# Resource-Efficient Convolutional Networks: A Survey on Model-, Arithmetic-, and Implementation-Level Techniques

JUNKYU LEE, Queen's University Belfast
LEV MUKHANOV, Queen's University Belfast and Queen Mary University of London
AMIR SABBAGH MOLAHOSSEINI, UMAR MINHAS, YANG HUA, and
JESUS MARTINEZ DEL RINCON, Queen's University Belfast
KIRIL DICHEV, University of Cambridge
CHEOL-HO HONG, Chung-Ang University
HANS VANDIERENDONCK, Queen's University Belfast

Convolutional neural networks (CNNs) are used in our daily life, including self-driving cars, virtual assistants, social network services, healthcare services, and face recognition, among others. However, deep CNNs demand substantial compute resources during training and inference. The machine learning community has mainly focused on model-level optimizations such as architectural compression of CNNs, whereas the system community has focused on implementation-level optimization. In between, various arithmetic-level optimization techniques have been proposed in the arithmetic community. This article provides a survey on resource-efficient CNN techniques in terms of model-, arithmetic-, and implementation-level techniques, and identifies the research gaps for resource-efficient CNN techniques across the three different level techniques. Our survey clarifies the influence from higher- to lower-level techniques based on our resource efficiency metric definition and discusses the future trend for resource-efficient CNN research.

CCS Concepts: • **Computing methodologies** → **Computer vision**; **Neural networks;**

Additional Key Words and Phrases: Convolutional neural networks, neural networks, resource efficiency, arithmetic utilization



This project received funding by the Engineering and Physical Sciences Research Council under grant agreements EP/T022345/1 (DiPET) and EP/V02860X/1 (RAPID), by CHIST-ERA under grant agreement CHIST-ERA-18-SDCDN-002 (DiPET), and by the Commission Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement 101031148 (SoftNum). This research was also supported by a Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (P0017011, HRD Program for Industrial Innovation).
Authors' addresses: J. Lee, A. S. Molahosseini, U. Minhas, Y. Hua, J. Martinez del Rincon, and H. Vandierendonck, Queen's University Belfast, University Road, Belfast, Northern Ireland, BT7 1NN, UK; emails: eecejk@gmail.com; {a.sabbaghmolahosseini, u.minhas, y.hua, j.martinez-del-rincon, h.vandierendonck}@qub.ac.uk; L. Mukhanov, Queen Mary University of London, Mile End Road, London, England, E1 4NS, UK; email: l.mukhanov@qub.ac.uk; K. Dichev, University of Cambridge, The Old Schools, Trinity Lane, Cambridge, England, CB2 1TN, UK; email: kiril.dichev@gmail.com; C.-H. Hong (corresponding author), Chung-Ang University, 84 Heukseok-ro, Dongjak-gu, Seoul, Republic of Korea; email: cheolhohong@cau.ac.kr.


0360-0300/2023/07-ART276
https://doi.org/10.1145/3587095

**ACM Reference format:**
JunKyu Lee, Lev Mukhanov, Amir Sabbagh Molahosseini, Umar Minhas, Yang Hua, Jesus Martinez del Rincon, Kiril Dichev, Cheol-Ho Hong, and Hans Vandierendonck. 2023. Resource-Efficient Convolutional Networks: A Survey on Model-, Arithmetic-, and Implementation-Level Techniques. *ACM Comput. Surv.* 55, 13s, Article 276 (July 2023), 36 pages.
https://doi.org/10.1145/3587095

## 1 INTRODUCTION

Recent improvements in network and storage devices have provided the machine learning community with the opportunity to utilize immense data sources, leading to the golden age of AI and deep learning [22]. Since modern **Deep Neural Networks (DNNs)** require considerable computing resources and are deployed in a variety of compute devices, ranging from high-end servers to mobile devices with limited computational resources, there is a strong need to realize economical DNNs that fit within the resource constraints [128, 160, 161]. Resource-efficient DNN research has vividly been carried out independently in various research communities including the machine learning, computer arithmetic, and computing system communities. Recently, DeepMind proposed the resource-efficient deep learning benchmark metric, which is the accuracy along with the required memory footprint and number of operations [84].

With this regard, this article surveys resource-efficient techniques for **Convolutional Neural Networks (CNNs)** based on the three-level categorization: the model-, arithmetic-, and implementation-level techniques along with various resource efficiency metrics as shown in Figure 1, since CNN is one of the most widely used DNN architectures [100]. Our resource efficiency metrics include the accuracy per parameter, operation, memory footprint, core utilization, memory access, and Joule. For the resource efficiency comparison between the baseline CNN and a CNN utilizing resource-efficient techniques, the accuracy should be equivalent between the two CNNs. In other words, it is not fair to compare the resource efficiency between a CNN producing a high accuracy and a CNN producing a low accuracy since the resource efficiency is significantly higher in a low-performing CNN based on our resource metrics. We categorize the resource-efficient techniques into the *model-level resource-efficient techniques* if they compress the CNN model sizes, the *arithmetic-level resource-efficient techniques* if they utilize reduced precision arithmetic and/or customized arithmetic rules, and the *implementation-level resource-efficient techniques* if they apply hardware optimization techniques to the CNNs (e.g., locating local memory near **Processing Elements (PEs)**) to improve physical resource efficiency such as the accuracy per compute resource and per Joule.

In Figure 1, CNNs can be considered as a resource-efficient technique since they improve the accuracy per parameter, per operation, and per memory footprint, compared to fully connected neural networks. The resource efficiency from CNNs can be further improved by applying the model-, arithmetic-, and implementation-level techniques. The model- and arithmetic-level techniques can affect the accuracy since they affect either the CNN model structure or the arithmetic rule, whereas the implementation-level techniques generally do not affect the accuracy. The model-level techniques mostly contribute to improving abstract resource efficiency, whereas the implementation-level techniques contribute to improve physical resource efficiency. Without careful consideration at the intersection between the model- and the implementation-level techniques, a CNN model compressed by the model-level techniques might require significant runtime compute resources, incurring longer training time and inference latency than the original model [31, 119]. Thus, to optimize the performance and energy efficiency on a particular hardware, it is essential to consider the joint effect of the model-, arithmetic-, and implementation-level optimizations.

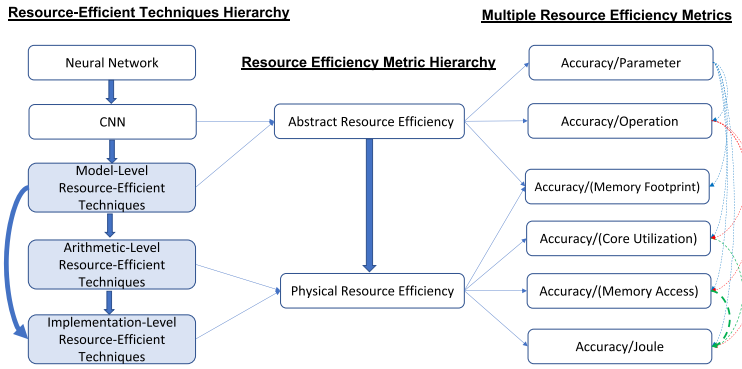


Fig. 1. Survey on resource-efficient CNN techniques based on resource efficiency metrics.

Related survey works are as follows. Sze et al. [155] provided a comprehensive tutorial and survey toward efficient processing of DNNs, discussing DNN architectures, software frameworks (e.g., PyTorch, TensorFlow, Keras), and the implementation methods optimizing **Multiply-and-Accumulate Computations (MACs)** of CNNs on given compute platforms. Cheng et al. [35, 36] conducted a survey on the model compression techniques including pruning, low-rank factorization, compact convolution, and knowledge distillation. Deng et al. [45] discussed joint model-compression methods that combined multiple model-level compression techniques, and their efficient implementation on particular computing platforms. Wang et al. [166] provided a survey on custom hardware implementations of DNNs and evaluated their performance using the Roofline model of Williams et al. [171]. Hoefler et al. [82] provided a survey on pruning techniques to generate sparse DNNs and a tutorial of how to train such sparse DNNs. Ghimire et al. [58] provided a survey on model compression methods and computing platforms suitable for accelerating CNNs.

Unlike the previous survey works, we conduct a comprehensive survey on resource-efficient CNN techniques in terms of the model-, arithmetic-, and implementation-level techniques by clarifying which resource efficiency can be improved with particular techniques according to our resource efficiency metrics as defined in Section 2.2. Such clarification would provide machine learning engineers, computer arithmetic designers, software developers, and hardware manufacturers with useful information to improve particular resource efficiency for their CNN applications. Besides, since we notice that fast wireless communication and edge computing development affects CNN applications [190], our survey also includes cutting-edge resource-efficient techniques for distributed AI such as early exiting techniques [160, 161]. The holistic and multi-facet view for resource-efficient techniques for CNN from our survey would allow for a better understanding of the available techniques and, as consequence, a better global optimization, compared to previous survey works. The main contributions of our article include the following:

- This article first provides a comprehensive survey coverage of the recent resource-efficient techniques for CNNs in terms of the model-, arithmetic-, and implementation-level techniques.
- To the best of our knowledge, our work is the first to provide a comprehensive survey on arithmetic-level utilization techniques for CNNs.
- This work utilizes multiple resource efficiency metrics to clarify which resource efficiency metrics each technique improves.
- This article provides the influence of resource-efficient CNN techniques from higher- to lower-level techniques (refer to Figure 1).
- We discuss the future trend for resource-efficient CNN techniques.

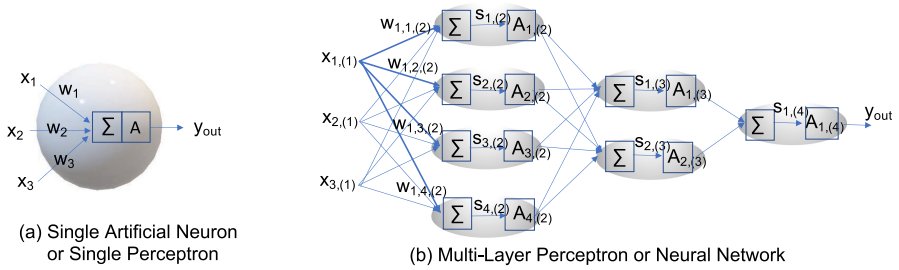


Fig. 2. Perceptron and neural network model.

We discuss our resource efficiency metrics for CNNs in Section 2, the model-level resource-efficient techniques in Section 3, the arithmetic-level techniques in Section 4, the implementation-level techniques in Section 5, and the influences between different-level techniques and the future research trends in Section 6, and we present our conclusion in Section 7. Our article excludes higher-level training procedure manipulation techniques such as one-pass ImageNet [84], bag of freebies [20], and data augmentation. We have predominantly collected papers that have been (1) highly cited or (2) published in world-leading machine learning or computing system conferences/journals (e.g., CORE: $A/A^*$ or JCR Q1).

## 2 BACKGROUND ON CNNS AND RESOURCE EFFICIENCY

This section describes our CNN overview and resource efficiency metric, as preparatory to the description of resource-efficient techniques via the three different levels.

### 2.1 Deep Learning Overview

*Deep learning* is defined as "learning multiple levels of representation" [17] and often utilizes CNNs to learn the multiple levels of representation. CNNs are trained using the training dataset, and their prediction accuracy is evaluated using the test dataset [5, 100]. In this section, we describe the perceptron model (i.e., artificial neuron) first and then CNNs later.

*2.1.1 Perceptron Model.* The McCulloch and Pitts's neuron (a.k.a. M-P neuron) [123], proposed in 1943, was a system mimicking the neuron in the nervous system, receiving multiple binary inputs and producing one binary output based on a threshold. Inspired by the work of McCulloch and Pitts [123], Rosenblatt [139] proposed the "perceptron" model consisting of multiple weights, a summation, and an activation function as shown in Figure 2(a).

Equation (1) describes a perceptron's firing activity $y_{out}$ using the inputs $x_i$ associated with their weights $w_i$, where the $i$ represents an index to indicate one of multiple inputs:

$$y_{out} = \begin{cases} 1, & \text{if } (\Sigma_{i=1}^{n_{in}} w_i \times x_i > threshold) \text{ or } (\Sigma_{i=1}^{n_{in}} w_i \times x_i + bias > 0) \\ 0, & \text{if } (\Sigma_{i=1}^{n_{in}} w_i \times x_i \leq threshold) \text{ or } (\Sigma_{i=1}^{n_{in}} w_i \times x_i + bias \leq 0), \end{cases} \tag{1}$$

where $n_{in}$ is the number of the inputs. The function that determines the firing activity is referred to as the *activation function*, and the *bias* is in proportion to the probability of the firing activation [126]. Since the single perceptron model is suitable only for linearly separable problems, a **Multi-Layer Perceptron (MLP)** model can be used for non-linearly separable problems as shown in Figure 2(b), where $w_{j,k,(i)}$ represents a weight linking the $j^{\text{th}}$ neuron in the $(i-1)^{\text{th}}$ layer to the $k^{\text{th}}$ neuron in the $i^{\text{th}}$ layer. The signal $s_{j,(i)}$ in Figure 2 follows Equation (2):

$$s_{j,(l)} = \Sigma_{i=1}^{n_{in}^{(l-1)}} (w_{i,j,(l)} \times x_{i,(l-1)}) = (\mathbf{W}_{(l)}^T \mathbf{x}_{(l-1)})_j, \tag{2}$$

and $x_{j,(l)} = \theta_P(s_{j,(l)})$, where $\theta_P(s)$ is a perceptron's activation function that follows Equation (1) (i.e., step function), and $\mathbf{W}_{(l)}$ consists of the matrix elements, $w_{i,j,(l)}$s, for the $i^{\text{th}}$ row and the $j^{\text{th}}$ column.

*2.1.2 Deep Neural Network.* Since it requires tremendous efforts for human to optimize MLPs manually, neural networks that adopt a soft threshold *activation function* $\theta_N$ (*sigmoid*, *ReLU*, etc.) were proposed to train the weights according to the training data [169, 172]. term *neural network* is sometimes interchangeably used with MLP [126]. For clarity, we name an algorithm as an MLP if it utilizes a step function for its activation functions and as a neural network if it utilizes a soft threshold function. In Figure 2(b), the output from the $i^{\text{th}}$ neuron at the $l^{\text{th}}$ layer in a neural network employing a soft threshold activation function, $\theta_N(\cdot)$, can be represented as Equation (3):

$$x_{i,(l)} = \theta_N(s_{i,(l)}). \tag{3}$$

A neural network allows the weights and the biases to be trained using backpropagation [5]. A neural network model is often referred to as a *feed-forward* model in that the weights always link the neurons in the current layer to the neurons in the very next layer. In a neural network, the middle layers located between the input and output layer are often referred to as *hidden layers* (e.g., two hidden layers in Figure 2(b)). A neural network with multiple hidden layers is referred to as a *DNN* [155].

*2.1.3 Training: Backpropagation.* In the forward pass, the neuron outputs are propagated in the forward direction based on the matrix-vector multiplications as shown in Equation (2). Likewise, the weights and the biases can be trained in the backward direction using matrix-vector multiplications. This method is called *backpropagation*. The backpropagation method consists of three steps, allowing a gradient descent algorithm to be implemented efficiently on computers. It finds the activation gradients, $\delta_{j,(l)}$s (i.e., the gradients with respect to all the signals, $s_{j,(l)}$s, in Equation (2)), in step 1, finds the weight gradients (i.e., the gradients with respect to all the weights) using the activation gradients in step 2, and finally updates the weights using the weight gradients in step 3. All $\delta_{j,(l-1)}$s are found in the backward direction using the matrix-vector multiplications by replacing $\mathbf{W}_{(l)}^T$ to $\mathbf{W}_{(l)}$ and $x_{j,(l-1)}$ to $\delta_{j,(l)}$ in Equation (3). After all activation gradients have been found, the weight gradients can be found. Finally, the weights are updated using the weight gradients. The backpropagation requires additional storage to store all the weights and activation values. Once a DNN is trained, the DNN is used for the inference task using the trained weights. Please refer to the work of Abu-Mostafa et al. [5] for further details on the backpropagation method. After a DNN is trained, the DNN's accuracy is evaluated using the validation dataset which is unseen from the training.

*2.1.4 Convolutional Neural Network.* CNN employs multiple convolutional layers, and each convolutional layer utilizes multiple filters to perform convolutions independently with respect to each filter as shown in Figure 3, where a *filter* at a convolutional layer consists of as many *kernels* as the number of the channels at the input layer (e.g., three kernels per filter in Figure 3). For example, each $3 \times 3$ filter has nine weight parameters and slides from the top-left to the bottom-right position, generating four output values with respect to each position (e.g., top-left, top-right, bottom-left, and bottom-right positions) in Figure 3. The outputs from the convolutions are often called *feature maps* and are fed into activation functions. Modern CNNs such as ResNet [72] often employ a batch normalization layer [90] between the convolutional layer and the ReLU layer to improve the accuracy.

The CNN is a resource-efficient DNN architecture in terms of the accuracy per parameter and per operation by leveraging the two properties: *local receptive field* and *shared weights* [103]. For

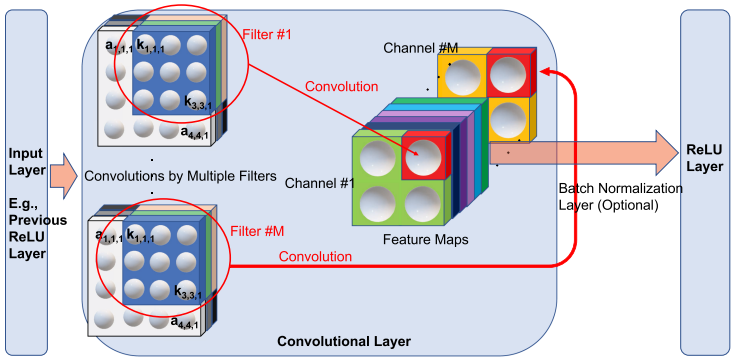


Fig. 3. Convolution operations in a CNN.

example, performing convolutions using multiple small kernels extracts multiple local receptive features from the input image during training, and each kernel contains some meaningful pattern from the input image after being trained [189]. Thus, a CNN utilizes much fewer weights than a fully connected DNN, since the kernel's height and weight are generally much smaller than the height and the width at the input layer, leading to the improved resource efficiency. Notice that a convolutional layer becomes a fully connected layer if the height and the width at the input layer are matched with each kernel's height and width. The number of total weights in a layer in a CNN is much less than used in a fully connected neural network, since the local receptive weights are shared over the entire feature on a layer.

Training a CNN also utilizes backpropagation using the transpose of kernel matrices in a filter to update the weights in the filter. The mini-batch gradient descent algorithm is widely used to train CNNs, which utilizes part of training data to update the weights per iteration. The number of data used per iteration is often referred to as the *batch size* $B$ (e.g., $B = 64$ or 128). Each *epoch* consumes the entire training data, consisting of $N/B$ iterations, where $N$ is the number of the entire training data. The mini-batch gradient descent method is a resource-efficient training algorithm in terms of the accuracy per operation, compared to the batch gradient descent method that utilizes entire training dataset per iteration (i.e., the batch gradient descent method updates the weights per epoch). For parallel backpropagation implementation with respect to $B$ data samples in one mini-batch, all the weights and all the activation values using $B$ training samples should be stored to update the weights per the mini-batch iteration, requiring $B\times$ additional storage, compared to the backpropagation using a stochastic gradient descent algorithm which updates the weights per training sample.

## 2.2 Resource Efficiency Metrics

Recently, researchers from DeepMind [84] proposed the metrics for resource-efficient deep learning benchmarks, including the top-1 accuracy, the required memory footprint for training, and the required number of floating operations for training, and evaluated the resource efficiency for deep learning applications by jointly considering the three metrics. The Roofline model [171] discussed attainable performance in terms of the operational intensity defined as the number of **Floating-Point (FP)** operations per DRAM access. Motivated by other works [84, 171], our resource efficiency metrics include the accuracy per parameter, per operation, per memory footprint, per core utilization, per memory access, and per Joule as shown in Figure 1.

*2.2.1 Accuracy per Parameter.* We consider the accuracy per parameter (i.e., weight) a resource efficiency metric. The accuracy per parameter is an abstract resource efficiency metric since

higher accuracy per parameter does not always imply higher physical resource efficiency after its implementation [1, 84].

*2.2.2 Accuracy per Operation.* We consider the accuracy per arithmetic operation a resource efficiency metric. This is also an abstract metric, since it can be evaluated prior to the implementation.

*2.2.3 Accuracy per Compute Resource.* Instruction-driven architecture such as CPU or GPU requires substantial memory accesses due to instruction fetch and decode operations, whereas data-driven architecture such as ASIC or FPGA can minimize the number of memory accesses, resulting in energy efficiency. We further categorize such compute resource into core utilization, memory footprint, and memory access, required to operate a CNN on given computing platforms. For example, the memory access can be interpreted as GPU DRAM access (or off-chip memory) for a GPU and as FPGA on-chip memory access (or off-chip memory) for a FPGA.

*Accuracy per Core Utilization.* The core utilization in this work represents the utilization percentage of the processing cores or PEs.

*Accuracy per Memory Footprint.* The accuracy per memory footprint is related to both physical and abstract resource efficiency as shown in Figure 1. The memory footprint is in proportion to the number of the parameters, but it can be varied according to a precision level applied for arithmetic. For example, if a half precision arithmetic is applied for a CNN, the memory footprint can be saved by 2× compared to a single-precision arithmetic CNN.

*Accuracy per Memory Access.* A computing kernel having a low operational intensity cannot approach a peak performance defined by hardware specification since the data supply rate from DRAM to CPU cannot catch up with the data consumption rate by arithmetic operations. Such kernels are called *memory bound kernels* in the work of Williams et al. [171]. Other type kernels are called *compute bound kernels*, which can approach a peak performance defined by hardware specification. Utilizing reduced precision arithmetic can improve the performance for both memory bound kernels by improving the data supply rate from DRAM to CPU and compute bound kernels by increasing word-level parallelism on SIMD architectures [105].

*2.2.4 Accuracy per Joule.* Dynamic power consumption is the main factor to determine energy consumption required for computationally intensive tasks (e.g., CNN training/inference tasks). The dynamic power consumption, $P_D$, follows:

$$P_D = \#_{TTR} \times C_{CP} \times V_{CP}^2 \times f_{CP}, \tag{4}$$

where $\#_{TTR}$ is the number of toggled transistors, $C_{CP}$ is an effective capacitance, $V_{CP}$ is an operational voltage, and $f_{CP}$ is an operational frequency for a given computing platform $CP$. Generally, the required minimum operational voltage is in proportion to the operational frequency. Therefore, adapting the frequency to the voltage scaling can save power cubically (a.k.a. Dynamic Voltage Frequency Scaling [147]). For example, minimizing the operations required to operate a CNN during runtime contributes to minimizing $\#_{TTR}$, resulting in power reduction and energy savings; we discuss the resource-efficient techniques leveraging this in Section 5.3.1.

*2.2.5 Interrelated Influence of Resource Efficiency Metrics.* A higher-level resource efficiency metric influences a lower-level metric as shown in Figure 1. For example, accuracy per parameter affects accuracy per memory footprint and/or memory access. Reduced parameters require less compute memory footprints and minimize the number of arithmetic operations, leading to reduced core utilization, memory accesses, and energy consumption. Energy consumption of CNNs on CPUs and GPUs mainly comes from memory access [7, 180]. Therefore, minimizing memory access cost is the main consideration for energy savings for CNNs [155]. For example, most

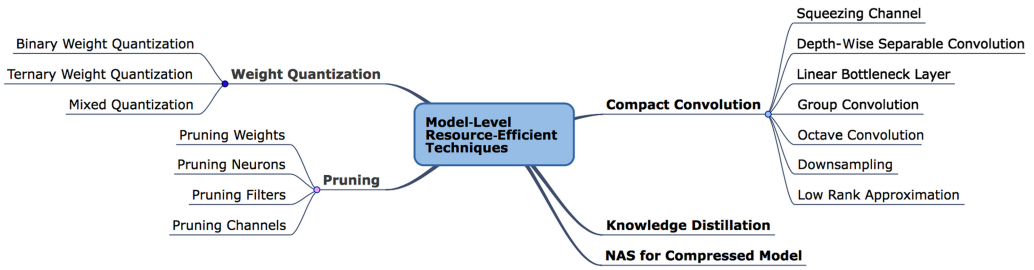


Fig. 4. Categorization of model-level resource-efficient techniques.

implementation-level resource-efficient techniques focus on minimizing memory access cost to save energy consumption.

## 3 MODEL-LEVEL RESOURCE-EFFICIENT TECHNIQUES

The model-level resource-efficient techniques, mostly developed from the machine learning community, aim at reducing the CNN model size to fit the models to resource-constrained systems such as mobile devices and IoT. We categorize the model-level resource-efficient techniques as shown in Figure 4.

### 3.1 Weight Quantization

The weight quantization techniques quantize the weights with a smaller number of bits, improving the accuracy per memory footprint. The training procedure should be amended according to the weight quantization schemes.

*3.1.1 Binary Weight Quantization.* The BinaryConnect training scheme [41] allowed a CNN to represent the weights using one bit. In step 1, the weights are encoded to $\{-1, 1\}$ using a stochastic clipping function. In step 2, the forward pass is performed using the encoded binary weights. In step 3, backpropagation seeks all activation gradients using full precision. In step 4, the weights are updated using full precision, and the training procedure goes back to step 1 for the training using the next mini-batch. This method required only one bit to represent the weights, thus improving the accuracy per memory footprint. In addition, the binary weight quantization also removed the need for multiplication arithmetic operations for MAC operations, improving the accuracy per operation. Moreover, if the activations are also quantized to the binary value, all MAC operations in the CNN can be implemented only with XNOR gates and a counter [42, 135].

*3.1.2 Ternary Weight Quantization.* Li et al. [108] proposed ternary weight networks that utilized ternary weights, improving accuracy, compared to the binary weight networks. All the weights on each layer were quantized into three values, requiring only two bits to represent the quantized weights. The overall training procedure was similar to that of Courbariaux et al. [41], but with ternary valued weights instead of the binary weights. The ternary weight network showed equivalent accuracy to various single-precision networks with MNIST, CIFAR-10, and ImageNet, whereas the binary weight quantization [41] showed minor accuracy loss. Zhu et al. [196] scaled the ternary weights independently for each layer with a layer-wise scaling approach, improving the accuracy further, compared to Li et al. [108].

*3.1.3 Mixed Quantization.* Hubara et al. [88] proposed the "Quantized Neural Network," which quantizes the activations and the weights to arbitrary lower-precision format. For example,

quantizing the weights to one bit and the activations to two bits improved the accuracy compared to the binarized CNN of Courbariaux et al. [42].

### 3.2 Pruning

Pruning unimportant neurons, filters, and channels can save computational resources for CNN applications without sacrificing accuracy, improving the accuracy per parameter and per operation. Coarse-grained pruning methods such as pruning filters or channels are not flexible to achieve a prescribed accuracy but can be implemented efficiently on hardware [114], implying higher physical resource efficiency than fine-grained pruning such as pruning weights. Notice that such pruning methods can degrade confidence scores without careful retraining, even though they did not affect top-1 accuracy [184].

*3.2.1 Pruning Weights.* In 1990, LeCun et al. [104] proposed a weight pruning method to generate sparse DNNs with fewer weights without losing accuracy. In 2015, the weight pruning approach was revisited [70], and the weights were pruned based on their magnitudes after training—the pruned CNNs were retrained to regain the lost accuracy. The pruning and retraining procedures could be performed iteratively to prune the weights further. This method reduced the number of weights of AlexNet by 9× without losing accuracy. In 2016, Guo et al. [63] noticed that pruning wrong weights could not be revived, and proposed to prune and splice the weights per mini-batch training to minimize the risk from pruning wrong weights from previous mini-batch training. For example, the pruned weights were also used in the weight update procedure during the back-propagation and were restored when they were reconsidered as the important weights. In 2017, Yang et al. [180] proposed an energy-aware weight pruning method in which the energy consumption of a CNN was directly measured to guide the pruning process. In 2019, Frankle and Carbin [53] demonstrated that some of pruned models outperformed the original model by re-training the pruned models with replacing the survived weights with the initial random weights used for training the original model.

*3.2.2 Pruning Neurons.* Instead of pruning individual weights, pruning a neuron can remove a group of the weights belonging to the neuron [85, 120, 152, 188]. In 2015, Srinivas and Babu [152] pruned the redundant neurons having similar weight values in a trained CNN model. For example, the weights in a baseline neuron were compared to the weights in other neurons at the same layer, and the neurons having similar weights to the baseline neuron were fused to the baseline neuron based on a Euclidean distance metric in the weight values between the two neurons. In 2016, Mariet and Sra [120] pruned the redundant neurons based on the "determinantal point process" metric. Hu et al. [85] measured the average percentage of zero activations per neuron and pruned the neurons having a high percentage of zero activations according to a given compression rate. Yu et al. [188] pruned unimportant neurons based on the effect of the pruning error propagation on the final response layer (e.g., the neurons were pruned backward from the final layer to the first layer). The methods of pruning neurons improved the resource efficiency, such as the accuracy per parameter and per operation.

*3.2.3 Pruning Filters.* Pruning insignificant filters after training can improve the accuracy per parameter and per operation. The feature maps associated with the pruned filters and the next kernels associated with the pruned feature maps should be also pruned. Pruning filters can maintain the dense structure of DNN unlike pruning weights, implying that it is highly probable to improve physical resource efficiency further, compared to pruning weights. Li et al. [109] pruned unimportant filters based on the summation of absolute weight values in the filter. The pruned CNNs were retrained with the survived filters to regain the lost accuracy. Yang et al. [181]

pruned filters based on a platform-aware magnitude-based metric depending on the resource-constrained devices. *ThiNet* [118] calculated the significance of the filters using the outputs of the next layer and pruned the insignificant filters based on this significance measurement.

*3.2.4 Pruning Channels.* Unlike pruning filters, pruning channels removes the filters at the current layer and the kernels at the next layer associated with the pruned channels. The network slimming approach [114] pruned insignificant channels, producing compact models while keeping equivalent accuracy, compared to the models prior to pruning. For example, insignificant channels were identified based on scaling factors generated from the batch normalization of Ioffe and Szegedy [90], and the channels associated with lower scaling factors were pruned. After the initial training, the channels associated with relatively low scaling factors were first pruned, and retraining was then performed to refine the network. He et al. [75] identified unimportant channels using LASSO regression from a pre-trained CNN model and pruned them. The channel pruning brought 5× speedup on VGG16 with minor accuracy loss. Lin et al. [112] pruned unimportant channels during runtime based on a decision maker trained by reinforcement learning. Gao et al. [56] proposed another dynamic channel pruning method that dynamically skipped the convolution operations associated with unimportant channels.

### 3.3 Compact Convolution

To improve resource efficiency such as the accuracy per operation and per parameter from computationally intensive convolution operations, many compact convolution methods have been proposed.

*3.3.1 Squeezing Channel.* In 2016, Iandola et al. [89] proposed *SqueezeNet*, in which each network block utilized the number of $1 \times 1$ filters less than the number of the input channels to reduce the network width in the squeezing stage and then utilized multiple $1 \times 1$ and $3 \times 3$ kernels in the expansion stage. The computational complexity was significantly reduced by squeezing the width while compensating the accuracy in the expansion stage. SqueezeNet reduced the number of parameters by 50×, compared to AlexNet on ImageNet without losing accuracy, improving accuracy per parameter. Gholami et al. [59] proposed SqueezeNext that utilized separable convolutions in the expansion stage; a $k \times k$ filter was divided into a $k \times 1$ and a $1 \times k$ filter. Such separable convolutions reduced the number of parameters further compared to SqueezeNet while maintaining AlexNet's accuracy on ImageNet, improving accuracy per parameter further, compared to SqueezeNet.

*3.3.2 Depth-Wise Separable Convolution. Xception* [38] utilized depth-wise separable convolutions, which replace 3D convolutions with 2D separable convolutions followed by 1D convolutions (i.e., point-wise convolutions) as shown in Figure 5, to reduce computational complexity. The 2D separable convolutions are performed separately with respect to different channels. Howard et al. [83] proposed *MobileNet v1* that utilizes depth-wise separable convolutions with two hyperparameters, "width multiplier and resolution multiplier," to fit CNNs to resource-constrained devices by fully leveraging the accuracy and resource tradeoff in the CNNs. MobileNet v1 showed equivalent accuracy to GoogleNet and VGG16 on the ImageNet dataset with less computational complexity, improving the accuracy per parameter and per operation.

*3.3.3 Linear Bottleneck Layer.* In general, the manifold of interest (i.e., the subspace formed by the set of activations at each layer) could be embedded in low-dimensional subspaces in CNNs. Inspired by this, Sandler et al. [142] proposed MobileNet v2 consisting of a series of bottleneck layer blocks. Each bottleneck layer block as shown in Figure 6 received lower-dimensional input, expanded the input to high-dimensional intermediate feature maps, and projected the

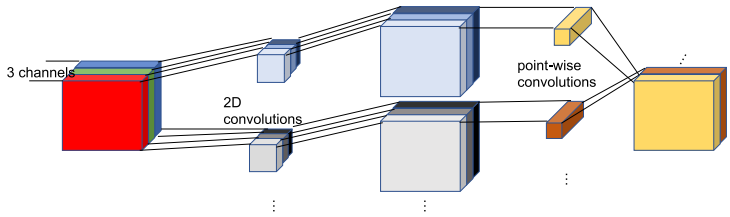


Fig. 5. Depth-wise convolution used in the work of Howard et al. [83].

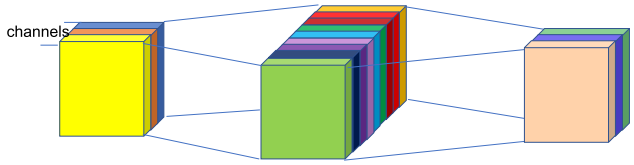


Fig. 6. Bottleneck layer block used in the work of Sandler et al. [142].

high-dimensional intermediate features onto low-dimensional features. Keeping linearity for the output feature maps was crucial to avoid destroying information from non-linear activations, so linear activation functions were used at the end of each bottleneck block.

*3.3.4 Group Convolution.* In a group convolution method, the input channels are divided into several groups, and the channels in each group are separately participated in convolution with other groups. For example, the input channels with three groups required three separate convolutions. Since group convolution does not communicate with the channels in other groups, communication between different groups is performed after the separate convolutions. Group convolution methods [86, 87, 119, 193] reduced the number of MAC operations, improving the accuracy per operation, compared to CNNs using regular convolution. In 2012, AlexNet utilized group convolution to train the CNNs effectively using two NVIDIA GTX580 GPUs [100]. Surprisingly, AlexNet using group convolution showed superior accuracy to AlexNet using regular convolution, improving the accuracy per operation. In 2017, ResNext [177] utilized group convolution based on ResNet [72] using a cardinality parameter (i.e., the number of groups). In 2018, Zhang et al. [193] noticed that the point-wise convolutions were computationally intensive in practice in the depth-wise convolutions and proposed ShuffleNet that applied group convolution to every point-wise convolution to reduce compute complexity further, compared to MobileNet v1. ShuffleNet shuffled the output channels from the grouped point-wise convolution to communicate with different grouped convolutions, demonstrating superior accuracy to MobileNet v1 on ImageNet and COCO datasets, given the same arithmetic operation cost budget. Ma et al. [119] proposed ShuffleNet v2, which improved physical resource efficiency further compared to ShuffleNet [193] by employing equal channel width for input and output channels where applicable and minimizing the number of operations required for $1 \times 1$ convolutions. Rather than choosing each group randomly and shuffling them, Huang et al. [86] proposed to learn each group for a group convolution during training. The "learned group convolution" was applied in *DenseNet* [87], and DenseNet improved the accuracy per parameter and per operation, compared to ShuffleNet, given a prescribed accuracy.

*3.3.5 Octave Convolution.* Chen et al. [32] decomposed feature maps into a higher and a lower frequency part to save the feature maps' memory footprint and reduce the computational cost.

The decomposed feature maps were used by specific convolution called *octave convolution*, which performs a convolution between the higher and lower frequency part. The application of the octave convolution to ResNet-152 architecture achieved higher accuracy using ImageNet dataset than the regular convolution, improving the accuracy per operation and per memory footprint.

*3.3.6 Downsampling.* Qin et al. [132] applied a downsampling approach (e.g., a larger stride size for a convolution) to MobileNet v1, improving the top-1 accuracy by 5.5% over MobileNet v1 on the ILSVRC 2012 dataset, given a 12M arithmetic operations budget.

*3.3.7 Low Rank Approximation.* Denton et al. [47] proposed a low rank approximation that compresses the kernel tensors in the convolutional layers and the weight matrices in the fully connected layers by using singular value decomposition. Another low rank approximation [98] used Tucker decomposition to compress the feature maps, resulting in significant reductions in the model size, the inference latency, and the energy consumption. Such low rank approximation methods improve the accuracy per parameter, per operation, and per memory footprint.

### 3.4 Knowledge Distillation

The knowledge from a large-scale high-performing model (teacher network) could be transferred to a compact neural network (student network) to improve resource efficiency such as accuracy per parameter and per operation for inference tasks [23, 29, 138]. Buciluǎ et al. [23] utilized data with the labels generated from the teacher model (i.e., a large-scale ensemble model) to train a compact neural network. The compact model was trained with the pseudo training data generated from the teacher model, demonstrating equivalent accuracy to the teacher model. Ba and Caruana [14] noticed that the softmax outputs often resulted in the student network ignoring the information of the other categorizations than the one with the highest probability, and utilized the values prior to the softmax layer, from the teacher network, for the training labels to allow the student network to learn the teacher network more efficiently. Hinton et al. [78] added a "temperature" term for the labels to enrich the information from the teacher network and train the student network more efficiently, compared to Ba and Caruana [14]. Romeo et al. [138] utilized both labels and intermediate representations from a wider teacher network to compress it to a thinner and deeper student network. The "hint layer" was chosen from the teacher network, and the "guided layer" was chosen from the student network. The student network was then trained so that the intermediate representation deviation between the outputs from the hint layers and guided layers could be minimized. A thinner student network employed 10.4× less weight parameters, compared to a wider teacher network, while improving accuracy. This technique is also known as "hint learning." Hint learning was applied to both the region proposal and classification components for object detection applications [29].

### 3.5 Neural Architecture Search for Compressed Models

Zoph and Le [198] proposed the **Neural Architectural Search (NAS)** technique to seek optimal DNN models in the space of hyperparameters of network width, depth, and resolution. In case that compute resource budget was limited (e.g., mobile devices), many NAS variants exploited the tradeoff between accuracy and latency to maximize resource efficiency given a compute resource budget [74, 157–159, 174]. He et al. [74] proposed a NAS employing reinforcement learning, *AutoML*, that sampled the least sufficient candidate design space to compress the DNN models. *MnasNet* [157] utilized reinforcement learning with a balanced reward function between the accuracy and the latency to seek a compact neural network model. Wu et al. [174] proposed a gradient-based NAS that produced a DNN model with 2.4× model size reduction compared to a MobileNet v2 without losing accuracy on the ImageNet dataset. Scheidegger

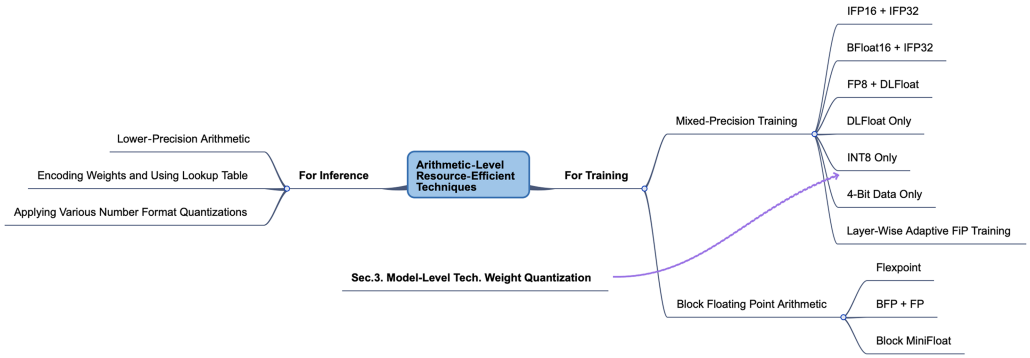


Fig. 7. Categorization of arithmetic-level resource-efficient techniques.

et al. [143] proposed a narrow-space NAS to generate low-resource DNNs satisfying a strict memory budget and inference time requirement for IoT applications. Anderson et al. [13] noticed that conventional NAS might improve abstract resource efficiency rather than physical resource efficiency, and utilized the hardware information including the inference latency for a NAS to ensure that the candidate models could improve the physical resource efficiency in practice. *EfficientNet* [158] utilized a NAS with compound scaling of depth, width, and resolution to seek optimal DNN models given fixed compute resource budgets. Another NAS utilizing compound scaling, *EfficientDet*, was proposed for object detection applications [159]. EfficientDet improved accuracy using the COCO dataset with 4 to 9× model size reduction, compared to state-of-the-art object detectors, improving the accuracy per parameters. Recently, Cai et al. [25] proposed a feed-forward NAS approach that produced a customized DNN, given compute resource and latency constraint.

## 4 ARITHMETIC-LEVEL RESOURCE-EFFICIENT TECHNIQUES

Utilizing lower-precision arithmetic reduces the memory footprint and the time spent transferring data across buses and interconnections [52, 125, 182, 197]. Employing least sufficient arithmetic precision for CNN applications can improve the accuracy per memory footprint and the accuracy per memory access. We categorize the arithmetic-level resource-efficient techniques into the two categories as shown in Figure 7, *Arithmetic-Level Techniques for Inference* and *Arithmetic-Level Techniques for Training*. We discuss different number formats first and the deployment of such number formats on CNNs later.

### 4.1 Number Formats for CNNs

This section describes various number formats for CNN applications, as preparatory to explaining the arithmetic-level resource-efficient techniques. **Fixed-Point (FiP)** number format utilizes a binary FiP between the fraction and integer parts. For example, an eight-bit FiP format, "01.100000," represents 1.5 (i.e., $\ldots 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} \ldots$) for the decimal representation, and the point between the integer part and fraction part is fixed for arithmetic operations. Therefore, it could be implemented with simple circuits, but the available data range is quite limited [130].

We exemplify the IEEE 754 general-purpose FP standard [4] to explain FP format and its arithmetic, since this standard is used for most commercially available CPUs and GPUs. The IEEE 754 Floating-Point (IFP) data format [4] consists of sign, exponent, and significand as shown in Equation (5). For example, an FP number has a $(p + 1)$-bit significand (including the hidden one), an $e$-bit exponent, and a one sign bit. The machine epsilon $\epsilon_{mach}$ is defined as $2^{-(p+1)}$. The value

represented by FP is as follows:

$$y_{out} = \begin{cases} \text{normal mode:} & (-1)^{sign} \times (1 \times 2^0 + d_1 \times 2^{-1} + \cdots + d_p \times 2^{-p}) \times 2^{exponent-bias} \\ \text{subnormal mode:} & (-1)^{sign} \times (d_1 \times 2^{-1} + \cdots + d_p \times 2^{-p}) \times 2^{1-bias}, \end{cases} \tag{5}$$

where $d_1, \ldots, d_p$ represent binary digits, the '1' associated with the coefficient $2^0$ is referred to as the hidden '1', the *exponent* is stored in offset notation, and the *bias* is a positive constant. If the absolute value of exponent is zero, the FP value is represented by the subnormal mode. IEEE 754 standard requires exact rounding for addition, subtraction, multiplication, and division; the FP arithmetic result should be identical to the one obtained from the final rounding after exact calculation. For example, based on the IEEE 754 rounding to nearest mode standard, FP arithmetic should follow Equation (6):

$$fl(x_1 \odot x_2) = (x_1 \odot x_2)(1 + \epsilon_r), \tag{6}$$

where $|\epsilon_r| \leq \epsilon_{mach}$, $\odot$ is one of the four arithmetic operations, and $fl(\cdot)$ represents the result from the FP arithmetic. Notice that *quantization* quantizes data to lower precision, whereas *arithmetic* is a rule applied to arithmetic operations between the two operands. For example, the quantization affects the values for the two operands, $x_1$ and $x_2$, in Equation (6), whereas arithmetic affects the rounding error, $\epsilon_r$.

*4.1.1 Half, Single, and Double Precision.* The IEEE FP 32- (IFP32 or single-precision) and 64-bit (IFP64 or double-precision) versions are available on most off-the-shelf conventional processors. Additionally, the IEEE 754 standard includes a 16-bit FP format (IFP16 or half precision) [4]. $p = 52$, $e = 11$, and $bias = 1023$ for IFP64, $p = 23$, $e = 8$, and $bias = 127$ for IFP32, and $p = 10$, $e = 5$, and $bias = 15$ for IFP16. IFP16 is currently supported in hardware on some modern GPUs to accelerate DNN applications [40, 79].

*4.1.2 Brain FP Using 16 Bits (BFloat16).* In 2018, a 16-bit Brain FP format [24, 186] was proposed that was tailored to CNN applications. BFloat16 consists of an eight-bit exponent and a seven-bit significand, supporting a wider dynamic data range than IFP16. BFloat16 is currently supported in hardware in Intel Cooper Lake Xeon processors, NVIDIA A100 GPUs, and Google TPUs.

*4.1.3 DLFloat.* In the race of designing specific FP formats for CNNs, some authors [6, 167] proposed another 16-bit precision format, DLFloat, consisting of a 6-bit exponent and a 9-bit significand to provide better balance between dynamic data range and precision than IFP16 and BFloat16 formats for some CNN applications.

*4.1.4 TensorFloat32 (TF32).* NVIDIA proposed a 19-bit data format, TF32, consisting of a 1-bit sign, an 8-bit exponent, and a 10-bit significand to accelerate CNN applications on A100 GPUs with the same dynamic range support as IFP32 [51]. TF32 tensor cores in an A100 truncate IFP32 operands to 19-bit TF32 format but accumulate them using IFP32 arithmetic for MAC operations.

### 4.2 Arithmetic-Level Techniques for Inference

This section discusses various resource-efficient arithmetic-level techniques based on pre-trained CNNs for the inference tasks.

*4.2.1 Lower-Precision Arithmetic.* Lower-precision FiP arithmetic has been widely used to deploy CNNs on edge devices [179]. The effects of deploying various lower-precision arithmetic to the CNN inference tasks were explored in terms of accuracy and latency [168, 175]. The *BitFusion* method accelerated CNN inference tasks by employing variable bit-width FiP formats dynamically depending on the different layers [148]. Similarly, Tambe et al. [156] proposed *AdaptiveFloat* that

adjusted dynamic ranges of FP numbers depending on the different layers, resulting in higher energy efficiency than FiP-based methods, given the same accuracy requirement.

*4.2.2 Encoding Weights and Using a Lookup Table.* Bordawekar et al. [21] leveraged the fact that the exponent values of most of the weights were located within a narrow range and encoded the frequent exponent values of the weights with fewer bits using a Huffman coding scheme, improving the accuracy per memory footprint for natural language processing applications. A lookup table, located between the memory and FP arithmetic units, is used to convert the encoded exponent values into FP exponent values.

*4.2.3 Applying Various Number Format Quantizations to CNNs.* The **Residue Number System (RNS)** is a parallel and carry-limited number system that transforms a big natural number to several smaller residues. Therefore, RNS was often used to perform parallel and independent calculations on residues without carry-propagation, and it exploited such parallelism to accelerate CNN computation [28]. In an RNS-based CNN, the weights of a pre-trained model were transformed to RNS presentation. Recently, RNS was used to replace costly multiplication operations with simple logical operations such as multiplexing and shifting, accelerating CNN applications [115, 140, 141]. The Logarithmic Number System applies the logarithm to the absolute values of the real numbers [130]. The main advantage of the Logarithmic Number System is in the capability of transforming multiplications into additions and divisions into subtractions. In 2018, Vogel et al. [165] utilized a five-bit logarithmic format using arbitrary log bases to improve resource efficiency such as accuracy per memory footprint and per operations by replacing costly multiplication arithmetic operations to simple bit-shift operations [165]. The Posit number format [66] employs multiple separate exponent fields to represent dynamic range effectively. Recently, CNNs utilizing Posit showed higher accuracy than various FP8 formats using Mushroom and Iris datasets [26, 27].

### 4.3 Arithmetic-Level Techniques for Training

This section discusses arithmetic-level resource-efficient techniques used for CNN training tasks. Training CNNs generally requires higher-precision arithmetic due to extremely small weight gradient values [41, 196]. Adjusting arithmetic precision according to different training procedures such as forward propagation, activation gradient updates, and weight updates can accelerate CNN training [64]. Training quantized CNNs often required stochastic rounding schemes [65, 176, 182, 195].

*4.3.1 Mixed-Precision Training.* A conventional mixed-precision training applied lower-precision arithmetic to the multiplications in MACs, including both forward and backward paths, and higher-precision arithmetic to the accumulations in the MACs using lower-precision quantized operands [95, 125]. The higher-precision outcomes from MACs were quantized to a lower-precision format to be used for consequent operations. In the following (X + Y) formats, X represents the data format used for MAC operations and Y represents the arithmetic applied for the accumulations in MAC operations (refer to the work of Gupta and Ranga [64] for details on lower- and higher-precision arithmetic usage).

*IFP16 + IFP32.* In 2018, Micikevicius et al. [125] noticed that the weights were updated using very small weight gradient values, and applied a lower-precision arithmetic IFP16 to the multiplications and a higher-precision IFP32 to the accumulations for the weight updates. For example, in the mixed-precision training approach in their work [125], IFP16 was used to store weights, activations, activation gradients, and weight gradients, whereas IFP32 was used to keep the weight copies for their updates. Along with accumulating IFP16 operands using IFP32 arithmetic, the use of loss scaling allowed the mixed-precision training to achieve equivalent accuracy to the IFP32 training while reducing the memory footprint.

*BFloat16 + IFP32.* In 2018, mixed-precision CNN training using (BFloat16 + IFP32) was explored by Ying et al. [186]. In 2019, Kalamkar et al. [95] studied BFloat16's feasibility for mixed-precision training for various CNNs including AlexNet, ResNet, and GAN, among others, and concluded that the (BFloat16 + IFP32) scheme outperformed the (IFP16 + IFP32) scheme since BFloat16 could represent the same dynamic range of data as IFP32 while using fewer bits.

*FP8 + DLFloat.* In 2018, Wang et al. [167] proposed a mixed-precision training method that applies the 5eFP8 format (one sign bit, five-bit exponent, and two bits for the significand) to the multiplications and DLFloat to the accumulations in MAC operations. The mixed-precision method improved resource efficiency such as accuracy per memory footprint and accuracy per memory access compared to various (FP16 + IFP32) schemes with respect to different FP16 formats. Compared to the previous (FP16 + IFP32) methods, the chunk-accumulation and stochastic rounding schemes were additionally used to minimize the accuracy loss in the work of Wang et al. [167], The chunk-based accumulation utilized 64 data per chunk instead of one long sequential accumulation to reduce rounding errors. Utilizing a stochastic rounding scheme with limited-precision format for deep learning was proposed earlier by Gupta et al. [65]. Sun et al. [153] noted that (5eFP8 + DLFloat) training degraded accuracy for CNNs utilizing depth-wise convolutions such as MobileNets. To overcome this issue, Sun et al. [153] proposed to employ two different eight-bit FP formats each for forward and backward propagation to minimize the accuracy degradation for compressed CNNs. The mixed-precision training utilized 5eFP8 for backpropagation and another eight-bit FP format with (sign, exponent, significand) = (1, 4, 3), 4eFP8, for forward propagation.

*DLFloat Only.* In 2019, Agrawal et al. [6] employed DLFloat for the entire training procedure, removing the necessity of data conversions between the multiplications and the accumulations and found that DLFloat could provide better balance between dynamic range and precision than IFP16 and BFloat16 for LSTM networks [80] using the Penn Treebank dataset. The DLFloat arithmetic units removed subnormal mode and supported the round-to-nearest up mode to minimize computational complexity. In the work of Agrawal et al. [6], the DLFloat arithmetic showed equivalent performance to IFP32 on ResNet-32 using CIFAR10 and ResNet-50 using ImageNet while using a half of IFP32 bit width.

*INT8 Only.* Yang et al. [182] noticed that previously proposed mixed-precision training schemes did not quantize the data in the batch normalization layer, requiring high FP arithmetic in some parts of the data paths. To overcome this issue, a unified INT8-based quantization framework was used to quantize all data paths in a CNN including weights, activation, gradient, batch normalization, and weight update, among others, into INT8-based data. However, this training method degraded the accuracy to some extent. In 2020, Zhu et al. [197] improved the accuracy, compared to the work of Yang et al. [182], while keeping a unified INT8-based quantization framework. In the work of Zhu et al. [197], the deviation of the activation gradient direction between before and after quantization was minimized by measuring the distance during runtime based on the inner product between the two normalized gradient vectors generated before and after quantization.

*Four-Bit Data Only (INT4 + FP4).* In 2020, Sun et al. [154] proposed a new training technique using only four bits to represent weights, activations, and their gradient values, requiring mixed-precision INT4-FP4 MAC operations. The weights and activations are stored using INT4 format since their data range is not wide, whereas their gradient values are stored using four-bit FP format (FP4) since their data range is wide. The proposed FP4 format uses one bit for the sign bit and three bits for the exponent (i.e., no mantissa bit) with radix-4 to represent wide-ranged gradient values. Therefore, this training method can improve the accuracy per memory footprint and memory access. A layer-wise gradient scaling approach with a customized rounding scheme was employed to minimize the impact of large quantization errors caused by the four-bit

representation for gradient values on the accuracy. This training method brought minor accuracy loss using CIFAR10 and ImageNet datasets with a ResNet18, compared to IFP32.

*Layer-Wise Adaptive FiP Training.* In 2020, Zhang et al. [192] proposed a layer-wise adaptive quantization scheme. For example, activation gradient distributions at fully connected layers followed a narrower distribution, requiring more bit width for the quantizations. AlexNet was quantized using INT8 for all the weights and activations and both INT8 (22%) and INT16 (78%) for the activation gradients. The quantized AlexNet achieved equivalent accuracy to the one using IFP32 for entire training on the ImageNet dataset.

*4.3.2 Block Floating-Point Training.* **Block Floating-Point (BFP)** format utilizes a shared exponent for a series of numbers in a data block to reduce data size [170]. Applying BFP to CNNs can improve the resource efficiency in terms of accuracy per memory footprint and per memory access. In addition, BFP utilizes less transistors for simpler adders and multipliers than FP adders and multipliers, resulting in improving accuracy per Joule. Various versions of CNN training methods using BFP were proposed to improve resource efficiency.

*Flexpoint.* A DNN-optimized BFP format, *Flexpoint* [99], was proposed by Intel, and it was used with the Nervana neural processors. The BFP format used 5 bits for a shared exponent and 16 bits for the significand for the data in a data block. Flexpoint utilized the format of *(Flex N) + M*, where *Flex N* represents variable number of bits for the shared exponent according to the different epochs, and *M* represents the number of bits for the separated significand. For example, the number of exponent bits is adapted based on the dynamic range of the weight values depending on the number of iterations; the dynamic range of the weight values at the current iteration was predicted at the previous iteration. The (*Flex N + 16*) format produced equivalent accuracy to IFP32 in AlexNet using the ImageNet dataset and a ResNet using the CIFAR-10 dataset, resulting in significant resource efficiency improvement in terms of accuracy per memory footprint and accuracy per memory access.

*BFP + FP training.* Drumond et al. [48] proposed a hybrid use of BFP and FP for CNN training that uses BFP only for MAC operations and FP for the other operations. Such hybrid training method brought 8.5× potential throughput improvement with minor accuracy loss in WideResNet28-10 using the CIFAR-100 dataset on a Stratix V FPGA.

*Block MiniFloat.* It was noticed that ordinary BFP formats were limited in minimizing original data loss with fewer bits and improving arithmetic density per memory access for CNN applications [52]. To address the two issues, Fox et al. [52] proposed **Block Minifloat (BM)** along with customized hardware circuit design. The BM<e,m> format follows:

$$y_{out} = \begin{cases} \text{normal mode:} & (-1)^{sign} \times (1 \times 2^0 + d_1 \times 2^{-1} + \cdots + d_m \times 2^{-m}) \times 2^{exponent - bias - BIAS_{SE}} \\ \text{subnormal mode:} & (-1)^{sign} \times (d_1 \times 2^{-1} + \cdots + d_m \times 2^{-m}) \times 2^{1 - bias - BIAS_{SE}}, \end{cases} \tag{7}$$

where $bias = 2^{e-1} - 1$ and $BIAS_{SE}$ is a shared exponent value. $BIAS_{SE}$ is scaled according to the maximum value of the data for dot-product operations. For example, BM<2,3> represents a six-bit data format having one sign bit, one-bit exponent, and three-bit significand. Such BM variant formats were applied for training. Utilizing these six-bit BM formats produced equivalent accuracy to IFP32 formats but with fewer bits using CIFAR 10 and 100 datasets with ResNet-18, resulting in reduced memory traffic and low energy consumption. Therefore, BM improved the resource efficiency in term of accuracy per memory access.

## 5 IMPLEMENTATION-LEVEL RESOURCE-EFFICIENT TECHNIQUES

Figure 8 classifies the implementation-level resource-efficient techniques. Most implementation-level techniques have focused on improving energy efficiency and computational speed for MAC

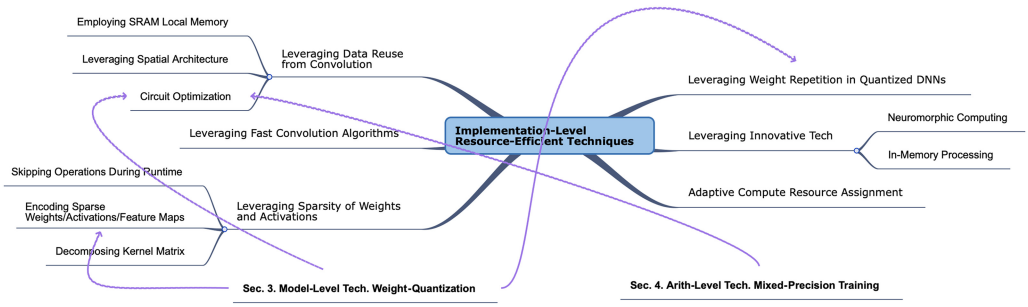


Fig. 8. Categorization of implementation-level resource-efficient techniques.

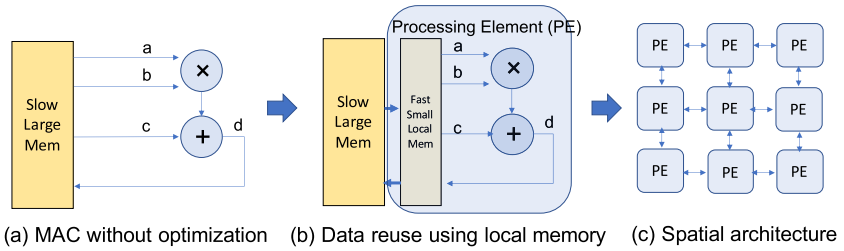


Fig. 9. MAC dataflow.

operations, since MACs generally occupy more than 90% of computational workload for both training and inference tasks in CNNs [155]. The implementation-level resource-efficient techniques exploited the characteristics of MACs in the CNN including data reuse, sparsity of weights and activations, and weight repetition from quantized CNNs.

## 5.1 Leveraging Data Reuse from Convolution

The weights and the activations are heavily reused in convolution operations. For example, the weights of a filter are reused $((H - k_H + 1) \times (W - k_W + 1))/stride$ times, where $H = W = 4$ (height and width at input channel) and $k_H = k_W = 3$ (height and width for a kernel) in Figure 3. Generally, $H$ and $W$ are three orders of magnitude (128, 256, etc.), $k_H$ and $k_W$ are one order of magnitude (3, 5, etc.), and $stride$ is either 1 or 2. For example, if $H = W = 128$, $k_H = k_W = 3$, and $stride = 1$, each filter is reused 16,129 times for convolutional operations. Each input element at a convolutional layer is also reused approximately $M \times k_H \times k_W$ times, where $M$ is the number of the total kernels used in the layer. Figure 9 describes the data access patterns for MAC operations used for convolutional layers. In each MAC in Figure 9(a), the data, $a$, $b$, and $c$, are read from the memory for multiply and add computation, and the result $d$ is written back to the memory, where $c$ contains a partial sum for the MAC. To save energy consumption, highly reused data for MACs can be stored in small local memory as shown in Figure 9(b). For example, the power consumption required to access data depends on where the data are located—accessing data from off-chip memory, DRAM, generally requires two orders of magnitude more than from on-chip memory [31]. Many research works have presented how to leverage such data reuse properties to improve resource efficiency.

*5.1.1 Employing SRAM Local Memory Near PEs.* The use of SRAM buffers reduces the energy consumed by CNNs by up to two orders of magnitude compared to DRAM. Similar to Figure 9(b), the *DianNao* architecture [30] employed one **Neural Functional Unit (NFU)** integrated with

three separated local buffers, each for holding 16 input neurons, $16 \times 16$ weights, and 16 output neurons, to optimize circuitry for MAC operations. The weights and the activations stored to the local memories were reused efficiently by additionally using internal registers to store the partial sums and the circular buffer. The NFU is a three-stage pipelined architecture consisting of the multiplication, the adder-tree, and the activation stage. In the multiplication stage, 256 multipliers support the multiplications based on the weight connections between 16 input and 16 output neurons. In the adder-tree stage, 16 feature maps are generated from the multiplications based on adder-tree structure. In the activation stage, the 16 feature maps are approximated for the 16 activations by using piece-wise linear function approximation. DianNao with a 65-nm ASIC layout brought up to 118× speedup and reduced the energy consumption by 21× compared to a 128-bit 2-GHz SIMD processor over the benchmarks used in the work of Chen et al. [30]. One of the following studies adapted DianNao to deploy it on a supercomputer and named it *DaDianNao* [33]. Since the number of weights is generally larger than the number of input activations for convolution operations, DaDianNao stored a big chunk of weights and shared them to multiple NFUs by using a *central embedded DRAM* to reduce the data movement cost in delivering the weights associated to each NFU.

*5.1.2 Leveraging Spatial Architecture.* Designing PEs and their local memory according to data reuse properties of MAC operations improved energy efficiency on FPGAs and ASIC [31, 34]. For example, Google TPU employs a systolic array architecture to send the data directly to an adjacent PE as shown in Figure 9(c) [3]. Chen et al. [31] noticed that the computational throughput and energy consumption of CNNs mainly depended on data movement rather than computation and proposed a "row-stationary" spatial architecture (a variant of Figure 9(c)), *Eyeriss*, to support parallel processing with minimal data movement energy cost by fully exploiting the data reuse property. For example, the three PEs in the first column in Figure 9(c) can be assigned to compute the first row of the convolution output using a $3 \times 3$ filter—the three elements on each row of the kernel are stored to the local memory on each PE (i.e., "row-stationary" structure in the work of Sze et al. [155]), and all the elements in the kernel are reused during convolution, generating the first row of the output. In this case, the partial summation values are stored back to the local memory on each PE.

*5.1.3 Circuit Optimization.* Exploring binary weights [41] with binary inputs offered the opportunity to explore XNOR gates for the efficient implementation of a CNN [134], improving the accuracy per memory foot print and per Joule. In 2021, Zhao et al. [194] proposed hardware-friendly statistic-based quantization units and near data processing engines to accelerate mixed-precision training schemes by minimizing the number of accesses to higher-precision data.

### 5.2 Leveraging Fast Convolution Algorithms

For commercially available CPUs or GPUs, transforming convolution operations into matrix multiplications can leverage data reuse properties to accelerate the convolution operations by utilizing highly optimized BLAS libraries [163]; unrolling the weights of a 3D filter to a 1D vector transforms 3D convolution operations into 2D matrix multiplications (a.k.a. im2col convolution). Mathieu et al. [121] implemented circular convolutions with a **Fast Fourier Transform (FFT)** algorithm for CNNs. Using FFT, a circular convolution can be implemented with element-wise multiplications between the spectrum signals of the image and the kernel in the frequency domain. The number of multiplications required for a single FFT-based convolution between an $N \times N$ resolution image and a $k \times k$ kernel requires $N^2(6log_2N + 4)$ multiplications ($=6N^2log_2N$ (two FFTs and one inverse FFT) + $4N^2$ (element-wise multiplications between two complex numbers)), whereas conventional convolution takes $N^2k^2$ multiplications. Since the number of multiplications required

for an FFT-based convolution does not depend on the kernel size, FFT-based convolutions are useful when kernel sizes are relatively large (e.g., $k^2 > (6log_2N + 4)$). Vasilache et al. [164] demonstrated that CNNs implemented with NVIDIA cuFFT showed 1.4 to 14.5× speedups over cuDNN for various kernel sizes. Highlander and Rodriguez [77] implemented FFT with an Overlap and Add algorithm to reduce the FFT cost from $O(N^2log_2N)$ to $O(N^2log_2k)$ to apply FFT-based convolutions for smaller-sized kernels. Zhang and Prasanna [191] customized the FFT-based convolution architecture on FPGAs. Notice that such FFT-based convolutions achieve higher accuracy per operation with the cost of additional memory footprint. Lavin and Gray [102] leveraged a Winograd minimal filtering algorithm [173] to reduce the number of multiplication operations required for convolutions to improve the accuracy per operation. Lu et al. [117] proposed an efficient architecture for Winograd-based convolutions on FPGAs. Cong and Xiao [39] reformulated convolution operations between an input tensor and multiple filters into a single matrix multiplication form named *convolution matrix multiplication* to reduce the number of multiplications with the Strassen algorithm.

### 5.3 Leveraging Sparsity of Weights and Activations

In the forward pass, negative feature map values are converted to zeros after ReLU activation functions, making the activation data structure sparse. In addition, the trained weight values follow a sharp Gaussian distribution centered at zero, locating most of the weights near to zero. Quantizing such weights makes the weight data structure sparse, so the sparse weights can be fully exploited on the quantized networks such as binarized CNNs [41, 42] and ternary weight CNNs [108, 196].

*5.3.1 Skipping Operations During Runtime.* In 2016, several methods to conditionally skip MAC operations were proposed simultaneously [10, 31, 113]. Eyeriss [31] employed clock gating to block the convolution operations during runtime when either the weight or the activation was detected as zero to save computational power. *Cnvlutin* [10] skipped MAC operations associated with zero activations by employing separated "neuron lanes" according to different channels. Similarly, Liu et al. [113] proposed *Cambricon-X* that fetches the activations associated with any non-zero weights for convolutions by using "step indexing" to skip the MAC operations associated with the zero weights. Cambricon-X brought 6× resource efficiency improvement in terms of accuracy per Joule compared to the original *DianNao* architecture. In 2017, Kim et al. [96] proposed *ZeNa* that performs MAC operations only if both the weights and the activations are non-zero values. In 2018, Akhlaghi et al. [8] proposed a runtime technique, *SnaPEA*, that performs MAC operations associated with positive weights first and then negative weights later while monitoring the sign of the partial sum value. Since the activation values from ReLU are always greater than or equal to zero, the convolution operation can be terminated once the partial sum value becomes negative. Notice that such decision should be performed during runtime, since the zero valued activation patterns depend on the test images. In 2021, another method skipping zero operations, *GoSPA* [44], was proposed, which is similar to ZeNa in that MAC operations were performed only when both input activations and weights were non-zero values. Deng et al. [44] constructed a "Static Sparsity Filter" module by leveraging the property that the weight values are static while the activation values are dynamic to filter out zero activations associated with non-zero weights on the fly before MAC operations. Such skipping operation optimization techniques improved the accuracy per Joule, since the transistors associated with skipped operations were not toggled during runtime, saving dynamic power consumption.

*5.3.2 Encoding Sparse Weights/Activations/Feature Maps.* Since memory access operations dominate the power consumption in CNN applications, fetching the weights less frequently from memory by encoding and compressing the weights and the activations can improve resource efficiency

such as the accuracy per memory footprint, per memory access, and per Joule. Han et al. [68, 69] utilized the Huffman encoding scheme to compress the weights. The quantized CNN reduced the memory footprint of AlexNet on the ImageNet dataset by 35× without losing accuracy. In the work of Han et al. [68, 69], a three-stage pipelined operation was performed to reduce the memory footprint of CNNs as follows. The pruned sparse quantized weights were stored with Compressed Sparse Row (CSR) format and then divided into several groups. The weights in the same group were shared with the average value over the weights in the group, and they were retrained thereafter. Huffman coding was used to compress the weights further. Parashar et al. [129] employed an encoding scheme to compress sparse weights and activations and designed an associated dataflow, SCNN (Compressed-Sparse Convolutional Neural Network), to minimize data transfer and reduce memory footprint. Aimar et al. [7] proposed *NullHop* that encodes the sparse feature maps by using two sequentially ordered (i.e., internally indexed) additional storage, one for a 3D mask to indicate the positions of non-zero values and the other for storing the non-zero data sequentially in the feature map. For example, '0's are marked at the position of zero values in the 3D mask, and otherwise '1's are marked. Decoding refers to both the 3D mask and the non-zero value list. Rhu et al. [137] presented *HashedNet* that utilizes a low-cost hash function to compress sparse activations. The virtualized DNN (vDNN) [136] compressed sparse activation units using the "zero-value compression" technique to minimize the data transfer cost between GPU and CPU. The vDNN allowed users to utilize both GPU and CPU memory for DNN training.

*5.3.3 Decomposing Kernel Matrix.* Li et al. [110] proposed SqueezeFlow that reduces the number of operations for convolutions by decomposing the kernel matrix into non-zero valued sub-matrices and zero-valued sub-matrices. This method can improve the accuracy per Joule.

### 5.4 Leveraging Weight Repetition in Quantized CNNs

Hedge et al. [76] noticed that the identical weight values were often repeated in quantized CNNs such as binary weight CNNs [41, 42] and ternary weight CNNs [108, 196] and proposed the **Unique Weight CNN Accelerator (UCNN)** that reduces the number of memory accesses and the number of operations by leveraging the repeated weight values in the quantized CNNs. For example, if a $2 \times 2$ kernel consisting of $\{k_{1,1}, k_{1,2}, k_{2,1}, k_{2,2}\}$ performs a convolution with the activation maps, $\{a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2}\}$. The conventional convolutional operation, $\Sigma_{i=1,j=1}^{i=2,j=2} k_{i,j} \times a_{i,j}$, requires eight read memory accesses, four multiplications, and three additions. If two of the weights in the kernel are identical (e.g., $k_{1,1} = k_{2,2}$ and $k_{1,2} = k_{2,1}$), the convolutional operation can be performed using $k_{1,1}(a_{1,1} + a_{2,2}) + k_{1,2}(a_{1,2} + a_{2,1})$, requiring six read memory accesses, two multiplications, and three additions. The UCNN improved the accuracy per Joule by 1.2 to 4× in AlexNet and LeNet on Eyeriss architecture using the ImageNet dataset.

### 5.5 Leveraging Innovative Technology

Many research attempts have leveraged innovative computing architecture technologies such as neuromorphic computing and in-memory processing as follows.

*5.5.1 Neuromorphic Computing.* Neuromorphic computing mimics the brain, including brain components such as neurons and synapses; furthermore, biological neural systems include axons, dendrites, glial cells, and spiking signal transferring mechanisms [91]. The memristor, "memory resistor," is one of the most widely used devices in neuromorphic computing. *ISAAC* [146] replaced MAC operation units with the memristor crossbars based on the DaDianNao architecture. In the crossbars, every wire in horizontal wire array was connected to every wire in vertical wire array with a resistor. Different level voltages, $V = [v_1, v_2, \ldots, v_m]$, were applied to the horizontal

wires connected to a vertical wire by the different resistors, $R = [1/g_1, 1/g_2, \ldots, 1/g_m]$. With mapping $v_i$ to input elements and $g_i$ to weights, where $i = 1, \ldots, m$, the output current, $I$, from the vertical wire can be represented as MAC operations in a layer, $I = \Sigma_i^m (v_i \times g_i)$, based on Kirchhoff's law. Multiple MAC operations can be performed by collecting the currents from multiple vertical wires. ISAAC employed digital-to-analog converters to receive the input elements and convert them into the appropriate voltages and analog-to-digital converters to convert the current values into digitized feature map values. Due to lack of reprogrammability of resistors in the crossbars, the ISAAC architecture was only available for the inference tasks. ISAAC improved 5.5× accuracy per Joule compared to full-fledged DaDianNao. As another neuromorphic computing approach, many research attempts implemented Hodgkin-Huxley and Morris Lecar models [81] that describes the activity of neurons using non-linear differential equations in hardware simulators [15, 16, 49, 60, 149–151]. Several studies implemented neuromorphic architectures in ASIC, including TrueNorth [9], SpiNNaker [127], Neurogrid [18], BrainScaleS [144], and IFAT [131]. Please refer to the work of Schuman et al. [145] for a comprehensive survey of neuromorphic computing.

*5.5.2 In-Memory Processing.* Scaling down the size of transistors enables energy efficiency and high performance for Von-Neumann computing systems. However, it became quite challenging in the era of sub-10-nm technologies due to physical limitations [11, 54, 124]. To address this challenge, researchers proposed a paradigm of in-memory processing to improve performance and energy efficiency by integrating computations units into memory devices [67, 111, 178]. Several studies proposed to enable in-memory processing to accelerate CNNs [12, 46, 55, 73, 97, 185]. The in-memory accelerators can be implemented using non-volatile memory technologies, such as phase change memory [92, 101], resistive random access memory [37, 61, 62, 94, 133], and SRAM memory devices [185]. Several studies projected that the non-volatile memory based accelerators are able to improve performance significantly compared to state-of-the-art microprocessors [61, 62]. The XNOR-SRAM technology [185] integrating the XNOR gates and accumulation logic into SRAM can fetch data from SRAM and perform MAC operation in one cycle.

### 5.6 Adaptive Compute Resource Assignment

This section comprises the methods assigning runtime compute resources adaptively to the CNN inference workload to improve resource efficiency. The implementation of the CNNs can be adapted to the accuracy requirements of the applications by using various runtime implementation techniques as follows.

*5.6.1 Early Exiting.* The required depth of CNN depends on the problem complexity. The "early exiting" technique allows a CNN to classify an object as early as possible by having multiple exit classifier points in a single CNN [128, 160, 161]. The early exiting technique was applied to distributed computing systems, addressing concern about privacy, response time, and higher quality of experience [161]. Such early exiting methods minimized the compute resources and the inference latency, improving the accuracy per Joule, per operation, and per core utilization. Please refer to the work of Matsubara et al. [122] for details on the early exiting techniques.

*5.6.2 Runtime Channel Width Adaptation.* The runtime channel width adaptation pruned unimportant filters during runtime. In 2018, Fang et al. [50] presented a single DNN model, *NestDNN*, being able to switch between multiple capacities of the DNN during runtime according to the accuracy and inference latency requirement. During training, unimportant filters from the original model were pruned to generate the smallest possible model, the "seed model." Each retraining, some of pruned filters were added to the seed model while fixing the filter parameters from the previous training. Since the seed model was descended from the original model, the accuracy for each

capacity in NestDNN was higher than the model having the identical capacity trained from the scratch. Similarly, Yu et al. [187] proposed another runtime switchable DNN model, the *Slimmable Neural Network*, in which a larger capacity model shared the filter parameters from a smaller capacity model.

*5.6.3 Runtime Model Switching.* Lee et al. [107] selected the best-performing object detectors between multiple DNN detectors during runtime according to dynamic video content to improve the accuracy per core utilization and per Joule. Lou et al. [116] switched between multiple DNNs, generated from the Once-for-All NAS of Cai et al. [25], during runtime according to dynamic workload. For example, when the inference latency of a DNN was increased due to a newly assigned workload, a runtime decision maker downgraded the current DNN during runtime to meet a latency constraint. Such runtime model switching approaches were appropriate when memory resources were sufficient, since the multiple DNNs should be pre-loaded in DRAM.

## 6 INTERRELATED INFLUENCES, INSIGHTS, AND FUTURE TRENDS

This section discusses the influence from higher- to lower-level techniques as shown in Figure 1, resource efficiency metrics according to the techniques, insights into resource-efficient CNNs, and future research trends based on the insights.

### 6.1 Influences of Model-Level Techniques on Arithmetic-Level Techniques

Weight quantization [41, 108, 196] in model-level techniques influenced arithmetic-level techniques as shown in Figure 7. The multiplications using the quantized binary weights can be replaced with multiplexers, removing multiplication arithmetic operations. The resource efficiency from the model-level techniques can be further improved by utilizing the arithmetic-level techniques. For example, quantized CNNs such as ternary weight and binarized CNNs allowed INT8 arithmetic to be used in training [176, 182]. When reduced-precision CNNs suffered from zero gradients, the reduced precision arithmetic was replaced with a hybrid version arithmetic using both BFP and FP [48] or the BM format [52].

### 6.2 Influences of Model-Level Techniques on Implementation-Level Techniques

Weight quantization in model-level techniques influenced the implementation-level techniques as shown in Figure 8. Pruning weights can bring sparsity in the hardware architecture while pruning filters [109, 118] maintains dense structure. Weight quantization in the model-level techniques allows a CNN to utilize fewer bits for weights to save memory resource usage, requiring customized hardware. For example, EIE [68] is an inference accelerator with weights quantized by four bits. To implement the weight quantization method effectively, EIE utilized weight sharing to reduce the model size further and fit the compressed CNN into the on-chip SRAM. Exploring binary weights [42] with binary inputs offered the opportunity to explore XNOR gates for the efficient implementation of CNNs [134], improving the accuracy per memory footprint and per Joule. In the work of Tridgell et al. [162], ternary neural networks [108, 196] were implemented on FPGAs by unrolling convolution operations. Since quantized CNNs [108, 196] increased the number of repeated weights in CNNs, UCNN [76] leveraged the property of the repeated weight values in quantized CNNs to improve resource efficiency such as accuracy per memory access and per operation. As the main limitation, weight quantization methods (e.g., [41, 108, 196]) were not suitable for commercially available CPUs and GPUs, since such computing platforms do not support binary and ternary weights in hardware. Therefore, the implementation of weight quantization methods on CPUs or GPUs might not improve accuracy per Joule, as higher-precision arithmetic still was required in part of data path in training and inference. The bottleneck structures generated by

compact convolutions in other works [89, 142] can be used to reduce the data size transferred between a local device and an edge server for the efficient implementation of edge-based AI [122].

### 6.3 Influences of Arithmetic-Level Techniques on Implementation-Level Techniques

The arithmetic-level techniques influenced the implementation-level techniques as follows.

First, the research in arithmetic utilization acted as a catalyst for commodity CPUs and GPUs. For example, mixed-precision research [125] laid a foundation for tensor cores in the latest NVIDIA GPUs, which can accelerate the performance of CNN workloads by supporting a fused multiply-add operation and the mixed-precision training capability in hardware [19]. The BFloat16 format [24] designed by Google overcomes the limited accuracy issue of the IFP16 format by providing the same dynamic range as IFP32, and it is supported in hardware in Intel Cooper Lake Xeon processors, NVIDIA A100 GPUs, and Google TPUs. In 2016, NVIDIA Pascal GPUs supported IFP16 arithmetic in hardware to accelerate CNN applications. In 2017, NVIDIA Volta GPUs supported IFP16 tensor cores. In 2020, the NVIDIA Ampere architecture supported tensor cores, TF32, BFloat16, and sparsity acceleration in hardware to accelerate MACs [2]. The Graphcore company developed the **Intelligent Processing Unit (IPU)**, which employs local memory assigned to each processing unit with support for a large number of independently operating hardware threads [93]. The IPU is an efficient computing architecture customized to "fine-grained, irregular computation that exhibits irregular data access."

Second, the arithmetic-level techniques led to specialized custom accelerators for CNNs. There is ample evidence in the arithmetic-level literature (e.g., [21, 48, 52, 167, 180]) that even smaller operators (e.g., 16 bits or even less) have almost no impact on the accuracy of CNNs. For example, DianNao [30] and DaDianNao [33] were customized to 16-bit FiP arithmetic operators instead of word-size (e.g., 32-bit) FP operators. ISAAC [146] is a fully fledged crossbar-based CNN accelerator architecture, which implemented a memristor-based logic based on resistive memory, suitable for 16-bit arithmetic for CNN workloads. Wang et al. [167] designed their customized 8-bit FP arithmetic multiplications with 16-bit accumulations on an ASIC-based hardware platform with a 14-nm silicon technology to support energy-efficient CNN training. The Eyeriss [31] and SnaPEA [8] accelerators were customized to 16-bit arithmetic. UCNN [76] utilized 8-bit and 16-bit FiP configurations. SCNN [129] utilized 16-bit multiplication and 24-bit accumulation.

Last, the mixed-precision training schemes were accelerated in hardware by minimizing the data conversion overhead between lower- and higher-precision formats in updating weights and activations [194]. Additionally, the stochastic rounding scheme was supported in hardware in the Intel Loihi processor [43] and Graphcore IPU [93], since it was often required for quantizing weights and activations during training [65, 176, 182].

### 6.4 Resource Efficiency Metrics According to Techniques

Table 1 shows the resource efficiency metrics improved by specific techniques discussed in Sections 3, 4, and 5, by mapping rows for techniques, columns for metrics, and cells marked if the corresponding technique improves the corresponding metric ('◯' for an improved metric directly by a technique and 'E' for the expected improved metric from directly improved metrics.). In Table 1, A/Param, A/Op, A/MF, A/MA, A/CU, and A/Joule represent accuracy per parameter, accuracy per operation, accuracy per memory footprint, accuracy per memory access, accuracy per core utilization, and accuracy per Joule, respectively.

### 6.5 New Insights into Resource-Efficient CNNs and Future Research Trends

Model-level techniques focus on seeking the least sufficient connections between neurons to reduce the number of parameters, whereas arithmetic-level techniques focus on seeking the least

Table 1. Resource Efficiency Metrics According to Techniques

| Category | Techniques | A/Param | A/Op | A/MF | A/MA | A/CU | A/Joule |
|---|---|---|---|---|---|---|---|
| Model Level | Weight Quantization | | | O | E | | E |
| | Pruning | O | O | E | E | E | O/E |
| | Compact Convolution | O | O | E | E | E | E |
| | Knowledge Distillation | O | O | E | E | E | E |
| | Neural Architecture Search | O | O | O | E | E | O |
| Arithmetic Level | Lower-Precision Inferences | | | O | E | E | E |
| | Encoding and Using LUT | | | O | O | | E |
| | Various Number Formats | | | | | O | E |
| | Mixed-Precision Training | | | | O | O | E |
| | BFP Training | | | O | O | O | E |
| Implementation Level | Leveraging Data Reuse | | | | O | | E |
| | Leveraging Fast Convolutoin Algorithms | | O | | | E | |
| | Leveraging Sparsity-Skipping Operations | | | | O | O | E |
| | Leveraging Sparsity-Encoding | | | O | O | O | E |
| | Leveraging Sparsity-Decomposing | | | | O | O | E |
| | Leveraging Weight Repetition | | | | O | O | E |
| | Leveraging Innovative Tech | | | | | | O |
| | Adaptation-Early Exiting | | O | | E | E | E |
| | Adaptation-Channel Width | O | | E | E | E | E |
| | Adaptation-Model Switching | | | | | O | E |

A/Param, accuracy per parameter; A/Op, accuracy per operation; A/MF, accuracy per memory footprint; A/MA, accuracy per memory access; A/CU, accuracy per core utilization; A/Joule, accuracy per Joule; O, directly improved metric by a footprint; A/MA, accuracy per memory access;

sufficient number of bits to represent the data. We observe that CNNs having fewer connections often perform better than the original model [53], whereas lower-precision CNNs do not perform better than the original model. Therefore, we draw insight into resource-efficient CNNs that it is highly probable that reducing number of weights has a regularization effect, whereas reducing the number of bits to represent weights, activations, and gradients does not have a regularization effect according to the bias-variance tradeoff [57]. For example, lower-precision arithmetic can incur overfitting due to the increased rounding errors [5, 106] (e.g., four-bit CNNs in the work of Sun et al. [154] are fit for ResNet18 rather than ResNet50). From this insight, incorporating the arithmetic-level techniques into the model-level techniques will be a promising research direction since we expect that overfitting magnified by lower-precision arithmetic can be minimized by regularizing the model with model-level techniques.

Model-level techniques [70, 75, 78, 159] have reduced parameters by 5 to 10× in general compared to baseline models such as AlexNets, VGGNets, and ResNets, improving the accuracy per parameter. On the contrary, the resource efficiency metrics improved by the arithmetic- and implementation-level techniques rely on computing architectures. For example, reducing the number of bits by 2× generally accelerates multiplications by 4× on ASICs/FPGAs [105]. It has recently been demonstrated that using four bits could be sufficient to represent weights, activations, and their gradients with a specific rounding error scheme called *two-phased rounding* [154]. Since the four-bit FP arithmetic potentially brings approximately 64× speedup to multiplications on ASICs/FPGAs compared to 32-bit FP arithmetic, the arithmetic-level techniques with a special rounding scheme can improve the throughput of training significantly. From this insight, we expect that incorporating the arithmetic-level techniques into ASICs is a promising research direction in the future.

Most resource-efficient CNNs lack in explaining the effects of their techniques to the accuracy according to other dynamic variables such as sample complexity (the number of training samples), target complexity, and CNN complexity. It has been recently reported that increasing the depth of the CNN follows a classic bias-variance tradeoff, whereas increasing the width of the CNN follows the "bell-shaped" variance curve rather than a monotonically increasing pattern [183]. For example,

if the CNN width is already saturated (beyond the peak of variance), increasing the CNN width even with reduced precision arithmetic can improve the accuracy [71]. However, if the CNN width is not saturated yet, increasing the CNN width might hurt the accuracy even with full precision. Increasing the depth of the CNN with reduced-precision arithmetic will have different impacts. In this regard, we suggest a meaningful unexplored research question: given a sample, a target, and a CNN complexity, how can we determine if we need to increase (or decrease) depth (or width) to improve the accuracy of reduced-precision CNNs according to the bias-variance tradeoff?

## 7 CONCLUSION

To the best of our knowledge, our survey is the first to provide a comprehensive survey coverage of the recent resource-efficient CNN techniques based on the three-level hierarchy including model-, arithmetic-, and implementation-level techniques. Our survey also utilizes multiple resource efficiency metrics to clarify which resource efficiency metrics each technique can improve. For example, most model-level resource-efficient techniques contribute to improving abstract resource efficiency, whereas the arithmetic- and implementation-level techniques directly contribute to improving physical resource efficiency by employing reduced-precision arithmetic and/or optimizing the dataflow of CNN architectures. Therefore, the efficient implementation of model-level techniques on given compute platforms is essential to improve physical resource efficiency. It is our hope that this work will contribute to the machine learning, arithmetic, and system communities by providing them with a comprehensive survey for various resource-efficient CNN techniques as guidelines to seek CNN structures using the least sufficient parameters and the least sufficient precision arithmetic on particular compute platforms, customized to the problem complexity and the training data quantity and quality.

## REFERENCES

[1] Papers With Code. (n.d.) ImageNet Benchmark (Image Classification on ImageNet). Retrieved March 15, 2023 from https://paperswithcode.com/sota/image-classification-on-imagenet.

[2] NVIDIA. (n.d.) NVIDIA Ampere Architecture White Paper. Retrieved March 15, 2023 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[3] Google Cloud. (n.d.) Quantifying the Performance of the TPU, Our First Machine Learning Chip. Retrieved March 15, 2023 from https://cloud.google.com/blog/products/gcp/quantifying-the-performance-of-the-tpu-our-first-machine-learning-chip.

[4] IEEE. 2019. *754-2019—IEEE Standard forFloating-Point Arithmetic.* (Revision of IEEE 754-2008). IEEE, Los Alamitos, CA, 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[5] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. 2012. *Learning From Data.* AMLBook.

[6] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan. 2019. DLFloat: A 16-b floating point format designed for deep learning training and inference. In *Proceedings of the 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH'19).* 92–95.

[7] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, et al. 2019. NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems* 30, 3 (2019), 644–656. https://doi.org/10.1109/TNNLS.2018.2852335

[8] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh. 2018. SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18).* IEEE, Los Alamitos, CA, 662–673. https://doi.org/10.1109/ISCA.2018.00061

[9] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, et al. 2015. TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (2015), 1537–1557. https://doi.org/10.1109/TCAD.2015.2474396

[10] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16).* 1–13. https://doi.org/10.1109/ISCA.2016.11

[11] Mustafa F. Ali, Robert Andrawis, and Kaushik Roy. 2020. Dynamic read current sensing with amplified bit-line voltage for STT-MRAMs. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 3 (2020), 551–555. https://doi.org/10.1109/TCSII.2019.2915822

[12] Mustafa F. Ali, Akhilesh Jaiswal, and Kaushik Roy. 2020. In-memory low-cost bit-serial addition using commodity DRAM technology. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 1 (2020), 155–165. https://doi.org/10.1109/TCSI.2019.2945617

[13] Andrew Anderson, Jing Su, Rozenn Dahyot, and David Gregg. 2019. Performance-oriented neural architecture search. In *Proceedings of the 2019 International Conference on High Performance Computing and Simulation (HPCS'19)*. 177–184. https://doi.org/10.1109/HPCS48598.2019.9188213

[14] Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems (NeurIPS'14)*.

[15] Arindam Basu. 2012. Small-signal neural models and their applications. *IEEE Transactions on Biomedical Circuits and Systems* 6, 1 (2012), 64–75. https://doi.org/10.1109/TBCAS.2011.2158314

[16] Arindam Basu, Csaba Petre, and Paul Hasler. 2008. Bifurcations in a silicon neuron. In *Proceedings of the 2008 IEEE International Symposium on Circuits and Systems*. 428–431. https://doi.org/10.1109/ISCAS.2008.4541446

[17] Yoshua Bengio. 2012. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, and Daniel Silver (Eds.), Proceedings of Machine Learning Research, Vol. 27. PMLR, Bellevue, WA, 17–36. http://proceedings.mlr.press/v27/bengio12a.html.

[18] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R. Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V. Arthur, Paul A. Merolla, and Kwabena Boahen. 2014. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE* 102, 5 (2014), 699–716. https://doi.org/10.1109/JPROC.2014.2313565

[19] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. 2020. Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM Journal on Scientific Computing* 42, 3 (2020), C124–C141.

[20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal speed and accuracy of object detection. *arXiv:cs.CV/2004.10934* (2020).

[21] R. Bordawekar, B. Abali, and M. H. Chen. 2021. EFloat: Entropy-coded floating point format for deep learning. *arXiv:2102.02705* (2021).

[22] Léon Bottou and Yann Le Cun. 2004. Large scale online learning. In *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

[23] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*. ACM, New York, NY, 535–541. https://doi.org/10.1145/1150402.1150464

[24] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. 2019. Bfloat16 processing for neural networks. In *Proceedings of the 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH'19)*. 88–91.

[25] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*.

[26] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. 2019. Deep positron: A deep neural network using the posit number system. In *Proceedings of the 2019 Design, Automation, and Test in Europe Conference and Exhibition (DATE'19)*. 1421–1426.

[27] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. In *Proceedings of the 2019 Conference for Next Generation Arithmetic (CoNGA'19)*. ACM, New York, NY, Article 3, 9 pages. https://doi.org/10.1145/3316279.3316282

[28] Chip-Hong Chang, Amir Sabbagh Molahosseini, Azadeh Alsadat Emrani Zarandi, and Tian Fatt Tay. 2015. Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. *IEEE Circuits and Systems Magazine* 15, 4 (2015), 26–44. https://doi.org/10.1109/MCAS.2015.2484118

[29] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. 2017. Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Red Hook, NY, 742–751. http://papers.nips.cc/paper/6676-learning-efficient-object-detection-models-with-knowledge-distillation.pdf.

[30] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGPLAN Notices* 49, 4 (Feb. 2014), 269–284. https://doi.org/10.1145/2644865.2541967

[31] Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 367–379.

[32] Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Shuicheng Yan, and Jiashi Feng. 2019. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV'19)*.

[33] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE, Los Alamitos, CA, 609–622. https://doi.org/10.1109/MICRO.2014.58

[34] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[35] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv:1710.09282* (2017). https://doi.org/10.48550/ARXIV.1710.09282

[36] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. 2018. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine* 35, 1 (2018), 126–136.

[37] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[38] Francois Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[39] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning—ICANN 2014*, Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa (Eds.). Springer International Publishing, Cham, Switzerland, 281–290.

[40] NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture*. WP-08608-001v1.1. NVIDIA.

[41] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Red Hook, NY, 3123–3131. http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.pdf.

[42] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv:cs.LG/1602.02830* (2016).

[43] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, et al. 2018. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 1 (2018), 82–99. https://doi.org/10.1109/MM.2018.112130359

[44] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 1110–1123. https://doi.org/10.1109/ISCA52012.2021.00090

[45] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. 2020. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE* 108, 4 (2020), 485–532. https://doi.org/10.1109/JPROC.2020.2976475

[46] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. 2018. DrAcc: A DRAM based accelerator for accurate CNN inference. In *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC'18)*. 1–6. https://doi.org/10.1109/DAC.2018.8465866

[47] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems—Volume 1 (NeurIPS'14)*. MIT Press, Cambridge, MA, 1269–1277.

[48] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with hybrid block floating point. In *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

[49] D. Dupeyron, S. Le Masson, Y. Deval, G. Le Masson, and J.-P. Dom. 1996. A BiCMOS implementation of the Hodgkin-Huxley formalism. In *Proceedings of 5th International Conference on Microelectronics for Neural Networks*. 311–316. https://doi.org/10.1109/MNNFS.1996.493808

[50] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision *(MobiCom'18)*. ACM, New York, NY. https://doi.org/10.1145/3241539.3241559

[51] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh. 2021. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science* 7, e330 (2021), 1–19. https://doi.org/10.7717/peerj-cs.330

[52] Sean Fox, Seyedramin Rasoulinezhad, Julian Faraone, David Boland, and Philip Leong. 2021. A block minifloat representation for training deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR'21)*.

[53] Jonathan Frankle and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*.

[54] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. 1–14. https://doi.org/10.1109/HPCA.2019.00023

[55] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and efficient neural network acceleration with 3D memory. *ACM SIGARCH Computer Architecture News* 45, 1 (April 2017), 751–764. https://doi.org/10.1145/3093337.3037702

[56] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng Zhong Xu. 2019. Dynamic channel pruning: Feature boosting and suppression. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*.

[57] Stuart Geman, Elie Bienenstock, and René Doursat. 1992. Neural networks and the bias/variance dilemma. *Neural Computation* 4, 1 (1992), 1–58. https://doi.org/10.1162/neco.1992.4.1.1

[58] Deepak Ghimire, Dayoung Kil, and Seong-Heum Kim. 2022. A survey on efficient convolutional neural networks and hardware acceleration. *Electronics* 11, 6 (2022), 945. https://doi.org/10.3390/electronics11060945

[59] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. 2018. SqueezeNext: Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR'18)*.

[60] Meisam Gholami and Saeed Saeedi. 2015. Digital cellular implementation of Morris-Lecar neuron model. In *Proceedings of the 2015 23rd Iranian Conference on Electrical Engineering*. 1235–1239. https://doi.org/10.1109/IranianCEE.2015.7146404

[61] Tayfun Gokmen, Murat Onen, and Wilfried Haensch. 2017. Training deep convolutional neural networks with resistive cross-point devices. *Frontiers in Neuroscience* 11 (2017), 1–13. https://doi.org/10.3389/fnins.2017.00538

[62] Tayfun Gokmen and Yurii Vlasov. 2016. Acceleration of deep neural network training with resistive cross-point devices: Design considerations. *Frontiers in Neuroscience* 10 (2016), 1–13. https://doi.org/10.3389/fnins.2016.00333

[63] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic network surgery for efficient DNNs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates, Red Hook, NY, 1387–1395.

[64] Rishi Raj Gupta and Virender Ranga. 2021. Comparative study of different reduced precision techniques in deep neural network. In *Proceedings of International Conference on Big Data, Machine Learning and Their Applications*, Shailesh Tiwari, Erma Suryani, Andrew Keong Ng, K. K. Mishra, and Nitin Singh (Eds.). Springer Singapore, Singapore, 123–136.

[65] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning—Volume 37 (ICML'15)*. 1737–1746.

[66] Gustafson and Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

[67] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: A framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, New York, NY, 329–345. https://doi.org/10.1145/3445814.3446749

[68] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE, Los Alamitos, CA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[69] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.

[70] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Red Hook, NY, 1135–1143. http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf.

[71] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. 2017. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'17)*. 1478–1483.

[72] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[73] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*. 372–385. https://doi.org/10.1109/MICRO50266.2020.00040

[74] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*.

[75] Y. He, X. Zhang, and J. Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*. 1398–1406.

[76] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 674–687. https://doi.org/10.1109/ISCA.2018.00062

[77] Tyler Highlander and Andres Rodriguez. 2015. Very efficient training of convolutional neural networks using fast Fourier transform and overlap-and-add. In *Proceedings of the British Machine Vision Conference (BMVC'15)*. Article 160, 9 pages. https://doi.org/10.5244/C.29.160

[78] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv:stat.ML/1503.02531 (2015)*.

[79] Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in NVIDIA GPUs. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091072

[80] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[81] A. L. Hodgkin and A. F. Huxley. 1952. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology* 117, 4 (1952), 500–544. https://doi.org/10.1113/jphysiol.1952.sp004764

[82] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2022. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 1 (July 2022), Article 241, 124 pages.

[83] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:cs.CV/1704.04861 (2017).

[84] Huiyi Hu, Ang Li, Daniele Calandriello, and Dilan Gorur. 2021. One pass ImageNet. In *Proceedings of the NeurIPS 2021 Workshop on ImageNet: Past, Present and Future*.

[85] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. arXiv:cs.NE/1607.03250 (2016).

[86] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. CondenseNet: An efficient DenseNet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*.

[87] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[88] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research* 18, 1 (Jan. 2017), 6869–6898.

[89] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:cs.CV/1602.07360* (2016).

[90] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, Francis Bach and David Blei (Eds.). Proceedings of Machine Learning Research, Vol. 37. PMLR, Lille, France, 448–456. http://proceedings.mlr.press/v37/ioffe15.html.

[91] E. M. Izhikevich. 2004. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks* 15, 5 (2004), 1063–1070. https://doi.org/10.1109/TNN.2004.832719

[92] Bryan L. Jackson, Bipin Rajendran, Gregory S. Corrado, Matthew Breitwisch, Geoffrey W. Burr, Roger Cheek, Kailash Gopalakrishnan, et al. 2013. Nanoscale electronic synapses using phase change devices. *ACM Journal on Emerging Technologies in Computing Systems* 9, 2 (2013), Article 12, 20 pages. https://doi.org/10.1145/2463585.2463588

[93] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413* (2019).

[94] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B. Bhadviya, Pinaki Mazumder, and Wei Lu. 2010. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Letters* 10, 4 (2010), 1297–1301. https://doi.org/10.1021/nl904092h

[95] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, et al. 2019. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).

[96] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2018. ZeNA: Zero-aware neural network accelerator. *IEEE Design & Test* 35, 1 (2018), 39–46. https://doi.org/10.1109/MDAT.2017.2741463

[97] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 380–392. https://doi.org/10.1109/ISCA.2016.41

[98] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2016. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.

[99] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, et al. 2017. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in Neural Information Processing Systems*. https://proceedings.neurips.cc/paper/2017/file/a0160709701140704575d499c997b6ca-Paper.pdf.

[100] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[101] Duygu Kuzum, Shimeng Yu, and H.-S. Philip Wong. 2013. Synaptic electronics: Materials, devices and applications. *Nanotechnology* 24, 38 (Sept. 2013), 382001. https://doi.org/10.1088/0957-4484/24/38/382001

[102] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[103] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[104] Yann LeCun, John Denker, and Sara Solla. 1990. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann.

[105] JunKyu Lee, Gregory D. Peterson, Dimitrios S. Nikolopoulos, and Hans Vandierendonck. 2020. AIR: Iterative refinement acceleration using arbitrary dynamic precision. *Parallel Computing* 97 (2020), 102663. https://doi.org/10.1016/j.parco.2020.102663

[106] JunKyu Lee and Hans Vandierendonck. 2021. Towards lower precision adaptive filters: Facts from backward error analysis of RLS. *IEEE Transactions on Signal Processing* 69 (2021), 3446–3458. https://doi.org/10.1109/TSP.2021.3086355

[107] JunKyu Lee, Blesson Varghese, Roger Woods, and Hans Vandierendonck. 2021. TOD: Transprecise object detection to maximise real-time accuracy on the edge. In *Proceedings of the IEEE International Conference on Fog and Edge Computing*. 53–60.

[108] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. In *Proceedings of the Workshop on Efficient Methods for Deep Neural Networks in the 30th International Conference on Neural Information Processing Systems (NeurIPS'16)*.

[109] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient ConvNets. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

[110] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. 2019. SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules. *IEEE Transactions on Computers* 68, 11 (2019), 1663–1677. https://doi.org/10.1109/TC.2019.2924215

[111] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. 288–301.

[112] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime neural pruning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS'17)*. Curran Associates, Red Hook, NY, 2178–2188.

[113] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 393–405.

[114] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*. 2755–2763.

[115] Zhi-Gang Liu and Matthew Mattina. 2020. Efficient residue number system based Winograd convolution. In *Proceedings of the European Conference on Computer Vision (ECCV'20)*. 53–68.

[116] Wei Lou, Lei Xun, Amin Sabet, Jia Bi, Jonathon Hare, and Geoff V. Merrett. 2021. Dynamic-OFA: Runtime DNN architecture switching for performance scaling on heterogeneous embedded platforms. In *Proceedings of the 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPR'21)*. 3104–3112. https://doi.org/10.1109/CVPRW53098.2021.00347

[117] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. 101–108. https://doi.org/10.1109/FCCM.2017.64

[118] J. Luo, J. Wu, and W. Lin. 2017. ThiNet: A filter level pruning method for deep neural network compression. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV'17)*. 5068–5076.

[119] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*.

[120] Zelda Mariet and Suvrit Sra. 2016. Diversity networks: Neural network compression using determinantal point processes. In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.

[121] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through FFTs. *arXiv:1312.5851* (2013). https://doi.org/10.48550/ARXIV.1312.5851

[122] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. 2021. Split computing and early exiting for deep learning applications: Survey and research challenges. *arXiv:eess.SP/2103.04505* (2021).

[123] Warren McCulloch and Walter Pitts. 1943. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5 (1943), 127–147.

[124] Sally A. McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF'04)*. ACM, New York, NY, 162. https://doi.org/10.1145/977091.977115

[125] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich K. Elsen, David Garcia, Boris Ginsburg, et al. 2018. Mixed precision training. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*.

[126] Michael A. Nielsen. 2018. Neural Networks and Deep Learning. Retrieved March 15, 2023 from http://neuralnetworksanddeeplearning.com/.

[127] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R. Lester, Andrew D. Brown, and Steve B. Furber. 2013. SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits* 48, 8 (2013), 1943–1953. https://doi.org/10.1109/JSSC.2013.2259038

[128] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. 2016. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *Proceedings of the 2016 Design, Automation, and Test in Europe Conference and Exhibition (DATE'16)*. 475–480.

[129] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 27–40. https://doi.org/10.1145/3079856.3080254

[130] Behrooz Parhami. 2010. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, NY.

[131] Jongkil Park, Sohmyung Ha, Theodore Yu, Emre Neftci, and Gert Cauwenberghs. 2014. A 65k-neuron 73-mevents/s 22-pj/event asynchronous micro-pipelined integrate-and-fire array transceiver. In *Proceedings of the 2014 IEEE Biomedical Circuits and Systems Conference (BioCAS'14)*. 675–678. https://doi.org/10.1109/BioCAS.2014.6981816

[132] Z. Qin, Z. Zhang, X. Chen, C. Wang, and Y. Peng. 2018. FD-MobileNet: Improved MobileNet with a fast downsampling strategy. In *Proceedings of the 2018 25th IEEE International Conference on Image Processing (ICIP'18)*. 1363–1367.

[133] Malte Rasch, Tayfun Gokmen, Mattia Rigotti, and Wilfried Haensch. 2019. RAPA-ConvNets: Modified convolutional networks for accelerated training on architectures with analog arrays. *Frontiers in Neuroscience* 13 (July 2019), 1–13. https://doi.org/10.3389/fnins.2019.00753

[134] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Computer Vision—ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, Switzerland, 525–542.

[135] Arthur J. Redfern, Lijun Zhu, and Molly K. Newquist. 2021. BCNN: A binary CNN with all matrix ops quantized to 1 bit precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'21)*. 4604–4612.

[136] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, Los Alamitos, CA, Article 18, 13 pages.

[137] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. 2018. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 78–91.

[138] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2015. FitNets: Hints for thin deep nets. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*.

[139] F. Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* 65, 6 (1958), 386–408. https://doi.org/10.1037/h0042519

[140] S. Salamat, M. Imani, S. Gupta, and T. Rosing. 2018. RNSnet: In-memory neural network acceleration using residue number system. In *Proceedings of the 2018 IEEE International Conference on Rebooting Computing (ICRC'18)*. 1–12.

[141] N. Samimi, M. Kamal, A. Afzali-Kusha, and M. Pedram. 2020. Res-DNN: A residue number system-based DNN accelerator unit. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 2 (2020), 658–671.

[142] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*.

[143] Florian Scheidegger, Luca Benini, Costas Bekas, and A. Cristiano I. Malossi. 2019. Constrained deep neural network architecture search for IoT devices accounting for hardware calibration. In *Advances in Neural Information Processing Systems 32*. 6056–6066.

[144] Johannes Schemmel, Andreas Grübl, Stephan Hartmann, Alexander Kononov, Christian Mayr, Karlheinz Meier, Sebastian Millner, et al. 2012. Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system. In *Proceedings of the 2012 IEEE International Symposium on Circuits and Systems (ISCAS'12)*. 702–702. https://doi.org/10.1109/ISCAS.2012.6272131

[145] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. 2017. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963* (2017).

[146] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 14–26.

[147] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. 2002. Dynamic power consumption in virtex™-II FPGA family. In *Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays (FPGA'02)*. ACM, New York, NY, 157–164. https://doi.org/10.1145/503048.503072

[148] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 764–775. https://doi.org/10.1109/ISCA.2018.00069

[149] M. F. Simoni, G. S. Cymbalyuk, M. E. Sorensen, R. L. Calabrese, and S. P. DeWeerth. 2004. A multiconductance silicon neuron with biologically matched dynamics. *IEEE Transactions on Biomedical Engineering* 51, 2 (2004), 342–354. https://doi.org/10.1109/TBME.2003.820390

[150] M. F. Simoni and S. P. DeWeerth. 1999. Adaptation in a VLSI model of a neuron. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 46, 7 (1999), 967–970. https://doi.org/10.1109/82.775396

[151] Mario F. Simoni, Gennady S. Cymbalyuk, Michael Elliott Sorensen, Ronald L. Calabrese, and Stephen P. DeWeerth. 2000. Development of hybrid systems: Interfacing a silicon neuron to a leech heart interneuron. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NeurIPS) 2000, Denver, CO, USA*, Todd K. Leen, Thomas G. Dietterich, and Volker Tresp (Eds.). MIT Press, Cambridge, MA, 173–179.

[152] Suraj Srinivas and R. Venkatesh Babu. 2015. Data-free parameter pruning for deep neural networks. In *Proceedings of the British Machine Vision Conference (BMVC'15)*. Article 31, 12 pages. https://doi.org/10.5244/C.29.31

[153] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. 2019. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Red Hook, NY, 4900–4909. http://papers.nips.cc/paper/8736-hybrid-8-bit-floating-point-hfp8-training-and-inference-for-deep-neural-networks.pdf.

[154] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Swagath Venkataramani Xiaodong, Cui, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. 2020. Ultra-low precision 4-bit training of deep neural networks. In *Advances in Neural Information Processing Systems 33*. 1796–1807.

[155] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE* 105, 12 (2017), 2295–2329.

[156] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2020. Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference. In *Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218516

[157] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*.

[158] Mingxing Tan and Quoc Le. 2019. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). Proceedings of Machine Learning Research, Vol. 97. PMLR, Long Beach, CA, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html.

[159] Mingxing Tan, Ruoming Pang, and Quoc V. Le. 2020. EfficientDet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'20)*.

[160] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *Proceedings of the 2016 23rd International Conference on Pattern Recognition (ICPR'16)*. 2464–2469. https://doi.org/10.1109/ICPR.2016.7900006

[161] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. 328–339. https://doi.org/10.1109/ICDCS.2017.226

[162] Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, and Philip H. W. Leong. 2019. Unrolling ternary neural networks. *ACM Transactions on Reconfigurable Technology and Systems* 12, 4 (Oct. 2019), Article 22, 23 pages. https://doi.org/10.1145/3359983

[163] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NeurIPS'11)*.

[164] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast convolutional nets with fbfft: A GPU performance evaluation. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*.

[165] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid. 2018. Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. 1–8.

[166] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. 2019. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys* 52, 2 (May 2019), Article 40, 39 pages. https://doi.org/10.1145/3309551

[167] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Red Hook, NY, 7675–7684. http://papers.nips.cc/paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.pdf.

[168] Peisong Wang, Xiangyu He, Qiang Chen, Anda Cheng, Qingshan Liu, and Jian Cheng. 2021. Unsupervised network quantization via fixed-point factorization. *IEEE Transactions on Neural Networks and Learning Systems* 32, 6 (2021), 2706–2720.

[169] P. J. Werbos. 1990. Backpropagation through time: What it does and how to do it. *Proceeding of the IEEE* 78, 10 (1990), 1550–1560. https://doi.org/10.1109/5.58337

[170] J. H. Wilkinson (Ed.). 1988. *The Algebraic Eigenvalue Problem*. Oxford University Press, New York, NY.

[171] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[172] H. R. Wilson and Jack D. Cowan. 1972. Excitatory and inhibitory interactions in localized populations of model neurons. *Biophysical Journal* 12 (1972), 1–24.

[173] Shmuel Winograd. 1980. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611970364

[174] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*.

[175] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv:cs.LG/2004.09602* (2020).

[176] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680* (2018).

[177] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. 5987–5995. https://doi.org/10.1109/CVPR.2017.634

[178] Xin Xin, Youtao Zhang, and Jun Yang. 2020. ELP2IM: Efficient and low power bitwise operation processing in DRAM. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 303–314. https://doi.org/10.1109/HPCA47549.2020.00033

[179] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-Sheng Hua. 2019. Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*. 7308–7316.

[180] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[181] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. NetAdapt: Platform-aware neural network adaptation for mobile applications. In *Computer Vision—ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer International Publishing, Cham, Switzerland, 289–304.

[182] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. 2020. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks* 125 (2020), 70–82. https://doi.org/10.1016/j.neunet.2019.12.027

[183] Zitong Yang, Yaodong Yu, Chong You, Jacob Steinhardt, and Yi Ma. 2020. Rethinking bias-variance trade-off for generalization of neural networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. Article 998, 11 pages.

[184] Reza Yazdani, Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. The dark side of DNN pruning. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 790–801. https://doi.org/10.1109/ISCA.2018.00071

[185] S. Yin, Z. Jiang, J. Seo, and M. Seok. 2020. XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits* 55, 6 (2020), 1733–1743.

[186] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992* (2018).

[187] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. 2019. Slimmable neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*.

[188] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. Lin, and L. S. Davis. 2018. NISP: Pruning networks using neuron importance score propagation. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'18)*. 9194–9203.

[189] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision—ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, Switzerland, 818–833.

[190] C. Zhang, P. Patras, and H. Haddadi. 2019. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys Tutorials* 21, 3 (2019), 2224–2287.

[191] Chi Zhang and Viktor Prasanna. 2017. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 35–44. https://doi.org/10.1145/3020078.3021727

[192] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. 2020. Fixed-point back-propagation training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'20)*.

[193] X. Zhang, X. Zhou, M. Lin, and J. Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'18)*. 6848–6856.

[194] Yongwei Zhao, Chang Liu, Zidong Du, Qi Guo, Xing Hu, Yimin Zhuang, Zhenxing Zhang, et al. 2021. Cambricon-Q: A hybrid architecture for efficient training. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*. 706–719. https://doi.org/10.1109/ISCA52012.2021.00061

[195] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2018. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv:cs.NE/1606.06160.

[196] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained ternary quantization. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

[197] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified INT8 training for convolutional neural network. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'20)*. 1966–1976. https://doi.org/10.1109/CVPR42600.2020.00204

[198] Barret Zoph and Quoc Le. 2017. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.

Received 22 December 2021; revised 1 January 2023; accepted 23 February 2023