KYUSHU INSTITUTE OF TECHNOLOGY

Mixed-precision Weights Network for Field-programmable Gate Array

FUENGFUSIN NINNART

18899039

A thesis submitted in partial fulfillment for the
degree of Doctor of Engineering

TAMUKOH - LABORATORY
DEPARTMENT OF LIFE SCIENCE AND SYSTEMS ENGINEERING
GRADUATE SCHOOL OF LIFE SCIENCE AND SYSTEMS ENGINEERING
KYUSHU INSTITUTE OF TECHNOLOGY

September 2021, Japan

Declaration of Authorship

I, Ninnart Fuengfusin, declare that this thesis titled, 'Mixed Precision Weight Network for Field-Programmable Gate Array' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a doctoral degree at Kyushu Institute of Technology.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

KYUSHU INSTITUTE OF TECHNOLOGY

Abstract

DEPARTMENT OF LIFE SCIENCE AND SYSTEMS ENGINEERING
GRADUATE SCHOOL OF LIFE SCIENCE AND SYSTEMS ENGINEERING
KYUSHU INSTITUTE OF TECHNOLOGY

Doctor of Engineering

FUENGFUSIN NINNART 18899039

In this study, I introduced a mixed-precision weights network (MPWN), which is a quantization neural network that jointly utilizes three different weight spaces: binary $\{-1, 1\}$, ternary $\{-1, 0, 1\}$, and 32-bit floating-point. I further developed the MPWN from both software and hardware aspects. From the software aspect, I evaluated the MPWN on the Fashion-MNIST, CIFAR10, and ILSVRC 2012 datasets. I systematized the accuracy sparsity bit score, which is a linear combination of accuracy, sparsity, and number of bits. This score allows Bayesian optimization to be used efficiently to search for MPWN weight space combinations. From the hardware aspect, I proposed XOR signed-bits to explore floating-point and binary weight spaces in the MPWN. XOR signed-bits is an efficient implementation equivalent to the multiplication of floating-point and binary weight spaces. Using the concept from XOR signed bits, I also provide a ternary bitwise operation that is an efficient implementation equivalent to the multiplication of floating-point and ternary weight space. To demonstrate the compatibility of the MPWN with hardware implementation, I synthesized and implemented the MPWN in a field-programmable gate array using high-level synthesis. My proposed MPWN implementation utilized up to 1.68-4.89 times less hardware resources depending on the type of resources than a conventional the 32-bit floating-point model. In addition, my implementation reduced the latency up to 31.55 times compared to 32-bit floating-point model without optimizations.

Keywords: Deep Learning, Quantization Neural Networks, FPGA implementation.

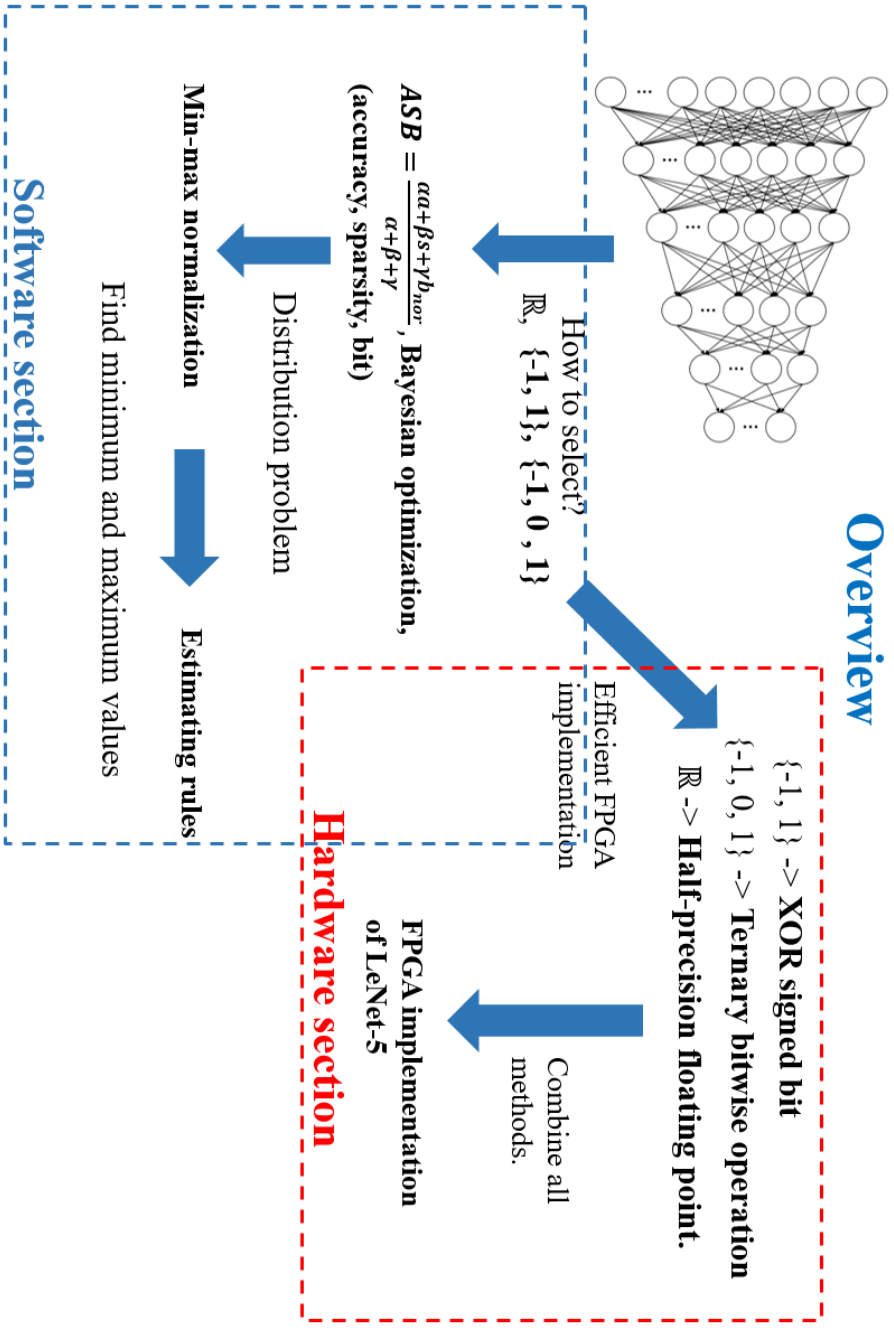

Fig 1. Graphical abstract of this research

Acknowledgements

First of all, I would like to express my appreciation to my supervisor, Professor Hakaru Tamukoh, for advising on my research and study. This dissertation can not be done without his comprehension.

I also would like to thank outstanding members of the Tamukoh laboratory especially, Dinda Pramanta, for encouraging and helping many things related to Japan's life and assisting during the hard times; Yeoh Yoeng Jye for providing knowledgeable insight in both researches and studies. This dissertation can not be done without his verification.

Last but not least, I would like to thank my family for providing suggestions, care, and encouragement during the hard times.

Sincerely,

Ninnart Fuengfusin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Trend of Convolutional Neural Networks

The convolutional neural network (CNN) has attracted attention owing to its abilities to achieve the state-of-the-art results in various computer vision tasks: image recognition [1], semantic segmentation [2], and object detection [3].

One of the advantages of CNN is its scalability which allows it to increase its parameters to operate with larger and more complex data. For example, LeNet-5 [4], one of first CNN models, was proposed with around 60,000 learnable parameters to operate on a handwritten digit dataset, MNIST [5]. The AlexNet [1] model, the winner of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 [6], increased this number of parameters further to 62 million. This number has increased to 144 million (549 MB) with VGG-19 [7], which was designed for ILSVRC 2014.

1.2 Problems of Convolutional Neural Network in Edge Devices

Although the increase in the weight parameters leads to a better performance of the CNN model, this significantly increases the memory consumption to store weight parameters into the device. In addition, storing or loading large models in the dynamic random access memory (DRAM) consumes energy. In general, the information can be stored in the DRAM. Within the DRAM, variables are encoded as bit information (either charged or uncharged) in capacitors. This sequence of bits can be translated into binary codes on demand. However, electric charges in capacitors are leaked constantly; therefore, additional energy is required to maintain the DRAM state. The larger the

model, the more memory access operations (e.g., read and store a variable in DRAM) to perform. This 32-bit memory access operation to DRAM consumes 173 times more energy comparing the 32-bit floating multiplication [8]. This indicates that applying a large-scale CNN directly to edge devices may not be an optimal choice because the edge device has limitations in both its energy and memory storage.

1.2.1 Researches on Convolutional Neural Network for Edge Devices

To solve these problems, various approaches have been proposed, including network pruning [9], knowledge distillation [10], efficient architecture design [11, 12], and quantization neural network (QNN) [13–15]. In this section, I will briefly discuss each approach.

The network pruning [9] was proposed to reduce the model size by zero-out weights that have lower magnitudes than a pre-defined threshold. This causes a large proportion between a number of zero-valued weights to all weights. This property of the distribution biased toward zeros is called sparsity. Using this property with sparse matrix representations, i.e., compressed sparse row (CSR), coordinate list (COO), a 2-dimensional weight array can be decomposed into smaller three one-dimensional arrays. The number of variables in these arrays depends on the magnitude of sparsity. The higher sparsity, the lower number of variables are stored in the memory.

The knowledge distillation [10] is to distill knowledge from a teacher model that has a better performance compared with a student model. By feeding an input sample to the teacher model and receive a prediction from the teacher model, the student learns from the teachers prediction to reduce the error between its output and teachers outputs. By learning how to imitate the teacher model, the student model performs better, and it allows to deploy the student model that usually contains lesser parameters than the teacher model.

The efficient architecture is to discover how to design an efficient neural network (NN) with lesser latency, memory footprint, or multiply-accumulate operation (MAC) while delivering high performance. For instance, SqueezeNet [11] reduces 50 times fewer parameters to AlexNet [1] while provides same accuracy to AlexNet. SqueezeNet provides three strategies to design its NN module. In general, this strategy is to modify its structure by reducing or modifying the size of the filter, the number of input channels, and size of activations.

Another research field is a quantization neural network (QNN). QNN reduces its 32-bit into a fewer-bit representation. An immediate effect is to reduce the overall memory footprint of the model. The conventional data type of a CNN is floating-point, which

is not suitable for hardware implementation due to its complexity. QNN can restrict the floating-point variables to a hardware-friendly datatype, such as the fixed-point. However, to utilize these advantages, specialized hardware may be necessary to exploit the specific data types as the commercial central processing unit (CPU), and graphics processing unit (GPU) do not support fixed-point arithmetic or data types.

1.3 Why Using Field-programmable Gate Array to Implement Neural Networks?

General-purpose programming languages, i.e., C, C++, and Python, are commonly used to construct CNNs. These programming codes are compiled or interpreted into instruction set architectures and transfer to a CPU to perform the computation. CPU has an advantage comparing with other computing devices (i.e., GPU) from its high clock frequency. This indicates that sequential processing is performed faster with the CPU. However, CPU has a disadvantage in processing data that contains rich information, such as images or videos.

To fill CPU disadvantage, GPU has been created to process the graphical information. With the independence between each image pixel, multi-thread parallelism can be utilized to accelerate the visual information. However, the general-purpose programming language cannot be directly used to control GPU. The additional modules to support the parallel programming are necessary. To reduce overall complexity in parallel programming with GPU and NN, many neural network frameworks, i.e., Chainer [16], PyTorch [17], and TensorFlow [18] were created as Python wrappers of CUDA and C++ languages. This decreases an overall development time and prototypes the NN algorithm. However, it still contains some tradeoff in a bottleneck between the communication between the Python wrapper, and C++ compiled binary.

One of the major problems of GPU is power consumption. The cutting-edge GPU can consume up to 300 Watt; therefore, deploying a GPU in an edge device is problematic. One alternative to the GPU is a field-programmable gate array (FPGA). FPGA supports parallelism as same as GPU while provides better power efficiency than the GPU, especially when the complexity of the pipelines grows [19]. Another advantage of an FPGA is that it allows users to define and compute an arbitrary data type. The disadvantage of FPGA over GPU is it requires domain knowledge of hardware to design and utilize FPGA efficiently.

However, deploying floating-point CNN into FPGA is not optimum because the floating-point operations consume a high amount of hardware utilization. A high amount of

floating-point operations may cause a negative slack, or the FPGA design can not achieve a targeted operating frequency. To utilize parallelism while able to maintains lower power consumption and hardware resources, this research focuses on implementing QNN into FPGA.

1.4 Goal of this research

To deploy QNN into FPGA is a suitable choice because QNN provides lower latency and consumes fewer hardware resources. However, with lower precision, QNN comes with a quantization error. The higher the quantization error, the lower the performance of QNN becomes. To handle this QNN performance problem, I propose a mixed-precision weights network (MPWN). The MPWN is designed to deliver a model with a performance close to a conventional 32-bit floating-point model while maintaining a low-bit width and properties of QNN. The MPWN is a QNN that consists of three different weight spaces: binary $\{-1, 1\}$, ternary $\{-1, 0, 1\}$, and, 16-bit floating-point. With three possible weight spaces per weight layer, the search range increases exponentially when the number of layers increases. Each search is expensive, as it requires both time and resources to train the model from scratch. Therefore, finding an optimal combination by random searching may not be effective. In this work, I propose an accuracy sparsity bit (ASB) metric, which quantifies the quality of the MPWN model in terms of three desired properties: accuracy, sparsity, and a number of bit. Defining a scalar score enables searches with Bayesian optimization (BO).

In this study, I utilized Xilinx Vivado HLS (VHLS) [20] for both hardware synthesis and implementation. I demonstrated that by exploiting the weight spaces of the MPWN, the hardware utilization of multiplication could be replaced with XOR-signed bits (XSB) and ternary bitwise operation (TBO) [21]. XSB is a VHLS algorithm for XOR between signed bits of operands. TBO is an XSB with an additional ability to detect a zero. If TBO detects the zero, it will zero-out the output. I demonstrate that XSB, TBO, and a specific data type can be used to significantly reduce overall latency and hardware resources compared with directly implement a floating-point model.

The main contributions of this study are as follows:

- I evaluated the MPWN on the Fashion-MNIST [22], CIFAR10 [23] and ILSVRC 2012 [24] datasets.
- I provided an insightful analysis of MPWN weight spaces and heuristic rules with grid search for all possible combinations of the MPWN.

- I proposed the ASB score, which makes it possible to systematically search for the optimal combination of the MPWN with BO.
- I designed XSB, a replacement for multiplication between floating-point and binary values $1, -1$ for VHLS implementation.
- I synthesized and implemented the MPWN in an FPGA and demonstrated its effectiveness in terms of both latency and area.

1.5 Outline of thesis

This thesis consists of six chapters as follows: introduction, background, related works, experimental results and discussion, and conclusion. Each chapter is detailed in the below sections.

1.5.1 Chapter 1 - Introduction

The introduction section covers the trend of CNN, problems to deploy CNN to edge devices, and the goals of this research.

1.5.2 Chapter 2 - Background

The background section covers fundamental elements of NN, CNN, and FPGA, which will be applied in the experimental results and discussion section.

1.5.3 Chapter 3 - Related Works

The related works section discusses various QNNs and FPGA implementation of QNNs.

1.5.4 Chapter 4 - Mixed Precision Weight Networks

This section covers concepts of the proposed method, MPWN, in both software and hardware directions.

1.5.5 Chapter 5 - Experimental Results and Discussion

The experimental settings and results are described in this section.

1.5.6 Chapter 6 - Conclusion

This section covers the conclusion of this research.

Chapter 2

Background

2.1 Artificial Neural Network

Artificial Neural Network is a mathematical model created to imitate a biological neural network. One of early the NN model can be traced back to 1958 when Perceptron was published [25]. In general, NN consists of neurons. Each neuron connects to other neurons through edges called weights. The weight determines the strength of the connection between neurons. The neurons can receive an input signal and produces the corresponding outputs to other neurons. The output of neurons can be calculated using a summation of the input signal weighted with the weights. Then, these outputs are required to pass through a non-linear activation function.

2.2 Elements within Artificial Neural Networks

This section will cover essential elements within the modern NN, especially in an image classification task.

2.2.1 Hyper-Parameters

To train a NN, there are parameters to be considered to control the behavior of the learning algorithm [26], for instance, learning rates, weight decay, etc. The optimal values of hyper-parameters may be diverse across models and datasets. These parameters are known as hyper-parameters. To discover hyper-parameters can be done via using searching algorithms, such as Bayesian optimization.

2.2.2 Fully-connected Layer

The fully-connected layer is one of the weight layers in NN that connects an input element to all of its weight parameters. The operation of this layer can be represented as matrix multiplication. The fully-connected layer can be summarized into Equation (2.1), where $W \in \mathbb{R}^{R \times C}$,$X \in \mathbb{R}^{C}$, $b \in \mathbb{R}^{C}$, R denotes the number of rows of the weight, and C denotes the number of columns of the weight.

$$F = \sum_{j=1}^{C} \sum_{i=1}^{R} W_{j,i} X_j + b_j \tag{2.1}$$

2.2.3 Convolutional Layer

CNN was first realized from Neocognitron [27]. Neocognitron was created to imitate the primary visual cortex. LeCun et al. further modified Neocognitron into a modern CNN structure which was introduced in the LeNet-5 model [4].

One of the most important elements of CNN is a convolutional layer. The convolutional layer was motivated by the brain's ability to recognize objects from images' patterns and features. Arranging convolutional layers allows CNN to extract low-level features in the early convolutional layer and high-level features in the late convolutional layer. The convolutional operation can be summarized into Equation (2.2), where $W \in \mathbb{R}^{C_{out} \times C_{in} \times K_r \times K_c}$, $X \in \mathbb{R}^{C_{in} \times I_r \times I_c}$, K_r denotes the number of kernel rows, K_c denotes the number of kernel columns, C_{in} denotes the number input channels, C_{out} denotes the number of output channels, I_r is the input activation row, I_c is the input activation column, O_r is the output feature row, O_c is the output feature channel, and F is the feature map.

$$F = \sum_{n=1}^{C_{out}} \sum_{m=1}^{O_r} \sum_{l=1}^{O_c} \sum_{k=1}^{C_{in}} \sum_{j=1}^{K_r} \sum_{i=1}^{K_c} W_{n,k,j,i} X_{k,j+m,i+l} \tag{2.2}$$

By applying a convolutional operation, the output shape may not be the same as the input shape. The output width and height can be found using Equation 2.3. Where Z_{out} can be either an output width or height, Z_{in} can be either an input width and height, C is a size of convolution filter, P is a number of zero paddings and S is a number of strides.

$$Z_{out} = \frac{Z_{in} - C + 2P}{S} + 1 \tag{2.3}$$

The main difference between the convolutional and fully-connected layer is the number of weight parameters within the convolution layer is to a great degree less than the number of weight parameters in a fully-connected layer. For instance, in the case of VGG-16 [7], roughly 10 percent of all weight parameters in VGG-16 are in 13 convolutional layers. The remaining 90 percent is in only three fully-connected layers.

2.2.4 Activation function

An activation function is mainly applied to an output signal of the weight layer. In view of neuroscientific, the activation function is used to imitate the action potential signal process or either to fire or not fire an electric spike into another neuron [28]. Moreover, in the perception of machine learning practitioners, the activation function is required to introduce a non-linear property to NN.

Without the non-linear property, NN makes no difference from a combination of linear combinations. The linear function can approximate only a linear function; however, the non-linear can replicate both linear and non-linear functions [26]. Thus, the linear function is not robust enough to deploy in a real-world environment.

2.2.4.1 Rectified Linear Unit

The rectifier linear units (ReLU) is considered as one of the most widely used activation function in NN. The ReLU can be defined as Equations (2.4) or (2.5). Where x is the input signal, $f(x)$ is an output of ReLU and $max(a, b)$ is a function which returns a maximum variable between a and b.

$$f(x) = \begin{cases} x, & x >= 0 \\ 0, & x < 0 \end{cases} \tag{2.4}$$

$$f(x) = max(x, 0) \tag{2.5}$$

The relation between an input variable x and output variable $f(x)$ can be plotted as shown in Figure 2.1.

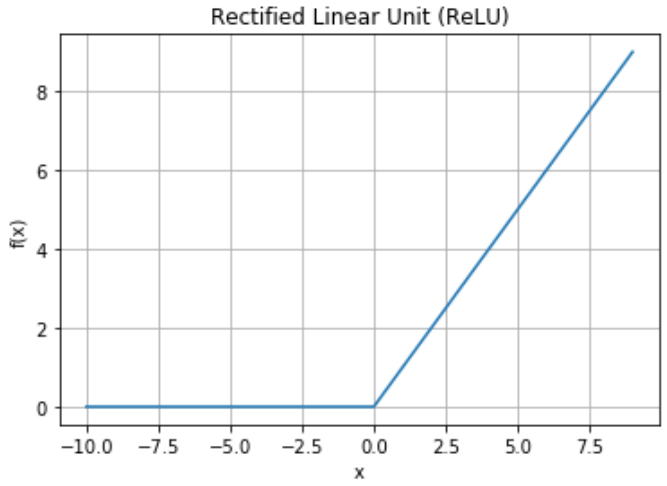

Fig 2.1. Rectified Linear Unit outputs the input if the input is a positive value, otherwise it will output a zero.

The ReLU has advantages comparing with other activation functions. The first reason is ReLU is not saturated from a gradient-vanish problem. The input variable of ReLU in the range $[0, \infty)$ always contains gradients to update with. Compared with the Sigmoid activation function, the gradient will have vanished when input variables are either very positive or negative. This problem may occur when NN has been trained for a long time.

2.2.5 Max-pooling

Max-pooling layer is applied to down-sample or reduce number parameters of activation or feature maps. This reduces overall the memory consumption and number of memory access operations. With less parameters to compute, the training and inference of NN also can be done faster. The max-pooling can be represented as Figure 2.2. In the figure, max pooling with the size of 2×2 and stride of 2 is applied. The output of max pooling has less number of parameters than an input two times.

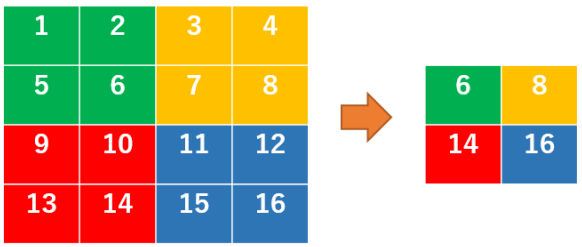

Fig 2.2. Max pooling with size 2×2 and stride 2. Each color represents a region for each max pooling operation. The output of each region represents a maximum value within the input region.

2.2.6 Cost or Loss Function

The cost function is an indicator for determining how well NN operates with a given task. The different cost functions are designed for various tasks or situations. For example, the L2-distance or Euclidean distance for multi-class image classification may not suit this task. This is due to the range of L2-distance that can extend to out-of-range of the number classes. Therefore, some priors are necessary to limit the range output. In this case, softmax with cross-entropy is more preferred to the L2-distance loss.

2.2.6.1 Softmax

The softmax activation function is usually applied in the last layer of NN, especially in the multi-classification task. The softmax squeezes a real number array into the range of $[0, 1]$. Therefore, the softmax output can be defined as probabilities to predict labels correctly. The interpretation comes from a summation of softmax output array always equal to one. The softmax function can be shown in Equation (2.6). Where x is inputs with an order i, n is a number of classes, and $f(x)$ is a output of softmax function.

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.6}$$

Using normalized exponential function encourages behavior that is the magnitude of one confident class is significantly more than other classes. With this behavior, the softmax function assumes that the input data contains only a class and that class is independent of other classes. Otherwise, this may affect the model's performance when objects come from the same categories.

2.2.6.2 Cross-entropy

The cross-entropy was realized from an information theory. Cross-entropy is applied to calculate the loss of information while transmitting and receiving a message. The cross-entropy is assumed that the input is a probability distribution; therefore, it operates well with a softmax activation function that provides the output as a probability distribution. In general, the cross-entropy loss is high if the model wrongly predicts with high confidence and correctly predicts with low confidence. Suppose the training dataset has almost the same distribution as the test data. In that case, NNs can also minimize cross-entropy loss of test data via only learning from the trend of training data.

2.2.7 Optimization

After defining how well the model performs or the cost function, NN must adjust the weight parameters to minimize the cost function. One algorithm to use for optimization is gradient descent.

2.2.7.1 Gradient Descent

The gradient descent is a basic optimization algorithm for multi-layers NN. The gradient descent is realized from the partial-derivatives with respect to learn-able parameters to the cost function. The outcome of partial-derivative is a slope of the cost function with learn-able parameters. This slope can be used to find a direction to tune learn-able parameters to a local minimum of the cost function. This gradient descent can be summarized into Equation (2.7). Where η is a learning rate, C is a cost function, W^{t-1} are weights or biases before updating, W^t are weights or biases after updating.

$$W^t = W^{t-1} - \eta \frac{\partial C}{\partial W^{t-1}} \tag{2.7}$$

The stochastic gradient descent [29] is similar to the gradient descent. The significant difference is that NN is randomly fed with a batch image in each training iteration. With this stochastic process, a converging to the local minimum can be done faster than standard gradient descent.

2.2.7.2 Gradient Descent with Momentum

The gradient descent has a problem in which the gradient may oscillate back and forth in orthogonal directions that are not related to the local minimum direction. This back and forth reduces the overall magnitude of gradients reaching the local minimum direction and consumes longer times to train NN. The gradient descent with momentum [30] was proposed to address this issue. The oscillations can be canceled by averaging the noises in opposite directions. This averaging can be done during training with exponentially weighted averages. The gradient descent with momentum can be represented with Equations (2.8) and (2.9). Where η is a learning rate, V is a global averaged gradient, β is a hyper-parameter recommended as 0.9, C is a cost function, t is a current number of forward propagate.

$$W^t = W^{t-1} - \eta V^{t-1} \tag{2.8}$$

$$V^t = \beta V^{t-1} + (1-\beta)\frac{\partial C}{\partial W^{t-1}} \tag{2.9}$$

2.2.7.3 Adam Optimization

Adam optimization [31] combines two different type of optimizations together, RMSProp [32] and gradient descent with momentum. In some cases, Adam optimization reduces the convergence time comparing with SGD and SGD with momentum. There are still some concerns related to Adam optimization that may be stuck in a local minimum. Therefore, some implementations [33] may utilize with Adam first, then switches to SGD.

2.3 Regularization

The regularization is utilized to address an over-fitting problem when an NN model learns both noise and data. This over-fitting also can be described as when the NN model remembers inputs and corresponding outputs. The learning of both noise and data patterns occurs when the NN has too high a capacity. Therefore, some limitations are necessary to constraint NN.

2.3.1 Dropout

Dropout [34] is to randomly drop neurons or zero-out weights in a particular layer with a probability $1-p$ in each training iteration, where p is a probability of neuron exists. To give an example, a weight layer with a dropout $p = 0.7$ indicates that this weight layer has a possibility of 30 percent drop weight elements. This can be illustrated in Figure 2.3. To ensure the expected value of the output layer remains the same with and without dropping, the output of dropout is scaled with $\frac{1}{p}$.

The dropout is designed to prevent the cooperation between neurons. For instance, a a neuron may make a mistake; however, instead of a neuron reconfigured its own weights. As a result, the b neuron adjusts itself to correct the a neuron output. This cooperation between neurons costs the over-fitting because the b neuron remembers the mistake from the a neuron. In general, the dropout should be applied with low amounts of p to the first and last layer of NN.

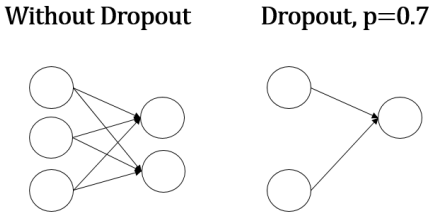

Fig 2.3. Five neurons connected to each other with six weight values. Left: without dropout. Right: with dropout $p = 0.7$.

2.3.2 Batch Normalization

One of the problems within deep NN is an internal covariate shift problem. This problem is defined as distributions of activation in each layer of NN are diverge in both mean and variance during training. For example, since the input to output layers are connected, if one layer activation is shifted, then the following activations are also shifted as shown in Figure 2.4.

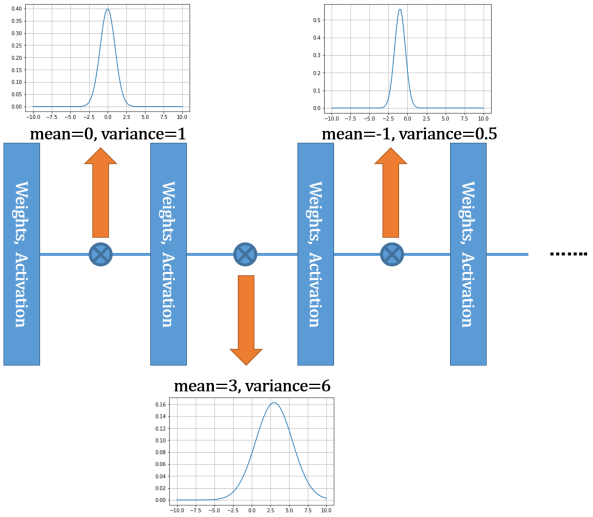

Fig 2.4. The internal covariate shift occurs when distributions of each layer activation of NN are unstable.

Batch normalization (BN) [35] was proposed to solve this problem by normalizing the mean and variance of activation layer-wise into 0 and 1, respectively. This process can be shown in Equation (2.10), where $\hat{x_i}$ is a normalized batch images, x_i is an input batch, μ_B is an mean of batch, σ_B^2 is a variance of a batch and ϵ is a hyperparameter that is used to avoid zero as a denominator.

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.10}$$

BN further utilizes a linear transform using two learn-able parameters β and γ with Equation (2.11). The NN can use these parameters to search for a suitable mean and variance.

$$\hat{y_i} = \gamma \hat{x_i} + \beta \tag{2.11}$$

Since Equations 2.10 and 2.11 operate in the batch-wise direction, to track the mean and variance of activation across of dataset requires a tracking method. This can be done using an exponentially weighted average as shown in Equation (2.12), where μ^t is a global mean after the update, μ^{t-1} is a global mean before the update, β is a hyper-parameter and μ_B is a mean from image batch which used to update the global mean.

$$\mu^t = \beta \mu^{t-1} + (1 - \beta)\mu_B \tag{2.12}$$

2.3.3 L2 Weight Decay

L2 weight decay or L2 regulation discourages magnitude weight parameters by adding a norm L2 summation of weights into the cost function. This constraint causes NN not able to focus on distributing the magnitude of weights to a certain neuron. It requires distributing the magnitude across multi-neurons. This reduces the chance of over-fitting of NN. L2 weight decay is defined as Equation 2.13, where W is all weights in NN and λ is a hyper-parameter.

$$L_R = \frac{\lambda}{2}||W||_2 \tag{2.13}$$

By adding a weight decay term into the cost function, the gradient descent will minimize the magnitude of weights. As a result, the weight distribution with L2 weight decay condenses around zero compared without L2 weight decay.

2.4 Learning Rate Decay

The gradient becomes too large when a learning rate is too high, and the optimization process may not reach a local minimum. However, if the learning rate is too low, the training time will be too long. One of the solutions to solve this problem is to utilize the learning rate decay. In general, the initial learning can be started with a high magnitude

and decrease over time. There are many methods to reduce the learning rate. In this section, I will cover the learning rate step down.

2.4.1 Step Down Learning Rate

After training for n epochs, the learning rate α is set to be reduced with factor of δ as shown in Equation (2.14).

$$\alpha = \frac{1}{\delta}\alpha_0 \tag{2.14}$$

The step-down learning rate can be visualized in Figure 2.5, where the y-axis is a learning rate α and x-axis is an epoch.

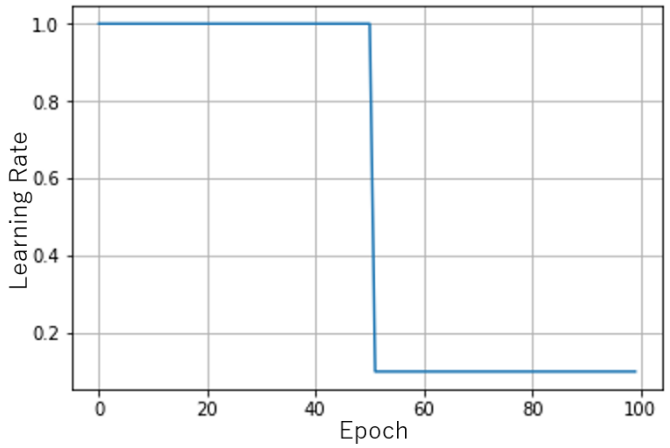

Fig 2.5. The initial rate of 1 and it stepped down to 0.1 with $\delta = 10$ at 50 epochs

2.5 Feed-Forward Neural Network Models

This section covers several feed-forward NN models that I utilized in Chapter 5: Experiments and Discussion section.

2.5.1 Multiple Layers Perceptron

Multiple Layers Perceptron (MLP) is one of the basis NN models, a stack of fully-connected layers. The MLP requires a 1-dimensional input; however, an input image has 2- or 3-dimensions. Therefore, this image cannot be directly fed to MLP. Thus, the input image must be flattened to a dimension to be able to fit in MLP.

2.5.2 LeNet-5

LeNet-5 [4] originally consists of three layers of convolutional layers and two fully connected layers. Therefore, LeNet-5 consists of five weight layers. In addition, there are also spatial reduction layers or max-pooling layers. The LeNet-5 was created for a hand-written digit classification task. In this research, I modified the LeNet-5 architecture to two convolutional layers and three fully-connected layers to operate with the Fashion-MNIST dataset without utilizing any padding.

2.5.3 ResNet-18

ResNet [36] is the CNN architectures that won ILSVRC 2015. The ResNet introduces a skip- or shortcut-connection architecture that allows feature maps to skip a layer set. This allows ResNet architecture to avoid the gradient vanishes problem, which occurs when the NN is too large. One of the proposed ResNet architectures is ResNet-18 that consists of 18 weight layers and two pooling layers without including residual connections. In these 18 layers, there are 17 convolutional layers and one fully-connected layer.

2.6 Benchmark Dataset

Benchmark datasets are designed to measure the performance of NN models. In the field of computer vision, one of the benchmark datasets is the image dataset. The dataset is generally separated into two sections: training and test dataset. The training dataset is for a model to learn, while the test dataset is used to check the ability of the model to generalize. The hyper-parameters may also be causing the over-fitting. To solve this problem, the image dataset may be separated into three sections: training, validation, and test dataset. The validation set is used to tune hyper-parameters.

2.6.1 Fashion-MNIST

Fashion-MNIST [37] is a clothing image dataset that consists of 60,000 training images and 10,000 test images. Each image is a gray-scale image that consists of 28x28 pixels. The labels consist of t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. Fashion-MNIST is created as an alternative to the MNIST dataset; therefore, it is designed to have the same image size and amount of training and test samples as the MNIST dataset. Fashion-MNIST addresses one of the MNIST problems:

the MNIST dataset becomes an unchallenging task for the cutting edge models. Therefore, Fashion-MNIST provides a more challenging task to able to benchmark these models.

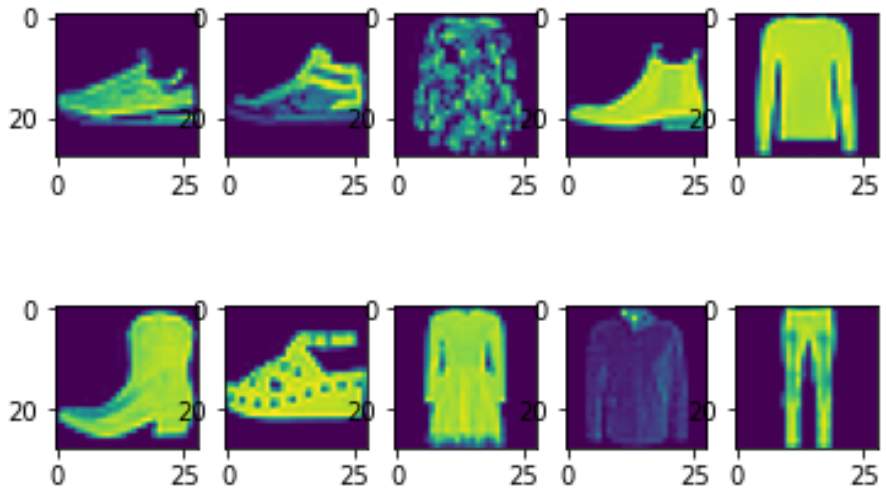

Fig 2.6. 10 randomly selected images from Fashion-MNIST dataset

2.6.2 CIFAR-10

Canadian Institute For Advanced Research 10 (CIFAR-10) [38] is an image dataset that consists of 60,000 images belonging to 10 different classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The data are separated into 50,000 training images and 10,000 test images. The size of each image is $32 \times 32 \times 3$ pixels. The main difference between Fashion-MNIST is the grayscale image dataset which has only a channel. However, each CIFAR-10 image consists of three RGB channels. With more information to process, this causes the CIFAR-10 to consume more time to train with than Fashion-MNIST.

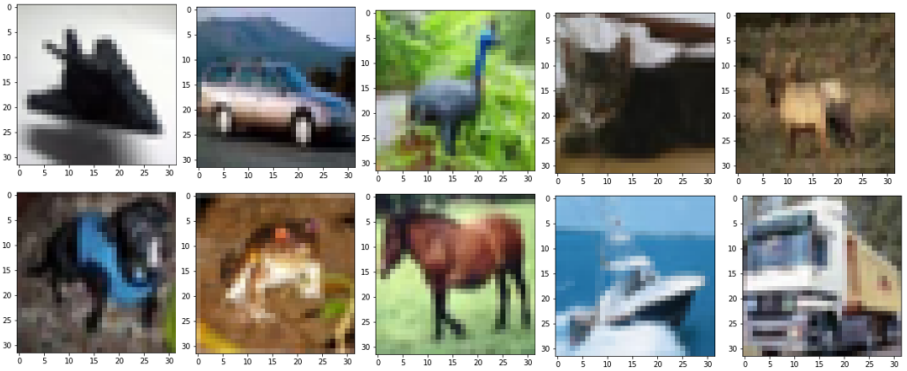

Fig 2.7. CIFAR-10 images with classes in the left-to-right order from airplane, automobile, bird, cat, deer, dog, frog, horse, ship, to truck

2.6.3 ILSVRC 2012

ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC 2012) [6] or ImageNet 2012 is a large-scale image dataset that consists of 1,281,167 training images and 50,000 validation images. Each image belongs to one of thousand classes. The shape of images is varied throughout the dataset; therefore, some of the image pre-processing is necessary to feed these images to the model that requires a fixed shape of the image.

2.7 Image Pre-processing

Raw pixels from the image dataset can directly feed to a NN model, however with the significant loss in terms of performance of the model. Therefore, image pre-processing is necessary. Two basic pre-processing will be covered in this section. The first is a basic image normalization, and the second is an image normalization with statistical information from the training data.

2.7.1 Basic Image Normalization

Basic image normalization normalizes the input image to a range of $[0, 1]$. To normalize the image to this range, maximum pixel intensity is required to be found. In this case, each image is assumed to have a color depth of 8-bit. Hence, the highest value of pixel is $2^8 - 1 = 255$ and this process can be summarized to Equation (2.15), where $\hat{x_i}$ is normalized images, x_i is input images and d is the color depth of image.

$$\hat{x_i} = \frac{x_i}{2^d - 1} \tag{2.15}$$

2.7.2 Normalization with Mean and Standard Deviation of Training Dataset

This normalization is to normalize the input image distribution to a zero-mean and unit standard deviation. By assuming training, validation, and test distributions have identical distributions, the mean and standard deviation of the training dataset can be used to normalize the training, validation, and test datasets. This process can be described using Equation (2.16), where $\hat{x_i}$ is normalized test images, x_i is input images, μ_{train} is a mean of all training images, σ^2_{train} is the standard deviation of all training images and ϵ is a very small number which is used to protect a denominator as zero. If

images contain three channels, the standard practice with the color image dataset is to normalize in the channel-wise direction.

$$\hat{x_i} = \frac{x_i - \mu_{train}}{\sqrt{\sigma^2_{train} + \epsilon}} \tag{2.16}$$

2.8 Quantization Neural Networks

Quantization Neural Network (QNN) is a NN whose weights are constrained with low-bitwidth. This process causes the memory of NN less than usual and allows NN to deploy into edge devices easier. However, the tradeoff of the quantization process is a drop in the performance from quantization loss. Another quantization process is by constraint weights into the hardware-friendly format (fixed-point or integer formats); this allows NN's operations can be done with fewer hardware resources and latency. I will cover QNN in detail in Chapter 3: Related Works.

2.9 Field-programmable Gate Array

FPGA is a programmable integrated circuit consisting of reconfigurable logic blocks, such as look-up tables (LUT) and switches. These logic blocks in FPGA can be re-configured using either a hardware description language (HDL) or high-level synthesis (HLS). In general, LUT can be used to replicate every logic function from its ability to remember the mapping between input and output. By wiring LUT together, these combinations of LUT can imitate an arbitrary digital circuit.

Zynq series is one of the recent series of FPGA provided by Xilinx. One of Zynq series, Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit board, is shown in Figure 2.8. The Zynq cooperates with a processor system (PS) and programmable logic (PL) together [39]. In other words, Zynq has the built-in CPU ARM-Cortex and FPGA chip into the board. This also provides an AXI connection between PS and PL to allow the low latency connection and further accelerate the computing.

Fig 2.8. A image of a FPGA board, Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

2.10 High-level Synthesis

One advantage of an FPGA over the conventional software language is it enables bitwise manipulation. Another advantage is that it allows the user to define, store, and compute an arbitrary data type. However, the disadvantage of an FPGA design over a conventional software design is the long development time. To gain advantages from both software and hardware, high-level synthesis (HLS) has been developed. HLS is a development platform that converts C-like languages (C, C++, and System C) to the hardware description language (HDL) (Verilog and VHDL) language. This allows users to rapidly develop applications with an interface of a C-like language with fewer constraints from the HDL language. With this property, HLS is especially useful when applies HLS with the deep learning algorithm that its state-of-the-art algorithm has been rapidly changed. On the other hand, a drawback of HLS compared to optimized handcrafted HDL is that HLS-generated HDL code may cause a higher latency and hardware utilization. For instance, Ordaz et al. [40] compared between HDL and HLS implementations of cyclic redundancy check (CRC) and found out that HDL implementation consumes less than LUT by 1.58 times and less amount of latency by 2.63 times.

2.10.1 Directives and Hardware Design in Vivado High-level Synthesis

Several metrics for hardware synthesis and implementation include latency, hardware utilization, and power consumption. In terms of hardware resources in recent FPGAs, four fundamental hardware resources are block RAM (BRAM-18K), DSP48F block, flip-flop (FF), and look-up table (LUT).

One advantage of an FPGA over a CPU is that it can be designed to perform large-scale parallelism. Although parallelism dramatically reduces the latency, the trade-off is that it also significantly increases hardware resources. Since Vivado High-level Synthesis (VHLS) receives a C-likes language as input and C-like languages are not designed for hardware implementation, VHLS solves this problem by providing users with hints regarding converting C-type functions into hardware. VHLS provides directives or *pragma* in C-like languages as hints. We can use the directives to VHLS how to manage parallelism, memory, and other aspects.

There are three directives that we utilize throughout this study. The first two directives are related to parallelism, while the third directive is related to memory management. The first directive is the $PIPELINE$ directive. Placing $PIPELINE$ directives after a *for loop* indicates to VHLS that the *for loop* can be accelerated by data pipeline parallelism. The second is the $UNROLL$ directive, which indicates that the computation in the *for loop* can be executed in parallel across the *for loop*. The third directive, the $ARRAY\_PARTITION$ directive, is mostly applied with either $PIPELINE$ or $UNROLL$. Performing parallel computing requires accessing multiple data at the same time. By using $ARRAY\_PARTITION$, a large memory is divided into multiple smaller blocks of memory. Therefore, it can access smaller memory in the same clock cycle without access conflict. With these directives, we can apply parallel computing throughout our model to accelerate computation and reduce latency.

Chapter 3

Mixed Precision Weight Network and FPGA Design

In this chapter, I describe my proposed method, MPWN, in terms of algorithm and hardware design. This includes the MPWN algorithm and its effective design in hardware.

3.1 Mixed-precision Weights Network

Mixed-precision weights network (MPWN) is designed to utilize the advantages of the weight spaces from BC, TWN, and 32-bit floating-point. The MPWN is partly motivated by dropout [34], which is a regularization method that randomly drops weight connections of a neural network in each batch training. Randomly dropping weights reduces the tendency of neurons to cooperate with other neurons. This cooperation between neurons may lead to over-fitting. The optimal dropout rate is not always equal throughout the weight layers. The optimal rate depends on the layer order and the type of weight layer (e.g., convolutional layer, fully-connected layer). The difference in the optimal dropout rate reveals the sensitivity of the weight layer to the constraint. Disturbing a high sensitivity weight layer causes a greater negative effect in terms of performance than disturbing a low-sensitivity layer. Using this concept, the MPWN places different constraints depending on the sensitivity of the weight layers.

For hardware implementation, the performance of the TWN is still acceptable without the scaling factor S_i. Therefore, I utilize the TWN without the scaling factor, which also reduces the overall complexity of the hardware implementation. During inference, the 32-bit floating-point model can be reduced to the 16-bit floating-point without

decreasing in performance. Therefore, I utilize with 16-bit floating-point instead of 32-bit floating-point.

I define the notations for representing MPWN layers as follows: **F** indicates that the layer with 16-bit floating-point or full-precision, **B** indicates that the layer with BC, and **T** indicates that the layer is TWN without the scaling factor. **FBT** refers to a CNN with three layers where the first layer is 16-bit floating-point, the second layer is BC, and the third layer is TWN. The advantage of each weight space is summarized in Table 3.1. Overall, **F** is correlated with the performance of the model. **B** reduces the bit width of model the most, and **T** introduces sparsity into the model.

Table 3.1. Overview of properties of each mixed-precision weights network weight space.

	Accuracy	**Sparsity**	**Bits per Weight (bit)**	**Weight space**
Full precision weights **(F)**	**High**	None	16	$\mathbb{R}$
Ternary weights **(T)**	Mid	**High**	2	$\{-1, 0, 1\}$
Binary weight **(B)**	Low	None	**1**	$\{-1, 1\}$

With several metrics to optimize in the MPWN, I reduce these scores to a single score called the accuracy sparsity bit (ASB) score. The ASB score is a linear combination of accuracy, sparsity, the number of bits, as expressed in Equation (3.1), where a is the model test accuracy, s is the sparsity of the model, and b_{nor} is 1 minus the normalized number of bits. b_{nor} is defined in Equation (3.2), where b_{max} is the number of bits from the 16-bit floating-point model and b is the number of bits of given MPWN. Using Equation (3.2) instead of $\frac{b}{b_{max}}$, the optimization direction reverses from minimization to maximization which is the same direction as a and s.

$$ASB = \frac{a + s + b_{nor}}{3} \tag{3.1}$$

$$b_{nor} = 1 - \frac{b}{b_{max}} \tag{3.2}$$

In Equation (3.1), a, s, and b_{nor} does not contribute equally due to different in distributions. One of the metrics in ASB may distribute with a low variance relative to other metrics. Therefore, the metric with low variance contributes to ASB less than others. One of the factors of this low variance issue is a narrow range between the minimum and maximum values of the metric. To solve this issue, I introduce a min-max

normalization to re-distribute each metric of ASB into the same range [0, 1]. The min-max normalization is defined in Equation (3.3) where x is an original value, x_{max} is a maximum value in the distribution, x_{min} is a minimum value in the distribution, and x_n is a normalized value.

$$x_n = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.3}$$

Finding maximum and minimum values in the worst-case is required to train all possible combinations of MPWN which is not feasible with a large model. Therefore, I introduce a method to estimate the maximum and minimum values of a, s, and b_{nor} instead. This method is called as estimating rules, which are formulated from properties from Table 3.1 and observations from grid-search with all possible combinations of MPWN with the LeNet-5 model. From this grid-search, the **F** model contains the highest amount of a and the lowest amount of b_{nor} and s. The **B** model contains the highest amount of b_{nor} and close to the lowest amount of a. The **T** model contains almost the highest amount of s. With these properties, I formulate the estimation rules are as follows:

- **F** model contains a maximum value of a while **B** model contains a minimum value of a.
- **B** model contains a maximum value of b_{nor} while **F** contains a minimum value of b_{nor}.
- **T** model contains a maximum value of s while **F** contains a minimum value of s.

By using these estimation rules, I can estimate the maximum and minimum values for a, s, and b_{nor} by only using information from **F**, **B**, and **T** models. Using the min-max normalization improves the variance issue to an extent by improving the range of the metric with low variance. To further improve on the variance situation, one of the methods to increase or decrease the variance of a distribution is to multiply with a constant. Therefore,the variance of each metric of ASB can be balanced via modifying Equation (3.1) to a weighted average as shown in Equation (3.4) using with weights: α, β, and γ. Equation (3.4) is equal to Equation (3.1) when $\alpha = 1, \beta = 1$, and $\gamma = 1$. α, β, and γ can be adjusted to indicate which degree the a, s, and b contribute to ASB.

$$ASB = \frac{\alpha a + \beta s + \gamma b_{nor}}{\alpha + \beta + \gamma} \tag{3.4}$$

The goal of the MPWN is to maximize the vector of quantization layers $\boldsymbol{l}$, as shown in Equation (3.5), where $\boldsymbol{l} \in l^n$, $l \in \{\boldsymbol{F}, \boldsymbol{B}, \boldsymbol{T}\}$, and n denotes the number of weight layers.

$$\boldsymbol{l}^* = \underset{\boldsymbol{l}}{\operatorname{argmax}}\, ASB \tag{3.5}$$

The computational complexity of identifying the global maximum of the MPWN is $O(3^n)$. Each iteration of the search is expensive in terms of training time. To avoid examining all possible combinations of layers, in a previous study [41], I proposed human-based knowledge rules or three heuristics rules to identify a reasonable optimized $\boldsymbol{l}$. The heuristic rules are as follows:

1. Layers that contain a large number of weight parameters should be **T**.
2. Layers that contain a small number of weight parameters should be **B**.
3. The first and last layers should be **F**.

In this case, I define a large number as a number that has more than one positive standard deviation from a mean, and I define a small number as a number that is less than or equal to one positive standard deviation from the mean. To provide an example, I apply these heuristic rules to LeNet-5 [4]. The result is the **FBTBF** model since the third layer of LeNet-5 is a fully-connected layer with the number of weights that exceeds one standard deviation from the mean. These heuristic rules originate from several observations. For the first rule, placing **T** into a layer with the largest number of weight parameters causes the high sparsity in the model. Therefore, by only fixing **T** to certain layers, I can optimize other layers with other types of weight spaces. The second rule states that the default weight layer should be **B** to optimize the number of bits. **B** is the most suitable in terms of hardware implementation. The third rule states that placing **F** into the first layer significantly improves the accuracy. The last layer affects the confidence of the prediction; therefore, I also select the last layer as **F**.

The three heuristic rules allow the MPWN to perform a single search to find a suitable combination for the ASB score; however, the heuristic rules do not guarantee a global maximum. Systematizing the search process using the ASB score allows Bayesian optimization (BO) to be used. In general, BO is conventionally applied to search the optimal hyperparameters. BO is summarized in [42]. In general, BO optimizes a given cost function with two objectives. The first objective is to explore the function and map the surrogate model from the obtained information. The second objective is to search for a local optimal location of the given function. BO is suitable for my layer search from the two following aspects:

- BO does not require gradient; therefore, I can optimize my MPWN model with the sparsity and number of bits, which are not differentiable metrics.
- BO has defined with the constraint that each iteration is expensive to evaluate, which meets the requirements of the problem [42].

The utilization of BO with MPWN can be summarized in Figure 3.1.

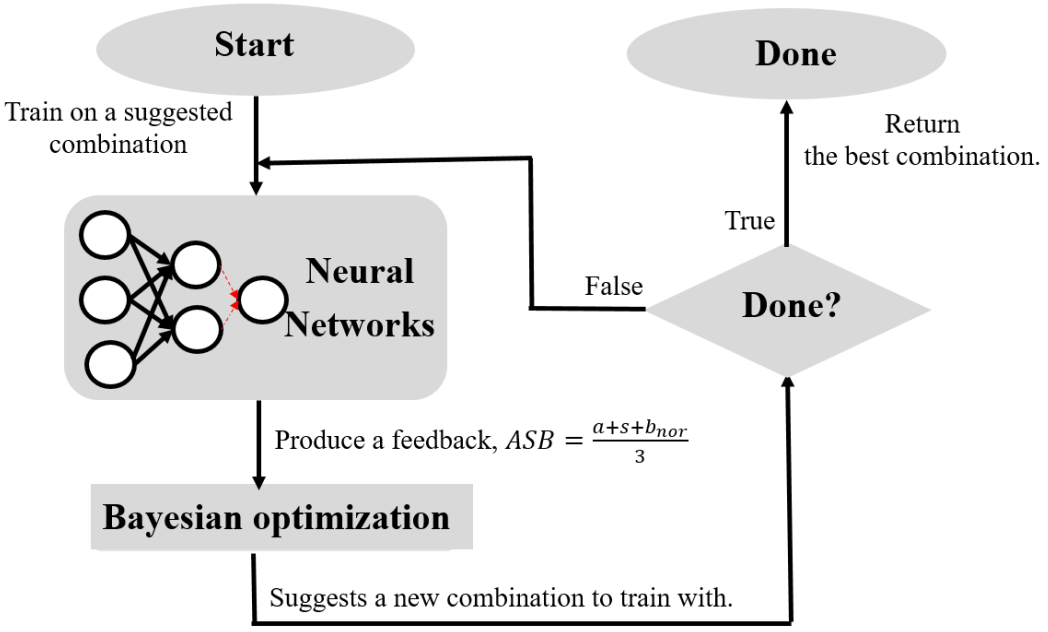

Fig 3.1. The overview of utilization of MPWN with BO. At first, BO generates a random MPWN combination. Then, the generated combination of MPWN is trained to acquire an ASB score. This ASB is used as feedback to BO to update its surrogate model. After the update, BO suggests a new MPWN combination to train. These processes can be repeated until satisfied.

3.1.1 1- and 2-bit Signed Integer

To fully utilize an FPGA with the MPWN, VHLS provides support to access arbitrary data types, such as signed integer (*int*), unsigned integer (*uint*), and the fixed-point (*fixed*) data type. To optimize the data type of **B** and **T**, I utilize 1-bit (*int1*) and 2-bits signed integer (*int2*), respectively. However, the range of *int1* can contain only in the set of $\{-1, 0\}$, which does not cover 1 in the **B** weight space. To address this problem, I replace 1 with 0. This does not affect the performance of the MPWN implementation, as the replacement still holds the same signed bit information used in XSB. Reducing the number of bits with a specific data type substantially reduces memory resources in my FPGA implementation.

3.1.2 Half-precision Floating-point

VHLS supports another data type: a half-precision floating-point or 16-bit floating-point (*half*). A QNN displays the robustness of a CNN against reducing precision, and I exploit this property by assigning the *half* data type to my MPWN model. Compared with 32-bit floating-point (*float*) and 64-bit floating-point (*double*) in terms of multiplication of two variables with of the same data type, *half* has the potential to reduce both the hardware resources and latency, as illustrated in Table 3.2. Table 3.2 presents the results of the implementation generated by VHLS, where the target device is Zynq UltraScale+ MPSoC ZCU102 or xczu9eg-ffvb1156-2-i. Table 3.2 demonstrates that *half* can reduce the resources to roughly half of those of *float* to one-fifth of those of *double*.

Table 3.2. Latency and hardware resources for multiplication of two variables with the same data type.

Data type	**Latency (Clock cycle)**	**Hardware Resource**		
		DSP48E	FF	LUT
double	4	11	304	236
float	1	3	130	150
half	**1**	**2**	**66**	**49**

3.1.3 XOR Signed-bits

The MPWN may consist of multiple **B**. Therefore, it is necessary to optimize the computation in the weight space further. The multiplication between *float* and binary values $\{-1, 1\}$ causes a sign to be changed. However, without a specific design, the multiplication between them is treated as a floating-point multiplication that consumes more resources than necessary. XOR signed bits (XSB) is designed as a replacement for multiplication between *float* and binary values. Multiplication between these variables is reduced with the XOR operation between the sign bits of the two operands, as illustrated in Figure 3.2.

In general, implementing XSB in HDL is simple, whereas implementing it in VHLS is complicate. I present an XSB algorithm in C++ for VHLS, as illustrated in Listing 3.1. VHLS provides C++ libraries that support bitwise manipulation; however, the manipulation is constructed as methods within a built-in data type in VHLS. The problem is that I cannot apply these methods directly with an unsupported data type (*double*, *float*, and *half*). Therefore, it is required to convert an unsupported data type to bitwise supported data type. Then, I perform bitwise manipulation of the selected bit and convert it back to the unsupported data type.

Listing 3.1. XOR signed-bits for Vivado high-level synthesis.

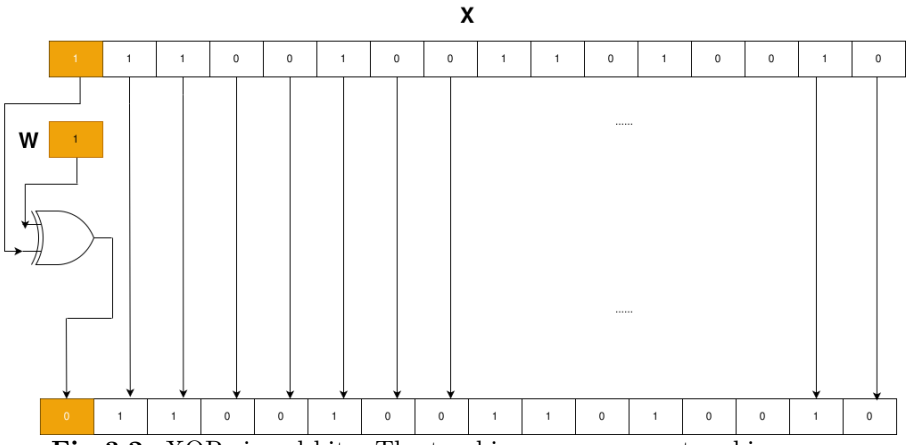

Fig 3.2. XOR signed bits. The top binary row presents a binary representation of the *half* data type, which represents a value of −123. The second binary row displays a binary representation of *int1*, which represents −1. By XOR only the most significant bit from both rows, the result is 123, which is the same as the answer to the general floating-point multiplication.

```
void xor_signed_bits(int1 w, half x, half &out)
{
    #pragma HLS INLINE OFF
    int16 tmp_x;
    int16 tmp_out;
    // Convert half floating point to int16-
    // to access ap_int built-in method.
    // Use uint1 because to cover {0, 1}.
    tmp_x = *reinterpret_cast<int16*>(&x);
    // XOR between signed-bit between weight and activation.
    uint1 sign = tmp_x.sign()^w.sign();
    // Get bits from 14 to 0, not include the signed bit.
    int15 notsign = tmp_x.range(14, 0);
    // Concate between XOR result and concatenate-
    // between the rest of activation bits.
    tmp_out = sign.concat(notsign);
    out = *reinterpret_cast<half*>(&tmp_out);
}
```

Here, *fixed* is considered to apply instead of *half*. However, VHLS provides the *fixed* implementation with a binary representation as 2's complement, which is not compatible with the XSB algorithm. Toggling the sign of *fixed* requires reversing all the bits of *fixed* and subtracting by 1, which is an expensive process. Therefore, I apply my MPWN with *half* instead of *fixed*.

3.1.4 Ternary Bitwise Operation

Ternary bitwise operation (TBO) was proposed by Honda et al. [21] as a replacement of multiplication with a ternary weight. TBO utilizes only an XOR and 15 AND gates as illustrated in Figure 3.3. TBO utilizes AND gates to detect whether the ternary weight

is a zero or one (both positive and negative). If the weight is zero, the AND gates reset the variable to zero. Otherwise, it lets the variable pass through. Using the same concept of XSB, I can implement TBO in VHLS as shown in Listing 3.2 below.

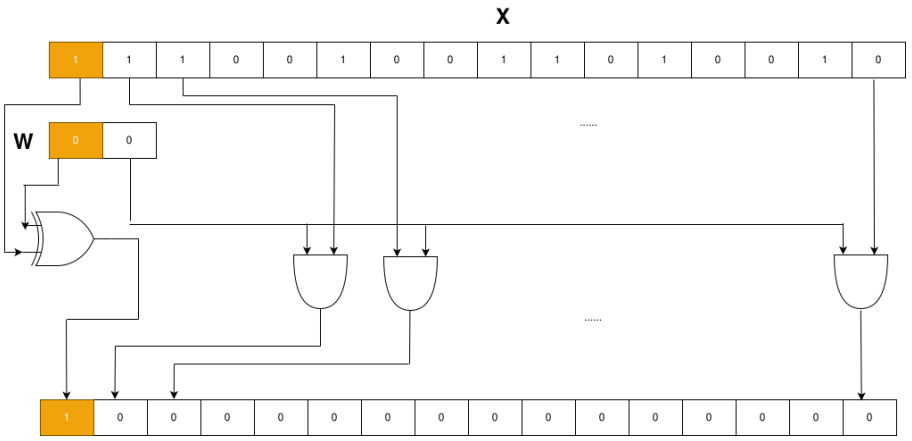

Fig 3.3. Ternary bitwise operation. The top binary row presents a binary representation of the *half* data type, which represents a value of −123. The second binary row displays a binary representation of *int2*, which represents 0. By using XOR and AND gates, the result is −0.

Listing 3.2. Ternary bit-wise operation for Vivado high-level synthesis.

```
void ternary_bitwise(int2 w, half x, half &out)
{
    #pragma HLS INLINE OFF
    int16 tmp_x;
    int16 tmp_o;
    tmp_x = *reinterpret_cast<int16*>(&x);
    // XOR between signed-bit between weight and activation.
    uint1 b15 = tmp_x.sign()^w.sign();
    uint1 w0 = w.range(0, 0);
    // AND between OR results and rest of activation bit.
    uint1 b0 = w0 && tmp_x.range(0, 0);
    uint1 b1 = w0 && tmp_x.range(1, 1);
    uint1 b2 = w0 && tmp_x.range(2, 2);
    uint1 b3 = w0 && tmp_x.range(3, 3);
    uint1 b4 = w0 && tmp_x.range(4, 4);
    uint1 b5 = w0 && tmp_x.range(5, 5);
    uint1 b6 = w0 && tmp_x.range(6, 6);
    uint1 b7 = w0 && tmp_x.range(7, 7);
    uint1 b8 = w0 && tmp_x.range(8, 8);
    uint1 b9 = w0 && tmp_x.range(9, 9);
    uint1 b10 = w0 && tmp_x.range(10, 10);
    uint1 b11 = w0 && tmp_x.range(11, 11);
    uint1 b12 = w0 && tmp_x.range(12, 12);
    uint1 b13 = w0 && tmp_x.range(13, 13);
    uint1 b14 = w0 && tmp_x.range(14, 14);
    // Concatenate between all resultant bit.
    uint15 b_con = (
        b14, b13, b12, b11, b10, b9, b8,
        b7, b6, b5, b4, b3, b2, b1, b0);
    tmp_o = b15.concat(b_con);
    out = *reinterpret_cast<half*>(&tmp_o);
```

```
}
```

3.1.5 Overview of FPGA Implementation

Using the ASB with BO allows us to find more optimized MPWN combinations than a combination from heuristic rules. Therefore, I implemented the model **FTTTF** that achieves the highest ASB. My implementation of **T** was implemented with TBO instead of sparse matrix multiplication and convolutional operation due to the high overhead of sparse matrix format. For instance, to decompose a weight of a fully-connected layer, W_f with a coordinate format (COO), if W_f does not contain any sparsity, COO decomposes W_f with three times the number of parameters comparing to W_f. In the case of the convolutional layer with the weight, W_c, this overhead becomes worse. The number of parameters becomes five times comparing to W_c. With **FTTTF** model, the sparsity in each **T** layer is roughly 0.5. Using COO format consumes 1.5 times the amount of origin parameters in the case of the fully-connected layer. This becomes worse in the case of the convolutional layer that consumes 2.5 times the original amount of parameters.

My implementation of the MPWN is designed layer by layer. I also optimized the layer by placing directives into it. Figure 3.4 shows an overview of the MPWN design on an FPGA. Since quantization is applied to convolutional and fully-connected layers only, I only evaluate the performance and apply directives in these layers. In Figure 3.4, I defined my notation as follows: $Conv\#N$ indicates a convolutional layer, $Fully\ connected\#N$ indicates a fully-connected layer, $BN\#N$ indicates a batch normalization layer [35], $Flatten$ indicates a rearranging layer to convert the shape activation to operate in the fully-connected layer, and N indicates the order of the weight layer.

To explain how I place the directives, I must first define a convolutional and fully-connected layer. The convolutional layer is defined in Equation (2.2) The convolutional layer operation consists of six for loops that can be accelerated with parallelism. It should be noted that the bias term is ignored because the convolutional layer is followed by batch normalization [35], which consists of a term that acts as a bias.

In the convolutional layer function in VHLS, I utilize parallelism by placing the $UNROLL$ directive inside the O_c in Equation (2.2), loop which hints that all loops below should be computed in parallel. In addition, I also place the $PIPELINE$ directive inside the O_r loop to further accelerate the operation that unrolled. Finally, in the convolutional layer, I apply $ARRAY\_PARTITION$ with a factor of 8 to both W and X.

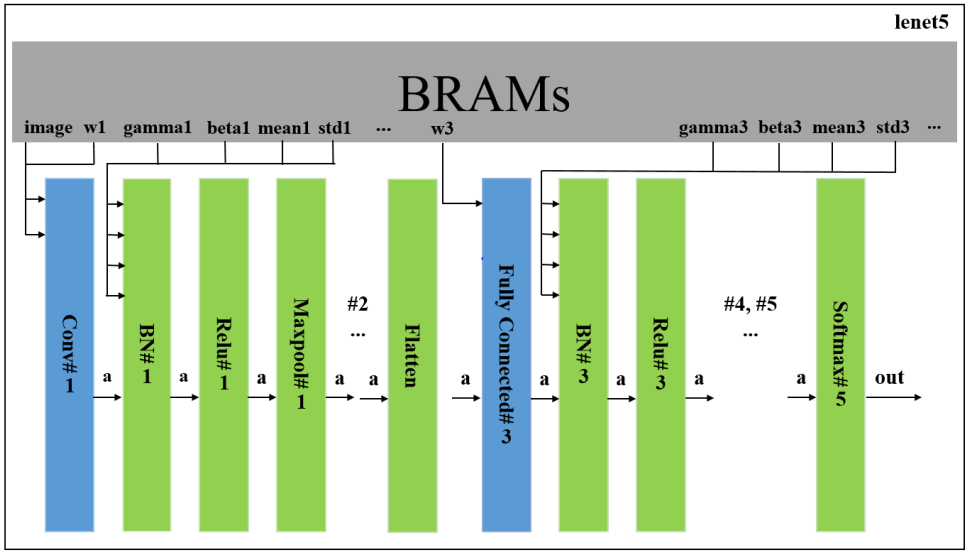

Fig 3.4. Overview of mixed-precision weights network implementation. All parameters of the MPWN are stored in BRAMs. Blue blocks indicate blocks that are optimized with directives, while green blocks indicate blocks that are not optimized with directives.

In the matrix multiplication function in VHLS, I place $PIPELINE$ inside C in Equation (2.1), for data pipelining of the process. I also apply $ARRAY\_PARTITION$ with a factor of 16 to both W and X. It should be noted that the first two fully-connected layers do not include the bias term b.

Chapter 4

Related Works

In this section, I will discuss related works and literature reviews of mixed-precision models and FPGA implementations of QNN.

4.1 BinaryConnect

BinaryConnect (BC) [13] is a QNN that binarizes its weights to the set $\{-1, 1\}$. With 2 possibilities, a binarized weight can be represented with a 1-bit. The BC quantization equation is expressed as Equation (4.1), where i is the index of the weight layer, W is the floating-point weight, and W^b is the binarized weight:

$$W_i^b = \begin{cases} 1, & W_i \geq 0, \\ -1, & W_i < 0 \end{cases} \tag{4.1}$$

However, Equation (4.1) cannot be used for back-propagation. Equation (4.1) causes the gradient become zeros everywhere. To modify this function to be trainable, BC overwrites the back propagation of Equation (4.1) to Equation (4.2), where L is the loss function. Equation (4.2) allows gradients to be able to pass through Equation (4.1) in the same manner as an identity function:

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial W_i^b} \tag{4.2}$$

In BC, there is an additional modification known as weight clipping to the updating equation as illustrated in Equation (4.3), where η is the learning rate. Equation (4.3) clips the updated weights into the range of [-1, 1]. This pushes the weights back into the active region [13].

$$W_i = min(max(W_i - \eta \frac{\partial L}{\partial W_i}, -1), 1) \tag{4.3}$$

4.2 Binarized Neural Networks

To further improve hardware friendliness to BC, Binarized neural network [14] (BNN) was proposed. Both weights and activation were quantized using Equation 4.1 instead of only weights comparing with BC. With both weights and activation of BNN are $\{-1, 1\}$, the multiplication between them can be replaced with XNOR logic operation, which is represented in Table 4.1.

Table 4.1. Truth table of XNOR gate operation, where inputs are A and B and the output is Y.

A	B	$Y = A \oplus B$
0	0	1
0	1	0
1	0	0
1	1	1

After XNOR operations, the results can be accumulated using the *bincount* operation which counts a number of differences between 1 and -1. This also further reduces the amount of hardware utilization. This quantization of activation cannot be directly done. BNN modifies the back propagation function of the activation quantization with the straight-through estimator [43].

4.3 Ternary Weight Networks

Ternary weight network (TWN) [15] is a QNN that quantizes its weights in each layer to the set $\{-S_i, 0, S_i\}$, where $S_i \in \mathbb{R}$, W^t represents the ternarized weights, and i is the order of layer. S_i can be determined using Equation (4.4). The TWN quantization equation is presented in Equation (4.5), where Δ is a threshold that can be determined by using Equation (4.6).

$$S_i = E_{i \in \{i||W_i|>\Delta\}}(|W_i|) \tag{4.4}$$

$$W_i^t = \begin{cases} S_i, & W_i > \Delta_i, \\ 0, & W_i \leq \Delta_i, \\ -S_i, & W_i < -\Delta_i \end{cases} \tag{4.5}$$

$$\Delta_i = 0.7 \times E(|W_i|) \tag{4.6}$$

In Equation (4.6), a constant 0.7 is realized by assuming the weight W_i are generated by a normal distribution with zero mean and σ^2 variance. By using a discrete optimization, the optimal threshold Δ_i can be found in this case the optimal threshold is 0.6σ that is equal to $0.75 \times E(|W_i|)$. To compute easily, the authors simplify down to $0.7 \times E(|W_i|)$ instead.

Compared to BC, with an additional zero in the weight space, the TWN has higher performance. However, by excluding scaling factors, this doubles the number of bits to represent its weight. With an additional zero, TWN introduces the concept of sparsity. Sparsity is defined as the number of zeros in a given array divided by the number of parameters in the array. Back-propagation of the TWN faces the same problem as back-propagation of BC. The TWN solves this problem with the same method as BC, which is using the redefined back-propagation of the quantization equation, as expressed in Equation (4.7):

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial W_i^t} \tag{4.7}$$

4.4 Mixed-precision Model

In prior works, there are at least two cases to deploy a mixed precision model. The first case is due to a limitation of low-precision models; for instance, Nakahara et al. [44] proposed a binarized YOLOv2 [45] designed for FPGA implementation. However, a binarized neural network (BNN) [14], or a QNN with both binary weights and activations

does not perform a bounding box prediction or regression task effectively. Nakahara et al. assigned the last layer of binarized YOLOv2 as the floating-point instead to address this problem. Another case is to improve the performance of the low-precision model as close as a 32-bit floating-point model. For instance, Chu et al. [46] proposed a quantization method to progressively reduce the bit-width from the input to the last layer. This method is realized from an observation that the feature distributions in the shallow layer contain a low quantity of class separability while in the deeper layers, the distributions have a high quantity of class separability. Wang et al. [47] proposed a method using reinforcement learning to search suitable bit-widths in the layer-wise direction. Using a hardware simulator, the energy and latency of the quantized model were utilized as direct feedback to the reinforcement learning controller.

4.5 FPGA Implementation of Quantization models

In terms of prior works in the FPGA, several works focus on QNNs with both weights and activations as either binary, fixed-point, or floating-point. For instance, FINN [48] and GUI-based binarized Neural NEtwork SyntheSizer toward FPGA (GUINNESS) [49] are frameworks to construct BNN to the FPGA. Both FINN and GUINNESS utilizes HLS as a backend component to deploy BNN models into FPGA. GUINNESS GUI can be shown in Figure 4.1. Rongshi et al. [50] and Cho et al. [51] also utilized HLS to construct a floating-point and fixed-point CNN, respectively.

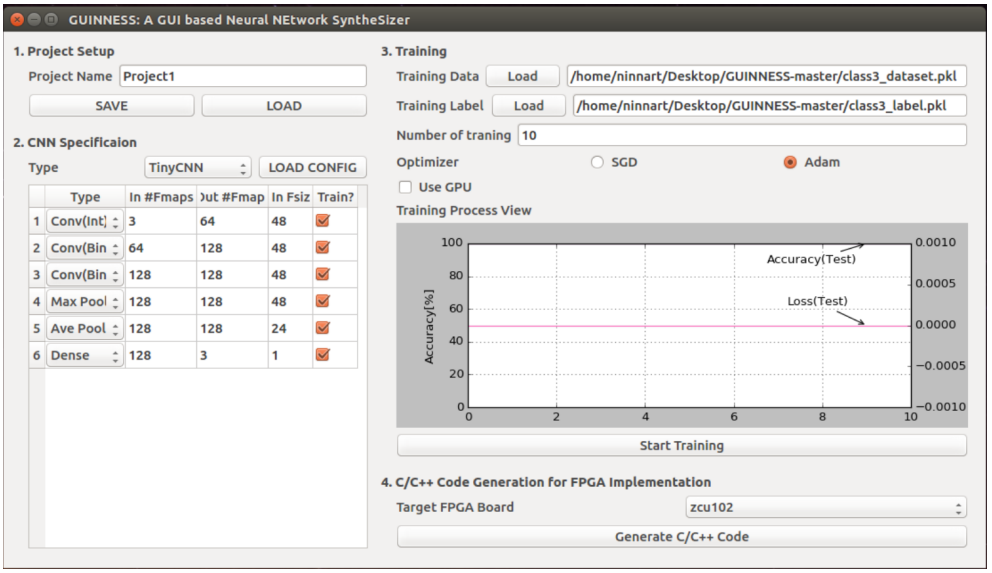

Fig 4.1. GUINNESS Graphical user interface. The specifications of BNN model can be selected for an extended. GUINNESS can train BNN with the selected specification using Chainer backend. After training, the user can utilize these weights to deploy with HLS.

4.6 Novelties

This study aims to achieve the performance of a 32-bit floating-point model while maintaining the properties of QNNs. To the best of my knowledge, comparing to previous researches in the mixed-precision network field, my novelty is I utilized a BO to search a suitable quantization layer instead of using the reinforcement learning, or differentiable architecture search [52]. I provide the ASB score that I specifically designed for the weight spaces. I included a sparsity as a part of ASB, and I also left a choice to not quantization into the search space.

Compared with prior works in the FPGA field, to the best of our knowledge, my novelty is to provide a first FPGA implementation of binary or ternary weights model with floating-point activation. Furthermore, to effectively deploy binary or ternary weights and floating-point activations, I also introduce XOR-signed bit (XSB) and ternary-bitwise operation (TBO) to replace floating-point multiplications with bitwise operations.

Chapter 5

Experimental Results and Discussion

5.1 Software Simulation

In this section, I separated into three sections: Fashion-MNIST, CIFAR10, and ILSVRC 2012 datasets. In the Fashion-MNIST section, I further evaluated my heuristic rules by running a grid search covering all possible combinations of the MPWN with the LeNet-5 model. Then, I applied BO to search for optimal MPWN combinations in terms of the ASB score. I evaluated a number of search iterations necessary to obtain a model with better ASB than heuristic rules. I also experimented to examine the effect of converting *half* to *float*. In the CIFAR10 section, I evaluated the robustness of my proposed methods by performing BO searches with ResNet-18 [36] model and CIFAR10 dataset. Finally, in ILSVRC 2012, I utilized the best MPWN combination from the CIFAR10 section and evaluated the generalization to ILSVRC 2012.

5.1.1 Fashion-MNIST

To evaluate the MPWN model, I used the Fashion-MNIST dataset [22] as a benchmark image dataset. Fashion-MNIST is a clothing image dataset that consists of 60,000 training images and 10,000 test images. Each image is a grayscale image consisting of 28x28 pixels. I preprocessed each pixel value to the range [0, 1] by dividing each pixel by the maximum pixel intensity, 255. In this part, my MPWN model was programmed using PyTorch [17], a deep learning framework. The base structure of the CNN that I applied to the MPWN was LeNet-5 [4] with the structure: $6C5 - MP2 - 16C5 - MP2 - 120FC - 84FC - 10Softmax$, where $C5$ is a 5×5 convolutional layer, $MP2$ is a 2×2

max-pooling layer, FC is a fully-connected layer and $Softmax$ is an output layer. I used the rectified linear unit (ReLU) as the activation function and applied dropout [34] in the fully-connected layers with $p = 0.5$ except in the last layer. I utilized batch normalization [35] to stabilize my training process, and my model was optimized with Adam [31] with initial learning of 10^{-3}. I trained all models for 200 epochs and stepped down the learning rate to one-tenth every 75 steps. I set the training batch size to 128 and I utilized ASB with $\alpha = 1, \beta = 1$, and $\gamma = 1$ in this experiment.

To visualize the heuristic rules, I performed a grid-search across all possible combinations of the MPWN; in other words, I trained $3^5 = 243$ combinations of the model. I summarized all metrics from these combinations into three box plots. In Figs 5.1 and 5.2. present box-plots that display the test accuracy on the y-axis and the type of weight layer on the x-axis. By running all possible combinations, **F** in the first and last layer correlates with the test accuracy compared with other weight layers. However, **F** in the last layer has an excessively high variance compared with **F** in the first layer. This reveals that I can update the last heuristic rules by changing the last layer of the CNN from **F** to other types of layers. However, the first should remain **F**.

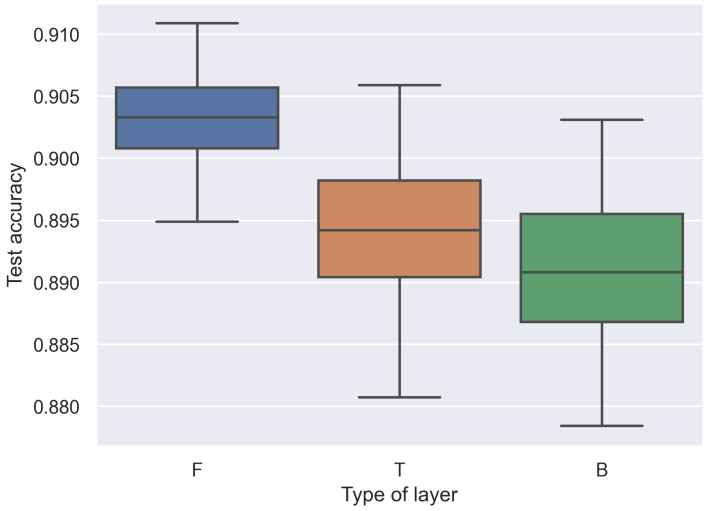

Fig 5.1. Box plot of test accuracy and effect of layer type in the first layer.

The third layer of LeNet-5 contains the highest number of parameters compared with other weight layers. From the first heuristic rule, setting the third layer as **T** affects the sparsity of the model, as illustrated in Figure 5.3. Note that the **T** layer in **FBTBF** contains sparsity within the layer as 0.4974. This amount of sparsity can be counted as 0.3495 sparsity of the model. This first heuristic rule contains another advantage. Placing **T** only in layers that contain a large number amount of parameters (fully-connected layer) eases the hardware implementation relative to the convolutional layer, which contains a large number amount of for loops.

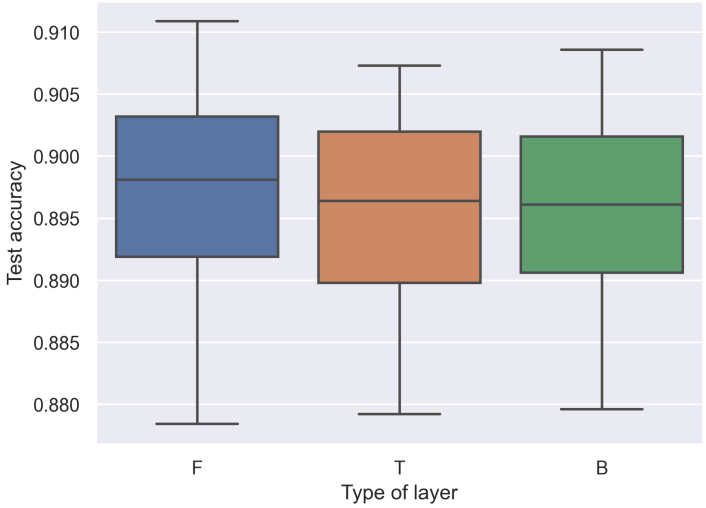

Fig 5.2. Box plot of test accuracy and effect of layer type in the last or fifth layer.

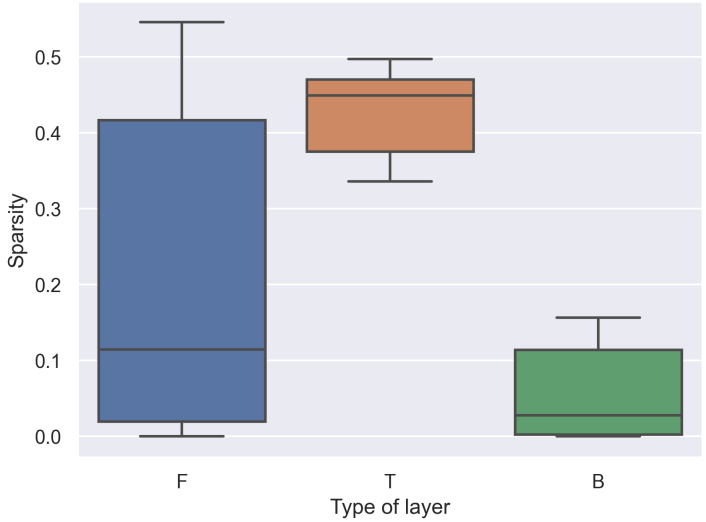

Fig 5.3. Box plot of sparsity and effect of layer type in the third layer.

The results of MPWN, TWN, BC, and BNN are presented in Table 5.1. In Table 5.1, I defined the **ASB before** as ASB without the min-max normalization and I also defined **ASB** as ASB with the min-max normalization. I report metrics in the training epoch that achieved the highest test accuracy. It should be noted that to scale with other methods; I did not apply the scaling factor S_i in Equation (4.4) to the TWN. I also did not clip weights in **B** layer with Equation (4.3). With heuristic rules, the **FBTBF** model is the optimized model. As displayed in Table 5.1, **FBTBF** obtains the advantages (i.e., accuracy, sparsity, and the number of bits) from the 16-bit floating-point, TWN, and BC models in a single model. However, by running all possible combinations of the MPWN, I found that the best ASB model without the min-max normalization was **FTTTT** with ASB as 0.7558. Comparing MPWN, TWN, and BC with BNN, BNN with binary activations promises a better hardware friendliness where its feature maps accumulation

can be replaced with popcount operation. However, its test accuracy is significantly dropped comparing with other methods.

Table 5.1. Comparison between different combinations the mixed-precision weights network. **ASB before** denotes ASB without the min-max normalization and **ASB** denotes ASB with the min-max normalization.

Type of layers	Test accuracy	Sparsity	Amount of bit	ASB before	ASB
Full Precision	**0.9109**	0.0	707,040	0.3036	0.3333
TWN [15]	0.8928	0.4852	88,380	0.751	0.7551
BC [13]	0.8798	0.0	**44,190**	0.6057	0.3477
BNN [14]	0.8475	0.0	**44,190**	0.595	0.0164
FBTBF	0.8984	0.3495	89,760	0.7069	0.7289
FTTTT	0.9036	**0.4919**	90,480	**0.7558**	0.8689
FTTTF	0.9071	0.4911	102,240	0.7512	**0.8984**

I plotted a box-plot of distributions of each element in ASB score as illustrated in Figure 5.4. In Figure 5.4, there are differences in mean and variance between each metric. Therefore, each metric contributes differently to ASB. I applied the Pearson correlation to measure the contributions of a, s, and b to ASB. I found that the Pearson correlation between a, s, and b to ASB are -0.2249, 0.5575, and 0.8493, respectively. The main issue in this ASB is the correlation between a and ASB is negative. I expected this issue is caused from the variance of a is insignificant or 5.36×10^{-5} comparing with s and b_{nor} that is 0.09865 and 0.03881, respectively.

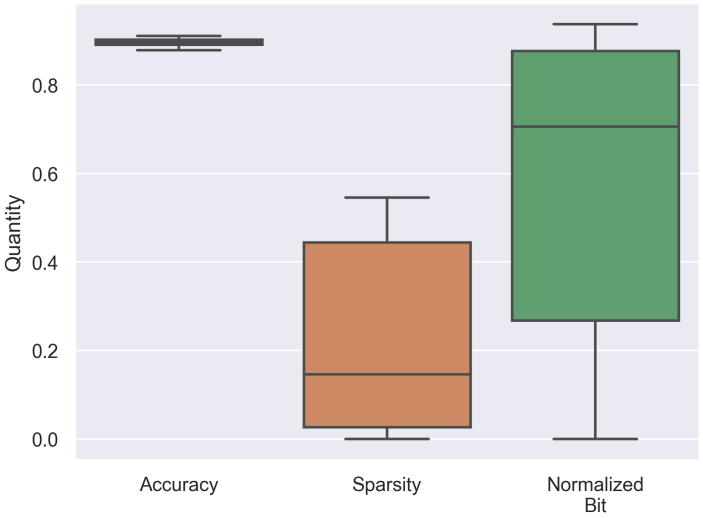

Fig 5.4. Distributions of accuracy, sparsity, and normalized bit from all possible combinations of MPWN with LeNet-5.

One of main contribution of the small variance of a is that it contains a narrow range of distribution from the minimum value at 0.8784 and maximum value at 0.9109. To improve the variance issue of a, I can rescale all element in ASB into the same scale by using a min-max normalization. By applying with the min-max normalization, normalized metrics are illustrated in Figure 5.5. The Pearson correlation between a, s, and b_{nor}

to ASB became 0.2816, 0.7743, and 0.5153, respectively and the variance of a, s, and b_{nor} becomes 0.05074, 0.1302, and 0.0112, respectively. The best combination in term of ASB is changed from **FTTTT** to **FTTTF**. I expected by rescaling the range of a, this signified the correlation between the **F** layer to ASB score, therefore the **FTTTF** becomes more important than **FTTTT**.

The min-max normalization requires minimum and maximum values of a, s, and b_{nor} to perform the normalization. However, to find the actual minimum and maximum values, in the worst case, this requires training all possible MPWN combinations, which is not feasible with the large model. Therefore, I estimated the minimum and maximum values by using the estimating rules that I mentioned in the mixed-precision weights network section instead. I compared the mean square error (MSE) between normalized values from the estimating rules to the normalized values from the actual maximum and minimum values to evaluate the estimating rules. I also compared normalized values from maximum and minimum values that were known from the random search. These comparisons are shown in Table 5.2. The box-plot of ASB after min-max normalization with actual and estimated values are illustrated in Figure 5.5.

Table 5.2. Comparison between the min-max normalization from the random search and the estimating rules. **Proposed** denotes the minimum and maximum values from the estimating rules. **GPU time** indicates the total training time with the same setting as the Fashion-MNIST section using NVIDIA GeForce GTX 1080 and Intel Xeon CPU E5-1620 v3.

Round of random search	**MSE of a**	**MSE of s**	**MSE of b**	**MSE of *ASB***	**GPU time (minute)**
10	0.0251	0.0119	1.377×10^{-3}	4.496×10^{-3}	208
30	2.71×10^{-3}	3.307×10^{-3}	0.2697×10^{-3}	0.6016×10^{-3}	603
50	2.01×10^{-3}	**0.0**	2.769×10^{-6}	$\mathbf{0.211 \times 10^{-3}}$	991
Proposed (3)	$\mathbf{0.524 \times 10^{-3}}$	4.827×10^{-3}	**0.0**	0.366×10^{-3}	**59**

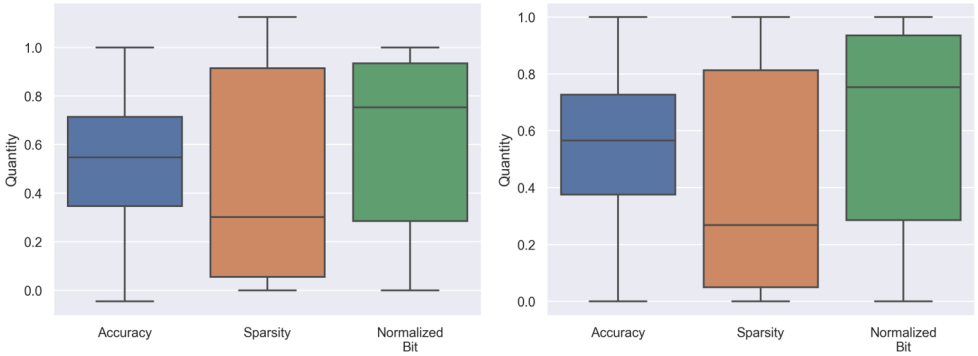

Fig 5.5. Box plots of each elements in ***ASB*** after the min-max normalization. Left: after the min-max normalization with estimated minimum and maximum values. Right: after the min-max normalization with actual minimum and maximum values.

From Table 5.2, by using the random search for 50 iterations or approximately one-fifth of all possible combinations, the random search achieved the lower MSE of ASB comparing with the estimating rules or **Proposed**. These results indicate that my estimating rules

did not perfectly estimate the minimum and maximum values. However, my estimating rules still provide a better alternative if I do not wish to train for 50 different models.

5.1.1.1 Bayesian Optimization

I conducted an experiment with BO and the ASB score with $\alpha = 1$, $\beta = 1$, and $\gamma = 1$. I evaluated the effectiveness of BO in finding optimal combinations of the MPWN. I applied BO from an implementation from `hyperopt` [53], and used BO to search for optimized MPWN models for 100 iterations. I searched with BO for 100 different combinations of the MPWN. In each iteration, BO received feedback or the ASB score and learned the subsequent combination of MPWN should be. Figure 5.6 presents the ASB score improvement of BO in each iteration. The ASB score on the y-axis changed only when BO found a new combination that produced a higher ASB score. I determined that BO was able to find the global maximum **FTTTF** after 79 iterations (i.e., approximately one-third of all possible search numbers). BO was able to find a better alternative to **FBTBF** or a model with an ASB score higher than 0.7289 after 5 iterations of searching. Therefore, heuristic rules still provide a better alternative if I do not wish to spend the time and resources to train 5 different models.

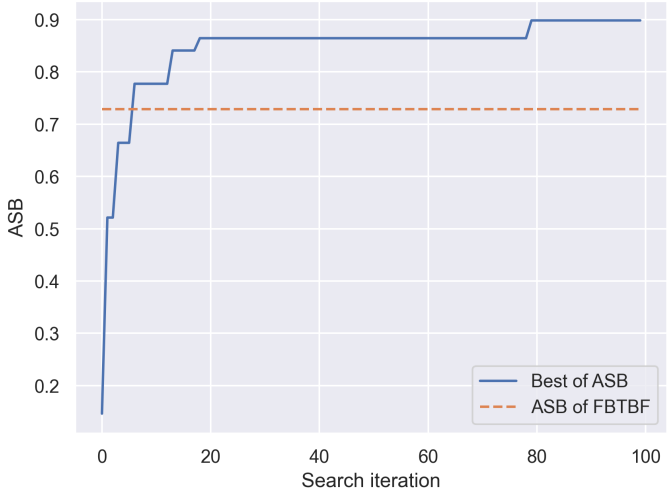

Fig 5.6. Bayesian optimization search with ASB score. This graph displays the best ASB in the current iteration search. The score changes when a higher score is found. The orange dashed line indicates the normalized ASB score of the heuristic rule (0.7289).

To visualize how this BO operates, I performed 100 iterations of BO with `optuna` [54]. By using BO with `optuna`, an `optuna` slice plot can be accessed and used to visualize as shown in Figure 5.7. This figure indicates that BO treads to discover high ASB scored combinations after BO had trained for certain iterations. In addition, this plot also indicates how ASB correlates with each weight space. For instance, in *w0* section, weight space f or **F** treads to out-performs all other combinations in term of ASB. This suggests

that **F** should be utilized in the first layer to maximize ASB scores. With a same concept, in term of *w1*, the suggested weight space can be either **F**, **B** and **T**. In *w4*, the weight space can be either **F** and **T**. In *w2* and *w3*, the suggested weight spaces are **T**.

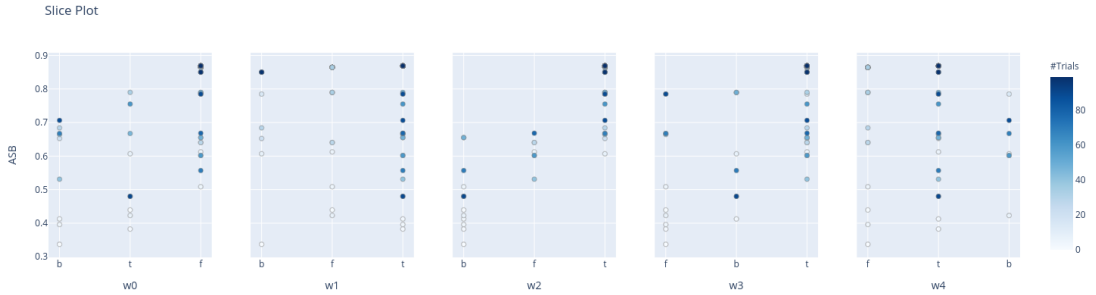

Fig 5.7. The slice plot from optuna shows the darker the color of data point the higher number of search BO performed. *w0* indicates a first layer of the model, while *w4* indicates the last layer of the model. On x-axis, *f*, *b*, and *t* are type of weight space which equivalent to **F**, **B**, and **T** respectively.

5.1.1.2 Effect of *float* and *half* on MPWN model

After training the **FTTTF** model for 200 epochs, I measured the effect of converting all parameters in the MPWN from *float* to *half*. I conducted this experiment in the PyTorch environment. The results are presented in Table 5.3. The performance of the **FTTTF** does not change. Therefore, in this case, I was able to convert *float* to *half* without performance loss.

Table 5.3. Comparison between *float* and *half* from **FTTTF** model.

	Test accuracy	**Difference with *float***
float	0.9053	0.0
half	0.9053	0.0

5.1.2 CIFAR10

In this section, I further evaluated my proposed methods by applying BO with ResNet-18 [36] and CIFAR10 [23] dataset. ResNet-18 is a CNN consisting of 18 weight layers with several residual connections that allow both feature maps and gradients to flow. CIFAR10 is an image dataset that consists of 50,000 training images and 10,000 test images. Each image is an RGB image consisting of 32x32 pixels. I preprocessed each image in a channel-wise direction with means and standard deviations of the training dataset. To modify ResNet-18 to operate with CIFAR10, I adjusted the first convolutional layer of ResNet-18 to $64C3$ with a stride of one and removed the max-pooling layer. To avoid overfitting, I applied data augmentations by random padding border pixels of an image

with four pixels and random crop the image back to the original 32x32 pixels. The image was further augmented by random horizontal flipping. I trained the ResNet-18 with 256 batch size for 150 epochs with Adam [31]. I set an initial learning rate with 10^{-3} and step down to one-tenth every 50 epochs. I quantized all weight layers within ResNet-18, including weights from residual connections. For the notation, I denote weights from residual connections after the underscore. For instance, **FFFFF_FFF** indicates a neural network with five **F** weight layers and three residual connections with **F** weights.

I performed 70 rounds of BO search with ASB ($\alpha = 1$, $\beta = 1$, and $\gamma = 1$). Each metric of ASB was normalized with the estimating rules. Since there are 3^{18} possible combinations of MPWN, outcomes of this BO search are not expected to contain a global maximum of ASB. I displayed the top-5 ASB combinations from BO search in **Proposed** section of Table 5.4. I also included a **Baseline** section that consists **F**, **B**, and **T** models. Note that in **Baseline**, I also calculated ASB with the estimating rules.

Table 5.4. Top-5 combinations from each BO search. **Iteration** denotes the number of BO searches. Note that the a, s, and b are not normalized with the min-max normalization and **Iteration** starts with 0.

	Combinations	***ASB***	***a***	***s***	***b***	**Iteration**
Baseline	**FFFFFFFFFFFFFFFFF_FFF**	0.3333	**0.9366**	3.584×10^{-6}	178,629,632	-
	TTTTTTTTTTTTTTTTTT_TTT	**0.7852**	0.9221	**0.93557**	22,328,704	-
	BBBBBBBBBBBBBBBBB_BBB	0.3333	0.9209	6.27×10^{-7}	**11,164,352**	-
Proposed	**FFFTTTFTBBBFTTTTTF_TBT**	**0.8156**	0.9276	0.8086	32,713,728	**64**
	FFTBTTFTFBFFTTBTTF_TBT	0.8092	**0.935**	0.5996	40,860,672	37
	BFFTFTFBFBTFTTTTTF_TBT	0.7915	0.9248	**0.8486**	35,858,112	4
	FFFTTTFFBBTFTTTTTF_TBB	0.7717	0.9232	0.8413	35,236,864	22
	FFTTTTFBFBTFTTTBTF_TTB	0.7504	0.9286	0.5945	**32,394,240**	55

In Table 5.4, I discovered patterns of heuristic rules in the top-5 combinations. In ResNet-18, the number of weight parameters increases gradually from the first to the last convolutional layer. Therefore, the last three convolutional layers of ResNet-18 contribute 63.34 percent of all weight parameters. Using the first heuristic rule (layers that contain a large number of weight parameters should be **T**), these last three convolutional layers should be **T** to maximize the sparsity. This pattern of first and third heuristic rule (the first and last layers should be **F**) can be identified throughout **Proposed**. With patterns of heuristic rules appeared in top-5 combinations, this signified a correlation between the heuristic rules and optimal ASB combinations.

Using BO with ASB allows searching a better alternative to models from **Baseline**. For example, **FFFTTTFTBBBFTTTTTF_TBT** or the top-1 model in **Proposed** contains all desired properties of ASB or high test accuracy, sparsity while maintaining a low amount of bits in a single model.

5.1.3 ILSVRC 2012

I utilized the best combination or **FFFTTTFTBBBFTTTTTF_TBT** from CIFAR10 section and utilized with ILSVRC 2012 dataset. In contrast to CIFAR10 section, I used with original ResNet-18 with this dataset. I resized all images to 256x256. During training, I cropped the images to 224x224, randomly horizontal flipped images, and channel-wise normalization with a mean and standard deviation from the training dataset. During validation, I center-cropped images to 256x256 and resized the cropped images to 224x224. I trained the ResNet-18 with 256 batch size for 100 epochs with Adam [31]. I set an initial learning rate with 10^{-3} and step down to one-tenth every 30 epochs. I did not use weight decays with QNNs and MPWN while **F** model was utilized with a weight decay of 1×10^{-4}. The results are shown in Table 5.5. In this experiment, I set a as the top-1 accuracy.

Table 5.5. ILSVRC 2012 results with the best ASB combination from CIFAR10 section. Note that the a, s, and b are not normalized with the min-max normalization and **Iteration** starts with 0.

	Combinations	ASB	a	s	b
Baseline	**FFFFFFFFFFFFFFFFF_FFF**	0.3333	0.621	1.773×10^{-5}	178,752,512
	TTTTTTTTTTTTTTTTT_TTT	0.8944	0.6058	**0.4708**	22,344,064
	BBBBBBBBBBBBBBBBB_BBB	0.3333	0.5602	0.0	**11,172,032**
Proposed	**FFFTTTFTBBBFTTTTTF_TBT**	**0.9631**	**0.6335**	0.3828	32,836,608

In term of accuracy and **ASB**, our combination **FFFTTTFTBBBFTTTTTF_TBT** provides the best performance comparing with other baseline models. However, due to the long training time of this dataset, I cannot perform the hyper-parameter searches carefully for each model. These results in each combination may not be fully optimized. I think that these results can be improved with more hyper-parameter searches.

5.2 FPGA Synthesis and Implementation

In this section, I synthesized a **FTTTF** model with Vivado HLS version 2019.02 and implemented this model with Xilinx Vivado 2019.2 [55]. I evaluated the model in terms of latency and hardware area compared with a conventional 32-bit floating-point model. I also performed comparisons between models with directives and without directives.

All hardware synthesis results were obtained by using VHLS with the option of C synthesis. The target FPGA was Zynq UltraScale+ MPSoC ZCU102 (ZCU102) or xczu9eg-ffvb1156-2-i. ZCU102 consisted of the following hardware resources: 1824 BRAM_18K, 2520 DSP48E, 548,160 FF, and 274,080 LUT. Our FPGA operated with a target clock frequency of 100 MHz, and my implemented model used weights and biases from training in the software simulation part.

5.2.1 XOR signed-bit and ternary bitwise operation synthesis

I synthesized XSB and TBO as a replacement for multiplication between *half* and *int1* and *half* and *int2*, respectively. Therefore, I compared hardware resources and latency with multiplication between other data types, as illustrated in Table 5.6.

Table 5.6. Comparison of latency and hardware utilization of multiplication between two variables with data type 1 and 2, respectively. The latency unit is a clock cycle. The two last row represents XOR signed bit and ternary bitwise operation, respectively.

Data type 1	**Data type 2**	**Latency**	**Hardware Resource**		
			DSP48E	FF	LUT
double	*double*	4	11	304	236
float	*float*	1	3	130	150
half	*half*	1	2	66	49
half	***int1***	**0**	**0**	**0**	**2**
half	***int2***	**0**	**0**	**0**	**32**

Using XSB instead of floating-point arithmetic significantly reduced the latency and hardware resources. XSB consumed only two LUTs to perform multiplication, while TBO consumes 32 LUTs. Each pair of LUTs is used to construct a logic gate. Therefore, XSB utilizes a logic gate, while TBO utilizes 16 logic gates. Note that in both XSB and TBO, the latency cannot be zero in practice. I only displayed the results from VHLS C synthesis.

In terms of behavior, XSB should perform in the same manner as Listing 5.1. Listing 5.1 detects the most significant bit (MSE) of the binary weight and flips the MSE of *float*. However, detecting requires a control logic which is expensive VHLS. For example, by using Listing 5.1 to C synthesis with Vivado HLS 2019.2, it consumes 2 cycles at 100 MHz, 2 DSP48Es, 97 FFs, and 150 LUTs. Compared with XSB that consumes only 2 LUTs and 0 latency, XSB is more efficient in hardware utilization and latency.

Listing 5.1. XOR signed-bits using a control logic.

```
void xor_signed_bit_using_control_logic(int1 w, half x, half &o)
{
    #pragma HLS INLINE OFF
    o = w.sign() == 0 ? x : -x;
}
```

5.2.2 Hardware Synthesis

I synthesized the MPWN with **FTTTF** into an FPGA with VHLS. I performed several comparisons with the model with *float* and with and without directives. I used the

following notations. **Proposed** signifies that all *float* data type were replaced with *half*; in **T**, the multiplication was replaced with TBO. **Base-line** signifies that all data types were *float* and that all operations in the model were the floating-point arithmetic. **Directive** signifies that I applied optimization directives in VHLS to optimize the latency of the model with parallelism. However, this resulted in a trade-off of higher hardware utilization.

A comparison of the latency and resources using these methods is presented Tables 5.7 and 5.8, respectively. Table 5.7 includes comparisons with ARM Cortex-A53. I utilized ARM Cortex-A53 in Zynq UltraScale+ MPSoC ZCU102. To use this CPU, I generated a PetaLinux image [56] and executable files from C++ files using SDSoC 2018.3 [57]. The setting of these C++ files was the same as **Base-line**. I ran executable files three times and reported the latency mean and interval of two standard deviations.

I observed that the convolutional layers had a longer latency than the fully-connected layers even though there were fewer parameters. I hypothesized that this was due to the complexity of the convolutional layer, and the VHLS performed worse when dealing with a large number of for loops in the convolutional layer. With the directives, both **Proposed** and **Base-line** significantly improved the latency; however, they also significantly increased the hardware utilization. Compared with ARM Cortex-A53, I reduced the latency by 2.0 to 11.77 times depending on the type of layer.

Table 5.9 includes comparisons in term of latency of MPWNs, Rongshi et al. [50], GUINNESS [49], and Cho et al. [51]. Rongshi et al. proposed a 32-bit floating-point LeNet-5 on Xilinx Zybo Z7 board (zynq7020). Cho et al. proposed a fixed-point LeNet-5 model that targets xczu9eg-ffvb1156-2-i. Cho et al. utilized 20-bit fixed-point on the first layer and 16-bit fixed-point on the latter layers. GUINNESS is a graphical user interface for training BNN on a GPU and deploying BNN on an FPGA. I utilized the GUINNESS from [58] to construct a BNN with a default LeNet-5 configuration of GUINNESS. This BNN was set to target Zynq UltraScale+ MPSoC ZCU102 (xczu9eg-ffvb1156-2-i). All of the related works operate with the same 100 MHz frequency.

Note that there are several differences between my and related models. The first difference is Rongshi et al. , and Cho et al. applied the third layer as a convolutional layer instead of a fully-connected layer that I utilized. The second one is the default setting of GUINNESS for LeNet-5 is $64C3 - 64C3 - 64C3 - 32AP - 10FC$, where $32AP$ is a global average pooling that averages feature maps in width and height directions. The third one is Rongshi et al. , and Cho et al. did not apply batch normalization layers. The fourth difference is Cho et al. replaced max-pooling layers with average pooling layers and utilized a Tanh activation function instead of ReLU. The last one is GUINNESS, and Cho et al. expect grayscale 32x32 images as inputs instead of grayscale 28x28 images that

I used. Our **Proposed directive** performed 11.62, 10.03, and 4.556 times less latency comparing Rongshi et al. , GUINNESS, and Cho et al. , respectively.

Table 5.7. Comparison between different FPGA synthesis of LeNet-5 layer by layer in terms of latency (ms).

Layer	Base-line	Proposed	Base-line directive	Proposed directive	CPU ARM Cortex-A53	Comparing Proposed directive and CPU
Conv#1	7.364	4.738	0.027	**0.0264**	0.148 ± 0.009	5.61x
Conv#2	13.059	5.369	0.122	**0.119**	0.684 ± 0.009	5.75x
Fully connected#3	2.460	0.924	0.020	**0.0147**	0.173 ± 0.002	11.77x
Fully connected#4	0.808	0.304	0.009	**0.006**	0.066 ± 0.006	11.0x
Fully connected#5	0.067	0.042	0.004	**0.002**	0.004 ± 0.0	2.0x

Table 5.8. Comparison between different FPGA synthesis in terms of hardware utilization. The number inside parentheses indicates the percentage of hardware utilization of Zynq UltraScale+ MPSoC ZCU102. In the *Total* row, some layers may not be included, such as the flatten, max-pooling, and batch normalization layers.

Layer	Components	Base-line	Proposed	Base-line directive	Proposed directive
Conv#1	BRAM-18K	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	DSP48E	5 (0%)	**4 (0%)**	170 (7%)	**136 (5%)**
	FF	591 (0%)	**317 (0%)**	71,343 (13%)	**24,194 (4%)**
	LUT	814 (0%)	**589 (0%)**	45,469 (17%)	**24,858 (9%)**
Conv#2	BRAM-18K	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	DSP48E	5 (0%)	**2 (0%)**	70 (3%)	**28 (1%)**
	FF	598 (0%)	**254 (0%)**	120,868 (22%)	**47,089 (8%)**
	LUT	894 (0%)	**668 (0%)**	69,094 (25%)	**43,888 (16%)**
Fully connected#3	BRAM-18K	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	DSP48E	5 (0%)	**2 (0%)**	160 (6%)	**64 (2%)**
	FF	542 (0%)	**161 (0)**	44,342 (8%)	**24,294 (4%)**
	LUT	538 (0%)	**302 (0%)**	38,933 (14%)	**21,869 (8%)**
Fully connected#4	BRAM-18K	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	DSP48E	5 (0%)	**2 (0%)**	150 (6%)	**60 (2%)**
	FF	566 (0%)	**185 (0%)**	31,682 (6%)	**13,486 (2%)**
	LUT	544 (0%)	**302 (0%)**	27,134 (10%)	**14,874 (5%)**
Fully connected#5	BRAM-18K	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	DSP48E	5 (0%)	**4 (0%)**	140 (6%)	**112 (4%)**
	FF	526 (0%)	**227 (0%)**	26,240 (5%)	**12,891 (2%)**
	LUT	530 (0%)	**304 (0%)**	19,937 (7%)	**11,570 (4%)**
Total	BRAM-18K	40 (2%)	**25 (1%)**	51 (3%)	**29 (1%)**
	DSP48E	54 (2%)	**43 (1%)**	719 (29%)	**429 (17%)**
	FF	8,228 (2%)	**3,831 (0%)**	302,914 (55%)	**126,198 (23%)**
	LUT	14,360 (5%)	**9,232 (3%)**	214,386 (78%)	**126,787 (46%)**

Table 5.9. Comparison between baseline implementation of LeNet-5, my proposed method, and related works in term of latency.

	Latency (ms)	Comparison with Baseline
Baseline	24.799	1x
Proposed	11.995	2.07x
Baseline directive	1.222	20.29x
Proposed directive	**0.786**	**31.55x**
Rongshi et al. [50]	9.135	2.715x
Cho et al. [51]	3.581	6.925x
GUINNESS [49]	7.882	3.146x

5.2.3 Hardware Implementation

To implement the model, I exported my model in VHLS as intellectual property (IP). I used Vivado 2019.2 to implement the exported IP to ZCU102. The results of the implementation are provided in Tables 5.10 and 5.11. In Table 5.10, there is an additional hardware resource which is the look-up table random-access memory (LUTRAM). Comparing Tables 5.8 and 5.10, the synthesis and implementation displayed a different amount of hardware utilization. Note that in Table 5.10, Cho et al. did not provide the hardware utilization from the implementation. Therefore, I reported the Cho et al. synthesis results in Table 5.10. In terms of **Proposed** and **Proposed directive**, the implementation exhibited significantly reduced hardware utilization than the synthesis. Our **Proposed** reduced LUT 2.31 times, LUTRAM 11.25 times, FF 2.89 times, BRAM 1.6 times, and DSP 1.25 times compared to **Baseline**. **Proposed directive** further reduced LUT 2.59 times, LUTRAM 4.89 times, FF 2.92 times, BRAM 1.76 times, and DSP 1.68 times compared with **Baseline directive**. Comparing with Rongshi et al. , GUINNESS, and Cho et al. , my **Proposed directive** utilizes more hardware utilization except for BRAMs.

Both **Baseline directive** and **Proposed directive** utilizes higher overall hardware resources than other methods due to my FPGA implementation designed to minimize the latency by using a higher degree of parallelism comparing with other methods. Comparing between our **Proposed directive** with **GUINNESS** in term of hardware utilization indicates that our **Proposed directive** significantly consumes a lot of resource than **GUINNESS**. I hypothesized that one of the factors is this FPGA implementation still utilizes the floating-point accumulations.

Table 5.11 presents a comparison in terms of power utilization. **Proposed** reduced the power consumption of **Baseline** 1.18 times. However, **Proposed directive** further reduced the power consumption 2.05 times compared to **Base-line directive**. Although my **Proposed directive** consumed more hardware resources than Rongshi et al. , my power consumption of **Proposed directive** is less than Rongshi et al. 1.27 times. However, my **Proposed directive** still consumes 2.142 times more power consumption than GUINNESS.

Table 5.10. Comparison between FPGA implementations of LeNet-5 in the term of hardware utilization. In an improvement factor column displays pairwise comparisons between **Baseline** with **Proposed** and **Baseline directives** with **Proposed directives**. All improvement factors from related works are compared with **Baseline directives**.

	Components	Total amount	Improvement factor
Baseline	LUT	8,305	1.0x
	LUTRAM	180	1.0x
	FF	6,787	1.0x
	BRAM	20	1.0x
	DSP	55	1.0x
Proposed	LUT	**3,599**	**2.31x**
	LUTRAM	**16**	**11.25x**
	FF	**2,345**	**2.89x**
	BRAM	**12.5**	**1.6x**
	DSP	**44**	**1.25x**
Baseline directive	LUT	209,720	1.0x
	LUTRAM	49,477	1.0x
	FF	243,540	1.0x
	BRAM	25.5	1.0x
	DSP	719	1.0x
Proposed directive	LUT	81,050	2.59x
	LUTRAM	10,142	4.88x
	FF	83,266	2.92x
	BRAM	**14.50**	**1.76x**
	DSP	429	1.68x
Rongshi et al. [50]	LUT	14,659	14.31x
	FF	14,172	17.18x
	BRAM	119.5	0.2134x
	DSP	**125**	**5.752x**
Cho et al. [51]	LUT	32,589	6.435x
	FF	33,585	7.251x
	BRAM	95	0.2684x
	DSP	143	5.028x
GUINNESS [49]	LUT	**5,034**	**41.66x**
	LUTRAM	**278**	**178x**
	FF	**4,417**	**55.14x**
	BRAM	23.5	1.085x

Table 5.11. Comparison between implementations of LeNet-5 in terms of total on-chip power (W). The improvement factor column displays a pairwise comparison between **Baseline** with **Proposed** and **Baseline directives** with **Proposed directives**. All improvement factors from related works are compared with **Baseline directives**.

	Total on-chip power (W)	**Improvement factor**
Baseline	0.852	1.0x
Proposed	**0.72**	**1.18x**
Baseline directive	2.901	1.0x
Proposed directive	1.414	2.05x
Rongshi et al. [50]	1.8	1.612x
Cho et al. [51]	-	-
GUINNESS [49]	**0.66**	**4.395x**

Chapter 6

Conclusion

In this study, I introduced MPWN, a QNN that jointly utilizes three weight spaces: floating-point, binary, and ternary. I proposed a systematized search to find optimal MPWN combinations with BO and the ASB score. To ensure that each metric of ASB has a positive correlation with ASB, I introduced a min-max normalization to rescale each metric of ASB. To accelerate the min-max normalization process, I provided estimating rules to estimate the minimum and maximum values of each ASB metric using information from only three models. I further evaluated previously proposed heuristic rules and the trade-off between heuristic rules and BO search. Finally, my hardware implementation exploited the MPWN's weight space in the MPWN with TBO and a specific data type. These elements demonstrated that the MPWN could be implemented in an FPGA with significantly fewer hardware resources and lower on-chip power consumption and latency than a conventional 32-bit floating-point neural network.

This work provides a pipeline to deploy a mixed-precision model to an edge device in terms of application. The user can indicate which metrics in ASB to prioritize by adjusting α, β, and γ weights. By optimizing the user-defined ASB score with BO, the user may discover a suitable combination for a given task. This work also provides a replacement of multiplication between floating-point weights and ternary or binary weights. The user can utilize these TBO and BO to efficiently deliver low latency and area operations to the user model. Using this pipeline, the deployment of the mix-precision model should be simplified in both hardware and software directions.

In general, the performance and amount of bit of a model can be adjusted by another approach, the neural architecture search. This neural architecture search can adjust the type of layer, the number of weights, others. Our final goal in this research is to search for the best neural architecture and bit-width precision given the task and computing resource. Unifying both the neural architecture and mixed-precision search spaces allows

more possibilities. This may enable finding interesting bit-width combinations and novel models

6.1 Future Works

This work can be extended in both software and hardware directions. In the software section, floating-point accumulations in MPWN consume a significant proportion of overall hardware resources. To reduce this hardware requirements, one of possible solutions is an *int8* or 8-bit integer quantization [59]. The *int8* quantization provides an estimation of floating-point array using an *int8* array, floating-point scaling factor values and bias values. In general, converting 16-bit floating-point activation to *int8* activation should reduce overall hardware usages in accumulation operations.

To explore this possibility, I conducted an initial Python simulation experiment with *int8* quantization and MPWN. This *int8* was replicated from PyTorch implementation of *int8* quantization. Note that my replication still has some mismatches with PyTorch *int8* quantization. In this replication, I found that with **FTTTF** combination, the test accuracy is dropped by only 0.0002. I also experimented with Vitis HLS 2020.2 to check resource utilization and latency from *int8* addition. The result is shown as in Table 6.1. Comparing between *int8* addition and *float*, by replacing *int8* promises significant lower hardware utilization. With both experiments, I think this shows an interesting promise to *int8* quantization to MPWN; however more experiments are required to check to scalability and corrections.

Table 6.1. Comparison of latency and hardware utilization of addition between two variables with data type 1 and 2, respectively. The latency unit is a clock cycle.

Data type 1	**Data type 2**	**Latency**	**Hardware Resource**		
			DSP48E	FF	LUT
double	*double*	4	3	450	813
float	*float*	3	2	231	240
half	*half*	2	2	96	127
int8	***int8***	**1**	**0**	**0**	**15**

As mentioned in the conclusion section, I would like to add more search spaces from the neural architecture search to the MPWN search space. Currently, I did not know how to implement a model that can utilize both search spaces. Because the search space becomes significantly larger, for instance, the solutions for a low-bit-width model can be both a high-precision compact model or a low-precision large model.

In the hardware direction, my current FPGA implementation still does not support sparsity. To effectively utilize the sparsity, it requires a sparse matrix format as the

input to the layer. To support that, I implemented the COO (Coordinate list) format of the sparse matrix. COO provides a fixed number of variables across all arrays; therefore, COO is suitable for the hardware implementation comparing with other formats. However, without the time, my implementation of COO with a convolutional operation or matrix multiplication can be done for sequential processing only. Therefore, the parallel processing of COO should be considered in future works.

Publications

Journal

- Ninnart Fuengfusin, Hakaru Tamukoh, "Mixed-precision weights network for field-programmable gate array," *PLoS ONE*. 2021; 16(5):e0251329. https://doi.org/10.1371/journal.pone.0251329
- Ninnart Fuengfusin, Hakaru Tamukoh, "A Sub-Model Detachable Convolutional Neural Network," *Journal of Robotics, Networking and Artificial Life*, Accepted, 2021.
- Ninnart Fuengfusin, Hakaru Tamukoh, "Network with Sub-networks: Layer-wise Detachable Neural Network," *Journal of Robotics, Networking and Artificial Life*, Vol. 7, No. 4, pp. 240-244, March 2021.

Interational Conference Proceeding

- Ninnart Fuengfusin, Hakaru Tamukoh, "Convolutional Network with Sub-Networks," *The 2021 International Conference on Artificial Life and Robotics (ICAROB2021)*, OS19-1, Online, January 21- 24 (22), 2021.
- Ninnart Fuengfusin, Hakaru Tamukoh, "Multi-Sampling Classifiers for the Cooking Activity Recognition Challenge," *Activity and Behavior Computing (ABC 2020)*, ABC 264, Kitakyushu, Japan, August 26-29 (29), 2020.
- Ninnart Fuengfusin, Hakaru Tamukoh, "Network with Sub-Networks," *The 2020 International Conference on Artificial Life and Robotics (ICAROB2020)*, OS20-2, Oita, Japan, January 13-16 (14), 2020.
- Ninnart Fuengfusin, Hakaru Tamukoh "Mixed Precision Weight Networks: Training Neural Networks with Varied Precision Weights," *25th International Conference on Neural Information Processing (ICONIP2018)*, Siem Reap, Cambodia, December 13-16(15), 2018.

Bibliography

1. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

2. L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

3. M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10781–10790, 2020.

4. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

5. Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, p. 18, 2010.

6. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

7. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

8. M. Horowitz, "Energy table for 45nm process." Stanford VLSI wiki.

9. S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

10. G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

11. F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and less than 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

12. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

13. M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, pp. 3123–3131, 2015.

14. M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

15. F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

16. S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, pp. 1–6, 2015.

17. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

18. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

19. M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pp. 1–8, IEEE, 2019.

20. Xilinx, Vivado-HLS, "Vivado design suite user guide-high-level synthesis," 2019.

21. K. Honda and H. Tamukoh, "A hardware-oriented echo state network and its fpga implementation," *Journal of Robotics, Networking and Artificial Life*, vol. 7, pp. 58–62, 2020.

22. H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

23. A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www.cs.toronto.edu/kriz/cifar.html*, 2014.

24. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, IEEE, 2009.

25. F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

26. I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016.

27. K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, pp. 267–285, Springer, 1982.

28. A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

29. H. Robbins and S. Monro, "A stochastic approximation method," in *Herbert Robbins Selected Papers*, pp. 102–109, Springer, 1985.

30. N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

31. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

32. T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

33. Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

34. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

35. S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

36. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

37. H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

38. A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 and cifar-100 datasets," *URl: https://www.cs.toronto.edu/kriz/cifar.html*, vol. 6, 2009.

39. L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

40. J. R. G. Ordaz and D. Koch, "On the hls design of bit-level operations and custom data types," in *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pp. 1–8, VDE, 2017.

41. N. Fuengfusin and H. Tamukoh, "Mixed precision weight networks: Training neural networks with varied precision weights," in *International Conference on Neural Information Processing*, pp. 614–623, Springer, 2018.

42. A. Agnihotri and N. Batra, "Exploring bayesian optimization," *Distill*, 2020. https://doi.org/10.2991/jrnal.k.200512.012.

43. G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning," *Coursera, video lectures*, vol. 264, no. 1, 2012.

44. H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 31–40, ACM, 2018.

45. J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

46. T. Chu, Q. Luo, J. Yang, and X. Huang, "Mixed-precision quantized neural networks with progressively decreasing bitwidth," *Pattern Recognition*, vol. 111, p. 107647, 2021.

47. K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8612–8620, 2019.

48. Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

49. H. Nakahara, H. Yonekawa, T. Fujii, M. Shimoda, and S. Sato, "Guinness: A gui based binarized deep neural network framework for software programmers," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 5, pp. 1003–1011, 2019.

50. D. Rongshi and T. Yongming, "Accelerator implementation of lenet-5 convolution neural network based on fpga with hls," in *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, pp. 64–67, IEEE, 2019.

51. M. Cho and Y. Kim, "Implementation of data-optimized fpga-based accelerator for convolutional neural network," in *2020 International Conference on Electronics, Information, and Communication (ICEIC)*, pp. 1–2, IEEE, 2020.

52. H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

53. J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*, pp. 115–123, 2013.

54. T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

55. Xilinx, Vivado, "Vivado design suite user guide implementation," 2019.

56. Xilinx, PetaLinux, "Petalinux tools documentation," 2018.

57. Xilinx, SDSoC, "Sdsoc environment user guide," 2019.

58. H. Nakahara *et al.*, "Guinness: A gui based binarized neural network synthesizer toward an fpga." https://github.com/HirokiNakahara/GUINNESS, 2017. Accessed: 2021-04-02.

59. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "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, 2018.