IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 30, 2020, accepted July 27, 2020, date of publication August 3, 2020, date of current version August 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3013934

A Systematic Study of Tiny YOLO3 Inference: Toward Compact Brainware Processor With Less Memory and Logic Gate

TAO LI[1], YITAO MA[1,2,3], AND TETSUO ENDOH[1,2,3], (Senior Member, IEEE)
[1]Center for Innovative Integrated Electronic Systems (CIES), Tohoku University, Sendai 980-8572, Japan
[2]School of Engineering, Tohoku University, Sendai 980-8579, Japan
[3]Research Institute of Electrical Communication (RIEC), Tohoku University, Sendai 980-8577, Japan

Corresponding author: Tetsuo Endoh (tetsuo.endoh@cies.tohoku.ac.jp)

This work was partially supported by Crossministerial Strategic Innovation Promotion Program (SIP) 2nd Phase-Physical Space Digital Processing Platform: ''R&D of Ultra-Low Power IoT Devices and Its Technical Platform with MTJ/CMOS Hybrid Technologies for Society 5.0'', Cabinet Office, and Japan Science and Technology Agency-Open Innovation Platform with Enterprises, Research Institute and Academia (JSTOPERA), grant number JPMJOP1611, and Center for Innovative Integrated Electronic Systems (CIES) consortium.

ABSTRACT The emerging of deep neural networks, especially the convolutional neural network (CNN), substantially promotes the fast development of brainware processors in object detection. However, the vast network architecture brings severe challenges to the design of brainware processor, which requires a large number of logic gates and memories. Therefore, a compact brainware processor with less memory and logic gate has a high demand in object detection. Typically, the object detection involves single-shot and multi-shot detectors in accordance with different detection principle. In the early stage, the multi-shot detector has a leading role in solving object detection issues, such as region-based convolutional neural networks (R-CNNs), faster R-CNNs etc. However, the multi-shot detector suffers from a low detection rate comparing with the single-shot detector. The you only look once (YOLO) algorithm, as the state-of-the-art real-time object detection algorithm, receives extensive attention from the academics and industry. Particularly, the lightweight YOLO algorithm, tiny YOLO3, has excellent potential for circuit design of compact brainware processor. Nonetheless, systematic studies of tiny YOLO3 are still missing up to the present. This paper offers a thorough review of the tiny YOLO3 algorithm, which can fill the gap in the field of object detection. Furthermore, the open solutions of compressing the tiny YOLO3 algorithm are proposed from the aspects of algorithm, hardware and emerging technology. The comprehensive study presented in this paper can not only enhance understanding of the tiny YOLO3 algorithm for researchers or engineers but also make a significant contribution to accelerating the development of compact brainware processor.

INDEX TERMS Tiny YOLO3, brainware processor, deep neural network, CNN, hardware acceleration.

I. INTRODUCTION

Artificial intelligence (AI) has made remarkable achievements in academia and industry since its emergence in 1956, and its booming driven by deep learning happens in this century, especially since 2005. The deep learning technology, as a subset of AI, intends to rationally mimic human thinking and action using mathematical theories or models. The progression of deep learning dramatically boosts the economic growth in the market of autonomous vehicles [1], [2], neural language processing [3], medical diagnosis [4], and object detection [5], [6] etc. Brainware, or brain-inspired, processor attempts to deal with complex tasks by embedding large-scale deep learning models into a compact chip that takes advantage of logic gates to emulate the function of brain neurons and synapses. Recently, researchers have been engaged in the demanding work of building brainware computing system [7]–[10]. An in-hardware training chip has been fabricated and carried out a demonstration for data classification, which exhibits a noticeable reduction of power dissipation and latency [11]. Pei *et al.* has proposed a hybrid Tianjic chip architecture

The associate editor coordinating the review of this manuscript and approving it for publication was Fanbiao Li.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

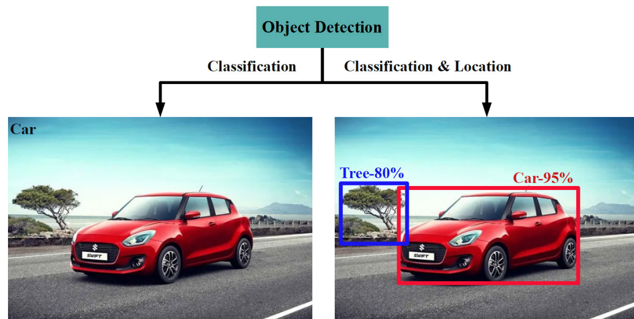


FIGURE 1. Types of object detection.

that consists of multiple cores, reconfigurable building blocks to achieve the precise control and real-time object detection of unmanned bicycle [12]. The brainware processor shows a significant promise for accurate and real-time object detection. However, the large-scale architecture of the neural network impedes its development. Therefore, the neural network with a lightweight structure for high-quality object detection is an urgent need for the evolution of brainware processors.

In computer vision, object detection aims to discriminate instances of the object from given categories in the database or determine object coordinates within an image [13]. As shown in Fig. 1, the object detection has two types, single object recognition and multi-object detection. The implementation of object detection involves feature extraction and classification [14]. In lieu of using conventional feature extraction approaches such as scale-invariant feature transform (SIFT) descriptor [15] and histogram of oriented gradients (HOG) [16], deep neural networks extracts the object features with convolution operation. Meanwhile, the target features are attained, deploying a classifier that locates at the end of the neural network.

In the early stage, the object category is determined by a pre-trained network model, which only affords a single label for the test image or frame. A typical example is the usage of trained neural networks to recognize images in the ImageNet database [17]. The objective of multi-object detection is to discover all the possible objects and the corresponding location in the image. The objects with high probabilities (over the pre-defined threshold) are affirmed as targets. The multi-object detection can be realized by single-stage and two-stage detectors. The single-stage detector omits the search of region proposal that is the first step in the two-stage detector. Region-based convolutional neural networks (R-CNNs) [18], as a two-stage detector, takes advantage of selective search methods to extract the region's proposal of interests which are used for the inputs of object feature classification (second stage). It takes a large amount of time to train the neural networks because the interested proposals are considered as the stand-alone image for the input of the neural network. To speed up the R-CNNs, the faster R-CNNs inputs the entire image to the neural network, and the interested proposals are selected from the feature maps of image [19]. However, the selective search method utilized in the region proposals of both R-CNNs and fast R-CNNs has the issue of time-consuming. To solve the bottleneck of proposal selection, the region proposal network (RPN) has been proposed to learn the interested proposals directly in faster R-CNNs [20]. The R-CNNs decompose the detection process into two stages: bounding box regression (first stage) and anchors classification (second stage). Instead of using the two-stage detector, the single-stage detector directly predicts the coordinates of bounding boxes and the class probability. The single-shot detector (SSD) adopts a pyramidal feature hierarchy to detect the coordinates of bounding boxes and the class probability simultaneously [21]. However, the dimension of feature maps is reduced by increasing the depth of the neural network, which degrades the detection accuracy with low spatial resolution features. The you only look once (YOLO) algorithm uses the concatenated feature maps in different scales to the bounding box of the object, which allows the feature maps to be fully utilized. YOLO3 is the state-of-the-art fastest real-time object detection system, which has received wide attention in academics and industry. Likewise, the YOLO3 employs deep convolutional layers (53 layers) to perform precise object detection, which is prone to be challenging for the design of compact brainware processor.

The tiny YOLO3, as the lightweight version of YOLO3, is one of the best alternatives for the design of compact brainware processor for the sake of its lightweight network architecture and high-precision detection rate. The tiny YOLO3 algorithm received extensive attention and is widely used in the field of object detection. Recently, compressing the network structure and design of tiny YOLO application-specific integrated circuits (ASICs) become indispensable in academics and industry. However, there is no systematic studies concerning the tiny YOLO3 algorithm. It is difficult to develop a compact processor without fully understanding of the mechanism and principle of the tiny YOLO3 algorithm. In this paper, we initially provide a systematic study of tiny YOLO3 algorithm, which will provide a good foundation for algorithm understanding and make a contribution to accelerating the development of compact brainware processor.

II. RELATED WORK

A. YOLO DEVELOPMENT

J. Redmon firstly proposed the YOLO algorithm in May 2016 [22], and it has been evolved to four generations within four years. The base YOLO, motivated by fast R-CNNs, introduces the region-based concept to the neural network. Peculiarly, the input image is divided into different grid cells where two bounding boxes are predicted. In each bounding box, the center coordinate of object, confidence scores and the class probabilities are predicted. The confidence score is responsible for checking whether or not the object exists in the bounding box. The base YOLO has a good advantage in the perspective of speed and acceptable

accuracy, but the following drawbacks retard the development of base YOLO:

- The base YOLO is hard to handle the situation that the distance of two objects is very close. The detector may only predict one object in this condition, which degrades the detector's inference rate.
- Even though each grid cell predicts two bounding boxes, the predicted results are only for one category of objects. Hence, it cannot provide correct results if two objects locate in the same grid cell.
- The base YOLO adopts a fully-connected layer to output the predictions, which requires the inputs of the fully-connected layer to possess the same dimension.

To address the shortcomings of the base YOLO algorithm, YOLO2 absorbs a wide variety of essences from other algorithms to make itself better, faster and stronger [23]. The YOLO2 has the capability to detect over 9000 object categories, while the number of detection categories is only twenty in the base YOLO. Furthermore, the YOLO2 proposes a new classification model (Darknet-19), which consists of nineteen convolutional layers and five pooling layers. With the PASCAL VOC2007 dataset, the mean average precision (mAP) of YOLO2 increases from 63.4 to 78.6 and the processing speed reduces from 45 frames per second (FPS) to 10 FPS in contrast to the based YOLO algorithm. The substantial enhancement of YOLO2 is the introduction of batch normalization and the anchor box. The following items list the improvement of YOLO2,

- Adding batch normalization after each convolution operation can speed up the convergence of neural network training and eliminate the need to adjust parameters manually.
- Instead of predicting bounding boxes with fully-connected layers, the YOLO2 utilizes the anchor boxes to predict the parameters of bounding boxes. Using anchor boxes increases the number of predicted bounding boxes in each grid cell, but fewer mAP is decreased. Moreover, the dimension of anchor boxes is determined by taking advantage of the K-means clustering algorithm, which can shorten the training time of the neural network.

YOLO2 made a significant change on base YOLO, and its performance was greatly enhanced. However, the performance of YOLO2 is still restrained by the following deficiencies:

- It is assumed that the detected object only has a single label, but one object may belong to multiple groups. For example, a car may be labelled as "car" or "vehicle", but the "softmax" function in YOLO2 only issues one label for the detected object.
- In the case of small objects, the prediction accuracy of YOLO2 is relatively low and needs to be improved.

In 2018, the upgraded YOLO, YOLO3, was proposed, and some new ideas were added based on YOLO2 [24]. Compared to the Darknet-19, the YOLO3 takes advantage of 53 convolution layers (Darknet-53) to deepen the network structure, which also inserts the residual block to the network [25]. Instead of using the "softmax" function, the logistic function is introduced for multiple label predictions. Moreover, the significant achievement of YOLO3 is the multi-scale prediction, which improves the algorithm's ability to predict small objects. On the basis of YOLO3, YOLO4 integrates some novel technology such as weighted residual connections, cross stage partial connections, and cross mini-batch normalization etc. to improve the speed and accuracy of object detection, which is published in 2020 [26].

B. HARDWARE ACCELERATION OF YOLO

Although YOLO3 has the advantages of high precision and fast detection speed, it is challenging to transplant the full algorithm to the field programming gate array (FPGA) or ASICs owing to its large memory and gate utilization. The tiny YOLO3, as the lightweight version of YOLO3, uses less convolutional layers but shows receptible detection accuracy. Meanwhile, the weight parameters of tiny YOLO3 is reduced around 10× comparing with YOLO3 (from 237 MB to 33.8 MB), which made the hardware acceleration of YOLO algorithm to be practicable.

Many studies have been published for the hardware acceleration of YOLO algorithm with FPGA. In the reference of [27], an FPGA-based lightweight YOLO2 utilizing the binarized weight and support vector regression has been demonstrated to achieve object classification and localization. The detection speed of lightweight YOLO2 with the ZCU102 evaluation board (Xilinx Inc. California, United States) is up to 40.81 FPS. Another FPGA-based tiny YOLO2 implementation was proposed in literature [28], which adopts 16-bit fixed-point data to compress the YOLO model. The peak throughput of 21 Giga operations per second (GOPs) is attained with 100 MHz frequency. In the literature of [29], the authors proposed a Tera-OPS streaming structure to accelerate the YOLO algorithm, and it achieves a throughput of 1.877 Tera operations per second (TOPs) using binarized weight and fully paralleled convolutional layer. A parameterized FPGA-tailored architecture was proposed for low-latency detection with tiny YOLO3 in [30], which has 1.88 FPS frame rate and 10.45 GOPs throughput with the low-cost FPGA evaluation board. However, the FPGA-based neural network suffers from large power consumption [31] comparing to ASICs. A great deal of complementary metal-oxide-semiconductor (CMOS) based accelerators have been proposed such as Origami [32], Eyeriss [33], iFPNA [34] etc. Moreover, with the development of magnetic tunnel junction (MTJ) technology, the hybrid CMOS/MTJ based logic circuit can significantly decrease the power consumption of brainware processor [8], [35], [36]. The CMOS or hybrid CMOS/MTJ based brainware processors will play a major role in object detection. The processor embedded with tiny YOLO3 will be the state-of-the-art most fast and accurate detection algorithm, which has a great demand in the market. However, the lack of understanding of the tiny YOLO3 will

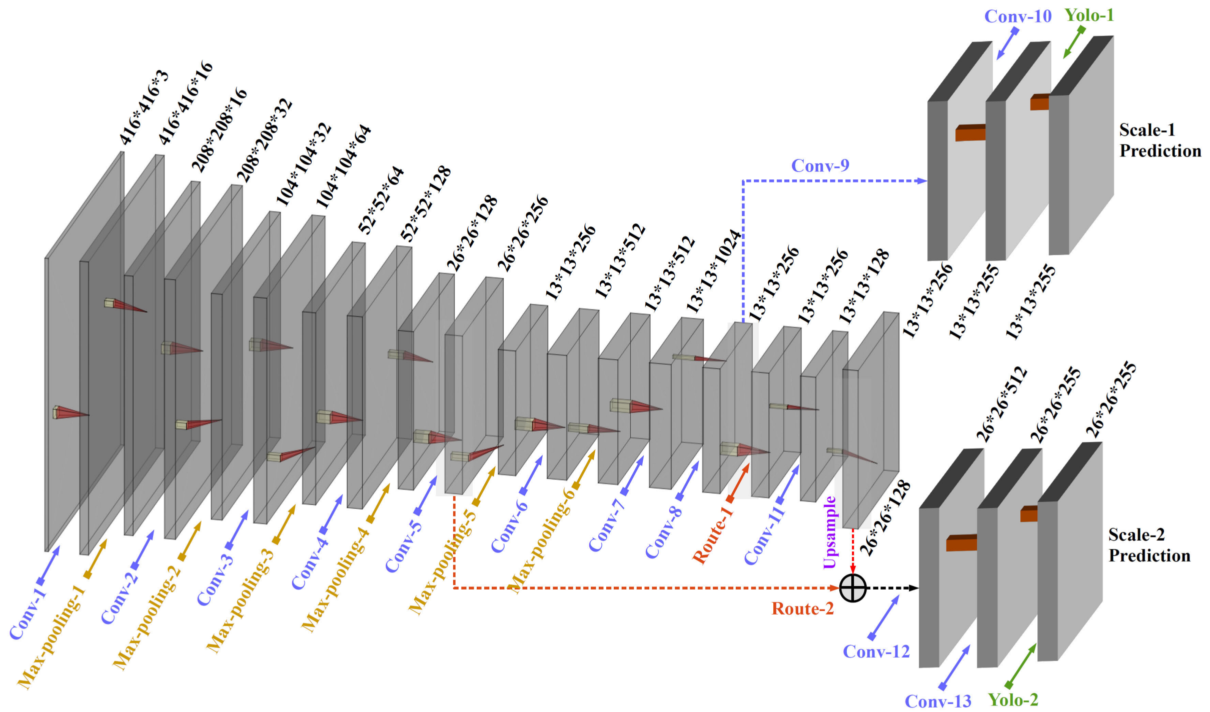


FIGURE 2. Network structure of tiny YOLO3. It consists of 13 convolution layers, 6 max-pooling layers, 2 route layers, 1 upsampling layer, and 2 YOLO layers.

retard the enhancement of brainware processor. Therefore, this paper presents a comprehensive study of tiny YOLO3 for accurate object detection. The key contributions of this paper are summarized as follows,

- A discussion of the development of the YOLO algorithm and its hardware acceleration has been provided.
- The systematic study and analysis of tiny YOLO3 have been presented.
- Challenges and open solutions from algorithm, hardware and emerging semiconductor technology level have been proposed for the design of the compact brainware processor.
- The techniques of approximate computing and approximate circuits are proposed for condensing the tiny YOLO3 algorithm.

The rest of this paper is organized as follows. Section III gives a full review of tiny YOLO3, which includes the data pre-processing and post-processing, implementation details of each layer. Section IV and Section V illustrate the metrics challenges, and open solutions for the design of the compact brainware processor. Finally, the summary are presented in Section VI.

III. STRUCTURE OF TINY YOLO3

The structure of tiny YOLO3 is defined in the configuration file ("cfg" file, refer to Appendix A), and the start point of each layer is defined by the symbol [•]. As shown in Fig. 2, the tiny YOLO3 consists of six kinds of layers, totally 24 layers: *net*, *convolutional*, *maxpool*, *yolo*, *route*, and *upsample* layers. *net* layer configures parameters of the entire network. The remainder of this section will thoroughly describe the network structure of tiny YOLO3 based on distinct layers.

A. CONVOLUTION LAYER

The convolution layer has three modules: convolution operation, batch normalization and activation function. Moreover, the convolution operation includes the feature map conversion and general matrix multiplication. The following subsection will give a detailed description of the convolution layers.

a: CONVOLUTION

Mathematically, an image is described as three dimensions: width, height, and depth (or channels). The convolution operation between images and filters (or kernels) enables to extract effective information for object detection. The prerequisite of convolution operation is that the depth of filters and input image arrays should keep consistency. By scanning the image array utilizing a filter array that has a fixed stride (S), the output dimension of feature maps is defined as following expressions.

$$h_o = (h + 2 \times P - f_h)/S + 1 \quad (1)$$
$$w_o = (w + 2 \times P - f_w)/S + 1 \quad (2)$$

where h_o and w_o are the output height and width of convolution. h and w are the height and width of the input image array, respectively. P is the number of zero-padding. f_h and f_w are the height and width of filter, accordingly. All of the

filters in tiny YOLO3 are 3 × 3 arrays ($f_w = f_h = 3$) with depth f_d that equals to the depth dimension of previous feature map. The output depth of convolution operation is identical to the number of filters. Taking the second convolution layer of tiny YOLO3 as example (refer to Fig. 2), the dimension of convolution result between input image array ($208 \times 208 \times 16$) and 32 filters ($3 \times 3 \times 16$) is $208 \times 208 \times 32$ ($S = 1, P = 1$).

The convolution operation of tiny YOLO3 can be achieved by the following two steps: (1) feature matrix conversion; (2) general matrix multiplication (GEMM). The "img2col" and "gemm" are the main functions for the implementation of feature matrix conversion and GEMM, respectively. Theoretically, a memory stores 2-dimensional array in rows and columns, but the memory address of the computer is linearly ordered. As shown in Fig. 3, a $4 \times 4 \times 3$ image array is stretched in memory serially as a 1-dimensional array with 48 elements, and the address of image array in memory is continuously connected channel by channel.

b: FEATURE MATRIX CONVERSION

The purpose of feature matrix conversion is to achieve point-to-point convolution between image arrays and filters. The feature matrix conversion adopts the image arrays with zero-padding in which the zeros are inserted to the boarders of each channels (refer to Fig. 3). Appendix B briefly illustrates the pseudo codes of zero-padding (*zeroPadding*). At first, the image array is converted to a feature matrix (1-dimensional array) that involves all the elements scanned by the filter windows. The output row of feature matrix conversion is the number of elements in filters (f_n), which is calculated by the following equation,

$$f_n = f_w \times f_h \times f_d \tag{3}$$

Meanwhile, the output column of feature matrix conversion (f_c) is defined as follows,

$$f_c = h_o \times w_o \tag{4}$$

In summary, the output of feature matrix conversion is a matrix with f_n rows and f_c columns, and the matrix locates in the memory as a 1-dimensional array ($f_n \times f_c$).

By scanning the one-dimensional image array, tiny YOLO3 reorganizes the array elements row-by-row instead of column-by-column. Specifically, the first w_o elements are placed on the first row of output matrix, and then the second w_o elements picked up by striding the image array are placed after the first w_o elements. It totally has $h_o \times w_o$ elements in the first row of the output matrix. As shown in Fig. 3, a matrix with 27 ($3 \times 3 \times 3$) rows and 16 columns (4×4) is achieved by converting the $4 \times 4 \times 3$ image array with $3 \times 3 \times 3$ filter. Likewise, the feature map of tiny YOLO3 ($208 \times 208 \times 16$) passing into the second convolution layer is converted to $3 \times 3 \times 16$ rows and 208×208 columns matrix with *im2col* function as well. And so forth, all input matrices and feature maps of neural network are converted into a large matrix that is stored in the computer's memory as a one-dimensional array. Algorithm 1 (*img2col*) shows the pseudo codes of feature matrix conversion.

Algorithm 1 Feature Matrix Conversion of Input Image Array

```
Inputs: Image array with zero-padding (img_pad_out), rows
        and columns of image array (img_rows, img_cols),
        rows, columns and depth of filter (filter_rows,
        filter_cols, filter_depth), paddings, stridding.
Outputs: Results of feature matrix conversion (img_out)
Function img2col(img_pad_out, img_rows, img_cols,
filter_rows, filter_cols, filter_depth, paddings, stridding):
    h_o = (img_rows + 2 * paddings − filter_rows)/stridding + 1; w_o = (img_cols +
    2 * paddings − filter_cols)/stridding + 1;
    f_n = filter_rows * filter_cols * filter_depth;
    for c ∈ f_n do
        w_offset = c % filter_rows; h_offset =
        (c / filter_cols) % filter_cols; c_index =
        c / filter_rows / filter_cols; for h ∈ h_o do
            for w ∈ w_o do
                row_index = h_offset + h * stridding;
                col_index = w_offset + w * stridding;
                out_index = (c * h_o + h) * w_o + w;
                in_index = (img_rows + 2 * paddings) *
                (img_cols + 2 * paddings) * c_index +
                row_index * (img_rows + 2 * paddings) +
                col_index; img_out[out_index] =
                img_pad_out[in_index];
    return img_out
```

c: GEMM

Feature matrix conversion aims to assure that the convolution operation can be accomplished using the GEMM of the basic linear algebra subprograms (BLAS) standard. The GEMM is defined as following equation [37],

$$C = \alpha \times A \times B + \beta \times C \tag{5}$$

where A ($N \times f_n$) and B ($f_n \times f_c$) are the filter and input image array, respectively. N is the number of filters. C represents the convolution result. α and β are the constant scalars. Expanding Eq. 5, the GEMM calculation of tiny YOLO3 is rearranged as following equation,

$$C_{ij} = \sum_{i=0}^{f_d-1}\sum_{k=0}^{f_n-1}\sum_{j=0}^{f_c-1} \alpha \times A_{ik} \times B_{kj} + \sum_{i=0}^{f_d-1}\sum_{j=0}^{f_c-1} \beta \times C_{ij} \tag{6}$$

where the constant scalars $\alpha = \beta = 1$. Substituting the constant scalars to Eq. 6, the convolution operation is reorganized as follows,

$$C_{ij} = \sum_{i=0}^{f_d-1}\sum_{k=0}^{f_n-1}\sum_{j=0}^{f_c-1} A_{ik} \times B_{kj} \tag{7}$$

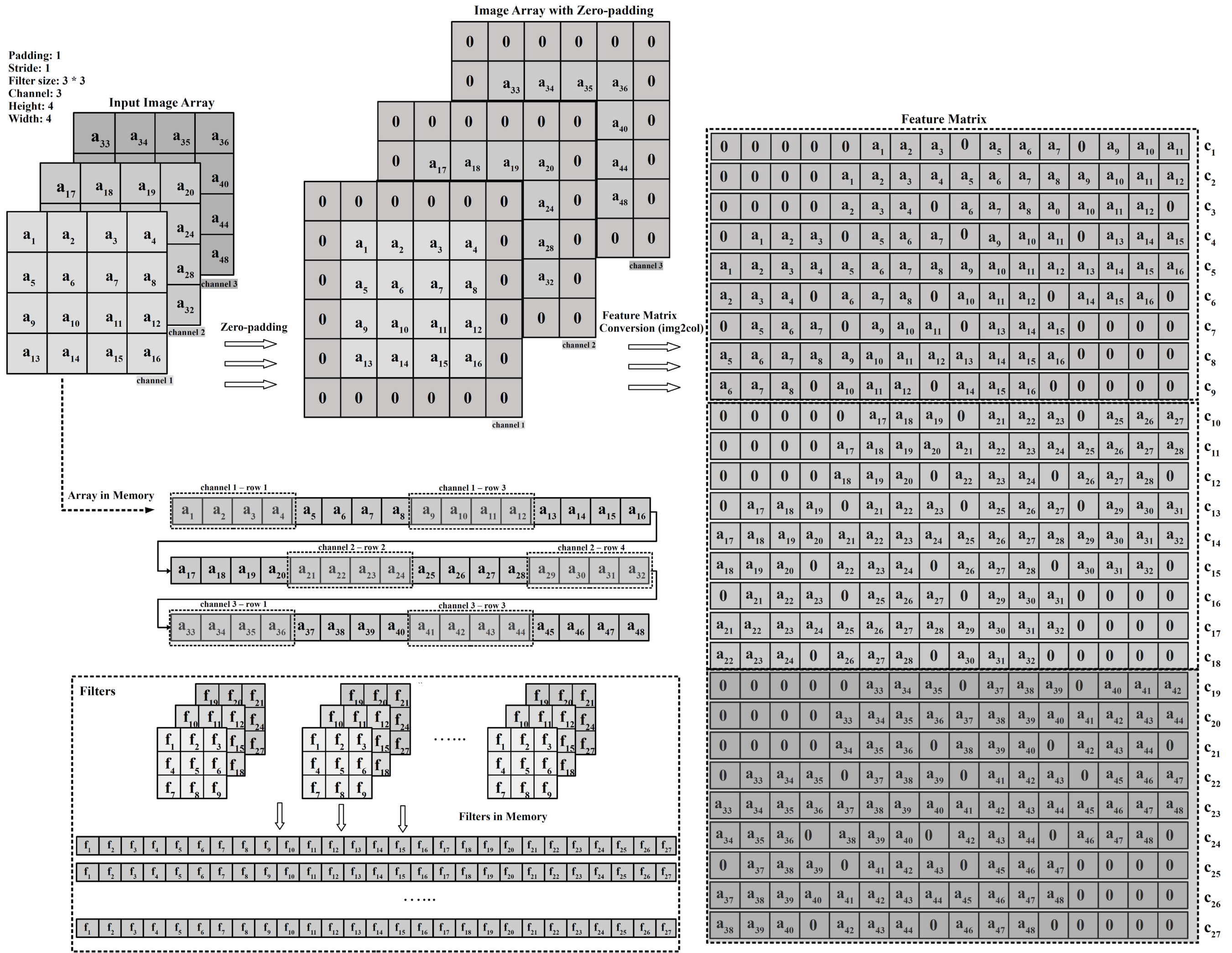


FIGURE 3. Feature matrix conversion and array arrangement in memory. The elements in array are stored one-by-one. The input image array is expanded with zero-padding, and then converted to feature matrix for convolution operation.

As shown in Fig. 4, the GEMM is implemented with two steps: one-by-one multiplication between filter parameters and feature matrix elements and sum of multiplication results. More specifically, the first element in each filter multiplies to the first row of the element in feature matrix and the multiplication result stores in the f_c address of memory. After the multiplication is completed between the second element and the element in the second row, the calculation results between the first and the second elements are summed and stored in memory. The convolution of the first filter ends until the multiplication of the last element in the filter, and the last row in the feature matrix is completed. The above operation is repeated until the convolution of the N filter is finished, and $N \times f_c$ ($N \times h_o \times w_o$) elements are achieved eventually.

Tab. 1 summarizes the number of parameters and the inputs or outputs dimension in each convolution layer. The first five parameters within the weight file of tiny YOLO3 are not filters' parameters. Although the binary file of tiny YOLO3 weight includes 8858739 parameters, the 8858734 (8845488 + 9552 + 3694) parameters are used for convolution operation and batch normalization. As expressed in Tab. 1, the tiny YOLO3 has a total of 13 convolutional layers, and the 1×1 convolution is used in the 10th and 13th layers where the batch normalization is not included. Fig. 5 shows the format of weights arranged in memory. The convolution parameters are stored in the order of biases, scales, means, variance and filter weights. The scales, means and variance (with the same numbers in each layer) are used for batch normalization that will be discussed in the following subsection. The filter weights are utilized for the convolution operation with GEMM, and the dimension of filter weights is $N \times f_c$.

d: BATCH NORMALIZATION

The batch normalization (BN) layer aims to ensure that the input data in each layer of the neural network has a similar

TABLE 1. Number of parameters in convolution layer.

Convolution	Filters	Inputs	Outputs	BFL	Weights	Batch Normalization	Biases
Layer 1	16	416×416×3	416×416×16	0.150	16×3×3×3 (432)	16×3 (48)	16
Layer 2	32	208×208×16	208×208×32	0.399	32×3×3×16 (4608)	32×3 (96)	32
Layer 3	64	104×104×32	104×104×64	0.399	64×3×3×32 (18432)	64×3 (192)	64
Layer 4	128	52×52×64	52×52×128	0.399	128×3×3×64 (73728)	128×3 (384)	128
Layer 5	256	26×26×128	26×26×256	0.399	256×3×3×128 (294912)	256×3 (768)	256
Layer 6	512	13×13×256	13×13×512	0.399	512×3×3×256 (1179648)	512×3 (1536)	512
Layer 7	1024	13×13×512	13×13×1024	1.595	1024×3×3×512 (4718592)	1024×3 (3072)	1024
Layer 8	256	13×13×1024	13×13×256	0.089	256×1×1×1024 (262144)	256×3 (768)	256
Layer 9	512	13×13×256	13×13×512	0.399	512×3×3×256 (1179648)	512×3 (1536)	512
Layer 10	255	13×13×512	13×13×255	0.044	255×1×1×512 (130560)	–	255
Layer 11	128	13×13×256	13×13×128	0.011	128×1×1×256 (32768)	128×3 (384)	128
Layer 12	256	26×26×384	26×26×256	1.196	256×3×3×384 (884736)	256×3 (768)	256
Layer 13	255	26×26×256	26×26×255	0.088	255×1×1×256 (65280)	–	255
Total	3694	–	–	–	8845488	9552	3694

Note: BFL – Billion floating-point operations per second, and symbol "–" denotes the item does not exist.

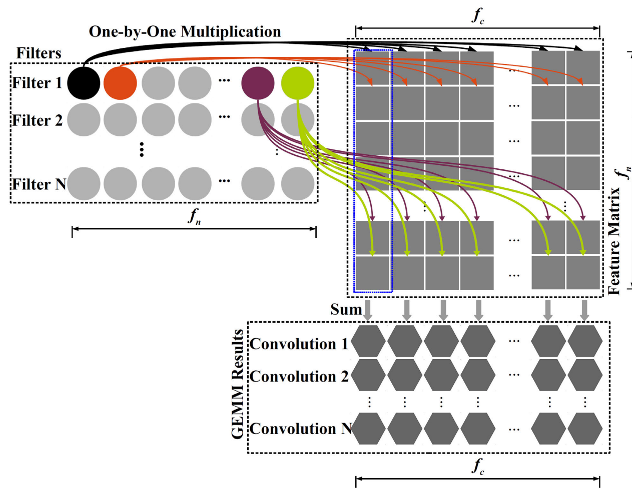


FIGURE 4. Convolution operation with GEMM. The dimension of filters and feature matrix are $N \times f_n$ and $f_n \times f_c$, respectively.

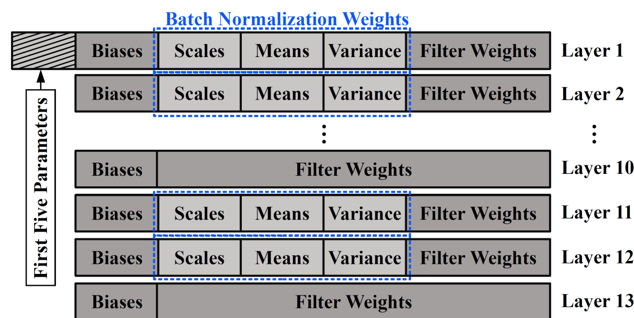


FIGURE 5. Weights arrangement of convolutional layers in memory. The details about the number of convolution parameters are illustrated in Tab. 1. The weight size of tiny YOLO3 is 33.8 Megabytes (8858734 × 32 / 8 / 1024 / 1024) with 32-bit floating-point format.

distribution (zero mean and unit variance). Since the BN layer locates between the convolution layer and the activation layer, the input of the batch layer comes from the output of the convolution layer. The normalization of convolution result ($\hat{C}_{ij}$) is calculated by the following equation,

$$\hat{C}_{ij} = \frac{C_{ij} - E[C_{ij}]}{\sqrt{Var[C_{ij}]}} \tag{8}$$

where the symbols of $E[\bullet]$ and $Var[\bullet]$ represent the expectation and variance operation, which are typically estimated using the mean and variance of a mini-batch with m elements, $\mu_{\mathcal{B}}$, and $\sigma^2_{\mathcal{B}}$. The batch normalization is formulated as follows,

$$\hat{C}_{ij} = \frac{C_{ij} - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{9}$$

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{p=1}^{m} C_{ij}^{(p)} \tag{10}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{p=1}^{m}\left(C_{ij}^{(p)} - \mu_{\mathcal{B}}\right) \tag{11}$$

where ϵ is a constant value (0.000001f) that is selected to avoid dividing by zero. In theory, $\sigma^2_{\mathcal{B}}$ is sample variance of mini-batch dataset. However, the test sample typically only has one batch during the inference period, so the batch normalization takes advantage of population variance and mean for the inference implementation. The sample variance is an unbiased estimator of the population variance, and the transformation relationship between them is expressed as follows,

$$\sigma^2 = E\left[\frac{\sum_{p=1}^{m}\left(C_{ij}^{(p)} - \mu_{\mathcal{B}}\right)}{m-1}\right] \tag{12}$$

where σ^2 and $\sigma^2_{\mathcal{B}}$ are population variance and sample variance, respectively. The expectation of sample variance can be obtained by reorganizing Eq. 11,

$$E\left[\sigma^2_{\mathcal{B}}\right] = E\left[\frac{1}{m}\sum_{p=1}^{m}\left(C_{ij}^{(p)} - \mu_{\mathcal{B}}\right)\right] \tag{13}$$

The above equation can be rewritten as follows,

$$E\left[\sum_{p=1}^{m}\left(C_{ij}^{(p)}-\mu_{\mathcal{B}}\right)\right]=m\times E\left[\sigma_{\mathcal{B}}^{2}\right] \tag{14}$$

Substituting Eq. 14 to Eq. 12, the population variance can be attained as follows,

$$\sigma^{2}=\frac{m}{m-1}\times E\left[\sigma_{\mathcal{B}}^{2}\right] \tag{15}$$

Meanwhile, the population mean (μ) can be estimated using the sample mean in the mini-batch, that is, $\mu=\mu_{\mathcal{B}}$. However, the normalization calculation to the data with zero mean and unit variance distribution in Eq. 9 reduces the range of data representation. As an illustration, the sigmoid activation function only works in the linear region if the inputs are zero mean. Adding the scale (γ) and shift (β) parameters to the model is an achievable solution to solve the issues mentioned above. By referencing Eq. 9 and Eq. 15, the output of batch normalization ($O_{ij}^{\mathcal{B}}$) can be represented as following equation,

$$\begin{aligned}O_{ij}^{\mathcal{B}}&=\gamma\times\left(\frac{C_{ij}-\mu}{\sqrt{\sigma^{2}+\epsilon}}\right)+\beta\\&=\frac{\gamma}{\sqrt{\sigma^{2}+\epsilon}}\times C_{ij}+\left(\beta-\frac{\gamma\times\mu}{\sqrt{\sigma^{2}+\epsilon}}\right)\end{aligned} \tag{16}$$

Similar to weight training, the γ and β are pre-trained parameters. Furthermore, the μ and σ are also pre-calculated parameters for inference in the period of training. Therefore, four more parameters in each filter are loaded for forward propagation of tiny YOLO3. As illustrated in Eq. 16, since the batch normalization is a linear transformation of the results in convolutional layer, the output dimension of batch normalization is the identical to the outputs in convolutional layer ($N\times h_o\times w_o$).

e: ACTIVATION FUNCTION

The activation function intends to bring in the non-linearity to the neural network, which enables the model to deal with complex conditions. The leaky rectified linear unit (ReLu) is utilized to fire the neurons of the neural network in tiny YOLO3. The activation function locates behind the batch normalization, that is, the inputs of activation function are the outputs of batch normalization. The leaky ReLU is defined as follows,

$$O_{ij}^{\mathcal{A}}=\begin{cases}0.1\times O_{ij}^{\mathcal{B}} & if\ O_{ij}^{\mathcal{B}}<0\\ O_{ij}^{\mathcal{B}} & if\ O_{ij}^{\mathcal{B}}\geqslant 0\end{cases} \tag{17}$$

where $O_{ij}^{\mathcal{A}}$ is the output of leaky ReLU. The above equation can be performed using C language code as "$O_{ij}^{\mathcal{A}}=O_{ij}^{\mathcal{B}}\ ?\ O_{ij}^{\mathcal{B}}\ :0.1*O_{ij}^{\mathcal{B}}$".

B. MAX-POOLING LAYER

The feature position of input images is precisely stored in the outputs of ReLU, which results in that the feature maps are susceptible to the rotation or translation of input images. The max-pooling, a downsampling strategy, can not only reduce the dimensionality of the feature map but also make the network structure more stable and robust. In other word, with the max-pooling method, the feature map's susceptibility to rotation or translation will be well addressed. The largest elements within the filter sub-region are the output of max-pooling, which is written as follows,

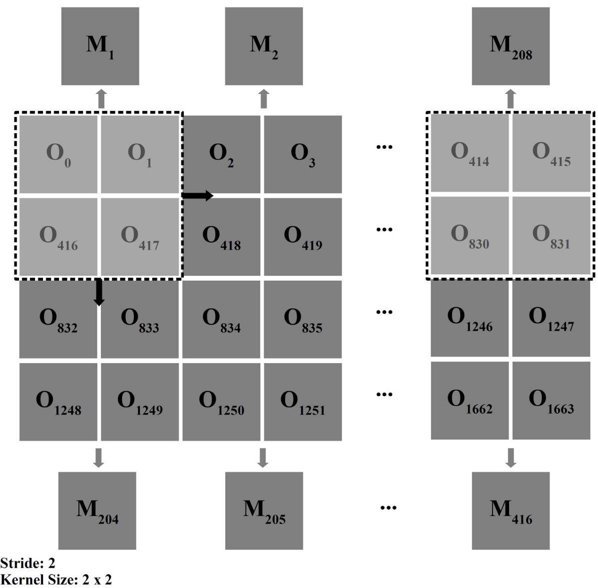


FIGURE 6. Max-pooling of tiny YOLO3.

$$M^{\mathcal{P}}=max\left(O^{\mathcal{A}}\right) \tag{18}$$

where $M^{\mathcal{P}}$ and $O^{\mathcal{A}}$ are the outputs and inputs of max-pooling, respectively. $max(\bullet)$ indicates the maximization operation. To make the expression easier, the superscript of max-pooling input ($\mathcal{A}$) is omitted. As shown in Fig. 6, the subscript value represents the address index of the corresponding element.

In tiny YOLO3, the stride of max-pooling is 2, and the filter size is 2×2. The max-pooling is achieved by the comparison of the four values in the filter window by traversing, which is calculated by following expressions,

$$max(O_0,-FLT_MAX)\Longrightarrow T_1 \tag{19}$$
$$max(T_1,O_1)\Longrightarrow T_2 \tag{20}$$
$$max(T_2,O_{416})\Longrightarrow T_3 \tag{21}$$
$$max(T_3,O_{417})\Longrightarrow M_1 \tag{22}$$

where FLT_MAX is the maximum floating-point number. T_1, T_2, and T_3 are the intermediate variables during calculations. The first max-pooling is accomplished by using the above equations. Fig. 6 only gives two rows of feature maps after the first ReLU outputs. The dimension of max-pooling relies on the value of stride, which can be evaluated as $N\times h_o/S\times w_o/S$. The detailed dimension of max-pooling output is listed in Fig. 2. In total, the entire architecture of tiny YOLO3 has six max-pooling operations.

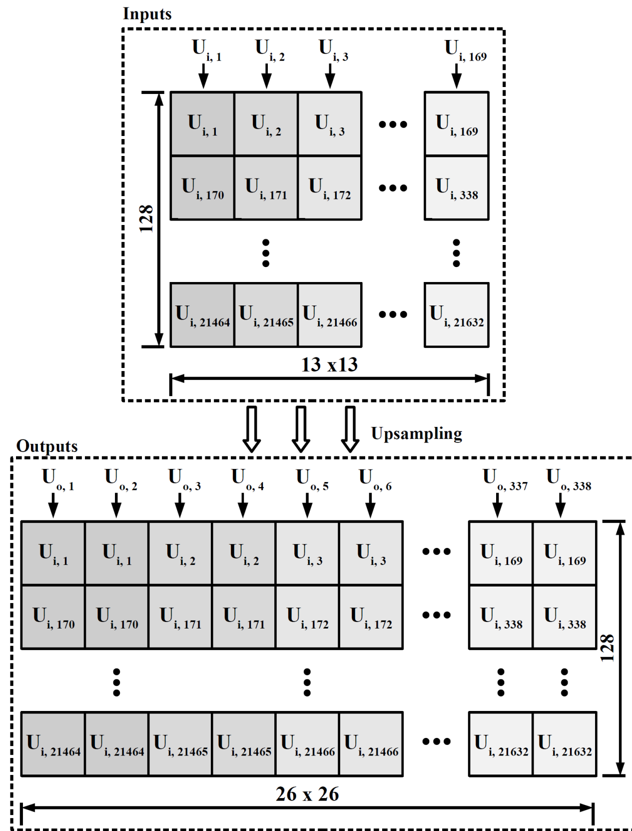


FIGURE 7. Upsampling results of tiny YOLO3.

C. UPSAMPLING LAYER

The upsampling layer attempts to convert the image from low resolution to high resolution, which enables YOLO's prediction to be implemented on another scale. In tiny YOLO3 the nearest-neighbor interpolation approach is utilized for upsampling, which is defined as follows,

$$U_o \longleftarrow \xi \times U_i \tag{23}$$

where ξ is the constant scale, $\xi = 1$. U_o and U_i are the outputs and inputs of upsampling layer. As shown in Fig. 7, the implementation of upsampling is attained by inserting a copy of the input vector before the next input vector. The stride of upsampling is 2 as well, which extends the inputs dimension from $128 \times 13 \times 13$ to $128 \times 26 \times 26$.

D. ROUTE LAYER

The route layer concatenates data from other layers into the feature map, which provides more valuable information for subsequent prediction. Two route layers are used in tiny YOLO3. The first route layer copy the output features from the 8th convolutional layer (Conv-8 in Fig. 2), and the second route layer concatenates the outputs from the 5th convolution layer (Conv-5) and upsampling layer. Specifically, the result of route layer is the union of different layers, which is defined by following expression,

$$R_o = R_1 \cup R_2 \cup \cdots R_n \tag{24}$$

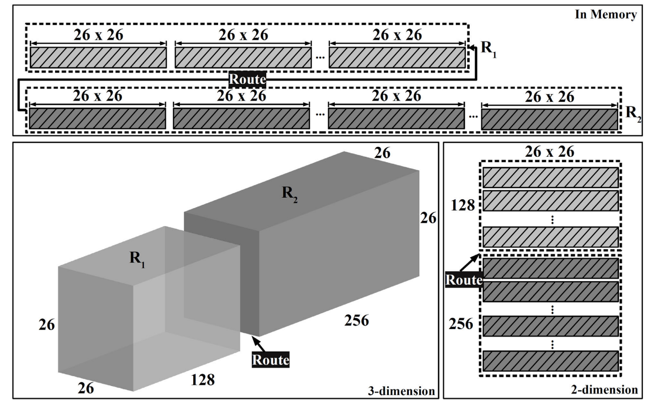


FIGURE 8. Schematic of route in tiny YOLO3.

where R_o and $R_{1,2,3,\cdots,n}$ are the output and input of route layer, respectively. Besides, it requires that the width and height of inputs in route layers should keep the same. The key point of the concatenation is to copy elements from interested addresses to the output address of the route layer. Fig. 8 illustrates the concatenation details between $26 \times 26 \times 128$ and $26 \times 26 \times 256$ feature maps, and the data can be represented in three different formats: 3-dimension data, 2-dimension data, and 1-dimension data stored in memory. The concatenation in 2-dimension or 3-dimension is implemented along the depth direction. In computer's memory, the address of route output sequentially copies the elements from each input layer to generate a new feature maps. Benefiting from the information fusion of different layers, the tiny YOLO3 exhibits great capability for small objects detection.

E. YOLO LAYER

The coordinates of bounding boxes and the probabilities of objects are affirmed in the "yolo" layer. The fully-connected layer in previous version of YOLO is superseded by the "yolo" layer. The rest of this subsection will provide a detailed description of the "yolo" layer. As shown in Fig. 2, the 10th convolution layer (16th layer in "cfg" file) attempts to predict the object coordinates using 255 filters. The dimension of each filter is $1 \times 1 \times 255$ (width $\times$ height $\times$ depth). The neural network predicts 3 anchor boxes for each cell of the input image, and each anchor box consists of 4 relative coordinates of the bounding box, 1 objectness score, and 80 classes (85 parameters totally). Once the relative coordinates (t_x, t_y, t_w, t_h) are confirmed, the center coordinates of the object and the dimension of the bounding box can be evaluated by following equation [24],

$$b_x = \sigma(t_x) + c_x \tag{25}$$

$$b_y = \sigma(t_y) + c_y \tag{26}$$

$$b_w = p_w \times e^{t_w} \tag{27}$$

$$b_h = p_h \times e^{t_h} \tag{28}$$

where b_x and b_y are the center coordinates of predicted box. b_w and b_h are the width and height of

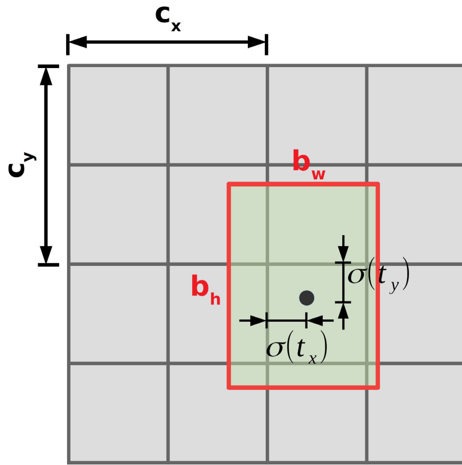


FIGURE 9. Parameters definition of bounding box. Figure adapted from [24].

predicted box, respectively. c_x and c_y are the offset of predicted cell to the top corner of the image. p_w and p_h are dimension of anchors (bounding box prior) which has six predefined settings in tiny YOLO3, [(10, 14), (23, 27), (37, 58), (81, 82), (135, 169), (344, 319)]. Fig. 9 shows the parameters definition of predicted bounding box. The $\sigma[\bullet]$ indicates the logistic function which is written by the following expression,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{29}$$

The objectness score (S_{obj}) and the category probabilities ($Pr_{1,2\cdots,80}$) are predicted utilizing above logistic function as well. Specifically, the 10th convolution layer outputs $255 \times 13 \times 13$ vectors which are considered as the first scale prediction. As shown in Fig. 10, each cell adopts three anchor boxes for prediction, and each column (255 vectors) of feature map includes a prediction of one cell. In other words, the elements in each column are the predicted parameters for the corresponding grid cell. Since t_w and t_h do not require a regression operation, the logistic function is separately implemented twice. The first logistic function handles the first two rows of vectors for each anchor box ($2 \times 13 \times 13$) of center coordinates, while the second function processes the vectors of the fifth to the last row of predicted parameters ($(1 + 80) \times 13 \times 13$). A total number of three loops are needed using the logistic function to traverse all predicted vectors ($255 \times 13 \times 13$). The evaluation results of the logistic function are stored in memory for subsequent processing. Once the parameters are predicted in the "yolo" layer, the target objects can be achieved by post-processing algorithms which consist of bounding box regression, post-processing of bounding box, and non-max suppression (refer to Appendix C).

A concrete example to verify the tiny YOLO3 algorithm is provided in Fig. 11. The dimension of frame captured from a USB camera (See3CAM, e-con Systems, India) is 640×480 pixels, but the input image dimension is resized

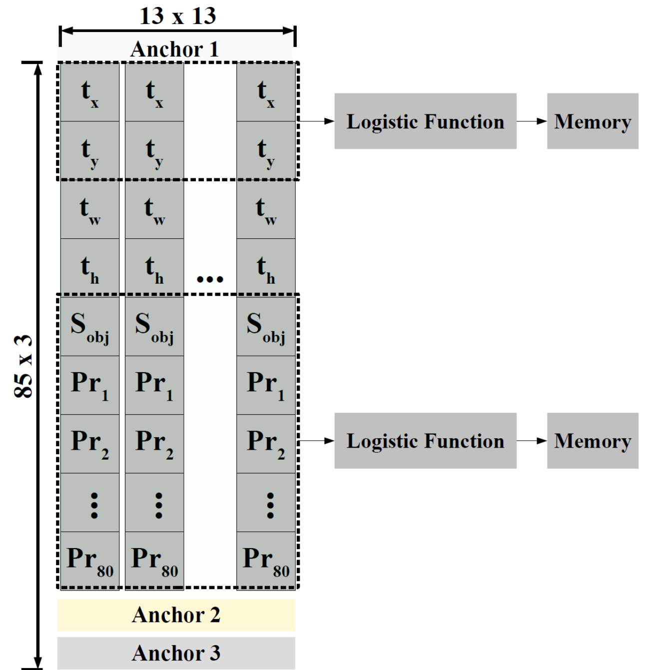


FIGURE 10. Logistic function in "yolo" layer.

to 416×416 pixels (same as the input image in Fig. 2). The demonstration software of tiny YOLO3 algorithm is executed on a CPU with the configuration of Intel Core i76920HQ, 2.9 GHz. As shown in Fig. 11, the processing time of current video frame is 386.2701 milliseconds, and two objects (bottle–62% and person–85%) are recognized in this demonstration.

However, in application of real-time object detection, this demonstration result is undesirable because the processing time is 10x greater than the real-time frame rate, 33 FPS. Therefore, it is imperative to develop a high-speed, energy-efficient compact brainware processor with the tiny YOLO3 algorithm.

IV. METRICS OF COMPACT BRAINWARE PROCESSOR WITH TINY YOLO3

Benefiting from the high accuracy and small network model, the tiny YOLO3 algorithm exhibits excellent potentials for the compact processor design. However, it is still challenging to develop brainware processor with tiny YOLO3 due to the high computation costs and high memory demand. Fortunately, techniques of approximation and compression enable compact brainware processors to be possible. The heuristic evaluation of approximation and compression algorithms can fundamentally analyze the bottleneck that hinders the design of compact brainware processor. The metrics of evaluating compact brainware processor are summarized as follows,

- *Throughput* – It is defined as the rate of production. The throughput for the YOLO-based processor is considered as an inference rate or detection rate. A high throughput indicates that more inferences or detection are produced

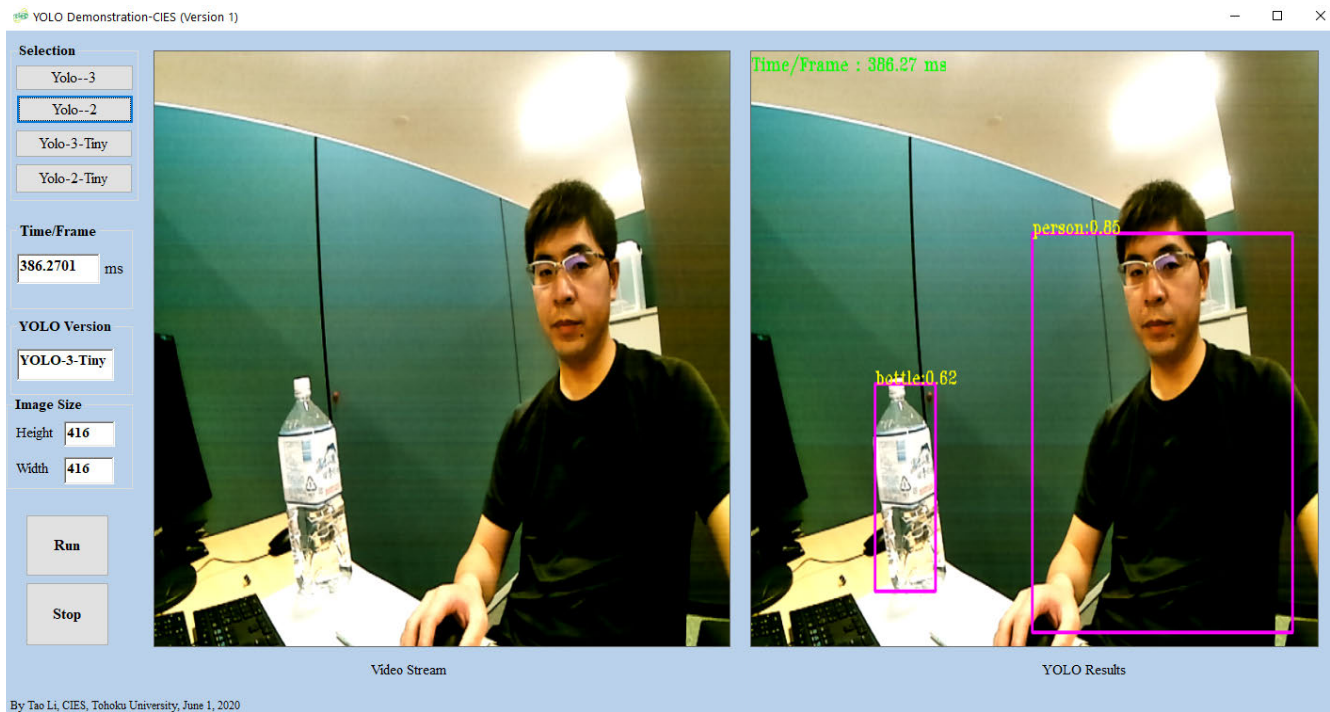


FIGURE 11. Demonstration with tiny YOLO3.

per second, which is measured by operations per second (OPS) or TOPS. In general, the throughput can be improved by increasing the batch size, which also introduces large latency to the neural network.

- *Latency* – It is the time interval (unit: seconds) to attain one inference or detection result. The latency plays a crucial role in evaluating real-time applications such as obstacle avoidance of autonomous driving, real-time navigation of unmanned aerial vehicles (UAVs) etc. A large latency will result in the failure of the real-time task. However, low latency may decrease the throughput, which presents a challenge for the development of brainware processor.
- *Accuracy* – It is a benchmark to evaluate the good or bad of a proposed algorithm. The accuracy is defined by the ratio between the correct inference results and entire testing samples. For algorithm improvement, accuracy is the most important metric. However, the accuracy of the contemporary neural network has exceeded the requirement of brainware processor design. Instead of pursuing high accuracy by increasing computation cost and deploying large memory, it is more necessary to develop high-performance compact brainware processors with low latency and high throughput in practical applications. Therefore, the main task is to realize the design of brainware processor by compressing and approximating the tiny YOLO3 algorithm while preserving acceptable accuracy as much as possible.
- *Energy-efficiency* – It mainly measures the ratio between throughput and power consumption, that is, performance per watt. There are growing demands for energy-efficient processors on the applications of wearable devices, smartphones etc. In addition, a large amount of heat is generated through the growth of power consumption, which also brings higher requirements for the cooling protection of the processor. The power consumption of brainware processors derives mainly from a vast majority of computation logics and on-chip memory or the access between processors and off-chip memory.
- *Cost* – It mainly refers to the development and production costs of a brainware processor, which involving the expenses of algorithm development, hardware design and device fabrication. The market expects low-cost processors, but cheap processors may be developed at the cost of performance degradation.
- *Lifecycle* – It refers to the amount of time that a brainware processor is worked. With the development of emerging semiconductor technologies, most electronic components have a long life cycle, and only long life cycle brainware processors have more opportunities of being selected by engineers or companies.
- *Dimension* – It is the physical size of brainware processor. A small or tiny dimension processor is preferred in some applications because it facilitates the integration of more electronic components in limited space.

V. CHALLENGES AND OPEN SOLUTIONS OF COMPACT BRAINWARE PROCESSOR WITH TINY YOLO3

Designing a compact brainware processor with tiny YOLO3 is a difficult task on account of many challenges that should be addressed. As described in Tab. 1, a total number of 8858734 parameters (33.8 Megabytes) are included in tiny YOLO3. Moreover, the maximum dimension of activation (with 32-bit floating-point) is up to 10.56 Megabytes ($416 \times 416 \times 16 \times 32$ / 8 / 1024 / 1024) within the first convolutional layer. The key challenge is that a considerable amount of memory resources are required to store those weights and activations of tiny YOLO3, which is the main reason to impede the development of compact brainware processor. The huge computation costs in the convolutional operation (refer to Fig. 3 and Fig. 4) slow down the processing speed of inference, which brings large latency to the tiny YOLO3 processor. Moreover, high computational complexity requires a large area of integrated circuit to implement, which also diminishes the throughput of compact brainware processor. Frequent accessing on-chip memory or external memory comes with a power consumption overhead for energy-efficient processors. Therefore, the following challenges need to be tackled to achieve a high-performance brainware processor,

- Memory – Large memory requirement (tensors, parameters, etc.) increases the circuit area and consumes a great deal of power.
- Logic gate – Complicated network architecture and massive convolutional operations need to be supported with a large number of logic gates, but less latency for real-time object detection.

Brainware processors with small memory capacity and fewer logic gates consumption are what industry and academia expect. Therefore, the essence of academic research is to achieve these two requirements with heuristic strategies, such as algorithm or hardware improvement, while ensuring that the detection result is readily acceptable. In the following section, the open solutions for designing compact brainware processors with tiny YOLO3 are proposed from the algorithm, hardware, and emerging semi-conductor technology level. Meanwhile, the challenges and pitfalls by virtual of distinct solutions are also illustrated. Tab. 2 gives a summary of open solutions for the design of compact brainware processor with tiny YOLO3, which will be thoroughly discussed in the remaining part of this section.

A. ALGORITHM LEVEL

There are undoubtedly four ways to achieve less memory and logic gate of compact brainware processor from algorithm level: (1) reducing the number of parameters and activations; (2) minimizing the precision (bit length) of parameters and activations; (3) reduce the number of computation; (4) compress the network model by refining the sophisticated trained model. The first three types of strategies can be achieved using approximated computing approach, while the last item can effectively implemented by knowledge distillation.

TABLE 2. Open solutions for the design of compact brainware processor with tiny YOLO3.

Level	Open Solutions
Algorithm	□ Approximate computing • Weight and activation sparsification • Weight and activation quantization • Loop perforation □ Knowledge distillation
Hardware	□ Approximate adder
Emerging Technology	□ Novel semi-conductor technology

1) APPROXIMATE COMPUTING

It is clear that the inference accuracy of YOLO-based algorithm (from YOLO1 to YOLO4) has been significantly improved since 2016. However, the growth of computation costs and memory demands of YOLO-based algorithm with redundant accuracy become the major challenge to develop brainware processor. The approximate computing is a prominent solution to exploit the intrinsic resilience of neuromorphic applications, which is able to considerably reduce the memory utilization and computation complexity of embedded processor while maintaining acceptable accuracy [38], [39]. The intrinsic resilience means that the applications still achieve tolerable results despite performing in error or approximate manners [40], which is attributed to the following reasons [41],

- The golden results are unachievable, and the human brains are hard to distinguish from the inference results to a certain extent. In other words, the processing results with intensive computation and large memory consumption are probably redundant, which tends to be the same as processing results with the approximate computing approach for human beings.
- The input data of the practical application is usually noisy and redundant. For a neural network, the system deploys different types of samples to explore the optimal model of neural network during training period, which fundamentally makes the neural network to be an approximate computing system. In addition, The utilization of a large amount of redundant data calculation is not necessary for limited perceptual capabilities of human brains.
- The impacts of approximate computing may be neutralized with the aggregation or iterative-refinement calculations in the neural networks. Optimizing the calculation pattern of neural network while employing approximate computing can effectively decrease the amount of computation and storage for the development of brainware processor.

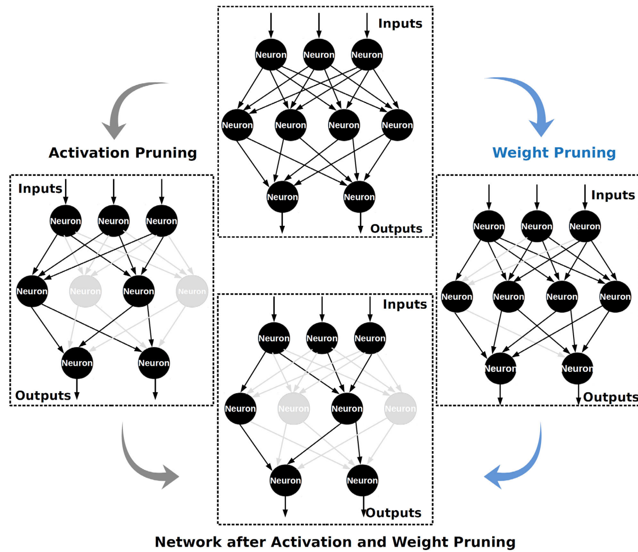


FIGURE 12. Weight and activation pruning.

The following section will propose some possible solutions for the development of the compact brainware processor with approximate computing strategy.

a: WEIGHT AND ACTIVATION SPARSIFICATION

The critical point of sparsification is that the data with a substantial contribution to the detection will be retained, while fewer contributions will be discarded. In general, the complexity of the neural network is reduced by using activation pruning and weight pruning, as shown in Fig. 12. The weight and activation (neuron) are selectively eliminated using weight and activation pruning. The weigh pruning and activation pruning belong to non-structured and structured pruning, respectively. The shortcoming of non-structured pruning is that the produced weight matrix leads to the neural network model with random connections, which reduce the efficiency to access the weights stored in memory [42]. The activation pruning, as a structured pruning method, considerably saves more memories for the storage of intermediate actions and trained weights while significantly reducing the computation resources of convolution operations. The activation pruning works better on reducing the memory utilization and computation cost than the weight pruning because the activation pruning not only removes the neurons but also eliminates the synapses connecting to the corresponding neurons. Weight pruning attempts to remove the weights with zero or small values that depend on the preset threshold.

In the early study, LeCun *et al.* [43] proposed the "Optimal Brain Damage" to selectively remove weights, which firstly deploys the pruning concept to the neural network. Although pruning technology has been studied in the past, it is not paid more attention until the rapid development of deep neural networks and the increasing demand for embedded processors. In the literature of [44], the number of weight is reduced by 13× for the VGG-16 model while with little accuracy loss, which attracts more scholars to focus on the pruning technique after the paper publication. The pruning experienced two steps: firstly, the weights less than the threshold are removed, and the final weights are achieved by retraining the network for fine-tuning in the second step. It is worth mentioning that the weight pruning technique in this algorithm is employed in all layers. A similar pruning strategy can be found in ThiNet [44], which achieves 3.31× FLOPS reduction, 16.63× compression on VGG-16 model with minimal performance loss. Another pruning method, called random pruning, randomly allocates the parameters in different layers, which exhibits poor performance compared to the implementation of pruning in all layers [46]. Srinvivas *et al.* proposed a method to remove the redundant neurons with a trained model [47], which mainly prunes the parameters in densely connected layers. A growing trend to compact the model of neural network is a pruning technique with group sparsity constraints (structured pruning) that attempts to eliminate all filters, channels, or neurons (activation pruning) within the group. By evaluating each channel's saliency, channel pruning simplifies the model of neural network by removing the input and output feature maps connected to the inconsequential channels. Liu. *et al.* proposed a channel pruning method based on the mean gradient, which demonstrates that 5.46× reduction in FLOPS (less than 1% accuracy loss) with CIFAR-10 dataset [48]. The disadvantage of channel pruning is that the accuracy is application-specific; in other words, the neural network is susceptible to the variation of input feature maps. To address these challenges, Gao *et al.* proposed a dynamic channel pruning, feature boosting and suppression (FBS), which dynamically boost and suppress output channels calculated from convolutional layers [49]. Specifically, the convolutional operations are skipped for the unimportant channels that are predicted with the channel saliency predictor. In contrast to channel pruning, these unimportant channels are not discarded in FBS but are skipped. More references for channel pruning techniques can be found in [50]–[52].

In addition, the number of weights can be reduced by the matrix or tensor decomposition. The singular vector decomposition (SVD) is a promising method for weight approximation, which can reduce the weight dimension by matrix factorization. The SVD of weight matrix (W) is defined by the following equation,

$$W_{m\times n} = U_{m\times n} \times \Sigma_{n\times n} \times V^T_{m\times n} \tag{30}$$

where U and V are the unitary matrices, and the corresponding columns of U and V are the right singular vectors and left singular vectors, respectively. Σ is the singular values of W, which is a diagonal matrix. $(\bullet)^T$ represents the matrix transpose. m and n are the row and column of the matrix, respectively. The parameters of the weight matrix can be factorized as follows,

$$U_{m\times n} = \begin{bmatrix} U_{a\times a} & U_{a\times(n-a)} \\ U_{(m-a)\times a} & U_{(m-a)\times(n-a)} \end{bmatrix} \tag{31}$$

$$\Sigma_{n\times n} = \begin{bmatrix} \Sigma_{a\times a} & O \\ O & \Sigma_{(n-a)\times(n-a)} \end{bmatrix} \quad (32)$$

$$V^T_{m\times n} = \begin{bmatrix} V^T_{a\times a} & V^T_{a\times(n-a)} \\ V^T_{(m-a)\times a} & V^T_{(m-a)\times(n-a)} \end{bmatrix} \quad (33)$$

where $a \times a$ is the dimension of the approximated matrix. The singular values in matrix Σ are in descending order, and the weight approximation ($\tilde{W}$) can be obtained by selecting the first a singular values of Σ,

$$\tilde{W}_{a\times a} = U_{a\times a} \times \Sigma_{a\times a} \times V^T_{a\times a} \quad (34)$$

As illustrated in [53], the number of weights with SVD approach is compressed from 103 million to 14.6 million, with only 0.04% accuracy degradation.

Discussion: Reducing the weight and activations of tiny YOLO3 with the sparsification technique is effective for alleviating computation burden and reducing memory utilization. Applying the SVD algorithm to the fully-connect layer rather than the convolution layer works better because it prefers large weight matrices. Since the filter dimensions of tiny YOLO3 are 3×3 and 1×1, the channel pruning and activation pruning approaches are promising for the design of brainware processor with tiny YOLO3 algorithm. The issue of channel pruning is that an efficient optimization algorithm is necessary to predict the salient channels of tiny YOLO3. Moreover, it is challenging for the hardware design engineers to exploit the accurate pruned algorithm because the neural network's retraining is indispensable. Nonetheless, a compact processor of tiny YOLO3 with an approximated weight strategy will attract more attention.

b: *WEIGHT AND ACTIVATION QUANTIZATION*

Generally, the state-of-the-art data stored in the central processing unit (CPU) or graphics processing unit (GPU) are floating-point format, which not only occupies a large number of memories but also increases the complexity of circuit design. Using low bit-width or fixed-point number enables to reduce the memory usage and complexity of circuit design. The quantization method can shorten the bit-length of data (floating-point or fixed-point) from 32 bits to 16 bits, 8 bits, 4 bits, 2 bits, or even 1 bit [54]–[56]. Two main types of quantization methods have been studied in the literature: deterministic quantization and stochastic quantization. The stochastic quantization randomly designates weights to be quantized, which exhibits low-precision characteristic. The deterministic quantization intends to find the optimized fixed-point number near to the floating-point number while the quantized values are randomly sampled from the real value [57]. It takes the following three advantages for hardware acceleration of neural networks adopting weight quantization [58],

- The short bit-width of weight reduces the arithmetic precision, which needs less logic gate for MAC operation.
- The reduction of bit-width can significantly decrease data storage space, which improves the feasibility of on-chip memory design.
- Fewer accesses with external memory can reduce the power consumption of the processor.

The quantization can be applied not only to the weight quantization, but also to activation quantification. This paper will explain its principle using weight quantification as an example. Rounding the weight to "+1" and "−1" is a simple and effective way to quantize the weight, which is also called binary neural network [59]. The weight quantization is defined as follows,

$$W^b = sign(W) = \begin{cases} +1 & W \geq 0 \\ -1 & otherwise \end{cases} \quad (35)$$

where W^b is the binarized weight. $sign(\bullet)$ is the sign function. Another advantage of binary weight is that the convolution operation can be implemented without multiplication operation, which is defined by the following equation,

$$cov(M, W) = (M \oplus sign(W)) \times \zeta \quad (36)$$

where M is the input image array. Symbol $\oplus$ represents the convolution with additions and subtractions, and $cov(\bullet)$ is the convolution operation. ζ is the scale factors of weight, which is the mean of weight elements,

$$\zeta = \frac{1}{p}\sum_{i=1}^{p} |W_i| \quad (37)$$

where p is the number of weight elements, and $|\bullet|$ is the absolute value symbol. In addition, the input image array also can be quantized based on Eq. 35, and the convolution operation is approximated by the XNOR gate and bit count function, which is written as follows [60],

$$cov(M, W) = (sign(M) \circledast sign(W)) \odot K \times \zeta \quad (38)$$

The symbol $\circledast$ represents the convolution with XNOR and bit count function, and $\odot$ indicates the element-wise multiplication. K is the scale factors of stride widow on the input image array, which is the mean of stride window of the input image array,

$$K_j = \frac{1}{q}\sum_{j=1}^{q} |M_j| \quad (39)$$

where q is the number of stride window. The neural network using the convolution calculation method in Eq. 38 is also called XNOR neural network, which has $58\times$ faster convolution operation and $32\times$ memory savings. The BinaryConnect, proposed by Courbariaux *et al.*, removes around 2/3 multiplications and saves $3\times$ training time using the binary weights during the forward and backward propagation [61]. Although binary neural networks can compress network models substantially, binary approximations severely degrade the accuracy of some models. In order to address the shortcomings of the binary neural network, the ternary weight network was proposed in 2016, which constrain the weights to +1, 0, and −1. It is illustrated that the multiplications with the

ternary weight network are reduced to 32× while showing slightly worse performance than full precision operations [62].

Discussion: The quantization of weight and activation can substantially reduce the dimension of the neural network model, which obviously can improve the efficiency of computation and reduce memory utilization. However, with the increase of the network model, the accuracy of the quantized model is decreasing. It is essential to effectively trade off the relationship between the accuracy of the neural network and the number of bit quantization. The maximum number of activation and weight of tiny YOLO3 are only 10.56 Megabytes and 18 Megabytes (floating-point format) within convolutional layers. Hence, the tiny YOLO3 with activation and weight quantization will be effective solution for the design of the compact brainware processor.

c: LOOP PERFORATION

The loop perforation attempts to reduce the computational overhead by selectively skipping some loop iterations [63]. The loop perforation works a similar way as the pruning techniques, except that the perforation selectively detaches the outputs rather than the weights or activations. The perforation rate (ρ) determines the expected percentage of loop iteration to be discarded, and the following equation defines the relationship between perforation rate and the expected number of execution loops (N_{exp}),

$$\rho = 1 - \frac{1}{N_{exp}} \tag{40}$$

The perforation rate defined in Eq. 40 is modulo perforation that is a type of static perforation. Other perforation methods can also be deployed to reduce the number of execution loops such as dynamic perforation, truncation perforation, and randomized perforation [64]. No matter what perforation method is adopted, the objective of perforation is to execute parts of the iterations while omitting some of the loops. Algorithm 2 is the pseudo codes of GEMM for the convolution operation of tiny YOLO3, which manifests that three main loops with complex computation are incorporated in the algorithm. Taking the last loop of GEMM as an example, the pseudo code with loop perforation is rewritten as follows,

$$for\ (int\ k = 0;\ k < D_o;\ k+ = N_{exp})$$

If the perforation rate is 0.75, the last loop of GEMM executes every four iterations, reducing 75% computations. The execution numbers can be reduced in each loop or the combination of different loops, depending on the accuracy requirements of applications. Primarily, the accuracy distortion (ϱ) is determined by the following expression,

$$\varrho = \frac{1}{k}\sum_{i=1}^{k} \omega_i \times |[|\,]\frac{o_i - \hat{o}_i}{o_i} \tag{41}$$

where k is the number of outputs for metric evaluation. ω_i is weight to assess the importance of outputs. $\hat{o}_i$ and o_i are the outputs with and without loop perforation, respectively. If the accuracy distortion is adequate, loop perforation can mitigate the computational burden in neural networks. Simultaneously, the reduction of computation indirectly reduces the demand for the memory of weights and feature maps. Motivating by the loop perforation, Figurnov *et al.* proposed the perforatedCNNS to eliminate the redundant convolutions within a convolutional neural network, which attests that the AlexNet and VGG-16 are accelerated by a factor of 2× – 4× using loop perforation [65].

Algorithm 2 GEMM for the Convolution Operation of Tiny YOLO3

Inputs: Image array (*img*), columns of image array (*img_cols*), weights array (*weight*), columns of weights array (*weight_cols*), columns of output array (*output_cols*), number of filters (N_f), size of filters (S_f), and dimension of output array (D_o)

Outputs: Results of convolution (*output*)

```
Function gemm(img, img_cols, weight, weight_cols, output_cols, N_f, S_f, D_o):
    for (int i = 0; i < N_f; i++) do
        for (int j = 0; j < S_f; j++) do
            for (int k = 0; k < D_o; k++) do
                output[i × out_cols + k] += weight[i × weight_cols + j] × img[j × img_cols + k]
    return output
```

Discussion: The loop perforation is another productive approach to reduce the computation complexity of the neural network. The challenge using the loop perforation is the trade-off of the accuracy distortion and compression rate. Although the computation consumption and memory utilization can be condensed with loop perforation, a massive loss of precision will result in object detection failure with tiny YOLO3. Hence, an effective loop selection or skipping algorithm will make loop perforation to be a potential solution for accelerating the tiny YOLO3 algorithm.

2) KNOWLEDGE DISTILLATION

The depletion of computation costs and memory utilization for tiny YOLO3 not only can be achieved by directly decreasing the number or precision of weights or activations but also can be accomplished employing network approximation and optimization. The remainder of this section will elaborate on the details of network compression with knowledge distillation.

In analogy to the phenomenon that caterpillars become butterflies in biology, Hinton *et al.* illustrated that training and inference of neural networks have different requirements model [66]. The key point of knowledge distillation is that a compact and shallow model is refined from a large and cumbersome model that trained with complex models. In general, the compact and cumbersome models are called student and teacher models, respectively. The knowledge distillation is

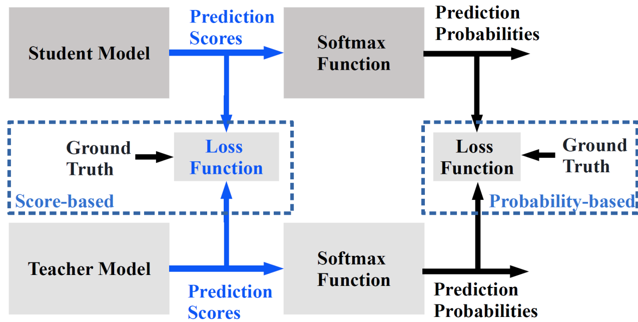


FIGURE 13. Knowledge distillation from teacher to student model.

the process that the student model generalizes the essentials from the teacher model's "soft-target" that refers to the intermediate feature maps after softmax function in different layers of the neural network. As shown in Fig. 13, the prediction probabilities or prediction scores can be utilized as knowledge for student to learn using probability-based [66] or score-based distillation [67]. Taking the probability-based distillation as an example, the following equations can define the predicted probabilities with the teacher model and student model (after softmax function).

$$p^T = \frac{e^{(z_i^T/T)}}{\sum_j e^{(z_j^T/T)}} \tag{42}$$

$$p^A = \frac{e^{(z_i^A/T)}}{\sum_j e^{(z_j^A/T)}} \tag{43}$$

where z_i^T and z_i^A are the un-normalized log probability values for teacher and student model, respectively. p^T and p^A are the corresponding probabilities. T is defined as the temperature of knowledge distillation. Therefore, the weights using knowledge distillation can be attained by training the student model with the following loss function [68],

$$\mathcal{L}(W^A) = (1-\lambda)\mathcal{H}(y_g, p^S) + \lambda\mathcal{H}(p^T, p^S) \tag{44}$$

where λ is the loss weight factor tuning the importance between soft-target and the ground truth (y_g). W^A is the weight of student model. $\mathcal{H}(\bullet)$ refers to the cross-entropy.

Lately, knowledge distillation has been widely studied for neural network compression. Motivating by the probability-based knowledge distillation, Romero *et al.* proposed an idea to train a thinner and deeper student model than the teacher model, which uses the prediction probabilities and uses the intermediate hints of the teacher model to train the student model [69]. Another activation-based attention was transferred from the teacher model to train the student model in [70], which achieves consequential performance improvement across a variety of datasets. Researches manifest that the student model's accuracy is close to the teacher model, even outperforms better than the teacher model [71], [72]. If the gap between teacher model and student model is large, the accuracy of the student model will be degraded. Mirzadeh *et al.* proposed an intermediate model, teacher assistant model, to bridge the gap between teacher model and student model [73].

Discussion: The knowledge distillation is an impressive technique to compress the neural network model while retaining good accuracy. Although the training transfer from teacher model to student model appears to be relatively complicated, the compressed tiny YOLO3 or YOLO3 model deploying knowledge distillation is a promising solution for the design of real-time brainware processor.

B. HARDWARE LEVEL

The circuit of a general processor contains the design of computation logic function, architecture optimization, and storage units. Therefore, the effective way to reduce the dimension and power consumption of brainware processor can be considered from different perspectives such as logic gate reduction and approximate memory techniques. The following section will introduce the compact brainware processor's open solutions in detail from the perspective of the hardware level.

Obviously, the computational complexity and memory utilization of tiny YOLO3 can be alleviated by quantization, pruning or sparsification at the algorithm level, and further reduce the usage of logic gates. Techniques for designing processors at the hardware level using quantization, pruning, and sparsification can be found in references [74]–[76]. Distinguish from the methods mentioned above, the compression of the logic circuit in the design of the processor can be realized by using approximate arithmetic circuits of adders. Owing to the majority of computation in tiny YOLO3 is the convolution computation that is mainly composed of adders, approximate adders will play a significant role in the design of compact brainware processors.

The basic building block for computation operations is an adder that implements the addition of two binary numbers. Among many adders, ripple-carry adder (RCA) and carry-lookahead adder (CLA) are the two most representative ones. An n-bit RCA is constructed by n cascaded full adders [77]. Since the circuit structure of RCA is composed of the full adder, the mathematical relationship between the circuit area and the bits number of the adder is linear, $\mathcal{O}(n)$. Because of the RCA circuit's cascade structure, the addition of each bit needs to wait for the carry from its previous bit addition before it starts, which makes the delay of RCA also proportional to the number of bits $\mathcal{O}(n)$. Unlike RCA, the CLA outputs each bit sum, propagate and generate in parallel, and the carry is processed in the carry-lookahead generator. The parallel processing architecture mitigates the computation delay of CLA, $\mathcal{O}(log(n))$, but the cost of the delay is the increase of circuit area, $\mathcal{O}(nlog(n))$ [78]. Since the circuit area is proportional to the power consumption of the circuit, CLA's power is greater than RCA for adders operating at the same bit. The purpose of the approximate adder is to reduce the complexity of the circuit architecture based on RCA and CLA architectures while sustaining

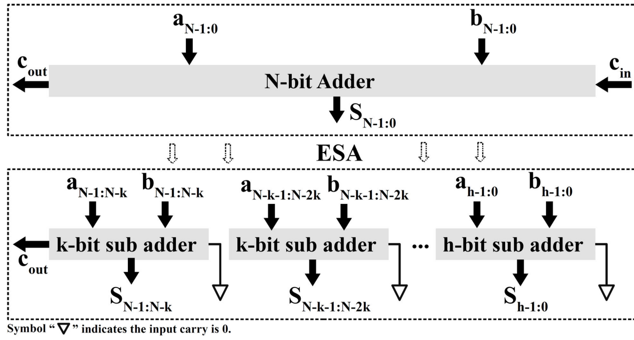


FIGURE 14. Adder approximation with ESA. c_{in} and c_{out} are the carry input and output, respectively. S is the sum output. Adapted from [81].

acceptable accuracy and further reducing the volume and power consumption of the circuit. There are mainly two directions to realize the adder approximation: truncating the carry propagation chain and reducing the number of transistors [79]. The physical implementations corresponding to these two methods are equal segmentation adder (ESA) and approximate full adder (AFA).

As shown in Fig. 14, the ESA segments an N-bit adder to several sub adders with fixed bit length (k) while the length of the least significant sub adder is h [80]. In ESA, all sub adders' input carry is set to zero, and all sub adders work in parallel. Since the bit length of sub adder determines the latency of ESA, $\mathcal{O}(log(k))$, the delay of adder reduces with a decrease of k. However, the accuracy of ESA grows with an increase of k, which makes it essential to trade-off the value of k in the light of the application's error-resilience when designing ESA. A straightforward way to improve ESA's accuracy is to increase the length of each sub adder and ensure that the number of sub adders remains the same through overlapping among the sub adders, which will not change the adder's delay [82], [83]. Another strategy to improve the adder's accuracy is to get more information for carry prediction by transferring the carry from adjacent sub adder to sum generator while retaining the length of sub adder unchanged. According to this technique, Zhu *et al.* proposed the error-tolerant adder type II (ETAII) that exhibits better precision than ESA [84]. However, the delay of ETAII increases from $\mathcal{O}(log(k))$ to $\mathcal{O}(log(2k))$ due to the carry transfer between adjacent sub adders. Meanwhile, a relatively complex circuit is desired to realize ETAII.

Another type of adder approximation is to divide the addition operation of an N-bit into the computation of the most significant bit (MSB) and the least significant bit (LSB). The MSB, including more valid information than the LSB, determines the accuracy of the entire arithmetic operation. In other words, if an error is introduced in the LSB, the result of the whole arithmetic operation may be slightly impacted. As shown in Fig. 15, the adder approximation can be accomplished using the inaccurate portion of computation (LSB), while the conventional adder is deployed in the accurate calculation portion (MSB). In view of this principle, Zhu *et al.*

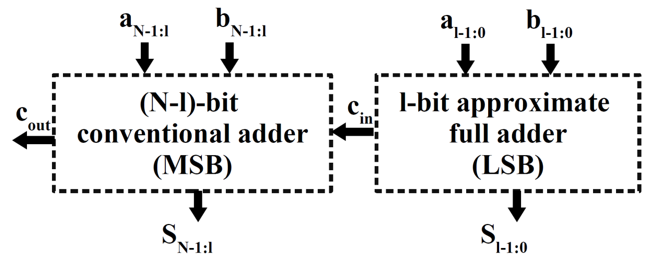


FIGURE 15. ESA for adder approximation.

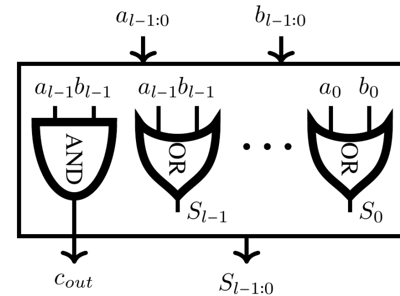


FIGURE 16. LOA for full adder approximation.

proposed another ESA by dividing the N-bit adder into two parts to approximate the adder's MSB which is approximated with XOR logics [85]. The accuracy and circuit complexity are dominated by the length of AFA (l). The adder's accuracy reduces with the increase of l, while the adder becomes complicated with the decrease of l. Similarly, AFA can be approximated by the OR gate, which is the so-called lower part OR adder (LOA) [86]. The core idea of LOA is that only the logic OR gate is utilized to approximate the MSB, and the carry is predicted by AND gate of the last bit. Fig. 16 shows the circuit implementation of LOA for l-bit LSB operation. The c_{out} of LOA will be served as the carry input of $(N - l)$-bit MSB. The circuit complexity of full adder within MSB can be significantly reduced with *AND* and *OR* gates-based approximation.

Discussion: Within the acceptable error range, the approximate adders can effectively mitigate the logic gates for the design of brainware processor. In the design of compact brainware processor, approximate multiplier and approximate memory are also valuable techniques. Although exploring the error-resilience of the approximated adder, multiplier or memory is challenging, the utilization of approximate circuits to the design of compact brainware processors will be a prospective solution shortly.

C. EMERGING SEMICONDUCTOR PROCESSING TECHNOLOGY

Deploying small transistor architecture with emerging semiconductor processing technology is a crucial way to reduce the circuit area of the compact processor. Many mature processors have been fabricated by foundry vendors using standard 65 *nm* processing technology such as Intel core,

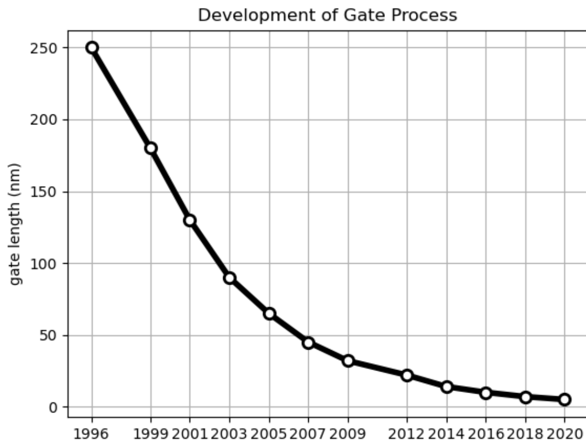


FIGURE 17. Development of gate process since 1996.

IBM cell, and NVIDIA GeForce GPU etc. Fig. 17 gives the development of the gate process from 1996 to 2020, which illustrates the gate length is decreased from 250 *nm* to 5 *nm* (50×). With the development of semiconductor processing technology, the 5 *nm* process node has been commercialized in Taiwan Semiconductor Manufacturing Company (TSMC) and Samsung Electronics based on multi-gate MOSFET (MuGFET) and fin field-effect transistors (FinFETs). Meanwhile, Samsung Electronics has announced the first 3 *nm* process based on nanosheet FET and will move into risk production in 2020 [87]. The chip leaves more space for memory storage or computing elements using new semiconductor processing technology such as 5 *nm* CMOS process, which offers larger logic density (256 Mb), lower power consumption [88]. The weight of standard YOLO3 with 416 × 416 inputs is 237 Mb, and tiny YOLO3 only has 33.8 Mb weight. Therefore, the full function implementation of YOLO3 or tiny YOLO3 algorithm embedded on the processor is achievable with novel semiconductor processing technology.

Discussion: In the near future, these emerging semiconductor devices will be widely utilized in the design of brainware processors. Moreover, the combination of algorithms, hardware and new semiconductor technologies will be the trend of designing compact brainware processors. Tab. 3 shows the expected evaluation of open solutions for the brainware processor design. The proposed open solution's characteristic is that the cost of accuracy may improve most of the performance. Since the human perception of error is limited, the other performance of processors such as throughput or latency can be enhanced by sacrificing accuracy if the errors of the brainware processor are within the range of human perception error or the error is acceptable for object detection.

TABLE 3. Expected evaluation of open solutions for the brainware processor design.

	THP	LT	Acc.	Power	Cost	LC	Dim.
Sparsification	✓	✓	×	✓	✓	–	✓
Quantization	✓	✓	×	✓	✓	–	✓
Perforation	✓	✓	×	✓	✓	–	✓
Knowledge Dis.	✓	✓	×	✓	✓	–	✓
Adder Appro.	✓	✓	×	✓	✓	–	✓
New Tech.	✓	✓	✓	✓	×	✓	✓

Note: THP–throughput, LT–latency, Acc.–accuracy, LC–lifecycle, Dim– dimension Dis.–distillation, Appro.–approximation, Tech.–technology.

VI. SUMMARY

In this paper, the systematic study of tiny YOLO3 inference has been presented. Meanwhile, the detailed analysis of the algorithm step by step is provided, and the complete definition of each parameter is illustrated. The paper gives a detailed explanation not only in theory but also in engineering implementation, which is a thorough review combining theory with practice. Moreover, the challenges and open solutions for the compact YOLO processor's design have proposed from algorithm, hardware, and emerging semiconductor processing technology level.

On account of the limited human perception towards errors, this paper introduces the technology of approximate computing and approximate circuit into the open solution of brainware processor design at the level of algorithm and hardware implementation. At the level of algorithm, weight or activation quantization and sparsification and loop perforation will be promising solutions to improve the throughput and reduce the delay of brainware processor within the acceptable error range. The approximate adders, multipliers, or memory will simplify the circuit complexity and reduce the utilization of logic gates and decrease the power consumption of the brainware processor. In brief, this paper's studies not only contribute to the algorithm's understanding of object detection but offer a valuable reference for researchers or engineers to develop compact brainware processor.

APPENDIX A
PARAMETER DEFINITION IN CONFIGURATION FILE

Tab. 4 provides details of parameters definition in the configuration file ("cfg" file) of DarkNet.

APPENDIX B
ZERO-PADDING AND IMAGE ARRAY RESHAPE

The feature matrix conversion is achieved channel by channel. Since image array laid out in memory is in BGR-BGR-BGR order (BGR represents to blue, green and red color), the image array should be converted to BBB-GGG-RRR format at first. Algorithm 3 (*zeroPadding*) shows the pseudo codes of zero-padding and image array reshape.

APPENDIX C
POST-PROCESSING OF TINY YOLO3

The post-processing of tiny YOLO3 includes the bounding box regression, post-processing of bounding box, and non-max suppression. The remainder of this section gives a brief introduction about post-processing of tiny YOLO3.

TABLE 4. Parameters definition in "cfg" file.

Parameters	Value	Definition
[•]	–	layer start indicator of neural network, [net] is the entire network configuration
batch	1	accumulated samples for back-propagation
subdivisions	1	division numbers of batch for forward propagation
width	416	width of input images
height	416	height of input images
channels	3	channels of input images
momentum	0.9	momentum of network
decay	0.0005	decay of network
angle	0	rotation angle of training images
saturation	1.5	saturation of training images
exposure	1.5	exposure of training images
hue	0.1	hue of training images
learning rate	0.001	learning rate of neural network
burn_in	1000	number of samples training for learning rate monitor
max_batches	1000	maximum training batches
policy	steps	tune policy of learning rate
steps	400000, 450000	steps of tuning learning rate after 400000 and 450000 steps
scales	0.1, 0.1	dedcay coefficient of learning rate
batch_normalize	1	flag of batch normalization processing
filters	32,64,128 ···	number of filters/kernels in each layer
size	1, 2 or 3	size of filters/kernels
stride	1 or 2	stride of filters/kernels
pad	1	zero-padding of filters/kernels
activation	leaky or linear	types of activation function
mask	0, 1, 2, 3, 4, 5,	index of anchor box
anchors	[10, 14], [23,27], [37, 58], [81, 82], [135,169], [344, 319]	size of prediction box
classes	80	number of object categories
num	6	number of predicted box in each grid cell
jitter	0.3	jitter noise to avoid over-fitting
ignore_threshold	0.7	threshold to ignore the bounding box
truth_threshold	1	threshold to determine whether computing IOU
random	1	flag of using multi-scale training
layers	[-1, 8], [-4]	layer index for concatenation

Note: IOU – intersection of union and symbol "–" denotes the item does not exist.

A. BOUNDING BOX REGRESSION

During the period of training, the data was normalized by the neural network. Specifically, the center coordinates and the dimension of the bounding box are confined from 0 to 1 by dividing the dimension of feature maps and the input image. The offsets of center coordinates (t_x, t_y) and the scale of bounding box (t_w, t_h) are defined by following expressions,

$$t_x = b_x \times l_w - c_x \tag{45}$$

$$t_y = b_y \times l_h - c_y \tag{46}$$

$$t_w = log\left(\frac{b_w \times w}{p_w}\right) \tag{47}$$

$$t_h = log\left(\frac{b_h \times h}{p_h}\right) \tag{48}$$

Algorithm 3 Zero-Padding and Image Array Reshape

Inputs: Image array (*img*), rows, columns and channels of image array (*img_rows*, *img_cols*, *img_channels*), *paddings*.

Outputs: Results of zero-padding (*img_pad_out*)

Function `zeroPadding`(*img, img_rows, img_cols, img_channels, paddings*) **:**

- $img_rows = img_rows + 2 * paddings$; $img_cols = img_cols + 2 * paddings$;
- **for** $k \in channels$ **do**
  - $ptr_index = 0$; **for** $i \in rows$ **do**
    - **if** $(i == 0) || (i == (img_rows - 1))$ **then**
      - **for** $j \in img_cols$ **do**
        - $img_index = (k * img_rows + i) * img_cols + j$;
        - $img_pad_out[img_index] = 0$;
    - **else**
      - $col_index = k$; $uchar * rowAddress = img.ptr < uchar > (ptr_index)$; **for** $j \in img_cols$ **do**
        - **if** $(j == 0) || (j == (img_cols - 1))$ **then**
          - $img_index = (k * img_rows + i) * img_cols + j$;
          - $img_pad_out[img_index] = 0$;
        - **else**
          - $img_index = (k * img_rows + i) * img_cols + j$;
          - $img_pad_out[img_index] = rowAddress[col_index]$;
          - $col_index = col_index + img_channels$;
      - $ptr_index = ptr_index + 1$;
- **return** *img_pad_out*

where l_w and l_h are the width and height of feature maps, that is, $l_w = l_h = 13$. The scale of bounding box is computed using logarithmic function $log(\bullet)$. Rearranging the above equation, the normalized center coordinates of the object and the dimension of the bounding box can be obtained according to the following equations,

$$b_x = \frac{\sigma(t_x) + c_x}{l_w} \tag{49}$$

$$b_y = \frac{\sigma(t_y) + c_y}{l_h} \tag{50}$$

$$b_w = \frac{p_w \times e^{t_w}}{w} \tag{51}$$

$$b_h = \frac{p_h \times e^{t_h}}{h} \tag{52}$$

Since the dimension of bounding boxes is defined in terms of the dimension of input images, the normalization of b_w and b_h are calculated by dividing the width and height of the input image. Therefore, the center coordinate

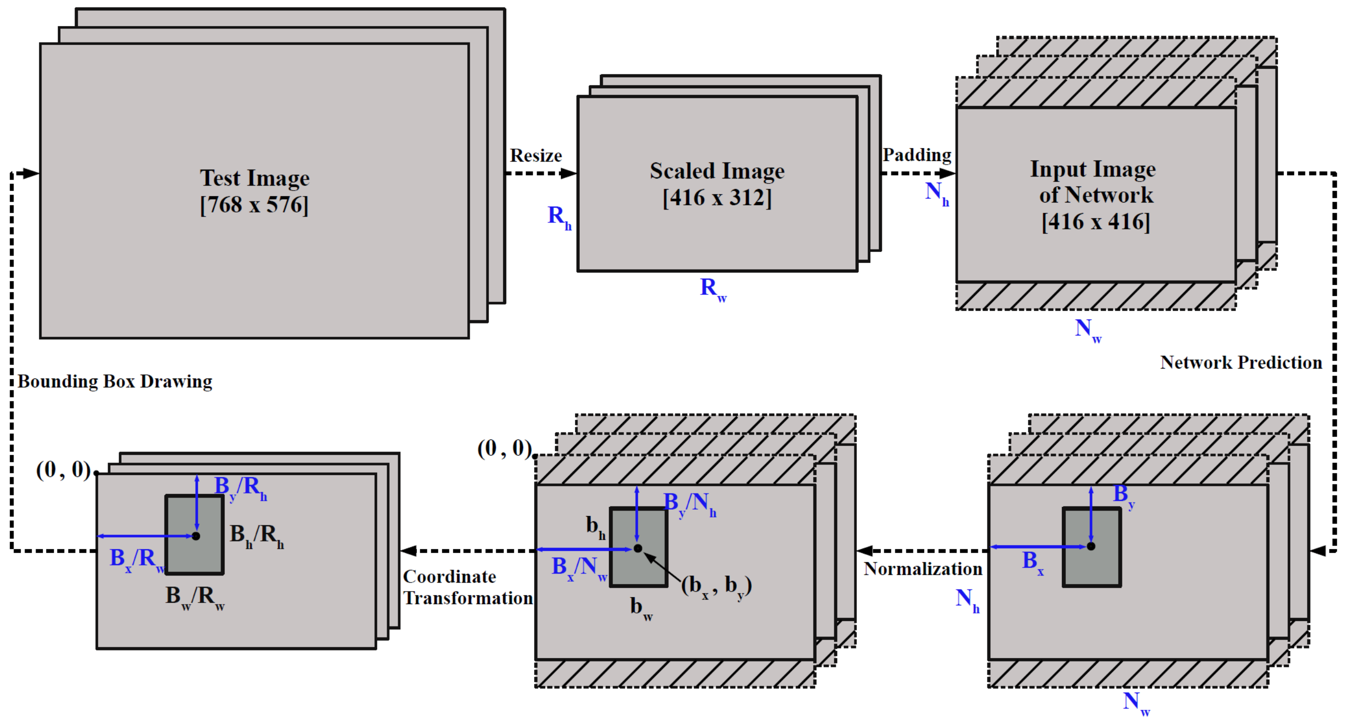


FIGURE 18. Post processing of bounding box.

and dimension of bounding boxes fall in the range of [0, 1]. In addition, two types of anchor boxes are utilized in different "yolo" layer. Low-resolution feature maps (13 × 13) are effective in predicting large objects, so a large pre-defined anchor box is adopted for the bounding box prediction under this condition, and vice versa. The parameters of anchor boxes are arranged sequentially in memory, and every two elements represent the width and height of each anchor box. The first "yolo" layer uses the last three pairs of anchor boxes ([(81, 82)(135, 169), (344, 319)]) while the second "yolo" layer utilizes the other three anchor boxes [(10, 14), (23, 27), (37, 58)] for bounding boxes prediction.

B. POST-PROCESSING OF BOUNDING BOX

The convolution operation down-samples the input image arrays and outputs feature maps for object prediction. In tiny YOLO3, the prediction is implemented on two scales using two distinct "yolo" layers. At the input of neural network, the dimension of image is resized to 416 × 416, which facilitates the neural network to adapt different dimensions of input images. As shown in Fig. 18, the test image (768 × 576) is resized to a scaled image with dimension of 416 × 312. Taking the priority over larger value in height and height, the interpolation approach is accommodated to resize the image. The image resizing is achieved along width and height scale. Specifically, the 768 × 576 image is resized to 416 × 576 according to width scale, and then the dimension is converted to 416 × 312 based on height scale, which is described as follows,

$$768 \times 576 \xrightarrow[\text{interpolation}]{\text{width scale}} 416 \times 576 \xrightarrow[\text{interpolation}]{\text{height scale}} 416 \times 312$$

The width scale and height scale are 1.84819 [(768-1)/(416-1)] and 1.84887 [(576-1)/(312-1)], respectively. In this illustration, since the width has a larger value, the scale factor (η) of resize is calculated based on width. The width of the scaled image (R_w) equals to the input width of the neural network (N_w) while the height is evaluated proportionally according to the scale factor,

$$\eta = \frac{N_w}{I_w} \tag{53}$$

$$R_h = \eta \times I_h \tag{54}$$

where I_w and I_h are the width and height of test image, respectively. The height of scaled image can be calculated according Eq. 54, that is, $R_h = 312$. To ensure that the input image has a fixed dimension (416 × 416), the height of the scaled image is extended to 416 by padding a constant pixel value (128) to the image. As described in Eq. 49 – Eq. 52, the center coordinate, width and height of predicted bounding box are relative values rather than absolute values. Moreover, the predicted parameters of bounding boxes are in the coordinate of the input image (416 × 416) of the neural network, which is extended by the padding method. To ensure that the bounding box can adapt to the size of test image, the predicted parameters of the bounding box should be transformed to the coordinate of the scaled image (416 × 312). Therefore, the center coordinate $(\tilde{b}_x, \tilde{b}_y)$, width

($\tilde{b}_w$) and height ($\tilde{b}_h$) of bounding box in the coordinate of scaled image can be evaluated according to b_x, b_y, b_w, and b_h based on the following expressions,

$$\begin{aligned}\tilde{b}_y &= \frac{B_y}{R_h} \\ &= \frac{b_y \times N_h - (N_h - R_h)/2}{R_h} \\ &= b_y \times \frac{N_h}{R_h} - \frac{N_h - R_h}{2} \times \frac{1}{R_h} \\ &= b_y \times \frac{N_h}{R_h} - \frac{N_h - R_h}{2 \times N_h} \times \frac{N_h}{R_h} \\ &= \left(b_y - \frac{N_h - R_h}{2 \times N_h}\right) \times \frac{N_h}{R_h} \end{aligned} \tag{55}$$

$$\tilde{b}_x = \left(b_x - \frac{N_w - R_w}{2 \times N_w}\right) \times \frac{N_w}{R_w} \tag{56}$$

$$\begin{aligned}\tilde{b}_h &= \frac{B_h}{R_h} \\ &= b_h \times \frac{N_h}{R_h}\end{aligned} \tag{57}$$

$$\tilde{b}_w = b_w \times \frac{N_w}{R_w} \tag{58}$$

where B_x, B_y represent the absolute height and absolute width of center points of the bounding box, and B_w, B_h are the absolute height and absolute width of the bounding box.

C. NON-MAX SUPPRESSION

The bounding boxes are filtered out if the corresponding objectiveness scores are less than the threshold (0.5) during the post-processing period. The remaining bounding boxes with the probability over than threshold are selected for category determination. In addition, the class probability is further evaluated using the multiplication of the objectiveness score and the predicted probability of each category, which is written as follows,

$$Pr_i = \begin{cases} S_{obj} \times Pr_i & if \ \left(S_{obj} \times Pr_i\right) > 0.5 \\ 0 & otherwise \end{cases} \tag{59}$$

With the class probabilities and the corresponding parameters of bounding boxes, the non-max suppression (NMS) approach is used to match the bounding boxes to the object categories. The essential idea of NMS is to discard the bounding boxes with low probabilities in the categories. As shown in Fig. 19, the feature maps with large objectiveness scores (greater than 0.5) are saved for affirmation of bounding boxes.

Assuming that the probabilities of the remaining five bounding boxes (B_1, B_2, B_3, B_4, B_m) are greater than 0.5, the first step of NMS is to sort the corresponding probabilities in descending order and search for the bounding box with maximum probability (B_m) in that correspondent category. If all probabilities in one of the categories are zero, it indicates that the corresponding category is not the detected object. The next step is to filter out the bounding boxes that has a large overlap with the bounding box having maximum probability.

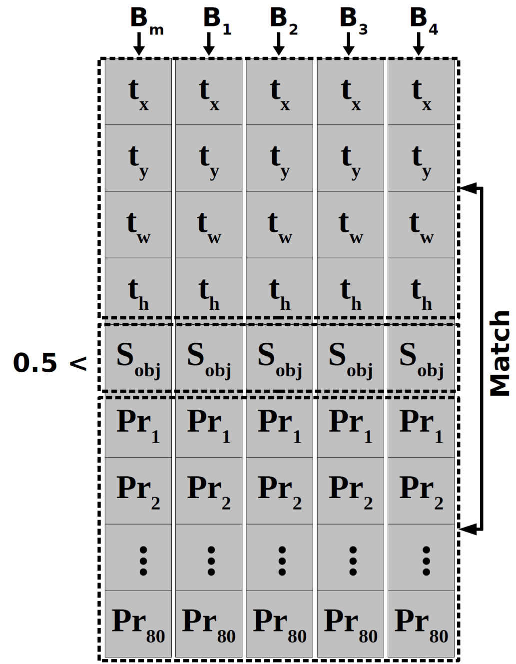


FIGURE 19. Feature maps for NMS.

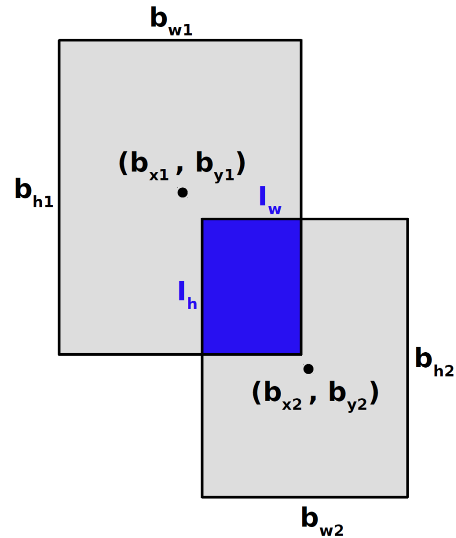


FIGURE 20. Definition of IOU between bounding boxes.

Fig. 20 shows the overlap definition between bounding boxes. The overlap between two bounding boxes is evaluated by the concept of the intersection of union (IOU), which is defined the ratio between intersection and union,

$$IOU = \frac{Intersection}{Union} \tag{60}$$

$$= \frac{B_m \cap B_j}{B_m \cup B_j} \tag{61}$$

where B_j is the other bounding box except the bounding box with maximum probability. $B_m \cap B_j$ is evaluated by the

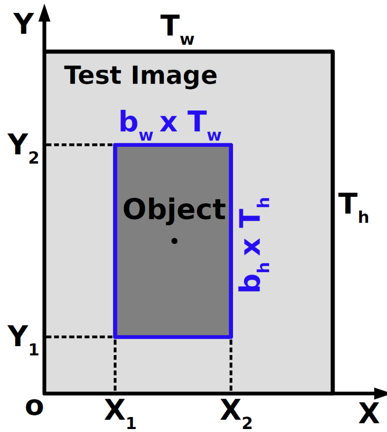


FIGURE 21. Coordinates definition of bounding box.

intersection area, which is defined by follows,

$$B_m \cap B_j = I_w \times I_h \tag{62}$$

where I_w and I_h are the width and height of the intersection area. The I_w is calculated by following expressions,

$$I_w = R_w - L_w \tag{63}$$
$$Lw = \begin{cases} L_1 & if\ L_1 > L2 \\ L_2 & otherwise \end{cases} \tag{64}$$
$$L_1 = b_{x1} - b_{w1}/2 \tag{65}$$
$$L_2 = b_{x2} - b_{w2}/2 \tag{66}$$
$$R_w = \begin{cases} R_1 & if\ R_1 < R_2 \\ R_2 & otherwise \end{cases} \tag{67}$$
$$R_1 = b_{x1} + b_{w1}/2 \tag{68}$$
$$R_2 = b_{x2} + b_{w2}/2 \tag{69}$$

where R_w and L_w are the rightmost and leftmost points of intersection area. $L1$, $L2$, R_1, and R_2 are the intermediate values. Similar to I_w, I_h can computed by following equation,

$$I_h = D_w - U_w \tag{70}$$
$$D_w = \begin{cases} U_1 & if\ U_1 > U2 \\ U_2 & otherwise \end{cases} \tag{71}$$
$$U_1 = b_{y1} - b_{h1}/2 \tag{72}$$
$$U_2 = b_{y2} - b_{h2}/2 \tag{73}$$
$$D_w = \begin{cases} D_1 & if\ D_1 < D_2 \\ D_2 & otherwise \end{cases} \tag{74}$$
$$D_1 = b_{y1} + b_{h1}/2 \tag{75}$$
$$D_2 = b_{y2} + b_{h2}/2 \tag{76}$$

where D_w and U_w are the uppermost and lowermost points of the intersection area. $U1$, $U2$, D_1, and D_2 are the intermediate values. Once the intersection is obtained, the union can be calculated by the following expression,

$$B_m \cup B_j = b_{w1} \times b_{h1} + b_{w2} \times b_{h2} - B_m \cap B_j \tag{77}$$

Substituting Eq. 62 and Eq. 77 into Eq. 61, the IOU defined in Fig. 20 can be achieved from the following equations,

$$IOU = \frac{I_w \times I_h}{b_{w1} \times b_{h1} + b_{w2} \times b_{h2} - I_w \times I_h} \tag{78}$$
$$I_w = (b_{x1} - b_{x2}) + (b_{w1} + b_{w2})/2 \tag{79}$$
$$I_h = (b_{y1} - b_{y2}) + (b_{h1} + b_{h2})/2 \tag{80}$$

If the IOU between B_m and B_j is greater than a pre-defined threshold such as 0.5, the bounding box B_j is filtered out. Fig. 21 illustrates the definition of actual coordinates of the detected object. The predicted coordinates of bounding boxes after IOU filtering is used to evaluate the coordinates of detected object in test image, which is written as follows,

$$X_1 = \left(b_x - \frac{b_w}{2}\right) \times T_w \tag{81}$$
$$X_2 = \left(b_x + \frac{b_w}{2}\right) \times T_w \tag{82}$$
$$Y_1 = \left(b_y - \frac{b_h}{2}\right) \times T_h \tag{83}$$
$$Y_2 = \left(b_y + \frac{b_h}{2}\right) \times T_w \tag{84}$$

where T_w and T_h are the width and height of test image.

REFERENCES

[1] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani, "Deep CNN-based real-time traffic light detector for self-driving vehicles," *IEEE Trans. Mobile Comput.*, vol. 19, no. 2, pp. 300–313, Feb. 2020.

[2] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, and D. Li, "Object classification using CNN-based fusion of vision and LIDAR in autonomous vehicle environment," *IEEE Trans. Ind. Informat.*, vol. 14, no. 9, pp. 4224–4231, Sep. 2018.

[3] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of CNN and RNN for natural language processing," 2017, *arXiv:1702.01923*. [Online]. Available: http://arxiv.org/abs/1702.01923

[4] K. Sekaran, P. Chandana, N. M. Krishna, and S. Kadry, "Deep learning convolutional neural network (CNN) with Gaussian mixture model for predicting pancreatic cancer," *Multimedia Tools. Appl.*, vol. 79, pp. 10223–10247, Mar. 2019.

[5] H. Lee, S. Eum, and H. Kwon, "ME R-CNN: Multi-expert R-CNN for object detection," *IEEE Trans. Image Process.*, vol. 29, pp. 1030–1044, 2020.

[6] H. Mao, S. Yao, T. Tang, B. Li, J. Yao, and Y. Wang, "Towards real-time object detection on embedded systems," *IEEE Trans. Emerg. Topics Comput.*, vol. 6, no. 3, pp. 417–431, Sep. 2018.

[7] W. Gross, N. Onizawa, K. Matsumiya, and T. Hanyu, "Application of stochastic computing in brainware," *Nonlinear Theory Appl., IEICE*, vol. 9, no. 4, pp. 406–422, 2018.

[8] M. Natsui, T. Chiba, and T. Hanyu, "Design of MTJ-based nonvolatile logic gates for quantized neural networks," *Microelectron. J.*, vol. 82, pp. 13–21, Dec. 2018.

[9] X. Wu, V. Saxena, K. Zhu, and S. Balagopal, "A CMOS spiking neuron for brain-inspired neural networks with resistive synapses and *in situ* learning," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 11, pp. 1088–1092, Nov. 2015.

[10] S. B. Furber, "Brain-inspired computing," *IET Comput. Digit. Techn.*, vol. 10, no. 6, pp. 299–305, Nov. 2016.

[11] N. Onizawa, S. C. Smithson, B. H. Meyer, W. J. Gross, and T. Hanyu, "In-hardware training chip based on CMOS invertible logic for machine learning," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 5, pp. 1541–1550, May 2020, doi: 10.1109/TCSI.2019.2960383.

[12] J. Pei *et al.*, "Towards artificial general intelligence with hybrid Tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, 2019.

[13] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep learning for generic object detection: A survey," *Int. J. Comput. Vis.*, vol. 128, no. 2, pp. 261–318, Feb. 2020.
[14] T. Li, Y. Ma, H. Shen, and T. Endoh, "FPGA implementation of real-time pedestrian detection using normalization-based validation of adaptive features clustering," *IEEE Trans. Veh. Technol.*, early access, Feb. 28, 2020, doi: 10.1109/TVT.2020.2976958.
[15] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, Nov. 2004.
[16] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2005, pp. 886–893.
[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.
[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.
[19] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.
[20] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[21] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. Comput. Vis. (ECCV)*, Amsterdam, The Netherlands, 2016, pp. 21–37.
[22] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.
[23] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7263–7271.
[24] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767
[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[26] A. Bochkovskiy, C.-Y. Wang, and H.-Y. Mark Liao, "YOLOv4: Optimal speed and accuracy of object detection," 2020, *arXiv:2004.10934*. [Online]. Available: http://arxiv.org/abs/2004.10934
[27] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 31–40.
[28] Y. J. Wai, Z. Bin, S. Irwan, and L. Kim, "Fixed point implementation of Tiny-Yolo-v2 using OpenCL on FPGA," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 10, pp. 506–512, 2018.
[29] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.
[30] Z. Yu and C.-S. Bouganis, "A parameterisable FPGA-tailored architecture for YOLOv3-tiny," in *Proc. Int. Symp. Appl. Reconf. Comput*, Toledo, Spain, 2020, pp. 330–344.
[31] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An architecture for ultralow power binary-weight CNN acceleration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 48–60, Jan. 2018.
[32] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W convolutional network accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 11, pp. 2461–2475, Nov. 2017.
[33] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[34] C. Chen, X. Liu, H. Peng, H. Ding, and C.-J. Richard Shi, "IFPNA: A flexible and efficient deep learning processor in 28-nm CMOS using a domain-specific instruction set and reconfigurable fabric," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 346–357, Jun. 2019.
[35] T. Hanyu, T. Endoh, D. Suzuki, H. Koike, Y. Ma, N. Onizawa, M. Natsui, S. Ikeda, and H. Ohno, "Standby-power-free integrated circuits using MTJ-based VLSI computing," *Proc. IEEE*, vol. 104, no. 10, pp. 1844–1863, Oct. 2016.
[36] M. Natsui, Y. Noguchi, M. Yasuhira, H. Sato, S. Ikeda, H. Ohno, T. Endoh, T. Hanyu, D. Suzuki, A. Tamakoshi, T. Watanabe, H. Honjo, H. Koike, T. Nasuno, Y. Ma, and T. Tanigawa, "A 47.14-μW 200-MHz MOS/MTJ-hybrid nonvolatile microcontroller unit embedding STT-MRAM and FPGA for IoT applications," *IEEE J. Solid-State Circuits*, vol. 54, no. 11, pp. 2991–3004, Nov. 2019.
[37] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.
[38] Sparsh Mittal. 2016, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, p. 62, May 2016, doi: 10.1145/2893356.
[39] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *Proc. Int. Symp. Low Power Electron. Design ISLPED*, 2014, pp. 27–32.
[40] P. Panda, A. Sengupta, S. S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, and K. Roy, "Invited–cross-layer approximations for neuromorphic computing: From devices to circuits and systems," in *Proc. 53rd Annu. Design Autom. Conf. DAC*, 2016, pp. 1–6.
[41] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proc. 52nd Annu. Design Autom. Conf. DAC*, 2015, p. 120.
[42] S. Lin, R. Ji, Y. Li, C. Deng, and X. Li, "Toward compact ConvNets via structure-sparsity regularized filter pruning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 2, pp. 574–588, Feb. 2020.
[43] Y. Le Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 2. 1990, pp. 598–605.
[44] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149
[45] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Venice, Italy, 2017, pp. 5058–5066.
[46] D. Blalock, J. Javier Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" 2020, *arXiv:2003.03033*. [Online]. Available: http://arxiv.org/abs/2003.03033
[47] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," in *Proc. Brit. Mach. Vis. Conf.*, 2015, p. 31.
[48] C. Liu and H. Wu, "Channel pruning based on mean gradient for accelerating convolutional neural networks," *Signal Process.*, vol. 156, pp. 84–91, Mar. 2019.
[49] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, New Orleans, LA, USA, 2019, pp. 1–14.
[50] Z. Chen, T.-B. Xu, C. Du, C.-L. Liu, and H. He, "Dynamical channel pruning by conditional accuracy change for deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Apr. 2, 2020, doi: 10.1109/TNNLS.2020.2979517.
[51] Z. Yao, K. Huang, H. Shen, and Z. Ming, "Deep neural network acceleration with sparse prediction layers," *IEEE Access*, vol. 8, pp. 6839–6848, 2020, doi: 10.1109/ACCESS.2020.2963941.
[52] Z. Wang, S. Lin, J. Xie, and Y. Lin, "Pruning blocks for CNN compression and acceleration via online ensemble distillation," *IEEE Access*, vol. 7, pp. 175703–175716, 2019, doi: 10.1109/ACCESS.2019.2957203.
[53] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays FPGA*, 2016, pp. 26–35.
[54] S. Yin and J.-S. Seo, "A 2.6 TOPS/W 16-bit fixed-point convolutional neural network learning processor in 65-nm CMOS," *IEEE Solid-State Circuits Lett.*, vol. 3, pp. 13–16, Jan. 2020.
[55] N. Mitschke, M. Heizmann, K.-H. Noffz, and R. Wittmann, "A fixed-point quantization technique for convolutional neural networks based on weight scaling," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2019, pp. 3836–3840.
[56] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014.
[57] Y. Guo, "A survey on methods and theories of quantized neural networks," 2018, *arXiv:1808.04752*. [Online]. Available: http://arxiv.org/abs/1808.04752
[58] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," 2015, *arXiv:1511.06488*. [Online]. Available: http://arxiv.org/abs/1511.06488

[59] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Barcelona, Spain, 2016, pp. 4107–4115.

[60] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Computer Vision—ECCV*. Springer, 2016, pp. 525–542.

[61] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, Montreal, QC, Canada, 2015, pp. 3105–3113.

[62] F. Li and B. Liu, "Ternary weight networks," in *Proc. NIPS Workshop Efficient Methods Deep Neural Netw.*, Barcelona, Spain, 2016, pp. 1–5.

[63] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62-1–62-33, Mar. 2016.

[64] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, respond to failures," Massachusetts Inst. Technol. (MIT), Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[65] M. Figurnov, D. Vetrov, and P. Kohli, "PerforatedCNNs: Acceleration through elimination of redundant convolutions," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, vol. 2016, pp. 947–955.

[66] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*. [Online]. Available: http://arxiv.org/abs/1503.02531

[67] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proc. SIGKDD*, 2006, pp. 535–541.

[68] S. Shin, Y. Boo, and W. Sung, "Knowledge distillation for optimization of quantized deep neural networks," 2019, *arXiv:1909.01688*. [Online]. Available: http://arxiv.org/abs/1909.01688

[69] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for thin deep nets," in *Proc. ICLR*, 2015, pp. 1–13.

[70] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," 2016, *arXiv:1612.03928*. [Online]. Available: http://arxiv.org/abs/1612.03928

[71] T. Furlanello, Z. C. Lipton, M. Tschannen, L. Itti, and A. Anandkumar, "Born-again neural networks," in *Proc. 35th Int. Conf. Mach. Learn., ICML*, Stockholm, Sweden, 2018, pp. 1602–1611.

[72] J. Yim, D. Joo, J. Bae, and J. Kim, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4133–4141.

[73] S.-I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, "Improved knowledge distillation via teacher assistant," 2019, *arXiv:1902.03393*. [Online]. Available: http://arxiv.org/abs/1902.03393

[74] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[75] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[76] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 1–39, 2019.

[77] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed, Reading, MA, USA: Addison-Wesley, Mar. 2010.

[78] H. Jiang, J. Han, and F. Lombardi, "A comparative review and evaluation of approximate adders," in *Proc. 25th Ed. Great Lakes Symp. VLSI GLSVLSI*, 2015, pp. 343–348, doi: 10.1145/2742060.2743760.

[79] C. Liu, J. Han, and F. Lombardi, "An analytical framework for evaluating the error characteristics of approximate adders," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1268–1281, May 2015.

[80] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han, "A review, classification, and comparative evaluation of approximate arithmetic circuits," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 4, pp. 60:1–60:34, Aug. 2017.

[81] S. Dutt, S. Nandi, and G. Trivedi, "A comparative survey of approximate adders," in *Proc. 26th Int. Conf. Radioelektronika (RADIOELEKTRONIKA)*, Apr. 2016, pp. 61–65.

[82] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, Mar. 2004.

[83] S. Dutt, S. Dash, S. Nandi, and G. Trivedi, "Analysis, modeling and optimization of equal segment based approximate adders," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 314–330, Mar. 2019.

[84] N. Zhu, W.-L. Goh, and K.-S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *Proc. 12th IEEE Int. Symp. Integr. Circuits (ISIC)*, Dec. 2009, pp. 69–72.

[85] N. Zhu, W. Ling Goh, W. Zhang, K. Seng Yeo, and Z. Hui Kong, "Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 8, pp. 1225–1229, Aug. 2010.

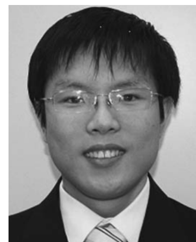
[86] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 4, pp. 850–862, Apr. 2010.

[87] M. LaPedus. (2019). *5nm Vs. 3Nm-Half Nodes, Different Transistor Types, and Numerous Other Options are Adding Uncertainty Everywhere*. [Online]. Available: https://semiengineering.com/5nm-vs-3nm/

[88] G. Yeap *et al.*, "5nm CMOS production technology platform featuring full-fledged EUV, and high mobility channel FinFETs with densest 0.021 μm2 SRAM cells for mobile SoC and high performance computing applications," in *IEDM Tech. Dig.*, San Francisco, CA, USA, 2019, pp. 36.7.1–36.7.4, doi: 10.1109/IEDM19573.2019.8993577.

TAO LI received the Ph.D. degree in navigation, guidance and control from Harbin Engineering University, Harbin, China, in 2016.

From 2013 to 2015, he was a Visiting Ph.D. student with the University of Calgary, Canada. From 2016 to 2018, he was an Academic Researcher with the Institute of Innovative Science and Technology, Tokai University, Hiratsuka, Japan. From 2018 to 2020, he was an Academic Researcher, served as a member of the Japan Science and Technology Agency-Open Innovation Platform with Enterprises, Research Institute and Academia (JST-OPERA) project, with the Graduate School of Engineering, Tohoku University, Sendai, Japan. He is currently an Assistant Professor with the Center for Innovative Integrated Electronic Systems (CIES), Tohoku University, Sendai, serving as a member of the Cross-ministerial Strategic Innovation Promotion Program (SIP) project. His main research interests include development and verification of nonvolatile brainware processor, digital signal processing, and inertial navigation.



YITAO MA received the B.S. degree from Osaka University, Osaka, Japan, in 2006, and the Ph.D. degree in electronic engineering from The University of Tokyo, Tokyo, Japan, in 2011.

From 2008 to 2011, he was a Research Assistant in the Global COE programs with The University of Tokyo, where he was engaged in research on the development of machine learning algorithms and VLSIs. From 2011 to 2013, he was a Research Associate with the Frontier Research Institute for Interdisciplinary Sciences, Tohoku University. He served as a member of the CSTI-FISRT program and the JST-CREST program, where he was engaged in research on high-performance processors based on advanced MOS devices and spintronic devices. Since 2014, he has been an Industry-Government-Academia Researcher with the Center for Spintronics Integrated Systems, Tohoku University. His research interest includes high-speed ultra-low-power nonvolatile brain-inspired VLSIs combined with emerging devices. He is currently serving as a member of the JST-ACCEL program and the CSTI-ImPACT program.

TETSUO ENDOH (Senior Member, IEEE) received the B.S. degree in physics from The University of Tokyo, Tokyo, Japan, in 1987, and the Ph.D. degree in electronic engineering from Tohoku University, Sendai, Japan, in 1995.

He joined Toshiba Corporation, in 1987, where he was engaged in the research on NAND-type flash memory and advanced CMOS device design at VLSI Research Laboratories, the Research and Development Center, Toshiba Corporation, Kawasaki, Japan. He was a Lecturer with the Research Institute of Electrical Communication, Tohoku University, in 1995, an Associate Professor, in 1997, and a Professor, in April 2008. He was a Professor with the Center for Interdisciplinary Research, Tohoku University, in May 2008. Since May 2012, he has been a Professor with the Graduate School of Engineering, Tohoku University. He has also been the Deputy Director of the Center for Spintronics Integrated Systems, Tohoku University, since March 2010, and the Director of the Center for Innovative Integrated Electronic Systems, Tohoku University, since October 2012. He has been engaged in the research on advanced CMOS device design (e.g., 3DMOS devices, vertical MOS devices), cleanroom technology, and low-power and high-speed circuit technology and advanced memory (e.g., 3D memory and memory based on novel principles). He was a member of the High-performance Low-power Consumption Spin Devices and Storage Systems program as part of the Research and Development for Next-Generation Information Technology program of the Ministry of Education, Culture, Sports, Science and Technology. He is currently engaged in the development of novel nano-LSI with hybrid technology between spin memory devices and extended CMOS technology. He has served as the Research Leader of the Research and Development of Vertical Body Channel MOSFET and its Integration Process project as part of the Research of Innovative Material, and Process for Creation of Next-generation Electronics Devices program of Japan Science and Technology-CREST. He has also served as a Research Sub-leader of the Research and Development of Ultra-low Power Spintronics-based Logic VLSIs program as part of the Funding Program for World-Leading Innovative Research and Development on Science and Technology (FIRST Program).

Dr. Endoh is a Fellow of the Japan Society of Applied Physics (JSAP) and a member of the Institute of Electronics, Information, and Communication Engineers (IEICE). He received the LSI IP Design Award, in 2001, the Japanese Journal of Applied Physics (JJAP) Paper Award, in 2009, the 6th Fellow Award of the Japan Society of Applied Physics, in 2012, and the 2012 International Conference on Solid State Devices and Materials (SSDM) Paper Award, in 2012. He assumed an International Collaboration Fellow of Sendai City, in 2012.