ELEKTROTEHNIŠKI VESTNIK 91(3): 95–107, 2024
ORIGINAL SCIENTIFIC PAPER

A Comprehensive analysis of Deployment Optimization Methods for CNN-Based Applications on Edge Devices

Qi Li[1,†], **Zhenling Su**[1,†], **Lin Meng**[2,‡]

[1]*Graduate School of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga, Japan 525-8577*
[2]*College of Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga, Japan 525-8577*

[†] *These authors contributed equally to this work.*
[‡] *menglin@fc.ritsumei.ac.jp*

Abstract. The development of the promising Artificial Intelligence of The things (AIoT) technology increases the demand for implementing Convolutional Neural Networks (CNN) algorithms on the edge devices. However, implementing huge CNN-based applications on the resource-constrained edge devices is considered challenging. Therefore, several CNN optimization methods are integrated into the deployment tools of the edge devices. Since this field evolves rapidly, relevant tools adopt non-uniform deployment optimization flows, and the optimization details are poorly explained. This fact hinders developers from further analyzing the bottlenecks of the CNN-based applications on the edge devices. Hence, the paper comprehensively analyzes the deployment optimization methods for the CNN-based applications on the edge devices. Optimization methods are classified into the Hardware-Agnostic and Hardware-Specific methods. Their ideas and processing details are analyzed, and some suggestions are proposed according to the deployment experiments with different architecture models.

Keywords: Convolutional neural networks; edge device deployment; model pruning; deployment optimization methods; quantization; channel pruning; knowledge distillation

Analiza optimizacijskih metod za izvedbo konvolucijskih nevronskih mrež v vgrajenih sistemih

Z razvojem umetne inteligence na vgrajenih sistemih interneta stvari narašča povpraševanje po izvedbi konvolucijskih nevronskih mrež (CNN) v napravah z omejenimi viri. Orodja za implementacijo nevronskih mrež vključujejo različne optimizacijske metode. Zaradi hitrega razvoja na tem področju ni enotnih in dobro razloženih optimizacijskih postopkov za vgrajene sisteme, kar otežuje analizo ozkih grl aplikacij umetne inteligence. Prispevek celovito analizira optimizacijske metode izvedb nevronskih mrež vgrajenih sistemov. Metode optimizacije smo analizirali in razvrstili v strojno neodvisne in strojno specifične metode. V zaključku podajamo predloge implementacij z različnimi arhitekturnimi modeli.

1 INTRODUCTION

The Artificial Intelligence of Things (AIoT), a promising integrated technology that combines artificial intelligence and the Intelligence of Things, plays an increasingly significant role in every aspect of life [1][2]. CNNs are highly regarded among the artificial intelligence technologies due to their impressive performance in a variety of applications such as object recognition [3][4][5], healthcare [6][7], image generation[8] and anomaly detection [9][10][11]. The recent trend is to deploy CNN-based applications to the edge devices to achieve faster feedback [12].

Reducing the latency of huge CNN-based applications on the edge devices has been considered important and challenging [13]. Optimization is commonly required before deploying CNN-based applications on the edge devices. As a result, a wide range of deployment optimization methods have been integrated into the official deployment tools of each device. We divide the CNN deployment optimization methods into Hardware-Agnostic methods and Hardware-Specific methods. The Hardware-agnostic methods, such as channel pruning, knowledge distillation, etc., can be widely used on most edge platforms. They are characterized by the compatibility with other optimization methods.

The other optimization methods consider the hardware architecture and specifically ameliorate the CNN algorithm. They are defined as the Hardware-Specific methods. Such methods take the hardware level knowledge deeply into account. For example, they are concerned with reducing the cost of the memory access. The development speed gap between processors and DRAM

Received 16 April 2024
Accepted 3 June 2024

memory is huge [14] [15]. When instructions reference the memory, the system response is significantly delayed due to the slow memory speed [16]. At this point, reducing the memory access cost in applications is no less important than reducing the computation cost.

Accordingly, the Hardware-specific methods commonly insert multiple processes in the CNN algorithms to achieve targets such as data reuse and transmission cost reduction. Understanding the details of these deployment flows is critical to analyzing a specific behavior and bottlenecks of CNN algorithms on the edge devices.

However, mastering the existing deployment flows can be challenging. The mainstream deployment process can be summarized as (1) adopting mature CNN libraries such as PyTorch and TensorFlow to design and train models, (2) converting them into the computation graph format with interoperability between different frameworks, (3) utilizing the deployment tool of the target device to optimize the computation graph and generate executable code. Since CNN and AIoT have been rapidly developing in recent years, related deployment tools are not mature, deployment optimization flows are constantly changing, and relevant documents may not explain the optimization process in detail, which hinders the development of the AIoT technology.

Therefore, the paper comprehensively analyzes the deployment optimization methods for the CNN-based applications on the edge devices. The analyzed methods consist of the Hardware-Agnostic methods, including model pruning, knowledge distillation, neural architecture search, and the Hardware-Specific methods, including computation graph optimization, image-to-column, data reuse, and quantization. The paper not only explains the deployment process of a platform but also analyzes the optimization strategy across multiple platforms. Its aim is to help relevant application developers to further analyze the bottlenecks of the CNN algorithms on the devices.

The paper is organized as follows: Section 2 reviews of the mainstream CNN models briefly. Section 3 describes the Hardware-Agnostic methods. Section 4 presents hardware-specific methods and deployment details. Section 5 deploys models with different architectures are deployed on the edge devices, and offers some suggestions for the model design. Section 5.3 offers conclusions. Figure 1 shows an overview of the paper.

2 CNN MODELS

We first explore the background of the CNN models. CNNs are made of several layers stacked one on top of the other. Each layer accepts the previous CNNs layer output and transfers the processed data to the following layer. The output of each layer is called a feature map. Its elements are called activations. CNNs mainly comprise convolutional ($Conv$) layers which specialize in capturing local features.

Let $\circledast$ be the convolution operation, b the bias, W the filter, k the size of the filter kernel, h, and w the height and width of the input feature map. With the input feature map $\mathbf{M}^{In}$ at channel $\{0, ..., C\}$, the output feature map at the i^{th} channel of the $Conv$ layer is:

$$\mathbf{M}_i^{Out} = \sum_m^C \mathbf{M}_m^{In} \circledast W_i + b \quad (1)$$

$$\mathbf{M} \circledast W = \sum_p^{h-k-1} \sum_q^{w-k-1} \mathbf{M}_{(p,q)} \cdot W \quad (2)$$

Since computing one output feature map requires many matrix multiplication operations on all input feature maps, the $Conv$ layer is resource-consuming [17].

There is a trend in recent studies that the proposed model architectures are based on one or two block structures, such as SqueezeNet [18], EfficientNet [19] and GoogLeNet [20]. Each block structure consists of multiple layers in a defined sequence. Several model variations are generated by adjusting the channel number and inserting the block structures. Among them, the deeper and wider variant is used for high-precision tasks, whereas the compact variant is preferred for a lightweight deployment. With this approach, the proposed model can be applied to different circumstances. In the next section, the common CNN models VGG [21], ResNet [22], MobileNetV2 [23] and their block structures are introduced.

2.1 VGG

The model of the VGG series achieves excellent results in the 2014 ImageNet Large Scale Visual Recognition Challenge competition. The main contribution of the VGG series is demonstrating that increasing the depth of the model improves the performance. In this work, 3 $\times$ 3 $Conv$ kernels are adopted instead of the large-size kernels, which increases the model depth and reduces the parameters.

The VGG blocks are the main component of the VGG model and are shown in Figure 2. The VGG block consists of two Conv-BN-Act structures and one pooling layer. The Conv-BN-Act structure is composed of one $Conv$ layer, one Batch Normalization (BN) layer [24], and one activation layer in sequence. In the VGG block, $Conv$ layers are set up as 3×3 kernels with one-pixel padding and no bias. Thus, the feature maps remain the same size in the VGG block before the pooling layer, with only the dimensions being increased.

Two other VGG block structures differ only by the number of the Conv-BN-Act structures. They contain three and four Conv-BN-Act structures, respectively.

The VGG model has three linear layers for classification at the end. Regarding the width, each VGG block has twice the number of the output channels as the previous one, i.e. up to 512. Therefore, the channel is increased when the feature map size is reduced. Several

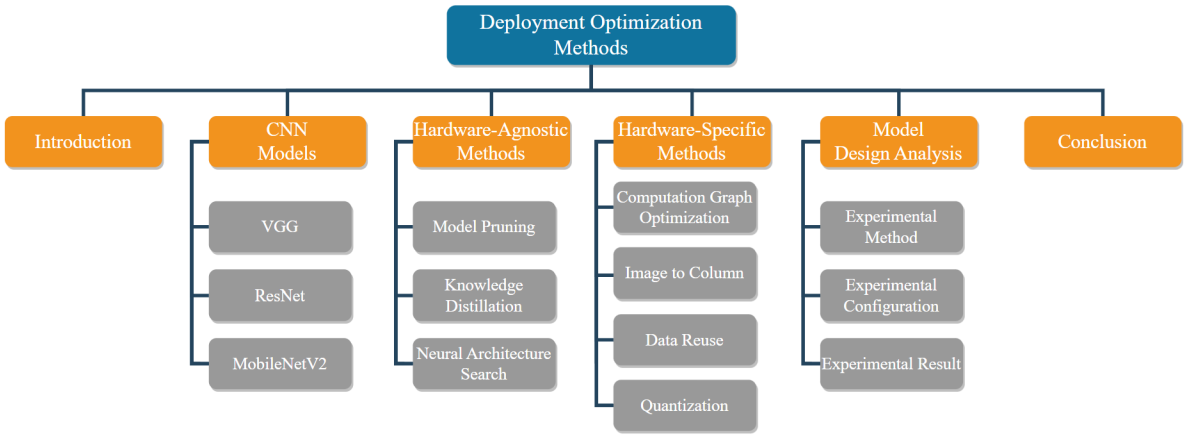


Figure 1. Overview of the paper.

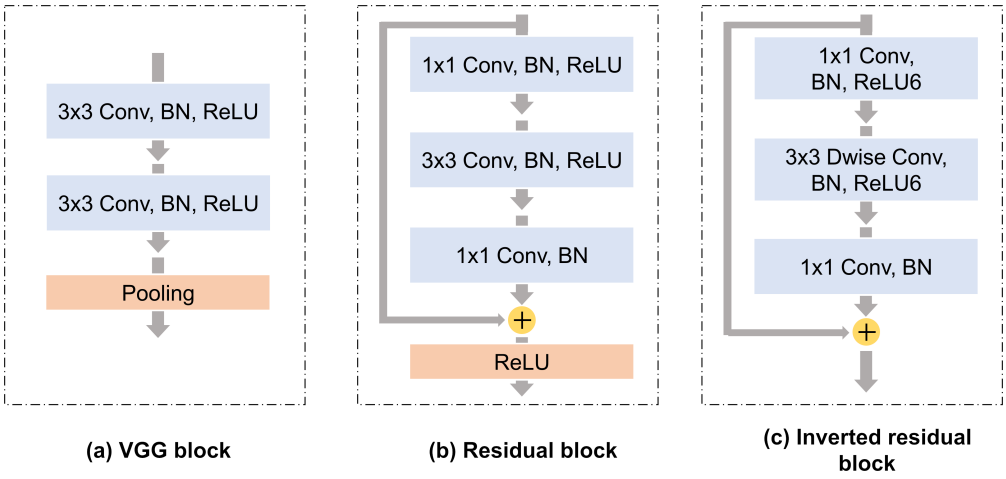


Figure 2. Diagram of common block structures.

model designs reference the Conv-BN-Act structure and VGG architecture.

2.2 *ResNet*

The success of the VGG series proves the performance of the deep models. However, the deep models are hindered by the gradient vanishing problem [25][26][27]. To mitigate the Vanishing Gradients problem, He et al. [22] propose a residual connection and the ResNet models (See Figure 2).

The residual block consists of the main path and the residual connection. The main path is a sub-network composed of three Conv-BN-Act structures. The first and third Conv-BN-Act structure employ 1×1 kernels for the feature dimension adjustment. The second Conv-BN-Act structure extracts the feature information, equipped with a 3×3 kernel. As a result, the second $Conv$ layer has more Floating Point Operations (FLOPs) than the others. The residual connection adds the input directly to the output of the main path.

In the residual block, the residual connection and the main route converge before the last ReLU layer. Let x denote the input of the residual block, $f()$ and $\mathbb{R}()$ the processing of the main path and ReLU, respectively. Then, final output $H(x)$ of the structure can be computed as:

$$H(x) = \mathbb{R}(f(x) + x) \tag{3}$$

Introducing residual connections mitigates the convergence difficulties of deep models training [28]. Therefore, residual connections are common in the following model design.

2.3 *MobileNetV2*

MobileNetV2 is proposed by Sandler et al. [23]. It designed for the CPU-based device inference such as the edge devices. The main proposal for MobileNetV2 is the inverted residual block (IR block) structure shown in Figure 2.

The IR block is designed to reduce FLOPs and improve the inference speed. Its structure is very similar to the residual block, and both consist of three $Conv$

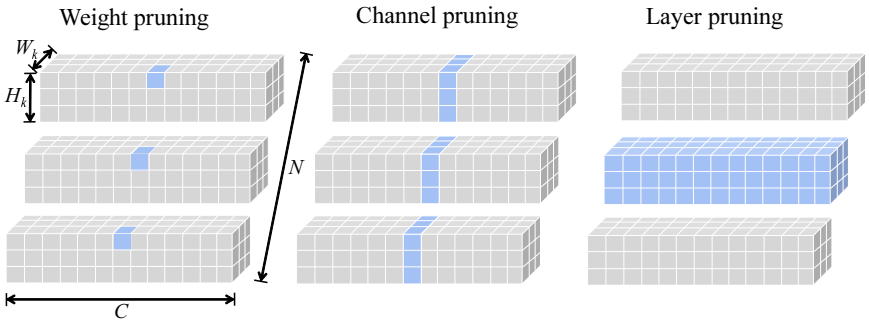


Figure 3. Model pruning methods.

layers with kernels of 1×1, 3×3, 1×1. Note that the feature dimension is first expanded and then compressed. The second $Conv$ layer adopts a grouped convolution where each filter processes only one input feature map, defined as a Depthwise Separable Convolution ($Dwise$). It is designed to achieve a high recognition performance with low FLOPs. $Dwise$ has a good separability, which is friendly to the CPU. However, accelerating $Dwise$ requires an exceptional support which is introduced in Section 4.1.3.

The activation layer adopts ReLU6 instead of ReLU in the IR block. ReLU is discarded at the last Conv-BN-Act structure, indicating that the block channel compression process is linear.

3 Hardware-Agnostic Methods

Due to the constraints of the hardware resources of edge devices and the prevalent redundancy problem of deep learning models, it is challenging to deploy CNN models directly on edge devices. CNN optimization provides handles to solve the problem and has attracted an extensive focus. Currently, there are many optimization methods for CNN, but only some of the schemes are designed for the hardware deployment. Low-rank decomposition schemes, for example, are algorithmic-level optimization tools that mathematically reduce the number of the parameters in the model and simplify the computational process [29]. This optimization is hardware-independent and does not directly target the utilization of hardware resources. The chapter focuses on model optimization schemes that are weakly correlated with hardware, including three main approaches: model pruning, knowledge distillation and neural architecture search (NAS).

3.1 Model Pruning

As one of the most frequently used model optimization methods, model pruning mainly aims to streamline CNN by pruning unimportant parameters. It generally consists of four steps: (1) Identifying redundant parameters, (2) Evaluating the importance, (3) Pruning the low-impact parameters, and (4) Model fine-tuning. Pruning CNNs can achieve compactness of the model parameters and directly improve the inference efficiency on hardware. From the perspective of the pruning granularity, pruning can be divided into two types: coarse-grained and fine-grained.

Coarse-grained pruning removes individual weights from the model with no regard to where the weights are located in the model or which particular filter, neuron, or layer they belong to. The method causes the weight matrix to become sparse. Thus, an ordinary hardware may not be able to accelerate these sparse computations efficiently. Conversely, fine-grained pruning simplifies the model structure by removing all the $Conv$ filters, neurons, channels, or even layers. After pruning using this method, the model is more compact and easily accelerated on hardware. Specifically, the optimization of the CNN models can be further broadly classified into the following types: weight pruning, channel pruning, and layer pruning.

Weight pruning: Weight pruning reduces the complexity and computational requirements by removing the connections with less weight in the model. LeCun et al. [30] propose to utilize the information of the second-order derivatives of the objective function to determine the importance of each weight in the model and then remove the redundant parameters. Ultimately, it improves the generalizability and the speed of running the model. To prune the model while maintaining the accuracy, Han et al. [31] filter the model from the dense to the sparse layers by training to eliminate connections below a weight threshold. The model is retrained to adjust the remaining connection structure, thus restoring the model accuracy. To address the problem of a limited energy supply in the edge devices, Yang et al. [32] propose an energy-aware pruning technique that prioritizes the pruning of the energy-dense layers and improves the weights to achieve an optimal energy reduction. Eventually, significant energy savings are demonstrated in models such as AlexNet and GoogLeNet.

Channel pruning: Compared to weight pruning,

channel pruning focuses more on unimportant or redundant channels in the model. The method not only reduces the number of the non-zero parameters of the model but also the computational complexity. After channel pruning, the model is more suitable for running on resource-constrained edge devices. Before channel pruning, it is crucial to identify the importance of the channels. Polyak et al. [33] propose pruning the input channels based on their contributing variance to reduce the computational complexity of the filters with no significant loss of the accuracy. While traditional studies tend to focus on the analysis of filter weights, He et al. [34] advocate the exploitation of the redundancy within the feature map. It first performs channel selection via LASSO regression and then reconstructs the feature map using linear least squares. The method, which relies on tensor factorization to improve the feature map reconstruction, is accomplished to accelerate models such as VGG and ResNet with a minimal loss of the accuracy.

Layer pruning: Unlike channel pruning, which reduces the complexity by reducing the width of the layers, layer pruning removes the entire redundant layers. In general, layer pruning reduces the depth of the model, which is more friendly to the deployment and acceleration of the model on the edge device, because CNN executes each layer sequentially during inference. However, hardware parallel processing can accelerate different channels running on the same layer. Therefore, fewer layers tend to result in a faster execution. But the model performance should not be neglected. To maximize the model resource use, a combination of the layer pruning and filter pruning is employed by Jordao et al. [35]. The importance of each CNN layer is assessed by using partial least squares, which allows for the pruning of CNNs to a resource-efficient depth while taking into account the constraints on the model size. Although pruning the layers reduces the model complexity, it destroys the model correlation to some extent. Therefore, Li et al. [36] propose a hardware-adaptive model optimization method for hardware characteristics and model layer redundancy. The method fully utilizes the parallel processing capability of the hardware and achieves significant pruning effects in both model parameters and FLOPs.

3.2 Knowledge Distillation

In contrast to pruning, which focuses on a systematical removal of unimportant weights, channels, or layers, the main aim of the knowledge distillation approach is to compress and transfer knowledge from a larger, complex model (teacher) to a smaller, more compact model (student). It focuses on how the student model is trained, specifically using the teacher model soft outputs to guide the student model training, ultimately effectively compressing the knowledge into a more resource-efficient framework. The concept of knowledge distillation is first introduced by Buciluǎ et al. [37] and popularized by the Hinton team [38]. They demonstrate the performance of knowledge distillation in improving task such as speech recognition, showing great potential for deploying high-performance models in resource-limited environments.

To further improve the performance of the student models across various datasets and CNN architectures, Zagoruyko et al. [39] suggest combining activation-based attention mechanisms with knowledge distillation. Ultimately, consistent improvements are shown across various datasets and CNN architectures. Chen et al. [40] introduce a trainable framework for a multi-class object detection using knowledge refinement and cue learning. It addresses the challenge of maintaining the accuracy while compressing the model to improve the speed. Unlike the traditional approach, which focuses on the size of the neuron response, Heo et al. [41] are more concerned with whether the neuron is activated or not. They propose a novel model activation loss transfer method that focuses on emphasizing the boundary of the neuron transfer activation between the teacher model and the student model. They also propose an alternative loss similar to the hinge loss of SVM that allows the student model to learn the activation boundary successfully.

3.3 Neural Architecture Search

NAS designs the neural network architectures automatically to optimize the performance for specific tasks. NAS focuses on discovering the optimal network structure from a broad search space, using algorithms to evaluate and select the best performance model. This is not the case with the traditional compression methods, which typically reduce the size and computational requirements of pre-designed models through techniques such as pruning and knowledge distillation. NAS simplifies and improves the model development process by automating architectural decisions. It is first proposed by Zoph et al. [42] and its performance is verified on two datasets, CIFAR10 [43] and Penn Treebank language modeling.

With limited computational and storage resources in the edge devices, many studies have maximized the search for compliant CNN architectures using NAS variants by balancing the two evaluation criteria of the performance and latency. He et al. [44] present an AutoML for the model compression that utilizes reinforcement learning to automatically compress models, improving the accuracy and speed on resource-limited devices without human intervention. Anderson et al. [45] combine NAS and hardware architecture information to improve the performance on keyword recognition tasks on the edge devices. In 2019, the EfficientNet is presented by Tan et al. [19]. The method utilizes NAS to develop a new baseline network and simultaneously balances the depth, width, and resolution of the network

through composite coefficients, which ultimately significantly improves the model efficiency and accuracy. The following year, the team builds on EfficientNet by introducing a weighted bidirectional feature pyramid network for enhanced multi-scale feature fusion, as well as a novel composite scaling method that uniformly scale the network dimensions [46]. The final results show that EfficientNet achieves higher accuracy and efficiency under various resource constraints.

4 HARDWARE-SPECIFIC METHODS

In this section, the Hardware-Specific methods are presented. They are designed by device developers based on the architecture of the processors. CNN algorithms are modified to achieve an efficient inference on the device, the modification methods are specialized for a certain kind of hardware. Some of the common methods are analyzed and details of their modification are explained.

4.1 Computation Graph Optimization

In most of the development processes for the CNN-based applications, CNN models are first designed on desktop computers and then implemented on the edge devices. Many mature software frameworks are on desktop computers, such as TensorFlow [47], PyTorch [48], and MXNet [49]. Because of the memory and power constraints and difference in architectures, these frameworks are unsuitable for the edge devices. Therefore, the deployment is to convert the CNN from a software framework to the corresponding framework of the device. In the process, the CNN algorithm is first mapped to a computation graph, with each layer operation viewed as a node and the data as an edge, then the computation graph is optimized. Common graph optimization methods are introduced in the following sections.

4.1.1 Batch Normalization Fusion: The BN layer is a common component of the CNN models, usually inserted following the $Conv$ layer to normalize the output feature map, which reduces optimization difficulties. Let γ_i and β_i denote the weight and bias of the BN layer on the i^{th} channel, μ_B and σ_B are the batch mean and variance, ϵ is an arbitrarily small constant. For the i^{th} channel input $\mathbf{M}_i$, the output of the BN layer is computed as:

$$BN(\mathbf{M}_i) = \gamma_i \frac{\mathbf{M}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta_i \tag{4}$$

During inference, μ_B and σ_B are the constant values. Therefore, the BN layer is considered as a linear process and can be fused into the previous or next node, typically $Conv$ and linear nodes. Equation 5 shows the updated weight $\hat{W}$ and bias $\hat{b}$ of the fused node. After updating, the BN nodes are deleted with no harm.

$$\begin{aligned} \hat{W}_i &= \frac{\gamma_i W_i}{\sqrt{\sigma_B^2 + \epsilon}}, \\ \hat{b}_i &= \beta_i - \frac{\gamma_i \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}. \end{aligned} \tag{5}$$

4.1.2 Activation Layer Fusion: The activation layers are typically fused at the output with preceding complex nodes, such as $Conv$, $Dwise$, and pooling nodes. For example, the most common ReLU node is converted to the max operations and performed before saving the calculation results to the memory.

4.1.3 Depthwise Separable Convolution Optimization: The $Dwise$ layer is a special case of $Conv$, which divides the convolution into multiple groups, each processing only one input feature map. In the traditional convolutional, calculating each activation in the output feature map requires partial feature maps on all input channels. When the memory capacity is not enough, the feature map should be read multiple times, which leads to an extra memory access cost. Each convolutional task of a $Dwise$ node is independent and has the desired parallelism. Therefore, there is a higher degree of freedom when it comes to memory scheduling and core task allocation. As a result, many deployment tools design operators specifically for the $Dwise$ nodes to take advantage of its separability.

4.2 Image to Column

The $Conv$ layer is recognized as a part of CNN that consumes the most computational power and resources [50]. Specifically, convolution extracts features by sliding multiple convolution kernels over the input feature map and performing a weighted summation operation on each local region [51]. The method requires frequent memory accesses to read the input data and weights and store the output, resulting in a significant latency.

Image to column ($im2col$) is a proposal to solve the problem. Different from traditional $Conv$, $im2col$ realizes $Conv$ by a dot product [52]. Specifically, the input feature maps and filter are unfolded (See Figure 4). Computing an output activation requires multiple sub-matrices with the same coordinates taken from all input feature maps. Each sub-matrix has the same size $k \times k$ as the convolution kernel. In $im2col$, each sub-matrix is the loaded and expanded to a vector by the input channel, and the input vector buffer of size $k \times k \times channel$ is obtained. Let the input buffer corresponding to the output (x, y) be denoted as $im2col(x, y)$. Similarly, the filter on the N^{th} input channel is expanded to the vector by the input channel, and a weight buffer $F(N)$ is obtained.

Consequently, the activation (x, y) of the N^{th} output feature map is calculated as:

$$O(x, y)^N = \mathrm{dot}(F(N), im2col(x, y)) \tag{6}$$

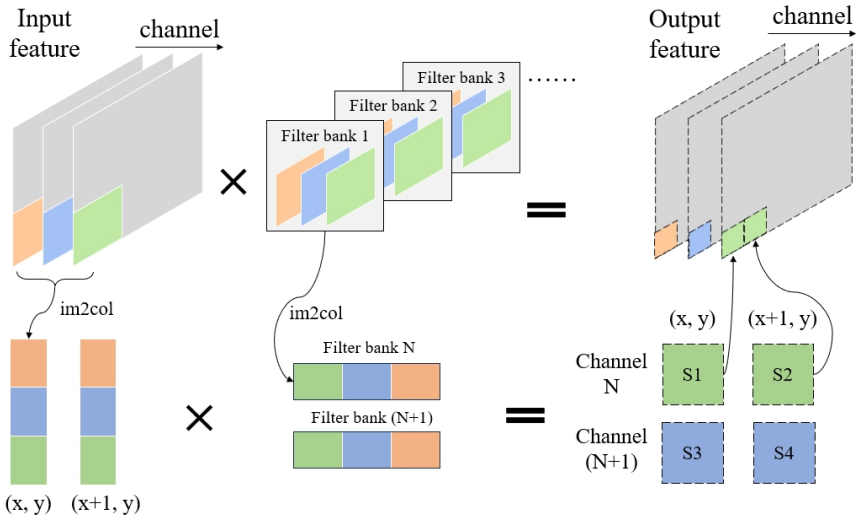


Figure 4. Image to column: converts convolution to matrix multiplication to speed up computation.

With this approach, convolution is converted to a dot product between $im2col(x, y)$ and $F(N)$. The Arithmetic Logic Unit (ALU) can simply load the corresponding elements from $im2col(x, y)$ and $F(N)$ and perform a Multiply-Accumulate (MAC) operation. The entire process has no branch operations or memory accesses; the computation is intensive, thus the execution is efficient.

To summary, the function of $im2col$ is to store the required data in the continuous memory during the calculation. The approach facilitates a direct loading of the data in the desired format at one time, effectively reducing the number of the memory accesses and thus reducing the computation time. Moreover, the access of $im2col$ is regular, so it can be combined with SIMD instructions to realize an efficient parallel computation.

4.3 Date Reuse

As mentioned above reducing the memory access is important for the edge devices. To further reduce the number of the memory accesses, increasing the data reuse rate is effective [53]. An approach is to keep the same amount of the input and weight data in the cache. It is easy to notice that when computing the output with input coordinates of (x, y) and $(x + 1, y)$, the weight matrices are the same. Therefore, the weight cache can be reused.

To provide a clear explanation, the inputs and weights are processed by $im2col$. The $im2col(x, y)$ input, $im2col(x + 1, y)$ and weights $F(N)$, $F(N + 1)$ are loaded to the cache, the $O(x, y)^N$, $O(x + 1, y)^N$, $O(x, y)^{N+1}$ and $O(x + 1, y)^{N+1}$ activations can be calculated. Four MAC operations are accomplished with four load causing by data reuse; the load efficiency is 1 MAC/load. When the number of both input buffers and weight buffers is increased to four, the load efficiency reaches two MAC/load. Note that the input buffer here corresponds to different coordinates, and the weight buffer corresponds to different channels. If the number of buffers continues to increase proportionally, the load efficiency is even higher. In contrast, if the input buffer is seven and the weight buffer is one, the load efficiency is only one MAC/load.

In practice, the cache is insufficient to keep so much data in many edge devices. Therefore, the deployment tool would schedule data loads based on the size of the cache and memory to achieve a high load efficiency.

4.4 Quantization

Quantization is a mature CNN acceleration technique. CNNs parameter are typically saved in the 32-bit floating point (FP32) during training. In quantization, parameters are stored with a lower bit precision, such as 16-bit integer (INT16) and 8-bit integer (INT8). Since the data precision is reduced, the memory overhead of the parameter storage and the computational cost of the matrix multiplication are significantly reduced. Also, CNNs are demonstrated robust to quantization, so they can be quantified with a minor performance loss [54]. Hence, quantization is widely used in the CNN deployment processes. This section concentrates on a common quantization approach and the fundamentals of running quantified models on fixed-point accelerators.

In the quantization, the weights and activations of the model are mapped to a low-precision fixed-point quantified representations, usually INT8. Then, the quantified representations are stored and computed on the device to achieve acceleration. There are many proposals for quantization. A uniform affine quantization is presented as an example [55], as it is the most common. The method is defined by three quantization parameters: scale factor s, zero point z, and bit width bit. Scaling

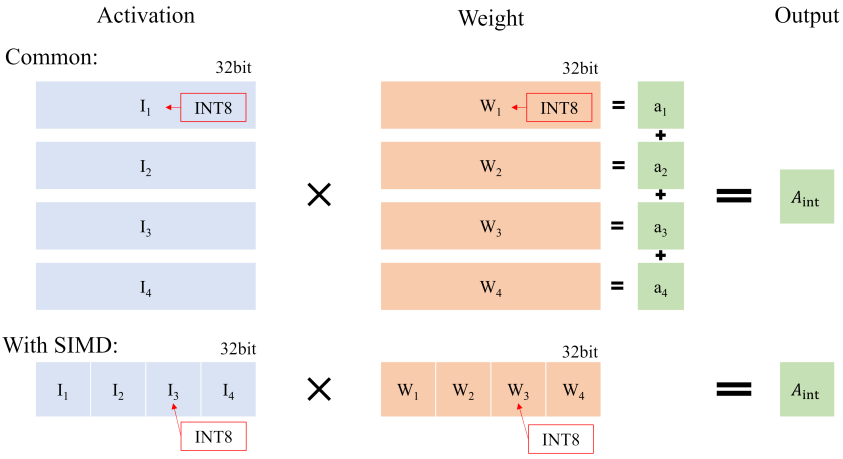


Figure 5. Diagram of the INT8 data MAC operation. $a_{\{1,2,3,4\}}$ denotes products stored in the same accumulator.

factors and zeros are employed to transform the floating-point values into integer values, where the scaling factor is a floating-point number and the zero point is an integer to determine the zero in the quantified representation.

The quantization process is explored based on these quantization parameters. The section shows an example of the inference of the INT8 quantified data on a 32-bit device. The convolution operation is simplified to multiple MAC operations. Given weight W and input activation I, output activation A is computed as:

$$A = \sum_m W_m \cdot I_m + b \tag{7}$$

Then taking activation as an example, mapping I to the integer value I_{int} is:

$$I_{\text{int}} = \text{clamp}(\frac{I}{s_i} + z_i; 0, 2^{bit} - 1) \tag{8}$$

The clamp operation is defined as:

$$\text{clamp}\,(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \le x \le c, \\ c, & x > c. \end{cases} \tag{9}$$

Commonly, the quantization parameters for weights and activations are separate [56], they are s_w and z_w for W_{int}. The scheme gives granularity to the quantization, and increasing the granularity reduces the accuracy loss due to quantization. According to Equation 8, the quantization parameters are computed on the whole tensor. Thus, the approach does not lead to a huge computation cost. Similarly, the quantization parameters are separated for each layer.

Now, the intermediate result of output activation A_{int} is obtained as:

$$A_{\text{int}} = W_{\text{int}} \cdot I_{\text{int}}. \tag{10}$$

Equation 10 is the main computation task on the devices. The calculations can be performed with fixed points. Thus, the cost of the memory access during reading and storage is low.

When SIMD (Single Instruction, Multiple Data) instructions are integrated into the device instruction set, the MAC executed per cycle can be significantly increased. Figure 5 shows the details of the MAC operations and the acceleration effect of the SIMD instructions. Obtaining each product requires one cycle to perform the multiplication of two INT8 data. Utilizing the SIMD instructions, four INT8 data can be stored in a 32-bit register, and four MAC operations are performed in one cycle. As a result, the MAC operation accelerate by 4×. Fortunately, nowadays, many CPU architectures support the SIMD instruction.

However, the difference between A_{int} and the correct activation A is huge. A_{int} should be de-quantified. Let $\widehat{A}$, $\widehat{W}$ and $\widehat{I}$ denote the de-quantification representations of A_{int}, W_{int}, I_{int}, which are approximate to the actual value. According to Equation 8, the de-quantification representations can be written as:

$$\widehat{I} = s_i(I_{\text{int}} - z_i) \approx I \tag{11}$$

$$\widehat{W} = s_w(W_{\text{int}} - z_w) \approx W \tag{12}$$

$$\widehat{A} = \sum_m \widehat{W}_m \cdot \widehat{I}_m + b \approx A \tag{13}$$

Combining the Equations 11 and 12, $\widehat{W}_m \cdot \widehat{I}_m$ is calculated as:

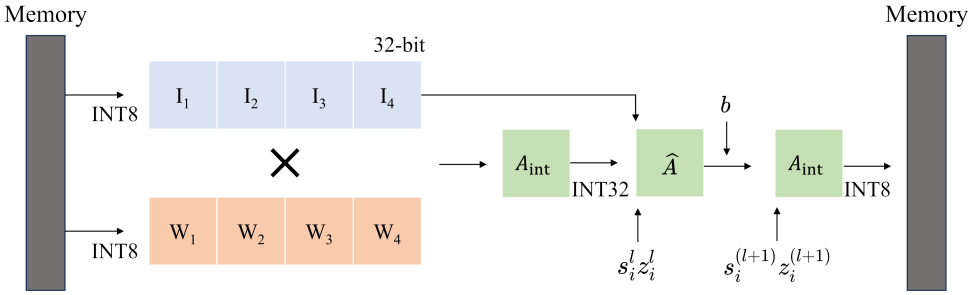


Figure 6. Diagram of the quantified model inference. $s_i^l z_i^l$ are the quantization parameters of the l^{th} layer

$$\begin{aligned}\widehat{W}\widehat{I} &= s_w(W_{\text{int}} - z_w)s_x(I_{\text{int}} - z_x) \\ &= s_w s_x W_{\text{int}} I_{\text{int}} - s_w z_w s_x I_{\text{int}} - s_w s_x z_x W_{\text{int}} \\ &+ s_w z_w s_x z_x \end{aligned} \tag{14}$$

Each value in the third and fourth term is determined before compilation, The compiler can pre-compute its result and combine it with the bias of the layer, so no added computational costs. The second term depends on unknown input I_{int}, i.e. an additional term should be computed in the inference. In the first term, all A_{int} are multiplied by the same floating point number, so it is computed after A_{int} are summed:

$$\begin{aligned}\widehat{A} &= \sum_m \widehat{W}_m \cdot \widehat{I}_m + b \\ &= s_w s_x \sum A_{\text{int}} - s_w z_w s_x \sum I_{\text{int}} + b \end{aligned} \tag{15}$$

As seen, a significant number of the intermediate calculations are accumulated in Equation15, leading to a high risk of overflowing A_{int}. Therefore A_{int} should be stored at INT32. Similarly, the bias is stored in INT32.

The activation $\widehat{A}$ stored in the accumulator should be stored in the memory before it is used by the next layer. To reduce the data transmission cost, $\widehat{A}$ is re-quantified to the low-precision. Since S_i is different for each layer, this step should be performed based on the quantization parameters of the next layer. In practice, de-quantization and re-quantization are combined into one process.

The Figure 6 shows the flow of the quantified model inference: (1) load the INT8 activations I_{int} and INT8 weights W_{int} from the memory and compute A_{int}. (2) de-quantify A_{int} by combining I_{int} and the quantization parameters $s_i^l z_i^l z_w^l$ of this layer, then re-quantify to INT8 by quantization parameters $s_i^{(l+1)} z_i^{(l+1)}$ of the next layer. (3) store the INT8 activations in the memory.

5 Model Design Analysis

The mature CNN deployment acceleration methods mitigate part of the resource-constrained problem on the edge devices. However, real-time CNN-based applications are still a challenge for the edge devices. Many studies focus on designing models with the low latency and high accuracy. In this section, we explore the relationship between the model architecture and accuracy, model FLOPs, and latency.

5.1 Experimental Method

A basic CNN framework with a reference to ResNet and MobileNetV2 (See Table 1). It consists of three parts: In-conv, Blocks and Classify. The In-conv contains a Conv-BN-Act structure where ReLU is employed as an activation layer. The Block consists of four block structures stacked on top of each other. The block structure is selected from the VGG block, residual block and IR block introduced in Section 3. The channels of each alternative block structure are carefully tuned so that models generated from the three block structures have similar FLOPs. Adjustments of the width and depth take place in the Block part. When the depth is increased by 1, each block structure is duplicated and inserted behind itself. The width is the scaling factor applied to the output channel number of each convolutional layer in the block structure. Regardless of the depth, the feature map size reduction occurs at the beginning and middle of the Block part. The Classify part consists of one average pooling layer and one linear layer that classifies the output of the Block part. The models are generated based on a preset width, depth, and selected block structure.

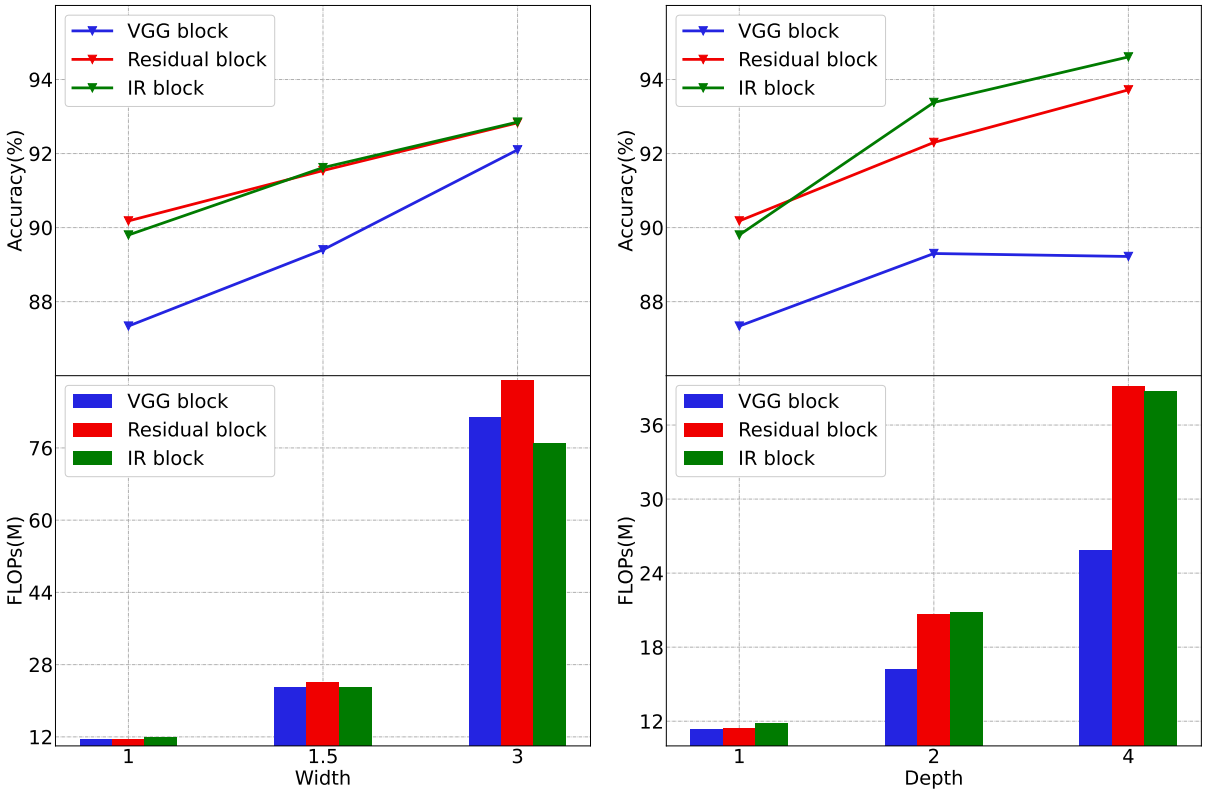
In the experiments, multiple sets of the width and depth are applied. Models are generated with three block structures, respectively, and the accuracy and latency of the models are measured. The models with their width and height set to one are considered the baseline models.

5.2 Experimental Configuration

Training configuration: CIFAR10 is adopted as an experimental dataset. CIFAR10 contains 50000 training images and 10000 test images with the size of 32 $\times$ 32. The training process consists of five epochs of warm-up and 800 epochs of training. The cosine learning rate schedule is employed with an initial value of 0.02, momentum of 0.9, and weight decay of 0.0005. The

Table 1. Proposed CNN framework. [3×3,16] denotes a convolutional layer with kernel size 3 and output channel 16.

Name	Output size	Architecture		
		VGG block	**Residual block**	**IR block**
In-conv	32×32	Conv-bn-ReLU, [3×3,16], stride=1		
Block 1	16×16	[3×3,16] [3×3,16] × Depth	[1×1,24] [3×3,24] × Depth [1×1,80]	[1×1,128] [3×3,128] × Depth [1×1,32]
Block 2	16×16	[3×3,16] [3×3,16] × Depth	[1×1,24] [3×3,24] × Depth [1×1,80]	[1×1,128] [3×3,128] × Depth [1×1,32]
Block 3	8×8	[3×3,32] [3×3,32] × Depth	[1×1,48] [3×3,48] × Depth [1×1,128]	[1×1,192] [3×3,192] × Depth [1×1,64]
Block 4	8×8	[3×3,32] [3×3,32] × Depth	[1×1,48] [3×3,48] × Depth [1×1,128]	[1×1,192] [3×3,192] × Depth [1×1,64]
Classify	1×1	Average pool		
		Linear		



(a) Width adjustment results.

(b) Depth adjustment results.

Figure 7. Results of the accuracy analysis experiments. The legend indicates the adopted block structure.

Table 2. Latency of the models with the same FLOPs on experimental platforms. (↑) indicates the percentage increase in the latency compared to the baseline model.

Block	Depth	Width	FLOPs (M)	Latency(ms)		
				FPGA (↑)	GAP8 (↑)	CPU (↑)
VGG	1	1	11.35	0.11	10.29	2.12
	4	1	25.82	0.15 (38.5%)	24.34 (136.6%)	5.86 (177.1%)
	1	1.6	25.15	0.20 (79.8%)	27.83 (170.5%)	2.87 (35.8%)
Residual	1	1	11.43	0.22	12.43	2.92
	3	1	29.88	0.43 (96.3%)	30.00 (141.3%)	7.07 (141.9%)
	1	1.7	29.93	0.31 (43.3%)	30.53 (145.5%)	3.88 (32.8%)
IR	1	1	11.85	0.21	10.21	3.58
	2	1	20.82	0.33 (60.4%)	17.30 (69.4%)	6.49 (81.5%)
	1	1.4	20.12	0.26 (23.2%)	15.80 (54.8%)	4.27 (19.3%)

models generated with each setting are trained four times from the initial. The one with the highest accuracy is taken as the result.

Deployment details: All the training processes are performed on Nvidia GeForce GTX 3080 Ti GPU and Intel i9-10900 CPU by PyTorch.

Regarding, the latency measurement, the generated models are implemented on GAP8 [57], Field Programmable Gate Array (FPGA) ZCU102, and desktop CPU i7-9700. The official deployment flow of each device is adopted in the experiments. GAP8 is an IoT application processor based on the RISC-V and PULP platform [58], developed by GreenWaves Technologies, which is featured by a low power consumption and parallel processing. For the deployment on GAP8, the generated models are quantified to INT-8 and compiled by NNTOOL and AutoTiler [59], then simulated on GVSoC [60]. The measured working time of the cluster cores is taken as the latency.

For the deployment on FPGA ZCU102, the generated models are quantified to INT-8 and compiled into the Xmodel with the Vitis-AI deployment environment [61]. It includes optimized IP cores, AI Quantizer for quantifying CNNs, and AI Compiler for optimizing and compiling CNNs computation graph. In the experiments, the software execution time of ZCU102 is regarded as the latency of the CNN application.

For the Intel i7-9700 desktop CPU platform, the generated models are implemented by PyTorch without Quantization. The latency is measured by the PyTorch Profiler. After 50 times of warm-up, each model performs 50 times of inference, and the average latency is adopted.

5.3 Experimental Result

First, the relationship between the accuracy and model architecture is explored. Figure 7a shows the change in the accuracy and FLOPs when increasing the width of the three baseline models to 1.5, 3, and Figure 7b shows the change in the accuracy and FLOPs when adjusting the depth of the three baseline models to 2, 4.

As seen, increasing both the width and depth of the model improves the accuracy. The effect of the depth on the accuracy is more obvious. In the model with IR block, the accuracy is improved from 89.80% to 94.61%, while the the adjustment of width improves only by 92.85%. In terms of the FLOPs, except for the VGG block, the FLOPs of the models with the depth of 2 are only about 1/4 of the FLOPs with the width of 3, while they have a similar accuracy.

It is worth noting that the accuracy of the model with the VGG block decreases instead after adding depth. This is due to the gradient vanishing problem explored above.

Next, the relationship between the latency and model FLOPs is explored. The width and height of the three baseline models are increased, so that the adjusted models have close FLOPs. The models are implemented on FPGA ZCU102, GAP8, and desktop CPU, and the inference latency is measured.

Table 2 shows the latency of each model and the increased percentage compared to the baseline model. For CPU, the models with an increased depth have much more latency than the models with an increased width, up to a 177.1% increase in the latency compared to the baseline model. Moreover, for the models adopting the Residual block and IR block, creasing depth results in increased latency on all platforms except GAP8. For the model with VGG blocks, increasing the width results in increased latency on FPGA and GAP8, while the results are reversed on CPU.

Despite the lack of experimental results on more platforms with more model architectures, our observations demonstrate the following: (1) For models with residual connections, increasing the number of the block structures improves the accuracy. (2) There is no direct relationship between FLOPs of the model and its latency, the model latency on the devices should not be analyzed based only on FLOPs, but also on the model

architectures.

CONCLUSION

With the rapidly development of AIoT and CNN technologies, the demand for implementing CNN applications on the edge devices is rising. Since the CNN algorithms are considered huge for the edge devices, several CNN optimization methods are integrated into the device deployment tools. Due to the rapid development, the optimization process adopted by the deploy tools is non-uniform, and the details are poorly explained. Hence, the paper provides a comprehensive analysis of the deployment optimization methods for the CNN-based applications on the edge devices. The paper analyzes the Hardware-Agnostic methods, including model pruning, knowledge distillation, neural architecture search and the Hardware-Specific methods, including computation graph optimization, image to column, data reuse quantization. Based on the results of training and deployment of several architectural models, suggestions for the model design are presented.

REFERENCES

[1] Z. Chang, S. Liu, X. Xiong, Z. Cai, and G. Tu, "A survey of recent advances in edge-computing-powered artificial intelligence of things," *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13849–13875, 2021.

[2] Z. Ward, J. Miller, J. Engel, M. A. S. Masoum, M. Shekaramiz, and A. Seibi, "Fuzzy-based image contrast enhancement for wind turbine detection: A case study using visual geometry group model 19, xception, and support vector machines," *Machines*, vol. 12, no. 1, 2024.

[3] X. Yue, H. Li, and L. Meng, "An ultralightweight object detection network for empty-dish recycling robots," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–12, 2023.

[4] X. Yue and L. Meng, "Yolo-msa: A multi-scale stereoscopic attention network for empty-dish recycling robots," *IEEE Transactions on Instrumentation and Measurement*, 2023.

[5] X. Yue, H. Li, M. Shimizu, S. Kawamura, and L. Meng, "Yolo-gd: A deep learning-based object detection algorithm for empty-dish recycling robots," *Machines*, vol. 10, no. 5, 2022.

[6] Y. Ge, Z. Li, X. Yue, H. Li, Q. Li, and L. Meng, "Iot-based automatic deep learning model generation and the application on empty-dish recycling robots," *Internet of Things*, p. 101047, 2023.

[7] J. Ren, H. Li, A. Wang, K. Saho, and L. Meng, "Radar-based gait analysis by transformer-liked network for dementia diagnosis," *Biomedical Signal Processing and Control*, vol. 91, p. 105986, 2024.

[8] B. Lyu, X. Yue, and L. Meng, "Japanese literature organization and spatiotemporal database system creation for natural disaster analysis," *Heritage Science*, vol. 12, p. 14, Jan 2024.

[9] Z. Li, Y. Ge, X. Wang, X. Yue, and L. Meng, "Industrial anomaly detection via teacher student network," in *2023 International Conference on Advanced Mechatronic Systems (ICAMechS)*, pp. 1–5, IEEE, 2023.

[10] I. Bae and S. Lee, "A multi-input convolutional neural network model for electric motor mechanical fault classification using multiple image transformation and merging methods," *Machines*, vol. 12, no. 2, 2024.

[11] E. Kim, S. Jung, M. Kim, J. Kim, B. Kim, J. Kim, and S. Kim, "Anomaly detection using puzzle-based data augmentation to overcome data imbalances and deficiencies," *Machines*, vol. 11, no. 11, 2023.

[12] S. Mittal, "A survey on optimized implementation of deep learning models on the nvidia jetson platform," *Journal of Systems Architecture*, vol. 97, pp. 428–442, 2019.

[13] Veeramanikandan, S. Sankaranarayanan, J. J. P. C. Rodrigues, V. Sugumaran, and S. Kozlov, "Data flow and distributed deep neural network based low latency iot-edge computation model for big data environment," *ENGINEERING APPLICATIONS OF ARTIFICIAL INTELLIGENCE*, vol. 94, SEP 2020.

[14] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[15] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[16] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[17] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[18] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[19] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML, Long Beach, California, USA* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97, pp. 6105–6114, 2019.

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[21] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

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

[23] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 448–456, PMLR, 07–09 Jul 2015.

[25] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[27] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, *Efficient BackProp*, pp. 9–50. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[28] A. E. Orhan and X. Pitkow, "Skip connections eliminate singularities," 2018.

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

[30] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," *Advances in neural information processing systems*, vol. 2, 1989.

[31] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[32] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5687–5695, 2017.

[33] A. Polyak and L. Wolf, "Channel-level acceleration of deep face representations," *IEEE Access*, vol. 3, pp. 2163–2175, 2015.

[34] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.

[35] A. Jordao, M. Lie, and W. R. Schwartz, "Discriminative layer pruning for convolutional neural networks," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 828–837, 2020.

[36] H. Li and L. Meng, "Hardware-aware approach to deep neural network optimization," *Neurocomputing*, vol. 559, p. 126808, 2023.

[37] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 535–541, 2006.

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

[39] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," *arXiv preprint arXiv:1612.03928*, 2016.

[40] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, "Learning efficient object detection models with knowledge distillation," *Advances in neural information processing systems*, vol. 30, 2017.

[41] B. Heo, M. Lee, S. Yun, and J. Y. Choi, "Knowledge transfer via distillation of activation boundaries formed by hidden neurons," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3779–3787, Jul. 2019.

[42] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[43] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[44] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 784–800, 2018.

[45] A. Anderson, J. Su, R. Dahyot, and D. Gregg, "Performance-oriented neural architecture search," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 177–184, IEEE, 2019.

[46] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10781–10790, 2020.

[47] M. Abadi, "Tensorflow: learning functions at scale," *SIGPLAN Not.*, vol. 51, p. 1, sep 2016.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[49] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[50] H. Li, Z. Wang, X. Yue, W. Wang, H. Tomiyama, and L. Meng, "An architecture-level analysis on deep learning models for low-impact computations," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 1971–2010, 2023.

[51] Z. Li, H. Li, and L. Meng, "Model compression for deep neural networks: A survey," *Computers*, vol. 12, no. 3, p. 60, 2023.

[52] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[53] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[54] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[55] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," *arXiv e-prints*, p. arXiv:2106.08295, june 2021.

[56] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–37, 2020.

[57] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "Gap-8: A risc-v soc for ai at the edge of the iot," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–4, IEEE, 2018.

[58] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[59] GreenWaves-Technologies, "Nntool." https://github.com/GreenWaves-Technologies/gap_sdk/tree/master/tools/nntool.

[60] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pp. 409–416, 2021.

[61] V. Kathail, "Xilinx vitis unified software platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 173–174, Association for Computing Machinery, 2020.

Qi Li is a Ph.D. student at the Graduate School of Science and Engineering at Ritsumeikan University, Japan. He received his master's degree in engineering from Ritsumeikan University. His research interests include computer architecture, deep learn model compression, Internet of Things (IoT) and smart edge computing. He is a student member of IEEE.

Zhenling Su is a Ph.D. student at the Graduate School of Science and Engineering at Ritsumeikan University, Japan. His research interests include computer architecture, compact Artificial Intelligence (AI) model design, and High-Performance Computing (IHPC), especially in edge devices. He is a student member of IEEE.

Lin Meng is a professor at the College of Science and Engineering at Ritsumeikan University, Japan. He received his Ph.D. from the Graduate School of Science and Engineering at Ritsumeikan University in 2012. In 2015, he was a visiting scholar in the Dept. of CSE at the University of Minnesota, Twin Cities, USA. His research interests include Computer Architecture, Parallel Processing, IHPC, FPGA-based Accelerator Design, AI, IoT and more. He is a senior member of IEEE and a member of ACM, IPSJ, IEICE, and IEE.