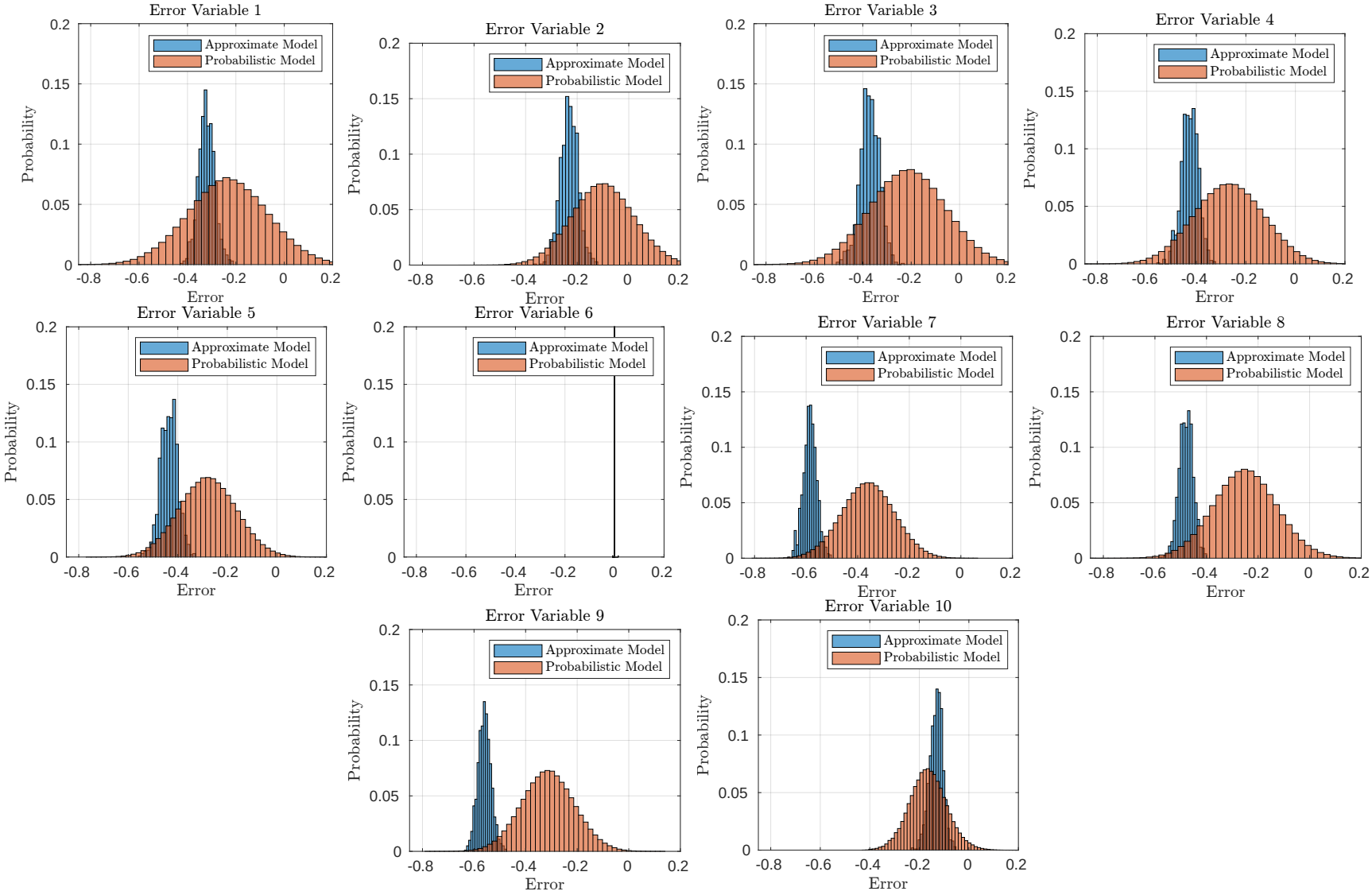

Figure E.12: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

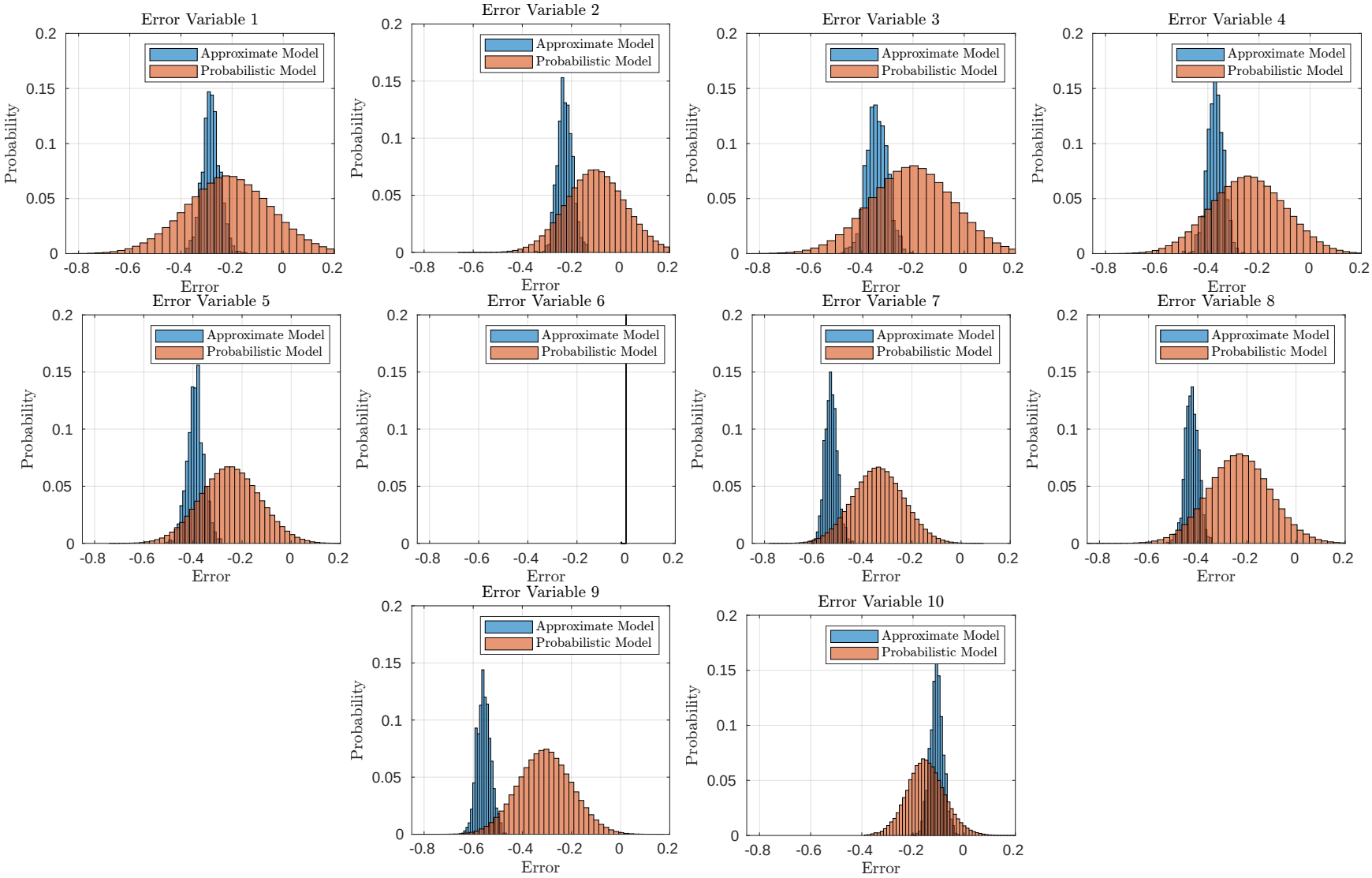

Figure E.13: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

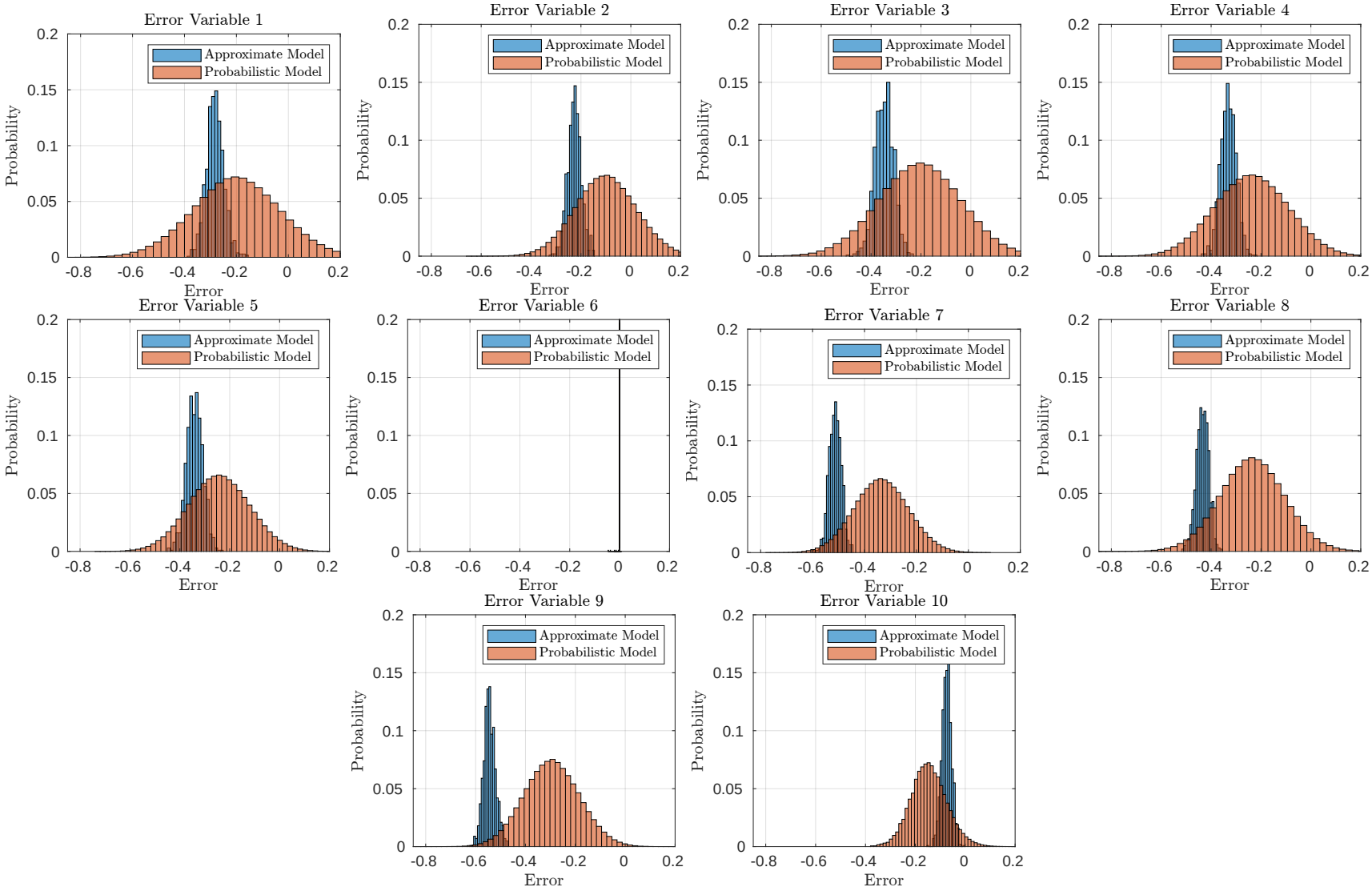

Figure E.14: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV8`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

The KL-divergence is calculated and the results are shown in Table E.12.

Table E.12: The KL-divergence of the `mul8s_1KV8`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 40 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
16.9	16.1	−8.5	25.4

The KL-divergence is increasing and is at the highest value yet. This has to do with both an increase in Mahalanobis and Scaling terms. The reason for this might be the training introduced to the approximate model using STE. Through optimisation it is possible that the network can learn some of the distinct deterministic errors, which do not apply to the probabilistic model, which would coincide with the observation of the 10^{th} output node in Figure E.14.

E.3.2 mul8s_1KV9

In the same manner as the `mul8s_1KV8` the `mul8s_1KV9` is investigated using the KL-divergence measure. From the results presented in Figure E.4 it is expected to see a generally higher KL-divergence than for the `mul8s_1KV8` however, it is hypothesised that the measure will decrease with the training on the *approximate model* as a convergence is seen in the classification accuracy.

45/0 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.15.

From these histograms, it is noticed that the negative magnitude of the errors is larger compared to the measurements performed using the `mul8s_1KV8`. The divergence between the two distributions also seems larger and is also calculated and the results are shown in Table E.13.

Table E.13: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 0 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
37	60.2	−8.5	22.3

The KL-divergence is significantly larger for the `mul8s_1KV9` than observed for the `mul8s_1KV8`, mainly due to a large increase in the Mahalanobis term (i.e. the difference in means). This is congruent with the histograms shown in Figure E.15.

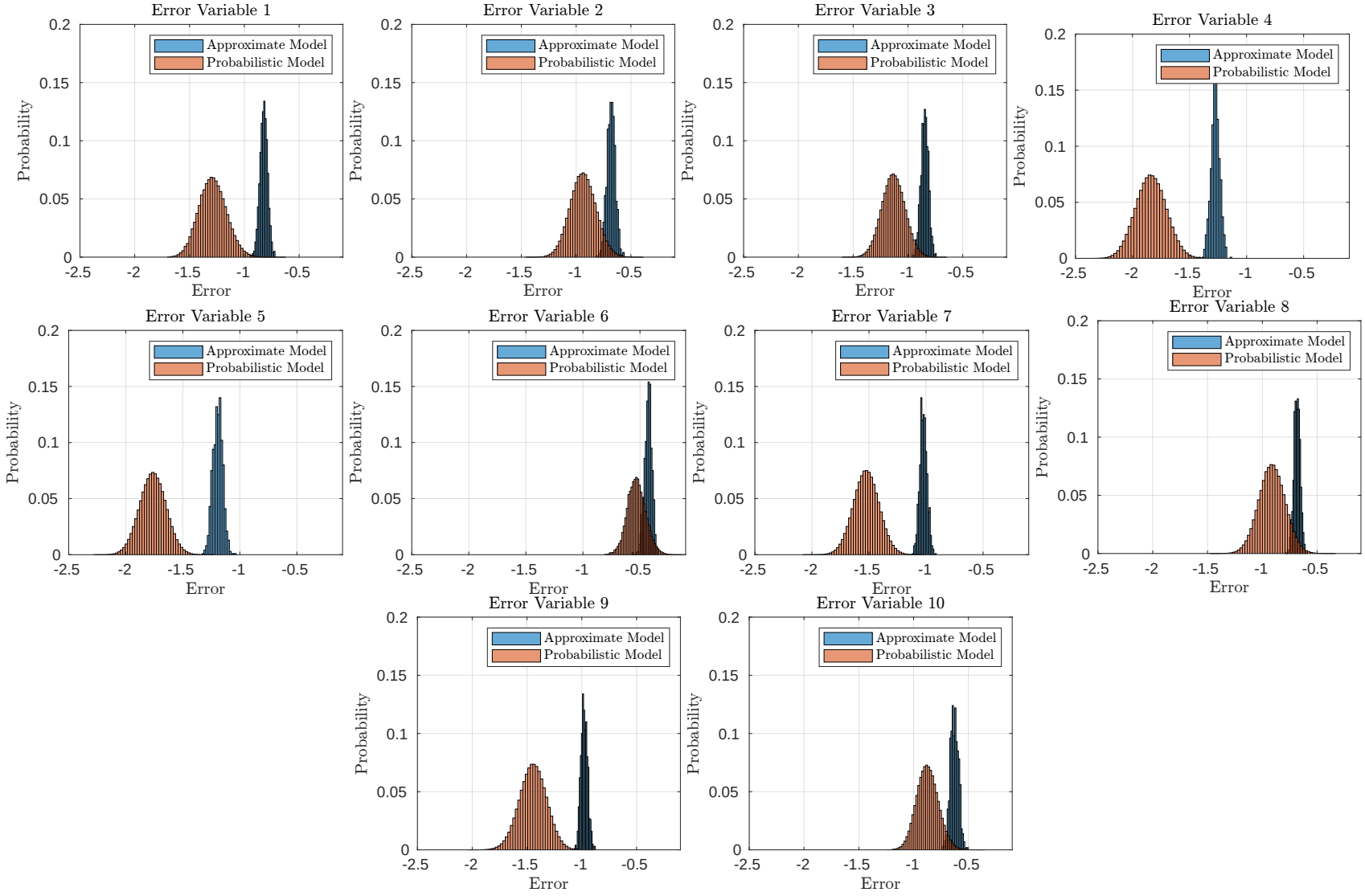

Figure E.15: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

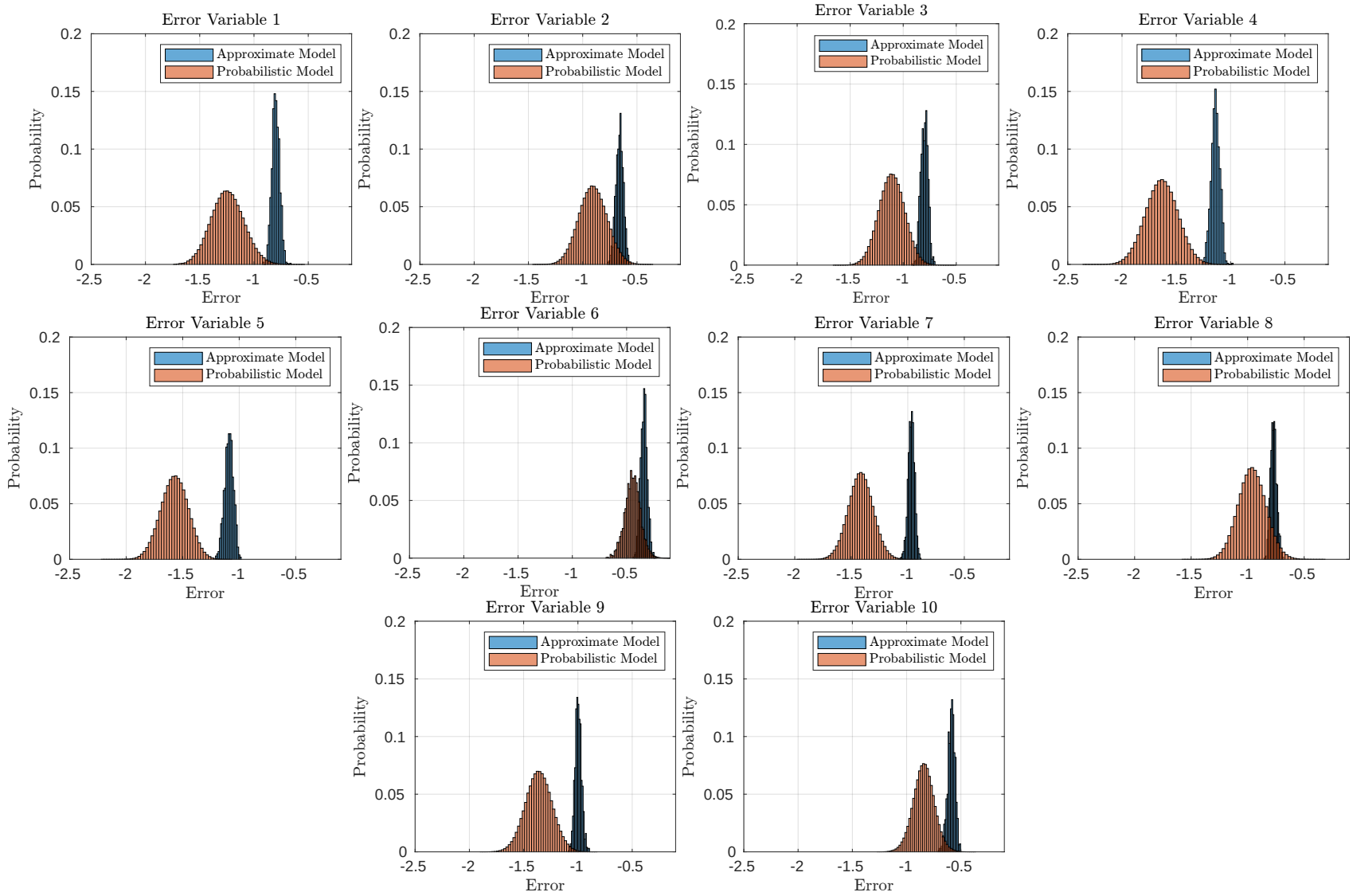

Figure E.16: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

45/5 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.16. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.14.

Table E.14: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 5 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
29.9	44.8	−8.8	23.8

The KL-divergence is reduced from the previous measurement when no approximate training was applied before inference, mainly due to a decrease in the Mahalanobis term (i.e. the difference in means).

45/10 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.17. The deactivation of the 6^{th} output node, is once more an issue, making the distributions very similar and are not normally distributed. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.15.

Table E.15: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 10 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
23.6	33.8	−8.4	21.8

Once again a decrease in the KL-divergence is observed. It is suspected that this is due to the 6^{th} output node partial deactivation.

45/15 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.18. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.16.

Table E.16: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 15 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
17.7	22.6	−7.5	20.3

The KL-divergence is still decreasing, possibly due to the deactivation of the 6^{th} output node.

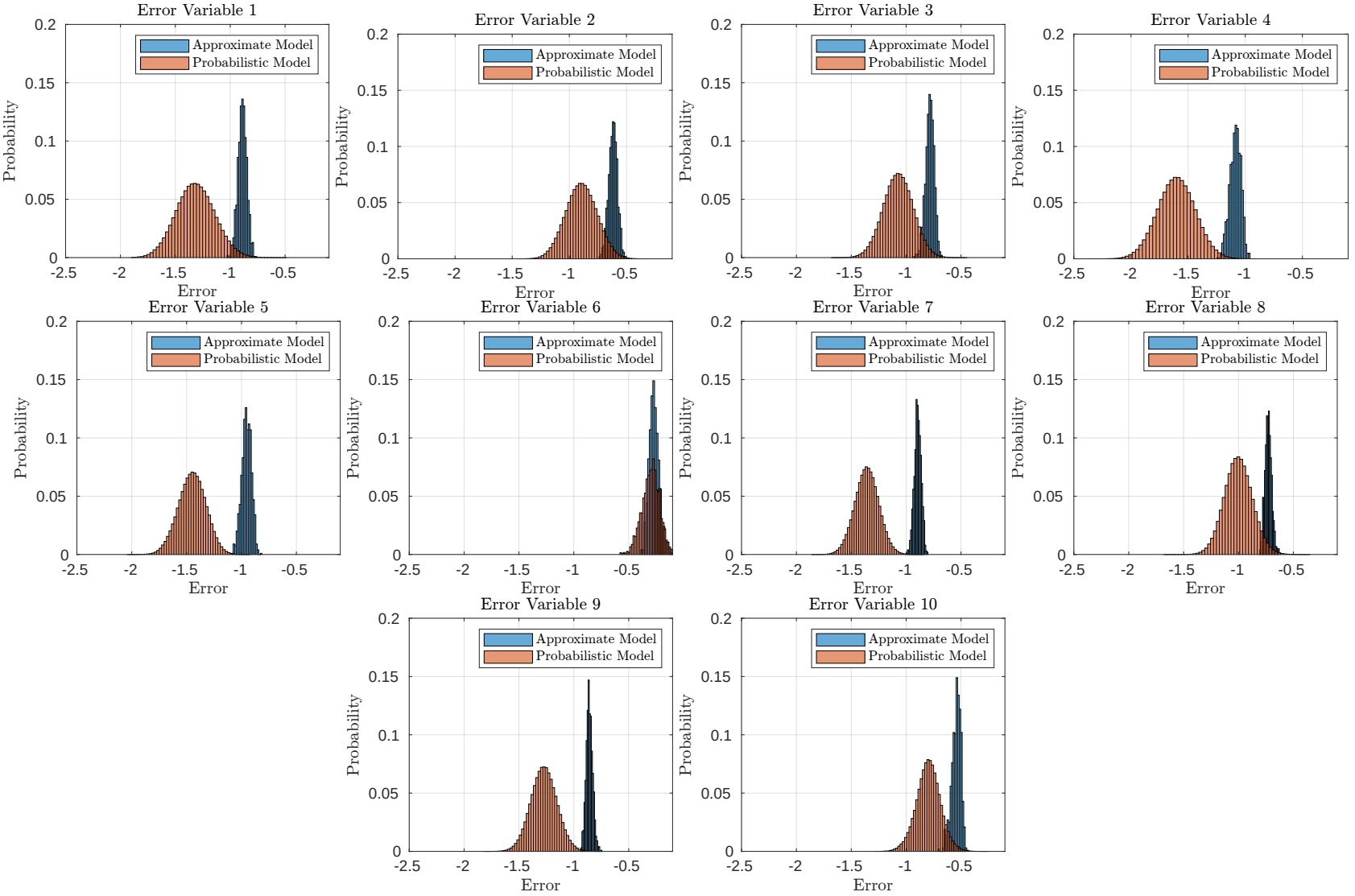

Figure E.17: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

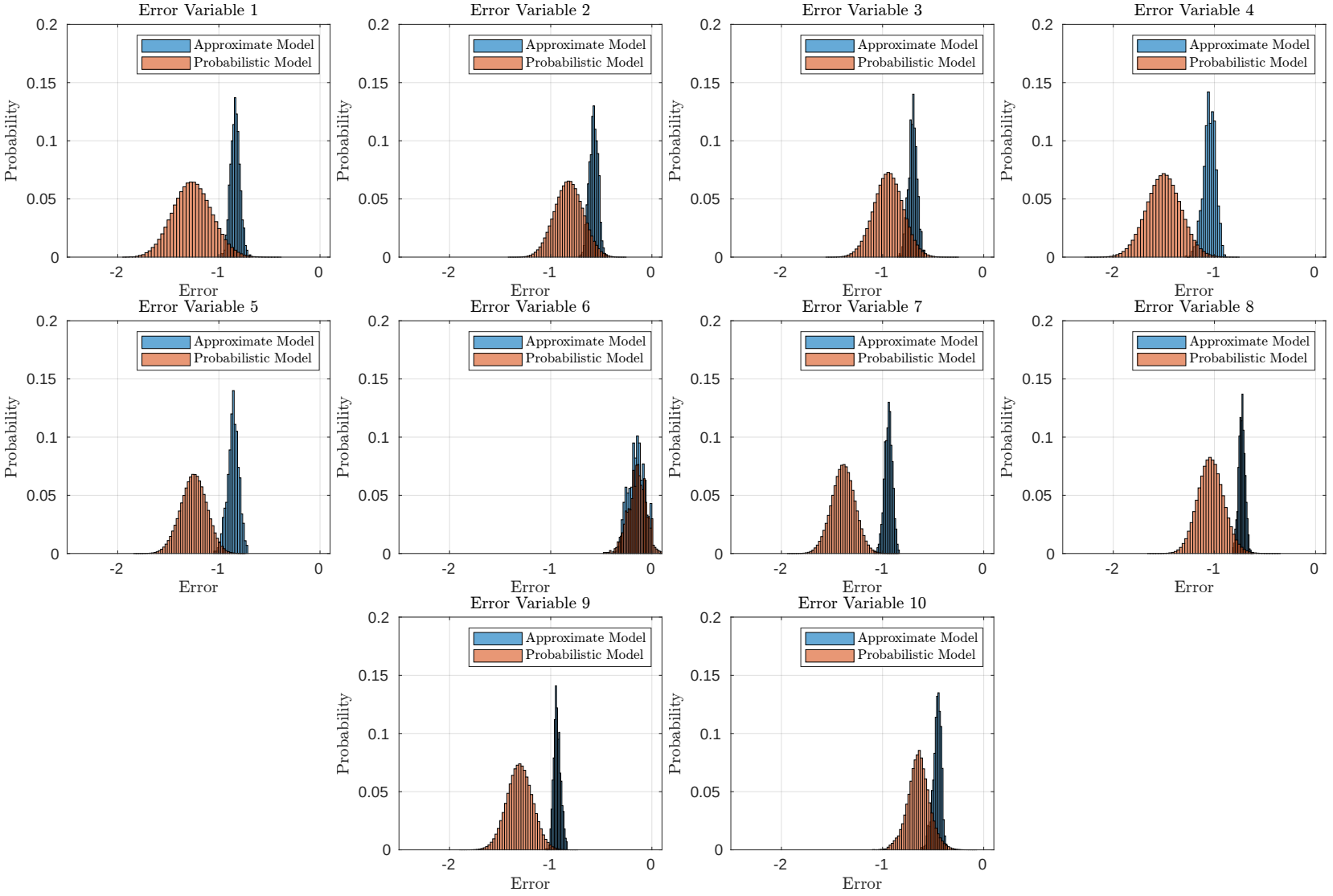

Figure E.18: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

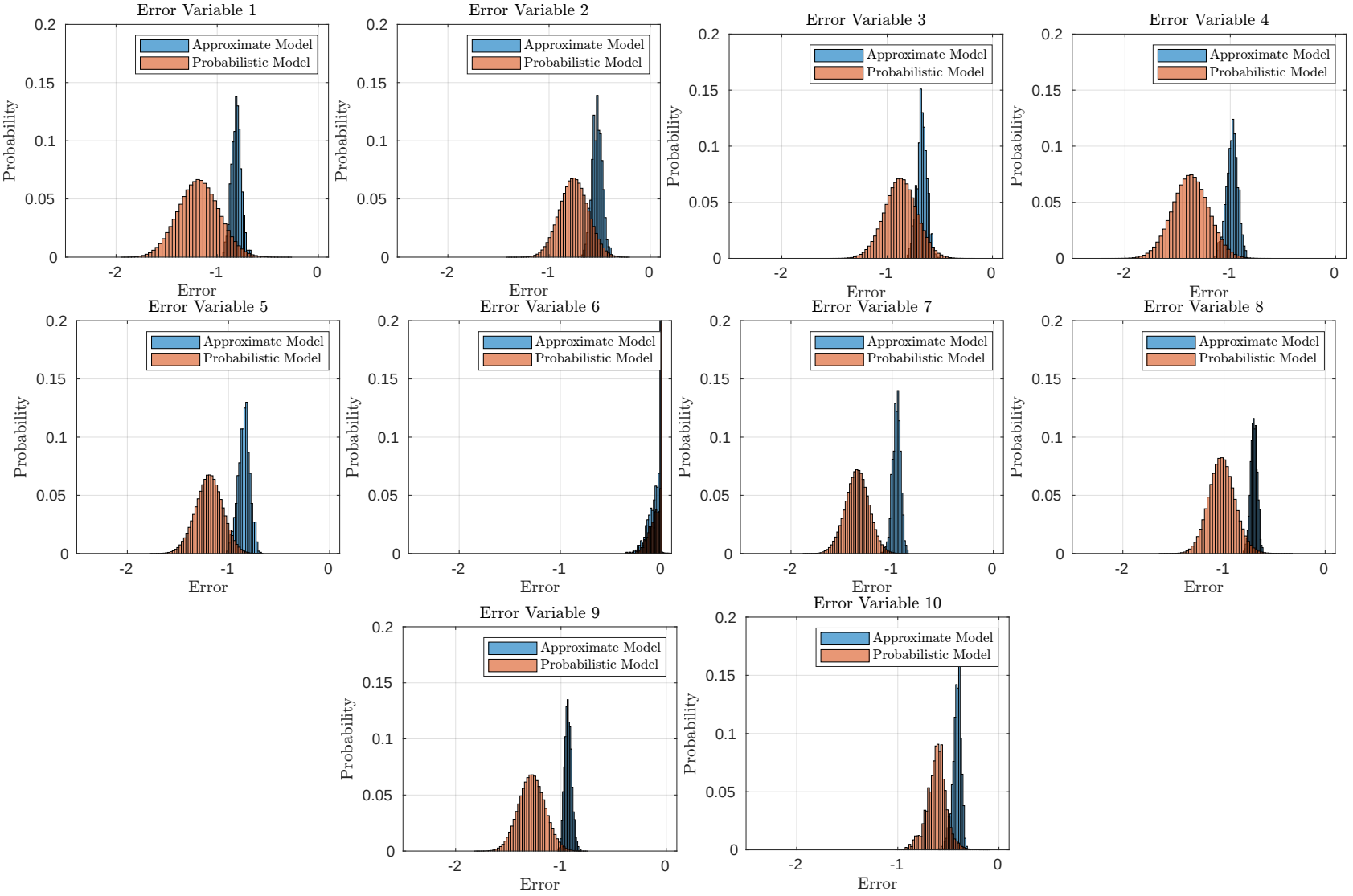

Figure E.19: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

45/20 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.19. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.17.

Table E.17: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 20 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
15.2	17.2	−7.2	19.85

45/25 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.20. The deactivation of the 6^{th} output node, is once more an issue, making the distributions very similar and are not normally distributed. Furthermore, this development seems to be happening for the 10^{th} output node as well. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.18.

Table E.18: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 25 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
14.6	17	−7.3	19.5

45/30 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.21. The KL-divergence between the two distributions is also calculated and the results are shown in Table E.19.

Table E.19: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 30 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
14.1	15.7	−7.4	19.9

The KL-divergence is now similar to the one observed for `mul8s_1KV8`. The main contributor is now the scaling term which was also the case for the `mul8s_1KV8`. It is noticed that the mean of the probabilistic model has a lower mean than the deterministic model for the `mul8s_1KV9`. This is the opposite of what was the case for `mul8s_1KV8`.

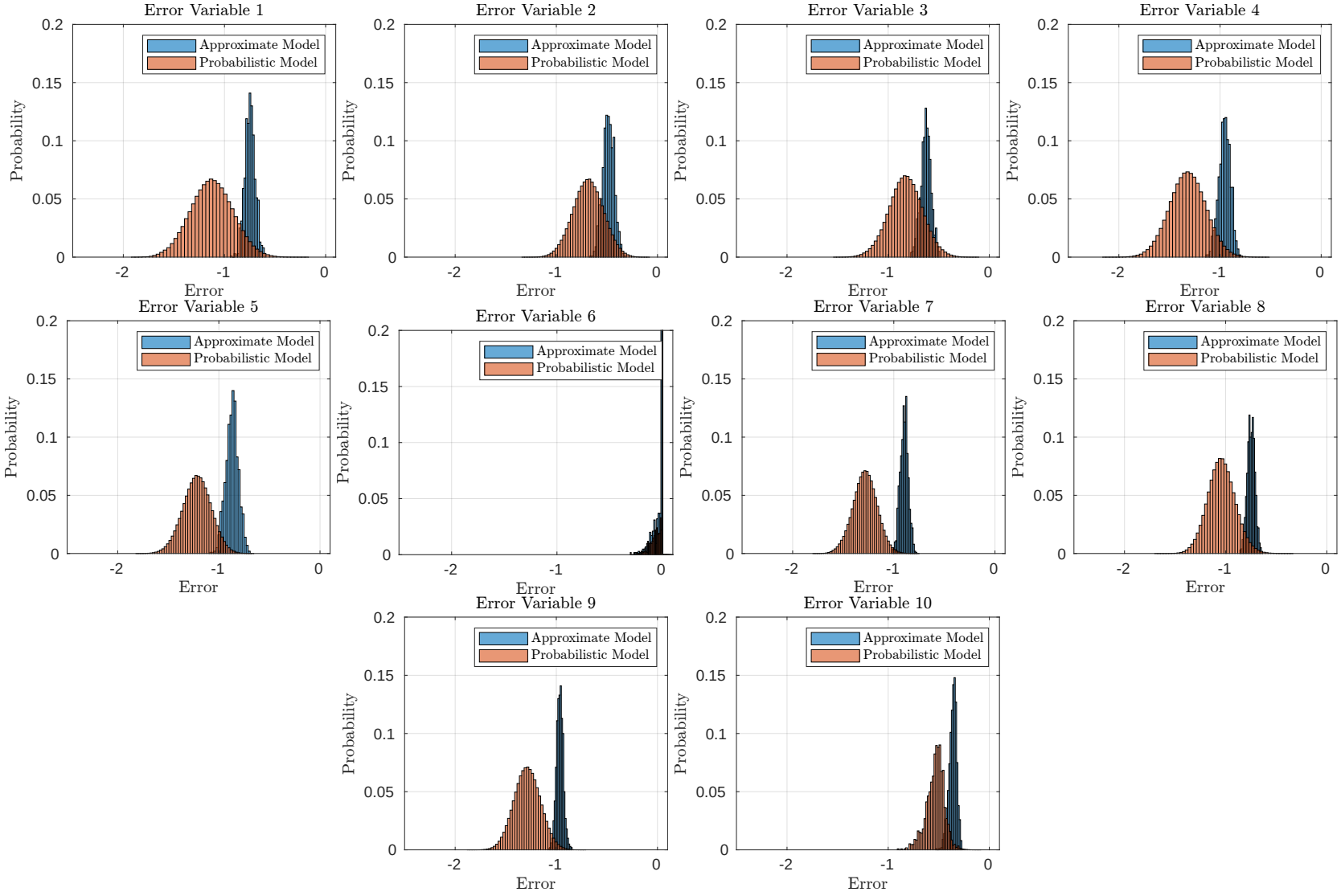

Figure E.20: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

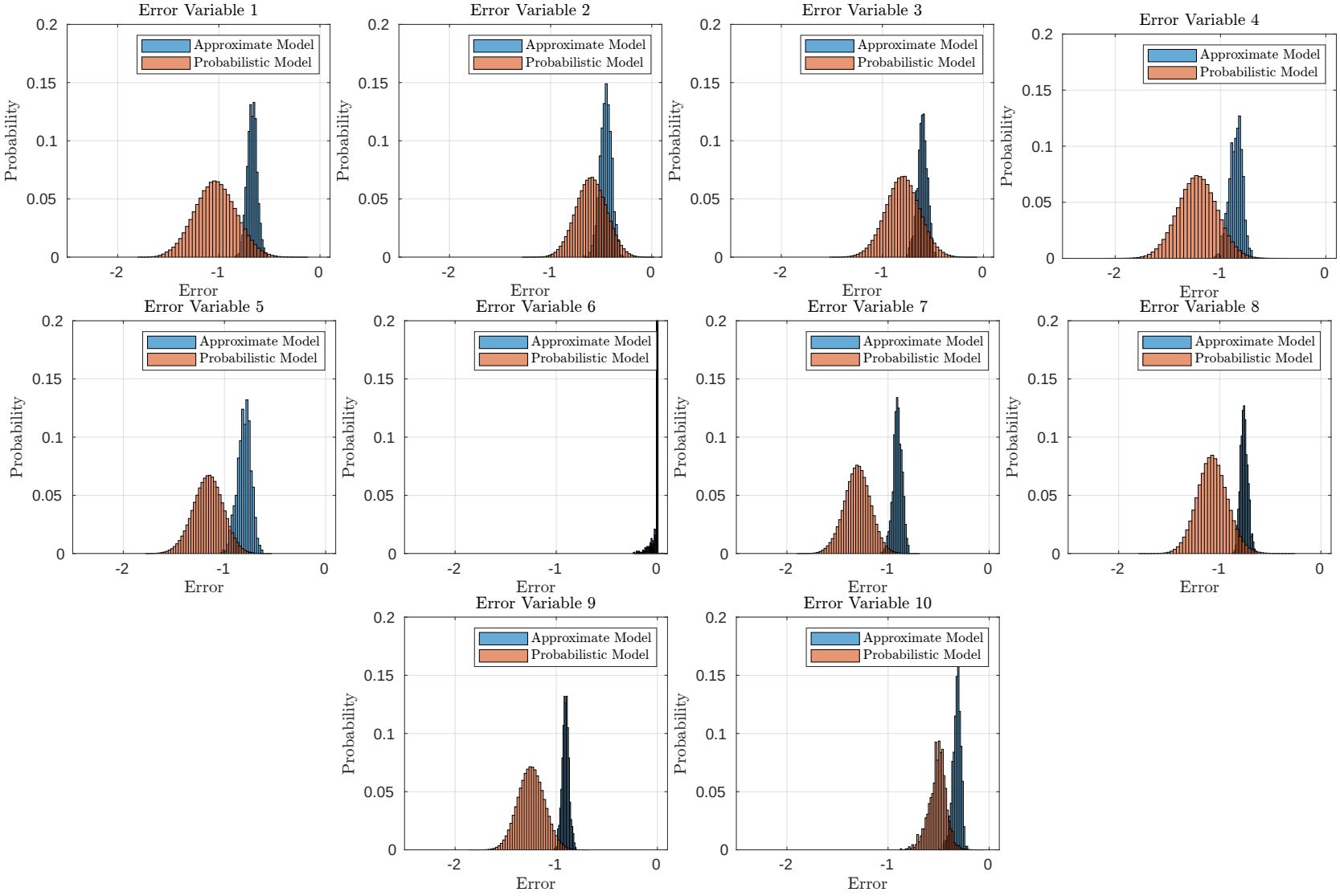

Figure E.21: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

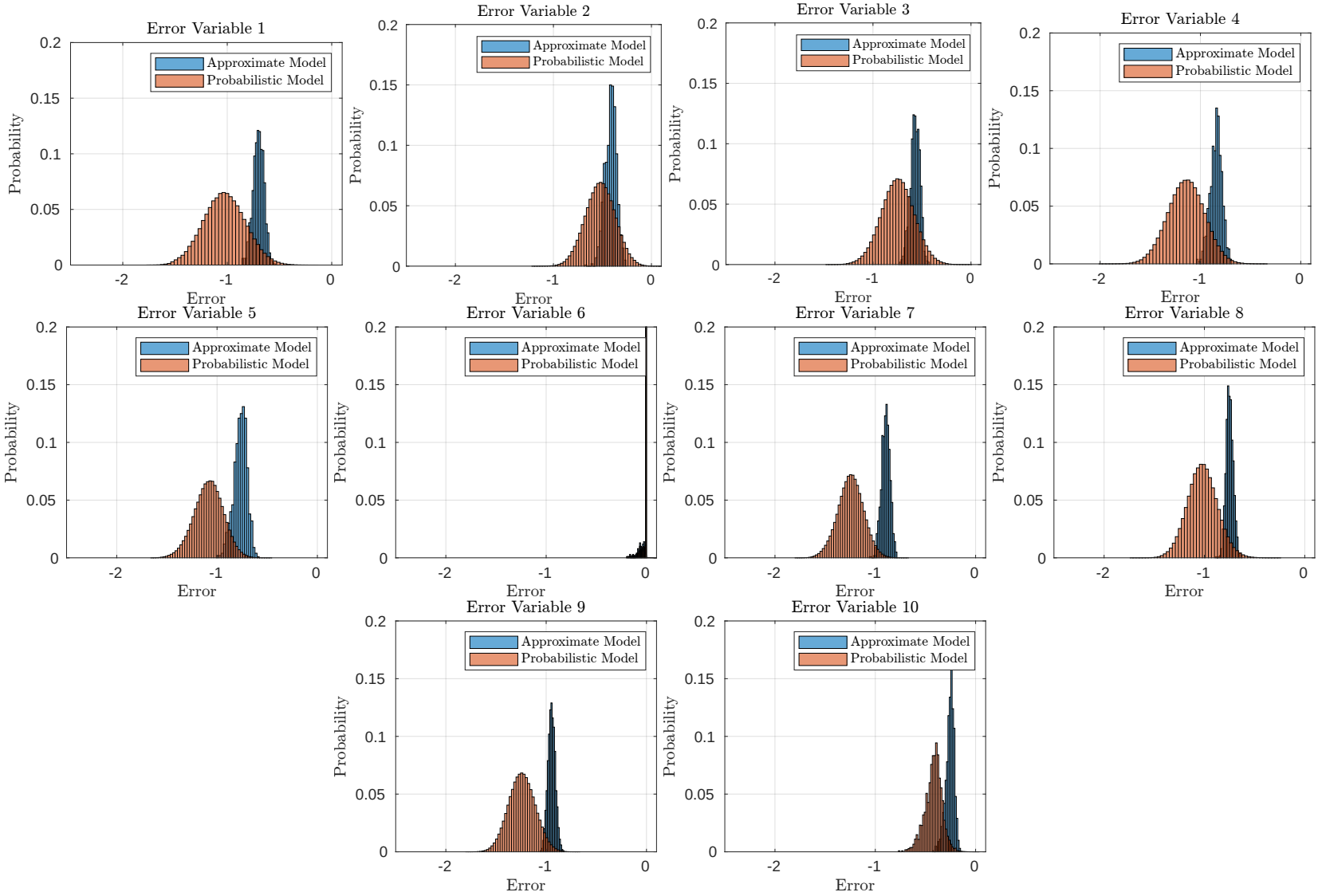

Figure E.22: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

45/35 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.22.

The KL-divergence between the two distributions is also calculated and the results are shown in Table E.20.

Table E.20: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 35 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
12.5	12.9	−7.5	19.6

45/40 is the ratio between exact and STE training epochs. The deterministic and probabilistic error distributions are plotted in Figure E.23.

The KL-divergence between the two distributions is also calculated and the results are shown in Table E.21.

Table E.21: The KL-divergence of the `mul8s_1KV9`, for inference on a CNN trained for 45 epochs trained using exact arithmetic and 40 epochs using STE on the *approximate model*.

KL-divergence	Mahalanobis term	Covariance term	Scaling term
13.3	14.2	−7.6	20

The KL-divergence increases slightly from the previous measurement. This could indicate that the modelling strategy limits the accuracy of the estimate. For all measurements, the scaling term is rather high, which is also noticed from the difference in scale in all histograms for the output nodes. This is interpreted as the *probabilistic model* introduces too large a standard deviation compared to the *approximate model*.

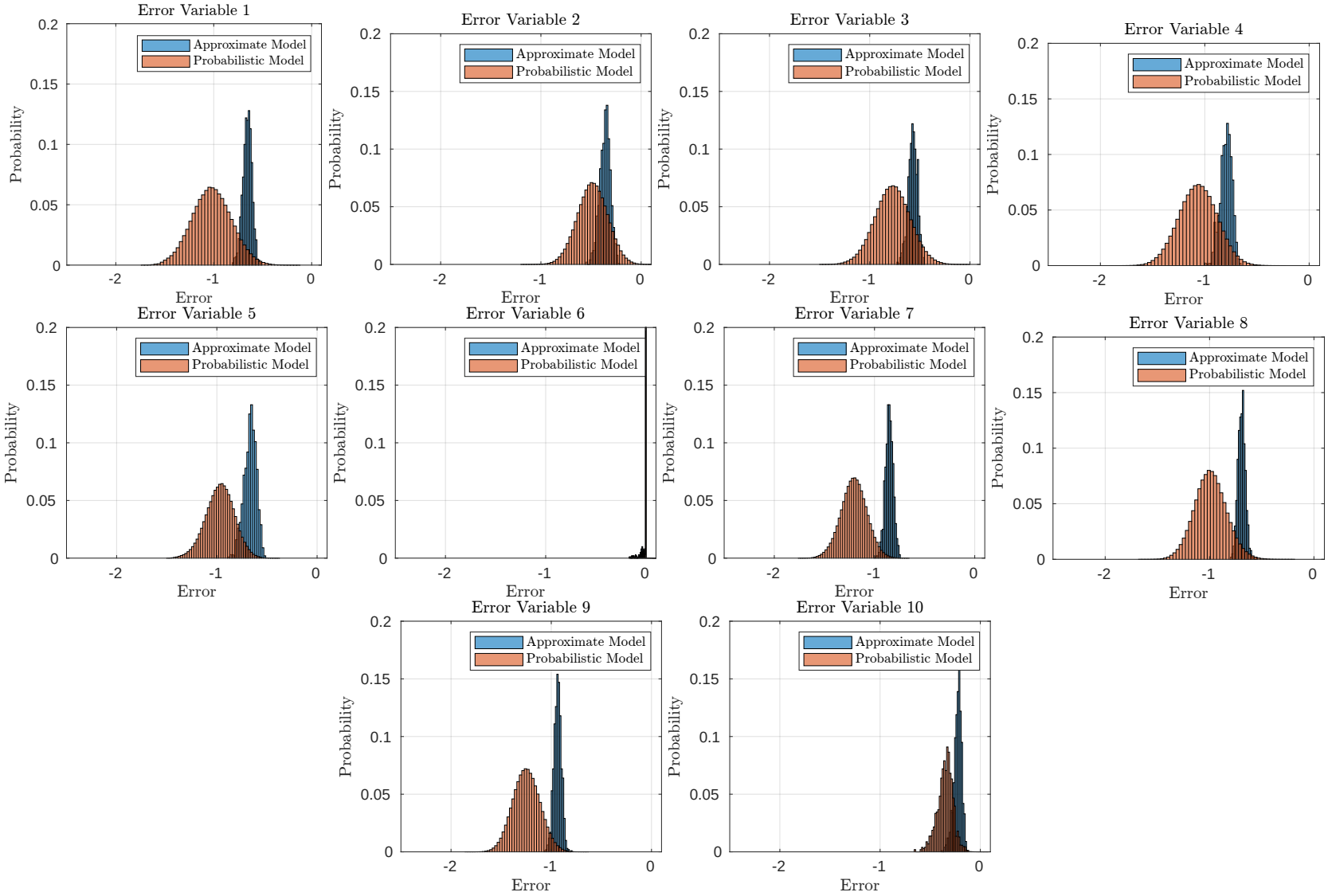

Figure E.23: Histogram of the deterministic and probabilistic error distributions for the `mul8s_1KV9`, given 1000 input images. The deterministic histograms are partitioned into 30 distinct bins of equal width, and the probabilistic histograms are 50. Each plot in this figure corresponds to one index of the error vectors.

E.3.3 Comparison of Kullback-Liebler Divergences

For comparison purposes the KL-divergences are plotted for the different multipliers, to illustrate the evolution of the metric over the epochs of STE-training on the *approximate model*. This is shown in Figure E.24.

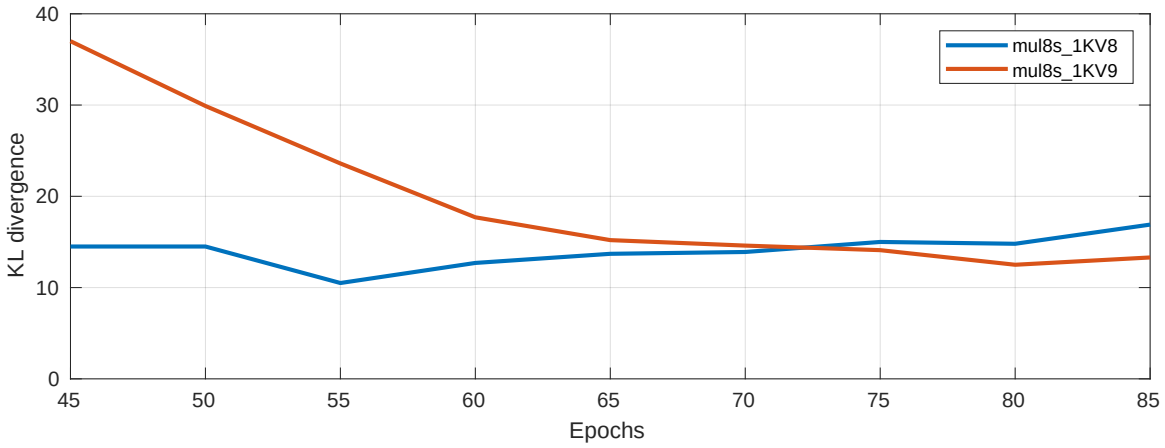

Figure E.24: Plot of the evolution of the KL-divergence, for progressing epochs. Both chosen multipliers are plotted for comparison purposes. The continuous lines are linear interpolations as measurements are only taken for each 5 epochs.

E.4 Conclusion

From the two experiments conducted in this appendix, it is concluded that the KL divergence is a measure that can be used for comparison of ''*how well*'' a probabilistic model emulates an observation from the *approximate model*. It is observed that it is possible to improve the divergence of the models, by training on the approximate model using STE. However, there seems to be a limit to how well the probabilistic model in its current form can model the deterministic error. The primary source of divergence is observed to be caused by the difference in standard deviation of the two models, where the probabilistic is larger than the deterministic. The differences in means of the two distributions are also contributing to the overall convergence, but it seems to be possible to reduce this impact through approximate training.