



Neuromorphic processor-oriented hybrid Q-format multiplication with adaptive quantization for tiny YOLO3

Tao Li^{1,2} · Yitao Ma^{2,3} · Tetsuo Endoh^{1,2,3}

Received: 6 May 2022 / Accepted: 6 January 2023 / Published online: 13 February 2023
© The Author(s) 2023

Abstract

Deep neural networks (DNNs) have delivered unprecedented achievements in the modern Internet of Everything society, encompassing autonomous driving, expert diagnosis, unmanned supermarkets, etc. It continues to be challenging for researchers and engineers to develop a high-performance neuromorphic processor for deployment in edge devices or embedded hardware. DNNs' superpower derives from their enormous and complex network architecture, which is computation-intensive, time-consuming, and energy-heavy. Due to the limited perceptual capacity of humans, accurate processing results from DNNs require a substantial amount of computing time, making them redundant in some applications. Utilizing adaptive quantization technology to compress the DNN model with sufficient accuracy is crucial for facilitating the deployment of neuromorphic processors in emerging edge applications. This study proposes a method to boost the development of neuromorphic processors by conducting fixed-point multiplication in a hybrid Q-format using an adaptive quantization technique on the convolution of tiny YOLO3. In particular, this work integrates the sign-bit check and bit roundoff techniques into the arithmetic of fixed-point multiplications to address overflow and roundoff issues within the convolution's adding and multiplying operations. In addition, a hybrid Q-format multiplication module is developed to assess the proposed method from a hardware perspective. The experimental results prove that the hybrid multiplication with adaptive quantization on the tiny YOLO3's weights and feature maps possesses a lower error rate than alternative fixed-point representation formats while sustaining the same object detection accuracy. Moreover, the fixed-point numbers represented by $Q(6.9)$ have a suboptimal error rate, which can be utilized as an alternative representation form for the tiny YOLO3 algorithm-based neuromorphic processor design. In addition, the 8-bit hybrid Q-format multiplication module exhibits low power consumption and low latency in contrast to benchmark multipliers.

Keywords Hybrid multiplication · Adaptive quantization · DNNs · Fixed-point representation · Neuromorphic processor

1 Introduction

The neuroscientists' efforts to explore the human brain's computational model lay a solid foundation for implementing intelligent perception and detection of electronic devices in the modern Internet of Everything society. In neuroscience, the communication theory of neuronal signals is vital for advancing the mathematical model and very large-scale integration (VLSI) circuit development of complex neural networks. Neurons mainly consist of dendrites, soma, axon hillock, axon, axon terminal, etc. The neuron is responsible for capturing and transferring signals across the entire body, while the synapse acts as the bridge between neuron communication. Specifically, as shown in Fig. 1, if the signal intensity surpasses the axon hillock's

✉ Tao Li
li.tao.e4@tohoku.ac.jp

Yitao Ma
mrmymt@cies.tohoku.ac.jp

Tetsuo Endoh
tetsuo.endoh@cies.tohoku.ac.jp

¹ School of Engineering, Tohoku University, 6-6, Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi 980-8579, Japan

² Center for Innovative Integrated Electronic Systems (CIES), Tohoku University, 468-1, Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi 980-8572, Japan

³ Research Institute of Electrical Communication (RIEC), Tohoku University, 2 Chome-1-1 Katahira, Aoba Ward, Sendai, Miyagi 980-8577, Japan

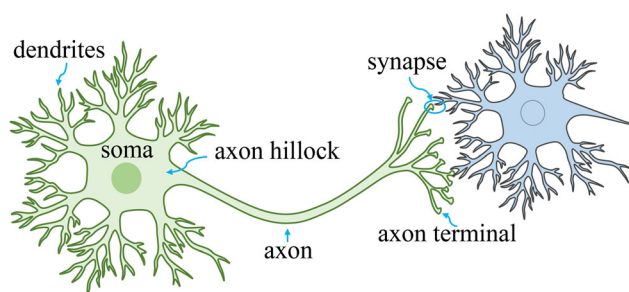


Fig. 1 Diagram of neuron and synapse. Information transfer occurs at the synapse, a junction between the axon terminal of the current neuron and the dendrite of the next neuron. Soma does not engage in the propagation of electrical signals, but it functions as the neuron's driving force to ensure its healthy operation

threshold, the dendrites transform the chemical signals released by another neuron into electrical impulses that are conveyed down the axon to the axon terminal.

Researchers in mathematics and electrical engineering attempt to imitate the functioning of neurons and synapses by employing mathematical models and logic gates based on the information exchange theory from the following two points. (1) Functional emulation: utilizing the electronic components to emulate the neuron or synapse's function rather than its existing architecture, typically represented by the hardware accelerator of convolutional neural networks. (2) Neurobiological mimicry: mimicking the brain's models, such as the Hodgkin–Huxley model and signal transmission of neurons via integrated circuits, which holds excellent promise in imitating human brain learning. The memristor-based spiking neural networks are an emerging research subject in this domain.

Neuromorphic computing, also referred to as brain-inspired computing, is an interdisciplinary field combined with electronic engineering and neuroscience. Neuromorphic computing aims to mimic human beings' brain structures and functions by deploying silicon transistors. Originating in the 1980s, it emulates the biological functions of the human brain using an electrical circuit [1]. Neuromorphic computing is distinguished from conventional computing with von Neumann architecture by its intimate relationship to the structure and parameters of neural networks and its use of advanced neural network models to imitate the processing processes of the human brain [2, 3]. Deep neural networks (DNNs) have become the soul of neuromorphic computing in recent years with the emergence of machine learning. Spiking neural networks, in particular, play a crucial role in propelling neuromorphic computing forward, both in terms of the algorithm (neuron model) [4, 5] and hardware (circuit architecture) [6]. Neuromorphic computing, based on a high-precision neural networks, new semiconductor materials [7, 8], or optimal circuit architecture [9], is the crucial

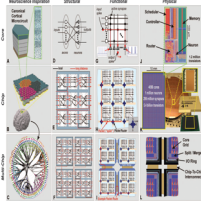
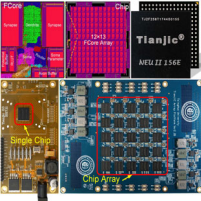
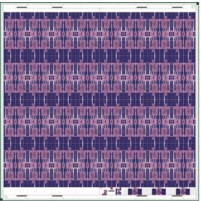

technology for achieving a neuromorphic processor design with low power consumption, high reliability, and low latency for modern industrial society. Conventional neuromorphic computing is constructed using the following assessment criteria and methods:

- *Lightweight* Typically, high-performance DNNs are composed of sophisticated network architecture and a multitude of parameters, which poses substantial hurdles for the neuromorphic processor design with on-chip memory. How to load entire weights into on-chip memory is the key difficulty to be addressed in neuromorphic computing. Current studies investigate lightweight neural networks that exploit parameter compression techniques such as weight/feature map sparsification and quantization [10].
- *Low-latency* Real-time industrial applications (e.g., autonomous driving and unmanned aerial vehicles) require a short response time for the neuromorphic processor; otherwise, it may pose substantial potential safety hazards to human beings. Employing parallel processing technologies [11] and approximate computing [12] can effectively reduce the processor's latency.
- *Energy-efficiency* One of the goals of industrial 5.0 is to decrease carbon dioxide dissipation to prevent the depletion of natural resources. Neuromorphic computing intends to circumvent the von Neumann bottleneck in traditional processors, consuming large amounts of power by shuffling data between memory and processor. Temporal and spatial on-chip memory design [13] and emerging semiconductor materials [14] can efficiently reduce the processor's energy consumption.

As indicated in Table 1, a vast variety of neuromorphic chips, including TrueNorth [15], Tianjic [16], Loihi/Loihi2 [17, 18], Neurogrid [19], etc., have developed in contemporary academia and industry.

TrueNorth incorporates 4096 neuromorphic cores, which include 5.4 billion transistors, to achieve the functionality of the neuromorphic processor. However, the power consumption is only 63 milliwatts for real-time object detection with 400×240 video inputs, which is mainly utilized for inference. Distinct from the TrueNorth chip, Loihi is a neuromorphic processor that combines inference and training functions with 128 neuromorphic cores (14 nm process). With the same processing technology (28 nm process) as TrueNorth, Tianjic is developed as an inference chip with 156 functional cores. Meantime, the rapid evolution of DNNs delivers great opportunities and challenges to neuromorphic processors in a variety of applications such as autonomous driving [20–22], 6 G network communication [23], intelligent medical diagnosis [24] and smart industrial automation [25] etc. Cutting-edge DNNs exhibit superior performance in various applications

Table 1 State-of-the-art neuromorphic processors

Neuromorphic Chips	Layout Images	Specification
TrueNorth (IBM) [15]		<ul style="list-style-type: none"> • 28 nm processing technology • 5.4 billion transistors • 4096 neurosynaptic cores • 1 million neurons • 256 million synapses • 63 milliwatts–(400×240) • Application: object detection (inference)
Tianjic (Tsinghua) [16]		<ul style="list-style-type: none"> • 28 nm processing technology • 156 functional cores • 40000 neurons • 10 million synapses • Power consumption: 0.95 watts • Power efficiency: 1.28 TOPS/W • Throughput: > 1.6·10² times of GPU • Application: inference
Loihi (Intel) [17]		<ul style="list-style-type: none"> • 14 nm processing technology • 2.07 billion transistors • 128 functional cores • 128000 neurons • 128 million synapses • Die area: 60 mm² • Memory: 128 KB • Application: inference and training
Loihi2 (Intel) [18]		<ul style="list-style-type: none"> • Intel 4 processing technology • 2.3 billion transistors • 128 functional cores • 1 million neurons • 120 million synapses • Die area: 31 mm² • Memory: 192 KB • Application: inference and training

by expanding the number of layers or deploying complex network architectures. Despite their higher performance, DNNs pose significant challenges for embedded hardware development in mobile and edge applications due to their high compute complexity, high energy consumption, and massive memory demands. Furthermore, the accuracy of DNNs tends to be redundant in practical applications since human capability for error-prone perception is limited. Therefore, compression of DNNs is imperative to facilitate the deployment of the neuromorphic processor in today's highly intelligent society.

The algorithm improvement and hardware approximation can accomplish DNNs compression. The essence of DNNs compression is to take advantage of approximate weights or feature maps, approximate arithmetic [26] or approximate circuit [27–29] to realize convolution operations. The weight or feature map sparsification intends to eliminate the redundant weights or feature maps that

contribute little to the accuracy of DNNs. The typical sparsification approach is weight or feature map pruning, which can significantly diminish weights or feature maps. The weight pruning aims to remove the redundant weights while feature map pruning decreases both feature maps and weights [30]. Han et al. introduced the deep compression to the DNNs by deploying the connection pruning approach, which demonstrates that the connections can be reduced by $9\times - 13\times$ [31]. Recently, other pruning techniques have been proposed such as random pruning [32, 33], channel pruning [34, 35] etc. However, it is essential to retrain the neural network after removing the unnecessary weights or feature maps with pruning method, which raises design challenges for neuromorphic processors. Single vector decomposition (SVD) of weights or feature maps, as another sparsification strategy, discards the weights or feature maps with small eigenvalues. Specially, the weight or feature map matrix is decomposed as the multiplications

with two unitary matrices (left single vector and right single vector) and one diagonal vector that determines the weight or feature map matrix's eigenvalues. The corresponding weights or feature maps will be removed if the eigenvalues are smaller than a pre-defined threshold [36]. Since the SVD algorithm prefers large matrices such as the matrix in fully-connection layers, it has poor performance for object detection with tiny YOLO3. Another effective DNNs compression approach, named knowledge distillation, is to refine a compact student model from a complex teacher model [37]. The knowledge distillation consists of score-based [38] and probability-based [39] distillation according to the different loss function definition. The student model generally presents equal or even better performance than the teacher model if the gap between them is small enough [40, 41]. Although knowledge distillation is promising in DNNs compression, it is still challenging to derive an effective student model from an intricate teacher model.

Weight or feature map quantization, as an alternative approach of DNNs compression, is assumed to be the most promising technique in the area of neuromorphic design owing to the following benefits,

- The quantization technique efficiently shortens the bit length of weights or feature maps, allowing it to utilize fewer logic gates to fulfill the arithmetic operation.
- The lack of sufficient layout space for on-chip memory is a major design bottleneck for neuromorphic processors based on tiny YOLO3. Deploying a short-bit representation format can lessen the memory requirement for the pre-trained weights' storage, which in turn reduces neuromorphic processors' power consumption due to less external memory access.

Quantization can be achieved by the following two perspectives: (1) training the DNNs using quantized weights or feature maps, which is generally utilized for both the training and inference stages; (2) offline quantization of weights or feature maps, which mainly contributes to the inference stage. Many studies on the topic of training with low-precision bits have been published [42–45]. Our work concentrates primarily on the inference of tiny YOLO3 with low-precision fixed-point representation format (second category) since retraining the DNNs model is time- and power-intensive. In the literature of [46], the authors proposed an optimization algorithm based on quantization errors for determining the bit length of feature maps in each layer. The experimental results indicate that it is up to 20–40% compression rate without sacrificing accuracy. Similarly, Zhu et al. deployed an adaptive quantization technique to the DNNs, which demonstrated that the model

size and computation cost are decreased using CIFAR10 and ImageNet2012 datasets [47]. An adaptive quantization technique proposed by Kwon et al. was applied to transfer the trained weights to the synaptic devices, attaining up to 98.09% accuracy rate [48]. The weights or feature maps can be quantized from 32-bit floating-point numbers to 16-bit, 8-bit, 4-bit, 2-bit, and even 1-bit fixed-point representation formats [49–52]. However, as the bit length of fixed-point representations decreases, the accuracy of neural networks drops, making it challenging to develop a high-performance neuromorphic processor with a long bit length representation. Therefore, exploring the low-bit representation of weights and feature maps while sustaining the algorithm's precision is meaningful for neuromorphic processor design. Since the representation range and resolution of fixed-point numbers are constrained by the length of integer bits and fraction bits, the correct bit length of weights or feature maps is crucial for determining whether a fixed-point number can accurately represent a floating-point value. The accuracy degrades when assigning the same bit length to the entire DNNs. This article proposes a neuromorphic processor-oriented hybrid multiplication with adaptive quantization for tiny YOLO3, and illustrates the addition and multiplication between two 16-bit fixed-point values for tiny YOLO3's convolution operation. Generally, using approximated fixed-point numbers to perform convolution often results in overflow problems and roundoff issues in some arithmetic operations, producing erroneous convolution results. Moreover, since the inputs of the convolutional layer in tiny YOLO3 are the previous layers' outputs except the first layer, it will introduce numerous errors to the entire neural network if the fixed-point numbers cannot correctly approximate the weights or feature maps. The proposed hybrid multiplication can effectively alleviate the overflow errors caused by addition or roundoff errors introduced by multiplication operations using approximated fixed-point weights and feature maps. In brief, the contributions of this study are briefly summarized as follows.

- This paper thoroughly illustrates the addition of a 16-bit fixed-point with adaptive quantization. The proposed sign-bit check approach can effectively reduce the overflow issues accompanied by the addition operation.
- An optimal strategy of bit length adjustment is proposed to mitigate the roundoff errors in this article. Because the bit length of multiplying two fixed-point numbers is more than 16 bits, an appropriate bit length adjustment can adequately ensure the validity of approximation results.
- An optimal and suboptimal representation formats of 16-bit fixed-point numbers has been attained for

neuromorphic processor design by investigating the conversion error rate of data (feature maps and weights) and the accumulated calculation error of convolution.

- A hybrid multiplication module is presented to assess the hardware cost of the adaptive quantization technique, and the experimental results prove that the proposed multiplication module has low power consumption and low latency in comparison with the benchmark multipliers.

The remainder of this paper is organized as follows. Section 2 offers the preliminaries of tiny YOLO3’s convolution operation. The details of the proposed hybrid multiplication with adaptive quantization are illustrated in Sect. 3 which includes an adaptive quantization algorithm, binary addition with sign-bit check, and binary multiplication with bit roundoff methods. Section 4 describes the experimental results and discussion regarding the hybrid multiplication with adaptive quantization, and the conclusion is presented in Sect. 5.

2 Convolution of tiny YOLO3

As shown in Fig. 2, the DNNs’ convolution is calculated by multiplying between intra-channel elements of weights and feature maps with inter-channel elements, and accumulating the results along the depth direction.

Specifically, the following expression defines the convolution (C) between weights (W) and feature maps (F), $c_i = \sum_{i=1}^N w_i \times f_i$, where $c_i \in C$ is the convolution result. $w_i \in W$ and $f_i \in F$ are the weights and feature maps in each intra-channel, respectively. From the layers, intra-channels, inter-channels, and depths of DNNs, it is possible to perform convolution operations with quantized weights and feature maps. Two main steps implement the convolution operation of tiny YOLO3: (1) feature matrix conversion (FMC) and (2) general matrix multiplication (GEMM). Concretely, the FMC converts the inputs to feature maps based on the window dimension of filters, and the convolution is achieved via an element-by-element multiply-accumulate operation (MAC) between the weights and feature maps. The tiny YOLO3 has 13 convolution layers and two types of filters (1×1 and 3×3 kernel size). It is required to convert inputs into feature maps using FMC for the filters with 3×3 kernels, while it does not need to convert input vectors for the filters with 1×1 kernel. As shown in Table 2, the eighth, tenth, eleventh, and thirteenth layers deploy filters with 1×1 kernel while other layers use filters with 3×3 kernel.

Since tiny YOLO3 employs one stride zero-padding, the width and height for the inputs and outputs of convolution are identical. The dimensions of inputs and filters

determine the dimension of output feature maps. Assuming that the dimensions of inputs and filters are $w \times h \times d$ (width, height and depth of input) and $f_w \times f_h \times f_d$ (width, height and depth of filter), respectively, the dimension of output FMC can be calculated by $d_{fmc} = w \times h \times d \times f_w \times f_h$, where d_{fmc} is the dimension of FMC outputs. The depth of the feature map and the weight should be the same in order for the convolution operation to be implemented. Table 2 concisely summarizes the dimension of output feature maps in each convolution layer of tiny YOLO3, revealing that the maximum amount of data in feature maps is over 6 million (layer 2). A total of 23.765625 Megabytes memory is required if the 32-bit floating-point format represents these feature maps. However, the memory utilization will be halved if the 16-bit fixed-point format represent these feature maps. The fixed-point numbers are represented by Q-format, which is denoted by $Q(L_{FI} \cdot L_{FR})$ or $Q(L_{FR})$. The symbol “.” indicates the radix point. L_{FI} and L_{FR} are the integer and fraction bit lengths of fixed-point representation, respectively.

The implementation of GEMM includes two steps: (1) element-by-element multiplication (MUL); (2) summation of the multiplication result (ADD). As illustrated in Fig. 3, each element in the filter is multiplied by each element in

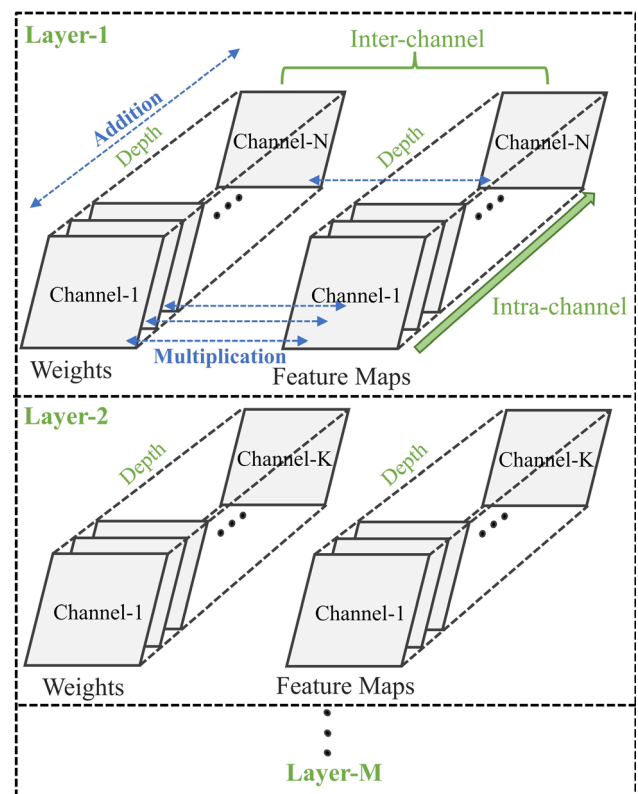


Fig. 2 Principle of convolution operation between weights and feature maps, and definitions of DNNs’ parameters

the first row of the feature map, and the result is stored in memory.

The second element of the filter is then multiplied by each element in the second row of the feature map, and the product is summed to each element in memory. The calculation process continues until the filter’s last element is multiplied by every element in the last row of the feature map. Then, the addition operation is conducted using the result of the previous summation. The convolution process between the first filter and the feature maps is now complete. In general, the filters of tiny YOLO3 is a 4-dimension vectors ($M \times w \times h \times d$), with each filter dimension specified as f_n which equals $w \times h \times d$. According to the principle of matrix multiplication, the output dimension of GEMM is $M \times f_k$ (refer to Fig. 3) where $f_k = f_w \times f_h$. Table 3 presents that the GEMM of tiny YOLO3 involves 8841794 16-bit fixed-point addition operations and 2782480896 16-bit fixed-point multiplication operations, which is the bottleneck for real-time object detection.

A simple way to convert a floating-point number to a fixed-point number is to multiply the floating-point number by the scaling factor, which is calculated by $X_{\text{fixed}} = \text{INT}(X_{\text{float}} \times 2^{LFR})$, where $\text{INT}(\bullet)$ indicates the function of rounding calculation result to the nearest integer number. X_{float} and X_{fixed} are the floating-point and fixed-point numbers, respectively. As an instance, the fixed-point number of -2.89037 can be derived by $-2.89037_{Q(2.13)} = \text{INT}(-2.89037_{\text{float}} \times 2^{13}) \approx -23678$. Therefore, the floating-point number -2.89037 can be represented by the binary: 1010001110000010_2 . “Appendix” provides the pseudo codes for the format conversion between floating-point, fixed-point formats and their corresponding binary representations.

3 Hybrid Q-format multiplication with adaptive quantization proposal

3.1 Adaptive quantization for tiny YOLO3

As shown in Fig. 4, suppose different fixed-point representation formats are employed among DNNs’ layers while designing a multi-layer based neuromorphic processor.

In this instance, it is vital to independently control the different arithmetic logic units (ALUs) in each layer. Moreover, since the ALUs output in the previous layer is the current layer’s input, a bit post-processing circuit of the feature maps is required to ensure that the two layers’ data representation formats are consistent. The M ALU modules shown in Fig. 4 share the control signal, and each module can be directly attached without the bit post-processing circuit if each layer and channel of weights and feature maps adopt an adaptive fixed-point representation format. Hence, in order to tackle the aforementioned challenging issues, this paper proposes a fully adaptive quantization proposal to improve the neuromorphic processor design. Typically, the following inequality equation is used to limit the range of integer bit length (L_{FI}) for fixed-point values,

$$L_{FI} \geq \begin{cases} \log_2 \left(\frac{X_{\text{fixed}}}{1 - 2^{1-L_b}} \right) & \text{if } X_{\text{fixed}} > 0 \\ \log_2 (-X_{\text{fixed}}) & \text{if } X_{\text{fixed}} < 0 \end{cases} \quad (1)$$

where L_b is the total bit length of a fixed-point representation, which is defined by,

$$L_b = \underbrace{L_{FI}}_{\text{integer bits}} + \underbrace{L_{FR}}_{\text{fraction bits}} + \underbrace{1}_{\text{sign bit}} \quad (2)$$

Eq. (1) constrains the length of an integer for positive and

Table 2 Dimension of feature maps in different convolution layers

Convolution	Inputs	Filters	Dimension of feature maps
Layer 1	416×416×3	3×3×3	416×416×3×3×3 (4672512)
Layer 2	208×208×16	3×3×16	208×208×3×3×16 (6230016)
Layer 3	104×104×32	3×3×32	104×104×3×3×32 (3115008)
Layer 4	52×52×64	3×3×64	52×52×3×3×64 (1557504)
Layer 5	26×26×128	3×3×128	26×26×3×3×128 (778752)
Layer 6	13×13×256	3×3×256	13×13×3×3×256 (389376)
Layer 7	13×13×512	3×3×512	13×13×3×3×512 (778752)
Layer 8	13×13×1024	1×1×1024	13×13×1×1×1024 (173056)
Layer 9	13×13×256	3×3×256	13×13×3×3×256 (389376)
Layer 10	13×13×512	1×1×512	13×13×1×1×512 (86528)
Layer 11	13×13×256	1×1×256	13×13×1×1×256 (43264)
Layer 12	26×26×384	3×3×384	26×26×3×3×384 (2336256)
Layer 13	26×26×256	1×1×256	26×26×1×1×256 (173056)

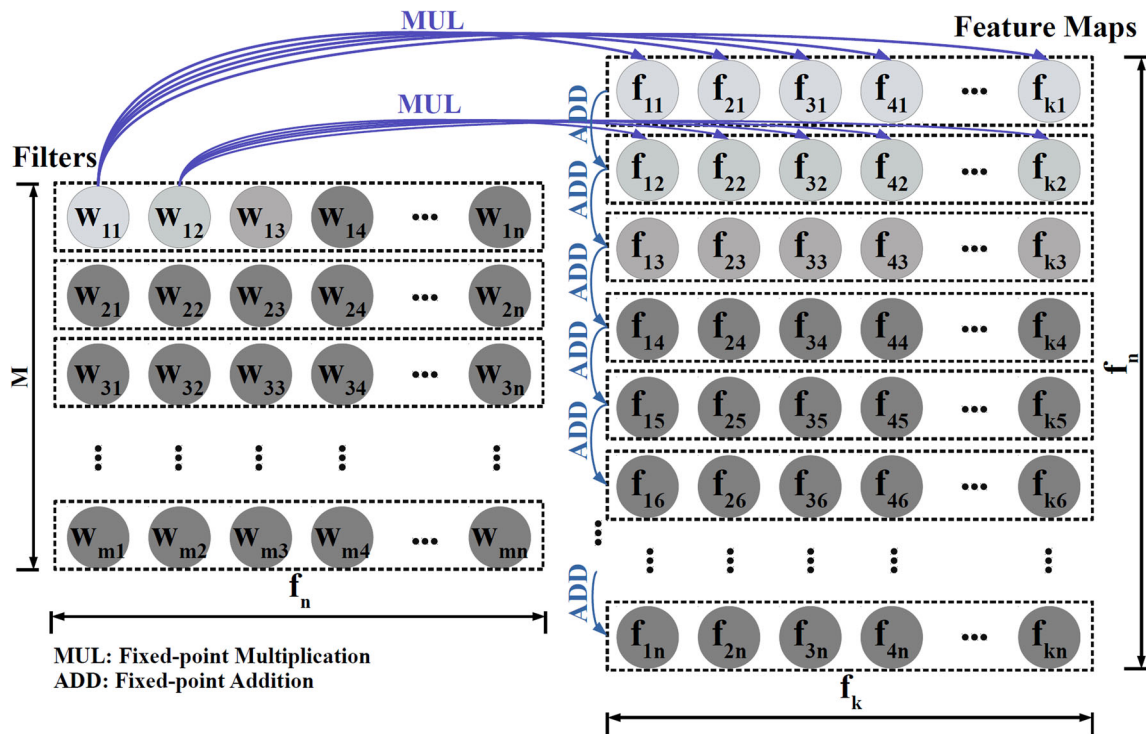


Fig. 3 Convolution operation with GEMM. The output dimension of GEMM is determined by the number of filters and the width of feature maps, $M \times f_k \leftarrow M \times f_n \times f_n \times f_k$

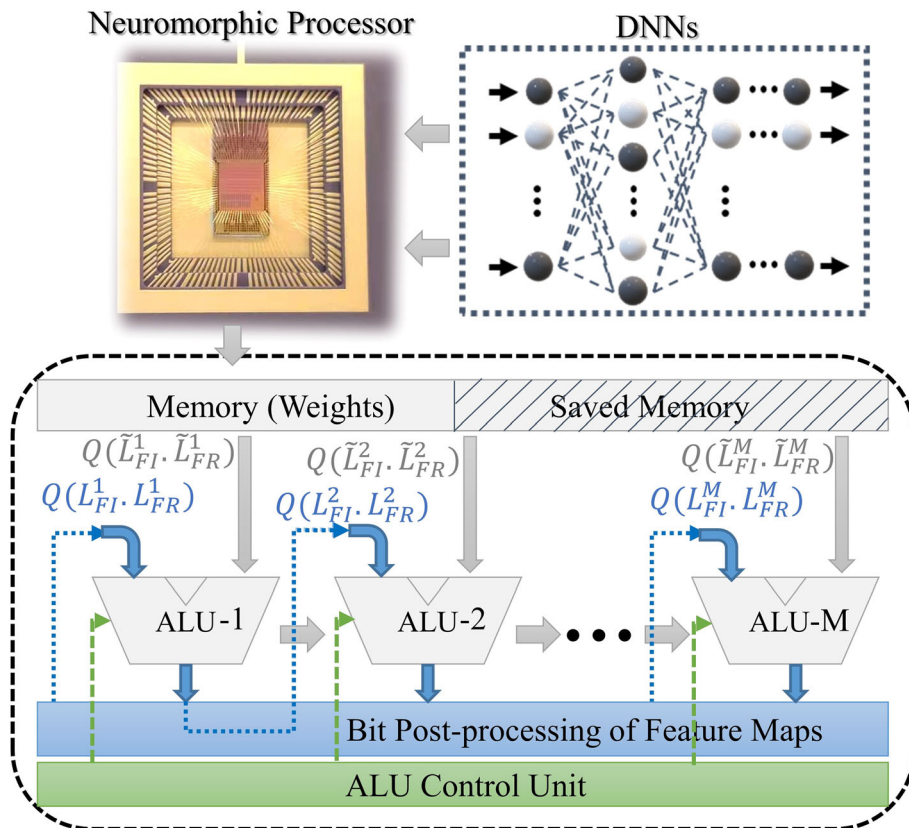
Table 3 Number of additions and multiplications for tiny YOLO3’s GEMM

Convolution	Filters	Addition	Multiplication
Layer 1	16	$16 \times (3 \times 3 \times 3 - 1)$	$16 \times 416 \times 416 \times 3 \times 3 \times 3$
Layer 2	32	$32 \times (3 \times 3 \times 16 - 1)$	$32 \times 208 \times 208 \times 3 \times 3 \times 16$
Layer 3	64	$64 \times (3 \times 3 \times 32 - 1)$	$64 \times 104 \times 104 \times 3 \times 3 \times 32$
Layer 4	128	$128 \times (3 \times 3 \times 64 - 1)$	$128 \times 52 \times 52 \times 3 \times 3 \times 64$
Layer 5	256	$256 \times (3 \times 3 \times 128 - 1)$	$256 \times 26 \times 26 \times 3 \times 3 \times 128$
Layer 6	512	$512 \times (3 \times 3 \times 256 - 1)$	$512 \times 13 \times 13 \times 3 \times 3 \times 256$
Layer 7	1024	$1024 \times (3 \times 3 \times 512 - 1)$	$1024 \times 13 \times 13 \times 3 \times 3 \times 512$
Layer 8	256	$256 \times (1 \times 1 \times 1024 - 1)$	$256 \times 13 \times 13 \times 1 \times 1 \times 1024$
Layer 9	512	$512 \times (3 \times 3 \times 256 - 1)$	$512 \times 13 \times 13 \times 3 \times 3 \times 256$
Layer 10	255	$255 \times (1 \times 1 \times 512 - 1)$	$255 \times 13 \times 13 \times 1 \times 1 \times 512$
Layer 11	128	$128 \times (1 \times 1 \times 256 - 1)$	$128 \times 13 \times 13 \times 1 \times 1 \times 256$
Layer 12	256	$256 \times (3 \times 3 \times 384 - 1)$	$256 \times 26 \times 26 \times 3 \times 3 \times 384$
Layer 13	255	$255 \times (1 \times 1 \times 256 - 1)$	$255 \times 26 \times 26 \times 1 \times 1 \times 256$
Total	3694	8841794	2782480896

negative numbers, respectively. However, the above inequality equation will be trivial if the result of the logarithm operation is less than or equal to -1 . As an illustration, the constrain of L_{FI} becomes $L_{FI} \geq 8$ when X_{fixed} equals 0.00390625. It can be observed from the above inequality equations that the output limit of the logarithmic operation that makes the expression meaningful is -1

because the length of integer bits should be equal or greater than 0 ($L_{FI} \geq 0$). This paper introduces an adaptive quantization (ADQ) method that flexibly determines the integer and fraction bits’ length to solve this issue. The bits length of integer in fixed-point number X_{fixed} is defined by the following equation,

Fig. 4 Challenges of neuromorphic processor’s design with traditional quantization method. The quantization can shorten the bit length of weights by a few bits, thereby decreasing the memory utilization



$$L_{FI} = \begin{cases} \text{floor} \left[\log_2 \left(\frac{X_{\text{fixed}}}{2^{-1} - 2^{1-L_b}} \right) \right] & \text{if } X_{\text{fixed}} \geq \sigma \\ \text{floor} \left[\log_2 (-X_{\text{fixed}}) \right] + 1 & \text{if } X_{\text{fixed}} \leq -0.5 \\ 0 & \text{others} \end{cases} \tag{3}$$

where $\sigma = 2^{-1} - 2^{1-L_b}$. Since the dynamic range of L_b satisfies $L_b \geq 2$, it can be deduced that $\sigma \geq 0$. The symbol $\text{floor}[\bullet]$ represents the floor function that provides the largest integer less than or equal to the input.

3.2 Binary addition with sign-bit check

When adding two fixed-point integers, the fundamental idea is to ensure that the addend’s radix point aligns with that of the augend. An incorrect alignment between addend and augend will result in an erroneous calculation. Aligning two binaries with varying L_{FI} will produce different bit lengths for both the integer and fraction component of the addend and augend. A sign extension method can be applied to circumvent the issue of inconsistent bit length between addend and augend. The specific implementation of sign extension is to add “1” before the sign bit of a negative number and “0” before the positive number sign bit. As negative numbers are stored in memory in the form of two’s complement, the sign extension will not change

the negative numbers. Similarly, extending “0” before a positive number has no impact on its value. It should be noted that the carrier generated in front of the sign bit will have an impact on the addition results between addend and augend. Suppose that two fixed-point numbers represented by $Q(L_{FI}^{(a)} \cdot L_{FR}^{(a)})$ and $Q(L_{FI}^{(b)} \cdot L_{FR}^{(b)})$ are added, and the length of the integer and fraction bits of the addition result are determined by the maximum integer and fraction bit length of addend and augend,

$$L_{FI}^{(c)} = \max(L_{FI}^{(a)}, L_{FI}^{(b)}) \tag{4}$$

$$L_{FR}^{(c)} = \max(L_{FR}^{(a)}, L_{FR}^{(b)}) \tag{5}$$

where $\max(\bullet)$ is the function that searches for the maximum value from its elements. $L_{FI}^{(c)}$ and $L_{FR}^{(c)}$ are the bits length of integer and fraction part attained from addition operation. For instance, the addition result between two 16-bit fixed-point numbers represented by $Q(0.15)$, $Q(2.13)$ will be expressed as $Q(2.15)$ format. Since the length of 18 bits $Q(2.15)$ is inconsistent with that of 16 bits, the least significant two bits are usually discarded, and the addition result is practically represented by the $Q(2.13)$ format.

The issue of bit overflow frequently occurs in binary addition, leading to inaccurate representations of addition results. The carryout of sign bits (most significant bit:

MSB) is closely associated with the location of the radix point. In other words, retaining or discarding the overflow bit modifies the length of integer and fraction bits in fixed-point numbers. As we know, adding two numbers with different sign bits will not induce the overflow problem. Therefore, the first step in judging whether overflow occurs in the addition of two fixed-point numbers is to determine whether the two numbers' sign bits are consistent. Generally, overflow happens when the sign bit of two numbers is the same but the sign bit of the addend or augend differs from the MSB of the addition result. As shown in Fig. 5a, we propose a sign-bit check approach to solve the overflow issue.

In this case, the overflow bit is added before the MSB of the addition result, and the value of the overflow bit is consistent with the sign bit of the addend or augend. It is essential to increase the L_{FI} since an extra bit is added

before the radix point, $L_{FI} = L_{FI} + 1$. The pseudo-codes for the overflow check can be found in Algorithm 1. However, the overflow bits can be discarded directly if the addend or augend sign is the same as the MSB of the addition result. Because the fixed-point numbers are stored in the two's complement format, it is unnecessary to keep all the sign bit before the radix point. The L_{FI} also has a close relationship with the number of the discarded sign bit. As shown in Fig. 5b, two additional sign bits can be discarded, and the following 16-bit binaries can be preserved to represent the addition result, which can effectively enhance the representation solution. In this case, only one bit needs to be reserved, and the other two bits can be removed, and the L_{FI} is zero, $L_{FI} = L_{FI} - 2$. Algorithm 1 shows the pseudo-codes of binary addition with overflow check technique.

Algorithm 1: Addition between Two Binaries with Overflow Check

Inputs: Addend (add), augend (aug), bit length (L_b)

Outputs: Addition result (A_{bin}), carryout flag (C_f)

Function `binaryAddition`(add, aug, L_b):

$A_{bin} \leftarrow 0$ // array for addend

$B_{bin} \leftarrow 0$ // array for augend

for $i < L_b$ **do**

[$A_{bin}[i] = add[i]$

[$B_{bin}[i] = aug[i]$

$S_f = A_{bin}[L_b - 1] \wedge B_{bin}[L_b - 1]$

$S_{add} = A_{bin}[L_b - 1]$

while $B_{bin} \neq 0$ **do**

[$C_{bin} = (A_{bin} \& B_{bin}) \ll 1$

[$A_{bin} = A_{bin} \wedge B_{bin}$

[$B_{bin} = C_{bin}$

$C_f = 0$

$S_{diff} = S_{add} \wedge A_{bin}[L_b - 1]$

if $S_f == 0 \& S_{diff} == 1$ **then**

[$E_{bin} \leftarrow 0$

for $i < L_b$ **do**

[$E_{bin}[i] = A_{bin}[i]$

[$E_{bin}[L_b] = S_{add}$

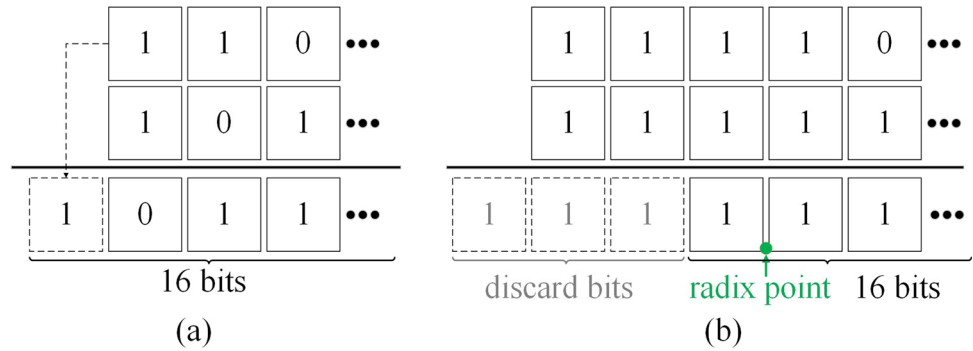
for $j < L_b$ **do**

[$A_{bin}[j] = E_{bin}[j + 1]$

[$C_f = 1$

return A_{bin}, C_f

Fig. 5 Example of sign-bit check and bit selection. **a** Addition with overflow; **b** 16-bit binaries selection



	SE	SE		RP																
	↓	↓		↓																
addend	1	1	1	.	0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	
augend	1	0	1	.	0	0	0	1	1	1	0	0	0	0	0	1	0	-	-	
sum	1	1	0	0	.	0	1	0	1	1	1	0	0	1	1	1	0	0	0	1
	↑																			
	OV																			

Fig. 6 Addition example of fixed-point numbers with adaptive quantization. The abbreviations “SE”, “RP”, and “OV” represent the sign extension bit, radix point, and overflow bit, respectively

In brief, the addition of two 32-bit floating-point numbers can be accomplished using the procedures below. (1) Quantized the 32-bit floating-point numbers to the 16-bit fixed-point numbers and align the radix point between addend and augend; (2) Extend the sign bit for the number with a small L_{FI} and fill zeros to the empty bits for the number with large L_{FI} ; (3) Implement the bit-to-bit addition between addend and augend. (4) Overflow check and integer or fraction bit length adjustment for the addition result. Appendix provides the pseudo codes of binary addition for fixed-point representations. Figure 6 illustrates the addition example of floating-point numbers $(-0.746783, -2.89037)$ implemented by adaptive quantization method.

According to the approach mentioned above, the 32-bit floating-point numbers are quantized as the following 16-bit fixed-point numbers: $-0.746783 \xrightarrow{Q(0.15)} 1010000001101001$, and $-2.89037 \xrightarrow{Q(2.13)} 1010001110000010$. The scaling factors of $Q(0.15)$ and $Q(2.13)$ are 2^{15} and 2^{13} representation formats, respectively. Therefore, the above binary numbers are evaluated by shifting the radix point 15 bits and 13 bits, respectively, toward the left from the rightmost: $1010000001101001 \xrightarrow{\ll 15} 1.010000001101001$, and $1010001110000010 \xrightarrow{\ll 13} 101.0001110000010$. Based on Eqs. (4), and (5), the addition result of the above fixed-point

numbers can be described by $Q(2.15)$, which produces an 18-bit fixed-point number. Hence, extending the bits for the addend and augend of the MSB or least significant bit (LSB) is needed. The sign extension technique is generally utilized to fill the sign bit to the MSB while zeros are usually added to the LSB (or leave the LSB empty).

According to the overflow check principle mentioned above, the overflow bit can be discarded since the sign bit of the addition result is consistent with the addend and augend. As shown in Fig. 6, the addition result is expressed as 100.010111001110001 if the overflow bit is discarded. In summary, the addition result is represented with $Q(2.13)$ format as 1000101110011100 (scaling factor: 2^{13}), reserving 16-bit data length (the last two bits are discarded). The following equation can be employed to verify the correctness of the addition result, $-0.746783 + (-2.89037) = -3.637153 \xrightarrow{Q(2.13)} -29796 = 1000101110011100_2$. This binary number can be converted to the fixed-point number by dividing the scaling factor (2^{13}), $1000101110011100 \xrightarrow{\ll 13} 100.0101110011100_2$, where the decimal binary can be represented by the fixed-point number, $100.0101110011100_2 = -3.63720703125 \approx -3.637153$.

3.3 Binary multiplication with bit roundoff

In contrast to addition, multiplication does not need the alignment of the radix point; rather, the proper use of sign extension is vital to fixed-point multiplication. In addition, identifying the sign bit of the product is a crucial step in establishing the accuracy of fixed-point multiplication. There is no difference between binary multiplication and decimal multiplication except for the sign bit multiplication (MSB). The two’s complement format represents the partial product for the sign bit multiplication if the multiplier sign is “1”. In other words, the partial product is represented by the two’s complement format if the multiplier is negative. Firstly, the binary multiplication calculates the partial product, followed by the addition of all the partial products to determine the final product. The integer and fraction bit length of multiplication results between $Q(L_{FI}^{(a)} \cdot L_{FR}^{(a)})$ and $Q(L_{FI}^{(b)} \cdot L_{FR}^{(b)})$ are determined by $L_{FI}^{(d)} = L_{FI}^{(a)} + L_{FI}^{(b)} + 1$ and $L_{FR}^{(d)} = L_{FR}^{(a)} + L_{FR}^{(b)}$, where $L_{FI}^{(d)}$ and $L_{FR}^{(d)}$ are the integer and fraction bits length of multiplication results. Evidently, the product of $Q(L_{FI}^{(d)} \cdot L_{FR}^{(d)})$ is represented by N_{mul} bits,

$$N_{mul} = \underbrace{L_{FI}^{(a)} + L_{FI}^{(b)}}_{\text{integer bits}} + \underbrace{L_{FR}^{(a)} + L_{FR}^{(b)}}_{\text{fraction bits}} + \underbrace{2}_{\text{sign bits}} \tag{6}$$

In brief, the length of the product has three parts: the number of integer bits, fraction bits, and sign bits. Given that each signed fixed-point number has a sign bit, the last component of the above equation includes two sign bits. In practice, the multiplication between two N -bit numbers generates a $2N - 1$ bits product. However, the Eq. 6 indicates that the bit number of products between two N -bit fixed-point values are $2N$ where two identical sign bit (with 1-bit sign extension) are included. Consequently, it is necessary to discard a sign bit by shifting the radix point one bit to the left.

Since the bit length of fixed-point multiplication between two N -bit fixed-point binaries is $2N$, roundoff error inevitably governs the accuracy of the product. Therefore, it is vital to retain the significant bits and discard the non-dominant bits to increase the accuracy of convolution operations. To reduce the impact of roundoff errors on object detection, we propose the bit roundoff approach to discover the optimal bit sequence for a product. The bit roundoff strategy aims to enhance the opportunity of selecting more significant bits and removing redundant sign bits. It is worth mentioning that the bit roundoff is also associated with the L_{FI} on account of the position change of the sign bit during the bit selection. The L_{FI} with bit roundoff is defined as $L'_{FI} = L_{FI} - N_{discard}$, where L'_{FI} is the updated integer bit length with bit roundoff method, and $N_{discard}$ is the number of discarded bits during bit roundoff calculation. As shown in Fig. 7, the bit length of the product between two 4-bit binaries is 8-bit.

Different locations of the multiplier or multiplicand’s radix point result in the selection of distinct binary sequences. If the radix point of the multiplicand is fixed in the middle of the binary sequence ($Q(1.2)$) and the radix point of the multiplier is adjusted to $Q(1.2)$, $Q(2.1)$, and $Q(3.0)$ in turn, different product sequences will be obtained, $0100_{Q(0.3)} \leftarrow 1100_{Q(1.2)} \times 1111_{Q(1.2)}$, $0100_{Q(0.3)} \leftarrow 1100_{Q(1.2)} \times 1111_{Q(2.1)}$, and $0100_{Q(1.2)} \leftarrow 1100_{Q(1.2)} \times 1111_{Q(3.0)}$. As introduced before, the $L_{FI} = 3, 4, 5$ for the multiplication between $Q(1.2)$ and $Q(1.2)$, $Q(1.2)$ and $Q(2.1)$, and $Q(1.2)$ and $Q(3.0)$, respectively. As shown in Fig. 7, the $N_{discard}$ is 3, 4, and 4 for each computation, accordingly. Therefore, the products can be represented by $Q(0.3)$, $Q(0.3)$ and $Q(1.2)$ representation formats if 4-bit memory is available to store the product. It is necessary to fill the product with zeros starting from the last bit if there are insufficient binaries to represent the product result due to sign bit discard. As an illustration, 1-bit with zero should be filled in the last bit of the product if 5-bit memory is available. The details of fixed-point multiplication will be explained using the same numbers as fixed-point number addition ($-0.746783 \times -2.89037$). For example, the multiplication between 1010000001101001 and 1010001110000010 generates a 31-bit binary sequence; hence, all partial products will be extended to 32 bits using sign extension bits. Algorithm 2 shows the pseudo codes of bit roundoff for the multiplication of fixed-point numbers.

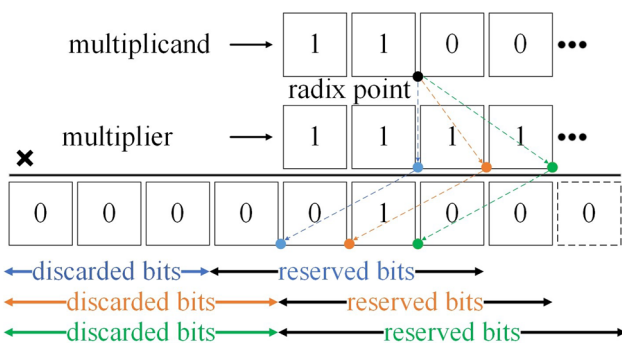


Fig. 7 Bits selection with bit roundoff strategy

Algorithm 2: Bit roundoff for the fixed-point multiplication

```

Inputs: Product ( $P_{fixed}$ ), bit length ( $L_b$ )
Outputs: Number of discarded bits ( $N_{discard}$ )
Function getSignbit( $P_{fixed}, L_b$ ):
     $I_{sign} = 0$  // number of sign bits
     $N_{discard} = 0$  // initialize  $N_{discard}$ 
    while  $I_{sign} < L_b$  do
        if  $P_{fixed}[I_{sign} + 1] - P_{fixed}[I_{sign}] == 0$  then
             $N_{discard}++ = 1$ 
             $I_{sign}++$ 
        else
            break
    return  $N_{discard}$ 

Inputs: Product ( $P_{fixed}$ ), integer bit length ( $L_{FI}$ ), bit length ( $L_b$ )
Outputs: Product with bit roundoff ( $P_{fixed}^r$ )
Function bitRoundoff( $P_{fixed}, L_{FI}, L_b$ ):
     $N_{discard} = getSignbit(P_{fixed}, L_{FI})$ 
     $L_{FI}^r = L_{FI} - N_{discard}$ 
    if  $N_{discard} > L_b$  then
        for  $n < L_b$  do
            if  $n < N_{discard} - L_b$  then
                 $P_{fixed}^r = P_{fixed}[n + L_b - N_{discard}]$ 
            else
                for  $n < L_b$  do
                     $P_{fixed}^r = P_{fixed}[n + L_b - N_{discard}]$ 
    return  $P_{fixed}^r$ 

```

The details of fixed-point multiplication will be explained using the same integers ($-0.746783 \times -2.89037$) as an example. As shown in Table 4, since the multiplication between 101000001101001 and 1010001110000010 generates a 31-bit binary sequence, all partial products are extended to 32 bits using sign extension bits.

It is worth mentioning that the two’s complement format represents the partial product for the row of $16 \times$ because the multiplier is negative. In other words, the product between the MSB of the multiplier and each bit of multiplicand is converted to two’s complement format, $101000001101001 \xrightarrow[\text{complementary}]{\text{two's}} 0101111110010111$.

The leftmost two bits of the product are sign bits, and the residual sign bit (leftmost bit) can be eliminated by left-shifting the product one bit. Therefore, the multiplication result between 101000001101001 and 010111110010111 is 01000101000100101010000010100100. The multiplication between $Q(0.15)$ and $Q(2.13)$ can be represented by $Q(3.28)$. On account of the product’s left shift,

the L_{FR} gains an extra bit while the L_{FI} losses one bit. Therefore, the multiplication result can be expressed by $Q(2.29)$. The rightmost 16 bits can be omitted if there is no overflow among the addition of partial product, and the multiplication result is 0100010100010010 $_{Q(2.13)}$.

In summary, the multiplication of fixed-point numbers can be accomplished by the following steps: (1) Fill the empty bits of partial product with zeros. Since the bit position of the valid partial product begins from the corresponding multiplier position, it is required to fill the partial product’s empty bits with zeros; (2) Calculate the partial product using the “and” bitwise operation and extend the sign bit; (3) Convert the binary representation to the format of two’s complement. If the multiplier is negative, the partial product between the sign bit of the multiplier and each bit of multiplicand should be represented in two’s complement format; (4) Sign extension. A 32-bit partial product is generated for the multiplication between two 16-bit fixed-point numbers. The bit length of the partial product is 16-bit. Therefore, it is essential to fill the

remaining bits with the sign extension approach; (5) Compute the sum of all partial products and shift the product 1-bit to the left. The function *mulFixed* in

Algorithm 3 illustrates details about binary multiplication of fixed-point representation.

Algorithm 3: Binary Multiplication of Fixed-point Representation

Inputs: Multiplicand (A_{mul}), multiplier (B_{mul}), integer bit length (L_{FI}), bit length (L_b)

Outputs: Product (P_{fixed}^r)

Function *mulFixed*(A_{mul} , B_{mul} , L_{FI} , L_b):

$P_{partial}, S_{partial}, T_{com} \leftarrow 0$ // array for partial product, sum of partial product, and two's complement

for ($i < L_b$) **do**

 // step 1: fill zeros to the empty bits

if $i > 0$ **then**

for ($n < L_b$) **do**
 | $P_{partial}[n] = 0$

 // step 2: calculate the partial product

for ($j < L_b$) **do**

 | $P_{partial}[i + j] = A_{mul} \& B_{mul}$

 // step 3: convert to the format of two's complement

if ($i == L_b - 1 \&\& B_{mul} == 1$) **then**

for ($p < L_b$) **do**

if $P_{partial}[L_b - 1 + p] == 0$ **then**

 | $P_{partial}[L_b - 1 + p] = 1$
 | $T_{com}[p] = P_{partial}[L_b - 1 + p]$

else

 | $P_{partial}[L_b - 1 + p] = 0$
 | $T_{com}[p] = P_{partial}[L_b - 1 + p]$

$T_{com} = \text{binaryAddition}(T_{com}, 1, L_b)$

for ($q < L_b$) **do**

 | $P_{partial}[L_b - 1 + q] = T_{com}[q]$

 // step 4: sign extension

for ($k = i + L_b; k < 2 \times L_b; k++$) **do**

if $P_{partial}[i + L_b - 1] == 0$ **then**

 | $P_{partial}[k] = 0$

else

 | $P_{partial}[k] = 1$

 // step 5: sum the partial product and shift 1-bit to the left

$S_{partial}, C_f \leftarrow \text{binaryAddition}(S_{partial}, P_{partial}, 2 \times L_b)$

if $i == L_b - 1$ **then**

if $C_f == 1$ **then**

for ($m < 2 \times L_b$) **do**

 | $P_{fixed}[m] = S_{partial}[m - 1]$

 | $P_{fixed}[0] = 0$

else

 | $P_{fixed}[m] = S_{partial}[m]$

$P_{fixed}^r = \text{bitRoundoff}(P_{fixed}, L_{FI}, L_b)$

return P_{fixed}^r

Table 4 Multiplication example of 16-bit fixed-point numbers

Multiplicand																
Multiplier																
1×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2×	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8×	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
9×	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0
10×	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0
11×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14×	1	1	1	1	0	1	0	0	0	0	0	0	1	1	0	1
15×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16×	0	0	1	0	1	1	1	1	1	1	0	0	1	0	1	1
product	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1
Multiplicand	1	0	1	0	0	0	0	0	0	1	1	0	1	0	0	1
Multiplier	1	0	1	0	0	0	1	1	1	0	0	0	0	0	1	0
1×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2×	0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	-
3×	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	-
4×	0	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-
5×	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
6×	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-
7×	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
8×	0	0	1	1	0	1	0	0	1	-	-	-	-	-	-	-
9×	0	1	1	0	1	0	0	1	-	-	-	-	-	-	-	-
10×	1	1	0	1	0	0	1	-	-	-	-	-	-	-	-	-
11×	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
12×	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-
13×	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
14×	0	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-
15×	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16×	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
product	0	1	0	1	0	0	0	0	0	1	0	1	0	0	1	0

Likewise, the following procedures can be deployed to validate the correctness of the above-described approach: (1) multiplication with floating-point numbers: $-0.746783 \times -2.89037 = 2.15847917971$; (2) conversion from floating-point to fixed-point numbers: $2.15847917971 \xrightarrow{Q(2.13)} 17682$; (3) binary conversion:

$17682 = 0100010100010010_2$. Hence, the product represented by a 16-bit fixed-point number between 1010000001101001 and 0101111110010111 is 0100010100010010 . The binary $0100010100010010_{Q(2.13)}$ can be converted to the decimal number as $0100010100010010_{Q(2.13)} \rightarrow 2.158447265625 \approx 2.15847917971$. The

conversion errors still exist even if the adaptive quantization approach is utilized.

4 Results and discussion

4.1 Algorithm verification

Figure 8 provides a comprehensive statistical analysis of the adaptive quantization for tiny YOLO3’s weights. The tiny YOLO3 has 8858734 weights, with maximum and minimum values of 400.63385009765625 and -17.461894989013672, respectively. It shows that a total number of 8850306 weights are represented by Q(0.15) format, which accounts for 99.9% of the weights in tiny YOLO3.

Evaluation of Conversion Errors The GEMM performs the convolution operation of tiny YOLO3 between feature maps and weights. Therefore, evaluating the conversion errors of these feature maps and weights is crucial. According to the dimension of feature maps in each convolution layer shown in Table 2, the percentage of Q-format representation for feature maps in each convolutional layer and weights are evaluated in this section. The detailed calculation method is illustrated in Algorithm 5. The

density of feature maps (from layer 1 to layer 13) and weights are described in Fig. 9, which shows that most of the feature maps and weights are located in the range of -1–1 (accounts for 60 – 90%).

In other words, the majority of feature maps and weights can be represented by Q(0.15) using adaptive quantization conversion. A small amount of data is represented by Q(1.14) and Q(4.11) formats. Totally, 20723456 feature maps and 8858734 weights (around 30 million parameters) are employed to evaluate the conversion between 32-bit floating-point to 16-bit fixed-point numbers with the adaptive quantization approach. The conversion error rate (error for each element in the corresponding convolutional layer) is utilized to assess the conversion error from 32-bit floating-point to 16-bit fixed-point numbers with the adaptive quantization algorithm. The conversion error rate (ζ) is defined as follows,

$$\zeta = \frac{1}{n} \|X_{\text{fixed}} - X_{\text{float}}\|_2 = \frac{1}{n} \sqrt{\sum_{i=0}^n (x_{\text{fixed}}^{(i)} - x_{\text{float}}^{(i)})^2} \quad (7)$$

where n is the number of floating-point representations converted by the adaptive quantization approach. Symbol $\|\bullet\|_2$ indicates the Euclidean norm. $x_{\text{fixed}}^{(i)} \in X_{\text{fixed}}$ and

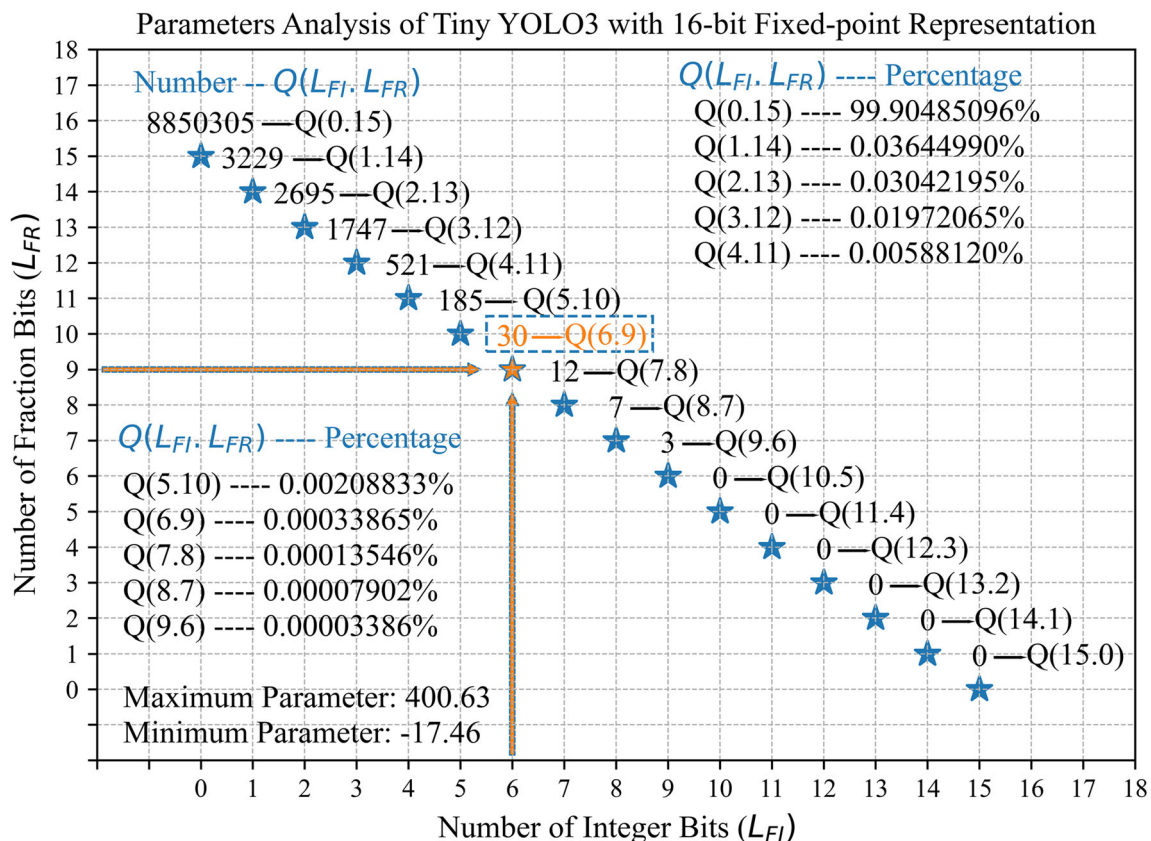


Fig. 8 Parameters analysis of 16-bit fixed-point representation format in tiny YOLO3

Distribution of Feature Maps and Parameters in Tiny YOLO3

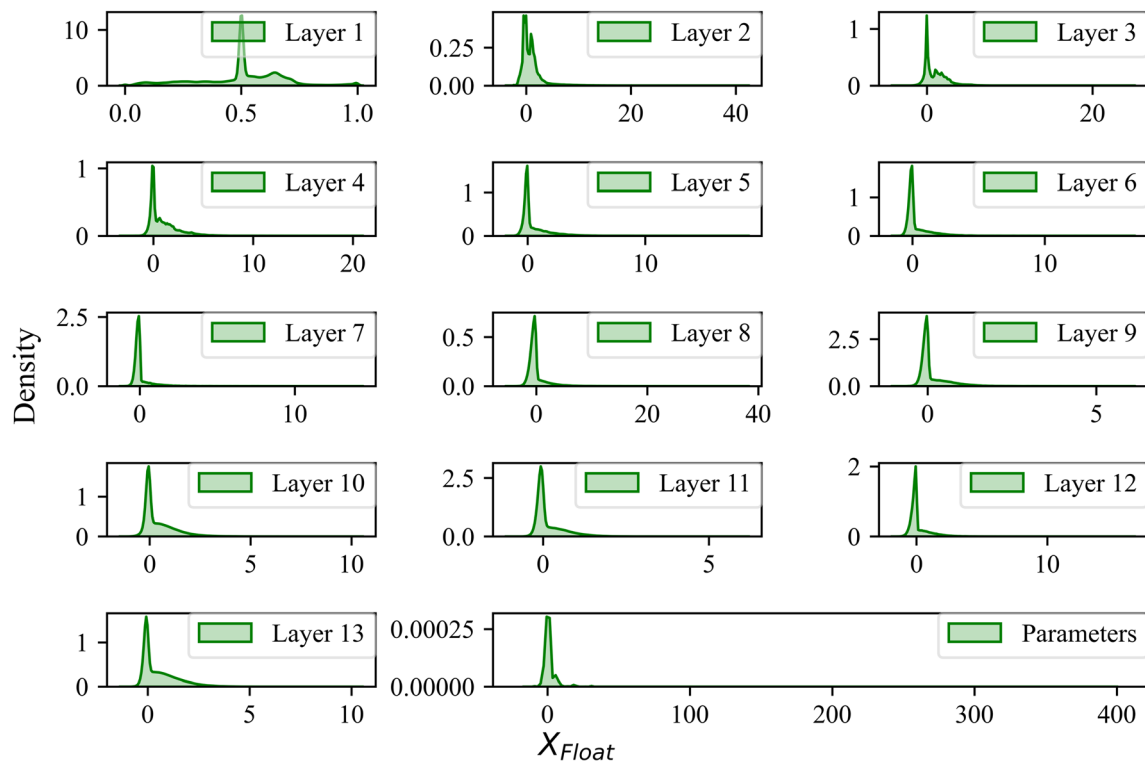


Fig. 9 Density of feature maps (from layer 1 to layer 13) and parameters in tiny YOLO3

$x_{\text{float}}^{(i)} \in X_{\text{float}}$ are the fixed-point and floating-point numbers, respectively. If the length of the integer bit is sufficiently enough, the range of the fixed-point number's representation expands at the expense of resolution. Converting from floating-point to fixed-point values with a high resolution or wide dynamic range will always result in rounding errors. Therefore, it is imperative to adopt adaptive quantization to explore the optimal representation for fixed-point numbers. To better highlight the comparison results, the conversion error rates are transformed by \log_{10} arithmetic operation, $\zeta \rightarrow \log_{10}(\zeta)$. It is worth noting that the longer the hist bar, the smaller the conversion error rates. The conversion error rates of adaptive quantization are much smaller than any other Q-format representations for feature maps and weights of tiny YOLO3.

To further explore a suitable representation format of weights and feature maps for neuromorphic processor design, Fig. 10 depicts the optimal and suboptimal L_{FI} for all weights and feature maps. The experimental results prove that the adaptive quantization on 16-bit fixed-point numbers exhibits a minimum conversion error rate, and it can be considered an optimal representation for 32-bit floating-point numbers. Besides, the suboptimal solution represented by $Q(6.9)$ has a relatively low accumulated error rate for all feature maps and weights of tiny YOLO3.

In addition, the accumulated errors of convolution results for each filter are calculated to evaluate the arithmetic errors of adaptive quantization on 16-bit fixed-point numbers. As mentioned before, since the number of convolutions is $M \times f_k$ in each layer, the total number of convolutions for accumulated error evaluation is 6164275. Furthermore, it shows that the relatively low conversion error rate of weights and feature maps is concentrated toward $L_{\text{FI}} = 6$ without using adaptive quantization. The convolution results for all different representation formats are demonstrated to search for the optimal representation format of tiny YOLO3. The optimal and suboptimal representation formats for all feature maps and weights are summarized in Fig. 10, which demonstrates that the adaptive quantization possesses optimal performance for the conversion from 32-bit floating-point to 16-bit fixed-point numbers. Figure 11 shows the error rate of GEMM operation with adaptive quantization, which demonstrates that the representation format with adaptive quantization presents a low computation error rate.

Moreover, Fig. 12 provides the accumulated conversion error rate with different representation formats for feature maps and weights in different layers, illustrating that the representation format with adaptive quantization and $Q(6.9)$ exhibits optimal and suboptimal solutions,

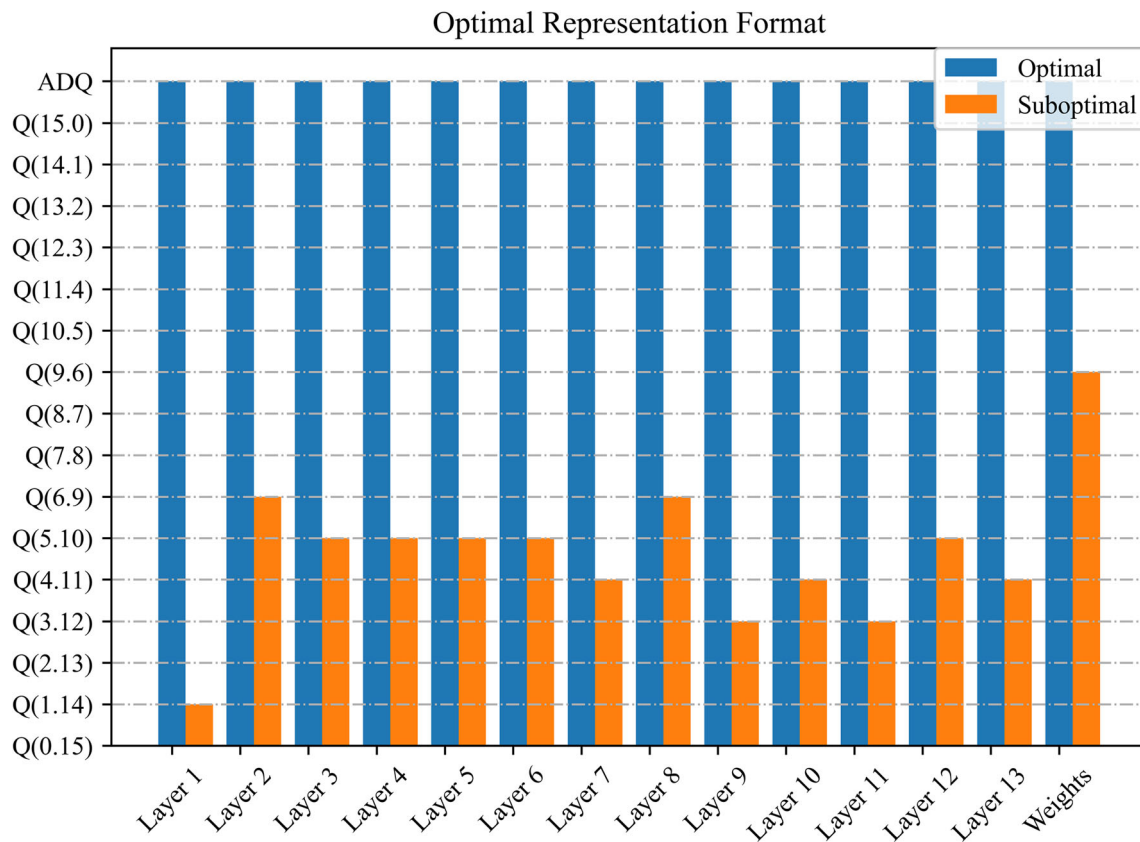


Fig. 10 Optimal representation formats of weights and feature maps in tiny YOLO3

respectively. Figure 13 presents the recognition results of tiny YOLO3 with different representation formats.

The experimental result shown in Fig. 13a, b, c, d, l, m, n, o, and p illustrate that the objects cannot be detected with the representation of $Q(0.15)$, $Q(1.14)$, $Q(2.13)$, $Q(3.12)$, $Q(11.4)$, $Q(12.3)$, $Q(13.2)$, $Q(14.1)$, and $Q(15.0)$, respectively. It can be observed that parts of the objects (compared with floating-point recognition result in Fig. 13r) are detected in Fig. 13e, f, j, and k. The representation formats with $Q(6.9)$, $Q(7.8)$, and $Q(8.7)$ shows correct detection results. There is no doubt that the representation with adaptive quantization shows readily acceptable detection results (refer to Fig. 13q). The experimental results show that the adaptive quantization algorithm not only has the minimum conversion error rate and minimum convolution error in tiny YOLO3 but also offers the same detection result as the 32-bit floating-point numbers by using the 16-bit fixed-point representation format.

In addition, as shown in Fig. 14, the statistic of recognition results indicates that the conversion error dominates the recognition results with the decrease of L_{FI} , and the rounding error becomes more and more significant with the decrease of L_{FI} .

Numbers represented by large L_{FI} can cover a wide representation range and have small conversion error while numbers represented by small L_{FI} has a high resolution and small roundoff error in convolution computation. Therefore, investigating an effective method to balance the L_{FI} is essential in designing a neuromorphic processor. Although the errors represented by $Q(6.9)$, $Q(7.8)$, and $Q(8.7)$ are larger than those represented by adaptive quantization, they can also be utilized as alternatives to convert the floating-point numbers to fixed-point numbers in the neuromorphic processor design.

Evaluation of Optimal Representation Format for Tiny YOLO3 The Microsoft common objects in context (MS COCO) 2014 and 2017 validation datasets are employed to search for the optimal representation format of tiny YOLO3. During the training process, 35504 samples are extracted from COCO-2014 to train the neural network. Therefore, 5000 remaining samples are selected from COCO-2014 to avoid utilizing training data for verification of the neural network's performance. Similarly, the COCO-2017 dataset contains an additional 5000 samples available for tiny YOLO3's performance evaluation. Since the tiny YOLO3's training is based on the 32-bit floating-point representation format, the first task is to explore the maximum weight distribution in each layer. The tiny YOLO3

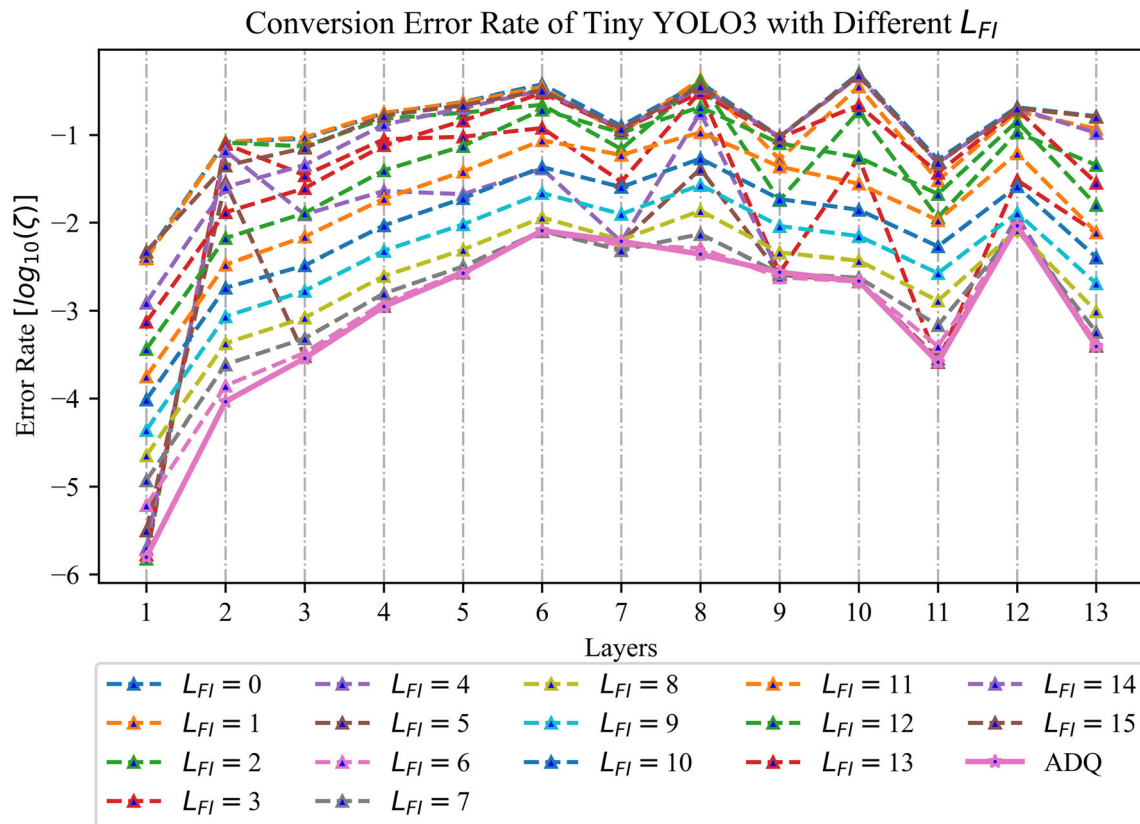


Fig. 11 Error comparison of GEMM with different L_{FI} formats

consists of 13 convolution layers and a total of 8845488 32-bit floating-point parameters.

The distribution of feature maps corresponding to different input data varies diversely. Therefore, this paper explores an average reference integer to determine the fixed-point representation format for tiny YOLO3. The benchmark COCO-2014 (5000 samples) and COCO-2017 (5000 samples) validation datasets are selected to investigate the optimal representation format of tiny YOLO3. The feature maps in different layers are compared one by one to search for the maximum elements. The feature maps' density distribution is obtained using the kernel density estimation (KDE) approach. The experimental result shows that the maximum density distribution of feature maps with two different databases in each neural network layer is almost identical. The maximum feature maps are 86.405121 and 88.912598 with the COCO-2014 and COCO-2017 datasets, respectively. It can be inferred that the largest feature map is represented by the form of $Q(7.8)$. Although the format of $Q(7.8)$ can cover all the feature maps of the tiny YOLO3, the density of $Q(7.8)$ is relatively low. In addition, the DNNs' precision represented by fixed-point format is determined by the fractional part's roundoff error and the integer part's overflow error. Thus, if the fixed-point representation can not cover all the

feature maps using average or minimum fixed-point representation formats, the overflow error of the integer part will be generated. On the contrary, using the largest fixed-point format to cover all the numbers will introduce the fractional part's roundoff error. $Q(7.8)$ can be utilized as the fixed-point representation format of tiny YOLO3, but it can only be served as a suboptimal solution. The comprehensive distribution of maximum feature maps with both COCO-2014 and COCO-2017 datasets is illustrated in Fig. 15.

The two databases, COCO-2014 and COCO-2017, deliver the optimal reference integers for adaptive quantization (49.50958 and 49.58947) equivalently, and both integers can be represented by the fixed-point format of $Q(6.9)$. From the above analysis results, it can be concluded that the optimal fixed-point representation format of tiny YOLO3 is $Q(6.9)$.

Performance Analysis of tiny YOLO3 The mean average precision (mAP) is one of the standard criteria to evaluate the performance of DNNs. This experiment will compare the tiny YOLO3's performance with the optimal 16-bit fixed-point representation ($Q(6.9)$) and the 32-bit floating-point representation format. Its high accuracy benefits from 32-bit floating-point representation. However, it requires complicated circuit control and interface connections

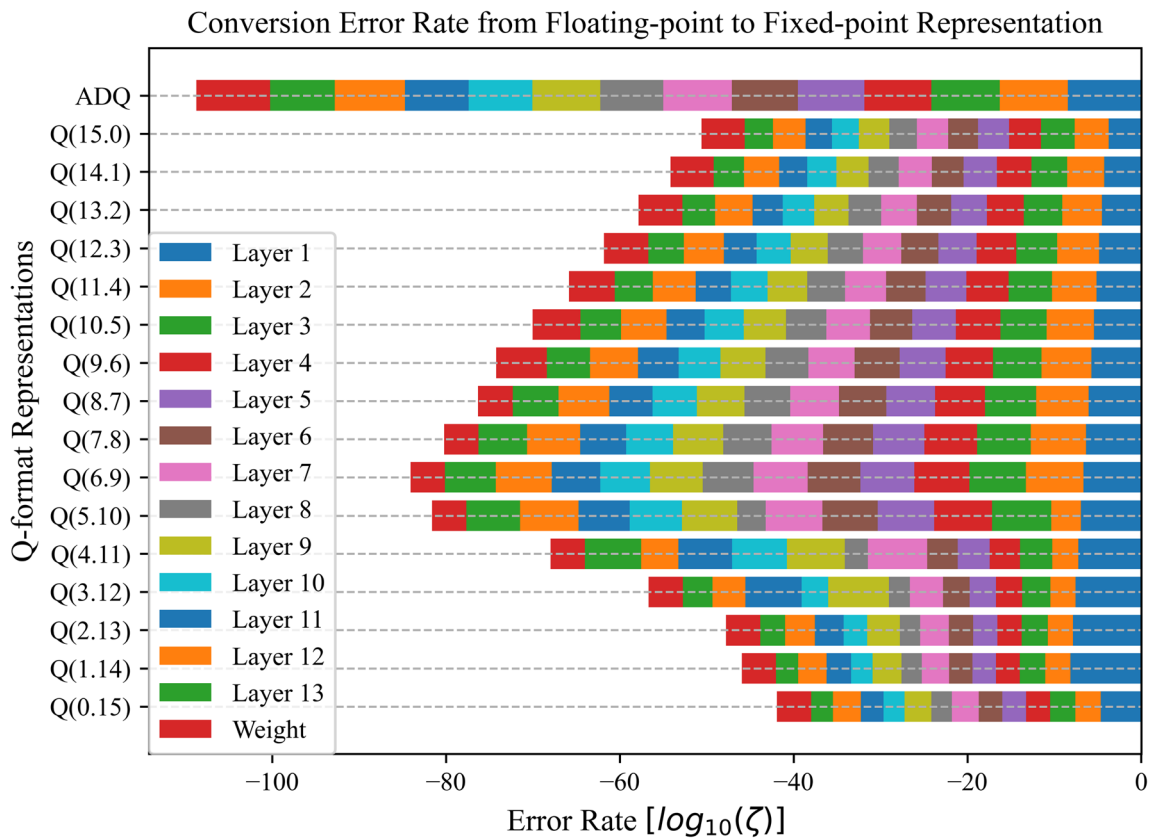


Fig. 12 Maximum, average and minimum error rate comparison of GEMM with different L_{FI}

among different layers to realize a multi-layer neuromorphic processor-oriented design. The fixed-point quantization technique affords a reliable solution for the tiny YOLO3’s hardware optimization to reduce memory utilization and circuit complexity. Similarly, the COCO-2014 and COCO-2017 datasets are deployed to assess tiny YOLO3’s performance with $Q(6.9)$ fixed-point representation format. Figure 16 illustrates the mAP of tiny YOLO3 using the $Q(6.9)$ and 32-bit floating-point representation formats, which demonstrates that the $Q(6.9)$ and 32-bit floating-points show almost the same mAP at each IoU threshold (floating-point: $[32.48, 36.40]@0.5$ and $Q(6.9)$: $[32.46, 36.28]@0.5$).

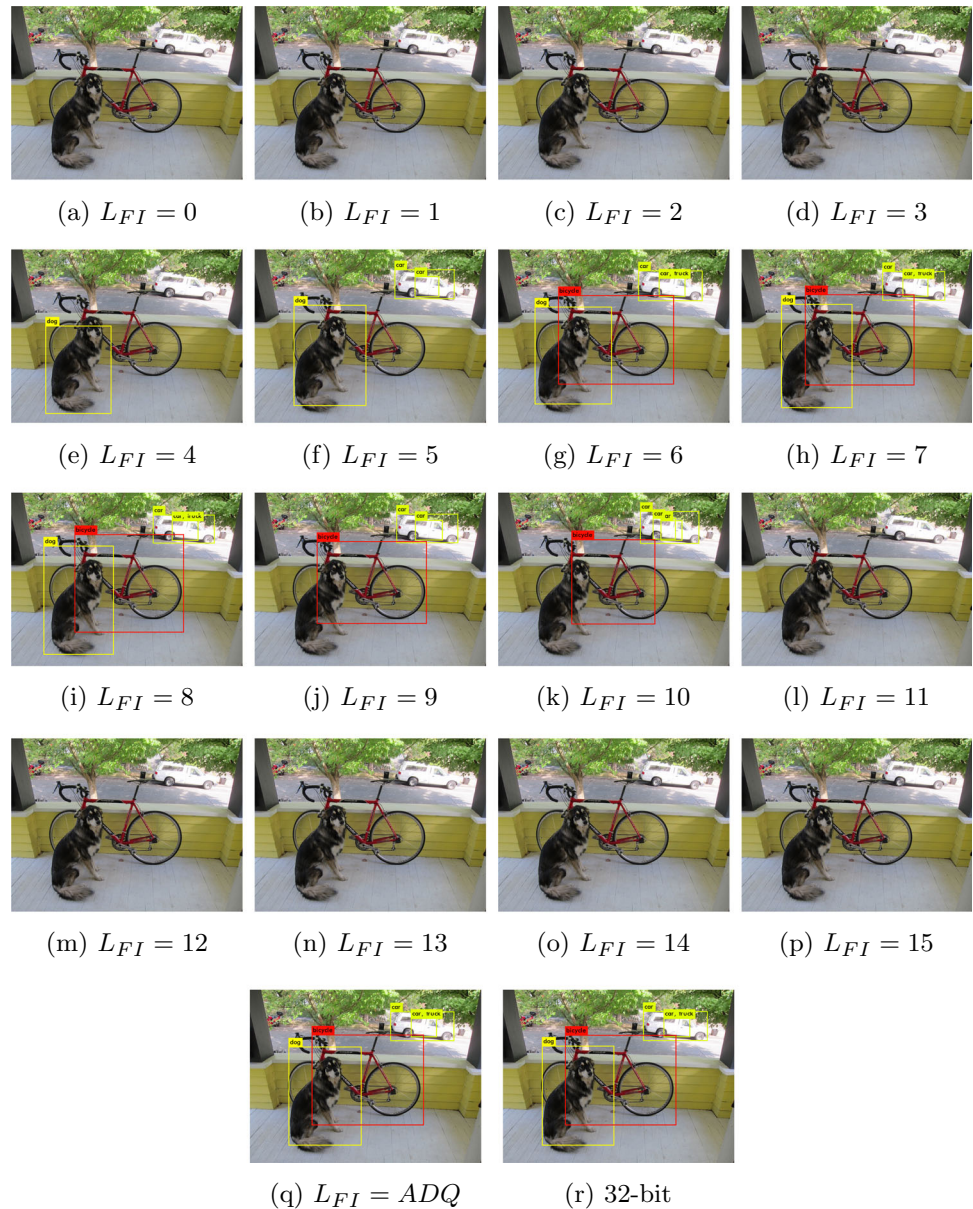
In addition, the mAP differences between the $Q(6.9)$ and 32-bit floating-point representation formats are in the range of $[-0.003, 0.002]$. The above result is adequate to validate that the $Q(6.9)$ representation format has the equivalent mAP as the 32-bit floating-point representation form in the tiny YOLO3’s performance evaluation. Meanwhile, the 16-bit fixed-point representation format can save half of the memory space and dramatically reduce the circuit design to realize its multi-layer neuromorphic processor.

4.2 Hardware verification

A hybrid multiplication module is designed to validate the hardware cost of the proposed method. Specifically, as shown in Fig. 17, weights and the feature maps represented by floating-point representation format are converted into integer binaries ($X_{INT}^W \rightarrow X_B^W, X_{INT}^F \rightarrow X_B^F$) according to Algorithm 4, and their corresponding integer bit length (L_{FI}^W, L_{FI}^F) are determined using Eq. 3.

The proposed hybrid Q-format multiplication module embraces both the integer bit length and the N -bit fixed-point binaries as the inputs. The magnitude of the variable N is designated by application requirement, and it can be 16-bit, 8-bit, or other values. The input bit length of L_{FI}^W and L_{FI}^F can be solved by $\log_2(N)$ if N is known. In this section, we develop a hybrid Q-format multiplication module that can accommodate varying bit lengths via the usage of a bit roundoff technique. Developing a uniform-length representation format for the multiplication module is crucial since most of the current neuromorphic processors deploy highly parallel general-purpose processing elements to emulate complicated DNNs’ models, as illustrated in Table 1. The product’s bit length of the multiplication module developed in this study is the same as the

Fig. 13 Recognition results with different L_{FI} representation formats



input bit length of the weights and feature maps, which are both N bits. As described in Sect. 3.3, since the multiplication of N -bit fixed-point binaries yields a $2N$ bits in length, the proposed module employs $\log_2(2N)$ to deduce the bit length of the product's L_{FI} (L_{FI}^{out}). The redundant extended sign bits can be eliminated without impacting the computation accuracy attributable to the bit roundoff block embedded into the multiplication module.

The post-synthesis of the proposed multiplication module utilizes Fujitsu 55 nm complementary metal-oxide semiconductor (CMOS) technology. Figure 18 depicts a post-synthesis simulation of hybrid Q-format multiplication, wherein the simulation's operating frequency and voltage are 100 MHz and 1.2 V, respectively.

As shown in Table 4, the signed binary product between $16'hA069$ and $16'hA382$ is $32'b001000101000100101000001010010 \rightarrow 32'h22895052$. Since the multiplication results between $Q(0.15)$ and $Q(2.13)$ can be represented by $Q(3.28)$, both "0"s at the MSB of $32'h22895052$ are signed bits. By shifting one bit to the left, the redundant sign bit can be removed, thereby increasing the number's resolution, as an illustration, $32'b00.100010100010010101000001010010 \xrightarrow{\ll 1} 32'b0.1000101000100101010000010100100$. Selecting the first 16-bit as the product yields $16'h4512$. Since there is only 1-bit sign extension, the fixed-point representation format for $Q(2.13)$ and $Q(2.13)$ only demands to shift one bit to the left while the radix point is located after the fifth bit, as explained in the following expression, $16'hA069_{Q(2.13)} \times 16'hA382_{Q(2.13)} \Rightarrow$

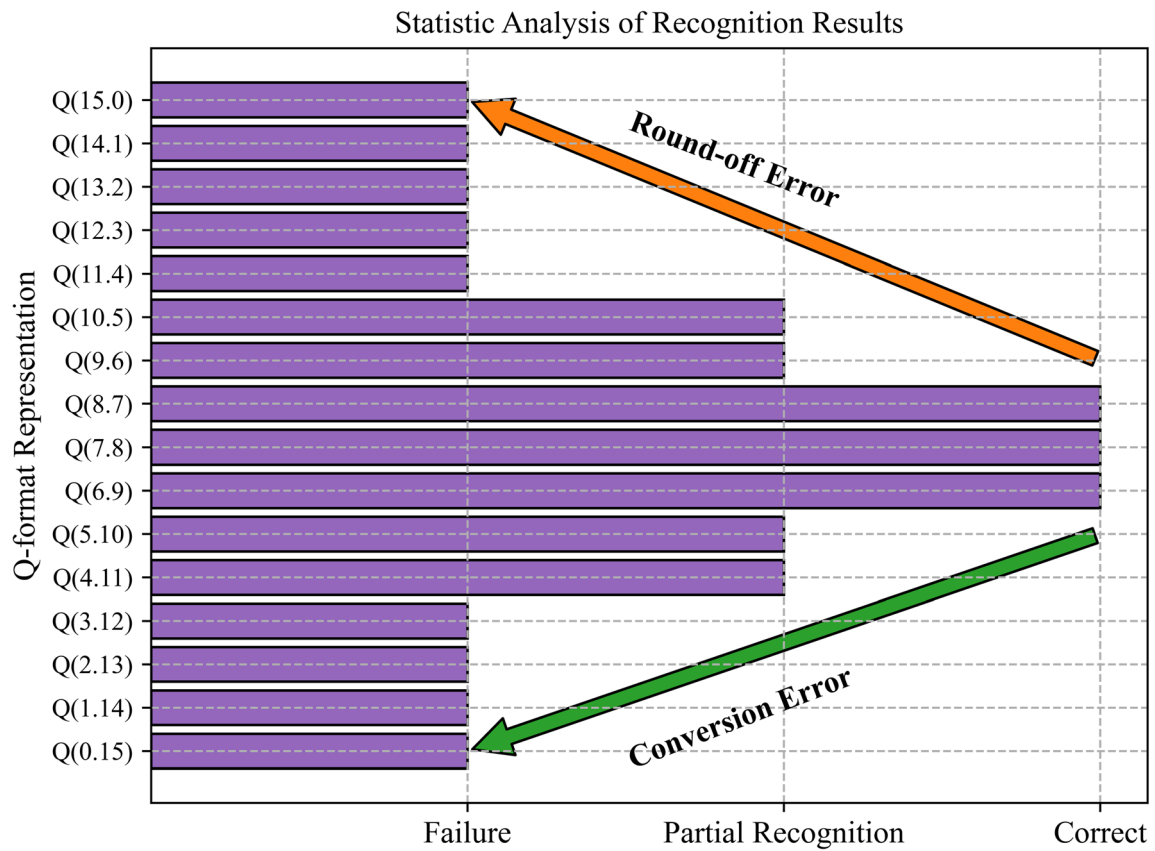


Fig. 14 Statistic of recognition results with different representation formats

Fig. 15 Maximum feature maps distribution of tiny YOLO3 evaluating with COCO-2014 and COCO-2017 datasets

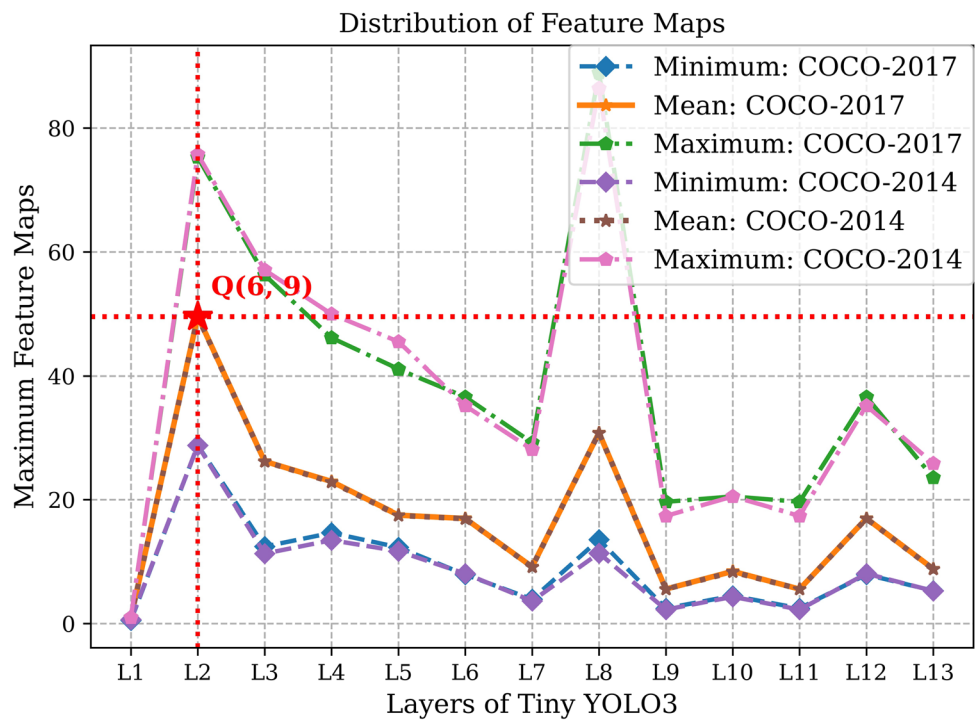


Fig. 16 Evaluation results of mAP with COCO dataset. diff(Q(6.9), Float) indicates the mAP difference between Q(6.9) fixed-point and floating-point representation formats

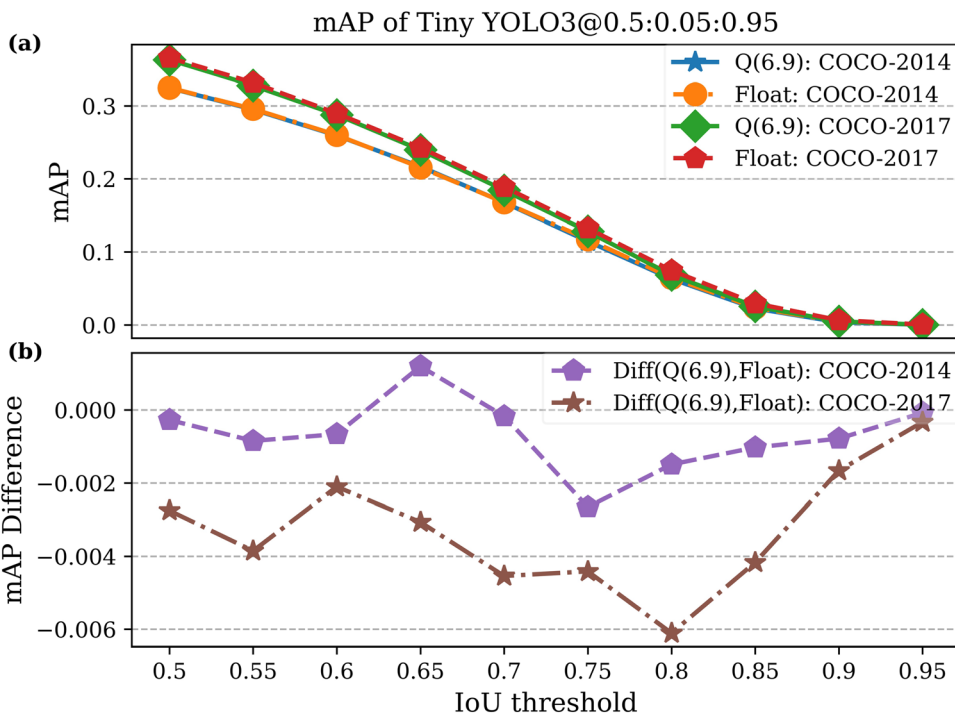
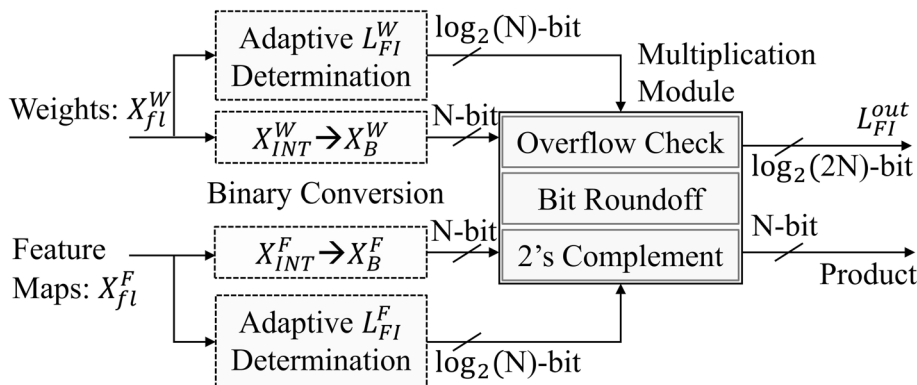


Fig. 17 Hardware simulation flow with a hybrid multiplication module



$32'b001000.10100010010101000001010010_{Q(5.26)} \lll 1$
 $32'b01000.101000100101010000010100100_{Q(4.27)} \xrightarrow{16-bit}$
 $16'h4512_{Q(4.11)} \quad (16'b01000.10100010010, L_{FI}^{out} = 4)$. It typically exhibits multiple extended sign bits in the product of relatively small fractional multiplication, as shown in Fig. 18 ($16'h0A32 \times 16'h0F13$). Precisely, the product of $16'h0A32$ and $16'h0F13$ is $32'b0.000000010011001101011110110110$, and the fixed-point representation format (refer to Sect. 3.3) for the multiplication between $Q(0.15)$ and $Q(0.15)$ is $Q(1.30)$. Left-shifting the product binaries by one bit and extracting the first 16 bits yield $16'h0A32_{Q(0.15)} \times 16'h0F13_{Q(0.15)} = 16'h0133$ ($16'b0.0000000100110011, L_{FI}^{out} = 0$). The multiplication of the same binaries with different fixed-point representation formats, $Q(3.12)$ and $Q(3.12)$ generates the product represented by $Q(7.24)$, i.e., $32'b00000000.10011001101011110110110$.

Likewise, shifting the binaries by six bits to the left and extracting the first 16 bits induce $16'h0A32_{Q(3.12)} \times 16'h0F13_{Q(3.12)} = 16'h4CD7$ ($16'b0.1001100110101111, L_{FI}^{out} = 0$). Similarly, the multiplication between $16'h2069$ and $16'h6F82$ can be derived as the following expressions, $16'h2069_{Q(0.15)} \times 16'h6F82_{Q(0.15)} = 16'h1C3B$ ($16'b0.0011100001110111, L_{FI}^{out} = 0$) and $16'h2069_{Q(1.14)} \times 16'h6F82_{Q(1.14)} = 16'h70EF$ ($16'b0.1110000110101111, L_{FI}^{out} = 0$).

In the simulation, we synthesize five distinct types of multiplication to verify the module's hardware cost, including 32-bit \times 32-bit, 16-bit \times 16-bit, 8-bit \times 8-bit, 4-bit \times 4-bit, and 2-bit \times 2-bit multiplication. Table 5 illustrates the post-synthesis results of the hybrid

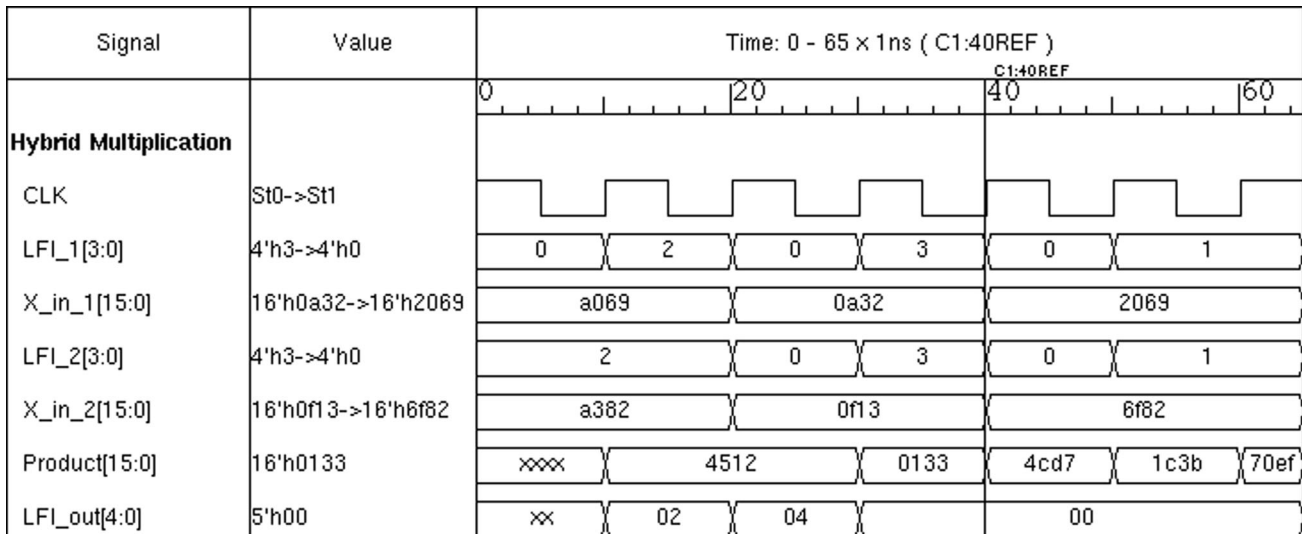


Fig. 18 Post-synthesis simulation of hybrid Q-format multiplication

multiplication module’s power consumption (dynamic power and static power), area, and delay.

Figure 19 gives a detailed analysis regarding the post-synthesis results. Specifically, as shown in Fig. 19a–e, the internal power dominates the whole module power consumption which is 58.9%, 59.6%, 58.6%, 58.9%, and 63.7% for 32-bit × 32-bit, 16-bit × 16-bit, 8-bit × 8-bit, 4-bit × 4-bit, and 2-bit × 2-bit multiplication, accordingly. Comparatively, the ratios of leakage power and switching power for various sorts of multiplication are in the range of 1.1–5.5% and 30.8–40.1%, correspondingly. 32-bit × 32-bit multiplication consumes 3.37, 21.27, 51.74, and 191.2 times more area than 16-bit × 16-bit, 8-bit × 8-bit, 4-bit × 4-bit, and 2-bit × 2-bit multiplication, respectively. Figure 19f offers a normalized area comparison among various multiplications. 32-bit multiplication module (maximum power and area: 1805.5 μW and 658441 μm²), as shown in Fig. 19g and h, totally requires 4.454 times, 32.63 times, 97.27 times, and 627.6 times as much energy as 16-bit, 8-bit, 4-bit, and 2-bit multiplication modules, respectively. The path between the LFI output registers and

their pins causes a maximum delay of 65.31 ps for all types of multiplication modules (refer to Fig. 19i).

Table 6 further contrasts the 8-bit hybrid Q-format multiplication (HQM) module with the benchmark multiplier in terms of power, delay, area, and power-delay product (PDP).

The power consumption range of benchmark multipliers is [0.2 mW, 164.8 mW], which is [3.614, 2978.062] times more than the HQM’s power consumption. The delay of the benchmark multipliers ranges from [0.62 ns, 16.69 ns], which is [9.493 to 255.55] times greater than the latency of our proposed multiplication module. Due to the adoption of different CMOS technologies, the synthesized area of HQM is [2.207, 97.705] times greater than those of the existing multipliers [316.81 μm², 14024 μm²]. Despite the increase in the circuit area, the PDP of the HQM is reduced by a factor of nearly [62.533, 56112.725] times compared to benchmark multipliers. In summary, the hybrid multiplier module proposed in this work offers considerably lower power consumption and latency characteristics than conventional multipliers.

Table 5 Post-synthesis results of hybrid multiplication module for different types of multiplication

Multiplication	Power [μW]			Area [μm ²]	Delay [ps]
	Internal	Switching	Leakage		
32-bit × 32-bit	1063.1	723.2949	19.0669	658441	65.31
16-bit × 16-bit	241.6986	158.0145	5.6827	195377	65.31
8-bit × 8-bit	32.4415	21.8551	1.0417	30954	65.31
4-bit × 4-bit	10.9410	7.1692	0.4504385	12726	65.31
2-bit × 2-bit	1.8339	0.8851732	0.1579009	3444	65.31

Hardware Simulation of Hybrid Q-format Multiplication Module

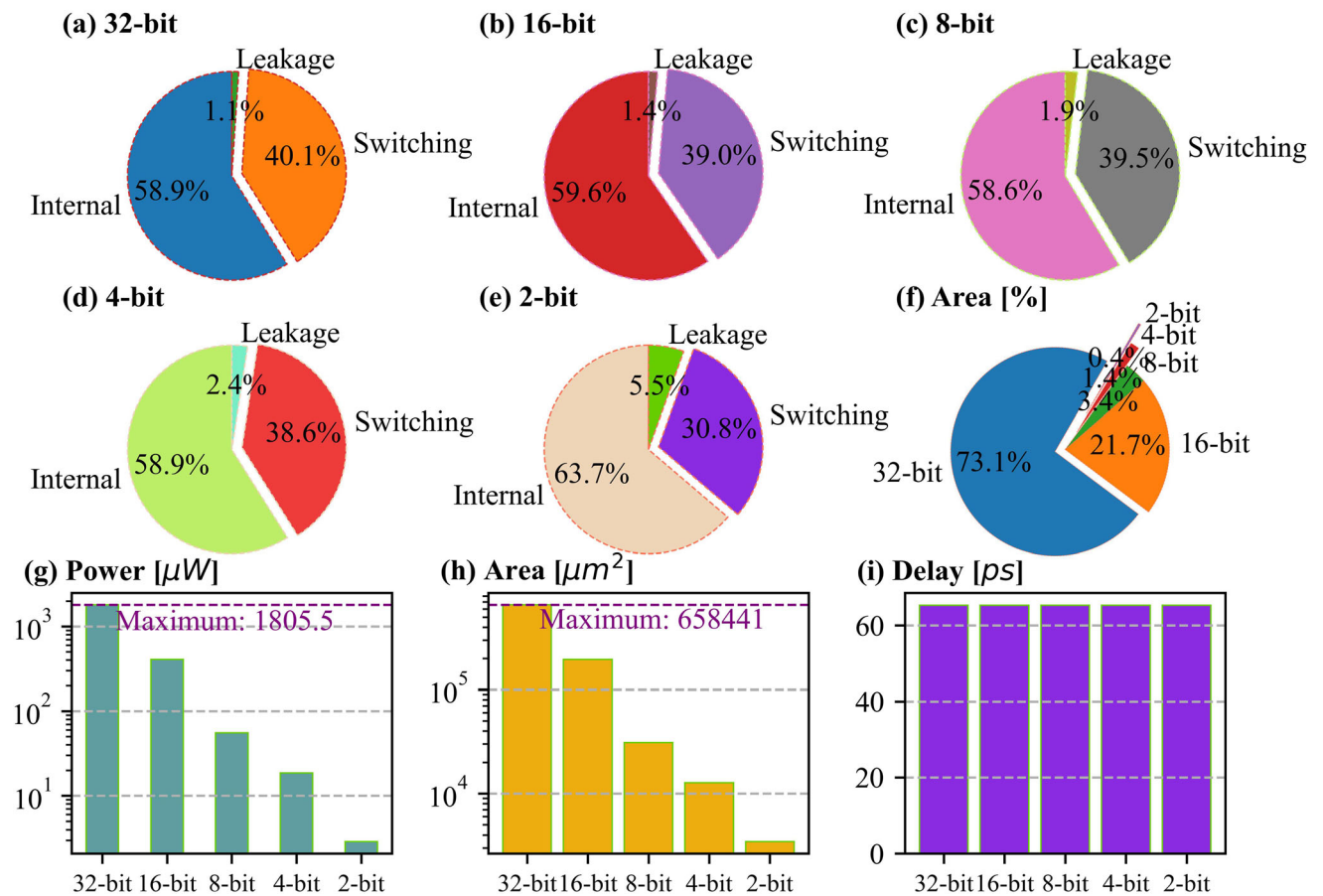


Fig. 19 Post-synthesis result of hybrid Q-format multiplication module

Table 6 Comparison of 8-bit hybrid Q-format multiplication module with benchmark multipliers

Technology	Multiplier	Power (mW)	Delay (ns)	Area (μm ²)	PDP (ns × mW)
180 nm	LOBO [53]	2.62	16.69	14024	43.7278
45 nm	R4ABM2 [54]	164.8	0.62	667.9	102.176
32 nm	Mitch-ω [55]	0.2	1.13	389	0.226
45 nm	ABM1-12 [56]	4.215	0.75	3054	3.16125
65 nm	RAD1024 [57]	63	3.219	2412	202.797
45 nm	RoBA [58]	3.422	0.65	2624	2.2243
65 nm	DRUM6 [59]	0.296	1.91	649.4	0.56536
45 nm	AL-LOA [60]	68.55	0.68	316.81	46.614
55 nm	Ours (HQM)	0.055338	0.06531	30954	0.0036141

5 Conclusion

In this paper, a neuromorphic processor-oriented hybrid multiplication strategy with an adaptive quantization method is proposed for the convolution operation of tiny YOLO3. The length of integer bits and the fraction bits of

16-bit fixed-point representations are adaptively determined based on the range of 32-bit floating-point numbers, overflow condition, and length of roundoff bits. The experimental result illustrates that the adaptive quantization on weights and feature maps maintains the same object detection accuracy while effectively reducing conversion

and roundoff errors from 16-bit fixed-point to 32-bit floating-point representations. In addition, the optimal representation formats ($Q(6.9)$) of 16-bit fixed-point values have been achieved as a reference for the neuromorphic processor design. Moreover, a hybrid multiplication module with low power consumption and low latency is also designed, laying a solid foundation for the development of neuromorphic processors.

Appendix: pseudo codes for simulation

Algorithm 4 provides the pseudo-codes for the conversion from fixed-point number to binary.

Algorithm 4: Conversion from 16-bit Fixed-point Numbers to Binary Representation

Inputs: Fixed-point numbers (X_{fixed}), bit length (L_b)

Outputs: Binary array (B_{array})

Function fixed2binary(X_{fixed} , L_b):

```

P_array ← 0      // array for positive numbers
N_array ← 1      // array for negative numbers
if  $X_{fixed} \neq 0$  &&  $abs(X_{fixed}) < 2^{L_b}$  then
     $S_{fixed} = X_{fixed} > 0 ? 0 : 1$ 
    // for positive numbers
    if  $S_{fixed} == 0$  then
        for ( $i = 0; X_{fixed} > 0; i++$ ) do
             $P_{array}[i] = X_{fixed} \% 2$ 
             $X_{fixed} = X_{fixed} / 2$ 
             $B_{array}[i] = P_{array}[i]$ 
        // for negative numbers
    else
         $X_{fixed} = -X_{fixed}$ 
        for ( $j = 0; X_{fixed} > 0; j++$ ) do
             $N_{array}[j] = X_{fixed} \% 2$ 
            if  $N_{array}[j] == 0$  then
                 $N_{array}[j] = 1$ 
            else
                 $N_{array}[j] = 0$ 
             $X_{fixed} = X_{fixed} / 2$ 
             $B_{array}[i] = P_{array}[i]$ 
         $B_{array} = binaryAddition(B_{array}, 1)$ 
    else if  $abs(X_{fixed}) > 2^{L_b}$  then
        for  $k < L_b$  do
             $B_{array}[k] = 1$ 
    else
        for  $m < L_b$  do
             $B_{array}[m] = 0$ 
    return  $B_{array}$ 

```

Algorithm 5 presents the pseudo-codes (float 2 fixed) for the conversion from 32-bit floating-point to 16-bit fixed-point numbers.

Algorithm 5: Conversion from 32-bit Floating-point to 16-bit Fixed-point Numbers

Inputs: Floating-point numbers (X_{float}), flag of using adaptive quantization (F_{adq}), predefined length of integer bits (L_{FI}^p)

Outputs: Fixed-point numbers (X_{fixed}), Q-format percentage (η), conversion error rate (ζ)

Function float2fixed(X_{float} , F_{adq} , L_{FI}^p):

```

 $N_q \leftarrow 0$  //  $N_q$  - Q-format counting
 $\epsilon \leftarrow 0$  //  $\epsilon$  - conversion error
 $\alpha \leftarrow 2^{-1} - 2^{-15}$  //  $\alpha$  - constant
for  $j < (\text{len}(X_{float}))$  do
  if  $F_{adq} = 0$  then
    if  $X_{float}[j] > -0.5$  &  $X_{float}[j] < \alpha$  then
       $L_{FI} = 0$ 
    if  $X_{float}[j] > \alpha$  then
       $L_{FI} = \text{floor}(\log_2(X_{float}[j]/\alpha))$ 
    else
       $L_{FI} = \text{floor}(\log_2(-X_{float}[j])) + 1$ 
     $N_q[j] = L_{FI}$ 
  else
     $L_{FI} = L_{FI}^p$ 
     $N_q[j] = L_{FI}$ 
   $X_{fixed} = \text{int}(\text{round}(X_{float}[j] * 2^{15-L_{FI}}))$ 
   $B_{array} = \text{fixed2binary}(X_{fixed})$ 
   $X_{fixed}^b = \text{binary2fixed}(B_{str}, L_{FI})$ 
   $\epsilon = (X_{fixed}^b - X_{float}[j])^2 + \epsilon$ 
 $\zeta = \text{sqr}t(\epsilon) / \text{len}(X_{float})$ 
 $N_{fixed} \leftarrow 0$  //  $N_{fixed}$  - number of fixed-point
for  $i < 16$  do
   $N_{fixed}[i] = N_q.\text{count}(i)$ 
 $N_{fixed}^{arr} = \text{array}(N_{fixed})$  // array of  $N_{fixed}$ 
 $\eta = N_{fixed}^{arr} / \text{len}(X_{float})$ 
return  $X_{fixed}$ ,  $\eta$ ,  $\zeta$ 

```

Algorithm 6 offers the pseudo-codes for implementing the binary addition of fixed-point representation format.

Algorithm 6: Binary Addition of Fixed-point Representation

Inputs: Addend (A_{add}), augend (B_{aug}), integer bit length of addend (L_{FI}^{add}), integer bit length of augend (L_{FI}^{aug}), bit length (L_b)

Outputs: Addition result (R_{add}), bit shift flag (S_{lfi})

Function float2fixed($A_{add}, B_{aug}, L_{FI}^{add}, L_{FI}^{aug}, L_b$):

```

 $L_{FI}^{sum} = \max(L_{FI}^{add}, L_{FI}^{aug})$ 
 $L_{FR}^{sum} = \max((L_b - 1 - L_{FI}^{add}), (L_b - 1 - L_{FI}^{aug}))$ 
 $N_{sum} = L_{FI}^{sum} + L_{FR}^{sum} + 1$ 
 $C_{temp} \leftarrow 0$  // array for temporary result
for  $i < N_{sum}$  do
     $D_{radix} \leftarrow 0$  // integer bits difference between addend and
    augend
    if ( $L_{FI}^{add} == L_{FI}^{aug}$ ) then
         $D_{radix} = 0$ 
         $C = \text{binaryAddition}(A_{add}, B_{aug}, N_{sum})$ 
        for  $v < L_b$  do
             $R_{add}[v] = C[v]$ 
    else
        if  $L_{FI}^{sum} == L_{FI}^{aug}$  then
             $D_{radix} = L_{FI}^{aug} - L_{FI}^{add}$ 
            // sign extension
            for  $j < D_{radix}$  do
                 $A_{add}[j + L_b] = A_{add}[L_b - 1]$ 
            // fill zeros to empty bits
            for  $k < N_{sum}$  do
                if  $k < D_{radix}$  then
                     $B_{aug}[k] = 0$ 
                else
                     $B_{aug}[k] = B_{aug}[k - D_{radix}]$ 
            else
                 $D_{radix} = L_{FI}^{add} - L_{FI}^{aug}$ 
                for  $p < D_{radix}$  do
                     $B_{aug}[p + L_b] = B_{aug}[L_b - 1]$ 
                for  $q < N_{sum}$  do
                    if  $q < D_{radix}$  then
                         $A_{add}[q] = 0$ 
                    else
                         $A_{add}[q] = A_{add}[q - D_{radix}]$ 
             $C, C_f \leftarrow \text{binaryAddition}(A_{add}, B_{aug}, N_{sum})$ 
            for  $v < L_b$  do
                 $R_{add}[v] = C[v + D_{radix}]$ 
            if  $C_f == 1$  then
                 $S_{lfi} = 1$ 
return  $R_{add}, S_{lfi}$ 

```

Acknowledgements This work was supported in part by JSPS KAKENHI under Grant JP21K17719, and in part by New Energy and Industrial Technology Development Organization (NEDO) and Center for Innovative Integrated Electronic Systems(CIES) consortium.

Data availability The datasets generated and analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors declare that they have no conflict of interest with respect to the research, authorship and/or publication of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Mead C (1990) Neuromorphic electronic systems. *Proc IEEE* 78(10):1629–1636
- Yang S, Wang J, Deng B, Azghadi MR, Linares-Barranco B (2021) Neuromorphic context-dependent learning framework with fault-tolerant spike routing. *IEEE Trans Neural Netw Learn Syst*. <https://doi.org/10.1109/TNNLS.2021.3084250>
- Schuman CD, Kulkarni SR, Parsa M, Mitchell JP, Kay B (2022) Opportunities for neuromorphic computing algorithms and applications. *Nat Comput Sci* 2(1):10–19
- Yang S, Gao T, Wang J, Deng B, Azghadi MR, Lei T, Linares-Barranco B (2022) SAM: a unified self-adaptive multicompartmental spiking neuron model for learning With working memory. *Front Neurosci* 16(850945):1–22
- Shaban A, Bezugam SS, Suri M (2021) An adaptive threshold neuron for recurrent spiking neural networks with nanodevice hardware implementation. *Nat Commun* 12(1):1–11
- Yang S, Wang H, Hao X, Li H, Wei X, Deng B, Loparo KA (2022) BiCoSS: toward large-scale cognition brain with multi-granular neuromorphic architecture. *IEEE Trans Neural Netw Learn Syst* 33(7):2801–2815
- Shastri BJ, Tait AN, Ferreira de Lima T, Pernice WH, Bhaskaran H, Wright CD, Prucnal PR (2021) Photonics for artificial intelligence and neuromorphic computing. *Nat Photonics* 15(2):102–114
- Li E, Wu X, Chen Q, Wu S, He L, Yu R, Hu Y, Chen H, Guo T (2021) Nanoscale channel organic ferroelectric synaptic transistor array for high recognition accuracy neuromorphic computing. *Nano Energy* 85(106010):1–9
- Yang S, Deng B, Wang J, Li H, Lu M, Che Y, Wei X, Loparo KA (2019) Scalable digital neuromorphic architecture for large-scale biophysically meaningful neural network with multi-compartment neurons. *IEEE Trans Neural Netw Learn Syst* 31(1):148–162
- Li T, Ma Y, Endoh T (2020) A systematic study of tiny YOLO3 inference: toward compact brainware processor with less memory and logic gate. *IEEE Access* 8:142931–142955
- Deng L, Li G, Han S, Shi L, Xie Y (2020) Model compression and hardware acceleration for neural networks: a comprehensive survey. *Proc IEEE* 108(4):485–532
- Venkataramani S, Sun X, Wang N, Chen CY, Choi J, Kang, Gopalakrishnan K et al. (2020) Efficient AI system design with cross-layer approximate computing. In: *Proceedings of the IEEE*, vol. 108, no. 12, pp 2232–2250
- Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. *Proc IEEE* 105(12):2295–2329
- Natsui M, Suzuki D, Tamakoshi A, Watanabe T, Honjo H, Koike H, Nasuno T, Ma Y et al (2019) A 47.14- μ W 200-MHz MOS/MTJ-hybrid nonvolatile microcontroller unit embedding STT-MRAM and FPGA for IoT applications. *IEEE J Solid-State Circuits* 54(11):2991–3004
- Merolla PA, Arthur JV, Alvarez-Icaza R, Cassidy AS, Sawada J, Akopyan F, Jackson BL, Imam N, Guo C, Nakamura Y, Brezzo B, Vo I, Esser SK, Appuswamy R, Taba B, Amir A, Flickner MD, Risk WP, Manohar R, Modha DS (2014) A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345(6197):668–673
- Pei J, Deng L, Song S, Zhao M, Zhang Y, Wu S, Wang G, Zou Z, Wu Z, He W et al (2019) Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* 572(7767):106–111
- Davies M, Srinivasa N, Lin T-H, Chinya G, Cao Y, Choday SH, Dimou G, Joshi P, Imam N, Jain S et al (2018) Loihi: a neuro-morphic many core processor with on-chip learning. *IEEE Micro* 38(1):82–99
- Davies M et al (2021) Taking neuromorphic computing to the next level with Loihi2. In: *Intel Labs' Loihi 2 Neuromorphic Research Chip and the Lava Software Framework*. Technology Brief, Intel, pp 1–7
- Benjamin BV, Gao P, McQuinn E, Choudhary S, Chandrasekaran AR, Bussat J, Alvarez-Icaza R, Arthur JV, Merolla PA, Boahen K (2014) Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc IEEE* 102(5):699–716
- Li T, Ma Y, Shen H, Endoh T (2020) FPGA implementation of real-time pedestrian detection using normalization-based validation of adaptive features clustering. *IEEE Trans Veh Technol* 69(9):9330–9341
- Yuan X, Huang G, Shi K (2020) Improved adaptive path following control system for autonomous vehicle in different velocities. *IEEE Trans Intell Transp Syst* 21(8):3247–3256
- Liu Y-T, Lin Y-Y, Wu S-L, Chuang C-H, Lin C-T (2015) Brain dynamics in predicting driving fatigue using a recurrent self-evolving fuzzy neural network. *IEEE Trans Neural Netw Learn Syst* 27(2):347–360
- Mao B, Kawamoto Y, Kato N (2020) AI-based joint optimization of QoS and security for 6G energy harvesting internet of things. *IEEE Internet Things J* 21:452
- Wong KK, Fortino G, Abbott D (2020) Deep learning-based cardiovascular image diagnosis: a promising challenge. *Futur Gener Comput Syst* 110:802–811
- Liang F, Yu W, Liu X, Griffith D, Golmie N (2020) Toward edge-based deep learning in industrial internet of things. *IEEE Internet Things J* 7(5):4329–4341
- Figurnov M, Ibraimova A, Vetrov DP, Kohli P (2016) PerforatedCNNs: acceleration through elimination of redundant convolutions. In: *Advances in Neural Information Processing Systems*, pp. 947–955

27. Dutt S, Dash S, Nandi S, Trivedi G (2019) Analysis, modeling and optimization of equal segment based approximate adders. *IEEE Trans Comput* 68(3):314–330
28. Liu C, Han J, Lombardi F (2015) An analytical framework for evaluating the error characteristics of approximate adders. *IEEE Trans Comput* 64(5):1268–1281
29. Zhu N, Goh WL, Zhang W, Yeo KS, Kong ZH (2009) Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. *J Trans Very Large Scale Integr (VLSI) Syst* 18(8):1225–1229
30. Deng L, Li G, Han S, Shi L, Xie Y (2020) Model compression and hardware acceleration for neural networks: a comprehensive survey. *Proc IEEE* 108(4):485–532
31. Han S, Mao H, Dally WJ (2016) Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. In: International conference on learning representations (ICLR), San Juan, Puerto Rico
32. Blalock D, Ortiz JGG, Frankle J, Gutttag J (2020) What is the state of neural network pruning? [arXiv:2003.03033](https://arxiv.org/abs/2003.03033)
33. Srinivas S, Babu RV (2015) Data-free parameter pruning for deep neural networks. [arXiv preprint arXiv:1507.06149](https://arxiv.org/abs/1507.06149)
34. Liu C, Wu H (2019) Channel pruning based on mean gradient for accelerating convolutional neural networks. *Signal Process* 156:84–91
35. Chen Z, Xu T-B, Du C, Liu C-L, He H (2020) Dynamical channel pruning by conditional accuracy change for deep neural networks. *IEEE Trans Neural Netw Learn Syst* 32(2):799–813
36. Yang H, Tang M, Wen W, Yan F, Hu D, Li A, Li, H, Chen Y (2020) Learning low-rank deep neural networks via singular vector orthogonality regularization and singular value sparsification. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops, pp. 678–679
37. Ba J, Caruana R (2014) Do deep nets really need to be deep? In: Advances in neural information processing systems, pp. 2654–2662
38. Buciluă C, Caruana R, Niculescu-Mizil A (2006) Model compression. In: Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 535–541
39. Hinton G, Vinyals O, Dean J (2015) Distilling the knowledge in a neural network. [arXiv preprint arXiv:1503.02531](https://arxiv.org/abs/1503.02531)
40. Furlanello T, Lipton ZC, Tschannen M, Itti L, Anandkumar A (2018) Born again neural networks. [arXiv preprint arXiv:1805.04770](https://arxiv.org/abs/1805.04770)
41. Romero A, Ballas N, Kahou SE, Chassang A, Gatta C, Bengio Y (2014) Fitnets: hints for thin deep nets. [arXiv preprint arXiv:1412.6550](https://arxiv.org/abs/1412.6550)
42. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y (2017) Quantized neural networks: training neural networks with low precision weights and activations. *J Mach Learn Res* 18(1):6869–6898
43. Jung S, Son C, Lee S, Son J, Han J-J, Kwak Y, Hwang SJ, Choi C (2019) Learning to quantize deep networks by optimizing quantization intervals with task loss. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), Long Beach, United States, pp. 4350–4359
44. Zhang X, Liu S, Zhang R, Liu C, Huang D, Zhou S, Guo J, Guo Q, Du Z, Zhi T et al. (2020) Fixed-point back-propagation training. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 2330–2338
45. Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P (2015) Deep learning with limited numerical precision. In: International conference on machine learning (ICML), Lille, France, pp 1737–1746
46. Zhou Y, Moosavi-Dezfooli S.M, Cheung N-M, Frossard P (2017) Adaptive quantization for deep neural network. [arXiv preprint arXiv:1712.01048](https://arxiv.org/abs/1712.01048)
47. Zhu X, Zhou W, Li H (2018) Adaptive layerwise quantization for deep neural network compression. In: 2018 IEEE international conference on multimedia and expo (ICME), San Diego, CA, USA, pp 1–6
48. Kwon D, Lim S, Bae J-H, Lee S-T, Kim H, Kim C-H, Park B-G, Lee J-H (2018) Adaptive weight quantization method for non-linear synaptic devices. *IEEE Trans Electron Devices* 66(1):395–401
49. Yin S, Seo J-S (2019) A 2.6 Tops/w 16-bit fixed-point convolutional neural network learning processor in 65-nm CMOS. *IEEE Solid-State Circuits Lett* 3:13–16
50. Lindstrom P (2014) Fixed-rate compressed floating-point arrays. *IEEE Trans Visual Comput Graph* 20(12):2674–2683
51. Rastegari M, Ordonez V, Redmon J, Farhadi A (2016) XNOR-Net: Imagenet classification using binary convolutional neural networks. In: European Conference on Computer Vision, Springer, Amsterdam, The Netherlands, pp 525–542
52. Courbariaux M, Bengio Y, David J-P (2015) Binaryconnect: training deep neural networks with binary weights during propagations. In: Advances in neural information processing systems, pp 3123–3131
53. Ratko P, Bulić P (2020) On the design of logarithmic multiplier using radix-4 booth encoding. *IEEE Access* 8:64578–64590
54. Liu W, Qian L, Wang C, Jiang H, Han J, Lombardi F (2017) Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Trans Comput* 66(8):1435–1441
55. Kim MS, Del Barrio AA, Oliveira LT, Hermida R, Bagherzadeh N (2018) Efficient Mitchell’s approximate log multipliers for convolutional neural networks. *IEEE Trans Comput* 68(5):660–675
56. Waris H, Wang C, Liu W (2020) Hybrid low radix encoding-based approximate booth multipliers. *IEEE Trans Circuits Syst II Express Briefs* 67(12):3367–3371
57. Leon V, Zervakis G, Soudris D, Pekmestzi K (2017) Approximate hybrid high radix encoding for energy-efficient inexact multipliers. *IEEE Trans Very Large Scale Integr (VLSI) Syst* 26(3):421–430
58. Zendegani R, Kamal M, Bahadori M, Afzali-Kusha A, Pedram M (2016) RoBA multiplier: a rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing. *IEEE Trans Very Large Scale Integr (VLSI) Syst* 25(2):393–401
59. Hashemi S, Bahar RI, Reda S (2015) DRUM: a dynamic range unbiased multiplier for approximate applications. In: 2015 IEEE/ACM International conference on computer-aided design (ICCAD), pp 418–425
60. Liu W, Xu J, Wang D, Wang C, Montuschi P, Lombardi F (2018) Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications. *IEEE Trans Circuits Syst I Regul Pap* 65(9):2856–2868