TRUSTWORTHY AI AT THE EDGE USING STOCHASTIC COMPUTING

by

Soroosh Khoram

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 04/27/2023

The dissertation is approved by the following members of the Final Oral Committee:
Mikko Lipasti, Professor, Electrical and Computer Engineering, UW-Madison
Joshua San Miguel, Asst. Professor, Electrical and Computer Engineering, UW-Madison
Stephen Wright, Professor, Computer Sciences, UW-Madison
Robert Nowak, Professor, Electrical and Computer Engineering, UW-Madison

To my brilliant partner Wenxin, my beautiful brother Erfan,
and my lovely parents Azam and Abolghasem.

ACKNOWLEDGMENTS

I extend my heartfelt gratitude to Professor Mikko Lipasti, my esteemed advisor, for their invaluable support and guidance. Their unwavering commitment to fostering an atmosphere of growth, collaboration, and student empowerment has been instrumental in accomplishing this significant milestone. I am truly grateful for their mentorship, which has not only shaped my academic journey but also nurtured my personal and professional development.

I would also like to thank Kyle Daruwalla, my dear friend and teammate for his contributions to this project. In particular, he synthesized neural network layers using Xilinx's vivado and provided power, energy, and delay metrics for neural network components which were used for evaluations in Chapter 7. I am deeply appreciative of his efforts which have greatly enhanced the quality of this work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

MY PUBLICATIONS

RELATED TO THESIS

Khoram, Soroosh, Kyle Daruwalla, and Mikko Lipasti (2023).
"Energy-Efficient Bayesian Inference Using Bitstream Computing".
In: *Computer Architecture Letters* (Minor Revision).

Khoram, Soroosh and Jing Li (2018). "Adaptive Quantization of Neural Networks".
In: *International Conference on Learning Representations*.
URL: https://openreview.net/forum?id=SyOK1Sg0W.

Shukla, Rohit et al. (2018).
"Computing Generalized Matrix Inverse on Spiking Neural Substrate".
In: *Frontiers in Neuroscience* 12. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00115.

OTHER PUBLICATIONS

Khoram, Soroosh, Yue Zha, and Jing Li (2018).
"An alternative analytical approach to associative processing".
In: *IEEE Computer Architecture Letters* 17.2, pp. 113–116.

Khoram, Soroosh, Yue Zha, Jialiang Zhang, et al. (2017). "Challenges and opportunities:
From near-memory computing to in-memory computing".
In: *Proceedings of the 2017 ACM on International Symposium on Physical Design*,
pp. 43–46.

Khoram, Soroosh, Jialiang Zhang, et al. (2017).
"Accelerating large-scale graph analytics with FPGA and HMC".
In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, pp. 82–82.

— (2018). "Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 239–248.

Zhang, Jialiang, Soroosh Khoram, and Jing Li (2017).
"Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 207–216.

— (2018). "Efficient large-scale approximate nearest neighbor search on OpenCL FPGA".
In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*,
pp. 4924–4932.

ABSTRACT

Machine learning systems often need to be able to evaluate their own predictive uncertainty, especially in high-risk areas, to activate safety measures when necessary. For example, a medical diagnosis model needs to inform the doctor if it is uncertain of a diagnosis so the doctor can account for that in their decision making. Alternatively, an autonomous vehicle that is unsure if it is detecting a pedestrian on the road it can transfer control to the driver. Essentially these models need to be able to say "I don't know." Unfortunately, conventional deep learning models are often overconfident in their predictions, even when encountering out-of-distribution inputs. Recent studies have addressed this challenge using Bayesian Neural Networks (BNNs), a type of neural network model that provides a prediction as well as its confidence (or more commonly *uncertainty*) in that prediction. However, the additional computations for BNNs compared to conventional neural networks can increase the energy cost of inference. Yet, in many edge applications where BNNs' Uncertainty Quantification (UQ) capability is crucial, power and energy can be constrained.

Previous works have discussed challenges of deploying deep learning models at the edge through various model compression techniques in DNNs. However, these studies usually do not consider the changes in the loss function when performing quantization, nor do they take the different importances of DNN model parameters to the accuracy into account. We address these issues in this paper by proposing a new method, called adaptive quantization.

We further approach these challenges in the context of BNNs by proposing the Bayesian Bitstream Processor (BBP) which uses Bitstream Computing (BC) to achieve low

cost, power, and energy. BC substrates perform computations serially and approximately. This allows for much simpler architecture design compared to architectures that perform precise computations using ALUs. They can this way offer smaller energy consumption. Furthermore, they inherently incorporate Random Number Generators (RNG). These RNGs can be further utilized during the random sampling steps of the BNN computations.

We first evaluate a basic design of BBP for an audio classification task and use it as a test case. Our results show that our approach can outperform a micro-controller baseline in energy by two orders of magnitude and delay by an order of magnitude, all while operating at lower power. While our results are promising, we identify a key challenge for deploying BNNs on bitstream computing substrates - designing random number generators that are suitable for this hardware.

To overcome this challenge, we present a novel bitstream generation architecture that can significantly reduce costs and, more importantly, allow for early decision making. Since bitstream computing substrates spread computations temporally, improving precision over time, early decision making can significantly improve delay. Our results show that the majority of sample data can often be evaluated quickly, allowing for uncertainty quantification at the edge at low costs.

In conclusion, our proposed approach of using stochastic bitstream computing substrates for deploying BNNs can significantly reduce power and costs while providing much-needed uncertainty quantification in machine learning applications. The novel bitstream generation architecture we present in this work is a crucial step towards making this approach more practical and efficient for edge devices.

1 INTRODUCTION

Artificial intelligence has become increasingly become pervasive in day-to-day life, demanding trust which cannot be granted without scrutiny. It is important that these systems be carefully studied not just through the lens of security, privacy, and reliability as computing systems often are but also their accuracy, robustness, fairness, uncertainty, and explainability (Wing, 2021). An important aspects that differentiates these systems is that they operate on probability calculations based on subjective beliefs. Moreover, these probabilities are heavily dependent on the availability of data. This facet is even more pronounced in recently developed machine learning algorithms based on deep models that heavily depend on large, diverse, representative datasets. This creates a critical challenge for AI that is trying to be trustworthy. That is, limitations of the data are going to be carried over to the trained model as gaps in knowledge and bias. Algorithms and methodologies that can quantify the uncertainty of the AI in it predictions therefore become an important pillar of trustworthy AI. They do this by assisting with identification of these gaps of knowledge and explaining the decisions of the model, holding it accountable to its predictions (Thiebes, Lins, and Sunyaev, 2021).

Uncertainty quantification is a crucial aspect of modeling, as it reflects the model's confidence in its predictions and its ability to accurately represent the data it learned from. However, achieving reliable uncertainty quantification can be challenging due to hardware limitations, particularly at the edge where energy, power, precision, and speed are tightly limited. Uncertainty quantification at the edge can be important where safety and security is of concern such as robotics and health monitoring. These computing paradigms can create conflicts with the theoretical guarantees provided by Machine Learning (ML)

research, which assumes precision and statistical characteristics of computations that may not hold after hardware optimizations for deployment at the edge. Therefore, finding scalable and efficient methods for uncertainty quantification is an ongoing challenge in AI research. Addressing this challenge can significantly improve the reliability and accuracy of AI systems across various applications.

While the computer systems research in the past several decades was mainly driven by improving computing performance and efficiency through technology scaling, with Moore's Law (Moore, 1998) slowing down (H. N. Khan, Hounshell, and Fuchs, 2018), efforts have been redirected. More specifically, there are three directions that promise to enable continued progress. These are namely *More Moore*, *More-Than-Moore*, and *Beyond Moore* (Semiconductors 2.0, 2015). More Moore and Beyond Moore pursue solutions, respectively, by finding new techniques to continue the current trend of technology scaling, and by pursuing new technology substrates that would allow technology to advance. On the other hand, More-Than-Moore seeks to address our current needs through *functional diversification*. This means specializing and optimizing existing hardware and technology for applications that need to be addressed today. It is important to keep in mind that how applications are prioritize is determined by demands of the market that may lead to the slow-down of progress for ones that do not offer immediate financial returns but are nevertheless necessary for the well-being of people and environment (Thompson and Spanuth, 2018).]. This presents a great opportunity of trustworthy AI with quantifiable uncertainty at the edge as the added complexity can be addressed through design.

The particular challenges faced when deploying Uncertainty Quantification at the edge are low power availability and the need for rapid random number generation. Practical implementations of uncertainty quantification often use Bayesian Inference methods that perform repeated inference on the same input with randomly sampled models. As a result, these methods can be computationally inefficient while also requiring high-throughput random sampling with complex probability density functions. Designing

hardware for uncertainty quantification at the edge therefore run into power, energy, and cost barriers. In this context, Bitstream Computing can be uniquely advantageous as it offers very low-power computing while also incorporating components that can be utilized in a Bayesian Inference platform such as random number generators. The rest of this chapter will expand high-level concepts related to uncertainty quantification and Bayesian inference, bitstream computing, and the challenges of deploying AI capable of uncertainty quantification at the edge using Bitstream Computing.

1.1 UNCERTAINTY QUANTIFICATION

Predictive uncertainty quantifies the confidence or uncertainty of an AI in its predictions and it is an important facet of achieving trustworthy AI. This is important to many safety-critical and security-critical applications, particularly at the edge. The ubiquity of mobile, Internet-Of-Things (IOT), and wearable devices monitoring and controlling our healthcare, our homes, and our cars and making decisions for our privacy, safety, and health highlights the importance of knowing when they make reliable predictions and when they do not. While there have been great strides made in improving the compute capability and accuracy of AI, there are fundamental limitations that necessitate humans in the loop when safety and security are concerns. This is due to the practical challenges of data collection as well as the simplifying assumptions we are bound to make when designing an AI model. Therefore, we need these AIs to be able to defer to backup measures and in essence be able to say "I don't know!"

AI model predictive uncertainty, provides a way to decide whether to activate backup measures. However, extracting reliable uncertainty is not possible with commonly used Deep Neural Networks as they have been shown to have overconfidence issues (Blundell et al., 2015). One solution that has been shown to provide useful uncertainty while preserving the high prediction accuracy of DNNs is Bayesian Neural Networks (G. E.

Hinton and Van Camp, 1993). We will discuss Bayesian Neural Networks (BNNs) and their difference with DNNs in more detail in the next chapter, but to summarize, BNNs virtually learn an ensemble of infinitely many DNNs. During inference a set of these DNNs are randomly selected, each of which computes a prediction for the input. Prediction is obtained as the average of their outputs while uncertainty is measured as the level of disagreement between them.

It is worth noting that randomly selecting Deep Neural Networks (DNNs) essentially means generating a new set of weights from a learned posterior distribution, which requires a high number of random number generations. Moreover, since all randomly sampled DNNs process the same input, the outputs of their layers can be similar, albeit with small random perturbations, as highlighted by Wan and Fu (2020). Therefore, employing Bayesian Neural Networks (BNNs) may not be computationally efficient. Already, deterministic deep models pose a challenge for edge computing platforms, given their large sizes and high computational requirements. Bayesian Inference, being more complex, further exacerbates this challenge, necessitating careful attention to both algorithm design and computer architecture design.

1.2 DEEP LEARNING AT THE EDGE

Typically, edge computing platforms for deterministic deep models rely on model compression to achieve low power and energy consumption in inference. Training usually requires high-precision computations. For inference on the other hand we can usually get away with low-precision computations and weights (F. Zhu et al., 2020). Most implementations of DNN computing platforms have used standard fixed-point precisions like 16-bit and 8-bit operations (Song et al., 2020) as there are well-supported training flows for such precisions (Jacob et al., 2018). Still, there have been proposals for training (F. Li, B. Zhang, and B. Liu, 2016; C. Zhu et al., 2016) and deployment (Sharma et al., 2018)

of smaller representations, i.e. 4-bit and 2-bit representations. Works by Courbariaux, Hubara, et al. (2016) and Rastegari et al. (2016) pushed this approach to the extreme by training models with weights represented using 1-bit representations.

Although these quantization methods can significantly reduce model complexity, they generally have two key constraints. First, they ignore the accuracy degradation resulting from quantization, during the quantization, and tend to remedy it, separately, through quantized learning schemes. However, such schemes have the disadvantage of converging very slowly compared to full-precision learning methods. Second, they treat all network parameters similarly and assign them the same *quantization width*, i.e. the number of bits used to store the fixed-point quantized value. This is while previous works (Han, Mao, and Dally, 2015; Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016a,b) have shown different parameters do not contribute to the model accuracy equally. Disregarding this variation limits the maximum achievable compression. We will further discuss this aspect of DNN model compression and quantization in chapter 3.

In the realm of BNNs, extreme quantization was proposed by Meng, Bachmann, and M. E. Khan (2020) by learning the weights of the network as Bernoulli distributions. This way they can provide the same advantages provided by Binarized Neural Networks (Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016b). Namely, using bitwise operations instead of arithmetic ones which can provide acceleration. Furthermore, as we will discuss shortly, this can have a straightforward implementation in Bitstream Computing as this computing paradigm views all parameters as Bernoulli distributions.

1.3 BITSTREAM COMPUTING

Bitstream Computing is a novel approach to approximate computing that encodes parameters temporally into bitstreams and performs computations bit-serially (Gaines, 1969; Gross and Gaudet, 2019). The accuracy of bitstream computation is directly proportional

to the length of the bitstreams, with longer bitstreams representing values at higher precisions, resulting in more accurate approximations of the output. However, this accuracy improvement comes at the cost of longer delay times. Despite this limitation, bitstream computing allows for the design of extremely low-cost computing hardware with low power consumption, making it an ideal solution for edge applications. This is particularly relevant for deep machine learning algorithms, where low-precision computations are sufficient for accurate results. Therefore, bitstream computing offers a promising solution to address the computational challenges faced by edge computing, providing a cost-effective and energy-efficient means of processing data in real-time.

Bitstream computing can efficiently overcome the two main challenges associated with BNN computations. Bitstream computing generates bitstreams in a stochastic manner by treating all numerical values as Bernoulli distributions [1] and repeatedly sampling this distribution. For this purpose, bitstream computing substrates incorporate many dedicated random number generator hardware. This makes bitstream computing substrates suitable for Bayesian inference, as these random number generators can be exploited for efficient sampling of BNNs' posterior distribution. Furthermore, bitstream computing is an effective solution for addressing the computational efficiency challenges of BNNs as it performs approximate computations. Short bitstreams generate rough estimates of the results, allowing for quick decisions on whether a longer bitstream is required for a more accurate calculation. By halting calculations early if the rough estimate is sufficient, bitstream computing can prevent inefficiencies (Hsiao, San Miguel, and Anderson, 2022).

[1] The underlying assumption here is that all values in bitstream computing are in the $[0,1]$ range. We will discuss bitstream computing in more detail in chapter 2.

1.4 MOTIVATING EXAMPLE

We demonstrate the capability and challenges of implementing BNNs on bitstream computing using a simple example. One straightforward way to implement Bayesian Neural Networks on a bitstream computing substrates is using the previously mentioned BayesBiNN (Meng, Bachmann, and M. E. Khan, 2020). In BayesBiNN, the learned posteriors follow Bernoulli distributions. As such, they can be easily implemented into a bitstream computing substrate without much design changes. In other words, BayesBiNN exploits the random number generators that exist for bitstream generation, for sampling its posterior distribution. Thus, in this scenario, bitstream computing substrates easily address the random sampling challenges for BNNs.

For this example, we use a synthetic classification dataset which we have depicted in Figure 1.1.

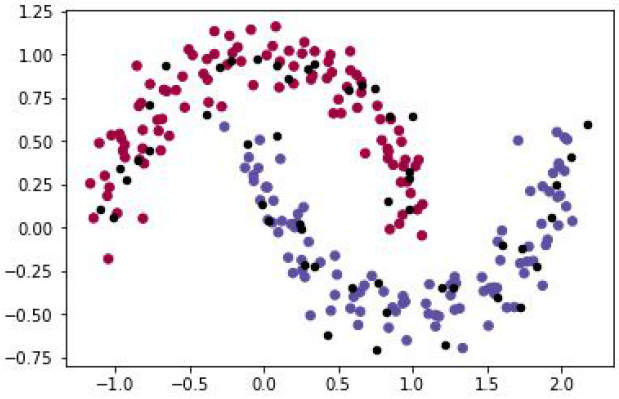
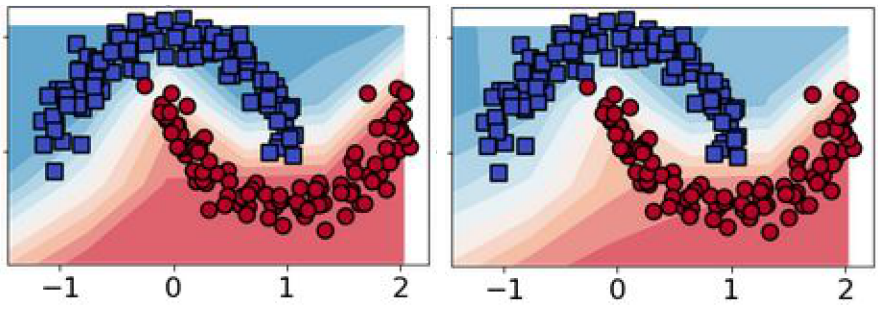


Figure 1.1: Synthetic classification dataset

We trained a Multi-layer perceptron for this classification task like the one used by Meng, Bachmann, and M. E. Khan (2020). We simulated implementation of this network on a bitstream computing substrate using the Bitstream Synthesizer and Designer (BitSAD), a simulation and development tool for bitstream computing designed by Daruwalla and

Zhuo (2019). We compare the results of this simulation with the baseline implemented on a CPU using standard floating-point computing in Figure 1.2.



(a) CPU (b) BitSAD

Figure 1.2: Comparison of BNN uncertainty evaluation between BitSAD and CPU (BayesBiNN)

In this figure, the background color in each region of the feature space shows the predicted class for that region while the intensity of the background color represents uncertainty. Lower intensity means higher uncertainty. In both figures, we see that as we move further away from regions for which we have training data, uncertainty increases confirming that the network can identify out-of-distribution inputs. Furthermore, we see that the overall results from BitSAD and the results from CPU are assigning higher uncertainties to similar regions. A key point to notice is that, near areas where we have training data, the reported results saturate to $+1$ or -1. As such, precise computations in these areas is not necessary and we can achieve higher computational efficiency by stopping when the output bitstream plateaus.

We have seen however that using heavily-quantized binarized neural networks can lead to networks with higher number of parameters (Khoram and J. Li, 2018) which can directly translate to larger circuits in bitstream computing settings. Furthermore, it is often the case that Bayesian Neural Networks learn more complex posterior distributions rather than simple Bernoulli ones (Blundell et al., 2015). Thus, it is important that our implementation of BNNs using bitstream computing is capable of deploying these higher precision networks with complex posterior structures. We address these design challenges in the rest of this dissertation.

1.5 SUMMARY

Uncertainty quantification is an important pillar of building trustworthy AI systems. Bayesian Neural Networks (BNNs) have been shown to provide useful uncertainty while preserving the high prediction accuracy of Deep Neural Networks (DNNs). However, employing BNNs at the edge introduces important challenges including the need for high-throughput random sampling and computational inefficiencies. We argue that bitstream computing can be a natural fit to address these challenges as they offer dedicated hardware for random sampling and can address computational inefficiencies through approximate computing. In the rest of this thesis, we will address the different challenges of deploying BNNs on bitstream computing substrates through various algorithm and computer architecture design methodologies. In particular, our main contributions are as follows:

- We will present a brief background on ML models for Bayesian Inference and discuss training techniques for adapting them for BC substrates.
- Present a case study for deploying Bayesian inference models on BC substrates.
- We will explore design of Pseudo Random Number Generators as a key component in Bayesian inference and propose a new design for BC substrates.
- Propose an early stopping method for Bayesian neural networks implemented on bitstream computing substrates to achieve computational efficiency.
- Evaluate the performance of the proposed design and show its advantages over conventional architecture designs.

We have organized the remainder of this document as follows. Chapter 2 provides detailed background on uncertainty quantification, BNNs, and bitstream computing. Chapter 3 discusses quantization as it pertains to conventional, deterministic DNNs and proposes a new DNN compression method. Chapter 4 explores issues of quantization,

precision, and numerical range for BNNs on bitstream computing and presents a baseline architecture for a BNN on a bitstream computing substrate. Chapter 6 will present the theoretical arguments for exploiting existing random number generators in bitstream computing substrates for BNN computations. This chapter further uses the proposed random number generator design for proposing an early stopping of bitstream computing to improve computational efficiency. Chapter 7 presents our experimental result. And finally, chapter 8 concludes this thesis and presents future directions of research.

2 RELATED WORKS

In recent years, the field of machine learning has seen a surge of interest in Bayesian methods, which offer a powerful framework for incorporating prior knowledge and uncertainty into the modeling process. Bayesian Inference and Bayesian Neural Networks (BNNs) have emerged as promising techniques for addressing some of the key challenges in machine learning, including overfitting, model selection, and uncertainty estimation. Furthermore, recent developments in hardware accelerators and stochastic computing have opened up new avenues for efficiently implementing BNNs in resource-constrained environments. In this related works chapter, we survey some of the recent advances in Bayesian Inference, BNNs, BNN Accelerator Hardware, and Stochastic Bitstream Computing, and discuss their potential impact on the field of machine learning.

2.1 BAYESIAN INFERENCE

Bayesian Inference is a powerful statistical tool for constructing and updating hypotheses to analyze unknown parameters based on observations. Let $\mathcal{H}$ be hypothesis we wish to investigate, associated with a set of observations like $\mathcal{D}$. The idea is to assume a *prior* probability distribution $p(\mathcal{H})$ which denotes the plausibility of all possible $\mathcal{H}$ prior to making any observations. The Bayesian approach then infers the probability of the hypothesis conditioned on the observed data using the Bayes' theorem. This is referred to as the *posterior* priority distribution, computed as:

$$p(\mathcal{H}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathcal{H})p(\mathcal{H})}{p(\mathcal{D})}$$

The posterior probability of the hypothesis is therefore proportional to the product of the likelihood and prior. The denominator is the marginal probability of the observations. Learning can be done through Maximum A Posteriori (MAP) by maximizing $p(\mathcal{D}|\mathcal{H})p(\mathcal{H})$. This principle has been successfully applied in a wide range of scientific fields such as medicine (Breda, Zavolan, and Nimwegen, 2021), environmental research (Cooley, Nychka, and Naveau, 2007), finance (Nirwan and Bertschinger, 2020), and astrophysics (Maselli et al., 2020).

As the complexity of a problem grows, it can become challenging to calculate the probabilities accurately. For instance, in a deep learning problem where where we often deal with high-dimensional data and the goal is to learn the parameters of a large network, computing the density of the network output will be intractable. In such cases, approximate methods are often employed to estimate the probability distribution of the network output.

Variational Inference (Blei, Kucukelbir, and McAuliffe, 2017) is a widely used approximate method for Bayesian Inference that has shown remarkable performance in many applications. It involves approximating the posterior distribution of the network parameters using a simpler distribution that can be calculated more easily. We then try to minimize the difference between this approximation and the true posterior using, for example, their Kullback-Leibler divergence. Variational Inference has several advantages over other approximate methods, such as Markov Chain Monte Carlo (S. Brooks et al., 2011), including being computationally faster and more scalable for large datasets. It was first applied to neural networks by G. E. Hinton and Van Camp (1993) and improved by Graves (2011) which allowed it to be employed for designing deep networks. Since then, there have been many studies in applications of Variational Inference in deep learning models (Bui et al., 2016; Hernández-Lobato and Adams, 2015; Wilson et al., 2016).

Variational Inference is a powerful technique that offers several key advantages in the realm of deep learning. A significant one of these is its ability to act as a principled regularization technique that helps to improve generalization (Blundell et al., 2015; Srivastava et al., 2014). It constrains the complexity of the model during training while introducing stochastic uncertainty to the neural network weights. This way Variational Inference helps to prevent overfitting and, at the same time, allows the model to explore different possible configurations, which can help it avoid getting stuck in suboptimal solutions.

Additionally, Variational Inference has been shown to be effective for aggressive compression purposes (Louizos, Ullrich, and Welling, 2017; Molchanov, Ashukha, and Vetrov, 2017; Ullrich, Meeds, and Welling, 2017). Ullrich, Meeds, and Welling (2017) argued for applying the Minimum Description Length principle (Grünwald, 2007) to the complexity of a deep learning model. By carefully choosing priors for the network weights, stochastic gradient descent can naturally produce a quantized and pruned network that maintains high accuracy.

Finally, a critical capability of Variational Inference is its ability to evaluate the uncertainty of a model (Durk P Kingma, Salimans, and Welling, 2015; Shridhar, Laumann, and Liwicki, 2018). Specifically, it enables us to quantify the uncertainty of the network's predictions for each input sample, which can be particularly useful in scenarios where the consequences of an incorrect prediction are significant. By providing a measure of uncertainty, we can make more informed decisions and take appropriate actions.

2.2 UNCERTAINTY QUANTIFICATION

While accuracy is often considered the default metric for evaluating AI systems, it can be a misleading measure, particularly in applications where misclassification can have catastrophic consequences. This is particularly relevant for AI systems that deal with

safety and privacy, such as wearable healthcare devices, smart assistants, and robotics. An erroneous classification can be life-threatening or expose sensitive personal information to unauthorized parties. In such cases, it is preferable to abstain from making a classification rather than making an incorrect one. For instance, misclassifying a patient's condition in a healthcare device can be fatal, while a smart assistant mistakenly recording and transmitting private conversations to the cloud can pose a severe privacy risk. Therefore, it is vital to evaluate the uncertainty of AI predictions to identify situations where the system lacks confidence in its predictions or is uncertain. This enables more informed decision-making, reduces the risk of errors, and enhances the safety and privacy of AI applications.

Quantifying uncertainty in traditional deep learning models trained using deterministic methods is a complex task. Various methods, such as margin sampling (Scheffer, Decomain, and Wrobel, 2001) or information entropy (Settles, 2009), have been proposed for measuring predictive uncertainty, but deterministic approaches tend to be overly confident (K. Wang et al., 2016). Although acceptable in low-risk scenarios like active learning, safety and security applications demand more reliable techniques. Bayesian Inference and Variational Inference, which approximates Bayesian Inference, are techniques that can provide measures of prediction variability. Several Variational Inference methods, including Bayesian Neural Networks (Neal, 2012) and Variational Dropout (Gal and Ghahramani, 2016), have been employed for uncertainty quantification in deep learning.

However, while Dropout is a widely used technique for uncertainty quantification due to its ability to avoid adding extra complexity to the model, its ease of scalability, and its ability to approximate Variational Inference (Durk P Kingma, Salimans, and Welling, 2015), studies indicate that it can still overestimate model confidence, even in simple cases (Gissin and Shalev-Shwartz, 2019). Alternatively, fully Bayesian Neural Networks can better approximate true uncertainty (Kendall and Gal, 2017). Additionally, methods

like Bayes-By-Backprop (Blundell et al., 2015) and BayesBiNN (Meng, Bachmann, and M. E. Khan, 2020) have demonstrated that training Bayesian Neural Networks using simple stochastic gradient descent and similar computational complexity to deterministic deep learning models can address scalability concerns. As such, it is necessary to consider these techniques to improve the reliability and safety of AI systems in safety-critical and security-critical applications.

Although Bayesian neural networks (BNNs) can provide a favorable trade-off between scalability and accurate uncertainty quantification, they present significant challenges to their underlying computing platform compared to non-Bayesian models. During the inference stage, BNNs require computing the output conditional distribution $p(y|\boldsymbol{x}, \theta, \mathcal{D})$ for each input $\boldsymbol{x}$, which makes them more computationally intensive than traditional neural networks that output a single point estimate. This increased computational cost can be a bottleneck for real-time and resource-constrained applications. Additionally, computing the posterior distribution involves integration over the high-dimensional space of network weights θ, which is typically achieved using Monte Carlo approximation (Geweke, 1989). This requires performing multiple forward passes, whereas a non-Bayesian model only requires one pass. Furthermore, estimating the output distribution numerically necessitates sampling the posterior distribution of the weights, which adds complexity to the conventional pipelines used in deep learning applications. Consequently, deploying BNNs requires careful design, especially at the edge, where there are strict limitations on power, energy, and cost.

2.3 HARDWARE ACCELERATORS FOR DEEP MODELS

Hardware accelerators have become increasingly popular for deep learning tasks due to their ability to provide significant speedups and energy efficiency over traditional CPUs and GPUs. In particular, the demand for real-time and low-power applications

has led to the development of specialized hardware accelerators designed specifically for deep neural networks. These accelerators can exploit the highly parallel nature of deep learning algorithms to achieve orders of magnitude faster processing speeds while consuming significantly less power than traditional computing platforms. However, although there have been significant advancements in hardware accelerators for traditional neural networks, Bayesian neural networks (BNNs) present additional challenges for their deployment. To better understand these challenges it is important that we take a look at some of the recent advances in hardware accelerators for deep models, including field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and tensor processing units (TPUs).

Deep learning has been propelled by specialization, which has enabled the development of efficient methodologies for training neural networks. While these methodologies have been around for decades (Rumelhart, G. E. Hinton, and Williams, 1986), the computations required for training were prohibitively expensive until the widespread availability of general-purpose computing on GPUs. While some may argue that "the history of deep learning began with AlexNet" (Alom et al., 2018), it was the implementation of convolution operations on GPUs that enabled the exploration of "interestingly large" convolutional neural networks (CNNs) (Krizhevsky, Sutskever, and G. E. Hinton, 2017). GPUs remain ubiquitous in machine learning applications, especially for training in data centers, where their higher precision is necessary (Reuther et al., 2020). There has been several prominent additions to the space of specialized ML hardware since. These include Microsoft's Brainwave project (Chung et al., 2018) which took advantage of the flexibility of FPGA's to enable precision-adaptable training on the cloud. Google on the other hand developed the TPU (Jouppi et al., 2017) as ASIC chip with optimized fixed-point integer operations to achieve high throughput and energy efficiency. However, computing at the edge presents a different challenge.

In most cases, the focus of edge applications is on inference rather than training. While techniques like federated learning (McMahan et al., 2017) exist for training models at the edge, they often come with challenges like privacy concerns (Boenisch et al., 2021). For inference at the edge, CPUs are common because they are widely available and easier to use (Hazelwood et al., 2018). Alternatively, highly optimized dedicated hardware accelerators can be used to take advantage of deep neural networks' tolerance of low-precision computing for inference. As the demand for intelligent edge devices grows, the development of more specialized and efficient hardware has more and more shaped the future of deep learning at the edge.

There have been more designs for deep learning accelerators that can be mentioned here. For an extensive list of such designs, take a look at studies by Reuther et al. (2019) and by Capra et al. (2020). Eyeriss (Y.-H. Chen, Emer, and Sze, 2016) is a notable example which used a spatial architecture and a row-stationary data-flow to achieve high energy efficiency. DianNao (T. Chen et al., 2014) similarly minimized memory access. Qualcomm developed Snapdragon processors for computing on quantized operations (Frumusanu, 2018). NVIDIA recently proposed NVIDIA Deep Learning Accelerator (NVDLA) (Farshchi, Q. Huang, and Yun, 2019), a modular and scalable open-source platform that supports a range of neural network models.

These architectures were originally created for conventional deep learning models and consequently encounter the same problems with overconfidence as Bayesian Neural Networks. Nonetheless, these architectures are not easily adaptable for deploying Bayesian Neural Networks due to their limited ability to generate random numbers at high throughputs, which is a critical requirement for sampling learned posterior distributions. Recently, there have been a few architecture designs that are suitable for BNNs. We will examine the related platform designs for Bayesian Inference next.

2.4 HARDWARE PLATFORMS FOR BNNS

BNNs offer unique advantages over traditional neural networks, including the ability to capture and quantify uncertainty in the model predictions. However, the implementation of BNNs in hardware presents several challenges, including computational efficiency as well as random number generation. As BNNs require several forward passes for each input, they can suffer from low computation efficiency. Furthermore, they need high-throughput generation of random numbers for all BNN weights. In recent years, researchers have proposed several hardware platforms specifically designed for Bayesian Inference and BNNs to address these challenges. In this section, we will discuss some of the most prominent hardware platforms.

Several works have proposed novel designs and methodologies for Bayesian inference, such as those by X. Zhang et al. (2021) and Chai et al. (2022) which proposed designs for Markov Chain Monte Carlo (Robert and Richardson, 1998) methods. While MCMC methods offer transparency and conceptual simplicity, these accelerators cannot be used for large BNN models. In another work, Fernandes *et. al.* (Fernandes et al., 2016) proposed a dedicated architecture for stochastic models using bitstream computing. However, their approach was limited to low-dimensional sensor data for simple robotics control using a shallow model. As such, their design cannot be used for deep BNN models.

FastBCNN (Wan and Fu, 2020) is another work that has proposed an accelerator for Bayesian neural networks. It took advantage of the fact that BNNs pass the same input through the network multiple times. They hypothesized that if the output of a ReLU activation unit is 0 in the first pass, it is likely to be 0 in the subsequent passes as well. Therefore, they can avoid recomputing this neuron unless its inputs change significantly. This allows them to improve computational efficiency by reducing redundant operations. However, the design is limited to networks that use Dropout layers, which can make it less applicable to a broader range of models. Not to mention, they were shown to be

overconfident compared to fully Bayesian Neural Networks. In contrast, we will propose an approach that aims to enable uncertainty quantification for a wide range of applications by offering an efficient hardware accelerator for deep networks with all-Bayesian layers. It is worth noting that as FastBCNN used Dropout for incorporating Bayesian Inference, they were able to get away with using simple random number generators.

VIBNN (Cai et al., 2018) and Shift-BNN (Wan, Xia, et al., 2021) are two works that have focused on optimizing the design of Random Number Generators (RNGs) for efficient random weight sampling in BNNs. VIBNN proposes a design using Wallace method that generates new Gaussian random samples through linear combination of Gaussians. The authors demonstrate that their approach can achieve comparable accuracy to conventional deep learning models on CPU. Shift-BNN, on the other hand, introduces a new approach to generating random weights based on random bit shifts and uses it for training BNNs. While these methods demonstrate good model accuracy, they do not discuss uncertainty quantification or its effect on decision making and cost.

Overall, while there have been several studies into accelerators for Bayesian inference, to our knowledge, a unified solution for the computational efficiency and random number generation remains a challenge as the existing designs address completely different Bayesian algorithms. Thus, an efficient hardware design that enables uncertainty quantification for deep neural networks with all-Bayesian layers still does not exist. We address these challenges through stochastic bitstream computing. Bitstream computing allows us to address computational efficiency issues through approximate computing. Furthermore, as bitstream computing paradigms already incorporate random number generators, they can easily adapt to BNN computing. In fact, we will propose a design that adapts the existing random number generators of bitstream computing architectures for Bayesian Neural Network random posterior samples. In the next section, we will discuss related works in bitstream computing.

2.5 BITSTREAM COMPUTING

Bitstream Computing offers an approximate computing paradigm that complements the computation flow of Bayesian Neural Networks, providing a promising solution to overcome the deployment challenges of such networks at the edge. This section provides a comprehensive overview of this paradigm, followed by a discussion of previous studies that have leveraged it to enhance Machine Learning applications.

2.5.1 *Bitstream Computing Overview*

The added computations of BNNs naturally translate to higher resource requirements. We plan to address this challenge using bitstream computing. Conceptually, BC performs computations between probabilities, represented as long-run bitstreams where frequency of observing a "1" bit represents the underlying probability. Figure 2.1 illustrates this concept for performing multiplications. On the left side, the two input bitstreams S_1 and S_2, represent underlying probabilities $\frac{6}{8}$ and $\frac{4}{8}$, respectively. Note that we have chosen to write these values this way since each bitstream has a length 8 and the bitstreams contain 6 and 4 non-zero bits. It is important that these bitstreams are generated randomly such that the probability of seeing a non-zero bit in each bitstream is equal to its underlying nominal value. Had we opted for longer bitstreams, the same values could be represented by having the same rates of non-zero bits.

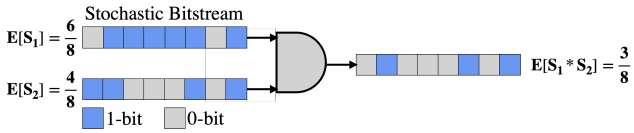


Figure 2.1: Multiplication in Bitstream Computing

The consequence of generating random samples is that the observed number of non-zero bits in a bitstream may not match exactly the desired frequency but only in

expectation. In fact, for a randomly generated bitstream of length T with the probability p, the observed value will approximately follow a Gaussian distribution like $\mathcal{N}\left(p, \frac{p(1-p)}{T}\right)$. Since the variance has an inverse relationship with the size of T, as we choose longer bitstreams, the intended values are represented more accurately.

In the example of Figure 2.1, the AND gate performs a multiplication between the two bitstreams. The output of an AND gate is non-zero only when both inputs are non-zero. Since such a case happens with a probability of $\frac{6}{8} \times \frac{4}{8} = \frac{3}{8}$, the expected output probability is the product of the input probabilities. Similar to when generating stochastic bitstreams, how accurately the output bitstream represents the desired output will directly depend on the length of the bitstream. As such, bitstream computing allows flexibly adjusting computation precision, though it introduces an important trade-off between delay and precision. There have been several proposals regarding the optimal lengths of bitstreams (Alaghi and Hayes, 2014; Hsiao, San Miguel, and Anderson, 2022) so that the output stabilizes without adding too much overhead, however, it is still important that the end-to-end accuracy of the results be verified experimentally.

Bitstream computing reduces the circuit cost as well as the power cost of performing computations. As we see in the example above, a multiplication can be performed, instead of with a complicated ALU, by a single AND gate. Other arithmetic operations can be performed using similarly simple circuits (Lee and Halim, 2020). In addition, previous studies have explored various stochastic implementations of non-linear functions using Finite State Machines for executing neural network operations (Y. Liu et al., 2020).

As such, compared to a conventional computing substrate a BC substrate requires much cheaper architectures which in turn will result in lower power consumption (Daruwalla and Zhuo, 2019). In addition, computations using bitstreams can result in sparse signals which will further lower power consumption. This low power and cost makes BC suitable for edge computing as it is the case that resources like power and energy are tightly limited at the edge.

2.5.2 *Bitstream Computing in Machine Learning*

The low-power consumption of Bitstream Computing circuits has made them attractive for machine learning applications at the edge. Therefore, various previous studies have exploited them for edge computing including in image processing (Alawad and M. Lin, 2014), control (D. Zhang and H. Li, 2008), and computer vision (B. Li, Najafi, and Lilja, 2016). These have studied the challenges of employing Bitstream Computing for ML applications at the edge.

Y. Zhang, S. Lin, et al. (2020) argued for exploiting and reinforcing the fault-tolerance aspect of Bitstream Computing for neural network applications. They argued that the stochastic nature of Bitstream Computing makes it more tolerant of errors due to its uniform coding scheme. This makes it suitable for harsh environments and low-precision computing which are both aspects of computing at the edge. They then designed architectures that reinforced these fault-tolerance characteristics.

One important challenge in employing Bitstream Computing for complex machine learning applications is efficient stochastic bitstream generation. A key assumption in the multiplication example of Figure 2.1 is that the input bitstreams are not correlated. In fact, correlation can be an important limiting factor in achieving deep models using Bitstream Computing (Alaghi and Hayes, 2013). This can be achieved by using separate independent stochastic bitstream generators for each input bitstream. As allocating individual stochastic bitstream generators to each value can be expensive for large models, Linear Feedback Shift Registers (LFSRs) have become standard in Bitstream Computing. LFSRs can produce uniformly distributed random numbers using simple hardware and can have low correlation which makes them ideal for Bitstream Computing (Jeavons, Cohen, and Shawe-Taylor, 1994). We will further discuss the architecture of an LFSR and how it is used to generate stochastic bitstreams in Chapter 6.

The random sampling present in the computation flow of Bitstream Computing resembles that of Bayesian Neural Networks. This can be helpful in computing at the edge where by using the LFSRs we may be able to generate the random weight samples needed. Exploiting the existing random number generation hardware can help streamline computations and enable higher performance and computational efficiency. This to our knowledge has not been done in the literature. Fernandes et al. (2016) proposed a dedicated architecture for stochastic models using bitstream computing. However, their approach was limited to low-dimensional sensor data for simple robotics control using a shallow model. As such, their design cannot be used for deep BNN models with complex posterior distributions. In this project, we will exploit this similarity to design Stochastic Bitstream Generators that are better suited for Bayesian Neural Networks.

Bitstream Computing performs computations serially which can be slow when bitstreams are long. Some efforts have tried to improve this by incorporating bit-parallel operations (Y. Zhang, R. Wang, et al., 2020). Other studies have addressed this challenge by adapting training of the neural network to tolerate stochasticity. Hirtzlin et al. (2019) showed that by adding stochastic noise to input images during training of a neural network, its Bitstream Computing implementation achieves high accuracy with shorter bitstreams. Others still have tried to determine the length of the bitstreams dynamically by stopping computations when the stochastic properties of the outputs bitstreams stabilize (Hsiao, San Miguel, and Anderson, 2022). This allows us to balance energy consumption and computation delay (Kim et al., 2016) through anytime computing algorithmic solutions (Zilberstein, 1996). In this work, we will build on this concept of dynamic early stopping and apply it to uncertainty quantification.

Finally, Daruwalla and Zhuo (2019) proposed the Bitstream Synthesizer and Designer (BitSAD), a domain-specific language for bitstream computing to address the lack of of sufficient development tools for bitstream computing. It supports scalar as well as matrix operations and allows for bit-level, cycle-accurate simulation of bitstream computations.

Furthermore, it can produce synthesizable verilog code to generate bitstream computing hardware. We use this tool for simulating and evaluating our designs.

2.6 SUMMARY

Bayesian Neural Networks have shown a promising ability to quantify uncertainty, which is critical for applications such as autonomous driving, medical diagnosis, and financial forecasting. By incorporating prior knowledge and uncertainty into the modeling process, BNNs can provide a more accurate estimate of the model's output and its associated uncertainty. However, deploying BNNs at the edge can be challenging due to resource constraints, such as limited energy and computational resources. Bitstream Computing can help overcome these challenges by providing low energy consumption. In addition, the existing random number hardware in Bitstream Computing substrates can be exploited for weight sampling of BNNs. Finally, Bitstream Computing can also improve computational efficiency through approximate computing, resulting in more accurate models with short bitstreams. Overall, the combination of BNNs and Bitstream Computing provides a promising solution for deploying machine learning models at the edge capable of uncertainty quantification while maintaining high accuracy and low energy consumption.

3 ADAPTIVE QUANTIZATION OF DEEP NEURAL NETWORKS

Deploying Deep Learning models onto resource-constrained edge computing devices is challenging due to their large size and complexity, even though they have state-of-the-art accuracy in various classification problems. Recent studies have reported success in reducing this complexity through quantization of DNN models, which can make inference at the edge more efficient. Before discussing bitstream computing which in a sense employs extreme quantization methods using 1-bit computations, and in Bayesian settings, it is important to carefully study the challenges of employing quantization and compression methods in deterministic contexts.

Inference can usually get away with low-precision computations and weights. This can be exploited for reducing computation costs at the edge. Early studies showed that by simply reducing the computation precision of a neural networks that was already trained using floating-point precision to standard fixed-point precisions like 16-bit and 8-bit operations, without losing much accuracy (Song et al., 2020). There were further proposals for training a neural networks with even smaller representations, i.e. 4-bit and 2-bit representations (F. Li, B. Zhang, and B. Liu, 2016; C. Zhu et al., 2016). Other works have further pushed quantization to the extreme by training models with weights represented using 1-bit representations (Courbariaux, Hubara, et al., 2016; Rastegari et al., 2016).

Although these quantization methods can significantly reduce model complexity, they generally have two key constraints. First, they ignore the accuracy degradation resulting from quantization, during the quantization, and tend to remedy it, separately, through

quantized learning schemes. However, such schemes have the disadvantage of converging very slowly compared to full-precision learning methods. Second, they treat all network parameters similarly and assign them the same *quantization width*, i.e. the number of bits used to store the fixed-point quantized value. This is while previous works have shown different parameters do not contribute to the model accuracy equally. Disregarding this variation limits the maximum achievable compression.

However, these studies usually do not consider the changes in the loss function when performing quantization, nor do they take the different importances of DNN model parameters to the accuracy into account. We address these issues in this chapter by proposing a new method, called adaptive quantization, which simplifies a trained DNN model by finding a unique, optimal precision for each network parameter such that the increase in loss is minimized. The optimization problem at the core of this method iteratively uses the loss function gradient to determine an error margin for each parameter and assigns it a precision accordingly. Since this problem uses linear functions, it is computationally cheap and, as we will show, has a closed-form approximate solution. Experiments on MNIST, CIFAR, and SVHN datasets showed that the proposed method can achieve near or better than state-of-the-art reduction in model size with similar error rates. Furthermore, it can achieve compressions close to floating-point model compression methods without loss of accuracy.

3.1 INTRODUCTION

Deep Neural Networks (DNNs) have achieved incredible accuracies in applications ranging from computer vision (Simonyan and Zisserman, 2014) to speech recognition (G. Hinton et al., 2012) and natural language processing (Devlin et al., 2014). One of the key enablers of the unprecedented success of DNNs is the availability of *very large model sizes*. While the increase in model size improves the classification accuracy, it

inevitably increases the computational complexity and memory requirement needed to train and store the network. This poses challenges in deploying these large models in resource-constrained edge computing environments, such as mobile devices. These challenges motivate *neural network compression*, which exploits the redundancy of neural networks to achieve drastic reductions in model sizes. The state-of-the-art neural network compression techniques include weight quantization (Courbariaux, Bengio, and David, 2015), weight pruning (Han, Mao, and Dally, 2015), weight sharing (Han, Mao, and Dally, 2015), and low rank approximation (Zhao et al., 2017). For instance, weight quantization has previously shown good accuracy with fixed-point 16-bit and 8-bit precisions (Qiu et al., 2016; Suda et al., 2016). Recent works attempt to push that even further towards reduced precision and have trained models with 4-bit, 2-bit, and 1-bit parameters using quantized training methods (Courbariaux, Bengio, and David, 2015; Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016a,b; Zhou et al., 2016).

Although these quantization methods can significantly reduce model complexity, they generally have two key constraints. First, they ignore the accuracy degradation resulting from quantization, during the quantization, and tend to remedy it, separately, through quantized learning schemes. However, such schemes have the disadvantage of converging very slowly compared to full-precision learning methods. Second, they treat all network parameters similarly and assign them the same *quantization width*[1]. This is while previous works (Han, Mao, and Dally, 2015; Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016a,b) have shown different parameters do not contribute to the model accuracy equally. Disregarding this variation limits the maximum achievable compression.

In this chapter, we address the aforementioned issues by proposing *adaptive quantization*. To take the different importances of network parameters into account, this method quantizes each network parameter of a trained network by a unique quantization width. This way, parameters that impact the accuracy the most can be represented using higher

[1] Number of bits used to store the fixed-point quantized value.

precisions (larger quantization widths), while low-impact parameters are represented with fewer bits or are pruned. Consequently, our method can reduce the model size significantly while maintaining a certain accuracy. The proposed method monitors the accuracy by incorporating the loss function into an optimization problem to minimize the models. The output of the optimization problem is an error margin associated to each parameter. This margin is computed based on the loss function gradient of the parameter and is used to determine its precision. We will show that the proposed optimization problem has a closed-form approximate solution, which can be iteratively applied to the same network to minimize its size. We test the proposed method using three classification benchmarks comprising MNIST, CIFAR-10, and SVHN. We show that, across all these benchmarks, we can achieve near or better compressions compared to state-of-the-art quantization techniques. Furthermore, we can achieve compressions similar to the state-of-the-art pruning and weight-sharing techniques which inherently require more computational resources for inference.

3.2 PROPOSED QUANTIZATION ALGORITHM

Despite their remarkable classification accuracies, large DNNs assimilate redundancies. Several recent works have studied these redundancies by abstracting the network from different levels and searching for, in particular, redundant filters, e.g. DenseNets (G. Huang, Z. Liu, and Weinberger, 2016), and redundant connections, e.g. Deep Compression (Han, Mao, and Dally, 2015). In this work, we present an even more fine-grained study of redundancy and extend it to fixed-point quantization of network parameters. That is, we approximate the minimum size of the network when each parameter is allowed to have a distinct number of precision bits. Our goal here is to represent parameters with high precisions only when they are critical to the accuracy of the network. In this sense, our approach is similar to weight pruning (Han, Mao, and Dally, 2015) which eliminates

all but the essential parameters, producing a sparse network. In the rest of this section, we first formally define the problem of minimizing the network size as an optimization problem. Then, we propose a trust region technique to approximately solve this problem. We will show that, in each iteration of the trust region method, our approach has a straightforward, closed-form solution. Finally, we will explain how the hyper parameters in the algorithm are chosen and discuss the implications of the proposed technique.

It is also important to discuss the benefits of this fine-grained quantization for performance. In particular, besides its storage advantages, we argue that this method can reduce the computation, if the target hardware can take advantage of the non-standard, yet small quantization depths that our approach produces. We note that there exist techniques on CPU and GPU for fixed-point arithmetics with non-standard quantization widths, e.g. SWAR (Cameron and D. Lin, 2009). But, we believe our proposed quantization is ideal for FPGAs and specialized hardware. The flexibility of these platforms allows for design of efficient computation units that directly process the resulting fixed-point quantized parameters. This was recently explored by Albericio et al. (2017), who designed specialized computation units for variable bit-width parameters that eliminated ineffectual computations. By minimizing the overall number of bits that need to be processed for a network, the proposed quantization achieves the same effect.

3.2.1 *Problem Definition*

A formal definition of the optimization problem that was discussed previously is presented here. Then, key characteristics of the objective function are derived. We will use these characteristics later when we solve the optimization problem.

We minimize the aggregate bit-widths of all network parameters while monitoring the training loss function. Due to the quantization noise, this function deviates from its optimum which was achieved through training. This noise effect can be controlled by

introducing an upper bound on the loss function in order to maintain it reasonably close to the optimum. Consequently, the solution of this minimization problem represents critical parameters with high precision to sustain high accuracy and assigns low precisions to ineffectual ones or prunes them. The problem described here can be defined in formal terms as below.

$$\min_{W} N_Q(W) = \sum_{i=1}^{n} N_q(\omega_i) \tag{3.1}$$

$$\mathcal{L}(W) \leq \overline{\ell} \tag{3.2}$$

Here, n is the number of model parameters, $W = [\omega_1 ... \omega_n]^T$ is a vector of all model parameters, ω_i is the i-th model parameter, and the function $N_q(\omega_i)$ is the minimum number of bits required to represent ω_i in its fixed-point format. As a result, $N_Q(W)$ is the total number of bits required to represent the model. In addition, $\mathcal{L}(W)$ is the loss function value of the model W over the training set (or a minibatch of the training set). Finally, $\overline{\ell}$ ($\geq \mathcal{L}(W_0)$) is a constant upper bound on the loss of the neural network, which is used to bound the accuracy loss, and W_0 is the vector of initial model parameters before quantization.

The optimization problem presented in Equations 3.1 and 3.2 is difficult to solve because of its non-smooth objective function. However, a smooth upper limit can be found for it. In the lemma below we derive such a bound. To our knowledge, this is the first time such a bound has been developed in the literature.

Lemma 3.1. *Given a vector of network parameters W, a vector of tolerance values $T = [\tau_1 ... \tau_n]^T$, and a vector of quantized network parameters $W_q = [\omega_1^q ... \omega_n^q]^T$, such that each $\omega_i^q, i \in [1, n]$ has a quantization error of at most τ_i, meaning it solves the constrained optimization function:*

$$\min_{\omega^q} N_q(\omega_i^q) \tag{3.3}$$

$$|\omega_i^q - \omega_i| \leq \tau_i \tag{3.4}$$

We have that:

$$N_Q(W) \leq \Phi(T) \tag{3.5}$$

Where $\Phi(T)$ is a smooth function, defined as:

$$\Phi(T) = -\sum_{i=1}^{n} \log_2 \tau_i \tag{3.6}$$

Proof. In Equation 3.3, we allow each parameter ω_i to be perturbed by at most τ_i to generate the quantized parameter ω_i^q. The problem of Equation 3.3 can be easily solved using Algorithm 1, which simply checks all possible solutions one-by-one. Here, we allocate at most 32 bits to each parameter. This should be more than enough as previous works have demonstrated that typical DNNs can be easily quantized to even 8 bits without a significant loss of accuracy.

Algorithm 1 Quantization of a parameter

```
procedure QUANTIZE_PARAMETER(ω_i, τ_i)
    ω_i^q ← 0
    N_q^i ← 0
    while N_q^i ≤ 32 do
        if |ω_i^q − ω_i| ≤ τ_i then
            break
        end if
        ω_i^q ← round(2^{N_q^i} ω_i) / 2^{N_q^i}
        N_q^i ← N_q^i + 1
    end while
    return N_q^i, ω_i^q
end procedure
```

Algorithm 1 estimates ω_i with a rounding error of up to $\frac{1}{2^{N_q^i+1}}$. Thus, for the worst case to be consistent with the constraint of Equation 3.3, we need:

$$\frac{1}{2^{N_q(\omega_i)+1}} \leq \tau_i \Rightarrow N_q(\omega_i) + 1 \geq -\log_2 \tau_i \tag{3.7}$$

In this case the minimization guarantees that the smallest $N_q(\omega_i)$ is chosen. This entails:

$$N_q(\omega_i) \leq -\lceil \log_2 \tau_i \rceil - 1 \leq -\log_2 \tau_i \tag{3.8}$$

In general, we can expect $N_q(\omega_i)$ to be smaller than this worst case. Consequently:

$$N_Q(W) = \sum_{i=1}^{n} N_q(\omega_i) \leq -\sum_{i=1}^{n} \log_2 \tau_i = \Phi(T) \tag{3.9}$$

□

We should note that for simplicity Algorithm 1 assumes that parameters ω_i are unsigned and in the range $[0, 1]$. In practice, for signed values, we quantize the absolute value using the same method, and use one bit to represent the sign. For models with parameters in a range larger than $[-1, 1]$, say $[-r, r]$ with $r > 1$, we perform the quantization similarly. The only difference is that we first scale the parameters and their tolerances by $\frac{1}{r}$ to bring them to $[-1, 1]$ range and then quantize the parameters. When computing the output of a quantized layer, then, we multiply the result by r to account for the scale. Following this, the same equations as above will hold.

3.2.2 *Solving the Optimization Problem*

The optimization problem of Equations 3.1 and 3.2 presents two key challenges. First, as was mentioned previously, the objective function is non-smooth. Second, not only is the constraint (Equation 3.2) non-linear, but also it is unknown. In this section, we present a method that circumvents these challenges and provides an approximate solution for this problem.

We sidestep the first challenge by using the upper bound of the objective function, $\Phi(T)$. Particularly, we approximate the minimum of $N_Q(W)$ by first finding the optimal

T for $\Phi(T)$ and then calculating the quantized parameters using Algorithm 1. This optimization problem can be defined as: minimize $\Phi(T)$ such that if each ω_i is perturbed by, at most, τ_i, the loss constraint of Equation 3.2 is not violated:

$$\min_{T} \Phi(T) \tag{3.10}$$

$$\mathcal{L}(W_0 + \Delta W) \leq \bar{\ell} \tag{3.11}$$

$$\forall \Delta W = [\Delta\omega_1 ... \Delta\omega_n]^T \quad * suchthat * \quad \forall i \in [1, n] : |\Delta\omega_i| \leq \tau_i \tag{3.12}$$

It is important to note that although the upper bound $\Phi(T)$ is smooth over all its domain, it can be tight only when $\forall i \in [1, n] : \tau_i = 2^{-k}, k \in \mathbb{N}$. This difference between the objective function and $\Phi(T)$ means that it is possible to iteratively reduce $N_Q(W)$ by repeating the steps of the indirect method, described above, and improve our approximation of the optimal quantization.

Such an iterative algorithm would also help address the second challenge. Specifically, it allows us to use a simple model of the loss function (e.g. a linear or quadratic model) as a stand-in for our complex loss function. If in some iteration of the algorithm the model is inaccurate, it can be adjusted in the following iteration. In our optimization algorithm, we will use a linear bound of the loss function and adopt a *Trust Region* method to monitor its accuracy.

Trust region optimization methods iteratively optimize an approximation of the objective function. In order to guarantee the accuracy of this approximation, they further define a neighborhood around each point in the domain, called a trust region, inside which the model is a "sufficiently good" approximation of the objective function. The quality of the approximation is evaluated each iteration and the size of the trust region is updated accordingly. In our algorithm we adopt a similar technique, but apply it to the constraint instead of the objective function.

Lemma 3.2. *We refer to $m_\ell : \mathcal{R}^n \to \mathcal{R}$ as an accurate estimation of $\mathcal{L}(W)$ in a subdomain of $\mathcal{L}$ like $\mathcal{D} \subset \mathcal{R}^n$, if:*

$$\forall W \in \mathcal{D} : m_\ell(W) \leq \overline{\ell} \Rightarrow \mathcal{L}(W) \leq \overline{\ell} \tag{3.13}$$

Now, let the radius Δ specify a spherical trust region around the point W_0 where the loss function $\mathcal{L}$ is accurately estimated by its first-order Taylor series expansion. Then, the constraint of Equation 3.16, when $\|T\|_2 \leq \Delta$, is equivalent to:

$$m_\ell(T) = \mathcal{L}(W_0) + G^T T \leq \overline{\ell} \tag{3.14}$$

Where $G = [g_1 ... g_n]^T$ and $g_i = |[\nabla_W \mathcal{L}(W_0)]_i|, \forall i \in [1, n]$.

Proof. Since $\|T\|_2 \leq \Delta$, then for ΔW as defined in Equation 3.12, $W_0 + \Delta W \in \mathcal{D}$. Therefore, we can write:

$$\mathcal{L}(W_0) + \nabla_W \mathcal{L}(W_0)^T \Delta W \leq \mathcal{L}(W_0) + G^T T = m_\ell(T) \leq \overline{\ell} \Rightarrow \mathcal{L}(W_0 + \Delta W) \leq \overline{\ell} \tag{3.15}$$

□

We note that we did not make any assumptions regarding the point W_0 and the result can be extended to any point in $\mathcal{R}^n$. Consequently, if we define such a trust region, we can simply use the following problem as a subproblem that we repeatedly solve in successive iterations. We will present the solution for this subproblem in the next section.

$$\min_T \Phi(T) \tag{3.16}$$

$$m_\ell(T) \leq \overline{\ell} \tag{3.17}$$

Algorithm 2 summarizes the proposed method of solving the original optimization problem (Equation 3.1). This algorithm first initializes the initial trust region radius Δ_0 and the loss function bound $\overline{\ell}$. It also quantizes the input floating-point parameters W_0 with 32 bits to generate the initial quantized parameters W^0. Here the function *Quantize(.,.)*

refers to a pass of Algorithm 1 over all parameters. Thus, using 0 tolerance results in quantization with 32 bits.

Subsequently, Algorithm 2 iteratively solves the subproblem of Equation 3.16 and calculates the quantized parameters. In iteration k, if the loss function corresponding to the quantized parameters W^k violates the loss bound of Equation 3.2, it means that the linear estimation was inaccurate over the trust region. Thus, we reduce Δ_k in the following iteration and solve the subproblem over a smaller trust region. In this case, the calculated quantized parameters are rejected. Otherwise, we update the parameters and enlarge the trust region. But we also make sure that the trust region radius does not grow past an upper bound like $\overline{\Delta}$.

We use two measures to examine convergence of the solution: the loss function and the trust region radius. We declare convergence, if the loss function of the quantized model converges to the loss bound $\overline{\ell}$ or the trust region radius becomes very small, indicated by values ϵ and η, respectively. Note that the protocol to update the trust region radius as well as the trust region convergence condition used in this algorithm are commonly used in trust region methods.

Algorithm 2 Adaptive Quantization

$k \leftarrow 0$
Initialize $\Delta_0, \overline{\Delta}$, and $\overline{\ell}$
$W^0 = Quantize(W_0, o)$
while $\overline{\ell} - \mathcal{L}(W^k) \geq \epsilon$ and $\Delta_k \geq \eta$ **do**
  Define $G^k = [g_1^k ... g_n^k]^T$ for $g_i^k = |[\nabla_W \mathcal{L}(W^k)]_i|$
  Define $m_\ell^k(T) = \mathcal{L}(W^k) + G^{k^T} T$
  Find T^k by solving the Trust Region Subproblem (Equation 3.16) with $m_\ell(T) = m_\ell^k(T)$ following section 2.3
  $\tilde{W} = Quantize(W^k, T^k)$
  if $\mathcal{L}(\tilde{W}) \leq \overline{\ell}$ **then**
    $W^{k+1} \leftarrow \tilde{W}$
    $\Delta_{k+1} \leftarrow min(2\Delta_k, \overline{\Delta})$
  else
    $\Delta_{k+1} \leftarrow \frac{1}{2}\Delta_k$
  end if
  $k \leftarrow k + 1$
end while

3.2.3 *Trust Region Subproblem*

In each iteration k, we can directly solve the subproblem of Equation 3.16 by writing its KKT conditions:

$$\nabla \Phi + \psi^k \nabla m_\ell^k = 0 \tag{3.18}$$

$$\psi^k (m_\ell^k - \overline{\ell}) = 0 \tag{3.19}$$

The gradient of the objective function, $\nabla\Phi = -\frac{1}{\ln 2}[\frac{1}{\tau_1}...\frac{1}{\tau_n}]$ cannot be zero. Hence, $\psi^k > 0$ and $m_\ell^k - \overline{\ell} = 0$. We can use this to calculate T^k, the solution of the subproblem.

$$\nabla\Phi + \psi^k \nabla m_\ell^k = 0 \Rightarrow \forall i \in [1,n] : -\frac{1}{\tau_i^k \ln 2} + \psi^k g_i^k = 0 \Rightarrow \forall i \in [1,n] : \tau_i^k g_i^k = \frac{1}{\psi^k \ln 2} \tag{3.20}$$

$$\Rightarrow G^{k^T} T^k = \frac{n}{\psi^k \ln 2} \tag{3.21}$$

Therefore, we can write:

$$m_\ell^k - \overline{\ell} = 0 \Rightarrow \mathcal{L}(W^k) + \frac{n}{\psi^k \ln 2} - \overline{\ell} = 0 \Rightarrow \frac{n}{\psi^k \ln 2} = \frac{\overline{\ell} - \mathcal{L}(W^k)}{n} \tag{3.22}$$

$$\Rightarrow \forall i \in [1,n] : \tau_i^k = \frac{\overline{\ell} - \mathcal{L}(W^k)}{n g_i^k} \tag{3.23}$$

If the resulting tolerance vector $\|T^k\|_2 > \Delta_k$, we scale T^k so that its norm would be equal to Δ_k.

This solution is correct only when $g_i^k > 0$ for all i. It is possible, however, that there exists some i for which $g_i^k = 0$. In this case, we use the following equation to calculate the solution T^k.

$$\tau_i^k = \begin{cases} \frac{\overline{\ell} - \mathcal{L}(W^k)}{n g_i^k} & g_i^k > 0 \\ |\omega_i^k| & g_i^k = 0 \end{cases} \tag{3.24}$$

We treat singular points this way for three reasons. First, a gradient of zero with respect to ω_i^k means that the loss is not sensitive to parameter values. Thus, it might be possible to eliminate it quickly. Second, this insensitivity of the loss $\mathcal{L}$ to ω_i^k means that large deviations in the parameter value would not significantly affect $\mathcal{L}$. Finally, setting the value of τ_i^k to a large value relative to other tolerances reduces their values after normalization to the trust region radius. Thus, the effect of a singularity on other parameters is reduced.

3.2.4 *Choosing the Hyper-parameters*

The hyper-parameters, $\overline{\Delta}$, and $\overline{\ell}$, determine the speed of convergence and the final classification accuracy. Smaller trust regions result in slower convergence, while larger trust regions produce higher error rates. For $\overline{\Delta}$, since remaining values could ultimately be represented by 0 bits (pruned), we choose $2^0\sqrt{n}$. Similarly, the higher the loss bound $\overline{\ell}$, the lower the accuracy, while small values of loss bound prevent effective model size reduction. We choose this value in our algorithm on the fly. That is, we start off by conservatively choosing a small value for $\overline{\ell}$ (e.g. $\overline{\ell} = \mathcal{L}(W_0)$) and then increase it every time Algorithm 2 converges. Algorithm 3 provides the details of this process. At the beginning the loss bound is chosen to be the loss of the floating-point trained model. After quantizing the model using this bound, the bound value is increased by *Scale*. These steps are repeated *Steps* times. In our experiments in the next section *Scale* and *Steps* are set to 1.1 and 20, respectively.

Algorithm 3 Choosing hyper-parameters

```
Initialize Steps and Scale
ℓ̄ ← L(W_0)
while Steps > 0 do
    Quantize model using Algorithm 2.
    Steps ← Steps − 1
    ℓ̄ ← ℓ̄ ∗ Scale
end while
```

3.3 EVALUATION

We evaluate adaptive quantization on three popular image classification benchmarks. For each, we first train a neural network in the floating-point domain, and then apply a pass of algorithm 3 to compress the trained model. In both these steps, we use the same batch size

to calculate the gradients and update the parameters. To further reduce the model size, we tune the accuracy of the quantized model in the floating-point domain and quantize the tuned model by reapplying a pass of algorithm 3. For each benchmark, we repeat this process three times, and experimentally show that this produces the smallest model. In the end we evaluate the accuracy and the size of the quantized models. Specifically, we determine the overall number of bits (quantization bits and the sign bits), and evaluate how much reduction in the model size has been achieved.

We note that it is also important to evaluate the potential overheads of bookkeeping for the quantization widths. However, we should keep in mind that bookkeeping has an intricate relationship with the target hardware, which may lead to radically different results on different hardware platforms. For example, our experiments show that on specialized hardware, such as the one designed by Albericio et al. (2017) for processing variable bit-width CNN, we can fully offset all bookkeeping overheads of storing quantization depths, while CPU/GPU may require up to 60% additional storage. We will study this complex relationship separately, in our future work, and in the context of hardware implementation. In this chapter, we limit the scope to algorithm analysis, independent of the underlying hardware architecture.

3.3.1 *Benchmarks*

We use MNIST (LeCun, Cortes, and Burges, 1998), CIFAR-10 (Krizhevsky, G. Hinton, et al., 2009), and SVHN (Netzer et al., 2011) benchmarks in our experiments. MNIST is a small dataset containing 28×28 images of handwritten digits from 0 to 9. For this dataset, we use the LeNet-5 network (LeCun, Bottou, et al., 1998). For both CIFAR-10, a dataset of 32×32 images from 10 object classes, and SVHN, a large dataset of 32×32 real-world images of digits, we employ the same organization of convolution layers and fully connected layers as networks used in BNN (Hubara, Courbariaux, Soudry, El-Yaniv,

et al., 2016b). These networks are based on VGG (Simonyan and Zisserman, 2014) but have parameters in full precision. The only difference is that in the case of CIFAR-10, we use 4096 neurons instead of the 1024 neurons used in BNN, and we do not use batch normalization layers. We train these models using a Cross entropy loss function. For CIFAR-10, we also add an L_2 regularization term. Table 7.2 shows the specifications of the baseline trained models.

Table 3.1: Baseline models

Dataset	Model	Model Size	Error Rate
MNIST	LeNet-5	12*Mb*	0.65%
CIFAR-10	VGG	612*Mb*	9.08%
SVHN	VGG	110*Mb*	7.26%

3.3.2 *Execution Time*

The key contributors to the computational load of the proposed technique are back propagation (to calculate gradients) and quantization (Algorithm 2). Both these operations can be completed in $\mathcal{O}(n)$ complexity. We implement the proposed quantization on Intel Core i7 CPU (3.5 GHz) with Titan X GPU performing training and quantization. The timing results of the algorithm have been summarized in Table 3.2.

3.3.3 *Quatization Results*

We evaluate the performance of the quantization algorithm by analyzing the compression rate of the models and their respective error rates after each pass of quantization excluding retraining. As discussed in section 2, the quantization algorithm will try to reduce

Table 3.2: Timing results for training and quantization of the benchmark models in **seconds**

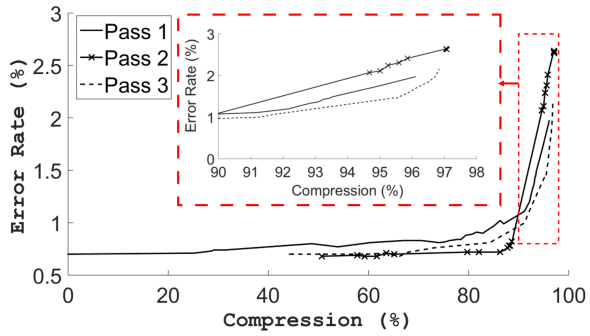
Dataset	Training	Quantization
MNIST	120	300
CIFAR-10	4560	4320
SVHN	1920	2580

parameter precisions while maintaining a minimum classification accuracy. Here we present the results of these experiments.

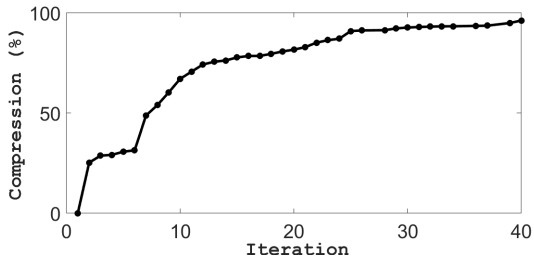
Figure 3.1a depicts the error rate of LeNet trained on the MNIST dataset as it is compressed through three passes of adaptive quantization and retraining. As this figure shows, the second and third passes tend to have smaller error rates compared to the first pass. But, at the end of the second pass the error rate is higher than the first one. This can be improved by increasing the number of epochs for retraining or choosing a lower cost bound ($\bar{\ell}$). By the end of the third pass, the highest compression rate is achieved. However, due to its small difference in compression rate compared to its preceding pass, we do not expect to achieve more improvements by continuing the retraining process.

We also evaluate the convergence speed of the algorithm by measuring the compression rate ($1 - \frac{\textit{size of quantized model}}{\textit{size of original model}}$) of the model after each iteration of the loop in Algorithm 2. The results of this experiment for one pass of quantization of Algorithm 3 have been depicted in Figure 3.1b. As shown in the figure, the size of the model is reduced quickly during the initial iterations. This portion of the experiment corresponds to the part of Figure 3.1a where the error rate is constant. However, as quantization continues, the model experiences diminishing returns in size reduction. After 25 iterations, little reduction in model size is achieved. The lower density of data points, past this point, is due to the generated steps failing the test on the upper bound on the loss function in Algorithm 2.

Consequently, the algorithm reduces the trust region size and recalculates the tolerance steps.



(a) Error rate increases as the model is compressed when retraining is applied



(b) Convergence of quantization. Iterations refers to iterations of the loop in algorithm 2.

Figure 3.1: Quantization of LeNet-5 model trained on MNIST dataset

3.4 DISCUSSION

The generality of the proposed technique makes it adaptable to other model compression techniques. In this section, we review some of the most notable of these techniques and examine the relationship between their approaches and ours. Specifically, we will explain how the proposed approach subsumes previous ones and how it can be specialized to implement them in an improved way.

Pruning: In pruning, small model parameters in a trained network are set to zero, which means that their corresponding connections are eliminated from the network. Han, Mao, and Dally (2015) showed that by using pruning, the number of connections in the network can be reduced substantially. Adaptive Quantization confirms similar observations, that in a DNN most parameters can be eliminated. In fact, Adaptive Quantization eliminates 5.6×, 5.6×, and 5.3× of the parameters in the networks trained for MNIST, CIFAR, and SVHN, respectively. These elimination rates are lower compared to their corresponding values achieved by Deep Compression. The reason for this difference is that Deep Compression amortizes for the loss of accuracy due to pruning of a large

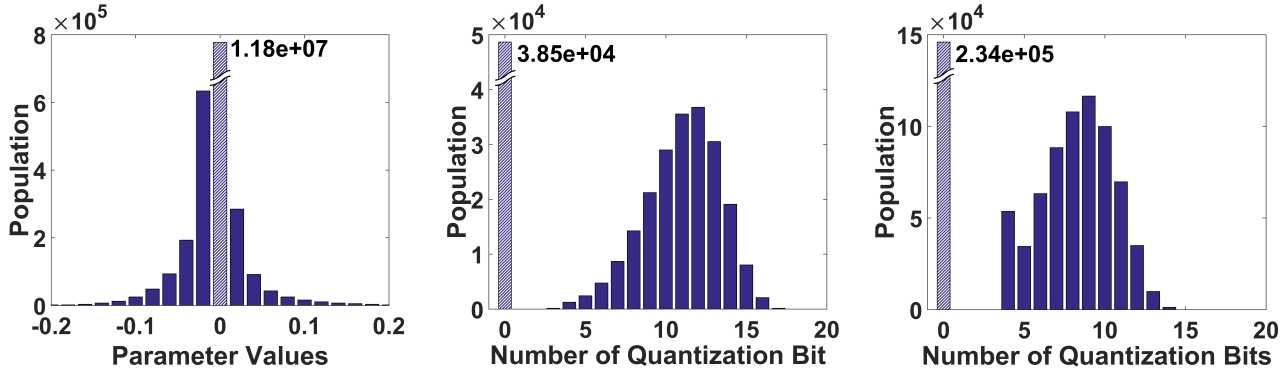
number of parameters by using full precision in the remaining ones. In contrast, Adaptive Quantization eliminates fewer parameters and instead quantizes the remaining ones.

While Adaptive Quantization identifies connections that can be eliminated automatically, Deep Compression identifies them by setting parameters below a certain threshold to zero. However, this technique might not be suitable in some cases. For an example, we consider the network model trained for CIFAR-10. This network has been trained using L_2 regularization, which helps avoid overfitting. As a result, the trained model has a large population of small-value parameters. Looking at Figure 3.2a, which shows the distribution of parameter values in the CIFAR-10 model, we see that most populate a small range around zero. Such cases can make choosing a good threshold for pruning difficult.

Weight Sharing: The goal of weight-sharing is to create a small dictionary of parameters (weights). The parameters are grouped into bins whose members will all have the same value. Therefore, instead of storing parameters themselves, we can replace them with their indexes in the dictionary. Deep Compression (Han, Mao, and Dally, 2015) implements weight-sharing by applying k-means clustering to trained network parameters. Similarly, Samragh, Ghasemzadeh, and Koushanfar (2017) implement weight sharing by iteratively applying k-means clustering and retraining in order to find the best dictionary . In both of these works, the number of dictionary entries are fixed in advance. Adaptive Quantization, on the other hand, produces the bins as a byproduct and does not make assumptions on the total number of bins.

Deep Compression also identifies that the accuracy of the network is less sensitive to the fully connected layers compared to the convolution layers. Because of that, it allocates a smaller dictionary for storing them. Looking at Figure 3.2b and Figure 3.2c, we can see that results of Adaptive Quantization are consistent with this observation. These figures show the distribution of quantization widths for one convolution layer and one fully

connected layer in the quantized CIFAR-10 network. It is clear from these figures that parameters in the fully connected layer, on average require smaller quantization widths.



(a) Distribution of parameter values after quantization.

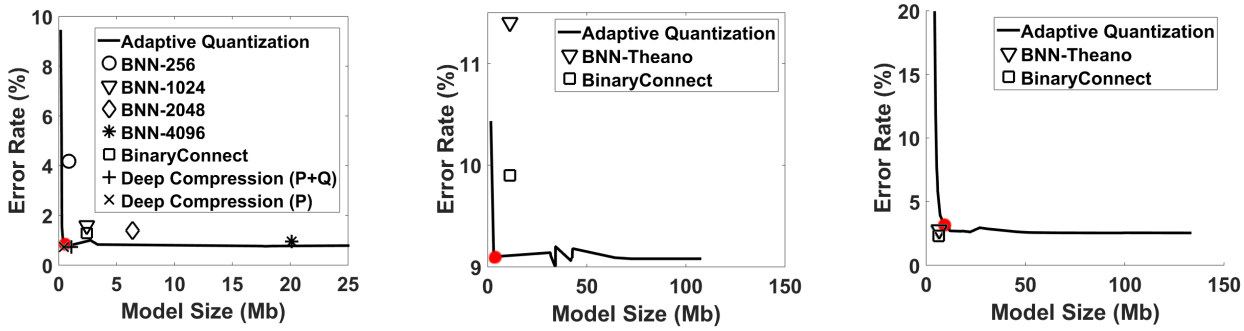
(b) Distribution of quantization widths in the first convolution layer.

(c) Distribution of quantization widths in the first fully connected layer.

Figure 3.2: Distributions of parameter values and quantization widths for the CIFAR-10 network. Pruned parameters have been depicted using a hatch pattern and their populations have been reported besides their corresponding bars.

Binarization and Quantization: Binarized Neural Networks (Courbariaux, Bengio, and David, 2015; Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016b) and Quantized Neural Networks (Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016a) can reduce model size by assuming the same quantization precision for all parameters in the network or all parameters in a layer. In the extreme case of BNNs, parameters are quantized to only 1 bit. In contrast, our experiments show that through pruning and quantization, the proposed approach quantizes parameters of the networks for MNIST, CIFAR-10, and SVHN by equivalent of 0.03, 0.27, and 1.3 bits per parameter with 0.12%, -0.02%, and 0.7% decrease in accuracy, respectively. Thus, our approach produces competitive quantizations. Furthermore, previous quantization techniques often design quantized training algorithms to maintain the same parameter precisions throughout training. These techniques can be slower than full-precision training. In contrast, Adaptive Quantization allows for unique quantization precisions for *each parameter*. This way, for a trained model, it can find a notably smaller network, indicating the limits of quantization. In addition, it does not require quantized training.

Next, we compare Adaptive Quantization with previous works on compressing neural network models, in reducing the model size. The results for these comparisons have been presented in Figure 3.3, which shows the trade-offs that Adaptive Quantization offers between accuracy and model size. As this figure shows, Adaptive Quantization in many cases outperforms previous methods and consistently produces compressions better than or comparable to state-of-the-art. In particular, we mark an optimal trade-off for each model in Figure 3.3 (red highlight). In these points, the proposed method achieves 64×, 35×, and 14× compression and correspond to 0.12%, -0.02%, and 0.7% decrease in accuracy, respectively. This always improves or is comparable to the state-of-the-art of Quantization (BNN and BinaryConnect) and Pruning and Weight-Sharing (Deep Compression). Below, we discuss these trade-offs in more details.



(a) MNIST (b) CIFAR-10 (c) SVHN

Figure 3.3: Trade-off between accuracy and error rate for benchmark datasets. The optimal points for MNIST, CIFAR-10, and SVHN (highlighted red) achieve 64×, 35×, and 14× compression and correspond to 0.12%, -0.02%, and 0.7% decrease in accuracy, respectively. BNN-2048, BNN-24096, BNN-Theano are from (Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016b); BinaryConnect is from (Courbariaux, Bengio, and David, 2015); and Deep Compression is from (Han, Mao, and Dally, 2015)

In both MNIST and CIFAR-10, Adaptive Quantization produces curves below the results achieved in BNN (Hubara, Courbariaux, Soudry, El-Yaniv, et al., 2016b) and BinaryConnect (Courbariaux, Bengio, and David, 2015), and it shows a smaller model size while maintaining the same error rate or, equivalently, the same model size with a smaller error rate. In the case of SVHN, BNN achieves a slightly better result compared

to Adaptive Quantization. This is due in part to the initial error rate of our full-precision model being higher than the BNN model.

Comparison of Adaptive Quantization against Deep Compression (when pruning is applied) for MNIST shows some similarity between the two techniques, although deep compression achieves a slightly smaller error rate for the same model size. This difference stems from the different approaches of the two techniques. First, Deep Compression uses full-precision, floating-point parameters while Adaptive Quantization uses fixed-point quantized parameters which reduce the computational complexity (fixed-point operation vs. floating-point operation). Second, after pruning the network, Deep Compression performs a complete retraining of the network. In contrast, Adaptive Quantization performs little retraining.

Furthermore, Adaptive Quantization can be used to identify groups of parameters that need to be represented in high-precision formats. In a more general sense, the flexibility of Adaptive Quantization allows it to be specialized for more constrained forms of quantization. For example, if the quantization requires that all parameters in the same layer have the same quantization width, Adaptive Quantization could find the best model. This could be implemented by solving the same minimization problem as in Equation 3.16, except the length of T would be equal to the number of layers. In such a case, G^k in Algorithm 2 will be defined as $G^k = [g_1^k ... g_m^k]^T$ assuming a total of m layers where g_i^k would be $\sum_{j \in J_i} |\frac{\partial \mathcal{L}}{\partial \omega_j}(W^k)|$, and J_i the set of parameter indexes belonging to layer i. Then, each of the resulting m tolerance values of the subproblem of section 2.3 will be applied to all parameters of one layer. The rest of the solution would be the same.

3.5 SUMMARY

Our approach aims to address the challenge of deploying deep neural network models on resource-constrained edge computing devices by reducing their size and complexity.

We achieve this through adaptive quantization, a technique that assigns an optimal precision to each network parameter to minimize the loss resulting from quantization while ensuring that critical parameters are represented with high precision. Unnecessary parameters are either pruned or assigned lower precisions to further reduce model size. We combine adaptive quantization with fixed-point computation methods such as SWAR and Bit-pragmatic computation to accelerate inference while maintaining high accuracy.

Our experiments on various benchmarks demonstrate the effectiveness of our approach. Adaptive quantization significantly reduces DNN model size without sacrificing accuracy, outperforming state-of-the-art quantization techniques. Moreover, our approach can achieve similar results to floating-point model compression methods. By minimizing data movement and simplifying computations, our approach enables efficient deployment of DNN models on edge devices.

In the next chapter, we explore issues related to quantization, precision, and numerical range for binary neural networks (BNNs) on bitstream computing. We present a baseline architecture for a BNN on a bitstream computing substrate and investigate the impact of quantization on model performance. Our findings will help inform the development of more efficient and accurate BNN models for edge computing.

4 IMPLEMENTING BNNS USING BITSTREAM COMPUTING

In this chapter, we will investigate the potential of implementing Bayesian Neural Networks (BNNs) on bitstream computing substrates and discuss the issues of precision and quantization. Bitstream Computing (BC) has a stochastic nature and is low-cost and power-efficient, which makes it a promising platform for deploying BNNs at the edge. The low power consumption of BC substrates is particularly advantageous for BNN computations, as many edge tasks requiring Uncertainty Quantification (UQ) have limited power and energy.

We will explore the case of audio classification for keyword spotting as a task that requires UQ at the edge. Keyword spotting is an essential task in audio processing, where systems such as Amazon Alexa, Google Assistant, etc., identify keywords and then start recording and analyzing further. However, this process raises concerns about privacy and security, as misclassification can result in the audio processing system recording the environment and leaking private conversations and data. A BNN can mitigate such risks by quantifying predictive uncertainty. If the system identifies a keyword with high uncertainty, it can prompt the user to repeat the keyword, thereby minimizing security risks.

To demonstrate the feasibility of using BNNs and BC for audio classification at the edge, we will present a design that leverages the strengths of these technologies. The rest of this chapter presents the details of our proposed design, our evaluation methodology, and our results. We also highlight the challenges of precision and quantization in implementing Bayesian neural networks in bitstream computing.

4.1 INTRODUCTION

Bayesian Neural Networks (BNNs) have the unique ability to effectively learn an infinite ensemble of neural networks, making predictions by averaging the outputs of a random subset from the ensemble and computing uncertainty as the level of disagreement between them. While the additional computations required for BNNs can increase the energy cost of inference, they are critical for many edge applications such as autonomous cars, robotics, wearable or implantable medical devices, and the Internet of Things, where Uncertainty Quantification (UQ) is crucial.

To address the challenge of implementing BNNs on low-power and energy-constrained devices, we propose the Bayesian Bitstream Processor (BBP), a hardware platform that uses Bitstream Computing (BC). We focus on the keyword spotting task from the MLPerf Tiny benchmark, which has been designed for benchmarking platforms for Tiny ML applications. We implement BBP using the BitSAD programming language, which allows us to simulate its operations and generate the Verilog code for evaluating its hardware.

Previous works on hardware for BNNs have been limited in scope. Fernandes *et. al.* (Fernandes et al., 2016) designed a BC circuit for shallow mixture models. VIBNN (Cai et al., 2018) and Shift-BNN (Wan, Xia, et al., 2021) focused on optimizing RNGs for efficiently generating random weight samples. Another work, FastBCNN (Wan and Fu, 2020) designed an accelerator for BNNs but it only applies to BNNs that use Dropout layers.

In contrast, we have designed an efficient hardware platform for deep networks with all-bayesian layers to allow UQ for a broad range of applications. Our contributions in this chapter include arguing that BC substrates can address the computation needs of BNNs, designing a hardware platform for deploying BNNs using BC to obtain low power and energy consumption, implementing the proposed hardware using the BitSAD

programming language, and evaluating its power and energy improvement over the MLPerf Tiny baseline.

4.2 OVERVIEW

Figure 4.1 1 depicts an overview of the proposed system based on bitstream computing. A controller manages the functions of the processor, providing the network parameters including the means and variances of the distributions of the BNN. Input from a microphone is received in the form of a bitstream and processed using a bitstream computing circuit which we refer to as the Bayesian Bitstream Processor (BBP). The network parameters from the controller are sent to the Gaussian Variate Generator (GVG) unit which produces random samples of the network parameter distributions and passes them to the BBP to use in the network's forward pass computations. The network output is sent back to the controller for decision making.

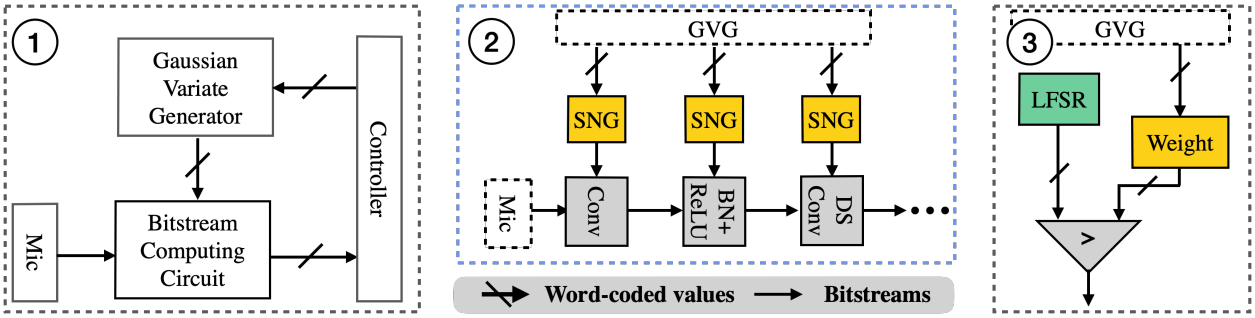


Figure 4.1: The proposed BNN platform. 1 overview of the architecture comprising the Gaussian Variate Generator (GVG) and Bayesian Bitstream Processor (BBP). 2 the BBP implementing DS-CNN. 3 Stochastic Number Generator (SNG) details.

We have assumed that the inputs arrive in the form of a bitstream. This is an acceptable assumption since many microphones generate Pulse-Density Modulated (PDM) outputs where values are represented in the density of pulses. In addition, we assume inputs are first processed by an onset detection system. As the microphone mainly records silence, the system only needs to be activated when input signals arrive. This is compatible the

tinyMLPerf benchmark which assumes input sounds are segmented before being processed by the classification mechanism. This also relaxes real-time processing constraints. We will later discuss how our system may achieve real-time processing of audio inputs.

The Gaussian Variate Generator produces Gaussian random samples for the BNN weights. We model this component after the one designed by (Alimohammad et al., 2008) which uses the Box Muller method (Golder and Settle, 1976) to generate random samples. It is capable of producing random samples at a rate higher than 1 Billion samples per second. This is fast enough to keep up with the BBP. The drawback of this method is that the Box Muller method does not have a long tail but in the case of BNNs this can be tolerated.

As specified in TinyMLPerf (Banbury et al., 2021), the BBP implements the DS-CNN (Sørensen, Epp, and May, 2020) network. This is a feed forward Convolutional Neural Network (CNN) that uses depthwise separable convolution layers, as shown in Figure 4.2. Next, we present the details of the implementation of this network in the BBP.

4.3 BITSTREAM COMPUTING CIRCUIT

We have depicted the details of the BBP in Figure 4.1 2 and 3. The BBP comprises the stochastic layers which implement the layers of DS-CNN in the bitstream computing domain. Each layer receives both activations and weights in the form of bitstreams. Activation bitstreams are directly received from the previous stochastic layer or the input on the left. Weight bitstreams are generated using Stochastic Number Generators (SNGs). Randomly-sampled, word-coded weights from the GVG are sent to the BBP and are received by Uniform Random Number Generators (URNGs) that convert them into stochastic bitstreams. At the end, the bitstreams are accumulated and the results are sent back to the controller.

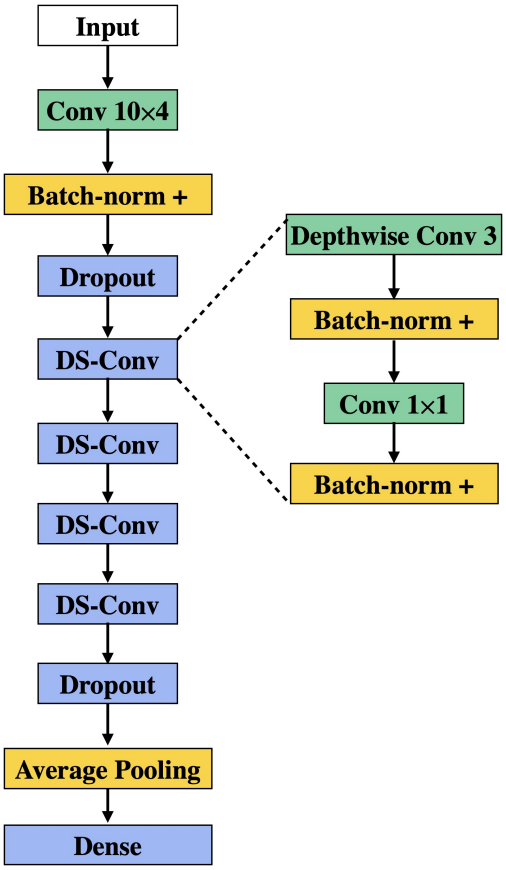


Figure 4.2: The architecture of DS-CNN

The SNGs receive the weight values from the GVG and produce stochastic bitstreams. They comprise 64-bit Linear-Feedback Shift Registers (LFSRs) and comparators. LFSRs (Gupta and Kumaresan, 1988) are widely-used random number generators that produce random values that are uniformly distributed and can be implemented cheaply. The comparator then generates the bitstream by comparing the random samples with the weight value from the GVG following a Bernoulli distribution.

We implement the BBP using the BitSAD programming language. BitSAD allows implementing the flow of the BBP, simulate the bitstream computing architecture, and generating the verilog code for it. The code for implementing and simulating one matrix multiplication has been depicted in Figure 4.3. Here, we specify the bitstreams for the feature maps and the filters using `SBitstream.()`. Then, using `generate.()` we can populate these bitstreams with randomly sample bits. We then define the function that

we plan to simulate for bitstream computing using `simulatable` which is then serially simulated. Here, we will discuss the functions we added to BitSAD in order to implement BBP as well as the details that need to be considered.

```
# bitstream length
T = 1000

# Create bitstreams
x = SBitstream.(activations)
w = SBitstream.(weights)

# Populate btstreams with stochastic bits
generate!.(x, T)
generate!.(w, T)

# Define the target function
# i.e. matrix multiplication
foo(x, w) = w * x
y = foo(x, w)

# Simulate bitstream computing
sim = simulatable(foo, x, w)
for t in 1:T
    push!.(y, pop!.(sim(foo, x, w)))
end
```

Figure 4.3: Matrix multiplication in BitSAD

Bitstream Generation: By default, the BitSAD's `generate.()` uses the software RNG which uses the Mersenne Twister, but we modify BitSAD to be able to specify custom LFSRs to source the random numbers.

Network Layers: BitSAD implements basic matrix operations as well as convolutions which are implemented using im2col followed by matrix multiplication. We also implement depthwise convolutions as well as average pooling layers that are used in DS-CNN. Similar to how regular convolutions are implemented in BitSAD, we use NNlib as the backend which allows us to maintain a similar syntax.

Scale Factors: When using bitstream computing in BitSAD, it is important to account for the range of parameters. BitSAD only represents parameters in the $[-1, 1]$ range. When an operation produces a bitstream with a value outside of this range, the bitstream saturates. To avoid saturation, inputs need to be appropriately scaled. The output is then multiplied by the inverse of the scale factor to produce correct results. In a matrix

multiplication, if one of the matrices is known beforehand and the other is in the BitSAD range, these scale factors can be precomputed (Shukla et al., 2018).

For each layer of DS-CNN, we follow the same method to compute the proper scale factor. Since BitSAD convolutions use the im2col method, we compute the scale factors for these layers according to the rearranged filter matrix. Furthermore, since instead of a deterministic CNN, we are implementing a BNN, we use the mean value of each parameter to compute the scale factors. We adjust the mean and variance of these parameters accordingly. The scale factor for each layer is then incorporated in the following batch normalization layer to compute the correct output.

It is important here to note that scaling parameters for the bitstream computing substrate proportionally reduces precision. Therefore, in order to recover the same precision, we would have to increase the length of the bitstream by the same factor. In particular, we have observed that the scale factor in many BNNs and DNNs can range from $100\times$ to $1000\times$. As delay and energy are linearly related to the bitstream length in bitstream computing substrates, this can significantly hinder application of the design to larger BNN models. Still, in chapter 7, we will evaluate the present design and show that while scaling can result in additional delays and energy costs, this design can outperform previous designs in the keyword spotting tasks in energy and power, with comparable or superior delay. In the coming chapters, we will address this inefficiency in detail using various design and algorithmic techniques.

4.4 SUMMARY

In this chapter, we presented an energy-efficient platform for deploying BNNs at the edge using BC. We argued that BC is uniquely suitable for BNNs due to its similarity in computations and low power and cost. Our platform was implemented using the BitSAD programming language, and we evaluated its power and energy consumption using an

audio classification case study, which is a security-critical application. In chapter 7, we will experimentally demonstrate that our platform outperforms other baselines by up to two orders of magnitude in energy and an order of magnitude in speed, highlighting the advantages of using BC for deploying BNNs at the edge.

However, we also argued that the straightforward implementation of BNNs on bitstream computing substrates poses challenges in terms of precision and quantization. All values in bitstream computing need to be represented in the $[0, 1]$ range, and values outside this range need to be scaled, leading to the need for longer bitstreams, which can add delay and energy. In the following chapters, we will discuss training Bayesian neural networks for deployment on bitstream computing substrates and propose new BC designs for BNNs to achieve higher performance and efficiency.

5 TRAINING BNNS FOR BITSTREAM COMPUTING

This chapter discusses the challenges of implementing BNNs on bitstream computing substrates and proposes approaches for training BNNs suitable for deployment on such hardware. It explores the implementation of non-linear activation functions in bitstream computing and suggests piece-wise linear activation functions as better candidates for BNNs on such substrates.

5.1 INTRODUCTION

In the previous chapter, we explored the design of an energy-efficient platform for deploying BNNs at the edge using bitstream computing by introducing the Bayesian Bitstream Processor (BBP). We demonstrated that BC is uniquely suitable for BNNs due to its low power and cost, and we presented a case study using an audio classification case study. However, we also showed that straightforward implementation of BNNs on bitstream computing substrates poses challenges in terms of precision and quantization.

Bitstream computing substrates require values to be represented in the $[0, 1]$ range and, when this condition is not satisfied, that they be scaled down. This in turn can cause precision errors. Therefore, BNNs trained without consideration for the intended inference substrate can result in limitations such as accuracy drops, unreliable uncertainty quantifications, longer delays, and higher energy consumption in the bitstream computing substrate. In this chapter, we address these challenges by discussing approaches for training BNNs for deployment on bitstream computing substrates.

In this chapter, we will delve into the specifics of training Bayesian neural networks (BNNs) for deployment on bitstream computing substrates. We will begin by highlighting the key differences between BNNs and conventional deep neural networks in terms of their architectures and training methodologies. Then, we will focus on the critical role of activation functions in training BNNs suitable for bitstream computing hardware. We will explore how non-linear activation functions can be implemented in bitstream computing. Additionally, we will explore piece-wise linear activation functions as better candidates for BNNs on bitstream computing substrates and how to train them.

5.2 COMPONENTS OF DNNS AND BNNS

Conventional, deterministic deep neural networks and Bayesian neural networks share many key components as both employ deep learning architectures. These similarities include convolutional layers, fully connected layers, pooling layers, skip connections, etc. However, several components of design and training commonly used in DNNs are inherently incorporated into BNNs as they utilize Bayesian principles. Below, we take a brief look at these.

Regularization: In order to avoid overfitting to the training data, DNNs often incorporate into their training loss function a regularization element like l1, l2, etc. However, Bayesian approaches are often robust to overfitting (Hernández-Lobato and Adams, 2015). Therefore, during training of a BNN, we do not include such a term. Instead, we usually use the Kullback-Leibler divergence of the weight densities which results in a loss function often referred to as the variational free energy (Neal and G. E. Hinton, 1998).

Dropout: Very often DNNs incorporate dropout layers which randomly sets input elements to 0 and can provide several advantages, including remedying overfitting and regularizing the network (Molchanov, Ashukha, and Vetrov, 2017; Srivastava et al., 2014). It has also been used as a way to estimate the uncertainty in DNN models (Gal and

Ghahramani, 2016; Laves et al., 2020). As they essentially try to approximate the same posterior. Furthermore, it has been shown that dropout can be overly confident when approximating uncertainty compared to a BNN (Yao et al., 2019).

Batch Normalization: Batch Normalization is a widely-adopted technique that helps faster training by controlling the distribution of the layers and smoothing the gradients (Santurkar et al., 2018). Furthermore, it has been suggested that it creates a model equivalent to a Bayesian one and can estimate uncertainty using the variability of samples in a batch (Teye, Azizpour, and Smith, 2018). As such, it can be redundant in a BNN.

5.3 ACTIVATION FUNCTIONS IN BNNS

Regularization techniques can be used to encourage weights in a neural network to fit within a desired range, such as the (0,1) range required for bitstream computing substrates. However, regularization techniques may not be as effective in controlling the range of activations, which can be a challenge when using activation functions like ReLU that produce unbounded outputs. One solution is to replace ReLU with bounded activation functions like sigmoid or tanh that can produce activations within the desired range. However, implementing these non-linear activation functions in bitstream computing substrates can be challenging, as they require additional computational resources and may introduce precision errors.

There are a number of ways that we can approximate transcendental functons in Bitstream Computing. Previous studies have implemented transcendental function using Finite State Machines (Y. Liu et al., 2020). The architecture for the FSM-based design has been depicted in Figure 5.1. Here, the Lookup Table (LUT) stores the input and output values of a tanh function. Then, based on the difference between input bitstream and the x value of the LUT, the LUT index is adjusted. On the output side, the difference between

the output bitstream generated in the previous cycles and the current y value of the LUT is stored in a counter which is used to generate a new bit for the output bitstream.

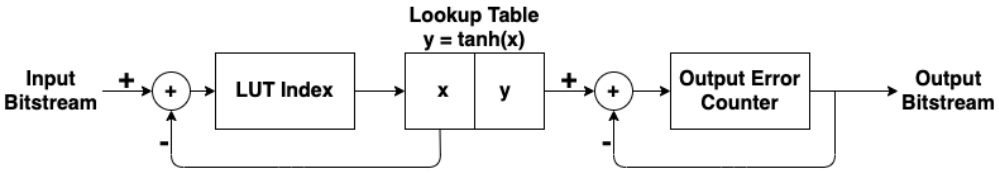


Figure 5.1: Design of tanh function in bitstream computing substrate

Another way to achieve the same effect is to use the Fourier expansion of these activation functions. This significantly simplifies the design since we can hard-code the first few components of the function into the bitstream computing substrate. In Figure 5.2, we compare the two approaches. This figure depicts the approximation error for the Taylor series with three components as well as the error for the LUT-based implementation with LUTs of different sizes. Here, N represents the size of the LUT and the T axis represents the length of the bitstream. For all cases, as the length of the bitstream increases, the error is reduced until it plateaus. Furthermore, while for short bitstreams the Taylor series performs well, for all LUT-based implementations with a LUT size bigger than 2, the LUT-based method has a better final error. Thus, with a small LUT, we can easily outperform the Taylor series implementation. We would also like to note that the LUT based implementation can easily produce any activation function while the Taylor series method needs to be reimplemented if the activation function is changed.

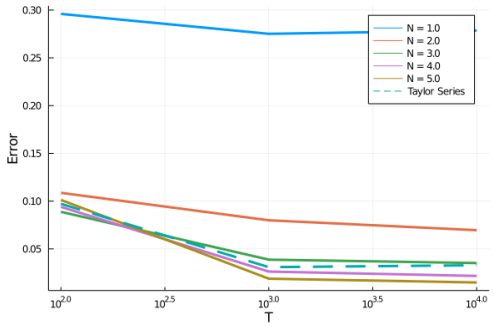


Figure 5.2: Comparison of implementation of tanh with Taylor series vs FSM. N is the size of the LUT.

However, each of these two approaches has its own limitations. The FSM-based approach has the drawback of taking up additional memory and creating dependence between bits in a bitstream which can be undesirable. On the other hand, the method based on the Taylor series expansion has higher error. It is therefore important to explore alternative, algorithmic approaches.

Rather than attempting to implement non-linear activation functions on bitstream computing substrates, a potential solution is to address this challenge algorithmically. One approach is to use piece-wise linear activation functions that can effectively bound the activations within the desired range while still being easily applicable to bitstream computing substrates. For instance, Gulcehre et al. (2016) explored the use of piece-wise linear activation functions, including piece-wise linear tanh, for training deep neural networks. In the tanh case, this approach involves only a clipping operation, making it well-suited for bitstream computing substrates. However, using such functions may result in zero-gradients in wide sections of their domains, resulting in sub-optimal final accuracies. To address this issue, the authors added noise to the output of these functions during training to encourage exploration by gradient descent. This approach has shown to achieve comparable or even superior performance.

The piecewise linear tanh (shown below) is easily implemented on bitstream computing substrates as bitstreams automatically clip values.

$$pwl - tanh(x) = max(min(x, 1), -1)$$

This function is naturally implemented by bitstream computing substrates without any additional circuits necessary and simplifies the design compared to a ReLU. Furthermore, keeping the activations withing the $[-1, 1]$ range removes the need for scaling. This will in turn improve computation precision which alleviates the need for long bitstreams meaning lower delays and energy. Figure 5.3 depicts the distribution of the activations for the keyword-spotting task with clipping (i.e. pwl-tanh) and ReLU.

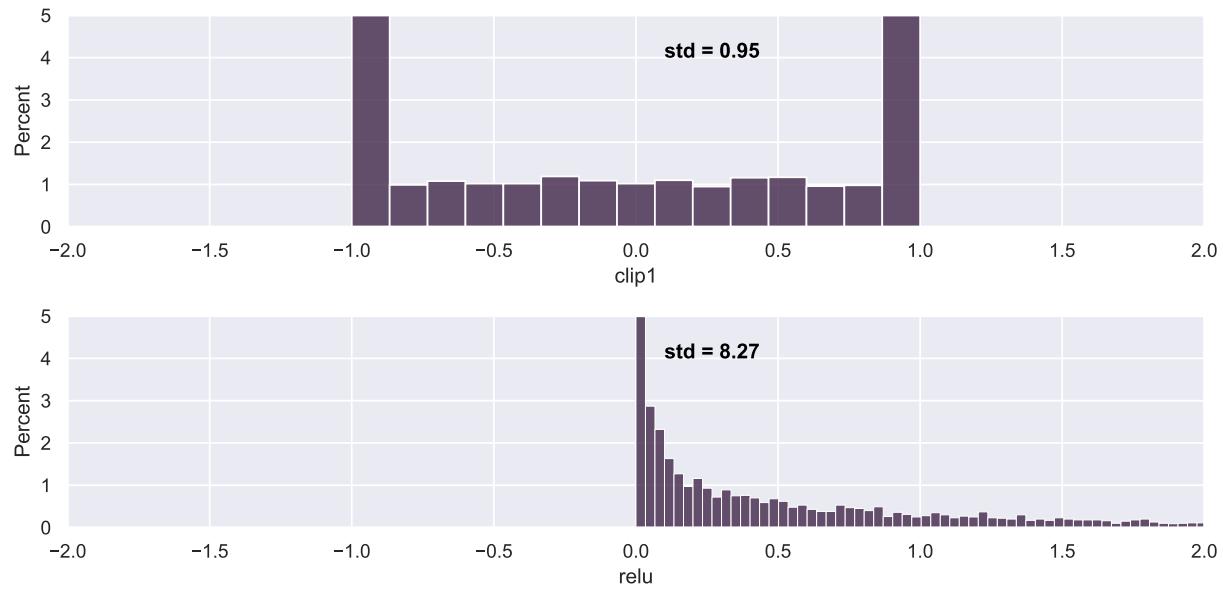


Figure 5.3: Distribution of activation values for piecewise tanh (clip1) and relu

As ReLU requires scaling of the inputs by about $10\times$, by replacing the activation function we achieve an order of magnitude delay and energy improvement. We will present results in chapter 7.

5.4 TRAINING FLOW

We incorporate the training flow described by Gulcehre et al. (2016) into the typical training of BNNs. Like a conventional BNN training flow, we use standard normal distributions for the weights and learn Gaussian posteriors by learning means and variances. As explained earlier, we use the KL-divergence between these two distribution as regularization. The coefficient for this regularization term is the inverse of the size of the training set, as is commonly done, and typically use a small learning rate. For training with noisy activation functions, we add a standard normal noise. Similar to the original work, this noise is annealed by a coefficient schedule of $\frac{c}{\sqrt{t+1}}$. Where c is the initial value of the coefficient and t is the training epoch. We remove this noise during inference. We

will summarize the hyper-parameter used in Table 7.1 in Chapter 7. Training for each task was done using the tensorflow (Martıén Abadi et al., 2015) with Adam (Diederik P Kingma and Ba, 2014) optimizer. For implementing BNNs, we used tensorflow-probability (Dillon et al., 2017) which provides Bayesian version of the layers of a Convolutional Neural Network. Finally, we used categorical cross-entropy loss for training classification tasks and the mean squared error loss for regression tasks.

5.5 SUMMARY

In this chapter, we explored the challenges of training BNNs for bitstream computing, taking into account their differences from DNNs and the limitations of the underlying hardware. One significant challenge we highlighted is the implementation of activation functions on bitstream computing substrates. While we discussed various approaches to implementing standard activation functions compatible with bitstream computing, each method presented unique challenges. Therefore, an algorithmic approach is the best way to overcome these issues. We have adopted piece-wise linear tanh activations, which have been demonstrated to perform well on DNNs, to address these challenges. In the next chapter, we will discuss another obstacle to deploying BNNs: generating random numbers with high throughput.

6 STOCHASTIC BITSTREAM GENERATION AND EARLY STOPPING

In the chapter 4, we introduced the Bayesian Bitstream Processor (BBP) and identified two major challenges in deploying Binary Neural Networks (BNNs) on it: generating random samples for all BNN weights and computational efficiency. In this chapter, we propose a solution to address both of these challenges. Firstly, we will present the Gaussian Stochastic Bitstream Generator, which can generate bitstreams that follow a Gaussian distribution for BNNs using Linear Feedback Shift Registers (LFSRs), which are already present in bitstream computing substrates. Not only does this design allow for high-throughput generation of bitstream with a Gaussian distribution, but also allows for concurrent computations for multiple BNN samples at a time. The concurrent computations enable us to begin computing predictive uncertainty immediately, improving the estimate as more bits are received in the output. If the low-precision, early estimates are low or high enough, we can make decisions on whether to reject or accept a sample and make an early prediction, stopping the computation and only performing high-resolution computations for inputs that are harder to predict. This approach provides computational efficiency.

6.1 INTRODUCTION

As we discussed in chapter 1, inference in Bayesian Neural Networks requires generating random samples with a high throughput. We further designed the Bayesian Bitstream Processor in which we approached this challenge by incorporating a dedicated unit, called the Gaussian Variate Generator that generated all the samples for BNN. This solution

might be limited as larger neural networks introduce heavier loads for this unit. In this chapter, we will address this limitation by taking advantage of the Linear-Feedback Shift Registers already present in bitstream computing substrates to generate Gaussian random samples. The key idea is based on the central limit theorem which indicates that the average of the LFSRs will follow a Gaussian distribution.

To design the target bitstream generator with a Gaussian distribution, we will take a closer look into our bitstream computing substrate. The order of operations in BBP were as follows:

1. Have the Gaussian Variate Generator generate a set of random Gaussian samples, one sample for each weight.
2. Generate bitstreams for the all network weights.
3. Perform a forward pass for the generated bitstreams and compute the output for one BNN pass.
4. Repeat steps 1 to 3 for all the subsequent BNN passes and store the results for all passes.
5. Compute prediction and uncertainty and report the outcome.

This introduces a key bottleneck in serializing the computation of BNN passes. BBP controller therefore needs to maintain the word-coded value of the output for each pass and perform the variance computations at the end. The change of domain is not desired as it adds additional buffer and computation that is unnecessary. We can imagine alternative processes where all BNN samples are processed concurrently. Computing bits of the bitstreams for all BNN samples concurrently would allow us to immediately start computing an estimate of the output prediction and uncertainty and progressively improve our approximation as we obtain more and more bits. We suggested a solution capable of concurrent computation previously, taking advantage of the GVG unit in BBP being able to support multiple copies of the DS-CNN bitstream computing implementation. The

limitation of such a solution is that it adds to the power cost and, for larger networks, may not be feasible.

To achieve concurrency, therefore, we cannot rely on the centralized Gaussian random sample generation. We will instead design a low-cost Gaussian sample generation process for bitstream computing that can be replicated for all network weights. This design has the added advantage of enabling anytime computing (Zilberstein, 1996), a feature of bitstream computing that allows for efficient BNN computations. Compared to the originally proposed Bayesian Bitstream Processor (BBP), this streamlined design allows us to begin computing the output variance immediately. By using early estimates, we can make faster decisions when possible and avoid unnecessary computations to achieve efficient BNN computations. We will take a closer look at the serialization bottleneck in BBP below and delve deeper into concurrency as well as early stopping.

6.2 CONCURRENCY

Similar to conventional computing architecture, concurrency allows bitstream computing to reduce the costs of functional units. But more than that, for deploying BNNs, it allows us to compute uncertainty in an *anytime computing* fashion (Zilberstein, 1996). That is, as soon as we observe a small number of bits at the output, we can start approximately computing predictive uncertainty, and as we observe more bits, we improve our approximation. By interleaving bitstreams and performing all computations for all BNN network samples concurrently. In addition, concurrency allows for signals flowing through the compute pipeline to share functional units. Figure 6.1 illustrates this for bitstream computing where concurrency is enabled by interleaving two or more bitstreams. Each bitstream x_i comprises an array of bits like $[b_1^i, b_2^i, ..., b_T^i]$ where $b_i \in \{0, 1\}$. Therefore, formally, the interleaved bitstream $\chi = [b_1, b_2, ..., b_{mT}]$, where

$$b_t = b^{t\%m+1}_{\lfloor \frac{t}{m} \rfloor}$$

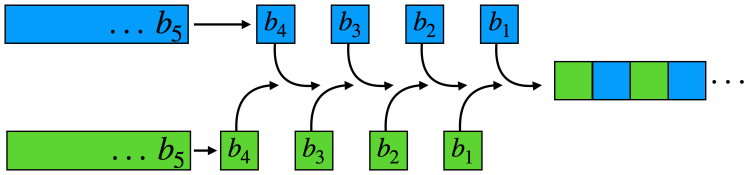


Figure 6.1: Two concurrent bitstreams

It is however important for concurrency to be implemented carefully. Bitstream computing spreads computations over time which creates opportunity for concurrent computations to corrupt one another. This can happen when the output bit of a functional unit depends on past inputs.

The *square* operation ($f(x) = x^2$) is an illustrative example of such an operation. For the multiplication operation needed for squaring, the inputs needs to be uncorrelated. As such, we need to generate decorrelated copies of a bitstream with the same nominal value. When dealing with one bitstream, this is easily done by delaying the bitstream by one clock cycle, $b_t^{\text{output}} = b_{t-1}^{\text{input}}$ like in Figure 6.2. The resulting bitstream is going to be uncorrelated from the original since each bit in the bitstream has been independently sampled. As such, the output of the multiplier (AND gate) will be $b_t^i b_{t-1}^i$ with the expected value:

$$\mathbb{E}(b_t^i b_{t-1}^i) = \mathbb{E}(b_t^i)\mathbb{E}(b_{t-1}^i) = x_i^2 \tag{6.1}$$

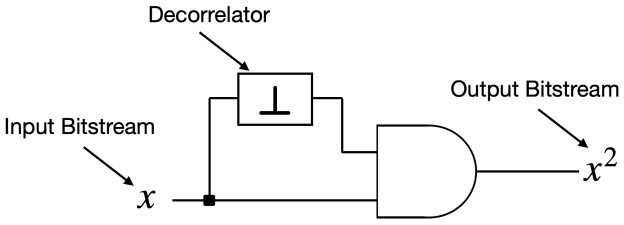


Figure 6.2: Square operation in bitstream computing

Thus a single-bit shift-register can be used as a decorrelator. This allows for squared function to be implemented like in Figure 6.2. When dealing with interleaved bitstreams, it is important to make sure the time-shifted output of the decorrelator from one bitstream does not intercept with another bitstream. We need to use a larger shift-register. In fact, the size of the shift-register needs to be equal to the number of the interleaved bitstreams, $b_t^{\text{output}} = b_{t-m}^{\text{input}}$. This way the output of the decorrelator will align correctly with the original interleaved bitstream. This allows us to compute running variance and uncertainty for interleaved bitstreams. We show the output of the squaring unit for interleaved bitstreams in Figure 6.3.

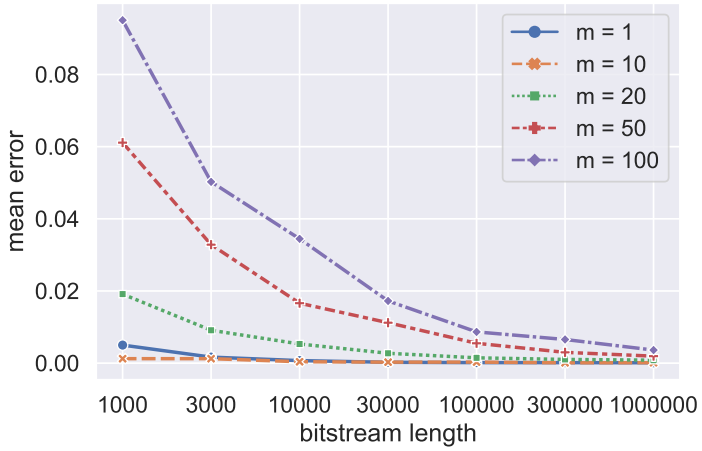


Figure 6.3: Average error of squaring operation on interleaved bitstreams

As this figure shows increasing the number of interleaved bitstreams m increases the error since each bitstream will be represented by fewer bits. But, increasing the overall number of bits improves the error.

This squaring operation allows us to compute running variance and uncertainty for interleaved bitstreams. Let us look at computing variance for interleaved bitstreams as an example. To compute variance, we follow $var(x) = \mathbb{E}(x^2) - \mathbb{E}^2(x)$ and compute $\mathbb{E}(x^2)$ and $\mathbb{E}^2(x)$ and then subtract them in floating-point format. The expectation is computed simply by averaging all the bits.

$$\mathbb{E}(\chi) = \frac{1}{m}\sum_{i=1}^{m} x_i = \frac{1}{mT}\sum_{i=1}^{m}\sum_{t=1}^{T} b_t^i$$

The second moment is computed the same way, only after using the square unit described above. We can therefore compute a crude estimate of the variance after observing at least m bits. As we observe more bits streaming through we can improve our approximation. The main issue that we need to address is how to generate interleaved bitstreams. Interleaving m bitstreams that are flowing through the bitstream computing circuit in parallel would require the interleaved bitstream to have a frequency m times higher than the original ones. We will address this issue by generating the bitstreams already interleaved from the beginning. In the rest of this chapter, we will discuss the process to achieve bitstreams that can be decomposed into m bitstreams whose values are sampled from a Gaussian distribution.

6.3 GAUSSIAN STOCHASTIC BITSTREAM GENERATOR

Our key design goal is to produce stochastic bitstream for the BNN parameters cheaply which would reduce costs and relax bottlenecks as discussed previously. Here, we will first discuss the key components of the hardware for the Gaussian Stochastic Bitstream Generator (GSBG) and how our choice helps achieve this goal. We then discuss the theoretical backing of our design and its characteristics.

6.3.1 *Hardware Random Number Generator*

Random sample generation can vary in complexity significantly from very simple to very complicated (Figure 6.4) and there exists plethora of techniques for producing random numbers in software and hardware (Kietzmann, Schmidt, and Wählisch, 2021). While

simpler methods are going to be easier to implement, they are bound to possess fewer desirable statistical qualities. High-quality methods on the other hand are naturally costly. As the costs directly depend on the available computing hardware, it is essential that the technique employed does not just provide good statistical guarantees but is also well-suited for the available hardware. We may take a look at various methods for generating random numbers to get a clearer idea.

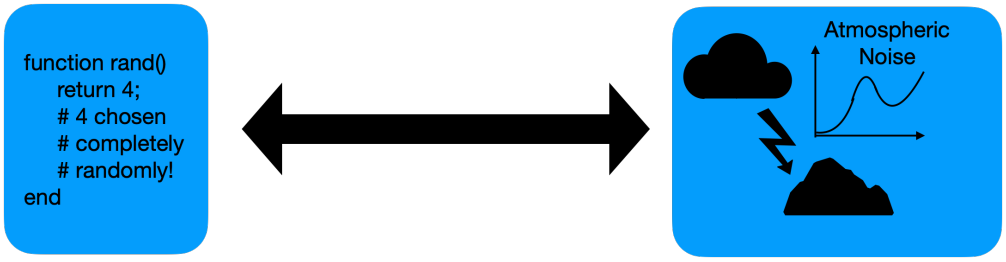


Figure 6.4: Random number generators can vary in complexity significantly!

It is, perhaps surprisingly, not very easy to generate random samples from a uniform distribution. Due to the deterministic nature of computer hardware, random number generation architectures are designed to output streams of bits that mimic randomness. This means the frequency of each possible output should be the same. This is a necessary but not sufficient condition as a m-bit random number generator that outputs all possible m-bit values in sequence satisfies it but is clearly producing not a random pattern. The quality of a Uniform Pseudo-Random Number Generator (UPRNG) is determined by randomness tests which, somewhat paradoxically, look for complex and rare patterns, and compare their frequency to their expected probability (Sys and Riha, 2014). These tests include looking for specific long patterns, number of ascending or descending outputs in a row, or even playing games (Alani, 2010). Not all UPRNGs pass all these tests. For example, the Mersenne Twister (Matsumoto and Nishimura, 1998) which is often used by many software applications passes many well-known test suites, but it has high memory requirements and low throughput. On the other hand Xorshift (Marsaglia, 2003) requires less memory and computations but also passes fewer statistical tests (O'neill, 2014). The

optimal choice of the random number generator then has to take the requirements of the application and the hardware capabilities into account.

Uncertainty quantification requires computing variance of the output posterior distribution and does not need to capture single rare events. This allows for simpler RNG design. It is therefore ideal if we can exploit the UPRNG circuits that bitstream computing architectures already use to generate their stochastic bitstreams. Bitstream computing substrates often use what is called a Linear-Feedback Shift Register (LFSR), shown in Figure 6.5.

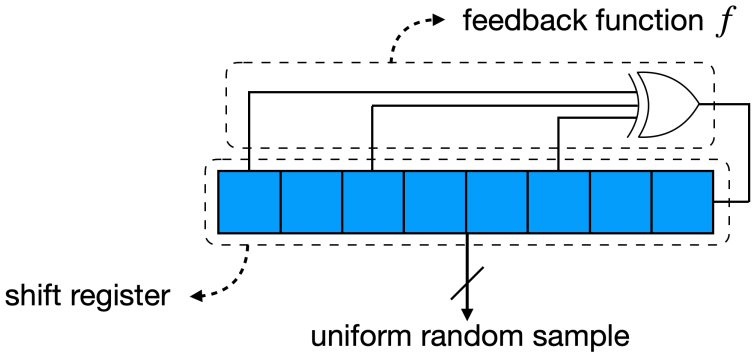


Figure 6.5: Architecture of a linear-feedback shift register

A LFSR comprises a shift register that stores the m-bit *state* of the random number generator. Each cycle, the state is shifted left one and a new bit is inserted to the least significant position whose value is a function of the old state:

$$\text{state}^{\text{new}} = clip(\text{state}^{\text{old}} \ll 1 + f(\text{state}^{\text{old}})) \tag{6.2}$$

Where the $clip$ function eliminates the most significant bit of the state variable and the function f is a feedback operation which computes the exclusive OR between select bits of the state register. The choice of the function f is key in designing a LFSR to provide guarantees for the random number generator.

As the equation 6.2 shows, the state of a LFSR at each cycle depends on the previous state. As such, if a state is repeated at any point during the operation of the LFSR,

the same sequence of outputs repeats. In order for the output sequence to cover all possible m-bit combinations, the function f is chosen carefully. Without getting into detail, this involves constructing the characteristic polynomial of f and proving that it is an irreducible and primitive polynomial. This can be done by brute-force search and there exist lists of precomputed maximum-period LFSRs for various m's.

The periodic sequence of output of a LFSR also means that if two m-bit LFSRs start from the same initial state, they will always produce the same sequence of numbers and are therefore perfectly correlated. To avoid this issue we need to make sure that our LFSRs are initialized to different values.

It is possible to further diminish the effect of determinism by choosing different m for different LFSRs, better yet if we choose m's that are relatively prime, the overall period of the system will be the product of the sequences of all LFSRs. This can be important for very large BNNs but in cases that are of interest here this is unnecessary. Note that there exist at least $67,000,000$ different functions f that would allow for a 32-bit maximum-period LFSR, each of which would have 2^{32} possible initializations (Koopman, n.d.).

We should also note that we can artificially change the period of a maximum-period m-bit LFSR to any value like $n < 2^m$, by resetting the state after n cycles. This is a feature we will take advantage of to design our proposed Gaussian stochastic bitstream generator.

6.3.2 *Bates and Distribution of a Bitstream*

Figure 6.6 depicts the architecture of a Stochastic Bitstream Generator in more detail. As we discussed before, the reference or nominal value of the bitstream is stored in a register that is connected to a comparator which compares the absolute value against the output of the LFSR. In each cycle, based on the result of this comparison and the sign of the reference value, one result is sent out and subsequently the LFSR is updated. The output

can be 1, 0, or -1. Similar to Daruwalla *et. al.* (Daruwalla and Zhuo, 2019), bitstreams carry ternary bits that comprise two binary bits for positive and negative values. The observed value is obtained by computing the mean of the bits in the output.

$$x_{observed} = \frac{1}{T}\sum_{t=1}^{T} b_t$$

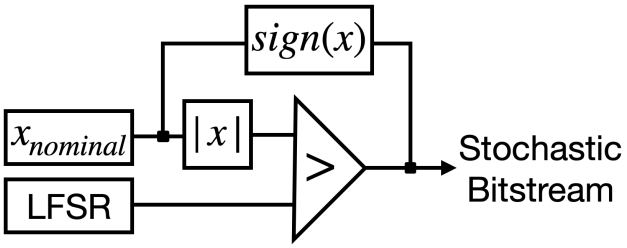


Figure 6.6: Architecture of a stochastic bitstream generator

Where like before, T is the size of the bitstream, b_t is the output bit which is 1 with the probability $x_{nominal}$. Since b_t are generated with the same probability, the observed output value $x_{observed}$ follows a binomial distribution. Therefore, we refer to this design as the Binomial Stochastic Bitstream Generator (B-SBG)

$$x_{observed} \sim \mathbb{B}(T, x_{nominal})$$

This is assuming $x_{nominal}$ is a constant. But, we can choose it to be a random variable itself. In fact, if $x_{nominal}$ is also a randomly sampled from a uniform distribution, we can show that the observed value will approximate a Gaussian distribution. To show this, we get help from *Bates* distribution.

The Bates distribution denotes the probability distribution of the average of m independent uniform random variables on the unit interval.

$$\chi = \frac{1}{m}\sum_{k=1}^{m} \mathbb{U}_k$$

Using the central limit theorem, it can be shown that as the value of m grows the Bates distribution approaches a Gaussian distribution.

$$\chi \sim \mathcal{N}(\frac{1}{2}, \frac{1}{12m})$$

With some minor adjustments, we can easily produce a standard Gaussian distribution. Instead of picking the uniform samples from $\mathbb{U}(0,1)$, we sample $\mathbb{U}' = \mathbb{U}(-\frac{1}{2}, \frac{1}{2})$ to shift the mean to 0. We further replace $\frac{1}{m}$ with $\sqrt{\frac{12}{m}}$.

$$\chi = \sqrt{\frac{12}{m}} \sum_{k=1}^{m} \mathbb{U}' \sim \mathcal{N}(0,1)$$

Now we can see how using a uniform random sample for $x_{nominal}$ can help generate a Gaussian sample. We will discuss the architecture of the proposed Gaussian Stochastic Bitstream Generator and analytically show how it produces Gaussian random samples.

6.3.3 *Architecture*

We make minor changes to the architecture of B-SBG to achieve the desirable effects. Figure 6.7 highlights the new components. First, the register where $x_{nominal}$ was stored has been replaced by another LFSR. This LFSR is associated with a register which stores the initial state which we can use to reset the LFSR at any point. This register will come in handy shortly. The new LFSR is what is going to generate random samples for $x_{nominal}$. We refer to the new LFSR as the Reference LFSR (R-LFSR) and the LFSR that existed in the original B-SBG to generate bitstreams as the Bitstream LFSR (B-LFSR). Since the R-LFSR is going to generate random samples from $\mathbb{U}(0,1)$ instead of U', we need to modify the comparator to account for an input offset of $-\frac{1}{2}$.

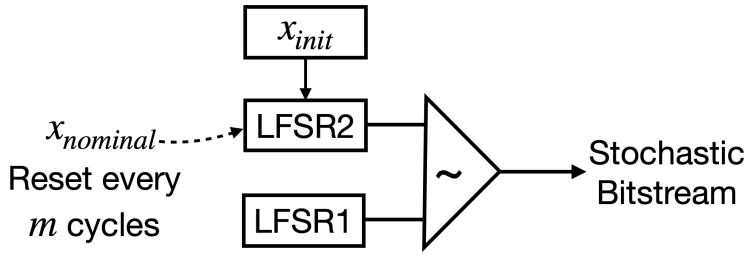


Figure 6.7: Architecture of the proposed Gaussian stochastic bitstream generator

The Figure 6.8 depicts the modified comparator design. Note that here since $x_{nominal}$ comes from a LFSR, it is always positive. Therefore, to generate a zero-mean output, we subtract -0.5. In summary, the comparator output is as follows:

$$b_t = sign(x_t - y_t - 0.5)$$

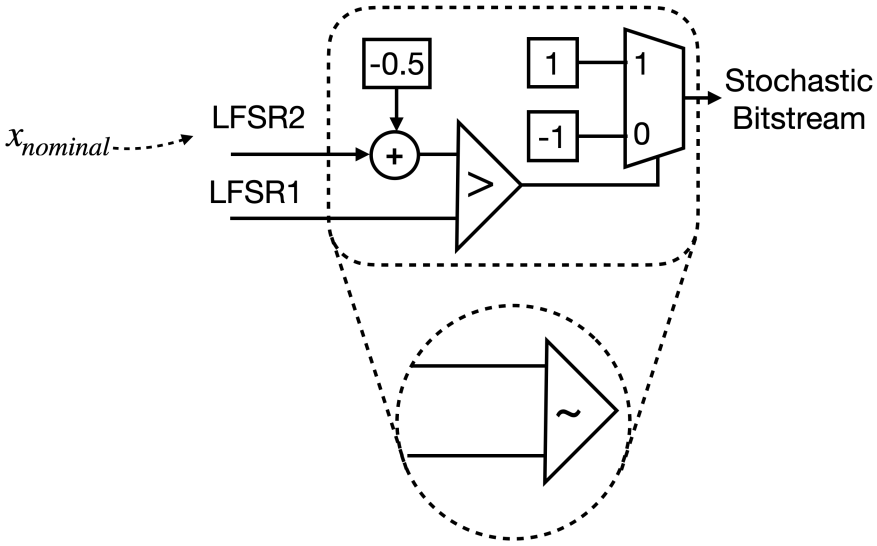


Figure 6.8: Comparator design for GSBG

Here, $sign$ is the sign function, x_t is the random sample for $x_{nominal}$ at time t and y_t is the random uniform sample from the original LFSR.

Note that both x_t and y_t are independent random samples from a standard uniform distribution. As such, their difference follows a standard triangular distribution.

$$\delta = x - y \sim \mathbb{T}_{\delta}(-1, 1) = \begin{cases} \delta + 1 & -1 < \delta < 0 \\ 1 - \delta & 0 \leq \delta < 1 \end{cases} \tag{6.3}$$

This would result in the output of the comparator to have always have the same probability. This is why we reset the R-LFSR. As such, all $x_{nominal}$ values are sampled at the beginning when the initial state of the R-LFSR is picked. However, from the perspective of B-LFSR, they are constants.

If the R-LFSR was indeed constant and we never updated its state, the generated bitstream would resemble a bitstream with the nominal value sampled from $\mathbb{U}'$.

$$x_t = x_0 = x_{nominal} \sim \mathbb{U}'$$

Furthermore, assuming we do update the state of R-LFSR while resetting it every m cycles, the observed bitstream will be an interleaved bitstream of m samples from $\mathbb{U}'$. This is because the reference value in R-LFSR will be the same every m cycles.

$$x_t = x^k_{nominal}, \quad k = t \,\%\, m + 1$$

Since all these m reference values are samples from $\mathbb{U}'$, the mean of the bitstream which adds all m values together will follow a Bates distribution or, as we saw earlier, an approximate standard Gaussian distribution.

Let,

$$\chi = \frac{1}{T}\sum_{t=1}^{T} b_t = \frac{1}{T}\sum_{k=1}^{m}\sum_{i=0}^{\frac{T}{m}} b_{i.m+k} = \frac{1}{T}\sum_{k=1}^{m}\frac{T}{m}x^k_{nominal} = \frac{1}{m}\sum_{k=1}^{m} x^k_{nominal} \sim \mathcal{N}(0, \frac{1}{12m})$$

Like before, if we replace $\frac{1}{T}$ with $\frac{\sqrt{12m}}{T}$,

$$\chi \sim \mathcal{N}(0,1)$$

Note that this scaling happens at the end of the computations and in how we compute the observed value of the output bitstream and has no effect on the design of the GSBG. Therefore, the proposed design can generate random samples with a Gaussian distribution without complex RNG architecture.

6.4 EXPERIMENTS

We use the simulation tool BitSAD (Daruwalla and Zhuo, 2019) as we did before to generate the bitstreams and compute the observed value of the bitstream. –>

We verify the distribution of the output of this design experimentally. Figure 6.9 depicts this distribution for various values of m for 10000 samples and a bitstream of length 100000 bits. As expected, for $m = 1$ the distribution is uniform. As we increase m, the distribution more resembles a Gaussian distribution.

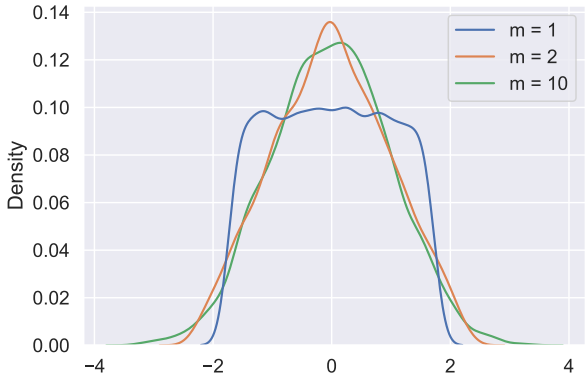


Figure 6.9: Distribution of the output of GSBG for various m

We further evaluate the standard deviation of this distribution in Figure 6.10 for various bitstream lengths. We can see that the standard deviation is near 1. Further,

as we increase m the standard deviation approaches 1 as the sum of uniforms better approximates a Gaussian distributions.

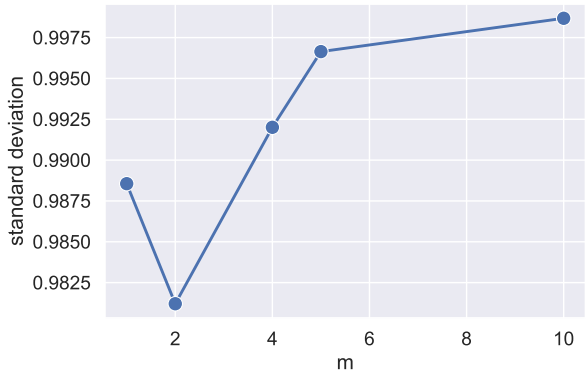


Figure 6.10: Standard deviation of the output bitstream

Another point to consider is that this design for the SBG does not affect the other components of the BNN bitstream computing circuit. We can see that the expected value of each bit follows the standard Gaussian distribution. As a result, computational components, which perform computations between probabilities, perform computations as expected. We can further demonstrate this numerically. We will use the addition operation as well as the square operation to demonstrate this.

6.5 CONCURRENT GAUSSIAN BITSTREAMS

We can take the design of GSBG further and use it to generate a series of interleaved Gaussian stochastic bitstreams. This can be done by simply regrouping the bitstream. In the original GSBG design, we took a bitstream comprising m interleaved Binomial bitstreams and showed that its expected value approximates a Gaussian distribution. We can instead look at these m interleaved bitstreams as a l interleaved groups of $\frac{m}{l}$ bitstreams. Figure 6.11 demonstrates this for $l = 2$ groups. Each of these l bitstream groups will also approximate a Gaussian distribution in expectation, generating l concurrent Gaussian bitstreams.

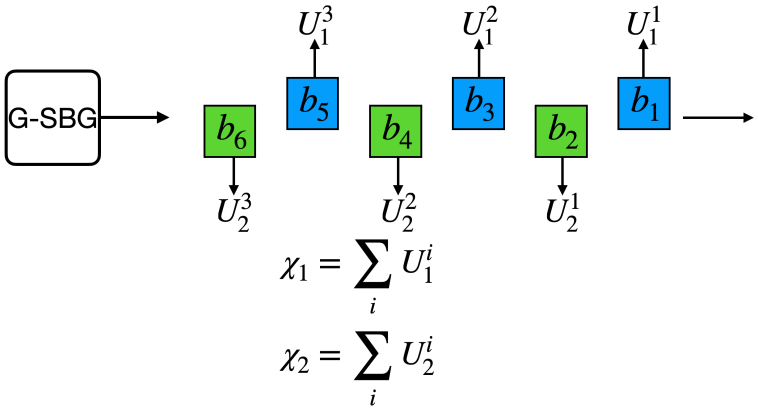


Figure 6.11: Decomposing Gaussian bitstream into two interleaved Gaussian bitstreams

There is little modification needed to the GSBG design to exploit this capability. The mean of the Gaussian samples can be computed without making any changes to our previous computations:

$$\mathbb{E}(\chi) = \frac{1}{l}\sum_{i=1}^{l}\chi_l = \frac{1}{l}\frac{l}{T}\sum_{i=1}^{l}\sum_{t=1}^{\frac{T}{l}} b_{(i-1)l+t} = \frac{1}{T}\sum_{t=1}^{T} b_t$$

To compute the variance, as before, we need to compute $\mathbb{E}(\chi^2)$. For this, the architecture needs to account for the grouping of the interleaved bitstreams. We saw earlier, for $l = 1$, we do not need to make any changes in the design of the square unit in bitstream computing. However, for $\mathbb{E}(\chi^2)$ we need to multiply samples from χ_i by themselves. This means that as we saw for designing bitstream computing components, we need to introduce longer decorrelators. Specifically, the decorrelator needs to be l bits long.

It may be possible to design a decorrelator with programmable l as most other components in the design are agnostic towards l. This would allow even more flexibility for any time computing of the BNN. This however is not core to the scope of our design. Here, we opt to fix the value of l in advance.

6.6 EARLY STOPPING

Building on the design of the Gaussian Stochastic Bitstream Generator, we propose to address the computational efficiency challenges of BNNs by leveraging the anytime computing feature of bitstream computing. Compared to BBP, which serializes computations for network samples as well as computations for each bit of the bitstreams, our proposed concurrent computations allow computations for each network sample to flow together while computations for bits in bitstreams remain serialized. This enables us to estimate uncertainty immediately and improve the estimate as we receive more bits in the output, allowing for progressive uncertainty computation. With low-precision, early estimates that are low or high enough, we can make decisions on whether to reject or accept a sample and make an early prediction, stopping the computation and only performing high-resolution computations for inputs that are harder to predict. We will examine this concurrency approach in more detail in the next section.

We note that this concept is different from previous early stopping methodologies like Streaming Accuracy by Hsiao, San Miguel, and Anderson (2022). Previous early stopping methods are generally concerned with identifying when the observed value of a bitstream has converged to its nominal value. In contrast, we are interested in making quality judgment based on the observed uncertainty value regarding whether the nominal value is going to be high or low. This is in fact an easier task as we can tolerate a higher margin of error. In chapter 7, we will experimentally show that based on early estimates, we can identify high-confidence and low-confidence samples with high probability.

6.7 SUMMARY

In this chapter, we addressed the challenge of generating random samples for BNN computations on bitstream computing substrates. In particular, we explored the concept

of concurrent computation in the BBP and proposed a design which we referred to as the Gaussian Stochastic Bitstream Generator (GSBG). Furthermore, our proposed solution involved designing a new stochastic bitstream generator that interleaved multiple bitstreams allowing for concurrent computation of uncertainty. A key advantage of our design is that if early estimates of uncertainty that are low-precision indicate a very high or very low confidence, we can stop computations and make a prediction early and only perform high-precision computations for difficult samples. This way, we can address computational efficiency limitations of bitstream computing deploying BNNs. In the next chapter, we will present experimental results to validate our proposed design.

7 EXPERIMENTS

In this chapter, we will present our experimental findings and methodologies. First, we will report the results of our experiments using the Bayesian Bitstream Processor (BBP) for the keyword stopping task and discuss precision loss resulting from scaling. Next, we present the results of experiments conducted on different activation functions and their noisy counterparts to address these challenges. Finally, we present the experimental evaluation of our proposed design for BBP with concurrent computations and early stopping. Our evaluation utilizes a more complete set of MLPerf tiny benchmarks, including keyword spotting, image classification, and visual wake words, which are challenging tasks for edge computing and require uncertainty quantification. We use these applications to demonstrate the benefits of concurrency and early stopping to improve delay and energy consumption. We compare our proposed design against an earlier version to highlight significant gains and perform detailed analysis to separate the effects of different design decisions. Throughout this chapter, we provide a comprehensive overview of our design methodology and present in-depth evaluation results to demonstrate the effectiveness and efficiency of our proposed approach.

7.1 EXPERIMENT METHODOLOGY

In this following, we will discuss our evaluation methodology, empirical results, and compare our performance with a previous hardware implementation of keyword spotting.

7.1.1 *MLPerf Tiny Classification Tasks*

MLPerf Tiny (Banbury et al., 2021) was proposed to address the lack of a standard evaluation benchmark for machine learning tasks on edge and embedded devices. We will use the classification tasks from this benchmark suit. Below is a short summary of these tasks:

Keyword Spotting (kws): We used the keyword spotting task from the MLPerf Tiny (Banbury et al., 2021) benchmark for the case study evaluation of BBP. This task classifies Google's Speech Commands (Warden, 2018) dataset and includes options to add various types of noise to the data. As we will show, adding noise to the audio can significantly degrade accuracy, however BNNs can detect noisy data by measuring their uncertainty. In addition, we use a bayesian version of DS-CNN (Sørensen, Epp, and May, 2020), the network used by the task. We train the model offline using tensorflow. We will further use this task for evaluating the efficacy of the GSBG and early stopping technique without added noise.

Image Classification (ic): MLPerf Tiny highlights image classification that has become a pillar of ML algorithm evaluation due to its wide array of applications. In particular, it is a necessary part of safety-critical systems in autonomous vehicles, medical image processing,robotics, etc. In particular, MLPerf Tiny uses benchmarks the training a ResNet (Targ, Almeida, and Lyman, 2016) on the CIFAR10 (Krizhevsky, G. Hinton, et al., 2009) dataset.

Visual Wake Words (vww): Visual Wake Words is binary classification challenge where the system is activated if it detects the presence of a person in the frame and is in a sense the visual counterpart to keyword spotting. As such, it carries similar privacy and safety concerns. MLPerf Tiny uses the Visual Wake Words dataset which is a subset of COCO (T.-Y. Lin et al., 2014) to train a MobileNet (Howard et al., 2017).

For each task, we use the same network as the one in the MLPerf Tiny suite with the modifications described in chapter 5, following the training flow described in Sextion 5.4. The hyper-parameters can be found in Table 7.1. Here, lr is the learning rate, c is the initial simulated annealing noise coefficient for noisy tanh activation training like Gulcehre et al. (2016), and λ is the coefficient for the KL-divergence term in training BNNs.

Table 7.1: Benchmark architectures and datasets

Task	Network	Dataset	Epochs	lr	c	λ
kws	DS-CNN	Speech Commands	100	0.0001	30	1×10^{-5}
ic	ResNet	Cifar10	500	0.0001	30	2×10^{-5}
vww	MobileNet	COCO	500	0.0001	30	1×10^{-5}

7.1.2 *Hardware Evaluation*

We will present comprehensive empirical analysis of our proposed BBP hardware. We used BitSAD (Daruwalla and Zhuo, 2019), a domain-specific programming language for Bitstream Computing, to simulate the Bitstream Computing circuit for each task. BitSAD further allowed us to generate the verilog code. We synthesized the generated verilog code for a CNN layer using Xilinx's vivado and extrapolated the power and delay to LP8K FPGA at 100 MHz. We used this result to create a spreadsheet power, energy, and delay model for the entire model since synthesizing the entire model proved prohibitively long. We derived the delay and power of the GVG unit for the BBP hardware model from a previous work by Alimohammad et al. (2008).

We compared our BBP hardware against two baselines, an ARM Cortex-M4 microcontroller (Banbury et al., 2021) and a NVDLA (X. Wang et al., 2019) accelerator. The microcontroller was listed as the state-of-the-art implementation for MLPerf Tiny (Banbury et al., 2021). We further used NVDLA as an example of a dedicated architecture

based on conventional computing paradigms. We estimated the delay and power of this architecture for the MLPerf Tiny tasks by extrapolating from the study by X. Wang et al. (2019) according to the number of arithmetic operation. For these baselines, we assumed that the random Gaussian samples are generated efficiently using a separate unit. We used a previous study by Kietzmann, Schmidt, and Wählisch (2021) for delay and power estimates of the RNGs. The parameters of these three architectures have been summarized in Table 7.2.

Table 7.2: Evaluated Architectures

	ARM Cortex-M4 (Banbury et al., 2021)	NVDLA (X. Wang et al., 2019)	BBP (Our Work)
Architecture	u-controller	ASIC	FPGA
Precision	Int32	Int16	Int32
Clock	$120MHz$	$200MHz$	$100MHz$
SRAM	$640KB$	$4MB$	$348KB$

7.2 BBP ANALYSIS

We evaluate the accuracy of the network for different numbers of network samples and depict the results in Figure 7.1 on test and validation sets. As expected, more samples achieve higher accuracy. After 10 samples the accuracy starts to plateau and we can achieve the highest accuracy using 20 samples. In the rest of the evaluations we use 10 network samples for all experiments to make sure similar accuracies for BBP as well as the baselines introduced below.

Next, we evaluate the ability of this network to provide meaningful UQ for noisy inputs. For this analysis, we added background noise to the datasets and measured the signal-to-noise ratio of each sample. We duplicated each dataset 5 times, each time adding

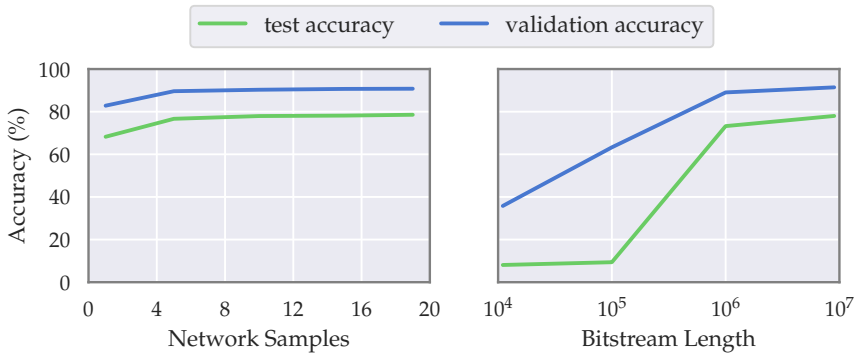


Figure 7.1: Accuracy of bayesian DS-CNN for different networks sample counts (left) and different bitstream lengths (right) for the keyword spotting task

random noise at a different volume to capture a more accurate picture of the performance of the model under noisy conditions.

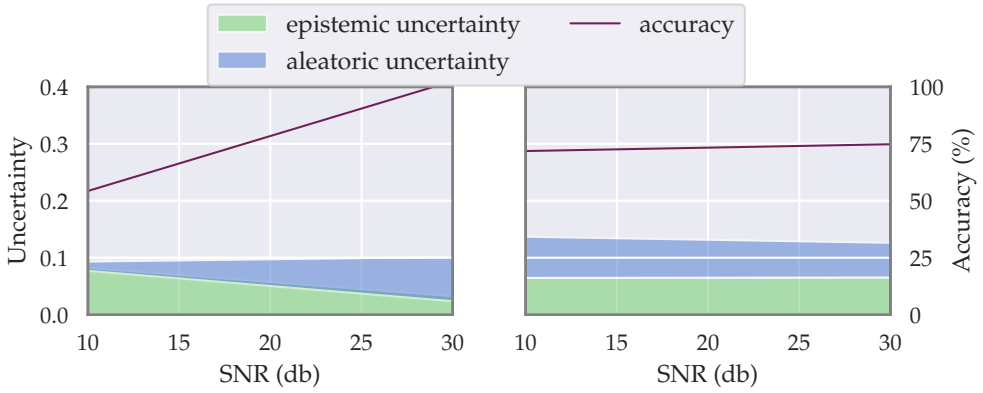


Figure 7.2: Uncertainty analysis of BNN under noisy conditions for the validation (left) and test (right) sets.

Figure 7.2 plots the analysis results.

The left-side y-axis quantifies the uncertainty measurements which have been depicted using the blue and green shades in the figure. We can see that the aleatoric uncertainty as well as the overall uncertainty decreases as SNR improves while epistemic uncertainty stays the same as the model for all samples. We can also see that noisy samples correspond with lower accuracy (right-side y-axis). Consequently, we can use the uncertainty measurements to identify noisy samples. We note that in the results for the test dataset, there is little change observed in uncertainties as noise levels increase. However, this cor-

responds to small changes in accuracy as well. As such, noisy samples are reliably being classified correctly which is reflected in the low uncertainty as well. This demonstrates that uncertainty quantifications can be used as a reliable way to identify samples that can deceive the model.

Next, we study the effect of different bitstream lengths on accuracy and uncertainty measurements of BNN and find the optimal BC implementation.

Accuracy: We first evaluate the accuracy of BC for different bitstream lengths with 20 network samples and without any noise. The results of this experiment for both validation set and the test set have been depicted in Figure 7.1.

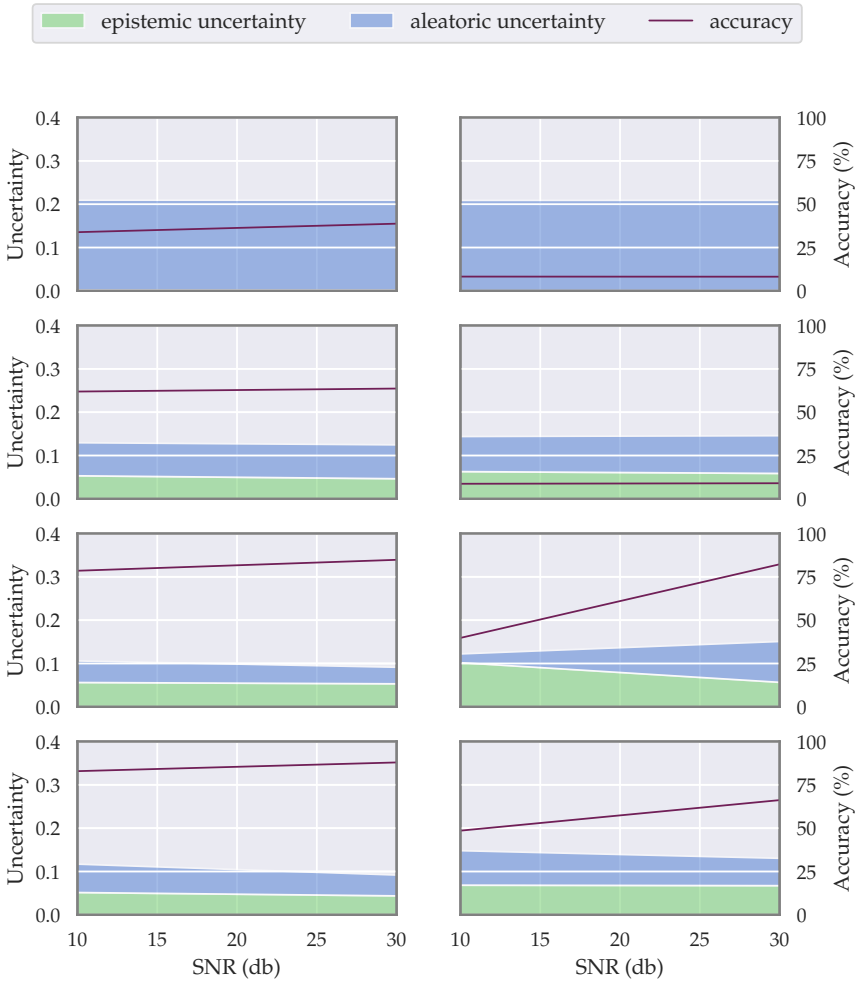


Figure 7.3: Uncertainty quantification of the validation (left) and test (right) sets for different bitstream lengths (n)

As this figure shows, for shorter bitstreams, the accuracy drops significantly. This is due to the inaccuracy of the bitstream representation of the weights which changes the distribution of the sampled weights.

Uncertainty Quantification: To evaluate the effect of BC on uncertainty quantification, we perform the same experiment as Figure 7.2 for different bitstream lengths. That is, we add random background noise data at different volumes and draw the accuracy and uncertainty measurements for different SNR. Same as before, each experiment uses 10 random network samples. The results of these experiments have been depicted in Figure 7.3. As we expected, the shorter bitstreams result in low accuracy as well as higher uncertainty measurements overall. As we increase the bitstream length not only does accuracy improve, but uncertainty decreases. Moreover, uncertainty measurements can more reliably identify higher noise in the samples. Finally, bitstream lengths of 10^6 and 10^7 perform best with 10^7 performing slightly better. Note that after the BC pipeline is full, all computations are performed in parallel. In contrast, in binary computation, layers are computed sequentially. Consequently, BC computing can achieve high performance even when parameters are represented using many bits.

7.3 BBP HARDWARE EVALUATION

Here, we compare BBP with implementations of the task on ARM Cortex-M4 (Banbury et al., 2021) and a NVDLA (X. Wang et al., 2019) accelerator. As the results shown in Table 7.3 show, BBP with a bitstream of length 10^6 outperforms both baselines in delay, energy and power. In particular, it achieves an order of magnitude energy improvement. The more accurate BBP design that uses bitstreams of size 10^7 has higher delay and energy costs. Nevertheless, it still provides better energy and power which are essential metrics for computing at the edge. We note that the on-set detection mechanism introduced in section 3 alleviates the need for very low latency. From the algorithm side, BBP delay can

be further improved using co-design techniques during training of BNN to achieve high accuracies with shorter bitstreams. In addition, from the hardware perspective, delay can be improved through parallelism. In particular, we can duplicate the BC unit to perform forward passes of the network in parallel. One GVG unit is capable of supporting {up to 45 BC units for bitstream size of 10^6}, without causing idle time . This can be used to improve computation speed and achieve real-time computations in a scenario that onset detection cannot be used. Finally, the energy breakdown shows that the overall BBP energy is dominated by BNN passes and the biggest portion of energy savings comes from using BC.

Table 7.3: Hardware evaluation of BBP

	ARM	NVDLA	BBP	BBP
Bitsream Length	-	-	$n = 10^6$	$n = 10^7$
Energy (mJ)	156.28	24.17	2.45	23.93
Network	147.47	15.35	2.04	20.0
RNG	8.81	8.81	0.40	3.93
Power (mW)	42.88	97.95	12.0	12.0
Delay (ms)	3644	247	204	2000

7.4 CONCURRENT BITSTREAM COMPUTATIONS AND EARLY STOPPING

We evaluate the design of a bitstream computing architecture for bayesian neural networks using the tasks described above. The architecture is designed similar to Figure 4.1 but without the dedicated Gaussian Random Variate Generator. This design uses the Gaussian Stochastic Bitstream Generator to produce interleaved Gaussian bitstreams, as was described in the previous chapter. In particular, we are interested in quantifying the effect of early stopping that was introduced in the previous chapter on delay and more importantly, energy. We will follow the methodology of Yong and Brintrup (2022) and

use accuracy-acceptance curves to evaluate our designs. We will also evaluate the effects of this design on energy and delay gains.

Accuracy-acceptance curve (or accuracy-rejection curve) is a way of visualizing the efficacy of uncertainty quantification algorithm. It works by accepting only the low-uncertainty input samples and rejecting high-uncertainty ones and measuring the accuracy only for the accepted samples. By adjusting the threshold where samples are accepted or rejected we obtain the accuracy-acceptance curve where we expect accuracy to be inversely related to acceptance rate. That is using a more strict criteria to accept a sample for classification, we are more likely to classify samples correctly. Furthermore in the context of bitstream computing, samples that are classified more confidently would require less computation precision and can be computed using a smaller bitstream. This will in turn shorten the average bitstream.

7.4.1 *Experimental Results*

We further evaluate the efficacy of the trained networks in quantifying the uncertainty for the benchmark tasks. The accuracy-acceptance curves for these experiments have been depicted in Figure 7.4. This figure illustrates the classification accuracy for different uncertainty thresholds. We can see as the threshold is relaxed, accepting more samples, accuracy drops. Consequently, low-uncertainty samples correspond to more reliable classification as expected. Of course a high acceptance rate, i.e. a loose threshold, can result in mispredictions that can have catastrophic effects. On the other hand, a low acceptance rate, i.e. a strict threshold, can result in frequent activation of backup measures. This can have a significant negative impact on user experience. The choice of the threshold is therefore highly dependent on the cost of backup mechanisms as well as subjective experience. As such, here we do not choose one specific threshold and evaluate the architecture for several threshold of different levels of strictness.

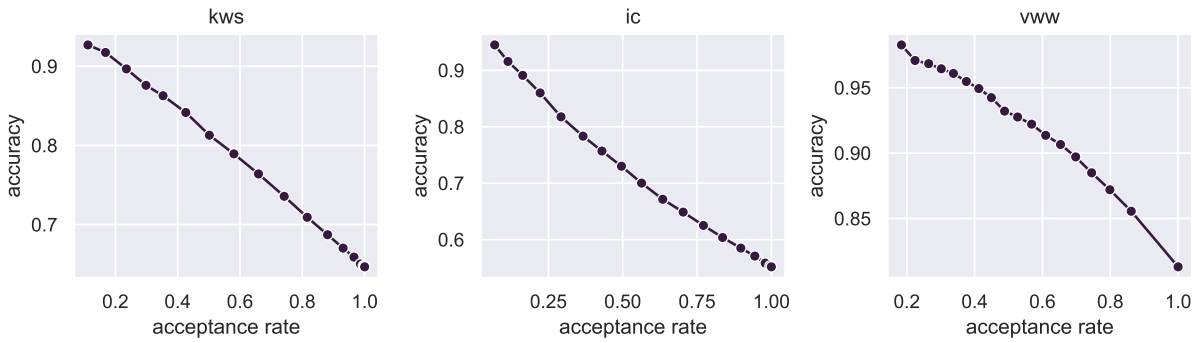


Figure 7.4: Accuracy-acceptance curves for bayesian neural networks trained for the kws, ic, vww tasks

We can therefore evaluate the ability of the bitstream computing architecture for evaluating the uncertainty for each of the benchmark tasks. We depict the accuracy for different thresholds and bitstream lengths in Figure 7.5. Here, as we move from left to right, we use tighter thresholds, accepting a smaller portion of the dataset that can be classified more confidently. However, we see here that accuracy for shorter bitstreams improves more slowly. This is because of quantization which results in the smaller changes in the threshold having no effect. Distinguishing between examples in finer granularity requires longer bitstreams for inputs that lie close to the uncertainty threshold.

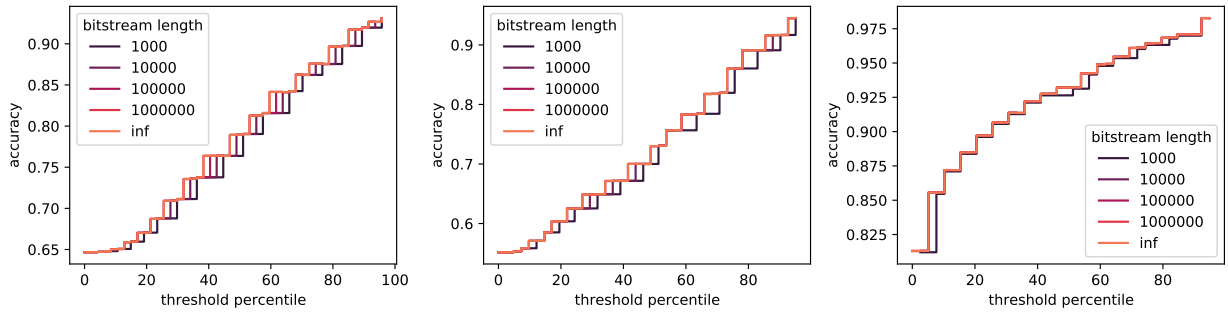


Figure 7.5: Accuracy of bayesian neural networks on kws, ic, vww for various uncertainty thresholds and bitstream length

Uncertainty quantities we derive can be used to make decision faster for easier inputs. We demonstrate this in Figure 7.6. Each plot shows the true positive percentage for different bitstream lengths for one threshold with threshold 1 being the most strict. That

is, how many samples can be classified correctly as a percentage of the total number of samples that are classified correctly for that threshold. We can see for all tasks, the majority of samples can be classified using bitstreams of length 10^3 or 10^4 and longer bitstreams are necessary only for a small portion of samples. This, combined with the relaxed requirement for scaling significantly lowers the average effective bitstream length. These improvements hold across tasks and thresholds.

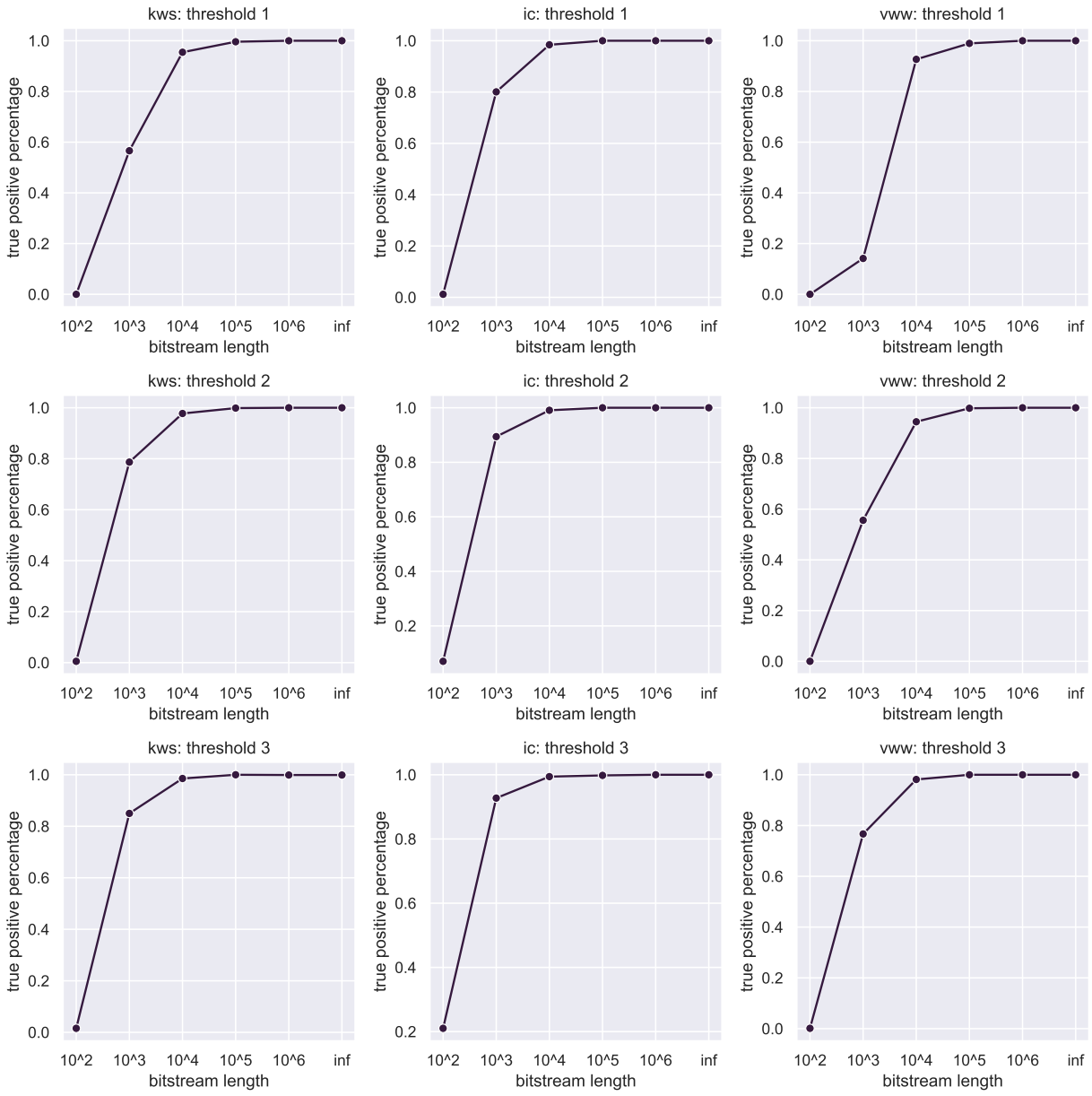


Figure 7.6: True positive percentage on kws, ic, and vww for different thresholds

Table 7.4: Accuracy evaluation of early stopping

	DNN	BNN	BBP+ES
kws	90%	89.7%	89.4%
ic	85%	83.7%	82.6%
vww	80%	96.8%	86.7%

To demonstrate this early stopping capability, we implemented an example early stopping policy. This policy, at certain time steps during the processing of an input looks at the running estimate of the uncertainty, if it is *very low* or *too high*, it stops computing, otherwise it continues computations. If the estimate is very low, the classification decision up to that point is reported. If it is too high, in contrast, the input is rejected. Very low thresholds at each time step are respectively defined as the lowest 10%, 20%, and 30% of the uncertainty range at 10^3, 10^4, and 10^5 bits. Too high is always defined as higher than the 30% of the range threshold.

Table 7.4 summarizes classification accuracy of this policy. For comparison, we include the original benchmark DNN classification accuracy as well as BNN accuracy on low-uncertainty samples in floating-point computations. Here, low-uncertainty refers samples whose uncertainty was in the lower 30% of the uncertainty range for each task. We see that in kws and ic, BNN after rejecting high uncertainty samples performs similar to DNN even though it was not trained to the same accuracy originally. While vww which was trained to a similar accuracy performs better. This underlines the efficacy of uncertainty quantification. Furthermore, we see that when using early stopping in bitstream computing, for kws and ic there is little accuracy drop. On the other hand. On the other hand, vww shows a larger accuracy drop due to it being a more difficult task and more sensitive to imprecise computations. This can be addressed through more precise computations with longer bitstreams or by optimizing the early stopping policy for this task.

Table 7.5: Hardware evaluation of different BBP designs

		BBP	BBP+GSBG	BBP+ES
Energy (uJ)	kws	1201	100	2.29
	ic	1205	100	2.26
	vww	1721	150	3.44
Power (mW)	kws	11.77	10	10
	ic	11.59	10	10
	vww	15.5	15	15
Delay (ms)	kws	102	10	0.23
	ic	104	10	0.22
	vww	111	10	0.23

Finally, we can evaluate the delay and energy gains that come with the shorter effective bitstream length. We depict these results in Table 7.5 which compares this design with other bitstream computing designs that do not use early stopping. More specifically, BBP refers to the original design with dedicated Gaussian Variate Generator, BBP+GSBG uses GSBG with pwl-tanh activations, and finally BBP+ES uses early stopping. We can see linear improvements in BBP+GSBG compared to the baseline BBP. That is because the majority of the gains come from the reduction in the bitstream length by eliminating the need for scaling. When using BBP+ES, the gains depend on the portion of the dataset that can be evaluated early. Nevertheless, these designs each offer up to 44× in energy savings. These savings can be critical in deploying uncertainty quantification at the edge.

7.5 SUMMARY

Throughout this chapter, we have demonstrated the effectiveness and efficiency of our proposed design based on bitstream computing. By combining our original design of the Bayesian Bitstream Processor with the novel Gaussian Stochastic Bitstream Generator, we were able to achieve higher performance and efficiency by computing uncertainty

quantities online, progressively in an anytime fashion that allowed for early stopping of the computations when possible. This approach enabled us to make decisions regarding the prediction of a sample earlier, significantly improving delay and energy consumption. Our experimental results showed that our proposed design outperformed the earlier version in terms of both delay and energy consumption, demonstrating the significant gains that can be achieved through our proposed approach. We also conducted detailed analyses to separate the effects of different design decisions and provided comprehensive evaluations of our proposed design methodology, demonstrating its effectiveness across a range of challenging tasks for edge computing. Overall, our findings suggest that our proposed approach has the potential to significantly advance BNN deployment and contribute to the development of more efficient and low-power neural network inference systems at the edge.

8 CONCLUSIONS

This thesis aimed to address the challenge of deploying trustworthy AI at the edge by combining the uncertainty quantification power of Bayesian Neural Networks (BNNs) and bitstream computing substrates. As AI continues to make its way into all aspects of daily life, we inevitably grow to depend on it more and more. However, as with any technology, it is important to be able to quantify the confidence in its predictions. Uncertainty quantification is essential for ensuring the reliability and trustworthiness of AI systems, but it is often in conflict with mobile and embedded platforms as it introduces additional computations. In contrast, edge computing deployment of AI aims to reduce computations to save energy, which is why we proposed the use of stochastic bitstream computing to address this challenge.

We first demonstrated that bitstream computing substrates can be suitable platforms for Bayesian Neural Networks that as they allow for very low-power and low-cost designs. We noted that the stochasticity of bitstream computing substrates can be leveraged to deploy BNNs more efficiently and their low power and cost can address the challenges of the added computations in BNNs. We proposed a novel design for BNNs based on bitstream computing that enables the efficient computation of both predictions and uncertainty, while minimizing the energy consumption required. Our proposed design leverages the inherent low power and cost of bitstream computing to perform the computations of BNNs in a highly efficient and scalable manner, while still maintaining the accuracy and reliability of the predictions.

We then identified the main bottleneck in the proposed design, that is, the serialization of all computations, and proposed to address it through concurrent computations. We

proposed new stochastic bitstream generation mechanisms to allow for the progressive computation of uncertainty. This approach enables us to make decisions regarding the prediction of a sample earlier when possible, significantly improving delay and energy consumption. By using this approach, we can compute uncertainty quantities online, progressively, which allows us to make more informed and timely decisions. We refer to the new mechanism as the Gaussian Stochastic Bitstream Generator, which enables concurrency. In addition, we conducted a series of experiments to demonstrate the effectiveness of our proposed design. The results of these experiments showed that our method was able to achieve a significant improvement in the efficiency of the Bayesian Neural Network models. Our experiments also demonstrated the ability of our approach to estimate uncertainties effectively and efficiently, which is a crucial requirement for edge computing scenarios.

Our proposed approach has the potential to enable the deployment of new, more powerful, and more trustworthy AI on tiny computing devices. By combining the power of Bayesian Neural Networks with the efficiency and scalability of bitstream computing, we can create new opportunities for edge computing and enable a wide range of applications that were previously not feasible. We believe that our proposed design methodology and experimental evaluation results provide strong evidence for the effectiveness and efficiency of our proposed approach. Overall, this thesis contributes to the advancement of AI and edge computing by providing a new and effective approach for deploying trustworthy AI systems.

8.1 REFLECTIONS AND FUTURE WORK

In addition to the findings and results presented in this dissertation, there are several directions for future research that can be explored. For example, as we have previously discussed, BayesBiNNs can be easily implemented in bitstream computing. However, our

experiments shown that training BayesBiNNs can be challenging for complex tasks. One potential direction for future research is exploring BayesBiNNs in challenging tinyML or complex control tasks. Furthermore, our approach has the potential to facilitate the deployment of BNNs with deeper models at the edge, due to its energy and power advantages as well as its streamlined Gaussian random sample generation. While our study focused on tinyML tasks, future works could explore larger applications such as video processing, which are more demanding tasks at the edge. Similarly, they could investigate more complex posterior distributions, such as mixtures of Gaussians, as our method may be extended to accommodate such cases with relative ease. Additionally, there are other potential directions that could be explored in future research that we will discuss below:

8.1.1 *Random Sampling Order*

The proposed implementation of Bayesian Binarized Neural Networks (BNNs) on Bitstream Computing substrates involves two types of random sampling. The first type is for sampling weights for the neural networks to compute the forward passes, which is necessary in BNN implementations on any hardware. The number of these samples is often determined in advance based on experiments, as we did in chapter 7. The second type of random sampling is for generating bitstreams, which we analyzed in this study and proposed a method for controlling dynamically. However, an alternative approach could be to control both sampling processes dynamically by sampling the network weights more if necessary. For example, when the output uncertainty quantification lies at the decision boundary, we may choose to increase the number of network weight samples or look at longer bitstreams, similar to our approach in this study. Finding the optimal trade-off between these two sampling processes would require in-depth theoretical analysis, which was beyond the scope of this study.

8.1.2 *Reduction Operation*

Matrix multiplications are the core operation in deep learning which comprise element-wise multiplication followed by reduction. Bitstream computing substrates are very good at performing multiplications but reductions require more complex hardware. In particular, the multiplications output ternary results, (i.e. $+1$, 0, or -1) which then need to be added together. There are various ways to perform this reduction in bitstream computing.

One way is to randomly select one of the outputs and pass it on using a MUX operation. It can be easily shown that the expected value of the result is equal to the reduction result. However, this can be problematic as the variance increases proportional to the number of multiplication outputs. In other words, the error increases linearly to the size of the matrix multiplication requiring increasing the bitstream lengths proportionally.

Another approach which is commonly used and we adopted in our design is to use bit counters to perform reductions. This method is more accurate but require more expensive hardware. We can further combine this approach with the previous one and select a subset of the element-wise multiplication outputs at random and use bit count to combine them. This simplifies the bit count circuit which can be expensive for larger matrix multiplications but adds complex random sampling to the reduction. This hybrid approach needs to be adopted carefully to find the optimal balance between bit count and random sampling of multiplication outputs.

Finally, we also explored the use of OR gates for reduction in our approach. If the input bitstreams are assumed to be sparse, meaning close to 0, an OR gate is equivalent to an addition operation in bitstream computing. However, it's important to note that if the inputs are large, the OR gate output can saturate. In our networks, we already employ a piecewise-linear tangent hyperbolic (pwl-tanh) activation function which performs clipping operations, making the OR gate output accurate when inputs are small or

large. Nevertheless, when inputs lie somewhere in between, the result may be inaccurate. Therefore, it is crucial to analyze this intermediate range of inputs to determine the error associated with employing OR gates.

BIBLIOGRAPHY

Alaghi, Armin and John P Hayes (2013). "Survey of stochastic computing". In: *ACM Transactions on Embedded computing systems (TECS)* 12.2s, pp. 1–19.

— (2014). "Fast and accurate computation using stochastic circuits". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1–4.

Alani, Mohammed M (2010). "Testing randomness in ciphertext of block-ciphers using DieHard tests". In: *Int. J. Comput. Sci. Netw. Secur* 10.4, pp. 53–57.

Alawad, Mohammed and Mingjie Lin (2014). "Energy-efficient imprecise reconfigurable computing through probabilistic domain transformation". In: *2014 IEEE Dallas Circuits and Systems Conference (DCAS)*. IEEE, pp. 1–4.

Albericio, Jorge et al. (2017). "Bit-pragmatic Deep Neural Network Computing". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, pp. 382–394. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123982. URL: http://doi.acm.org/10.1145/3123939.3123982.

Alimohammad, Amirhossein et al. (May 2008). "A compact and accurate gaussian variate generator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.5, pp. 517–527. DOI: 10.1109/TVLSI.2008.917552.

Alom, Md Zahangir et al. (2018). "The history began from alexnet: A comprehensive survey on deep learning approaches". In: *arXiv preprint arXiv:1803.01164*.

Banbury, Colby et al. (2021). "MLPerf Tiny Benchmark". In: *arXiv preprint arXiv:2106.07597*.

Blei, David M, Alp Kucukelbir, and Jon D McAuliffe (2017). "Variational inference: A review for statisticians". In: *Journal of the American statistical Association* 112.518, pp. 859–877.

Blundell, Charles et al. (2015). "Weight uncertainty in neural network". In: *International Conference on Machine Learning*. PMLR, pp. 1613–1622.

Boenisch, Franziska et al. (2021). "When the curious abandon honesty: Federated learning is not private". In: *arXiv preprint arXiv:2112.02918*.

Breda, Jérémie, Mihaela Zavolan, and Erik van Nimwegen (2021). "Bayesian inference of gene expression states from single-cell RNA-seq data". In: *Nature Biotechnology* 39.8, pp. 1008–1016.

Brooks, Steve et al. (2011). *Handbook of markov chain monte carlo*. CRC press.

Bui, Thang et al. (2016).
"Deep Gaussian processes for regression using approximate expectation propagation".
In: *International conference on machine learning*. PMLR, pp. 1472–1481.
Cai, Ruizhe et al. (2018). "Vibnn: Hardware acceleration of bayesian neural networks".
In: *ACM SIGPLAN Notices* 53.2, pp. 476–488.
Cameron, Robert D. and Dan Lin (Mar. 2009). "Architectural Support for SWAR Text Processing with Parallel Bit Streams: The Inductive Doubling Principle".
In: *SIGPLAN Not.* 44.3, pp. 337–348. ISSN: 0362-1340.
DOI: 10.1145/1508284.1508283.
URL: http://doi.acm.org/10.1145/1508284.1508283.
Capra, Maurizio et al. (2020). "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks". In: *Future Internet* 12.7, p. 113.
Chai, Yuji et al. (2022). "CoopMC: Algorithm-Architecture Co-Optimization for Markov Chain Monte Carlo Accelerators". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, pp. 38–52.
Chen, Yu-Hsin, Joel Emer, and Vivienne Sze (2016). "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks".
In: *ACM SIGARCH computer architecture news* 44.3, pp. 367–379.
Chen, Tianshi et al. (2014). "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning".
In: *ACM SIGARCH Computer Architecture News* 42.1, pp. 269–284.
Chung, Eric et al. (2018).
"Serving dnns in real time at datacenter scale with project brainwave".
In: *iEEE Micro* 38.2, pp. 8–20.
Cooley, Daniel, Douglas Nychka, and Philippe Naveau (2007).
"Bayesian spatial modeling of extreme precipitation return levels".
In: *Journal of the American Statistical Association* 102.479, pp. 824–840.
Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2015). "Binaryconnect: Training deep neural networks with binary weights during propagations".
In: *arXiv preprint arXiv:1511.00363*.
Courbariaux, Matthieu, Itay Hubara, et al. (2016). "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1".
In: *arXiv preprint arXiv:1602.02830*.
Daruwalla, Kyle and Heng Zhuo (2019).
"BitSAD: A domain-specific language for bitstream computing".
In: *Proceedings of the First ISCA Workshop on Unary Computing (WUC'19)*.
Devlin, Jacob et al. (2014).
"Fast and robust neural network joint models for statistical machine translation".
In: *In Proceedings of ACL2014*, pp. 1370–1380.
Dillon, Joshua V et al. (2017). "Tensorflow distributions".
In: *arXiv preprint arXiv:1711.10604*.
Farshchi, Farzad, Qijing Huang, and Heechul Yun (2019). "Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim".
In: *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, pp. 21–25.

Fernandes, Hugo et al. (2016). "Bayesian inference implemented on FPGA with stochastic bitstreams for an autonomous robot".
In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, pp. 1–4.

Frumusanu, Andrei (2018).
The Samsung Galaxy S9 and S9+ Review: Exynos and Snapdragon at 960fps.

Gaines, Brian R (1969). "Stochastic computing systems".
In: *Advances in Information Systems Science: Volume 2*, pp. 37–172.

Gal, Yarin and Zoubin Ghahramani (2016). "Dropout as a bayesian approximation: Representing model uncertainty in deep learning".
In: *international conference on machine learning*. PMLR, pp. 1050–1059.

Geweke, John (1989).
"Bayesian inference in econometric models using Monte Carlo integration".
In: *Econometrica: Journal of the Econometric Society*, pp. 1317–1339.

Gissin, Daniel and Shai Shalev-Shwartz (2019). "Discriminative active learning".
In: *arXiv preprint arXiv:1907.06347*.

Golder, ER and JG Settle (1976).
"The Box-Müller Method for Generating Pseudo-Random Normal Deviates".
In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 25.1, pp. 12–20.

Graves, Alex (2011). "Practical variational inference for neural networks".
In: *Advances in neural information processing systems* 24.

Gross, Warren J and Vincent C Gaudet (2019).
Stochastic computing: techniques and applications. Springer.

Grünwald, Peter D (2007). *The minimum description length principle*. MIT press.

Gulcehre, Caglar et al. (2016). "Noisy activation functions".
In: *International conference on machine learning*. PMLR, pp. 3059–3068.

Gupta, Prabhat Kumar and Ramdas Kumaresan (1988).
"Binary multiplication with PN sequences".
In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.4, pp. 603–606.

Han, Song, Huizi Mao, and William J Dally (2015). "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding".
In: *arXiv preprint arXiv:1510.00149*.

Hazelwood, Kim et al. (2018).
"Applied machine learning at facebook: A datacenter infrastructure perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, pp. 620–629.

Hernández-Lobato, José Miguel and Ryan Adams (2015).
"Probabilistic backpropagation for scalable learning of bayesian neural networks".
In: *International conference on machine learning*. PMLR, pp. 1861–1869.

Hinton, Geoffrey et al. (Nov. 2012). "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups".
In: *IEEE Signal Processing Magazine* 29.6, pp. 82–97. ISSN: 1053-5888.
DOI: 10.1109/MSP.2012.2205597.

Hinton, Geoffrey E and Drew Van Camp (1993). "Keeping the neural networks simple by minimizing the description length of the weights".
In: *Proceedings of the sixth annual conference on Computational learning theory*, pp. 5–13.

Hirtzlin, Tifenn et al. (2019).
"Stochastic computing for hardware implementation of binarized neural networks".
In: *IEEE Access* 7, pp. 76394–76403.

Howard, Andrew G et al. (2017).
"Mobilenets: Efficient convolutional neural networks for mobile vision applications".
In: *arXiv preprint arXiv:1704.04861*.

Hsiao, Hsuan, Joshua San Miguel, and Jason Anderson (2022).
"Streaming Accuracy: Characterizing Early Termination in Stochastic Computing".
In: *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, pp. 320–325.

Huang, Gao, Zhuang Liu, and Kilian Q. Weinberger (2016).
"Densely Connected Convolutional Networks". In: *CoRR* abs/1608.06993.
arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, et al. (2016a).
"Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *CoRR* abs/1609.07061. arXiv: 1609.07061.
URL: http://arxiv.org/abs/1609.07061.

— (2016b). "Binarized neural networks".
In: *Proceedings of the 30th international conference on neural information processing systems*. Citeseer, pp. 4114–4122.

Jacob, Benoit et al. (2018). "Quantization and training of neural networks for efficient integer-arithmetic-only inference".
In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713.

Jeavons, Peter, David A Cohen, and John Shawe-Taylor (1994).
"Generating binary sequences for stochastic computing".
In: *IEEE Transactions on Information Theory* 40.3, pp. 716–720.

Jouppi, Norman P et al. (2017).
"In-datacenter performance analysis of a tensor processing unit".
In: *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12.

Kendall, Alex and Yarin Gal (2017).
"What uncertainties do we need in bayesian deep learning for computer vision?"
In: *Advances in neural information processing systems* 30.

Khan, Hassan N, David A Hounshell, and Erica RH Fuchs (2018).
"Science and research policy at the end of Moore's law".
In: *Nature Electronics* 1.1, pp. 14–21.

Khoram, Soroosh and Jing Li (2018). "Adaptive quantization of neural networks".
In: *International Conference on Learning Representations*.

Kietzmann, Peter, Thomas C Schmidt, and Matthias Wählisch (2021).
"A guideline on pseudorandom number generation (PRNG) in the IoT".
In: *ACM Computing Surveys (CSUR)* 54.6, pp. 1–38.

Kim, Kyounghoon et al. (2016). "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks".
In: *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6.
Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization".
In: *arXiv preprint arXiv:1412.6980*.
Kingma, Durk P, Tim Salimans, and Max Welling (2015).
"Variational dropout and the local reparameterization trick".
In: *Advances in neural information processing systems* 28.
Koopman, Phil (n.d.). *Maximal Length LFSR Feedback Terms*.
URL: https://users.ece.cmu.edu/~koopman/lfsr/.
Krizhevsky, Alex, Geoffrey Hinton, et al. (2009).
"Learning multiple layers of features from tiny images". In.
Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2017).
"Imagenet classification with deep convolutional neural networks".
In: *Communications of the ACM* 60.6, pp. 84–90.
Laves, Max-Heinrich et al. (2020).
"Calibration of model uncertainty for dropout variational inference".
In: *arXiv preprint arXiv:2006.11584*.
LeCun, Yann, Léon Bottou, et al. (1998).
"Gradient-based learning applied to document recognition".
In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.
LeCun, Yann, Corinna Cortes, and Christopher JC Burges (1998).
The MNIST database of handwritten digits.
URL: http://yann.lecun.com (visited on 02/22/2018).
Lee, Yang Yang and Zaini Abdul Halim (2020).
"Stochastic computing in convolutional neural network implementation: a review".
In: *PeerJ Computer Science* 6, e309.
Li, Bingzhe, M Hassan Najafi, and David J Lilja (2016). "Using stochastic computing to reduce the hardware requirements for a restricted Boltzmann machine classifier".
In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 36–41.
Li, Fengfu, Bo Zhang, and Bin Liu (2016). "Ternary weight networks".
In: *arXiv preprint arXiv:1605.04711*.
Lin, Tsung-Yi et al. (2014). "Microsoft coco: Common objects in context".
In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, pp. 740–755.
Liu, Yidong et al. (2020).
"A survey of stochastic computing neural networks for machine learning applications".
In: *IEEE Transactions on Neural Networks and Learning Systems* 32.7, pp. 2809–2824.
Louizos, Christos, Karen Ullrich, and Max Welling (2017).
"Bayesian compression for deep learning".
In: *Advances in neural information processing systems* 30.
Marsaglia, George (2003). "Xorshift rngs". In: *Journal of Statistical software* 8, pp. 1–6.

Martıén Abadi et al. (2015).
TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.
Software available from tensorflow.org. URL: https://www.tensorflow.org/.
Maselli, Andrea et al. (2020).
"A new method to constrain neutron star structure from quasi-periodic oscillations".
In: *The Astrophysical Journal* 899.2, p. 139.
Matsumoto, Makoto and Takuji Nishimura (1998). "Mersenne twister: a
623-dimensionally equidistributed uniform pseudo-random number generator".
In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1, pp. 3–30.
McMahan, Brendan et al. (2017).
"Communication-efficient learning of deep networks from decentralized data".
In: *Artificial intelligence and statistics*. PMLR, pp. 1273–1282.
Meng, Xiangming, Roman Bachmann, and Mohammad Emtiyaz Khan (2020).
"Training binary neural networks using the bayesian learning rule".
In: *International Conference on Machine Learning*. PMLR, pp. 6852–6861.
Molchanov, Dmitry, Arsenii Ashukha, and Dmitry Vetrov (2017).
"Variational dropout sparsifies deep neural networks".
In: *International Conference on Machine Learning*. PMLR, pp. 2498–2507.
Moore, Gordon E (1998). "Cramming more components onto integrated circuits".
In: *Proceedings of the IEEE* 86.1, pp. 82–85.
Neal, Radford M (2012). *Bayesian learning for neural networks*. Vol. 118.
Springer Science & Business Media.
Neal, Radford M and Geoffrey E Hinton (1998).
"A view of the EM algorithm that justifies incremental, sparse, and other variants".
In: *Learning in graphical models*, pp. 355–368.
Netzer, Yuval et al. (2011).
"Reading digits in natural images with unsupervised feature learning". In.
Nirwan, Rajbir S and Nils Bertschinger (2020).
"Applications of Gaussian process latent variable models in finance".
In: *Intelligent Systems and Applications: Proceedings of the 2019 Intelligent Systems Conference (IntelliSys) Volume 2*. Springer, pp. 1209–1221.
O'neill, Melissa E (2014). "PCG: A family of simple fast space-efficient statistically good
algorithms for random number generation".
In: *ACM Transactions on Mathematical Software*.
Qiu, Jiantao et al. (2016).
"Going Deeper with Embedded FPGA Platform for Convolutional Neural Network".
In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: ACM, pp. 26–35.
ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847265.
URL: http://doi.acm.org/10.1145/2847263.2847265.
Rastegari, Mohammad et al. (2016).
"Xnor-net: Imagenet classification using binary convolutional neural networks".
In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*. Springer, pp. 525–542.

Reuther, Albert et al. (2019).
"Survey and benchmarking of machine learning accelerators".
In: *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE, pp. 1–9.
— (2020). "Survey of machine learning accelerators".
In: *2020 IEEE high performance extreme computing conference (HPEC)*. IEEE, pp. 1–12.
Robert, Christian P and Sylvia Richardson (1998). "Markov Chain Monte Carlo methods".
In: *Discretization and MCMC Convergence Assessment*, pp. 1–25.
Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986).
"Learning representations by back-propagating errors".
In: *nature* 323.6088, pp. 533–536.
Samragh, Mohammad, Mohammad Ghasemzadeh, and Farinaz Koushanfar (Apr. 2017).
"Customizing Neural Networks for Efficient FPGA Implementation".
In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 85–92. DOI: 10.1109/FCCM.2017.43.
Santurkar, Shibani et al. (2018). "How does batch normalization help optimization?"
In: *Advances in neural information processing systems* 31.
Scheffer, Tobias, Christian Decomain, and Stefan Wrobel (2001).
"Active hidden markov models for information extraction".
In: *Advances in Intelligent Data Analysis: 4th International Conference, IDA 2001 Cascais, Portugal, September 13–15, 2001 Proceedings 4*. Springer, pp. 309–318.
Semiconductors 2.0, International Technology Roadmap for (2015). *Beyond CMOS*.
URL: https://www.semiconductors.org/wp-content/uploads/2018/06/6_2015-ITRS-2.0-Beyond-CMOS.pdf.
Settles, Burr (2009). "Active learning literature survey". In.
Sharma, Hardik et al. (2018). "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, pp. 764–775.
Shridhar, Kumar, Felix Laumann, and Marcus Liwicki (2018).
"Uncertainty estimations by softplus normalization in bayesian convolutional neural networks with variational inference". In: *arXiv preprint arXiv:1806.05978*.
Shukla, Rohit et al. (2018).
"Computing generalized matrix inverse on spiking neural substrate".
In: *Frontiers in neuroscience* 12, p. 115.
Simonyan, Karen and Andrew Zisserman (2014).
"Very deep convolutional networks for large-scale image recognition".
In: *arXiv preprint arXiv:1409.1556*.
Song, Zhuoran et al. (2020).
"Drq: dynamic region-based quantization for deep neural network acceleration". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, pp. 1010–1021.
Sørensen, Peter Mølgaard, Bastian Epp, and Tobias May (2020). "A depthwise separable convolutional neural network for keyword spotting on an embedded system".
In: *EURASIP Journal on Audio, Speech, and Music Processing* 2020.1, pp. 1–14.

Srivastava, Nitish et al. (2014).
"Dropout: a simple way to prevent neural networks from overfitting".
In: *The journal of machine learning research* 15.1, pp. 1929–1958.
Suda, Naveen et al. (2016). "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25.
Sys, Marek and Zdenek Riha (2014).
"Faster randomness testing with the NIST statistical test suite".
In: *Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings 4*. Springer, pp. 272–284.
Targ, Sasha, Diogo Almeida, and Kevin Lyman (2016).
"Resnet in resnet: Generalizing residual architectures".
In: *arXiv preprint arXiv:1603.08029*.
Teye, Mattias, Hossein Azizpour, and Kevin Smith (2018).
"Bayesian uncertainty estimation for batch normalized deep networks".
In: *International Conference on Machine Learning*. PMLR, pp. 4907–4916.
Thiebes, Scott, Sebastian Lins, and Ali Sunyaev (2021).
"Trustworthy artificial intelligence". In: *Electronic Markets* 31, pp. 447–464.
Thompson, Neil and Svenja Spanuth (2018).
"The decline of computers as a general purpose technology: why deep learning and the end of Moore's Law are fragmenting computing". In: *Available at SSRN 3287769*.
Ullrich, Karen, Edward Meeds, and Max Welling (2017).
"Soft weight-sharing for neural network compression".
In: *arXiv preprint arXiv:1702.04008*.
Wan, Qiyu and Xin Fu (2020).
"Fast-BCNN: Massive Neuron Skipping in Bayesian Convolutional Neural Networks".
In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, pp. 229–240.
Wan, Qiyu, Haojun Xia, et al. (2021). "Shift-BNN: Highly-Efficient Probabilistic Bayesian Neural Network Training via Memory-Friendly Pattern Retrieving".
In: *arXiv preprint arXiv:2110.03553*.
Wang, Keze et al. (2016). "Cost-effective active learning for deep image classification".
In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.12, pp. 2591–2600.
Wang, Xingbin et al. (2019).
"NPUFort: A secure architecture of DNN accelerator against model inversion attack".
In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 190–196.
Warden, Pete (2018).
"Speech commands: A dataset for limited-vocabulary speech recognition".
In: *arXiv preprint arXiv:1804.03209*.
Wilson, Andrew G et al. (2016). "Stochastic variational deep kernel learning".
In: *Advances in neural information processing systems* 29.
Wing, Jeannette M (2021). "Trustworthy ai".
In: *Communications of the ACM* 64.10, pp. 64–71.

Yao, Jiayu et al. (2019).
"Quality of uncertainty quantification for Bayesian neural network inference".
In: *arXiv preprint arXiv:1906.09686*.
Yong, Bang Xiang and Alexandra Brintrup (2022). "Bayesian autoencoders with uncertainty quantification: Towards trustworthy anomaly detection".
In: *Expert Systems with Applications* 209, p. 118196.
Zhang, Da and Hui Li (2008). "A stochastic-based FPGA controller for an induction motor drive with integrated neural network algorithms".
In: *IEEE Transactions on Industrial Electronics* 55.2, pp. 551–561.
Zhang, Xiangyu et al. (2021).
"Statistical robustness of Markov chain Monte Carlo accelerators".
In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 959–974.
Zhang, Yawen, Sheng Lin, et al. (2020).
"When sorting network meets parallel bitstreams: A fault-tolerant parallel ternary neural network accelerator based on stochastic computing".
In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1287–1290.
Zhang, Yawen, Runsheng Wang, et al. (2020).
"Parallel hybrid stochastic-binary-based neural network accelerators".
In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12, pp. 3387–3391.
Zhao, Liang et al. (2017). "Theoretical properties for neural networks with weight matrices of low displacement rank". In: *international conference on machine learning*. PMLR, pp. 4082–4090.
Zhou, Shuchang et al. (2016). "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160.
arXiv: 1606.06160. URL: http://arxiv.org/abs/1606.06160.
Zhu, Chenzhuo et al. (2016). "Trained ternary quantization".
In: *arXiv preprint arXiv:1612.01064*.
Zhu, Feng et al. (2020). "Towards unified int8 training for convolutional neural network".
In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1969–1979.
Zilberstein, Shlomo (1996). "Using anytime algorithms in intelligent systems".
In: *AI magazine* 17.3, pp. 73–73.