IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022.Doi Number

Post-Training Quantization of CNN Accelerator with Mixed Precision Floating-Point Arithmetic Selection Based on a Genetic Algorithm

Muhammad Junaid[1], Hayotjon Aliev[1], Shoaib Sajid[1], and HyungWon Kim[1], (Member, IEEE)

[1]Department of Electronics, College of Electrical and Computer Engineering, Chungbuk National University, Cheongju 28644, Republic of Korea

Corresponding author: HyungWon Kim (e-mail: hwkim@chungbuk.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant for RLRC funded by the Korean government (MSIT) (No. 2022R1A5A8026986, RLRC)

ABSTRACT The increasing complexity of Deep Neural Network (DNN) models, coupled with rising demands for energy efficiency and computational speed in edge devices, necessitates the reevaluation of conventional numerical representations and computational strategies. Traditional quantization of DNNs, which often employs low-bit integers due to their fixed hardware implementation, applies uniform granularity across all values. This can be inefficient when certain data points require finer granularity to maintain accuracy. In contrast, floating-point (FP) quantization provides greater flexibility in bit allocation, enabling more precise granularity where needed. This paper proposes mixed-precision FP arithmetic based on a genetic algorithm to find the optimal precision for each layer. To minimize rounding errors, stochastic rounding is incorporated, and layer-specific exponent bias adjustments are implemented to enhance data representation precision. Experimental results from implementing mixed-precision FP quantization on the YOLOv2-tiny model reveal a 2.9 times reduction in energy consumption per image and a 50% reduction in memory requirements, with only a negligible loss of 0.13% in mean Average Precision (mAP@0.5) as evaluated on the VOC dataset, compared to Bfloat16, a known low-cost floating-point method.

INDEX TERMS CNN, floating point quantization, genetic algorithm, hardware accelerator

I. INTRODUCTION

Deploying Deep Neural Networks (DNNs) has witnessed exponential growth due to their groundbreaking success in various domains; from image recognition to autonomous driving. However, this success has been accompanied by significant burdens of high energy consumption and computational costs [1,2]. Statistics presented by [3] illustrate that the complexity of DNN algorithms is increasing at ~10x per decade. The data that needs to be processed has increased by ~1000x per decade since the early 70s. Graphical Processing Units (GPUs) are widely used to implement DNNs as they offer high throughput of up to 11 TFLOP/s due to their ability to process extensive data in parallel [4]. However, GPUs' high power consumption has prompted researchers to study FPGA-based CNN accelerators [5-8]. System-on-chip (SoC) based accelerators take one step ahead and offer higher hardware efficiency [9-10]. As the complexity of CNNs increases, deploying them on edge devices becomes increasingly challenging due to inherent constraints in computational capacity and memory resources on these platforms [11]. Researchers are finding ways to reduce the computational complexity of DNNs [12-15]. Converting parameters and activations from an IEEE 32-bit floating-point format to a low-bit floating-point or integer format can significantly reduce on-chip memory usage and computational costs [16].

II. RELATED WORK

Quantizing DNN expedites its deployment on hardware-constrained edge devices. Traditionally, the quantization of DNN models has been dominated by low-bit integer representations [17-19]. Although INT8 quantization achieves minimal accuracy loss and is computationally cost-effective for classification algorithms, it reduces accuracy significantly when applied to object detection or semantic segmentation tasks [20]. To address the challenges of accuracy degradation due to the limited dynamic range of INT8 data representation, [21-23] have explored asymmetric quantization, which

allocates different numbers of bits for the negative and positive ranges with a non-zero offset. Non-uniform quantization methods were used [24-27] to reduce the quantization errors, and more precision was assigned to the data deemed necessary.

Researchers are exploring the potential of low-bit FP formats for applications where accuracy is paramount. Quantization-Aware Training (QAT) was used by [28], and their experiment showed accuracy performance similar to baseline FP32 results. Like other training-based quantization, QAT also achieves high accuracy by training the CNN repeatedly with quantized activations and weights. This accuracy gain is overshadowed by the cost of excessive training time, the challenges of acquiring a complete dataset, and the failure to find optimal quantization for complex CNNs such as object detection models.

Post-training quantization (PTQ) methods use pre-trained models with full precision and quantize the activations and weights in low precisions [29-30]. Analysis of [31] identified that the most common distribution in neural networks is Gaussian; therefore, by analyzing the Mean Square Error (MSE), they evaluated the FP8 format with different configurations of exponent and mantissa. However, MSE treats all errors equally by averaging the squared differences and does not account for the non-uniform distribution of errors. A recent work [32] on FP8 PTQ evaluated the per-channel scaling for Large language models (LLMs) due to high inter-channel variance. However, per-channel scaling has minimal benefit for classification or object detection algorithms where inter-channel variance could be higher. A detailed study was conducted by [33] on PTQ for various formats, including INT8 and mixed FP8. Their experiment used clipping and round to near, which introduces a quantization bias and does not utilize the complete information in the lower significant bits. This loss of information can be particularly detrimental in classification and object detection models.

III. PAPER CONTRIBUTION

Previous works on low-precision inference focused mainly on CNN-based classification models [34,35]. However, less attention has been given to adapting these techniques for object detection algorithms. Additionally, existing methods couldn't exploit the full potential of FP quantization by using symmetric bias, deterministic rounding, and single precision for the whole network. We propose a mixed low-precision FP quantization to address the limitations of previous research works. The main contributions of our paper are summarized as follows:

1. Recognizing the limitations of both symmetric and conventional asymmetric exponent biases, we extensively studied DNN models and proposed layer-wise asymmetric bias adjustments. These adjustments are centered around the mean of the exponent distribution rather than the data itself. Our algorithm defines the range of the exponent based on its standard deviation and automatically identifies the optimal exponent to represent this range within the chosen standard deviation.

2. By evaluating the Signal-to-Quantization Noise Ratio (SQNR)—which accurately measures how closely a quantized model's output matches the original model's output—we calculated the mantissa widths that satisfy the SQNR threshold. These calculated widths are then used as parameters for the Genetic Algorithm that optimizes the mantissa width for each layer.

3. To address the quantization error introduced by deterministic rounding, such as the IEEE standard round to nearest, which consistently rounds to the nearest quantized value and introduces bias, we employed stochastic rounding in DNNs. This method probabilistically rounds values based on their proximity to potential quantized states, effectively minimizing this bias.

IV. BACKGROUND

A. FLOATING POINT QUANTIZATION PROCESS

Floating Point (FP) Quantization involves a judicious trade-off between the numerical range and precision. The width of the exponent determines the range of representable magnitudes. A wider exponent allows the representation of values with large range. On the other hand, the width of the mantissa influences the precision of these numbers.

Fig. 1 compare three floating point formats: FP4 (1-bit exponent, 3-bit mantissa), FP5 (2-bit exponent, 3-bit mantissa), and FP6 (2-bit exponent, 4-bit mantissa). As illustrated in Fig. 1, the distribution of positive values across various floating-point (FP) configurations highlights the significant impact of changes in exponent and mantissa widths on both the density of values and the range they can represent. This visualization emphasizes the critical considerations in optimizing FP quantization to achieve the desired computational efficiency and numerical accuracy.

The studies referenced in [28] reveal that the integer format performs best for uniform distributions. However, since most DNN layers typically exhibit a Gaussian distribution or otherwise non-uniform distributions, FP formats can outperform integer formats in terms of accuracy. Fig. 2 compare example distributions of three data formats: FP32,

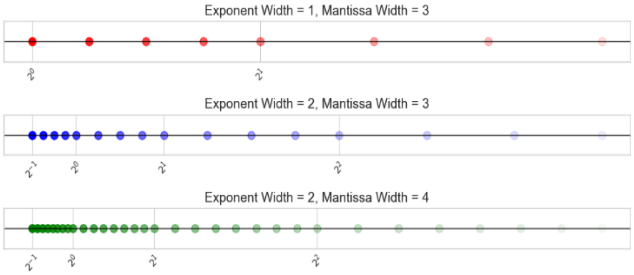


FIGURE 1. FP Quantization process using different precisions

FP8(1-bit sign, 3-bit exponent, 4-bit mantissa), and INT8. It demonstrates that the uniformly distributed quantization levels in this INT8 quantization example significantly wastes many

available grid points. In contrast, FP8 quantization concentrates grid points around the mean, utilizing most of available grid points and effectively covering most of the range required for higher accuracy.

B. MANTISSA WIDTH REDUCTION

The most straightforward method of achieving low-cost FP representation is by reducing the width of the mantissa. The FP32 format uses a 23-bit mantissa, allowing for a significant degree of precision. By reducing the mantissa width, it's possible to decrease the storage size of each number and

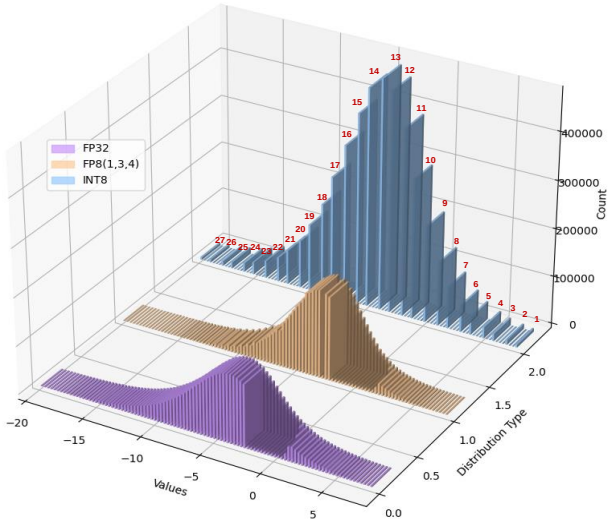


Figure 2. Comparing FP32 Distribution Quantization: FP8 vs INT8

increase the computational speed. Reducing the mantissa width can decrease the storage size required for each number and enhance computational speed. However, this adjustment comes at the cost of reduced precision and increases quantization errors. With a narrower mantissa width, real numbers can no longer be represented accurately, necessitating rounding to approximate these numbers within the available precision. Rounding decisions significantly impact the accuracy of numerical computations; therefore, the choice of rounding method is critical. The most prominent rounding methods are listed below, along with their advantages and limitations:

1. *Truncation:* This is the simplest method of rounding, in which the mantissa bits beyond the reduced precision limit are truncated. This method consistently underestimates or overestimates the values, leading to significant errors that can skew the results.
2. *Round to Nearest:* This is the default IEEE rounding mode, where the number is rounded to the nearest representable value. It minimizes the average rounding error but can introduce a bias, especially in aggregate computations where the rounding errors accumulate in a particular direction.
3. *Stochastic Rounding (SR):* In stochastic rounding (SR), rather than simply truncating or rounding to the nearest value, numbers are rounded up or down based on their proximity to potential representations, with the probability directly tied to their closeness. Fig. 3 presents two scenarios of this method: In the first, despite a low likelihood of rounding up, the number is rounded up when the random value generated is below the calculated probability for rounding up. In the second, although rounding up is more likely, the number is rounded down when the random value exceeds the probability of rounding up. The key advantage of SR lies in its ability to distribute rounding errors across the data evenly.

C. EXPONENT WIDTH REDUCTION

Another method for developing low-cost FP is by reducing the exponent width. Reducing the exponent width narrows the range of representable values, increasing the risk of overflow and underflow errors. Adjusting the bias is essential for recalibrating the midpoint of the exponent range and maintaining a balanced representation of both positive and negative values.

In neural networks, particularly when each layer's input feature maps (FMAPs) and weights have distinct biases, the

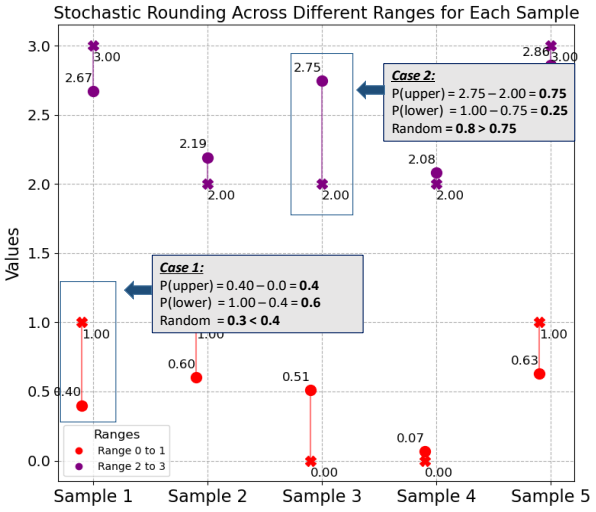


Figure 3. Example of Stochastic Rounding

bias is typically adjusted after the accumulation phase in convolutional or fully connected layers. Specifically, the previous layer's input FMAP bias has to be removed to nullify its effect on the current layer. The bias for the next layer's input FMAP, which is based on the distribution of the output of the current convolutional or fully connected layer and referred to as the output FMAP bias throughout this paper, is then added to prepare the data for appropriate scaling in the subsequent layers. Equation 1 shows this step that subtracts the current layer's input FMAP bias and adds the current layer's output FMAP bias, which is used as the input FMAP bias of the next layer.

$$E_{Adjusted} = E_{Accumulated} - Bias_{in\_fmap} + Bias_{out\_fmap} \quad (1)$$

In Equation 1, $E_{Accumulated}$ represents the exponent of the accumulated result from the convolution of fully connected layer outputs.

D. OPTIMIZING MIXED PRECISION QUANTIZATION FOR DNNS USING GENETIC ALGORITHM

Using a uniform bit width for all layers in Deep Neural Networks (DNNs) can lead to suboptimal performance on resource-constrained hardware. Each layer may have different precision and numerical range requirements, because some layers are more sensitive to quantization errors, requiring higher precision, while others can operate effectively with reduced precision without compromising the overall network's performance. To address this challenge, we need to explore mixed Floating-Point (FP) quantization strategies, which involve analyzing the network to identify the precision requirements of individual layers. Fig. 4 illustrates the impact of different quantization strategies on a simple DNN model like AlexNet, demonstrating how each strategy affects the model's accuracy and latency.

Searching all possible number formats exhaustively for the optimal bit width for weights and activations in every layer is impractical due to the vastness of the design space. For instance, the search space for ResNet-50 is estimated to be 10^{90} [36], which is higher than the total number of atoms in the universe (10^{80}).

Genetic Algorithms (GAs) can navigate the issue of ample search space by exploring and exploiting the space efficiently. Inspired by evolutionary processes, GAs use a chromosome-like data structure to represent potential solutions and employ mutation operators to retain essential information. This approach enables the exploration of multiple regions of the solution space simultaneously, making these algorithms more

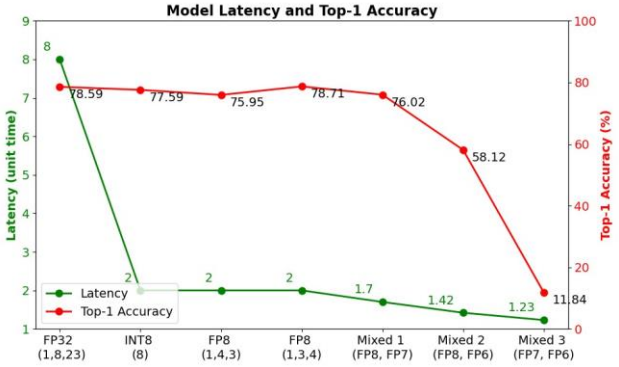


Figure 4. Impact of Quantization Strategies on AlexNet Example

resilient against local optima than traditional single-point search algorithms. Although GAs do not guarantee the global optimum, they are generally faster at finding a sufficiently good solution than exhaustive search methods.

V. METHODOLOGY

The proposed method implements mixed precision floating-point (FP) quantization through three key stages: (i) determining the optimal exponent bits by identifying the dynamic range that encapsulates the maximum amount of useful data; (ii) calculating the mantissa widths that meet the Signal-to-Quantization Noise Ratio (SQNR) criterion for at within each layer, minimizing quantization error through the use of stochastic rounding, and (iii) employing a Genetic Algorithm to fine-tune the precision of each layer in a deep neural network (DNN).

A. DYNAMIC RANGE OPTIMIZATION

Traditional FP quantization methods typically involve scaling and clipping the real value to a predefined range before applying uniform quantization levels. However, these methods are particularly effective for uniform distributions but become inadequate when dealing with the complex data distributions found in CNNs, especially those that exhibit characteristics closer to Gaussian or leptokurtic distributions.

This complexity is highlighted in Fig. 5, which shows the distribution of output feature data of Layer 0 of an example CNN mode, YOLOv2-Tiny.

In Fig. 5, over 84% of the data clusters around the mean, indicating that traditional clipping methods are insufficient for Gaussian or leptokurtic distributions where the data is significantly concentrated around the mean.

By analyzing the exponents distribution in Fig. 6, a more effective strategy can be deduced where the mean of the exponents is positioned at the midpoint of the exponent range. This method allocates the precision more selectively to the

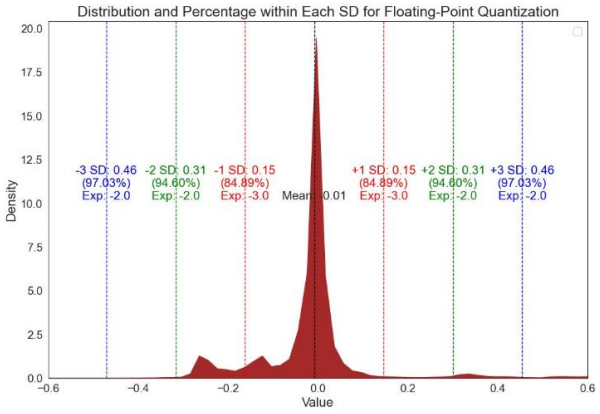


Figure 5. Example of Real Values with a Leptokurtic Distribution

areas where it is most needed while permitting the exclusion

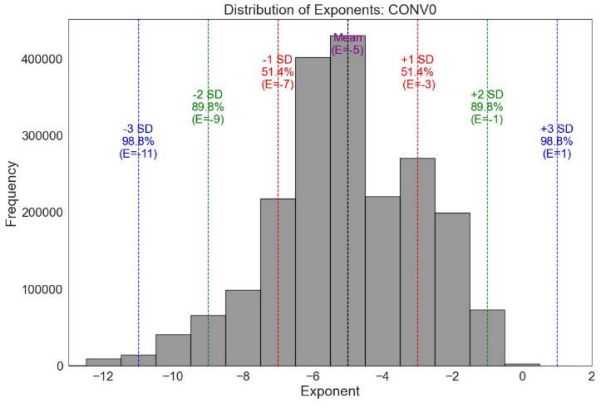


Figure 2. Example of Exponents from YOLOv2-tiny Layer 0 Output FMAP

of some outliers without a significant loss of overall information.

Algorithm 1: Exponent Range Optimizer

```
1:.procedure OptimizeExponentRanges(network, data, SD_Range, base_accuracy, d_thresh )
2:   Input:
3:     network: Neural Network Model
4:     data : Inference dataset to feed through the network
6:     SD_Range : Array of Standard Deviations e.g. [1, 2, 3, 4]
7:     base_accuracy : Baseline accuracy/mAP based on FP32 data
8:     d_thresh : Acceptable accuracy degradation
9:   Output: Optimized parameters for each layer and fmaps
11:  results ← [ ]
12:  layer_exponent_counts← aggregate_exponent(network, data)
13:  for SD in SD_Range do
14:    for each layer in network.layers do
15:      exponent_FMAPs ← layer_exponent_counts[layer.id]
16:      act_settings ←  OptExpSetting(exponent_FMAPs)
17:      weight_exponents ← extract_exponent_count( layer.weights)
18:      weight_settings ← OptExpSetting( weight_exponents)
19:    end for
20:    model_accuracy ← Accuracy(act_settings[SD], weight_settings[SD], network, data)
21:    degradation ← abs(model_accuracy -  base_accuracy )
22:    if degradation <  d_threshold then
23:        results.append(layer.id, act_settings[SD], weight_settings[SD])
24:        break
25:  return results
26: end procedure
```

Supporting Functions:

```
1: function  aggregate_exponent(network, data)
2:    exp_counts ← {}
3:    for input in data do
4:      all_layer_fmaps ← network.forward_all_layers(input)
5:      for layer_id, FMAPs in all_layer_fmaps do
6:        for fmap in FMAPs:
7:          exp ← extract_exponent(fmap)
8:          exp_counts[layer_id][exp]=exp_counts[layer_id].get(exp, 0) + 1
9:    for layer_id in exp_counts do
10:     total_fmaps = sum(exp_counts[layer_id].values())
11:     for exponent in exp_counts[layer_id]  do
12:       exp_counts[layer_id][exponent] /= total_fmaps
13:   return exp_counts
14:
15: function OptExpSetting (data,  SD_Range ,  base_accuracy , d_threshold)
16:    mean ← mean(data)
17:    std ← std(data)
18:    E_min, E_max ← mean - SD * std, mean + SD * std
19:    bits_required ← ceil(log2(E_max - E_min + 1))
20:    max_possible_values ← 2^bits_required
21:    current_range_values ← E_max - E_min + 1
22:    if max_possible_values > current_range_values then
23:      extra_val_per_side ← (max_possible_values - current_range_values) / 2
24:      E_min ← E_min - extra_val_per_side
25:      E_max ← E_max + extra_val_per_side
26:      layer_setting ← {E_min, E_max, Bias← -E_min,  bits_required }
27:    end if
28:   return  layer_setting
```

We have developed an algorithm inspired by these studies that optimizes the exponent range of data in a neural network. The objective is to determine the optimal exponent range for each layer's weights and feature maps (FMAPs) while maintaining the accuracy close to the baseline. In the context of classification models, this accuracy is measured by Top-1/Top-5 accuracy metrics. While for object detection models, it is evaluated by mAP@0.5.

Algorithm 1 consists of the main procedure and supporting functions that compute the optimal exponent settings and works as follows:

i. *FMAP Exponents Aggregation (Line 12):* The algorithm begins by collecting exponent of FMAPs (input image and output of convolution or fully connected) across the network using the '**aggregate_exponent'** function. This function analyzes the distribution of exponent values for each layer from the selected dataset, normalizing these exponent values to represent the average distribution of exponents per layer (Lines 1-13 in aggregate_exponent). This normalization facilitates a more standardized comparison of exponents across different layers within the same dataset. Fig. 7 shows the example normalization of exponent distributions from two separate activations within the same neural network layer.

ii. *Exponent Optimization (Lines 14-16):* For each layer, the algorithm extracts the exponent distribution for the layer's weights and FMAPs, and calculates the optimal exponent range using the '**OptExpSetting'** function (Lines 15-27 in the main procedure). It calculates the number of bits required to represent the initial range. If the number of representable values with the calculated bits exceeds the current range, the algorithm symmetrically expands the range to maximize the utilization of available bits (Lines 19-25 in OptExpSetting).

iii. *Accuracy Evaluation and Optimization(Lines 20-24):* Each proposed range is evaluated by simulating the model's accuracy with the given exponent range. The algorithm adjusts the ranges to minimize degradation relative to a baseline accuracy obtained from a 32-bit full-precision model. The simulation process is encapsulated in the Accuracy function

iv. *Results Compilation (Lines 23-25) :* The optimized exponent range, bias, and required bits to represent that range for each layer are then compiled into a result set. This provides detailed insights into how each layer's quantization parameters must be adjusted (Lines 18-20 in the main procedure).

B. QUANTIZATION ERROR MINIMIZATION AND OPTIMAL MANTISSA WIDTH DETERMINATION

Addressing the issue of rounding errors is crucial before finding the optimal number of mantissa bit widths for each layer. Traditional rounding methods such as Round to Nearest (RN) can introduce significant error by consistently rounding values in one direction, thereby distorting data distribution. To counteract this, we adopt Stochastic

Rounding (SR), which preserves the statistical properties of the original data more effectively.

As illustrated in Fig. 8, significant anomalies or spikes in the data distribution are observed when the mantissa is rounded using the RN method. These spikes indicate the systematic bias introduced by RN, where values tend to cluster around specific rounded numbers. In contrast, when SR is employed, the resulting distribution is smoother and closer to the original Floating Point 32-bit (FP32) distribution.

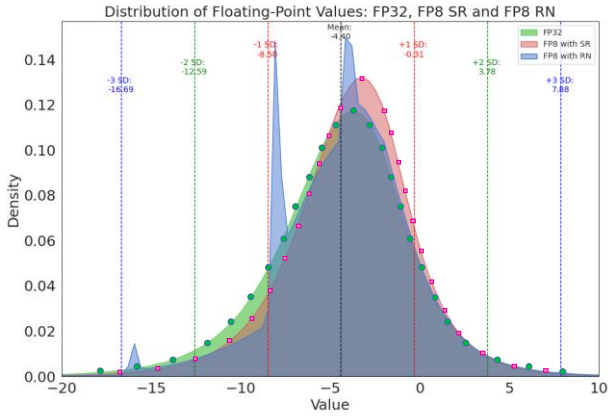


Figure 4. Rounding Error Analysis

We present an algorithm to identify the range of mantissa widths that meet the SQNR threshold for each layer, ultimately forming chromosomes that serve as initial solutions for the genetic algorithm. To calculate the mantissa widths, the algorithm 2 works as follows:

i. *Valid Widths for Images and Weights (Line 13-23):* The algorithm starts by calculating the valid mantissa widths for each image and each layer's weights using the '**FindValidWidths'** function. This function loops through potential mantissa widths from 1 to the maximum possible mantissa width. It then computes the Signal-to-Quantization-Noise Ratio (SQNR). If the SQNR meets or exceeds the specified threshold, the width is considered valid and added to the list of valid widths. If no valid width is found, the maximum mantissa width is added to the list.

ii. *Valid Widths for Output Activations (Lines 25-38):* For each image, the algorithm iterates through the layers, applying the corresponding weights to calculate the layer outputs. It then determines the valid mantissa widths for these outputs using the '**FindValidWidths**' function. If no valid width is found for the layer outputs, the maximum mantissa width is added to the list for that layer's outputs.

iii. *Termination and Output (Lines 40-41):* Once valid widths are determined for all layers, the algorithm returns three lists: valid mantissa widths for images, weights, and outputs. Since the valid mantissa widths lists for both images and output activations have more than one solution, these lists are used for generating the initial population of the genetic algorithm, ensuring that

Algorithm 2: Determining Mantissa Widths satisfying SQNR Threshold

1: **procedure** DetermineMantissaWidths(*images, weights, $SQNR_{thres}$, Max_{Man_WD}, Total_Layers*)
2: **Input**:
3: images: List of input images
4: weights: List of weights for each layer
5: $SQNR_{thres}$: Minimum SQNR threshold that each layer must meet
6: $Max_{Man\_WD:}$ Maximum allowed width of mantissa
7: Total_Layers: Total number of layers in the neural network
8: **Output**:
9: $Valid_{M\_WD\_WTs}$: Valid mantissa widths for each layer's weights
10: $Valid_{M\_WD\_IMGs\_OUTs}$: Combined Valid mantissa widths for images and output activations satisfying SQNR threshold
11:
12: $Valid_{M\_WD\_IMGs} \leftarrow [\,]$
13: **for** each image in *images* **do**
14: **valid_widths** ← FindValidWidths(*image, $SQNR_{thres}$, Max_{Man_WD}*)
15: $Valid_{M\_WD\_IMGs}$.append(*valid_widths*)
16: **end for**
17:
18: $Valid_{M\_WD\_WTs} \leftarrow [\,]$
19: **for** $layer_{weights}$ in *weights* do
20: *valid_widths* ← FindValidWidths($layer_{weights}$, $SQNR_{thres}$, Max_{Man_WD})
21: $Valid_{M\_WD\_WTs}$.append(*valid_widths*)
22: **end for**
23:
24: *total_images* ← len($Valid_{M\_WD\_IMGs}$)
25: $Valid_{M\_WD\_IMGs\_OUTs} \leftarrow [\,]$
26: $layer_{outputs} \leftarrow [\,]$
26: **for** i ← 0 to *total_images* - 1 **do**
27: $combined_{width}$ ← [$Valid_{M\_WD\_IMGs}$[i]] #Appending image widths
28: **for** j ← 0 to *Total_Layers* **do**
29: **if** j == 0 **then**
30: $layer_{output}$← ApplyLayer(*image[i], weights[j]*)
31: **else**
32: $layer_{output}$ ← ApplyLayer($layer_{outputs}$[i][j], *weights[j]*)
33: **end if**
34: $layer_{outputs}$ ← $layer_{output}$
35: $valid_{widths}$ ← FindValidWidths($layer_{outputs}$, $SQNR_{thres}$, Max_{Man_WD})
36: $combined_{width}$.append($valid_{widths}$)
37: **end for**
38: $Valid_{M\_WD\_IMGs\_OUTs}$.append($combined_{width}$) #Appending output activations mantissa widths
39: **end for**
40:
41: **return** $Valid_{M\_WD\_WTs}$, $Valid_{M\_WD\_IMGs\_OUTs}$
42: **end procedure**
43:
44: **function** FindValidWidths(*data, SQNR_thres, Max_Man_Width*)
45: *valid_widths* ← [ ]
46: **for** width ← 1 to *Max_Man_Width* **do**
47: *SQNR* ← CalculateSQNR(*data, width*)
48: **if** *SQNR* ≥ SQNR_thres then
49: *valid_widths*.append(*width*)
50: **end if**
51: **end for**
52: **if** len(*valid_widths*) == 0 **then**
53: *valid_widths*.append(**Max_Man_Width**)
54: **end if**
55: **return** *valid_widths*
56: **end function**
57:
58: **function** CalculateSQNR(*layer, mantissa_width*)
59: *signal* ← GetSignal(*layer*)
60: *quantized* ← Quantize(*signal, mantissa_width*)
61: *noise* ← *signal - quantized*
62: *signal_power* ← mean($signal^2$)
63: *noise_power* ← mean($noise^2$)
64: *SQNR* ← 10 * log10(*signal_power / noise_power*)
65: **return** *SQNR*

the widths determined for each layer conform to the SQNR requirement.

C. MIXED PRECISION USING GENETIC ALGORITHM

This section describes the proposed method that utilizes genetic algorithm (GA) to synergistically optimize the precision of all layers in the Deep Neural Network (DNN), which is presented by Algorithm 3. It identifies the optimal solution from a set of possible configurations. It is further broken down into five sub-algorithms, labeled 3.1 to 3.5, each detailing a specific process inspired by biological evolution. Below are the steps for the high-level Algorithm 3:

i. *Initialize Population (Line 11):* The algorithm begins by initializing a population of unique solutions using the layer-wise valid mantissa widths provided by Algorithm 2.
ii. *Evolution Iterations (Lines 17-43):* The process iterates through a predefined number of generations ($Max_{Generations}$), simulating the evolutionary cycle in which populations evolve.
iii. *Fitness Evaluation (Lines 21):* Each chromosome in the population is evaluated to determine how well it solves the problem. During this step, the algorithm also keeps track of the best solution found so far (*bestSolution*).
iv. *Parent Selection (Line 27):* Two parents are selected from the current population based on their fitness scores.
v. *Crossover (Line 28):* A new chromosome (child) is created by combining genetic information from the two selected parents.
vi. *Mutation (Line 29):* The new chromosome undergoes mutations with a probability defined by the (*MutationRate*). During mutation, random genes in the chromosomes are altered, introducing variability into the gene pool.
vii. *Population Replacement (Lines 30-31):* At the end of each generation, the old population is completely replaced by the new population, which contains the children created during the reproductive phase. This step aims to simulate the generational change in biological populations.
viii. *Escaping local Maxima Trap (Lines 32-42)*: The algorithm determines the fitness of each generation and compares it with the previous generation. If the fitness is not improving for more than a certain number of generations, then the mutation rate is adjusted by an adjustment factor to cause more genes to undergo mutation.
ix. *Return Best Solution (Line 30):* After all generations have been processed, the algorithm returns the best solution found during the entire evolutionary process.

a. POPULATION INITIALIZATION

Algorithm 3.1 initializes a population of random solutions. Each solution, represented as a chromosome, encapsulates a

Algorithm 3: Main Genetic Algorithm for Layer-wise Precision Optimization

1: **procedure** GeneticAlgorithm
2: **Input:**
3: N: Number of layers in DNN
4: $Mantissa_{w\_Image\_out}$: Initial mantissa widths for images and layer-wise output activations using Algorithm 2
5: P_{Size}: Size of the population
6: $Max_{Generations}$: number of generations genetic algorithm will run
7: InitialMutationRate: starting probability for mutations
8: $adjustment_{Factor}$: factor to adjust mutation rate and improvement threshold. E.g. 0.1 for 10% adjustment
9: **Output:** Best solution found
10:
11: *population*←InitializePopulation($Mantissa_{w\_Image\_out}$, P_{Size})
12: *bestSolution* ← None
13: *bestFitness* ← -∞
14: *mutationRate* ← InitialMutationRate
15: $noImprovement_{Counter}$ ← 0
16: $improvement_{Threshold}$ ← int($adjustment_{Factor}$ * $Max_{Generations}$)
17: **for** generation ← 1 to $Max_{Generations}$ **do**
18: *newPopulation* ← [ ]
19: **for** i ← 0 to P_{Size} - 1 **do**
20: *chromosome* ← population[*i*]
21: *fitness* ← CalculateFitness(*chromosome*)
22: **if** *fitness* > *bestFitness* **then**
23: *bestFitness* ← *fitness*
24: *bestSolution* ← chromosome
25: **end if**
26: **end for**
27: *parent1, parent2* ← SelectParents(*population*, $efficiency_{Score}$, P_{Size})
28: *child* ← Crossover(*parent1, parent2*)
29: $mutated_{child}$ ←Mutate(*child, mutationRate*, $Mantissa_{w\_Image\_out}$, *N*)
30: *newPopulation*.append($mutated_{child}$)
31: *population* ← newPopulation
32: **If** *bestFitness[generation]* > *bestFitness[generation -1]* **then**
33: $noImprovement_{Counter}$ ← 0 # Reset counter for improvement
34: **else**
35: $noImprovement_{Counter}$ += 1 # if no improvement
36: # Adaptive Mutation Rate Adjuster
37: **if** $noImprovement_{Counter}$ >= improvementThreshold **then**
38: # Decrease mutation probabilty for more genes to go mutation
39: *mutationRate* ← min(*0, mutationRate * (1 -* $adjustment_{Factor}$))
40: $noImprovement_{Counter}$ ← 0 # Reset counter after adjustment
41: **end if**
42: **end if**
43: **end for**
44: **return** *bestSolution*
45: **end procedure**

Algorithm 3.1: Population Initialization

1: **procedure** InitializePopulation($Mantissa_{w\_Image\_out}$, P_Size)
2: **Input**:
3: $Mantissa_{w\_Image\_out}$: valid mantissa widths for images & activations
4: P_Size: size of the population
5: **Output**: Initialized population
6: *population* ← [ ]
7: for i ← 0 to P_Size-1 do
8: chromosome ← random($Mantissa_{w\_Image\_out}$) // Select a random element from Mantissaw_Image_out
9: population.append(chromosome) // Append the chromosome to the population
10: end for
11: return population
12: end procedure

configuration for the mantissa widths of each layer, starting with the width for the input image followed by the widths for each layer's output activations obtained from Algorithm 2. This approach ensures that the initial population is composed of valid configurations that meet the SQNR criteria.

b. FITNESS CALCULATION

The core of the genetic algorithzm's efficiency lies in its ability to evaluate and identify the most promising solutions. The fitness of each chromosome in Algorithm 3.2 is determined by the accuracy of the target DNN model configured according to the mantissa widths specified by the chromosome. Specifically, the model is executed on a validation dataset, and its predictive accuracy is measured. However, the fitness score is not solely based on accuracy; it is calculated as the accuracy divided by the sum of the mantissa widths. This approach ensures that high fitness scores indicate configurations that achieve desirable accuracy while also minimizing the mantissa width sum. This balance makes these configurations candidates for perpetuation through subsequent generations.

Algorithm 3.2: Fitness Calculation

1: **procedure** CalculateFitness(chromosome)
2: **Input:** Chromosome representing mantissa widths
3: **Output:** Fitness value (accuracy for classification models or mean average precision for Object Detection)
4: *accuracy* ← EvaluateDNN(*chromosome)*
5: *precisionSum* ← Sum(*chromosome*) # Sum Layer wise widths
6: *fitness*← *accuracy*/precisionSum # Higher is better
7: **return** *fitness*
8: end procedure

Table 1 shows the general initial population and their corresponding fitness calculation while Table 2 displays an example of the YOLOv2 tiny initial population and fitness. Each chromosome in this population consists of genes representing the mantissa widths for input images (IMG) and output activations for each layer (L_0 to L_{Last}). The fitness metric is calculated by dividing the Top-1 or Top-5 accuracy for classification models, or the mean Average Precision (mAP@0.5) for object detection models, by the sum of mantissa widths. Other performance metrics can also be employed for different types of DNNs.

c. PARENTS SELECTION

Algorithm 3.3 functions by selecting the most promising solutions from a population of chromosomes, each representing a unique configuration of mantissa widths across the neural network's layers.

Algorithm 3.3: Select Parents based on Highest Fitness

1: **procedure** SelectParents(*population*, *nparents*, *fitness*)
2: **Input:**
3: population – array of chromosomes
4: nparents – number of parents to select
5: fitness – fitness calculated using Algorithm 3.2
6: **Output:** Array of selected parent chromosomes
7: *parents* ← [ ] # Create an empty parents list
8: **for** *i* ← 0 to *nparents* − 1 **do**
9: *best* ← index of maximum value in *fitness*
10: *parents*[*i*] ← *population*[*best*] #Store the best chromosome
11: *fitness*[*best*] ← 0 # Set fitness/accuracy zero to avoid reselection
12: **end for**
13: **return** *parents*
14: end procedure

The method of selecting the parents involves the following steps:

Table 1: Initial Population with Fitness

Population	IMG	L0	L1	L2	-	-	L_Last	Accuracy (%) (A)	Mantissa Widths Sum (S)	Fitness (F=A/S)
1	M1	M3	M3	M4	-	-	M2	A1	S1	A1/B1
2	M1	M4	M3	M1	-	-	M2	A2	S2	A2/B2
3	M2	M3	M4	M2	-	-	M1	A3	S3	A3/B3
-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-
P_SIZE	M1	M2	M3	M1	-	-	M4	A(P_SIZE)	S(P_SIZE)	A(P_SIZE)/S(P_SIZE)

Table 2. Initial Population with Fitness (YOLOv2 Tiny Example)

Population	IMG	L0	L1	L2	-	-	L8	Accuracy (%) (A)	Mantissa Widths Sum (S)	Fitness (F=A/S)
1	4	5	6	4	-	-	3	56.40	40	1.41
2	3	5	5	5	-	-	4	54.10	35	1.55
3	4	5	5	4	-	-	3	55.24	39	1.42
-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-
100	3	4	5	3	-	-	4	56.01	36	1.56

Table 3: Sorted Parents From Initial Population based on Fitness

Population	IMG	L0	L1	L2	-	-	L_Last	Fitness (F=A/S)
Parent 1 1 P(P_Size*2)	M2	M3	M4	M2	-	-	M1	F_HIGHEST[0]
Parent 2 2 P3	M1	M3	M3	M4	-	-	M2	F_HIGHEST[1]
3 P4	M1	M4	M3	M1	-	-	M2	F_HIGHEST[2]
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
P_SIZE P(P_Size*2)-1	M4	M4	M3	M1	-	-	M4	F_HIGHEST[P_SIZE]

Table 4. Sorted Parents From Initial Population based on Fitness (YOLOv2 Tiny Example)

Population	IMG	L0	L1	L2	-	-	L8	Fitness (F=A/S)
Parent 1 1 P200	3	4	5	3	-	-	4	1.62
Parent 2 2 P3	3	4	4	5	-	-	4	1.61
3 P4	4	5	5	3	-	-	3	1.59
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
100 P199	5	6	5	6	-	-	4	1.28

1. *Initialization:* The algorithm begins by initializing an empty array, parents, designed to store the selected chromosomes. The size of this array is determined by the parameter *nparents*, which specifies the number of parents to be selected.
2. *Selection Process:* For each parent to be selected, the algorithm identifies the chromosome with the highest fitness score from the fitness array, which contains the fitness evaluations of all chromosomes in the current population. Upon identifying the fittest chromosome, it is transferred to the parent array. This ensures that the genes associated with high fitness are carried over to the next generation.
3. *Preventing Reselection:* To maintain diversity in the selection process and prevent the algorithm from converging prematurely on local optima, the fitness score of the selected chromosome is artificially reduced to a zero. This modification prevents the reselection of the same chromosome within a single iteration, encouraging a broader exploration of the genetic landscape.
4. *Output:* The process repeats until the 'parents' array contains the required parent chromosomes.

Table 3 shows the general sorted parents in descending order of their fitness from the initial population, while Table 4 demonstrates an example of sorted parents in YOLOv2 Tiny.This sorting allows us to identify and select the most fit candidates for further genetic operations.

d. CROSSOVER

Once parents are selected, they undergo crossover to combine their genetic information and generate new offspring using Algorithm 3.4. A two-point crossover method is employed, where segments of the precision settings are inherited from different parents, increasing genetic diversity [37].

Fig. 9 illustrates the two-point crossover process using the first two parents listed in Table 3, showing how genetic

Algorithm 3.4: Two-Point Crossover

```
procedure Crossover(parent1, parent2)
  Input:
    parent1: First parent chromosome
    parent2: Second parent chromosome
  Output: A new child chromosome
  child ← [ ]
  point1 ← RandomInt(0, len(parent1) - 1)
  point2 ← RandomInt(point1 + 1, len(parent1))
  for k ← 0 to len(parent1) - 1 do
    if k < point1 or k >= point2 then
      child.append(parent1[k])
    else
      child.append(parent2[k])
    end if
  end for
  return child
end procedure
```

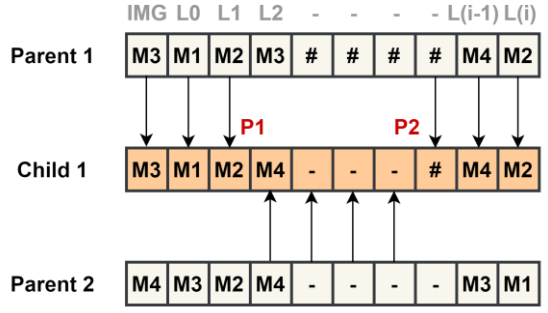


Figure 5. Example of Fixed Crossover

information from both parents is combined to create a new offspring for YOLOv2-tiny.

e. MUTATION

The Mutation in Algorithm 3.5 introduces population variability by randomly altering a layer's precision settings within a chromosome at a specified mutation rate.

Algorithm 3.5: Mutate

procedure Mutate($Child_{Chromosome}$, MutationRate, Layers, P_{size})
  Input:
    $Child_{Chromosome}$: Array of genes (mantissa widths of each layer)
    Mutation rate: Likelihood of each gene undergoing mutation
    Layers: Total number of Genes in a chromosome
    P_{size}: Population Size
  Output: A potentially mutated chromosome
  for j ← 0 to P_{Size} - 1 **do**
    for i ← 0 to *Layers* - 1 **do**
      if Random() < MutationRate **then**
        # Choose a random valid mantissa width for each gene
        $Mutated_{chromosome}[j][i]$ ← RandomChoice($Child_{Chromosome}[j][i]$)
      end if
    end for
  end for
  return $Mutated_{chromosome}$
end procedure

The algorithm performs mutation in the below steps:

1. *Loop Over Genes:* The procedure iterates over each gene in the chromosome, indexed by *i*, from 1 to *N* (where *N* is the total number of genes in the chromosome).
2. *Mutation Decision:* The algorithm generates a random number using **Random()**, a function that returns a floating-point number between 0.0 and 1.0 for each gene. It then compares this number to the mutation rate. If the random number is greater than the mutation rate, a mutation occur for this gene.
3. *Performing Mutation:* If a mutation is triggered, the gene at position *i* in the chromosome is replaced with a new value randomly selected from available mantissa widths, excluding its current value. This ensures that the mutation leads to a change in the gene's value.
4. *Completion:* This process is repeated for each gene in the chromosome. Once all genes have been potentially mutated, the modified chromosome is returned as the output.

Fig. 10 illustrates the mutation process for Child 1 from Fig. 9. In this example, the mutation rate was set to 0.9. Layer 4 of Child 1 was selected for mutation since the generated random number (0.91) exceeds the mutation rate. As a result, the value of Layer 4 is changed from 3 to 4, yielding the mutated child. These mutated children replace the old population.

The main genetic algorithm runs for a selected number of generations. The best solution is selected from all the generations, ensuring the highest accuracy is achieved. This best solution provides the optimal width for each layer of the DNN model.

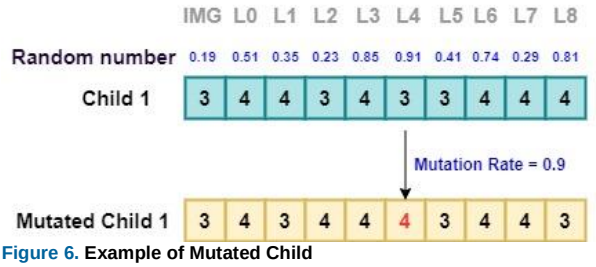


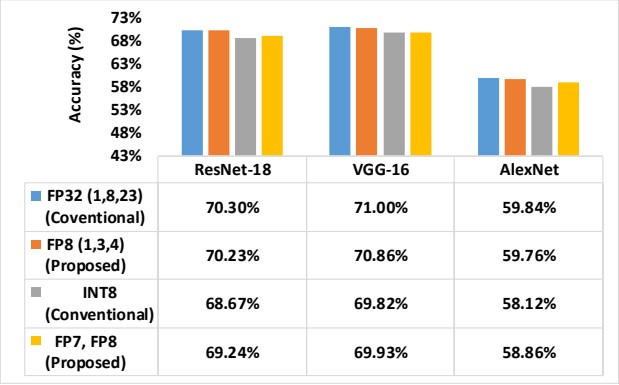
Figure 6. Example of Mutated Child

VI. EVALUATION

A. EXPERIMENTAL RESULTS

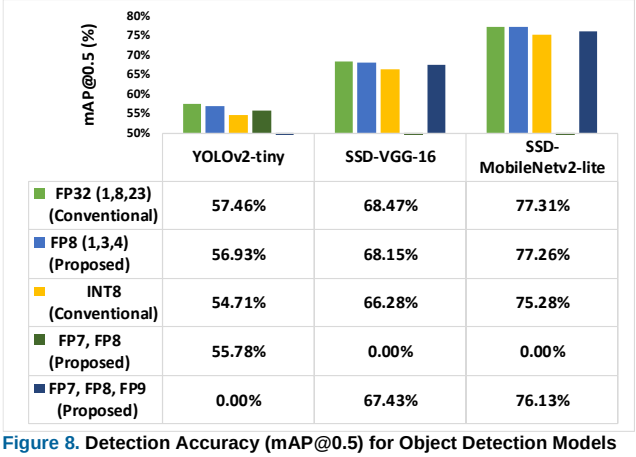
To evaluate the performance of our proposed method, we assessed several classification CNN models (AlexNet, VGG-16, and ResNet-18) on the ImageNet dataset and object detection CNN models (SSD-MobileNetv2-lite, SSD-VGG-16, and YOLOv2-tiny) on the VOC dataset.

Fig. 11 illustrates the Top-1 accuracy of common classification models on the ImageNet dataset. It compares the performance of the original 32-bit floating-point models denoted by FP32, 8-bit quantized integer models denoted by INT8 (without employing any special techniques such as batch



	ResNet-18	VGG-16	AlexNet
FP32 (1,8,23) (Coventional)	70.30%	71.00%	59.84%
FP8 (1,3,4) (Proposed)	70.23%	70.86%	59.76%
INT8 (Conventional)	68.67%	69.82%	58.12%
FP7, FP8 (Proposed)	69.24%	69.93%	58.86%

Figure 7. Top-1 Accuracy of Classification Models



	YOLOv2-tiny	SSD-VGG-16	SSD-MobileNetv2-lite
FP32 (1,8,23) (Conventional)	57.46%	68.47%	77.31%
FP8 (1,3,4) (Proposed)	56.93%	68.15%	77.26%
INT8 (Conventional)	54.71%	66.28%	75.28%
FP7, FP8 (Proposed)	55.78%	0.00%	0.00%
FP7, FP8, FP9 (Proposed)	0.00%	67.43%	76.13%

Figure 8. Detection Accuracy (mAP@0.5) for Object Detection Models

normalization folding), and the proposed methods employing FP8 and [FP7,FP8]. The mixed precision [FP7,FP8] is determined by the proposed Genetic Algorithm with stochastic rounding and flexible bias. Fig. 11 demonstrates that our method has superior classification accuracy compared to INT8, while incuring a marginal loss compared to FP32.

Fig. 12 shows the detection accuracy (mAP@0.5) for YOLOv2-tiny, SSD-VGG-16 and SSD-MobileNetv2-lite on the VOC-2007 validation dataset. The results demonstrate that our method significantly outperforms INT8, showcasing its effectiveness in object detection models with minimal loss in mAP relative to FP32.

B. Example of CNN Accelerator

We demonstrate an example CNN accelerator whose datapath and memory access are optimized by applying the proposed Mixed Precision FP Quantization Method on the architecture described in [38]. Its hardware structure and software organization are shown in Fig. 13. Initially, Algorithms 1, 2, and 3 run on software to quantize the activations for the first layer, apply layer-wise mixed precision to the weights, and determine the quantization parameters to generate the FP Quantized DNN Model. Subsequently, a microcode generator encodes the quantization parameters and control instructions for each layer. These activations, weights, and microcode are then transferred to the hardware accelerator. The accelerator utilizes DRAM to store feature maps (FMAP), weights, and microcode. The Top Controller reads the microcode from DRAM via an AXI full interface, decodes it, and loads the data into the Global Input Buffer. This data is then sent to the appropriate processing hardware blocks—Convolution/Batch Normalization, Activation, and Pooling. After processing, the output data is stored in the Global Output Buffer before being sent back to DRAM.

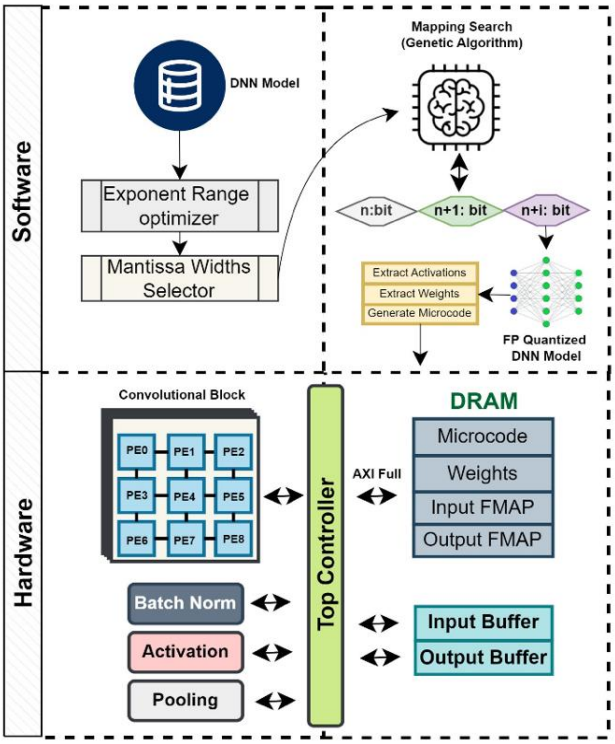


Figure 13. Proposed Mixed Precision FP Quantization Method applied to an example accelerator hardware for CNN models

Table 5: Area comparison of FP32, INT8, FP8 (1,3,4), FP7 (1,3,3)

Format	PE Area(μm2)		PE Area Reduction compared to FP32(%)	Normalizer Type	Normalizer Area(μm2)		Total Area (μm2)	Total Area Reduction compared to FP32(%)
	1 PE	576 PEs			1 Normalizer	16 Normalizers		
FP32	604	347904	0.00%	PE Integrated	0	0	348508	0.00%
INT8	117	67392	80.63%	Divider Based	2198	35168	102560	70.57%
				Shift Based	1251	20016	87408	74.92%
FP8	173	99648	71.36%	PE Integrated	0	0	99822	71.36%

Table 6. Layer-wise Energy consumption by Convolution Layer

	Convolutional Layer									Total
	L0	L1	L2	L3	L4	L5	L6	L7	L8	
Cycles	800k	400K	400K	400K	400K	512K	204K	409K	454K	951K
Time (ms)	3.21	1.60	1.60	1.60	1.60	2.05	8.19	16.38	1.82	38.06
Dynamic Power of 576 FP7 PEs (μW)	16819.20									
Dynamic Power of 576 FP8 PEs (μW)	18933.12									
Uniform Precision (U.P)	FP8	FP8	FP8	FP8	FP8	FP8	FP8	FP8	FP8	
Energy U.F (μJ)	61	30	30	30	30	39	155	311	34	772
Mixed Precision (M.P)	FP8	FP8	FP7	FP7	FP7	FP7	FP7	FP7	FP8	
Energy M.P (μJ)	61	30	27	27	27	39	138	276	34	660
Energy Difference (%)	14.50%									

To assess the hardware utilization of FP32, INT8, and FP8 we synthesized the hardware using Synopsys Design Compiler with TSMC's 65nm technology and at 250Mhz. As demonstrated in Table 5, the convolutional module (PE array) using INT8 shows an area reduction of 80.63%, while the FP8-based PE arrays shows an area reduction of 71.36% compared to the convolutional module (PE array) implemented using FP32. However, the additional logic required for INT8, such as the division-based normalization block, can offset this benefit. Although INT8 with shift-based normalization blocks still offers greater area reduction compared to FP8, these methods require additional strategies like learnable shift parameters to maintain accuracy [39]. In contrast, FP8 normalization, which involves simple mantissa shifts and exponent adjustments, is integrated within each PE with negligible impact on the area.

In our mixed precision configuration, we assume that the mixed precision [FP8, FP7] has the same chip area as FP8, since the larger FP format dominates the hardware size. The mixed precision [FP8, FP7] can have lower energy consumption than FP8, as the layers using FP7 have the MSB of FP8 hardware always set to 0.

Table 6 shows an example accelerator implemented for the YOLOv2-tiny CNN model using both uniform precision FP8 and mixed precision [FP8, FP7]. Layers L0, L1, and L8 use FP8 for mixed precision, while the other layers use FP7. Table 6 demonstrates that energy consumption varies across layers due to mixed precision arithmetic. Comparatively, the total energy consumption by the convolution module in each layer with mixed precision stands at 660 µJ. In contrast, it reaches 772 µJ when using FP8 exclusively for all layers, representing a 14.5% reduction. {*Include all numbers in the Table* → Tables are updated to accommodate the numbers mentioned in text}

C. HARDWARE COMPARISON WITH OTHER WORKS

We validated YOLOv2-tiny using the VOC dataset on the implemented accelerator discussed in section VI-B. Experimental results summarized in Table 7 reveal that our implementation achieves the highest energy efficiency and highest Frame Per Second (FPS) among previous implementations. Reference [40] uses the most LUTs among all previous work due to its reliance on 32-bit fixed-point image data and biases while employing 16-bit fixed points only for weights, accompanied by additional normalizer overhead. This normalizer is required to convert multiplication results from 48 bits to 32 bits for accumulation. While references [41] and [42] consume fewer LUTs than our proposed method, these methods come at the cost of reduced computation speed (represented as frames per second, FPS, in Table 7). [43] implements a similar CNN architecture with the same number of Processing Elements (PEs) as our work. However, as compared to their work, we were able to achieve:

i. 1.84 times smaller hardware size (represented by the number of FPGA LUTs)
ii. 2.9 times lower energy per image
iii. 1.34 times higher frames per second (FPS)

This improvement is attributed to the optimized floating point data size using mixed precision with 8 and 7 bits, compared to 16-bit fixed precision. These results underscore the potential

Table 7: Comparison with existing CNN accelerators.

Criteria	[40]	[41]	[42]	[43]	Proposed
Year	2020	2021	2023	2024	This Work
Precision	16-32-bit fixed	8-bit int	8-bit int	16-bit fixed	Mixed Precision*
CNN Model	YOLOv2-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv2-tiny	YOLOv2-tiny
Training dataset	COCO	VOC	VOC	VOC	VOC
FPGA Device	ZC706	Ultra96 V2	ZCU104	VCU118	VCU118
Clock Frequency	100MHz	250MHz	250MHz	250MHz	250MHz
Num. of DSPs	784	242	244	644	332
Num. of LUTs	182086	27300	30500	108076	58793
Num. of FFs	132869	38500	44200	64209	39412
FPS (image/sec)	7.77	8.26	9.57	13.23	17.82
DRAM Access per image (ms)	-	-	-	38.950	19.475
Time per image (ms)	128.70	121.06	104.49	75.58	56.10
Average Power (W)	11.80	4.26	1.40	4.46	2.08
Energy per image (J)	1.518	0.516	0.146	0.337	0.116

of the proposed floating point arithmetic selection and CNN accelerator design for deployment in resource-constrained environments.

VII. CONCLUSION

This paper introduces a mixed low-precision floating-point quantization method to enhance the accuracy of DNNs for resource-constrained edge devices. Our evaluation of this method on object detection and classification algorithms demonstrates superior accuracy consistently compared to traditional 8-bit integer (INT8) quantization methods.

Despite these improvements in accuracy, the hardware costs associated with our method remain comparable to those of INT8 across various scenarios. This is primarily due to the hardware support for the highest quantization level of FP8 within mixed-precision architecture. Further research is needed to explore the employment of even lower precision levels, such as FP6 or smaller, to achieve more significant advantages over INT8 quantization, particularly in terms of hardware utilization.

REFERENCES

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint, arXiv:1409:1–14, 2014.

[2] Abdelouahab, Kamel et al. “Accelerating CNN inference on FPGAs: A Survey.” ArXiv abs/1806.01683 (2018): n. pag.

[3] Sundaram, N. (2012). Making Computer Vision Computationally Efficient. UC Berkeley. ProQuest ID: Sundaram_berkeley_0028E_12563. Merritt ID: ark:/13030/m51j9fv6. Retrieved from https://escholarship.org/uc/item/8w9996x7

[4] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason OngGeeHock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays- FPGA ’17, pages 5–14, 2017.

[5] Kalin Ovtcharov, Olatunji Ruwase, Joo-young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. White paper, pages 3–6, 2 2015.

[6] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, andHuazhongYang. Going Deeper with Embedded FPGAPlatform for Convolutional Neural Network. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays- FPGA ’16, pages 26–35, New York, NY, USA, 2016. ACM.

[7] Y. Ma, Y. Cao, S. Vrudhula and J. -S. Seo, "Performance Modeling for CNN Inference Accelerators on FPGA," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 4, pp. 843-856, April 2020, doi: 10.1109/TCAD.2019.2897634.

[8] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17). Association for Computing Machinery, New York, NY, USA, Article 29, 1–6. https://doi.org/10.1145/3061639.3062207.

[9] J. Kwon, J. Lee, and H. Kim, "Pipelining of a Mobile SoC and an External NPU for Accelerating CNN Inference," in IEEE Embedded Systems Letters, doi: 10.1109/LES.2023.3305016.

[10] D. Giri, P. Mantovani and L. P. Carloni, "Accelerators and Coherence: An SoC Perspective," in IEEE Micro, vol. 38, no. 6, pp. 36-45, 1 Nov.-Dec. 2018, doi: 10.1109/MM.2018.2877288.

[11] Han, S.; Pool, J.; Tran, J.; Dally, W. Learning both weights and connections for efficient neural network. In Proceedings of the Advances in Neural Information Processing Systems 28 (NeurIPS 2015), Montreal, QC, Canada, 7–12 December 2015; pp. 1–8.

[12] Wen, G.; Li, M.; Luo, Y.; Shi, C.; Tan, Y. The improved YOLOv8 algorithm based on EMSPConv and SPE-head modules. Multimed. Tools Appl. 2024, 1–17. https://doi.org/10.1007/s11042-023-17957-4.

[13] Wang, Y.; Ha, J.-E. Improved Object Detection with Content and Position Separation in Transformer. Remote Sens. 2024, 16, 353. https://doi.org/10.3390/rs16020353.

[14] Xiong, C.; Zayed, T.; Abdelkader, E.M. A novel YOLOv8-GAM-Wise-IoU model for automated detection of bridge surface cracks. Constr. Build. Mater. 2024, 414, 135025.

[15] Yun, J.; Kang, B.; Rameau, F.; Fu, Z. In Defense of Pure 16-bit Floating-Point Neural Networks. arXiv 2023, arXiv:2305.10947.

[16] Nguyen, D.T.; Nguyen, T.N.; Kim, H.; Lee, H.J. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 2019, 17, 1861–1873.

[17] Nagel, M.; Baalen, M.V.; Blankevoort, T.; Welling, M. Data-free quantization through weight equalization and bias correction. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV 2019), Seoul, Republic of Korea, 27 October–2 November 2019; pp. 1325–1334.

[18] Wu, H.; Judd, P.; Zhang, X.; Isaev, M.; Micikevicius, P. Integer quantization for deep learning inference: Principles and empirical evaluation. arXiv 2020, arXiv:2004.09602.

[19] Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; and Kalenichenko, D. 2018a. Quantization and Training of Neural Networks for Efficient Integer Arithmetic-Only Inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[20] Jouppi, N. P.; Yoon, D. H.; Ashcraft, M.; Gottscho, M.; Jablin, T. B.; Kurian, G.; Laudon, J.; Li, S.; Ma, P. C.; Ma, X.; Norrie, T.; Patil, N.; Prasad, S.; Young, C.; Zhou, Z.; and Patterson, D. A. 2021. Ten Lessons From Three Generations Shaped Google’s TPUv4i: Industrial Product. In ISCA, 114. IEEE.

[21] Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. CoRR, abs/1806.08342, 2018. URL http://arxiv.org/ abs/1806.08342.

[22] Al-Hamid, A.A.; Kim, H. Unified Scaling-Based Pure-Integer Quantization for Low-Power Accelerator of Complex CNNs. Electronics 2023, 12, 2660. https://doi.org/10.3390/electronics12122660

[23] Bhalgat, Y., Lee, J., Nagel, M., Blankevoort, T., and Kwak, N. LSQ+: improving low-bit quantization through learnable offsets and better initialization. CoRR, abs/2004.09576, 2020. URL https://arxiv.org/ abs/2004.09576.

[24] Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. Incremental network quantization: Towards lossless CNNs with low-precision weights. In International Conference on Learning Representations, 2017. URL https://openreview.net/forum?id=HyQJ-mclg.

[25] Cai, Z., He, X., Sun, J., and Vasconcelos, N. Deep learning with low precision by half-wave gaussian quantization. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[26] Fang, J., Shafiee, A., Abdel-Aziz, H., Thorsley, D., Georgiadis, G., and Hassoun, J. Near-lossless post-training quantization of deep neural networks via a piecewise linear approximation. CoRR, abs/2002.00104, 2020. URL https://arxiv.org/abs/2002.00104.

[27] Li, Y., Dong, X., and Wang, W. Additive powers-of two quantization: An efficient non-uniform discretization for neural networks. In International Conference on Learning Representations, 2020. URL https:// openreview.net/forum?id=BkgXT24tDS.

[28] Van Baalen, M., Kuzmin, A., Nair, S. S., Ren, Y., Mahurin, E., Patel, C., Subramanian, S., Lee, S., Nagel, M., Soriaga, J., & Blankevoort, T. (2023). FP8 versus INT8 for efficient deep learning inference. ArXiv. /abs/2303.17951

[29] Wang, Z.; Wu, Z.; Lu, J.; Zhou, J. Bidet: An efficient binarized object detector. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020), Virtual Conference, 14-19 June 2020, pp. 2049-2058.

[30] Zhao, S.; Yue, T.; Hu, X. Distribution-aware adaptive multi-bit quantization. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020), Virtual Conference, 14-19 June 2020, pp. 9281-9290.
[31] Kuzmin, A., Van Baalen, M., Ren, Y., Nagel, M., Peters, J., & Blankevoort, T. (2022). FP8 Quantization: The Power of the Exponent. ArXiv. /abs/2208.09225
[32] Shen, H., Mellempudi, N., He, X., Gao, Q., Wang, C., & Wang, M. (2023). Efficient Post-training Quantization with FP8 Formats. ArXiv. /abs/2309.14592
[33] Zhang, Z., Zhang, Y., Shi, G., Shen, Y., Gong, R., Xia, X., Zhang, Q., Lu, L., & Liu, X. (2023). Exploring the Potential of Flexible 8-bit Format: Design and Algorithm. ArXiv. /abs/2310.13513
[34] Wang, K.; Liu, Z.; Lin, Y.; Lin, J.; Han, S. Haq: Hardware-aware automated quantization with mixed precision. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2019), Long Beach, CA, USA, 16–20 June 2019; pp. 8612–8620.
[35] Zhang, X.; Qin, H.; Ding, Y.; Gong, R.; Yan, Q.; Tao, R.; Li, Y.; Yu, F.; Liu, X. Diversifying sample generation for accurate data-free quantization. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2021), Virtual Conference, 19–25 June 2021; pp. 15658–15667.
[36] He, Kaiming et al. “Deep Residual Learning for Image Recognition.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 770-778.
[37] Spears, W.M., & Jong, K.A. (1990). An Analysis of Multi-Point Crossover. Foundations of Genetic Algorithms.
[38] Junaid, M.; Aliev, H.; Park, S.; Kim, H.; Yoo, H.; Sim, S. Hybrid Precision Floating-Point (HPFP) Selection to Optimize Hardware-Constrained Accelerator for CNN Training. Sensors 2024, 24, 2145. https://doi.org/10.3390/s24072145
[39] Li, Dachong, Li Li, Zhuangzhuang Chen, and Jianqiang Li. "Shift-ConvNets: Small Convolutional Kernel with Large Kernel Effects." ArXiv, (2024). Accessed June 10, 2024. /abs/2401.12736.
[40] Huang, Hongmin, et al. "Design space exploration for yolo neural network accelerator." Electronics 9.11 (2020): 1921.
[41] T. Adiono, A. Putra, N. Sutisna, I. Syafalni and R. Mulyawan, "Low Latency YOLOv3-Tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle," in IEEE Access, vol.
[42] T. Adiono, R. M. Ramadhan, N. Sutisna, I. Syafalni, R. Mulyawan and C. -H. Lin, "Fast and Scalable Multicore YOLOv3-Tiny Accelerator Using Input Stationary Systolic Architecture," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 31, no. 11, pp. 1774-1787, Nov. 2023, doi: 10.1109/TVLSI.2023.3305937.
[43] Park, Sangbo, “Reconfigurable CNN Training Accelerator Design Based on Efficient Memory Access Reduction Techniques,” M.S. thesis, Chungbuk National University, Cheongju, Republic of Korea, 2024.