UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Low power hardware architecture for Bayesian Neural Networks

Diploma Thesis

Antonios-Kyrillos Chatzimichail

Supervisor: Nikolaos Bellas

September 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Low power hardware architecture for Bayesian Neural Networks

Diploma Thesis

Antonios-Kyrillos Chatzimichail

Supervisor: Nikolaos Bellas

September 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αρχιτεκτονική υλικού χαμηλής ισχύος για νευρωνικά δίκτυα Bayes

Διπλωματική Εργασία

Αντώνιος-Κύριλλος Χατζημιχαήλ

Επιβλέπων: Νικόλαος Μπέλλας

Σεπτέμβριος 2022

Approved by the Examination Committee:

Supervisor	**Nikolaos Bellas** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Yehia Massoud** Professor, Department of Computer, Electrical and Mathematical Science & Engineering (CEMSE), King Abdullah University of Science and Technology (KAUST)
Member	**Christos Antonopoulos** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

A KAUST Remote Internship Program sponsored this work. The author would also like to acknowledge Dr. Charalampos Antoniadis, affiliated with Innovative Technologies Labs (ITL) at KAUST, for his continuous support and guidance throughout the internship.

Diploma Thesis

Low power hardware architecture for Bayesian Neural Networks

Antonios-Kyrillos Chatzimichail

Abstract

In recent years, Neural Networks (NNs) have proved their potential in various tasks such as image recognition or autonomous driving. However, conventional NNs do not express the uncertainty of their predictions; thus, they can not be trusted for critical applications that do not tolerate errors. On the other hand, because Bayesian Neural Networks (BNNs) consider probability distributions on the NN weights instead of scalars, they can mathematically measure the uncertainty of their predictions. There are two methods for implementing BNN inference, the Monte Carlo based method, which requires the sampling of weights distributions and multiple inference iterations, and moment propagation, where the mean and variance of a normal distribution is propagated through the BNN. Hardware implementations of moment propagation BNN inference consume less power because they complete the inference in a single forward pass. Because propagation of normal distributions through non-linear activation functions leads to large hardware designs, not suitable for resource-constrained platforms such as FPGAs, these activation functions are approximated by polynomials. Hardware implementations of moment propagation have been studied solely for fully-connected neural networks, while lacking optimal accuracy due to the approximation of the $ReLU$ activation function with a single polynomial term. Therefore, in this work we add one more polynomial term in the approximation of $ReLU$ providing better accuracy with negligible additional hardware. We also propose a polynomial approximation for another common activation function, $tanh$, and extend the hardware implementation to Convolutional Neural Networks (CNNs). Experimental results demonstrated that the proposed approximation of $ReLU$ outperforms the previously suggested single-term polynomial by achieving up to 5.9% higher accuracy with negligible hardware area and power overhead.

Keywords:

Bayesian Neural Network, FPGA, Moment Propagation, $ReLU$ polynomial approximation, $tanh$ polynomial approximation

Διπλωματική Εργασία

Αρχιτεκτονική υλικού χαμηλής ισχύος για νευρωνικά δίκτυα Bayes

Αντώνιος-Κύριλλος Χατζημιχαήλ

Περίληψη

Τα τελευταία χρόνια, τα νευρωνικά δίκτυα έχουν αποδείξει τις δυνατότητές τους σε διάφορες εργασίες όπως η αναγνώριση εικόνας ή η αυτόνομη οδήγηση. Ωστόσο, τα συμβατικά νευρωνικά δίκτυα δεν εκφράζουν την αβεβαιότητα των προβλέψεών τους. Επομένως, δεν μπορούν να είναι αξιόπιστα για κρίσιμες εφαρμογές που δεν ανέχονται σφάλματα. Από την άλλη πλευρά, επειδή τα Bayesian Neural Networks (BNNs) θεωρούν κατανομές στα βάρη είναι αντί για βαθμωτές τιμές, μπορούν να μετρήσουν μαθηματικά την αβεβαιότητα των προβλέψεών τους. Υπάρχουν δύο μέθοδοι για την υλοποίηση BNN inference, η μέθοδος Monte Carlo, η οποία απαιτεί δειγματοληψία των κατανομών-βαρών και πολλαπλές επαναλήψεις inference, και η μέθοδος moment propagation, όπου η μέση τιμή και η διακύμανση μιας κανονικής κατανομής διαδίδεται μέσω του BNN. Οι hardware υλοποιήσεις του BNN inference με moment propagation καταναλώνουν λιγότερη ισχύ επειδή ολοκληρώνουν το inference σε ένα μόνο πέρασμα. Επειδή η διάδοση κανονικών κατανομών μέσω μη γραμμικών συναρτήσεων ενεργοποίησης οδηγεί σε μεγάλες αρχιτεκτινικές, ακατάλληλες για πλατφόρμες με περιορισμούς πόρων όπως οι FPGAs, αυτές οι συναρτήσεις προσεγγίζονται με πολυώνυμα. Οι υλοποιήσεις υλικού του moment propagation έχουν μελετηθεί αποκλειστικά για πλήρως συνδεδεμένα BNNs, ενώ δεν έχουν βέλτιστη ακρίβεια λόγω της προσέγγισης της συνάρτησης ενεργοποίησης $ReLU$ με έναν μόνο πολυωνυμικό όρο. Επομένως, σε αυτήν την εργασία προσθέτουμε έναν ακόμη πολυωνυμικό όρο στην προσέγγιση της $ReLU$, παρέχοντας καλύτερη ακρίβεια με αμελητέα πρόσθετο υλικό. Προτείνουμε επίσης μια πολυωνυμική προσέγγιση για μια άλλη κοινή συνάρτηση ενεργοποίησης, την $tanh$, και επεκτείνουμε την hardware υλοποίηση στα Συνελικτικά BNNs. Τα πειράματα έδειξαν ότι η προτεινόμενη προσέγγιση της $ReLU$ υπερέχει του προηγούμενου πολυωνύμου ενός όρου, επιτυγχάνοντας έως και $5,9\%$ υψηλότερη ακρίβεια με αμελητέα επιβάρυνση σε επιφάνεια υλικού και ισχύ.

Λέξεις-κλειδιά:

Bayesian Νευρωνικό Δίκτυο, FPGA, Moment Propagation, πολυωνυμική προσέγγιση της $ReLU$, πολυωνυμική προσέγγιση της $tanh$

Table of contents

List of figures

List of tables

Abbreviations

2D	two-dimensional
AF	Activation Function
ANN	Artificial Neural Network
aPE	average Predictive Entropy
ASIC	Application-Specific Integrated Circuit
BN	Batch Normalization
BNN	Bayesian Neural Network
BRAM	Block Random Access Memory
CDF	Cumulative Distribution Function
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
etc.	et cetera
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
GRNG	Gaussian Random Number Generator
HDL	Hardware Description Language
HLS	High-Level Synthesis
II	Initiation Interval
MC	Monte Carlo
MCD	Monte Carlo Dropout
NN	Neural Network
PL	Programmable Logic

PS	Processing System
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RGB	red, green, and blue
RNN	Recurrent Neural Network
ROM	Read-Only Memory
SRAM	Static Random Access Memory
XPE	Xilinx Power Estimator

Chapter 1

Introduction

1.1 Motivation and related work

Nowadays, Artificial Neural Networks (ANNs) achieve noticeable success in multiple fields of science and life, such as image classification [1], autonomous driving [2], predicting COVID-19 pandemic cases [3], etc. However, standard neural networks are not suitable for critical applications in domains such as healthcare, self-driving cars, and air-crafts. Instead, Bayesian Neural Networks (BNNs) have been proposed for such applications that cannot tolerate any errors caused by overconfidence of the neural network.

What distinguishes Bayesian Neural Networks (BNNs) from other types of NNs is that the weights are distributions (usually normal with certain mean and standard deviation values) instead of scalars. The benefit of this difference is that the network's output represents a distribution, and thus, the measurement of uncertainty in the output can indicate how sure we can be that the network responds with an answer that can be trusted. For example, a typical ANN that classifies images of handwritten digits will falsely pick one of the 10 digits when the input image shows something random or irrelevant. However, a Bayesian NN can answer that it cannot be sure what is inside the image due to the high uncertainty in the result (Figure 1.1).

In the recent years, several works [4, 5, 6] focus on hardware acceleration of BNN inference, targeting Field-Programmable Gate Arrays (FPGAs) as development devices to achieve high performance and energy efficiency. Among these works, there are two main approaches for the BNN inference: the Monte Carlo (MC) approach and the Moment Propagation approach. The former approach computes the average of N network inference evaluations,

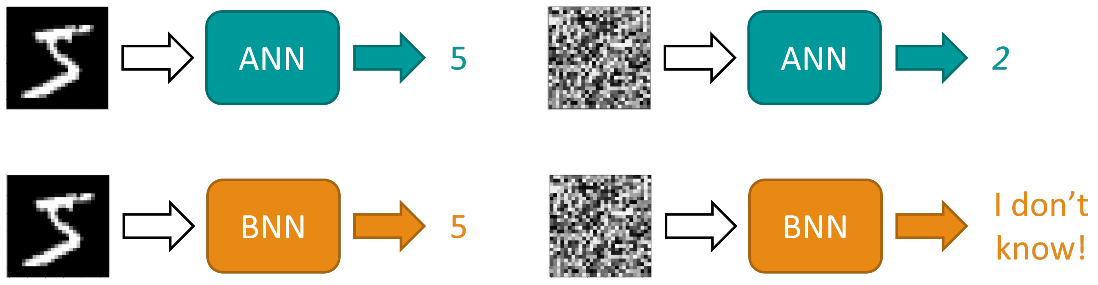


Figure 1.1: Difference between typical ANNs and BNNs

wherein for each network pass, the weights are obtained by random sampling of Gaussian distribution, providing different outputs on each evaluation. For a high performance to be achieved by this approach, the authors of [4] mainly concentrate on implementing fast hardware-based Gaussian Random Number Generators (GRNGs). An alternative MC suggested in [6] uses Monte Carlo Dropout, where conventional neural networks are trained and the uncertainty quantification occurs from the dropout, which discards the neurons of the NN with some probability in every inference pass. In [6], partial BNNs, which consist of both Bayesian and conventional deterministic layers, have also been studied in an effort to reduce the portion of the BNN that needs to be inferred multiple times. As for the latter approach for implementing BNN inference, Moment Propagation, the neurons represent Gaussian distributions whose moments, i.e., mean and variance, propagate from the first to the last layer [5]. Even though the last method requires more computations for a single pass, the advantage is that only one inference iteration is required, which leads to more power efficient hardware implementations.

Moment propagation technique seems to be promising since it requires only one iteration of the inference, but comes with a limitation: it can be efficiently implemented in hardware for specific activation functions. In the most recent hardware accelerator for Bayesian inference with moment propagation, BYNQNet [5], the quadratic activation function $f(x) = x^2$ has been used as a replacement of the Rectified Linear Unit (ReLU) activation function, and tested in a fully-connected network.

1.2 Thesis objective

The accuracy of models with the quadratic approximation $f(x) = x^2$ might be inadequate for some critical applications. Also, several interesting datasets are better suited for

Convolutional Neural Networks (CNNs), which have not been investigated in BYNQNet.

This work aims to widen the range of applications of the existing hardware implementation of BNN inference with moment propagation. First of all, an additional term in the polynomial approximation of $ReLU$ is proposed to improve the model accuracy ($\frac{1}{2}(x^2 + x)$ instead of x^2). A polynomial approximation ($x - \frac{1}{3}x^3$) is also suggested for the $tanh$ activation function. Moreover, the above functions are integrated not only in the 3-layer fully connected neural network used in [4] and [5] (also referred as $FC3L\ net$ in this thesis), but also in LeNet-5 [7], a more complex network that consists of 5 layers and includes convolutional and pooling layers as well.

1.2.1 Contribution

The contributions of the thesis are summarized below:

1. extension of the hardware implementation of Moment Propagation method to Bayesian Convolutional Neural Networks

2. enhancement of the polynomial approximation of $ReLU$ used in [5] to improve accuracy, and proposal of a $tanh$ polynomial approximation as activation function that produces an efficient hardware implementation of moment propagation

3. evaluation of moment propagation performance in complex datasets

1.3 Thesis structure

Chapter 2 explains the use of High-Level Synthesis for programming Field-Programmable Gate Arrays (FPGAs), provides fundamental ideas of Bayesian Neural Networks (BNNs), and elaborates the key parts of sampling-free BNN inference on FPGAs. Chapter 3 includes a presentation of the proposed activation functions for sampling-free Bayesian inference, and a detailed description of the optimizations made on the hardware design. In Chapter 4, there is an explanation of the experiments took place and the results derived from software and hardware metrics. Chapter 5 concludes the thesis and suggests some future work.

Chapter 2

Background

This chapter aims to introduce the reader to the two main fields of this work, Field-Programmable Gate Arrays (FPGAs) and Bayesian Neural Networks (BNNs). Moreover, it discusses the characteristics of sampling-free BNN inference on FPGAs.

2.1 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that are able to be reconfigured. This hardware programmable feature is an outcome of the internal structure of FPGAs. In particular, they consist of multiple Configurable Logic Blocks (CLBs) arranged in a matrix form, which are connected by programmable interconnects, as illustrated in Figure 2.1. In this way, the user can connect different blocks to implement the desired design. This procedure is achieved by generating a bitstream, which is loaded in SRAM inside the device. [8, 9]

In the rest of this section, two main use cases for FPGAs are emphasized, hardware prototyping and hardware acceleration. After that, it follows a description of the key concepts of High-Level Synthesis (HLS).

2.1.1 Use of Field-Programmable Gate Arrays

FPGAs are reprogrammable, meaning that they are an ideal platform for design prototyping, a vital step before Application-Specific Integrated Circuit (ASIC) implementation. ASICs are custom integrated circuit chips with a strictly particular purpose. Most of the times, they achieve design implementations with lower area, latency and power consumption than

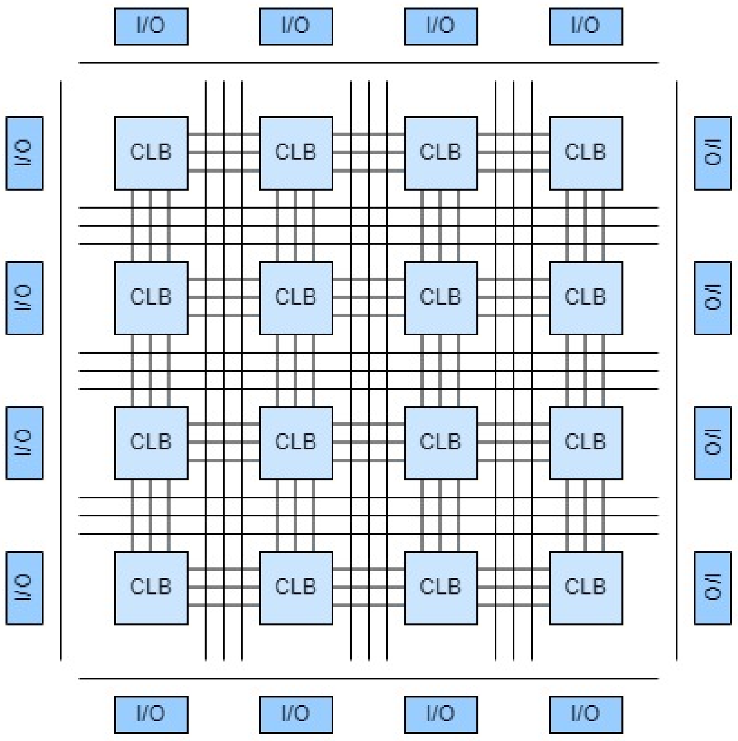


Figure 2.1: FPGA architecture

FPGA implementations. However, the manufacturing process of ASICs requires a lot of time and has high in advance cost. That leads to the need of a functioning design in the first place. As a matter of fact, the procedure of hardware verification is more efficient and low-cost in FPGAs. [8, 9, 10]

Hardware acceleration is another often field of use for FPGAs. Among various target platforms for applications, including general purpose CPUs and GPUs, FPGAs typically are the most appropriate for specific tasks. That is because all the available hardware resources are tailored to the needs of a certain application, making it possible for the application not only to run in less time but also to save energy. It has to be noted, though, that hardware accelerators come with a burden: the application development is more time-consuming than it would be in software, since the design needs to be manually optimized to utilize the FPGA's resources in the best way possible. [8, 10]

2.1.2 High-level synthesis for FPGAs

FPGAs are normally programmed using a Hardware Description Language (HDL), such as Verilog or VHDL. HDLs allow the programmer to specify circuits and implement algorithms at low level. Arithmetic and logical operations can be performed on bits and simple data types, led by a pulse signal, the clock. Although programming in an HDL provides a lot of control over the produced hardware, it requires expertise for the programmer to optimize the design. [10, 11]

Another way of programming FPGAs is by using High-Level Synthesis (HLS) tools, such as Xilinx Vitis HLS. These tools are able to convert a program written in a high-level language, like C and C++, to a low-level HDL. In this way, development time is reduced since high level languages are easier to learn and implement. Of course, there are many difficulties in this process. Firstly, despite the sequential nature of code, the tool needs to find parallelism in it to reach better performance. Moreover, there are parts of software code that are not synthesizable, meaning that they can not be directly converted to hardware. In this case, the programmer needs to alter the code to replace the parts that can not be synthesized. Additionally, in order to help the tool achieve decent performance, programmers need to modify even synthesizable programs. They can also use directives to instruct the HLS compiler to implement pipelines, perform parallelization in parts of code, etc. [11]

All in all, HLS provides less flexibility to the designer, but saves development time compared to HDLs. In many cases, HLS designs are proved to be efficient and achieve high performance.

2.2 Bayesian Neural Networks (BNNs)

In this section, the idea of Bayesian Neural Networks is briefly explained, following a description of the basic intuition led to Artificial Neural Networks in general.

2.2.1 Introduction to Artificial Neural Networks

Artificial Neural Networks (ANNs) have been inspired by the biological neural network. The human brain is not pre-programmed to perform certain tasks but learns from life experience. That is what ANNs try to imitate in order to form Artificial Intelligent systems.

Although ANNs lack in some respects compared to biological neural networks, they both share a common ground. The building block of neural networks (NNs) is the neuron, which, in biology, can also be called a nerve cell. A neuron receives signals from other neurons through dendrites, which are nerve fibres, and sends signals to other neurons by its axon, a fibre that forks into many branches that end up to the other neurons' synapses. One neuron is typically connected to thousand other neurons. The cell body of each neuron accumulates in a way the signals arriving at its synapses. Each synapse gets stronger or weaker, depending on the rate that is triggered. That makes the strength of each synapse adaptable and, therefore, trainable. [12]

Identically, the artificial neuron gathers the input values a_i of all of its connections with other neurons. The input values are multiplied by weights w_i to simulate the adaptation of synapses' strength. Usually, a constant bias term b is also added to the accumulated value, which, then, is given as input to a non-linear activation function f. The final output s of the neuron is shown in 2.1, where M is the number of input values.

$$s = f\left(\sum_{i=1}^{M}(w_i a_i) + b\right) \tag{2.1}$$

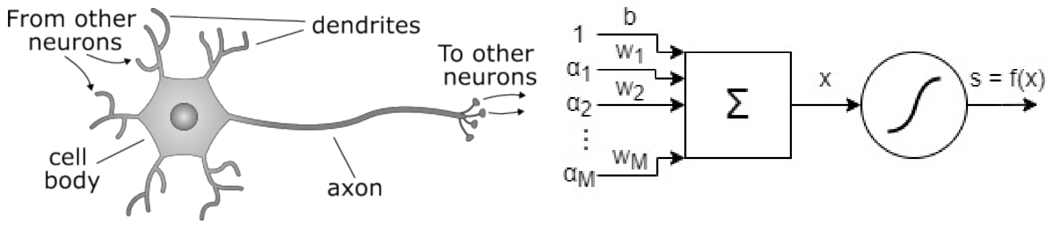


Figure 2.2: Biological and artificial neuron

An ANN consists of many neurons organized in layers. In a feed-forward NN, neurons of consecutive layers are connected. The layers of neurons are categorized according to their functionality. In this work, fully-connected, convolutional, pooling, Batch Normalization (BN) and dropout layers are used.

Fully-connected is the most common type of layer. Every neuron is connected to all neurons of the previous layer. [13] The output value of each neuron is computed by 2.1. An example of this type of layer is shown in Figure 2.3.

Convolutional (Figure 2.4) are named the layers that use matrices called filters or kernels to execute convolution operations on their input (Figure 2.5). The trainable parameters in this case are the values of the filters, along with a bias term. The output of a convolutional layer is called a feature map. [14] These layers work particularly well for tasks involving images, such is image classification [15], and are usually followed by fully-connected layers to optimize class scores. [14] There are a lot of parameters affecting the convolution operations, such is

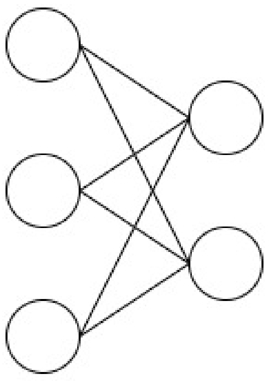


Figure 2.3: Example of a fully-connected layer with 2 neurons and 3 inputs
Each of the 2 neurons is connected with every neuron of the previous input layer.

the filter size and the number of input and output channels. [16] Channels are interpreted as color channels in the case of images, meaning, for example, that grayscale images have one channel, while RGB colored images have three.

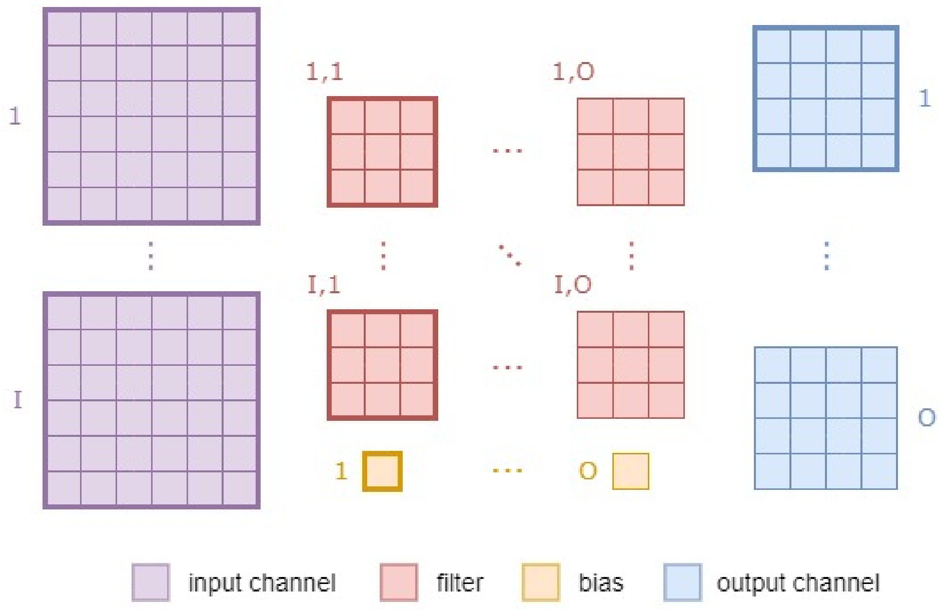


Figure 2.4: A 2D convolutional layer with I input and O output channels
Convolution operations are performed between the input channels and filters in bold outline, and then the bias in bold is added element-wise to produce the output channel in bold.

Pooling layers are often used in CNNs to reduce the dimension of the feature maps and, therefore, decrease the number of trainable parameters and computations. It replaces a block of adjacent values with a single value. That value is usually either the average or the maximum value of the elements inside the block, as Figure 2.6 demonstrates. In such cases, the layer is

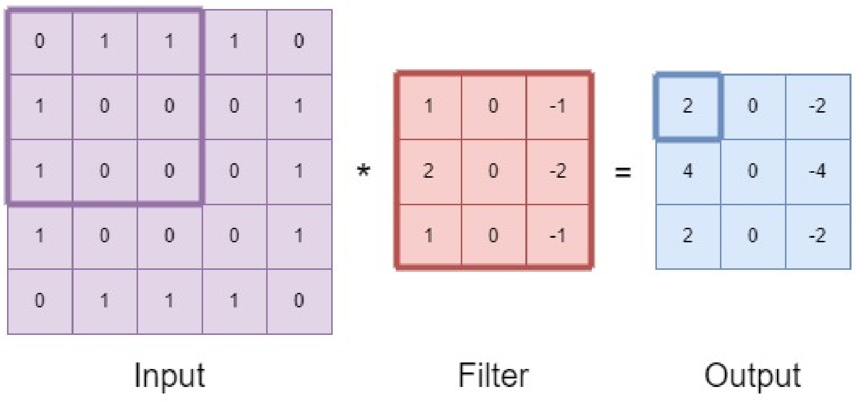


Figure 2.5: Example of a 2D convolution operation
The first element of the output is calculated by element-wise multiplication and accumulation of the filter and the marked input. For the rest of the output to be produced, the filter window slides over the whole input performing the same operation each time.

called Average or Max pooling respectively. Pooling also contributes to avoiding overfitting of the model. [17, 18]

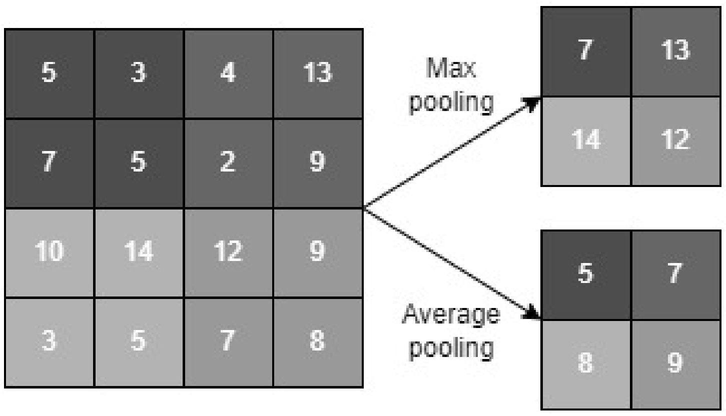


Figure 2.6: Example of a 2D Max and Average Pooling

Batch Normalization (BN) layers are used to normalize the input of hidden layers, meaning that all the inputs have the same scale. That leads to faster training of the network since the loss function converges to a minimum in less steps. A BN layer has trainable parameters, γ and β, which define the mean and variance of the layer's input. There are also non-trainable parameters, moving mean and moving variance, which are used to calculate the mean and variance in the inference. BN applies the transformation shown in equation 2.2, where μ and σ^2 are the moving mean and moving variance, and ϵ is a small constant value. The position of a BN layer originally suggested to be before the activation function, but there are also implementations where BN layers are after the activation functions. [19, 20]

$$\hat{x} = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{2.2}$$

Dropout layers drop some neurons randomly, according to a dropout probability that is set. When a neuron is dropped, it means it is excluded from the neural network. This technique can be applied during training for overfitting to be avoided. [21] It can also be applied to both training and inference, in which case it is called Monte Carlo Dropout, to implement Bayesian inference. [22]

2.2.2 Idea behind Bayesian Neural Networks

Bayesian Neural Networks (BNNs) are Neural Networks enhanced with the advantages of probabilistic models. [23] Instead of scalar values, they can produce a distribution for their predictions, meaning that they better avoid overfitting and have a better sense of the prediction confidence. In contrast to conventional neural networks, BNNs provide extra valuable information, which is the quantified uncertainty of the predictions they make.

A BNN considers a posterior probability distribution $P(\mathcal{W}|\mathcal{D})$ for the weights $\mathcal{W}$ given a dataset $\mathcal{D}$. According to the Bayes rule, the posterior probability distribution is given by:

$$P(\mathcal{W}|\mathcal{D}) = \frac{\mathcal{P}(\mathcal{D}|\mathcal{W})\mathcal{P}(\mathcal{W})}{\mathcal{P}(\mathcal{D})} \tag{2.3}$$

where $P(\mathcal{W})$ is a prior probability distribution, $P(\mathcal{D}|\mathcal{W})$ is called the likelihood and $P(\mathcal{D})$ is the integral of sum over all possible $\mathcal{W}$ variables [24, 4]:

$$P(\mathcal{D}) = \int_{\mathcal{W}} \mathcal{P}(\mathcal{D}|\mathcal{W}')\mathcal{P}(\mathcal{W}')\,\mathrm{d}\mathcal{W}' \tag{2.4}$$

Since the computation of eq. 2.4, and consequently eq. 2.3, is difficult, $P(\mathcal{W}|\mathcal{D})$ is approximated by a given distribution which is most of the times the normal distribution with trainable μ and σ. Therefore, the weights take the form of equation 2.5, where μ and σ are the mean and standard deviation of the weight distribution, and ϵ is sampled from a unit Gaussian distribution. In the inference of the BNN, the output of N separate network inferences is averaged, with each inference using different sampled weights. [24, 4] The structure of a Bayesian Neural Network can be seen in Figure 2.7.

$$w = \mu + \epsilon\sigma \tag{2.5}$$

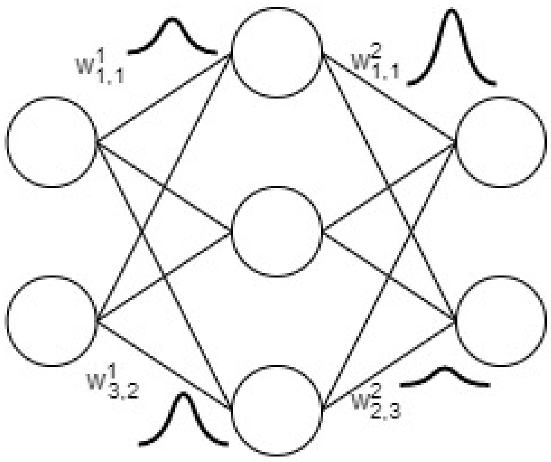


Figure 2.7: Example of a BNN

Unlike deterministic NNs, the weights of a BNN are distributions. Each weight distribution is sampled during inference.

The inference method described above requires sampling of weights, however, there are also sampling-free methods.

2.3 Sampling-free BNN inference on FPGAs

The sampling-free technique of moment propagation for BNN inference does not involve Monte Carlo (MC) sampling. Instead of being represented by a scalar value, each neuron carries two values, the mean and the variance of a Gaussian distribution. Moment propagation calculates the mean and variance (moments) of the neurons in one pass, starting with the neurons of the first layer and moving forward one layer at a time. In [5], it has been shown for the MNIST dataset [25] that the moment propagation method provides comparable results with 10 MC samples, in less time.

In moment propagation method, the input layer ($l = 0$) has neurons with zero variance $\mathbb{V}[h_i^0]$[1], and mean $\mathbb{E}[h_i^0]$ equal to the input values of the dataset. The mean and variance of the pre-activations of each next layer l are computed by equations 2.6, where h_j^{l-1} refers to each of the previous layer's post-activations, w_{ij}^l to the weight between the i-th neuron of layer l and j-th neuron of layer $l-1$, and b_i^l to the bias of i-th neuron of layer l. Since $\mathbb{E}[w_{ij}^l]$ and

[1]In general, variance $\mathbb{V}$ is given by $\mathbb{V} = \frac{\sum_{i=1}^{N}(x_i - x_\mu)^2}{N}$, where x_i is each value in the dataset, x_μ is the mean value, and N is the number of values.

$\mathbb{V}[w_{ij}^l]$ are known after training, parameters $w_{\mu,ij}^l = \mathbb{E}[w_{ij}^l]$, $w_{v,ij}^l = \left(\mathbb{V}[w_{ij}^l] + \mathbb{E}[w_{ij}^l]^2\right)$ and $w_{\mu^2,ij}^l = \mathbb{V}[w_{ij}^l]$ in 2.6 can be calculated before inference.

$$\begin{aligned}
\mathbb{E}[x_i^l] &= b_i^l + \sum_j \mathbb{E}[w_{ij}^l]\,\mathbb{E}[h_j^{l-1}] = \\
&= b_i^l + \sum_j w_{\mu,ij}^l\,\mathbb{E}[h_j^{l-1}] \\
\mathbb{V}[x_i^l] &= \sum_j \left[\left(\mathbb{V}[w_{ij}^l] + \mathbb{E}[w_{ij}^l]^2\right)\mathbb{V}[h_j^{l-1}] + \mathbb{V}[w_{ij}^l]\,\mathbb{E}[h_j^{l-1}]^2\right] = \\
&= \sum_j \left[w_{v,ij}^l\,\mathbb{V}[h_j^{l-1}] + w_{\mu^2,ij}^l\,\mathbb{E}[h_j^{l-1}]^2\right]
\end{aligned} \tag{2.6}$$

The next part of computations is the moment propagation through the activation function, using the pre-activation moments, as shown in Figure 2.8. As a matter of fact, the activation function of a layer affects the moment propagation computations. For the computation of moment propagation through some activation functions, non-trivial operations, such as cumulative distribution function (CDF) computation, need to be performed. These operations can not be implemented efficiently in hardware designs. On the other hand, moment propagation of polynomial activation functions that require only polynomial operations are better implemented in hardware. That leads to the need of replacing activation functions with respective polynomial approximations. The authors of [5] proposed the replacement of the ReLU activation function with the quadratic activation function, $f(x) = x^2$. The latter can be easily integrated in the moment propagation algorithm, and due to its small computational overhead, it results in a feasible and efficient hardware implementation.

An important detail on BNNs with polynomial approximation functions is the addition of Batch Normalization (BN) layers before activations. A significant reason for using BN layers is that they shrink the range of pre-activations to [-1,1], where the polynomials approximate more accurately common activation functions. In this case, $w_{\mu,ij}^l$, $w_{v,ij}^l$, $w_{\mu^2,ij}^l$ parameters and bias b_i^l in 2.6 are transformed into:

$$\begin{aligned}
\left(w_{\mu,ij}^l\right)' &= \frac{\gamma_i^l}{\sqrt{(\sigma_i^l)^2 + \epsilon}} w_{\mu,ij}^l \\
\left(w_{v,ij}^l\right)' &= \frac{(\gamma_i^l)^2}{(\sigma_i^l)^2 + \epsilon} w_{v,ij}^l \\
\left(w_{\mu^2,ij}^l\right)' &= \frac{(\gamma_i^l)^2}{(\sigma_i^l)^2 + \epsilon} w_{\mu^2,ij}^l \\
\left(b_i^l\right)' &= \beta_i^l + \gamma_i^l \frac{b_i^l - \mu_i^l}{\sqrt{(\sigma_i^l)^2 + \epsilon}}
\end{aligned} \tag{2.7}$$

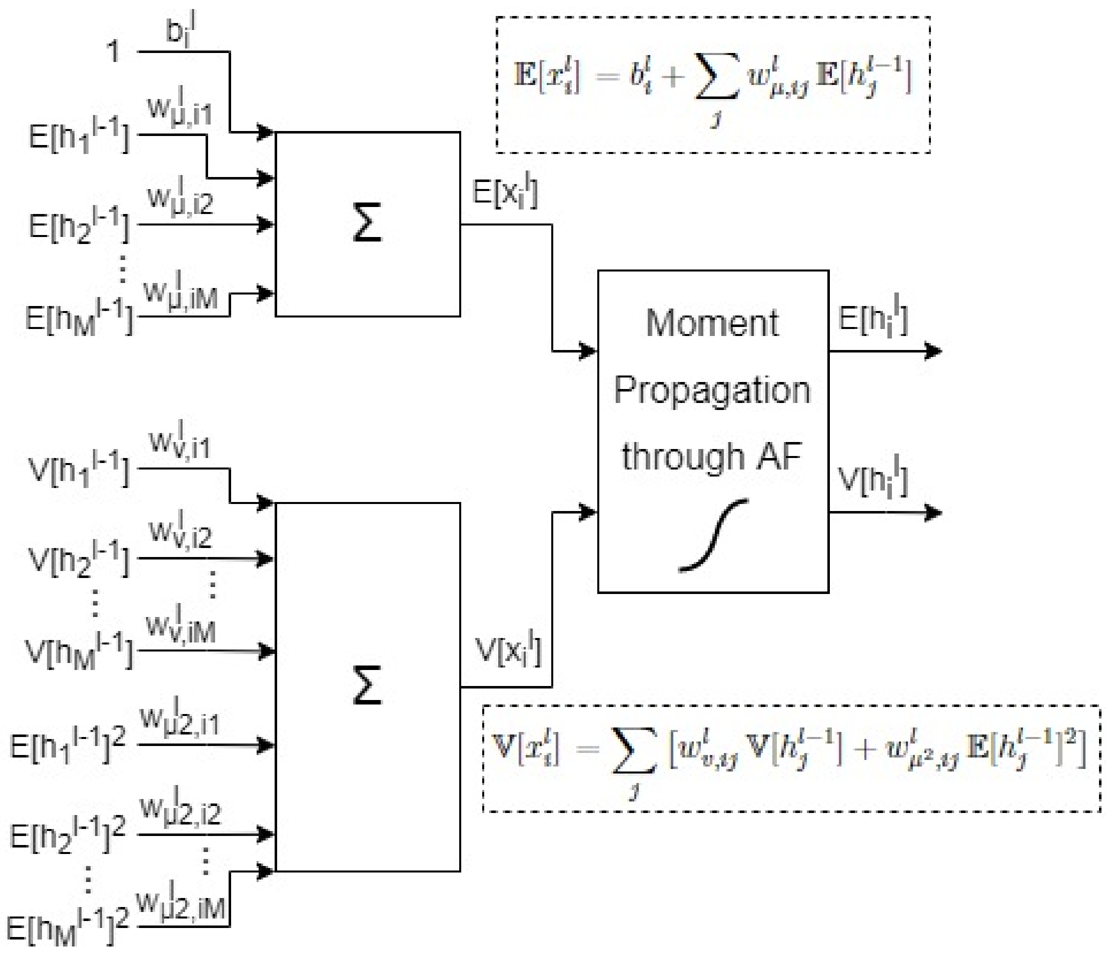


Figure 2.8: A neuron of a BNN implementing moment propagation

The mean $\mathbb{E}[x_i^l]$ and variance $\mathbb{V}[x_i^l]$ of the pre-activations are given by equations 2.6. The computation of the mean $\mathbb{E}[h_i^l]$ and variance $\mathbb{V}[h_i^l]$ of the post-activations depends on the activation function, and will be described in sections 3.1 and 3.3.

where the BN parameters γ_i^l, β_i^l, μ_i^l, σ_i^l, and the constant ϵ have already be defined in training.

Chapter 3

Proposed architecture and optimizations

In this chapter, polynomial functions are proposed as replacements of common activation functions, in order for moment propagation algorithm to be efficiently implemented in hardware. In particular, a new quadratic function is proposed to approximate $ReLU$ better than $f(x) = x^2$, which has been used in BYNQNet [5]. A polynomial approximation is also suggested for $tanh$. Additionally, a hardware implementation is presented for both fully-connected and convolutional BNNs.

3.1 Proposed quadratic approximation of the ReLU activation function

In [26], the authors empirically show that $f(x) = x^2$ is not the ideal second degree polynomial to replace the ReLU function. Instead, they propose $f(x) = \frac{1}{2}(x^2 + ax)$ as a better option for approximating ReLU in the range $[-a, a]$. For this function to be integrated in the moment propagation algorithm, its first and second moments need to be calculated.

Consider the pre-activation random variable x_i^l. Then, the first moment of the post-activation through $\frac{1}{2}({x_i^l}^2 + ax_i^l)$ is given by:

$$\mathbb{E}[h_i^l] = \mathbb{E}[\frac{1}{2}({x_i^l}^2 + ax_i^l)] = \frac{1}{2}(\mathbb{E}[{x_i^l}^2] + \mathbb{E}[ax_i^l]) = \frac{1}{2}(\mathbb{V}[x_i^l] + \mathbb{E}[x_i^l]^2 + a\,\mathbb{E}[x_i^l]) \tag{3.1}$$

where $\mathbb{E}[x_i^l]$ and $\mathbb{V}[x_i^l]$ are computed by equations 2.6. The second moment is given by:

$$\begin{aligned}\mathbb{V}[h_i^l] = \mathbb{V}[\frac{1}{2}({x_i^l}^2 + ax_i^l)] = \frac{1}{4}(\mathbb{V}[{x_i^l}^2] + a^2\,\mathbb{V}[x_i^l] + 2a * cov(x_i^l, {x_i^l}^2)) = \\ = \frac{1}{4}(\mathbb{V}[{x_i^l}^2] + a^2\,\mathbb{V}[x_i^l] + 2a * (\mathbb{E}[{x_i^l}^3] - \mathbb{E}[x_i^l]\,\mathbb{E}[{x_i^l}^2]))\end{aligned} \tag{3.2}$$

where $cov(X, Y)$ is the covariance between random variables X and Y and is defined as $cov(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\,\mathbb{E}[Y]$. Due to the assumption of normal distribution propagation and because the skewness of a normal distribution is 0, we have

$$\begin{aligned}
\mathbb{E}[(\frac{x_i^l - \mu}{\sigma})^3] = 0 \Rightarrow \mathbb{E}[(x_i^l - \mu)^3] = 0 \Rightarrow \mathbb{E}[{x_i^l}^3 - 3{x_i^l}^2\mu + 3x_i^l\mu^2 - \mu^3] = 0 \Rightarrow \\
\Rightarrow \mathbb{E}[{x_i^l}^3] - 3\,\mathbb{E}[{x_i^l}^2]\mu + 3\,\mathbb{E}[x_i^l]\mu^2 - \mu^3 = 0 \Rightarrow \\
\Rightarrow \mathbb{E}[{x_i^l}^3] - 3\,\mathbb{E}[{x_i^l}^2]\mu + 3\mu^3 - \mu^3 = 0 \Rightarrow \\
\Rightarrow \mathbb{E}[{x_i^l}^3] = 3\,\mathbb{E}[{x_i^l}^2]\mu - 2\mu^3
\end{aligned} \tag{3.3}$$

Then, by substituting $\mathbb{E}[{x_i^l}^3]$ in 3.2 with the result in 3.3, we obtain

$$\begin{aligned}
\mathbb{V}[\frac{1}{2}({x_i^l}^2 + ax_i^l)] = \frac{1}{4}(\mathbb{V}[{x_i^l}^2] + a^2\,\mathbb{V}[x_i^l] + 2a * (3\,\mathbb{E}[{x_i^l}^2]\mu - 2\mu^3 - \mu\,\mathbb{E}[{x_i^l}^2])) = \\
= \frac{1}{4}(\mathbb{V}[{x_i^l}^2] + a^2\,\mathbb{V}[x_i^l] + 2a * (2\,\mathbb{E}[{x_i^l}^2]\mu - 2\mu^3)) = \\
= \frac{1}{4}(\mathbb{V}[{x_i^l}^2] + a^2\,\mathbb{V}[x_i^l] + 4a * (\mathbb{E}[{x_i^l}^2]\mu - \mu^3)) = \\
= \frac{1}{4}(2\,\mathbb{V}[x_i^l](\mathbb{V}[x_i^l] + 2\,\mathbb{E}[x_i^l]^2) + a^2\,\mathbb{V}[x_i^l] + 4a * (\mathbb{E}[{x_i^l}^2]\mu - \mu^3)) = \\
= \frac{1}{4}(2\,\mathbb{V}[x_i^l](\mathbb{V}[x_i^l] + 2\,\mathbb{E}[x_i^l]^2) + a^2\,\mathbb{V}[x_i^l] + 4a * ((\mathbb{V}[x_i^l] + \mathbb{E}[x_i^l]^2)\,\mathbb{E}[x_i^l] - \mathbb{E}[x_i^l]^3)) = \\
= \frac{1}{4}(2\,\mathbb{V}[x_i^l](\mathbb{V}[x_i^l] + 2\,\mathbb{E}[x_i^l]^2) + a^2\,\mathbb{V}[x_i^l] + 4a * (\mathbb{V}[x_i^l]\,\mathbb{E}[x_i^l]))
\end{aligned} \tag{3.4}$$

Due to the use of normalized input and Batch Normalization layers, the range of interest for the pre-activations is $[-1, 1]$. Therefore, since $a = 1$, the proposed quadratic polynomial approximation function for ReLU is $f(x) = \frac{1}{2}(x^2 + x)$. This function is compared to $f(x) = x^2$ and ReLU in Figure 3.1.

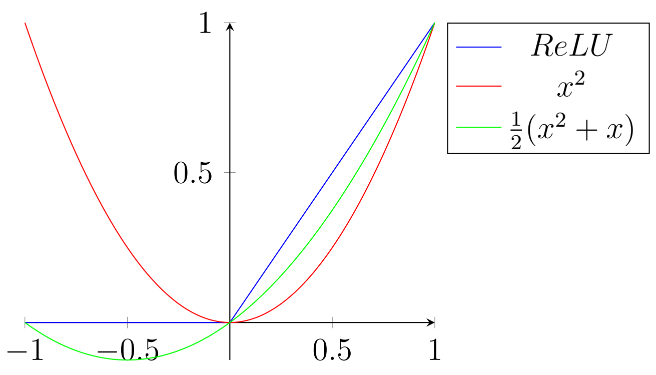
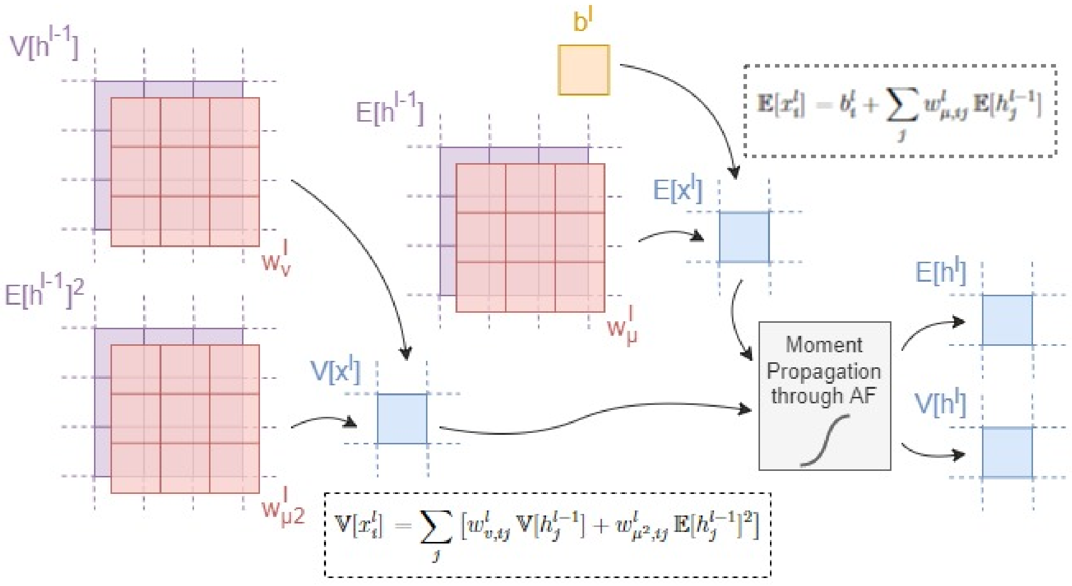


Figure 3.1: Comparison of polynomial functions with ReLU

3.2 Moment propagation in convolutional and pooling layers

Convolutional layers are widely used in image recognition and other tasks [27]. They are also used in BNNs [6]. The moment propagation computations for the convolutional layer is illustrated in Figure 3.2 and shares similar ideas to the fully connected layer. In essence, for each element of a layer's l output, there are two steps. In the first step, the mean and variance of the pre-activations, $\mathbb{E}[x^l]$ and $\mathbb{V}[x^l]$, are computed using equations 2.6, where, instead of dot products, convolution operations are performed between the post-activations of the previous layer $l-1$ and the pre-computed parameters w^l. Subsequently, the equations of moment propagation through the desired activation function are applied (for example, eq. 3.1, 3.4 with $a = 1$ in the case of $f(x) = \frac{1}{2}(x^2 + x)$). The produced mean and variance of the post-activations are then used as input by the next layer $l+1$.



$V[h^{l-1}]$
w_v^l
$E[h^{l-1}]^2$
$w_{\mu 2}^l$
b^l
$E[h^{l-1}]$
w_μ^l
$\mathbf{E}[x_i^l] = b_i^l + \sum_j w_{\mu,ij}^l \, \mathbf{E}[h_j^{l-1}]$
$E[x^l]$
$V[x^l]$
Moment Propagation through AF
$E[h^l]$
$V[h^l]$
$\mathbb{V}[x_i^l] = \sum_j \left[w_{v,ij}^l \, \mathbb{V}[h_j^{l-1}] + w_{\mu^2,ij}^l \, \mathbf{E}[h_j^{l-1}]^2 \right]$

Figure 3.2: Pre- and post-activation computation in a 2D convolutional layer

Pooling layers are frequently used after convolutional layers for subsampling. Among various types of pooling layers, Average and Max pooling layers are some of the most common. More specifically, these layers slide a window over the input, maintaining the average or maximum value of the elements inside the window. In the case of moment propagation, the window consists of Gaussian distributions. For Average pooling layers, the output is a new Gaussian distribution with its mean and variance calculated as shown in equations 3.5, where $\mathbb{E}[h_j^{l-1}]$ and $\mathbb{V}[h_j^{l-1}]$ are the mean and variance of the j-th element of the window and N is the number of elements in the window. As for Max pooling layers, the distribution with the largest mean is maintained.

$$\mathbb{E}[h_i^l] = \frac{\sum_{j=1}^{N} \mathbb{E}[h_j^{l-1}]}{N}, \quad \mathbb{V}[h_i^l] = \frac{\sum_{j=1}^{N} \mathbb{V}[h_j^{l-1}]}{N^2} \tag{3.5}$$

3.3 Proposed approximation of the tanh activation function

Tanh is another popular activation function for neural networks. Like ReLU, a polynomial approximation needs to be found in order for it to be applied in BNN infernce with moment propagation. As for tanh polynomial approximation, the first two nonzero terms of the respective Taylor expansion were selected, $x - \frac{1}{3}x^3$ (proof in Appendix A). In Figure 3.3, tanh and its polynomial approximation are compared in the range of interest $[-1, 1]$.

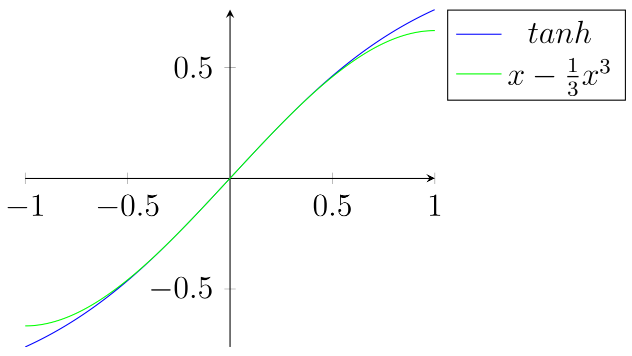


Figure 3.3: Comparison of tanh function and its polynomial approximation

Once again, the first two moments of the polynomial need to be calculated and integrated in the moment propagation algorithm. Considering a pre-activation random variable x_i^l, the first moment of the post-activation through $x_i^l - \frac{1}{3}{x_i^l}^3$ activation function is:

$$\begin{aligned} \mathbb{E}[h_i^l] = \mathbb{E}[x_i^l - \frac{1}{3}{x_i^l}^3] = \mathbb{E}[x_i^l] - \frac{1}{3}\,\mathbb{E}[{x_i^l}^3] = \\ = \mathbb{E}[x_i^l] - \frac{1}{3}(3\,\mathbb{E}[{x_i^l}^2]\,\mathbb{E}[x_i^l] - 2\,\mathbb{E}[x_i^l]^3) = \\ = \mathbb{E}[x_i^l] - \mathbb{E}[{x_i^l}^2]\,\mathbb{E}[x_i^l] + \frac{2}{3}\,\mathbb{E}[x_i^l]^3 = \\ = \mathbb{E}[x_i^l] - (\mathbb{V}[x_i^l] + \mathbb{E}[x_i^l]^2)\,\mathbb{E}[x_i^l] + \frac{2}{3}\,\mathbb{E}[x_i^l]^3 = \\ = \mathbb{E}[x_i^l] - \mathbb{V}[x_i^l]\,\mathbb{E}[x_i^l] - \mathbb{E}[x_i^l]^3 + \frac{2}{3}\,\mathbb{E}[x_i^l]^3 = \\ = \mathbb{E}[x_i^l] - \mathbb{V}[x_i^l]\,\mathbb{E}[x_i^l] - \frac{1}{3}\,\mathbb{E}[x_i^l]^3 \end{aligned} \tag{3.6}$$

and the second moment is given by:

$$\mathbb{V}[h_i^l] = \mathbb{V}[x_i^l - \frac{1}{3}x_i^{l^3}] = \mathbb{V}[x_i^l] + \frac{1}{9}\,\mathbb{V}[x_i^{l^3}] - \frac{2}{3}cov(x_i^l, x_i^{l^3}) \tag{3.7}$$

$$cov(x_i^l, x_i^{l^3}) = \mathbb{E}[x_i^{l^4}] - \mathbb{E}[x_i^l]\,\mathbb{E}[x_i^{l^3}] \tag{3.8}$$

Since $\mathbb{E}[X^4] = \mathbb{E}[X]^4 + 6\,\mathbb{E}[X]^2\,\mathbb{V}[X] + 3\,\mathbb{V}[X]^2$ for a Gaussian random variable X, 3.8 becomes

$$\begin{aligned} cov(x_i^l, x_i^{l^3}) &= \mathbb{E}[x_i^l]^4 + 6\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2 - \mathbb{E}[x_i^l](3\,\mathbb{E}[x_i^{l^2}]\,\mathbb{E}[x_i^l] - 2\,\mathbb{E}[x_i^l]^3) = \\ &= \mathbb{E}[x_i^l]^4 + 6\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2 - 3\,\mathbb{E}[x_i^{l^2}]\,\mathbb{E}[x_i^l]^2 + 2\,\mathbb{E}[x_i^l]^4 = \\ &= 3\,\mathbb{E}[x_i^l]^4 + 6\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2 - 3(\mathbb{V}[x_i^l] + \mathbb{E}[x_i^l]^2)\,\mathbb{E}[x_i^l]^2 = \\ &= 6\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2 - 3\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] = \\ &= 3\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2 \end{aligned} \tag{3.9}$$

Applying 3.9 to 3.7, and because $\mathbb{V}[X^3] = 9\,\mathbb{E}[X]^4\,\mathbb{V}[X] + 36\,\mathbb{E}[X]^2\,\mathbb{V}[X]^2 + 15\,\mathbb{V}[X]^3$, we have

$$\begin{aligned} \mathbb{V}[x_i^l - \frac{1}{3}x_i^{l^3}] &= \mathbb{V}[x_i^l] + \frac{1}{9}\,\mathbb{V}[x_i^{l^3}] - \frac{2}{3}(3\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] + 3\,\mathbb{V}[x_i^l]^2) = \\ = \mathbb{V}[x_i^l] + \frac{1}{9}&(9\,\mathbb{E}[x_i^l]^4\,\mathbb{V}[x_i^l] + 36\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l]^2 + 15\,\mathbb{V}[x_i^l]^3) - 2\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] - 2\,\mathbb{V}[x_i^l]^2 \\ &= \mathbb{V}[x_i^l] + \mathbb{E}[x_i^l]^4\,\mathbb{V}[x_i^l] + 4\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l]^2 + \frac{5}{3}\,\mathbb{V}[x_i^l]^3 - 2\,\mathbb{E}[x_i^l]^2\,\mathbb{V}[x_i^l] - 2\,\mathbb{V}[x_i^l]^2 \end{aligned} \tag{3.10}$$

3.4 Design flow

The design flow of this work's accelerator (Figure 3.4) starts from the BNN training phase, which generates all the weights that are used later in the inference. For this phase, each of the BNNs has been built in Python code utilizing TensorFlow and TensorFlow Probability libraries. The proposed polynomial activation functions are not built-in in TensorFlow, so they have been added. TensorFlow uses the automatic differentiation technique to compute the activation function's gradient and then, it trains the network by implementing backpropagation. For conventional NNs to be converted to Bayesian ones, all fully connected and convolutional layers need to use the reparameterization method supported by TensorFlow Probability. After the training ends, all the weights are exported to a file to be used in the inference. More specifically, these weights are the means and standard deviations of the weight distributions along with the biases of fully connected and convolutional layers, and the γ, β, μ and σ parameters of batch normalization layers (used to compute equations 2.7).

The next phase is the software implementation of the inference to validate the correctness of the calculations. The inference program, which has been developed in C, can be parameterized to implement a BNN's inference using either the moment propagation or the MC sampling method. Firstly, it initializes the BNN's layers and loads their weights from the weights file generated by the Python program at the end of training. After that, it loads from a file the dataset that feeds the network and then, it executes the inference. Lastly, the program measures the accuracy and the average predictive entropy (aPE) over the dataset, as shown in equation 3.11, where D is the dataset size, K is the number of classes and $p(y_d^k|x_d)$ is the probability of the BNN classification y_d to be k given input x_d. Accuracy indicates the correctness of the predictions and it is desired to be as high as possible, while aPE is a measurement that can quantify predictions' uncertainty and it should be in general as low as possible, but distinctively high for confusing input, like random Gaussian noise.

$$aPE = \frac{1}{D}\left[\sum_{d=1}^{D}\left(-\sum_{k=1}^{K} p(y_d^k|x_d)\log p(y_d^k|x_d)\right)\right] \tag{3.11}$$

After the verification that the C code produces right results, the part of the code implementing the moment propagation inference is moved to Xilinx Vitis High-Level Synthesis (HLS) to be converted to hardware using Vitis Kernel Flow. The produced bitstream file, along with the weights and images files, are loaded into the FPGA and, finally, a C program running on Zedboard's ARM launches the hardware accelerator and collects the results.

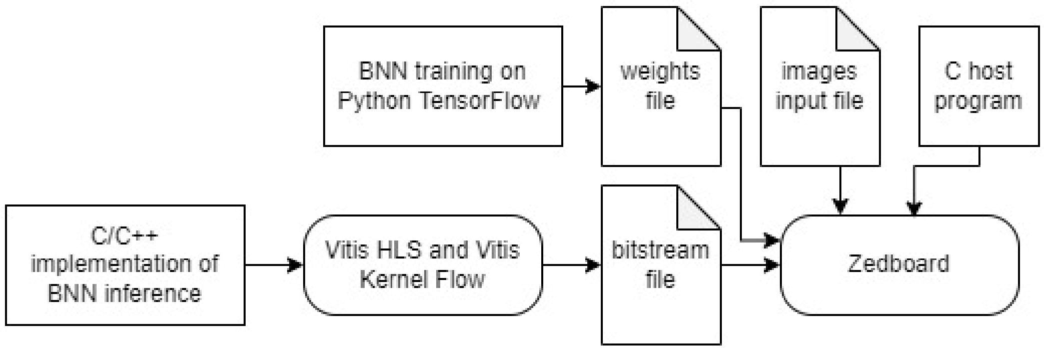

BNN training on Python TensorFlow
weights file
images input file
C host program
C/C++ implementation of BNN inference
Vitis HLS and Vitis Kernel Flow
bitstream file
Zedboard

Figure 3.4: Design flow of this work's BNN accelerator

3.5 Hardware architecture and implementation

Once the code is imported in Vitis HLS, several actions need to be taken to improve latency and throughput and manage hardware utilization area and, therefore, power consumption. Many optimizations can be applied to achieve less memory accesses and parallelism. Mainly, they are given as directives to the synthesizer tool, but quite often code transformations need to be done as well.

In a NN inference, there are specific parts that can be optimized to consume less time. Regarding latency, one direct optimization is the parallelization of the computations by using more hardware components. The inference requires a lot of computations and in fact, many of them can be done in parallel. And for the rest of them, they can be executed in a pipeline. Pipeline is a very strong technique for loops and functions. It boosts overall latency and throughput, by dividing the loop/function body into smaller blocks of instructions, forming several stages. So, in the optimal case, one loop iteration or function call starts in each clock cycle and another ends in each cycle. Similar to pipelining, ensuring a continuous data flow between layers is another vital optimization for NN inference. More specifically, each layer in a NN without feedback consumes data from the previous layer and produces data for the next one. In this way, each layer can work on different input, improving throughput. It has to be noted that the above methods that reduce execution time, result in a larger hardware area.

To optimize the area, one solution is to decrease the bit-length of the data. That, of course, is a trade-off of accuracy and correctness.

All the above optimizations, regarding the BNN inference, will be discussed in detail in the next subsections.

3.5.1 Optimizations in the top-level function

The top-level function of the hardware accelerator is responsible for the overall inference. Firstly, it takes the images as input, along with all the layers' weights, which are sent to the accelerator by the ARM processor. After that, it iterates through each image and feeds it to the first layer by passing it as argument to the relative function call. The results of the first layer are then passed to the next layer, and this continues until the last layer produces the BNN's output. The output is stored in a local BRAM first and once ready, it is sent to the DRAM.

The main optimization in the top-level function is the DATAFLOW directive. More specifically, DATAFLOW is used in the for-loop that iterates images. In this way, it creates separate hardware processes for each of the functions inside the loop body. These functions are the $fetch\_input$ function, which fetches an image from the main memory and stores it in local memory, the layer functions, which execute each layer's computations, and the $write\_output$ function, which sends the output from local memory to the main memory. All these stages are independent of each other for different input images, and there comes the benefit of being separated into processes that run concurrently. That means that when the inference for the first image is in the second layer stage, the second image is in the first layer stage and the third image is in the $fetch\_input$ stage. Algorithm 1 and Figure 3.5 help illustrate the optimized throughput achieved by DATAFLOW in $FC3L\_net$. The same idea also applies in LeNet-5.

Algorithm 1 Top-level function for $FC3L\ net$

```
1: Procedure fc3l_net (input $images$, layer weights $w^1$, $w^2$, $w^3$, output $\overline{\mathbb{E}[h^3]}$, $\overline{\mathbb{V}[h^3]}$)
2: for each $image$ in $images$ do
3:     #pragma HLS DATAFLOW
4:     fetch_input($image$, $image_{loc}$);   // the image is fetched to local memory
5:     fc($image_{loc}$, $\bar{0}$, $w^1$, $\overline{\mathbb{E}[h^1]}$, $\overline{\mathbb{V}[h^1]}$);   // first layer takes $image_{loc}$ as $\overline{\mathbb{E}[h^0]}$ and zeros as $\overline{\mathbb{V}[h^0]}$
6:     fc($\overline{\mathbb{E}[h^1]}$, $\overline{\mathbb{V}[h^1]}$, $w^2$, $\overline{\mathbb{E}[h^2]}$, $\overline{\mathbb{V}[h^2]}$);   // second layer takes first layer's output as input
7:     fc($\overline{\mathbb{E}[h^2]}$, $\overline{\mathbb{V}[h^2]}$, $w^3$, $\overline{\mathbb{E}[h^3]}_{loc}$, $\overline{\mathbb{V}[h^3]}_{loc}$);   // third layer takes second layer's output as input
8:     write_output($\overline{\mathbb{E}[h^3]}_{loc}$, $\overline{\mathbb{V}[h^3]}_{loc}$, $\overline{\mathbb{E}[h^3]}$, $\overline{\mathbb{V}[h^3]}$);   // write output to main memory
9: end for
10: End Procedure
```

By default, when working with Vitis Kernel Flow, all input and output arrays share the same bundled bus to communicate with the main memory. DATAFLOW directive, however, fails to be implemented, unless each function/process uses a separate memory bus. The specification of different memory buses for each array argument has been achieved by the INTERFACE directive, which paved the way for DATAFLOW to be successful.

3.5.2 Optimizations inside layer functions

Fully-connected and 2D convolutional layers are implemented in separate functions and have their own optimizations, even though their architecture (Figure 3.6) shares the same

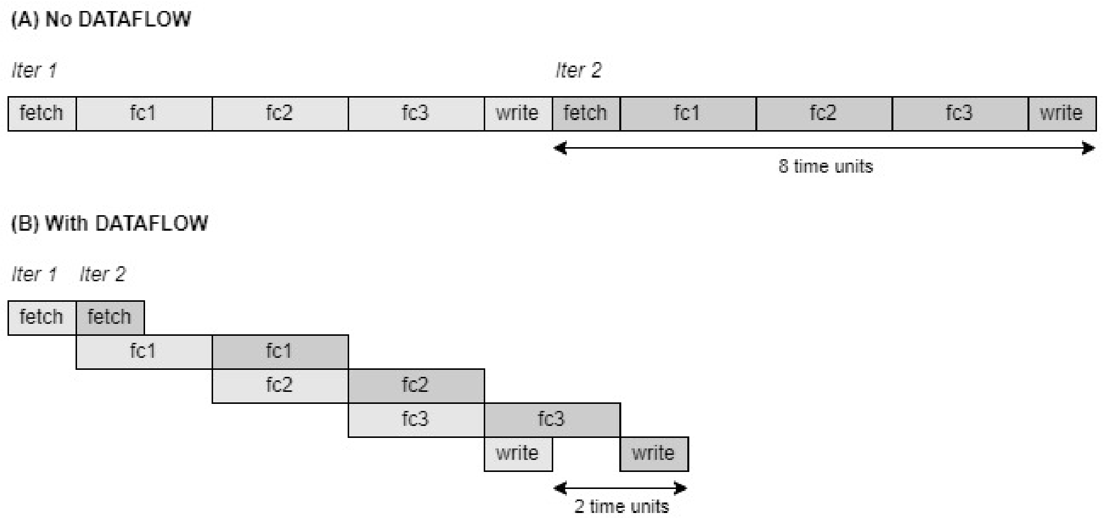


Figure 3.5: Throughput optimization by DATAFLOW directive.
Each function becomes a separate process. The time interval between the end of two consecutive iterations is reduced.

basic modules. The main optimization for these layers is loop PIPELINE.

Starting with fully-connected layer, there is a nested for-loop that iterates through each input and output neuron and performs the needed computations, as it can be seen in Algorithm 2. Essentially, the algorithm takes one input and it uses it to calculate a partial sum of all the output neurons. After it processes the last input, all the output neurons have accumulated their final pre-activation values. PIPELINE directive is applied to the inner loop and achieves initiation interval (II) of 1, meaning that a new loop iteration is launched in each clock cycle. PIPELINE is similar to DATAFLOW, since it divides the loop body into pipeline stages, which can be executed at the same time, like processes in DATAFLOW.

The order of the nested loops in Algorithm 2 could be easily inverted. The alternative algorithm 3 that occurs by that, iterates through output neurons and for each one, it uses the entire input to define its value. In this way, the storage of partial sums for all output neurons is not needed, since only one output is calculated each time. That change means that the inner loop, which is to be pipelined, iterates through the input and aggregates one output neuron's value. However, that causes a dependency between inner loop iterations, which is not ideal for PIPELINE. With the order of nested loops suggested in Algorithm 2, each iteration of the inner pipelined loop updates the value of a different output neuron, which leads to an efficient PIPELINE with II of 1. Another benefit with this approach is that the input is used serially,

Algorithm 2 Fully-connected layer function

```
1: Procedure fc (input $\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$, layer weights $w^l$, output $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$)
2: initialize($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$);  // initialize pre-activation means to biases ( $\overline{b^l}$ ) and variances to 0
3: for each ($\mathbb{E}[h_j^{l-1}]$, $\mathbb{V}[h_j^{l-1}]$) in ($\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$) do
4:     for each ($\mathbb{E}[x_i^l]$, $\mathbb{V}[x_i^l]$) in ($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$) do
5:         #pragma HLS PIPELINE
6:         $w\_pos$ = get_weight_position($j$, $i$);
7:         $\mathbb{E}[x_i^l]$ += $w_{\mu,w\_pos}^l * \mathbb{E}[h_j^{l-1}]$;
8:         $\mathbb{V}[x_i^l]$ += $w_{v,w\_pos}^l * \mathbb{V}[h_j^{l-1}] + w_{\mu^2,w\_pos}^l * \mathbb{E}[h_j^{l-1}]^2$;
9:     end for
10: end for
11: moment_propagation_through_activation_function($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$, $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$);
12: End Procedure
```

meaning that no input is read multiple times.

Algorithm 3 Alternative non-optimal fully-connected layer function

```
1: Procedure fc_alternative (input $\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$, layer weights $w^l$, output $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$)
2: for each ($\mathbb{E}[x_i^l]$, $\mathbb{V}[x_i^l]$) in ($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$) do
3:     initialize($\mathbb{E}[x_i^l]$, $\mathbb{V}[x_i^l]$);  // initialize pre-activation mean to bias ( $b_i^l$ ) and variance to 0
4:     for each ($\mathbb{E}[h_j^{l-1}]$, $\mathbb{V}[h_j^{l-1}]$) in ($\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$) do
5:         #pragma HLS PIPELINE
6:         $w\_pos$ = get_weight_position($j$, $i$);
7:         $\mathbb{E}[x_i^l]$ += $w_{\mu,w\_pos}^l * \mathbb{E}[h_j^{l-1}]$;
8:         $\mathbb{V}[x_i^l]$ += $w_{v,w\_pos}^l * \mathbb{V}[h_j^{l-1}] + w_{\mu^2,w\_pos}^l * \mathbb{E}[h_j^{l-1}]^2$;
9:     end for
10:     moment_propagation_through_activation_function($\mathbb{E}[x_i^l]$, $\mathbb{V}[x_i^l]$, $\mathbb{E}[h_i^l]$, $\mathbb{V}[h_i^l]$);
11: end for
12: End Procedure
```

As for the 2D convolutional layer, it shares the same key algorithmic steps with the fully-connected layer, i.e. output initialization, main computation consisting of nested loops and moment propagation through activation function. Again, the order of the nested loops is selected to be input iterations in the outer loop and output iterations in the inner loop, for the inner loop to be properly pipelined, as described earlier. Originally, there were 6 nested loops, which were unified into 2 nested loops, as shown in Algorithms 4 and 5 in Appendix B. In this way, not only the code has been clearer to the synthesizer, but also the PIPELINE directive has been applied in a loop with a larger body and number of iterations. Although PIPELINE can be used in the outer loop, it has been intentionally avoided, because in that case it fully unrolls all nested loops, resulting in significantly larger hardware area, which is not always acceptable. Finally, for II of 1 to be achieved, the DEPENDENCE directive needed to be used to instruct the compiler that the dependency found on updating output between successive iterations is false, since, in fact, each iteration updates a different output element.

The Average Pooling layer that is part of the LeNet-5 inference has nearly 100 times lower latency than the 2D convolutional layer. Since any latency optimization applied to it would increase the area without a distinctive difference in overall inference time, the Average Pooling layer has been intentionally left unoptimized.

3.5.3 Area optimization with fixed-point bit-widths of variables

The original BNN model uses 32-bit floating-point numbers to represent layer inputs, weights and outputs. Although floating-point arithmetic is quite standard in C and generally in software, fixed-point bit-widths can be more efficient in hardware. The decrease of bit-widths results not only in smaller storage size of variables, but also in smaller and faster hardware operators. Consequently, the latency can be optimized as well.

Of course, reducing the bit-width of variables limits both the range of numbers that can be represented and their precision. These affect the precision of the application, which, in this case, is translated to BNN accuracy. To fully benefit from low bit-widths without a huge impact in accuracy, input and batch normalization during training and inference are good techniques for the model to retain relatively small values that can be represented by less bits without significant precision loss.

In practice, 17-bit fixed-point representation proved to be the golden mean between area and accuracy, with the former almost halved and the latter degrading by 0.1% – 2%. By all means, the fixed-point approach is very sensitive to the application, meaning that experiments need to be done each time to find the optimal number of bits. Figure 3.7 illustrates the impact of various bit-widths to the overall accuracy and aPE metrics.

3.5.4 Further memory optimizations

One probable bottleneck that has not been resolved during optimization is the memory accesses for the network weights. In this work, the weights are coming from outside the hardware accelerator, as they are arguments in the top-level function (Algorithm 1). That means that the ARM CPU of the FPGA sends the weights from the DRAM to the hardware kernel by passing them as arguments when it launches the kernel. This procedure is more time-consuming compared to the case that the weights are already inside the accelerator. The latter can be achieved if the weights are stored in static arrays in a design header file [5]. In this way, the weights would be implemented as BRAMs, more specifically as ROMs, meaning that the access to them would have been faster. It has to be considered, though, that the weights would fill a lot of hardware area.

Unfortunately, some obstacles have prevented the above method from being implemented. The main problem is that synthesis on Vitis HLS stalls at a very early stage for a long time. This issue may be caused to some extent by ”ap_fixed” class that is responsible for the implementation of fixed-point low-bit numbers. The replacement of these with standard floating-point numbers led to a successful synthesis. However, the area overhead of storing floating-point weights to BRAM was huge, forcing the implementation step to fail.

Although the idea of the weights being stored inside the accelerator seems to be appealing for latency optimization, it leads not only to greater hardware area but also to lack of reparameterization after deployment on FPGA. In the contrary, when the weights, along with the network input, are received by the accelerator from an external source, the accelerator can be used for inference on different datasets that need different weights. This may not be a huge advantage in the case of FPGAs, because a new accelerator with different weights can be implemented relatively fast and with no extra cost, but if the hardware accelerator was intended to be implemented in ASIC, reconfigurability and, therefore, reusability would matter more.

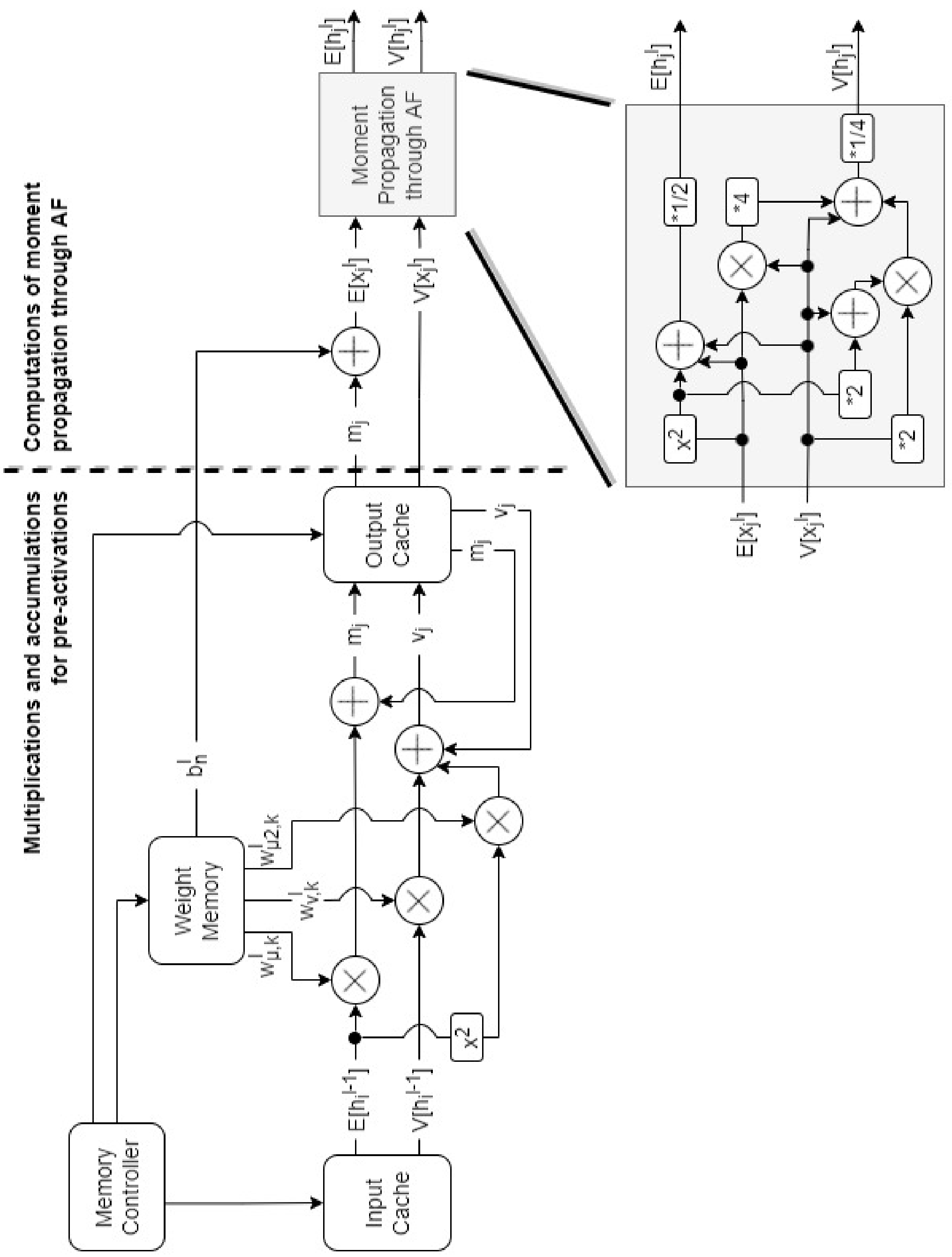


Figure 3.6: Architecture schematic for a convolutional or a fully-connected layer

The architecture is separated in two modules. The multiplication and accumulation module is responsible for the dot products and/or the convolution operations included in equations 2.6. A memory controller defines which input and weights are read and which output is updated. After the first module finishes, the computation of moment propagation through the desired activation function is performed. For this figure, activation function $f(x) = \frac{1}{2}(x^2 + x)$ is used, so equations 3.1 and 3.4 with $a = 1$ are computed. This architecture is indicative and does not necessarily match the generated design by Vitis HLS.

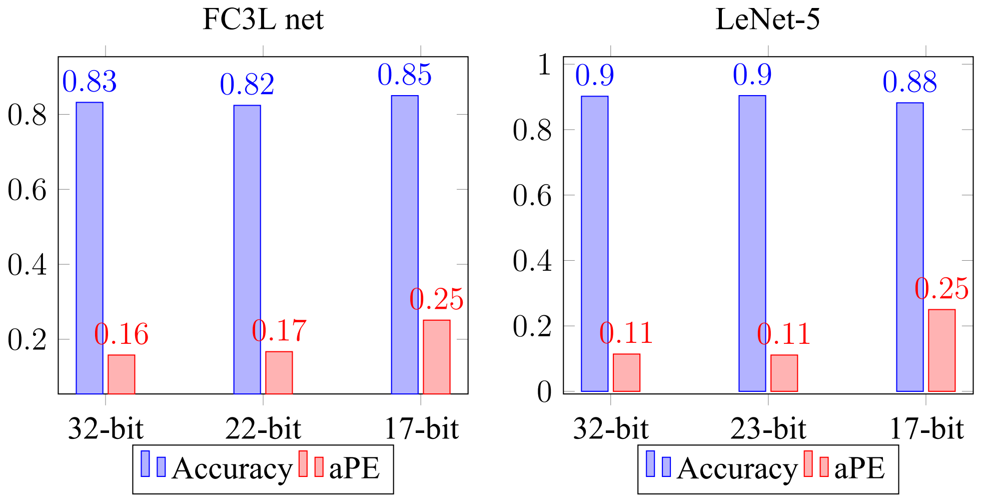


Figure 3.7: Accuracy and aPE for FC3L net and LeNet-5 over different bit-widths

Measurements refer to the first 1000 images of Fashion MNIST dataset. The activation function used is $f(x) = \frac{1}{2}(x^2 + x)$. Normally, accuracy is expected to drop when the bit-width decreases, as happens in the right figure for LeNet-5. The accuracy rise in the left figure for FC3L net should be considered coincidental.

Chapter 4

Experiments and results

The software-only measurements were obtained on a system equipped with an Intel core i7 @ 2.9 GHz CPU and a 8 GB RAM. Moreover, for the hardware implementation, we used the Xilinx Zedboard evaluation board, which includes an ARM Cortex A9 @ 667 MHz Processing System (PS) and Programmable Logic (PL). Finally, the PL includes 220 DSP slices, 106400 Flip-Flops, 53200 LUTs, and 560KB BRAMs.

4.1 Neural network benchmarks and datasets

Two NNs and three datasets have been targeted for the experiments. The first NN is the 3-layer fully-connected network (referred as $FC3L\ net$) used in [4] and [5], which consists of 784 (28*28) input neurons, a couple of hidden layers with 200 neurons each, and an output layer of 10 neurons. Each hidden layer is also fused with a Batch Normalization (BN) layer (Figure 4.1). FC3L net has been implemented twice, once with $f(x) = x^2$ as activation function and the other time with $f(x) = \frac{1}{2}(x^2 + x)$. The datasets used in this network are MNIST [25] and Fashion MNIST [28]. The former is a dataset for handwritten digits recognition, while the latter is a MNIST-like dataset of fashion products. They both contain 60000 28x28 grayscale images in the training set and 10000 images in the test set.

The second NN that has been used is LeNet-5 [7]. LeNet-5 is a CNN consisting of 5 layers, with the first 3 being convolutional and the last 2 being fully-connected. Average pooling layers follow the first two convolutional layers. BN layers have also been added before each activation function (Figure 4.2). Originally, LeNet-5 uses tanh as activation function, but, for this work, this has changed to the polynomial tanh approximation $x - \frac{1}{3}x^3$. The previous

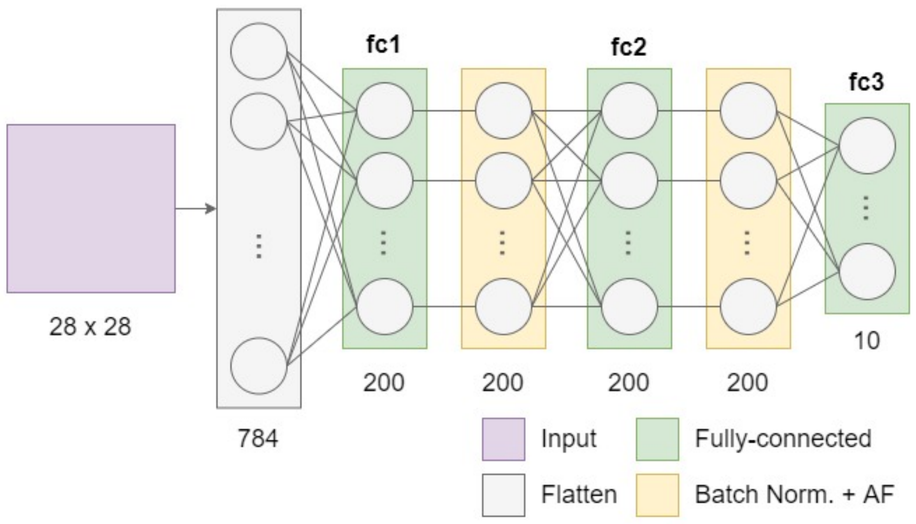


Figure 4.1: FC3L net architecture

polynomial activation functions (x^2, $\frac{1}{2}(x^2+x)$) have also been tested on this network. Another modification to the original network is the input size, which has changed from 1x32x32 meant for greyscale images, to 3x32x32 for colored images. In particular, SVHN [29] has been used as a colored-image dataset since it contains real-world street view house numbers in a MNIST-like format. SVHN has 73257 digits for training and 26032 digits for testing. MNIST and Fashion MNIST have also been tested in LeNet-5. Examples of images for these datasets can be seen in Figure 4.3. It has to be noted that LeNet-5 was not designed to be Bayesian, but by the replacement of scalar weights with distributions, it is converted to a BNN.

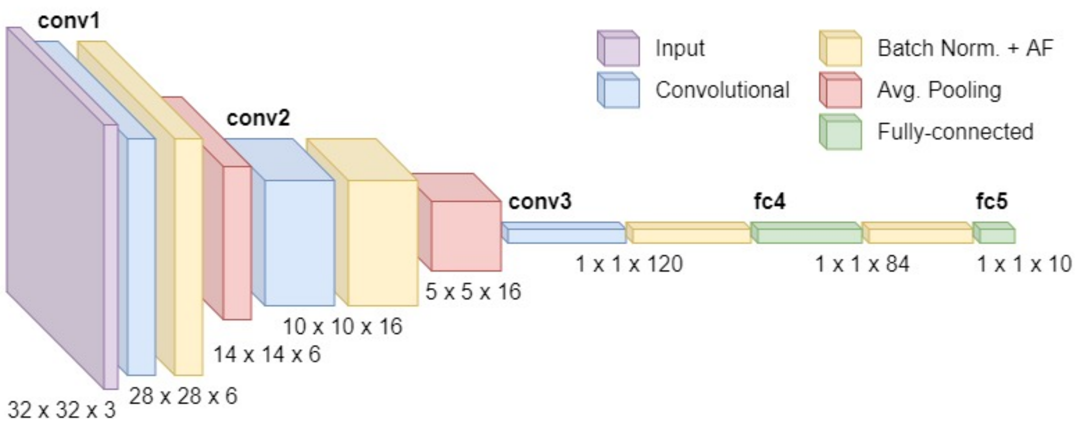


Figure 4.2: Modified LeNet-5 architecture

This work aimed to experiment on even more complicated networks, like VGG-11, an 11-layer CNN. However, the training of such networks suffers from the gradient explosion

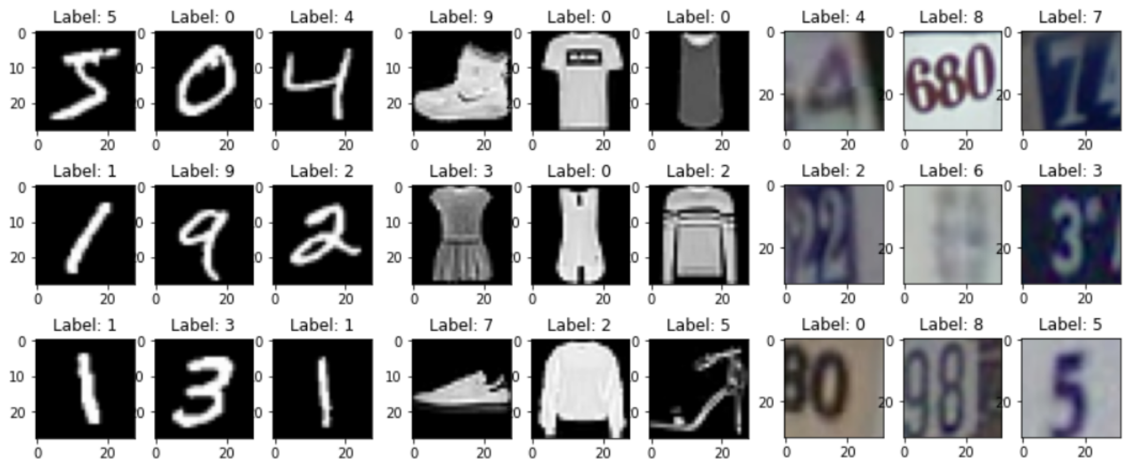


Figure 4.3: Example images of datasets

MNIST [25] (left), a dataset of handwritten digits, Fashion MNIST [28] (center), a dataset of MNIST-like fashion products categorized in 10 classes, such as T-shirt/top, pullover, sandal, etc., and SVHN [29] (right), a real-world dataset of street numbers that contain distractions apart from the central digit of interest.

problem when polynomial activation functions are used. Gradient clipping [30] has been applied to resolve the issue, but it worked efficiently only in LeNet-5, not VGG-11. It has to be noted that the BNN training is out of the scope of this work, so no more effort has been given to the training of more complex networks. After all, experiments on FC3L net and LeNet-5 have produced insightful results, as will be more evident in the next sections.

4.2 Accuracy and uncertainty of polynomial approximation activation functions

From the experiments held on FC3L net and LeNet-5 during the training phase, valuable information is gained about the effect of different activation functions on model accuracy and uncertainty. Table 4.1 explores the value of these metrics for both networks. More specifically, accuracy in the training and test set has been measured for different combinations of the network, dataset, and activation function. Moreover, uncertainty in the test set and in a random noise set has been evaluated in each case.

Table 4.1 contains some worth mentioning metrics. First of all, it can be seen that the accuracy achieved by the $f(x) = \frac{1}{2}(x^2 + x)$ activation function is in most cases higher than

$f(x) = x^2$, with the gap between the two of them reaching about 6% for the SVHN dataset on LeNet-5. However, in the same case, the accuracy of the polynomial approximation of tanh ($x - \frac{1}{3}x^3$) has been left behind by $\frac{1}{2}(x^2 + x)$. On the other hand, these two are closer to each other in the rest of the cases. The reason is the gradient explosion problem, which has failed to be resolved entirely by the gradient clipping method.

As for the uncertainty of the model for the test set, measured by the aPE metric defined in 3.11, minor changes are observed between different activation functions in most cases. Uncertainty of the model when random noise images are given as input should be distinctively higher than uncertainty with the standard test set images as input. This is true for most LeNet-5 combinations but none of FC3L net cases. That, along with the increased accuracy of LeNet-5, are the outcomes of LeNet-5 being more complex than FC3L net.

Table 4.1: Accuracy and uncertainty for various BNNs and datasets

BNN	Dataset	AF	AccTrain↑	AccTest↑	aPE_test↓	aPE_rand↑
FC3L net	MNIST	x^2	0.9939	0.9821	0.017	0.000
		$\frac{1}{2}(x^2+x)$	0.9931	0.9818	0.019	0.000
	F_MNIST	x^2	0.8853	0.8354	0.170	0.001
		$\frac{1}{2}(x^2+x)$	0.8863	0.8451	0.162	0.013
LeNet-5	MNIST	x^2	0.9960	0.9930	0.009	0.843
		$\frac{1}{2}(x^2+x)$	0.9961	0.9935	0.008	0.438
		$x-\frac{1}{3}x^3$	0.9946	0.9930	0.010	0.465
	F_MNIST	x^2	0.9142	0.8958	0.127	0.701
		$\frac{1}{2}(x^2+x)$	0.9215	0.9036	0.113	0.095
		$x-\frac{1}{3}x^3$	0.9166	0.8941	0.134	0.447
	SVHN	x^2	0.8664	0.8105	0.016	0.790
		$\frac{1}{2}(x^2+x)$	0.8998	0.8693	0.017	0.806
		$x-\frac{1}{3}x^3$	0.8424	0.8126	0.229	0.657

4.3 Power, Performance and Area of the hardware implementations

For the inference, the measurements focus on throughput, utilization area and energy consumption of the accelerated BNNs. Figure 4.4 demonstrates the throughput optimization for both FC3L net and LeNet-5. The hardware implementations of these networks are $\times 3.44$ and $\times 1.61$ times faster than the respective serial C code running on the CPU. At this point, a comparison with parallel C code would be more appropriate if the scope of this work was exclusively the optimization of latency. However, this work aims to propose low-power hardware architectures with decent latency. The proposed architectures achieve a throughput of at least 80 images per second, meaning that even real-time video could be processed. One last realization that needs to be noted about Figure 4.4 is that the throughput remains almost the same for any of the activation functions x^2, $\frac{1}{2}(x^2+x)$ and $x-\frac{1}{3}x^3$. Of course, this affects the hardware area.

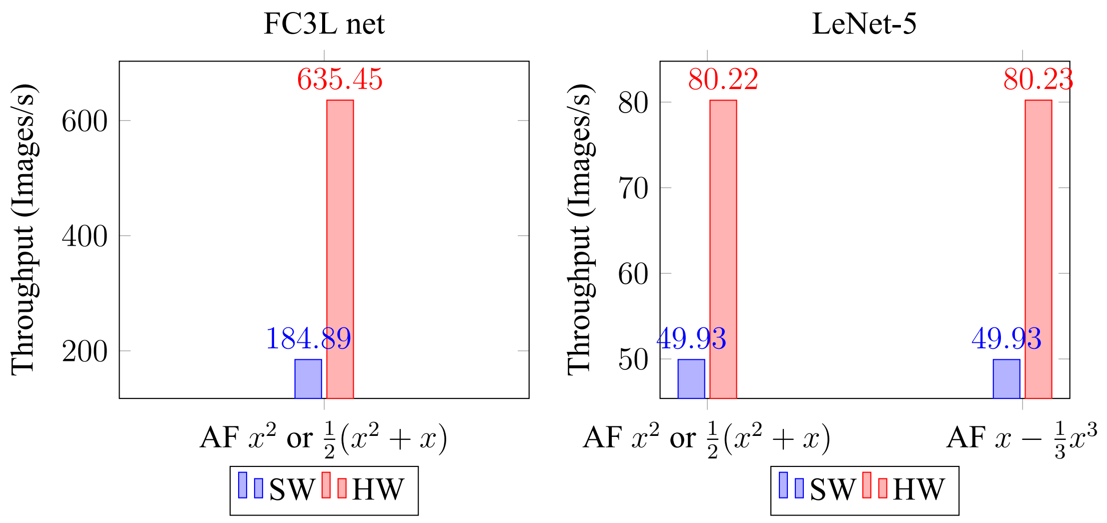


Figure 4.4: Throughput comparison of accelerators and respective serial CPU programs FC3L net hardware implementation has $\times 3.44$ higher throughput than software. LeNet-5 achieves $\times 1.61$ speedup on hardware compared to software.

A goal has been set for each accelerator to achieve the same performance regardless of the activation function, resulting in more hardware components for moment propagation through more computational heavy activation functions. In particular, as Table 4.2 indicates, DSP slices are the only hardware components that present a noticeable increase as the activation function becomes more complex, while Flip-Flops and LUTs slightly rise and BRAM usage remains the same. Lower throughput for more complex activation functions could be an option to preserve the area, but this has not been selected for this work.

Table 4.2: Area and energy of accelerators

	FC3L net with AF $f(x) =$		LeNet-5 with AF $f(x) =$		
	x^2	$\frac{1}{2}(x^2+x)$	x^2	$\frac{1}{2}(x^2+x)$	$x-\frac{1}{3}x^3$
Clock (MHz)	100	100	100	100	100
DSPs	17 (8%)	31 (14%)	35 (16%)	59 (26%)	95 (43%)
Flip-Flops	14189 (13%)	16110 (15%)	27397 (26%)	30681 (28%)	37126 (34%)
LUTs	15980 (30%)	17071 (32%)	29449 (55%)	31193 (58%)	33513 (62%)
BRAM	13%	13%	41%	41%	41%
Power (W)	1.479	1.496	1.636	1.665	1.711

As for power measurements, Xilinx Power Estimator (XPE), a spreadsheet-like tool that estimates the power consumption of FPGA designs [31], has shown that the power slightly increases as the design becomes more hardware demanding. The resulting low power of the designs can be attributed to the low clock frequency and area utilization. Another useful metric that can be obtained by power and throughput is energy expressed in images per Joule, as demonstrated by formula 4.1. For example, the energy for FC3L net with AF $\frac{1}{2}(x^2+x)$ is calculated as $635.45/1.496 = 424.77\ Images/J$.

$$Energy\ (Images/J) = \frac{Throughput\ (Images/s)}{Power\ (W\ or\ J/s)} \tag{4.1}$$

All the above measurements suggest that replacing x^2 by $\frac{1}{2}(x^2+x)$ activation function is generally advantageous. Regarding the experiments, network designs implemented with $\frac{1}{2}(x^2+x)$ had an average accuracy boost of 1.5% compared to x^2. Furthermore, the average rise in power was only $0.024\ W$, with acceptable area increase and no drop in throughput. Finally, focusing on the case of the SVHN dataset used in LeNet-5, merely $0.029\ W$ overhead has been required. At the same time, the design could achieve identical latency and an increase of 5.9% in accuracy.

Chapter 5

Conclusion

This is the final section of the thesis and aims to highlight some key conclusions and suggest some future work.

5.1 Summary and conclusions

This work has established the potential of hardware implementations to apply sampling-free Bayesian Neural Network inference in real world applications. It has shown that the sampling-free Bayesian inference is efficiently implemented in hardware even for Convolutional Neural Networks, with high prospects in image recognition tasks. Polynomial activation functions have been proposed for the replacement of $ReLU$ and $tanh$ activation functions, and they have achieved great accuracy. Activation function $f(x) = \frac{1}{2}(x^2 + x)$ managed to surpass the accuracy of the previously suggested quadratic activation function ($f(x) = x^2$) for the replacement of $ReLU$ by up to 5.9% with negligible additional hardware area. Finally, the hardware implementations of the sampling-free Bayesian inference achieved satisfactory throughput with low area utilization and power consumption.

5.2 Future work

Although this work has focused on a low power hardware architecture, some future work could aim to achieve optimal performance. A promising idea for this goal could be the implementation of an efficient weight caching mechanism that prevents multiple requests of the weights from DRAM, but also tries not to exceed the available hardware area.

Bibliography

[1] Boukaye Boubacar Traore, Bernard Kamsu-Foguem, and Fana Tangara. Deep convolution neural network for image recognition. *Ecological Informatics*, 48:257–268, 2018.

[2] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors*, 19(9), 2019.

[3] Suyel Namasudra, S. Dhamodharavadhani, and R. Rathipriya. Nonlinear neural network based forecasting model for predicting covid-19 cases. *Neural Processing Letters*, 2021.

[4] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. VIBNN. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, mar 2018.

[5] Hiromitsu Awano and Masanori Hashimoto. Bynqnet: Bayesian neural network with quadratic activations for sampling-free uncertainty estimation on fpga. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1402–1407, 2020.

[6] Hongxiang Fan, Martin Ferianc, Miguel Rodrigues, Hongyu Zhou, Xinyu Niu, and Wayne Luk. High-performance fpga-based accelerator for bayesian neural networks, 2021.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[8] Arm - what is an fpga? `https://www.arm.com/glossary/fpga`. Visited on: 20-07-2022.

[9] Xilinx - what is an fpga? `https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html`. Visited on: 20-07-2022.

[10] David Gschwend. Zynqnet: An fpga-accelerated embedded convolutional neural network. Master's thesis, ETH Zürich, Aug. 2016.

[11] Vitis high-level synthesis user guide. `https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf`. Visited on: 29-07-2022.

[12] Bayya Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[13] Fully-connected layers. `https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#fullyconnected-layer`. Visited on: 18-07-2022.

[14] Convolutional layers. `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks`. Visited on: 18-07-2022.

[15] What is a convolutional layer? `https://analyticsindiamag.com/what-is-a-convolutional-layer/`. Visited on: 18-07-2022.

[16] Conv2d: Finally understand what happens in the forward pass. `https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148`. Visited on: 18-07-2022.

[17] Pooling layers. `https://cs231n.github.io/convolutional-networks/#pool`. Visited on: 18-07-2022.

[18] Comprehensive guide to different pooling layers in deep learning. `https://analyticsindiamag.com/comprehensive-guide-to-different-pooling-layers-in-deep-learning/`. Visited on: 18-07-2022.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[20] Batch norm explained visually — how it works, and why neural networks need it. `https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739`. Visited on: 19-07-2022.

[21] Dropout regularization in neural networks: How it works and when to use it. `https://programmathically.com/dropout-regularization-in-neural-networks-how-it-works-and-when-to-use-it/`. Visited on: 19-07-2022.

[22] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, New York, USA, 20–22 Jun 2016. PMLR.

[23] Vikram Mullachery, Aniruddh Khera, and Amir Husain. Bayesian neural networks, 2018.

[24] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, 2022.

[25] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[26] Ramy E. Ali, Jinhyun So, and A. Salman Avestimehr. On polynomial approximations for privacy-preserving and verifiable relu networks, 2020.

[27] Samer Hijazi, Rishi Kumar, Chris Rowen, et al. Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, 9, 2015.

[28] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[29] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[30] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity, 2019.

[31] Xilinx power estimator (xpe). `https://www.xilinx.com/products/technology/power/xpe.html`. Visited on: 28-07-2022.

APPENDICES

Appendix A

Proof for Taylor expansion of $tanh$

The definition of the Taylor series of a function in the origin ($x_0 = 0$) is the following:

$$f(0) + f'(0) * x + \frac{f''(0)}{2!} * x^2 + \frac{f'''(0)}{3!} * x^3 + \dots \tag{A.1}$$

Now for $f(x) = tanh(x)$, the derivatives are:

$$\begin{aligned} f'(x) &= 1 - tanh^2(x) = 1 - f^2(x), \\ f''(x) &= -2f(x)f'(x), \\ f'''(x) &= -2\left[f(x)f''(x)f(x) + f'(x)f'(x)\right] = -2\left[f^2(x)f''(x) + (f'(x))^2\right] \end{aligned} \tag{A.2}$$

Evaluating these derivatives at $x = 0$ and substituting on A.1, we have:

$$\begin{aligned} 0 + 1 * x + \frac{0}{2!} * x^2 + \frac{-2}{3!} * x^3 + \dots = \\ = x - \frac{1}{3} * x^3 + \dots \end{aligned} \tag{A.3}$$

By keeping the first two nonzero terms of the Taylor series, the following polynomial approximation of $tanh$ is obtained:

$$tanh(x) \approx x - \frac{1}{3} * x^3 \tag{A.4}$$

Appendix B

Algorithms for 2D convolutional layers

Algorithm 4 Six-loop approach for 2D convolutional layer

```
1: Procedure conv2D (input $\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$, layer weights $w^l$, output $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$)
2: initialize($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$);
3: for $ci$ in $CHANNELS\_IN$ do
4:     for $row$ in $INPUT\_DIM$ do
5:         for $col$ in $INPUT\_DIM$ do
6:             $input\_idx$ = get_input_idx($ci$, $row$, $col$);
7:             for $co$ in $CHANNELS\_OUT$ do
8:                 for $f\_row$ in $FILTER\_DIM$ do
9:                     for $f\_col$ in $FILTER\_DIM$ do
10:                        #pragma HLS PIPELINE
11:                        $output\_idx$ = get_output_idx($row$, $col$, $co$, $f\_row$, $f\_col$);
12:                        $weight\_idx$ = get_weight_idx($ci$, $co$, $f\_row$, $f\_col$);
13:                        // perform pre-activation computations
14:                    end for
15:                end for
16:            end for
17:        end for
18:    end for
19: end for
20: moment_propagation_through_activation_function($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$, $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$);
21: End Procedure
```

Algorithm 5 Optimized two-loop approach for 2D convolutional layer

```
1: Procedure conv2D (input $\overline{\mathbb{E}[h^{l-1}]}$, $\overline{\mathbb{V}[h^{l-1}]}$, layer weights $w^l$, output $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$)
2: initialize($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$);
3: for $ci\_row\_col$ in $CHANNELS\_IN * INPUT\_DIM * INPUT\_DIM$ do
4:     $ci = ci\_row\_col \div (INPUT\_DIM * INPUT\_DIM)$;  // where $\div$ is whole division
5:     $rem = ci\_row\_col \mod (INPUT\_DIM * INPUT\_DIM)$;
6:     $row = rem \div INPUT\_DIM$;
7:     $col = rem \mod INPUT\_DIM$;
8:     $input\_idx$ = get_input_idx($ci$, $row$, $col$);
9:     for $co\_f\_row\_f\_col$ in $CHANNELS\_OUT * FILTER\_DIM * FILTER\_DIM$ do
10:        #pragma HLS PIPELINE
11:        $co = co\_f\_row\_f\_col \div (FILTER\_DIM * FILTER\_DIM)$;
12:        $rem = co\_f\_row\_f\_col \mod (FILTER\_DIM * FILTER\_DIM)$;
13:        $f\_row = rem \div FILTER\_DIM$;
14:        $f\_col = rem \mod FILTER\_DIM$;
15:        $output\_idx$ = get_output_idx($row$, $col$, $co$, $f\_row$, $f\_col$);
16:        $weight\_idx$ = get_weight_idx($ci$, $co$, $f\_row$, $f\_col$);
17:        // perform pre-activation computations
18:    end for
19: end for
20: moment_propagation_through_activation_function($\overline{\mathbb{E}[x^l]}$, $\overline{\mathbb{V}[x^l]}$, $\overline{\mathbb{E}[h^l]}$, $\overline{\mathbb{V}[h^l]}$);
21: End Procedure
```