SURVEY

Lightweight Deep Learning for Resource-Constrained Environments: A Survey

HOU I LIU, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

MARCO GALINDO, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

HONGXIA XIE, Jilin University, Changchun, Jilin, China

LAIKUAN WONG, Multimedia University, Cyberjaya, Selangor, Malaysia

HONGHAN SHUAI, National Yang Ming Chiao Tung University, Hsinchu, Taiwan

YUNGHUI LI

View all

Lightweight Deep Learning for Resource-Constrained Environments: A Survey

HOU-I LIU, National Yang Ming Chiao Tung University, Hsinchu, Taiwan
MARCO GALINDO, National Yang Ming Chiao Tung University, Hsinchu, Taiwan
HONGXIA XIE, Jilin University, Changchun, China
LAI-KUAN WONG, Multimedia University, Cyberjaya, Malaysia
HONG-HAN SHUAI, National Yang Ming Chiao Tung University, Hsinchu, Taiwan
YUNG-HUI LI, Foxconn Research, Taipei, Taiwan
WEN-HUANG CHENG, National Taiwan University, Taipei, Taiwan

Over the past decade, the dominance of deep learning has prevailed across various domains of artificial intelligence, including natural language processing, computer vision, and biomedical signal processing. While there have been remarkable improvements in model accuracy, deploying these models on lightweight devices, such as mobile phones and microcontrollers, is constrained by limited resources. In this survey, we provide comprehensive design guidance tailored for these devices, detailing the meticulous design of lightweight models, compression methods, and hardware acceleration strategies. The principal goal of this work is to explore methods and concepts for getting around hardware constraints without compromising the model's accuracy. Additionally, we explore two notable paths for lightweight deep learning in the future: deployment techniques for TinyML and Large Language Models. Although these paths undoubtedly have potential, they also present significant challenges, encouraging research into unexplored areas.

CCS Concepts: • **Computing methodologies → Neural networks**; **Artificial intelligence**; **Computer vision**; **Model compression**; • **Computer systems organization → Embedded systems**; • **Software and its engineering → Designing software**; **Software design techniques**;

Additional Key Words and Phrases: Lightweight model, efficient transformer, model compression, quantization, tinyML, large language models

1 INTRODUCTION

Over recent years, the importance of **neural networks (NNs)** has escalated tremendously, with their applications permeating various aspects of daily life and extending to support complex tasks [18, 84, 221]. However, since the publication of AlexNet [110] in 2012, there has been a prevailing trend toward creating deeper and more intricate networks to enhance accuracy. For instance, Model Soups [214] has achieved remarkable accuracy on the ImageNet dataset, but it comes at the cost of over 1,843 million parameters. Similarly, GPT-4 [10] has demonstrated outstanding performance on **natural language processing (NLP)** benchmarks, albeit with a staggering 1.76 trillion parameters. Notably, Amodei et al. [4] indicated that the computational demands of **deep learning (DL)** have surged dramatically, increasing by approximately 300,000 times from 2012 to 2018. This dramatic increase in size sets the stage for the challenges and developments explored in this article.

Concurrently, Green AI [168, 187] has arisen as a prominent concern over the past few years, labeling hefty DL models unsuitable due to their substantial GPU and training time demands, which can contribute to environmental degradation. Strubell et al. [177] extensively analyze the carbon footprint of language models trained on multiple GPUs. In parallel, lightweight devices have garnered increased attention due to their versatile applications and portability. According to Sinha [173], the number of connected IoT devices grew by 18% in 2022, reaching 14.4 billion, and has a projected escalation to 29.0 billion by 2027. A testament to this growing demand is the production of over 200 million iPhones since 2016. However, edge devices offer superior automation and energy efficiency compared to mobile devices, especially the deployment of ultra-low-cost **microcontrollers (MCUs)** in devices such as pacemakers and forehead thermometers [46].

In response to the practical demands outlined above, a significant body of research has emerged in recent years, focusing on lightweight modeling, model compression, and acceleration techniques. The Annual **Mobile AI (MAI)** workshops have been held consecutively during CVPR 2021–2023 [138–140], with a primary emphasis on the deployment of DL models for image processing on resource-constrained devices, such as ARM Mali GPUs and Raspberry Pi 4. Additionally, the **Advances in Image Manipulation (AIM)** workshops conducted at ICCV 2019, ICCV 2021, and ECCV 2022 [3] have organized challenges centered around image/video manipulation, restoration, and enhancement on mobile devices.

From our study, we discovered that the most effective approach for analyzing the development of an efficient, lightweight model, spanning from its design phase to deployment, involves incorporating three key elements into the pipeline: NN architecture design, compression methods, and hardware acceleration for lightweight DL models. Previous surveys [11, 62, 69, 121, 164] often focus on specific aspects of this pipeline, such as discussing only quantization methods, offering detailed insights into those segments. However, these surveys may not provide a comprehensive view of the entire process, potentially overlooking significant alternative approaches and techniques. In contrast, our survey covers lightweight architectures, compression methods, and hardware acceleration algorithms.

1.1 Neural Network Design

In the first part of this article, Section 2, we examine the classic lightweight architectures, categorizing them into family series for improved clarity. Some of these architectures made significant

strides by introducing innovative convolution blocks. For instance, depthwise separable convolutions [35] prioritize high accuracy and reduced computational demand. Sandler et al. [167] introduce an inverted residual bottleneck to enhance gradient propagation. Other architectures, such as ShuffleNet [247], were able to develop an optimized convolution operation, which applies group convolution [110] to achieve a parallel design and further improve the transferability between groups of data through shuffle operations. The ShiftNet [216] achieves an equivalence effect of traditional convolution with no parameters or **Floating Point Operations (FLOPs)**. The AdderNet [21] replaces the multiplication operation with the addition operation, greatly reducing computation requirements.

It is also important to note that parameters and FLOPs do not consistently correlate with inference time. Early lightweight architectures, such as SqueezeNet [98] and MobileNet [89], aim to reduce parameters and FLOPs. However, this reduction often increases **Memory Access Cost (MAC)** [137], leading to slower inference. Hence, we aim to contribute to the application of lightweight models by providing a more comprehensive and insightful review.

1.2 Neural Network Compression

In addition to lightweight architecture designs, Section 3 mentions various efficient algorithms that can be applied to compress a given architecture. For example, quantization methods [97, 131, 229] aim to reduce the required storage for data, often by substituting 32-bit floating-point numbers with 8-bit or 16-bit numbers or even utilizing binary values to represent the data. Pruning algorithms [54, 67, 114], in their simplest form, remove parameters from a model to eliminate unnecessary redundancies within the network. Yet, more sophisticated algorithms may remove entire channels or filters from the network [81, 134]. **Knowledge distillation (KD)** techniques [62, 85] explore the concept of transferring knowledge from one model, referred to as the "teacher," to another, called the "student." The teacher represents a large pre-trained model with the desired knowledge, whereas the student denotes an untrained smaller model tasked with extracting knowledge from the teacher. However, as methods evolved, some algorithms [5, 238] modify the methodology by using the same network twice, eliminating the need for an extra teacher model. As these various compression methods progress, it is common to observe the adoption of two or more techniques, exemplified by the fusion of methods such as pruning and quantization in the same model.

Additionally, we discuss **Neural Architecture Search (NAS)** algorithms, a set of techniques designed to automate the model creation process while reducing human intervention. These algorithms autonomously search for optimal factors within a defined search space, such as network depth and filter settings. Research in this domain primarily focuses on refining the definition, traversal, and evaluation of the search space to achieve high accuracy without excessive time and resource consumption.

1.3 Neural Network Deployment

In Section 4, we navigate through the landscape of prevalent hardware accelerators dedicated to DL applications, including **Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs)**, and **Tensor Processing Units (TPUs)**. Moreover, we describe various dataflow types [23, 65, 103, 127] and delve into data locality optimization methods [145, 176, 240], exploring the intricate techniques that underpin efficient processing in DL workflows. The narrative further unfolds with a discussion of popular DL libraries [1, 24, 152] tailored for accelerating DL processes. This review encompasses the diverse tools and frameworks playing pivotal roles in optimizing the utilization of hardware accelerators. Additionally, we investigate co-designed solutions [32, 151, 211], where achieving optimized and holistic results in accelerated DL requires careful consideration of hardware architecture and compression methods.

1.4 Challenge and Future Work

Last, in Section 5, we embark on an exploration of emerging TinyML techniques designed to execute DL models on ultra-low-power devices, like MCUs, which typically consume less than 1 mW of power. Additionally, our article delves into the intricacies of **Large Language Models (LLMs)**, which present deployment challenges on devices with limited resources due to their enormous model sizes. As promising avenues in computer vision, deploying these methods on edge devices is crucial for widespread application.

1.5 Contributions

This article aims to describe in a simple but accurate manner how lightweight architectures, compression methods, and hardware techniques can be leveraged to implement an accurate model in a resource-constrained device. Our main contributions are summarized below:

(1) Previous surveys only briefly reference a small number of works on lightweight architecture. We organize lightweight architectures into series, such as grouping MobileNetV1-V3 and MobileNeXt in the MobileNet series, and provide a history of lightweight architectures from their inception to the present.

(2) To cover the entire lightweight DL applications, we also cover the compression and hardware acceleration methods. Unlike many other surveys that do not explicitly establish connections between these techniques, our survey offers a thorough overview of each domain, providing a comprehensive understanding of their interconnections.

(3) As part of the forefront advancements in lightweight DL, we review the present challenges and explore future works. First, we explore TinyML, an emerging approach engineered for deploying DL models on devices with remarkably constrained resources. Subsequently, we investigate various contemporary initiatives harnessing LLMs on edge devices, a promising direction in the realm of lightweight DL.

2 LIGHTWEIGHT ARCHITECTURE DESIGN

To ease readers' comprehension, we first introduce the fundamental knowledge of lightweight architecture, including the general metrics to estimate the computation cost of the NN and the widely used mechanisms of model compression. Following that, we outline the lightweight CNN architecture and separate the sections by series, such as ShuffleNet and MobileNet series, according to their chronological order so they can reflect the evolution of lightweight design and the advantage of its efficiency. Additionally, we discuss the efficient transformer, which offers a promising model capacity while maintaining a lightweight architecture.

2.1 Prior Knowledge of Lightweight Architecture

2.1.1 Evaluation Metrics for Deep Learning Model. In DL, the three most commonly used metrics for model compression are **Floating Point Operations (FLOPs), Multiply-Accumulate Operations (MACs)**, and **Memory Access Cost (MAC)**. FLOPs is the number of arithmetic operations the model performs on the floating points, including addition, subtraction, multiplication, and division [7]. Similar to FLOPs, MACs also represent the total number of the floating point operations; however, MACs treat addition and multiplication as equivalent operations, in contrast to FLOPs, which distinguish between them [57]. Consequently, FLOPs $\approx$ 2×MACs. However, MAC represents the amount of memory footprint of an NN, which corresponds to RAM usage [137]. Let H and W be the spatial size of the input and output feature maps for a convolution layer, C_{in} is the number of input channels, C_{out} is the number of output channels, and the kernel size is k,

$$MAC = H \cdot W(C_{in} + C_{out}) + k \cdot k(C_{in} \times C_{out}). \tag{1}$$

Specifically, the first and second terms of Equation (1) depict the memory footprint of the feature maps and weights for that particular convolution layer.

Furthermore, the most widely used metrics for measuring the inference speed of a model are throughput and latency. Throughput refers to the amount of data that can be processed or the number of tasks executed within a specified period. During inference, throughput is measured by the number of inferences per second. Latency is a measure of timing between the input data arriving at a system and the output data being generated and can be expressed in seconds per inference. The relationship between throughput and latency can be derived directly, and the detailed formula can be found in Reference [183].

2.1.2 Pointwise Convolution. The pointwise convolution, also known as a 1×1 convolution, was first introduced in the inception module [185]. The inception module inserts the pointwise convolutions at the bottleneck to obtain deeper features with fewer FLOPs. Empowered by the adaptability of pointwise convolutions to accommodate modifications to the channel's dimensions, the Inception series of works was born [35, 184, 186]. Significantly, pointwise convolutions directly affect the model's computation time and the information richness of the architecture.

2.1.3 Group Convolution. The group convolution idea was proposed by AlexNet [110]. Group convolutions aim to divide the channels of feature maps into several groups and apply convolutions separately to each group. This process helps to reduce computational complexity by N times, where N represents the number of groups.

However, there are still several shortcomings in group convolutions. First, group assignments are fixed, and this factor restricts the information flow between groups, inevitably harming performance. Second, group convolutions cost additional MAC, especially when the number of groups is large, resulting in a much longer inference time. To solve the first problem, ShuffleNet [247] shared group features to obtain deeper channel information. CondenseNet [93] progressively prunes the unimportant connections using **learned group convolutions (LGCs)**. Several works [209, 250] attempt to improve the original LGC to learn better optimal group structures. Furthermore, **Dynamic Group Convolution (DGC)** [178] highlights the importance of input channels via a salience generator and then uses a channel selector to assign groups adaptively.

2.1.4 Depthwise Separable Convolution. The idea of a depthwise separable convolution was proposed in Xception [35], which is the advanced version of the Inception family [184, 186]. A depthwise separable convolution consists of a depthwise convolution followed by a pointwise convolution. According to the MAC, this is a computation-saving but time-consuming operation. To address this issue, Tan et al. [192] aggregate multiple kernel sizes into a single depthwise convolution and use AutoML [78] for navigating the search space.

2.2 Lightweight CNN Architecture

2.2.1 SqueezeNet Series. SqueezeNet series [59, 98] is an early application to reduce parameters using pointwise convolution. SqueezeNet [98] proposes the fire module that constitutes the squeeze layer and the expand layer. The squeeze layer consists of pointwise convolution. It first squeezes features into lower dimensions and then passes them through an expansion layer, which separates the convolution operation into a pointwise convolution and a 3×3 convolution. To solve the gradient vanishing problem and decrease the computation cost, SqueezeNext [59] keeps the shortcut concept from ResNet [77] and decomposed 3×3 kernel into two low-rank kernels with sizes of 3×1 and 1×3. This augmented design reduces the parameters of the kernels from k^2 to $2k$, hence solving the inefficient problem of using depthwise separable convolutions. Compared to

AlexNet [110], SqueezeNet and SqueezeNext reduce the parameters by 50× and 112×, respectively, while keeping AlexNet's level of accuracy on the ImageNet dataset.

2.2.2   ShuffleNet Series. The primary purpose of the ShuffleNet series [137, 247] is to improve the performance of group convolutions and the memory efficiency of depthwise separable convolutions. After a group convolution, each group's output features form an individual channel, and performance suffers due to information not being shared between channels. To address this limitation, ShuffleNet [247] applies a channel shuffle mechanism after the 1×1 group convolution to facilitate cross-group information exchange so features can maintain more global information channels.

ShuffleNetV2 [137] investigates four practical guidelines to design a memory-efficient and light-weight model that can avoid heavy MAC problems. First, equal input and output dimensions mean a smaller MAC. Second, MAC is large when groups are large, particularly for depthwise separable convolutions. Third, it is best to avoid designing a wide network like the Inception series [184–186], because network fragments can result in a large MAC. Last, since element-wise manipulation in a network requires extra computation, avoiding it is efficient. This is often overlooked, because it represents only a few FLOPs but increases MAC, as in depthwise separable convolutions.

2.2.3   CondenseNet Series. Since shortcut connections effectively prevent the gradient vanishing problem [77], some studies, such as DenseNet [94] and the CondenseNet series [93, 226], attempt to optimize NN structure based on shortcut connections. DenseNet [94] replaces the shortcut connections with dense connections, thus improving gradient flow within the bottleneck. Although dense connections increase the accuracy, CondenseNet [93] observes that the magnitude of the connections far from the layers will decay exponentially, causing them to be heavy and slow. To this end, CondenseNet utilizes **learned group convolutions (LGCs)** to prune connections progressively. Before training, the filters are split into G groups of equal size. Suppose C_{in} is the number of input channels, C_{out} is the number of output channels, and F_{ij}^g denotes the kernel weights, including the weights of jth input and ith output within a group $g \in G$. The importance of the jth incoming feature map for the filter group g is computed by the averaged absolute value of weights between them across all outputs within the group, i.e., $\sum_{i=1}^{\frac{C_{out}}{G}} |F_{ij}^g|$, where columns in F^g with small L1-norm value can be removed by zeroing them out. The structured sparsity within a group can be evaluated by applying the group-lasso regularizer [239],

$$r = \sum_{g=1}^{G} \sum_{j=1}^{C_{in}} \sqrt{\sum_{i=1}^{\frac{C_{out}}{G}} F_{ij}^{g\,2}}. \tag{2}$$

By using Equation (2), connections to less important features, represented by a small sparsity value, will be removed, resulting in effective pruning.

Recently, CondenseNetV2 [226] pointed out that the fixed connection mode limits the opportunities for feature reuse. To address this limitation, CondenseNetV2 aims to reactivate outdated features through a novel sparse feature reactivation module. In CondenseNetV2, the weight connections within each block are learned during training, as opposed to CondenseNet, which fixes the model's weight connections after pruning. As a result, this approach results in a performance gain by leveraging the underlying connection.

Figure 1 illustrates a graphical comparison, highlighting the architectural differences between DenseNet, CondenseNet, and CondenseNetV2. In DenseNet, weights between layers in a block are fully connected, where weights in one layer are connected to weights in all other layers (solid-colored arrows). To make the network more efficient, CondenseNet uses LGCs to prune weight
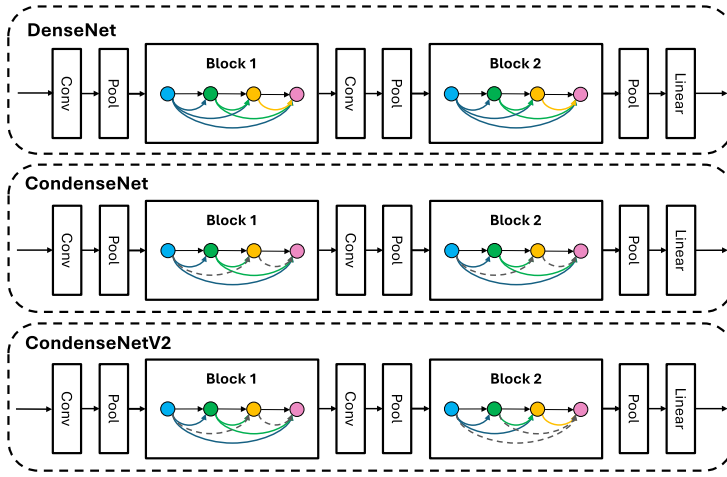
Fig. 1. Comparison of DenseNet, CondenseNet, and CondenseNetV2. Active weight connections are represented by solid color arrows, and pruned weight connections are represented by gray dashed arrows.

connections (gray dashed arrows), and once pruned, the connections for every block remain fixed, e.g., the connections in Block 1 and Block 2 are identical. CondenseNetV2 proposes a sparse feature reactivation mechanism to learn the connections' weights automatically during training. From Figure 1, we can observe that in Block 2 of CondenseNetV2, two pruned connections in Block 1 are reactivated while another two previously active connections in Block 1 are removed, demonstrating the dynamic nature of CondenseNetV2.

2.2.4   MobileNet Series. This series [88, 89, 167, 252] includes prominent CNN models that can be deployed on IoT devices. Based on VGG [172] architecture, MobileNet [89] applies depthwise separable convolutions to create an efficient model, which is shown to perform significantly faster across a broad range of tasks and applications. Discovering that ReLU activations can lead to severe information loss of features with lower dimensions, MobileNetV2 [167] replaces the ReLU activation with a linear combination in the last layer of the residual bottleneck to mitigate the information loss. In addition, MobileNetV2 introduces an inverted residual block, where the number of channels is first increased and then recovered in the residual bottleneck, improving the accuracy. Shortcut connections [77] are also added to enhance the gradient propagation.

NetAdapt [227] applies layer-wise optimization to simplify the network and to achieve high accuracy within limited hardware resources. Based on this, MobileNetV3 [88] leverages platform-aware NAS [189] to optimize the block-wise structure and implements SENet [91] (channel attention module) in the bottleneck structures, resulting in better accuracy. To reduce MAC and establish a quantization-friendly network, ReLU is replaced with H-swish activation. As an alternative to the inverted residual block, MobileNeXt [252] develops a Sandglass block by flipping the inverted residual block to enhance the transmission of wider architectures, since wider layers might lead to more gradient confusion, making model training harder.

2.2.5   Shift-based Series. CNN is computationally expensive due to many multiplication and addition operations. ShiftNet [216] pioneered the replacement of spatial convolutions with Group Shift convolution. Unlike standard convolutions, shift convolutions only perform shifting operations on feature maps and apply padding to those offset areas. In contrast to multiplication operations, shift convolution can achieve zero parameters and FLOPs, thus drastically reducing their number.

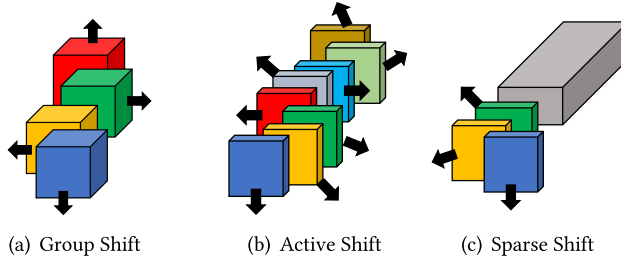(a) Group Shift   (b) Active Shift   (c) Sparse Shift

Fig. 2. The variant of shift-based convolution [26].

Some studies attempt to improve the performance based on shift convolution layers. For example, Jeon et al. [100] propose an Active Shift Layer that makes shifts learnable instead of heuristic assignments. Chen et al. [26] point out that because the number of shifts is fixed, implementing them requires a lot of trial-and-error, limiting the network's functionality. Thus, they propose a Sparse Shift Layer to eliminate meaningless memory movement. The non-shift channels remain the same. Figure 2 compares these three shift operations.

AddressNet [82] observes that a smaller amount of parameters or computation (FLOPs) does not always lead to a direct reduction in inference time, even with shift convolution's zero parameters and zero FLOPs [216]. To optimize the speed of GPU-based machines, AddressNet changes channel shuffle [247] to channel shift, since channel shuffle produces additional memory space and time-consuming permutations, further eliminating the redundant direction. Similar to AdderNet [21], DeepShift [48] is constructed solely with addition operations, replacing all multiplications with bit-wise shifts and sign flips, significantly reducing the operation time and energy consumption.

2.2.6 Add-based Series. Multiplication and addition operations constitute many convolution operations, resulting in extra calculations. AdderNet [21] attempts to exclusively use additions using an L1-norm distance as a response criterion between filters and feature maps. This operation is known as Absolute-difference-accumulation [200], and it accelerates the network and allows the reuse of computation results to reduce energy consumption.

You et al. [234] introduce ShiftAddnet, focusing more on hardware efficiency. ShiftAddnet proposes a new metric for performance comparison, expressive capacity, which refers to the accuracy achieved by the model under similar hardware conditions. Experimental results show that shift-based networks [26, 48, 82, 100, 216] provide greater hardware efficiency but have a lower expressive capacity than multiplication-based networks. Conversely, the fully additive network [21] is inefficient, since repeated additions are used to replace multiplications, although it can achieve better accuracy. Therefore, ShiftAddnet combines the benefits of bit-wise shifts [48] and the efficiency of additive networks [21] to achieve state-of-the-art results on two IoT datasets: FlatCam [188] and Head Pose [203].

2.2.7 EfficientNet Series. Almost all networks attempt to improve performance by adjusting depth, width, and resolution. To achieve the best performance and lightweight combination, it is crucial to pick the right combination. EfficientNet [190] proposes a simple grid search algorithm, compound scaling, to seek scaling factors (depth, width, and resolution), achieving accuracy with lower computation costs. EffectiveNetV2 [191] proposes a training-aware NAS to find a good trade-off for accuracy A, training speed S, and parameters P. It uses a search reward formulated as a simple weighted product, $A \cdot S^w \cdot P^v$, where $w = -0.07$ and $v = -0.05$ are empirically determined to balance the tradeoff. To address the inefficiency of depthwise convolution, EfficientNetV2 replaces stage 1-3 MBConv [167] with Fused-MBConv [71] in its architecture design, offering better

Table 1. Comparison of Lightweight CNN Architectures on the ImageNet Dataset

Model	Top-1	Top-5	Params. (M)	MACs (G)
AlexNet [110]	57.1	80.3	60.9	0.725
ResNet-50 [77]	76.0	93.0	26.0	4.100
SqueezeNet [98]	57.5	80.3	1.2	0.837
SqueezeNext [59]	59.1	82.6	0.7	0.282
ShuffleNetV1-1.5 [247]	71.5	–	3.4	0.292
ShuffleNetV2-1.5 [137]	72.6	90.6	3.5	0.299
1.0-MobileNetV1 [89]	70.6	–	4.2	0.569
MobileNetV2-1.4 [167]	74.7	–	6.9	0.585
MobileV3-S [88]	67.4	–	2.5	0.056
MobileV3-L [88]	75.2	–	5.4	0.219
MobileNeXt-1.0 [252]	74.0	–	3.4	0.300
ShiftResNet-20 [216]	68.6	–	0.2	0.046
ShiftResNet-56 [216]	72.1	–	0.6	0.102
ShiftNet-A [216]	70.1	89.7	4.1	1.400
ShiftNet-B [216]	61.2	83.6	1.1	0.371
FE-Net-1.0 [26]	72.9	–	3.7	0.301
FE-Net-1.37 [26]	75.0	–	5.9	0.563
AddressNet-20 [82]	68.7	–	**0.1**	**0.022**
AddressNet-44 [82]	73.3	–	0.2	0.053
AdderNet-Resnet18 [21]	67.0	87.6	3.6	–
AdderNet-Resnet50 [21]	74.9	91.7	7.7	–
DenseNet-169 [94]	76.2	93.2	14.0	3.500
DenseNet-264 [94]	77.9	93.9	34.0	6.000
CondenseNet [93]	71.0	90.0	2.9	0.274
CondenseV2-A [226]	64.4	84.5	2.0	0.046
CondenseV2-B [226]	71.9	90.3	3.6	0.146
EfficientNet-B1 [190]	79.2	94.5	7.8	0.700
EfficientNet-B7 [190]	84.4	**97.1**	66.0	37.000
EfficientNet-X-B7 [118]	84.7	–	73.0	91.000
EfficientNetV2-S [191]	83.9	–	24.0	8.800
EfficientNetV2-M [191]	85.1	–	55.0	24.000
EfficientNetV2-L [191]	**85.7**	-	121.0	53.000

Note that we use **bold** to emphasize the models with the best accuracy, least parameters,
and lowest MACs, with the respective values being also underlined for enhanced readability.

performance and tradeoff in terms of accuracy, parameters, and FLOPs. Besides, for a more robust network, EfficientNetV2 selects adaptive regulation during the training process, because using identical regularization terms for images of different resolutions is inefficient.

2.2.8 Discussion and Summary. Table 1 compares the performance of lightweight CNN architectures on the ImageNet dataset. The horizontal lines separate the models of different series. From the table, we can observe that there is no one-size-fits-all architecture. Oftentimes, it is a tradeoff between accuracy and efficiency. For example, AddressNet-20 maximizes efficiency at the expense of accuracy. Conversely, the most accurate variants of the EfficientNet series are among the least efficient ones. Drawing from this analysis, we provide recommendations on selecting the suitable models and hardware.
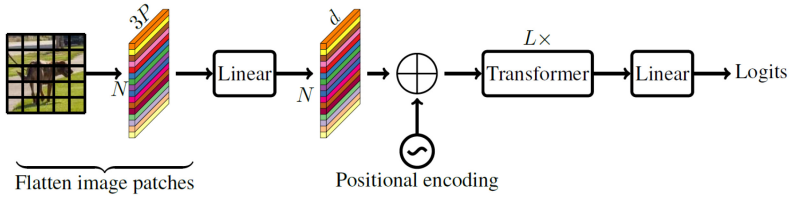
Fig. 3. Standard vision transformer, where $P = h \times w$, h, w represents the height and the width of the images. N is the number of image patches, L is the number of transformer blocks, and d is the dimension [144].

How to choose an adequate lightweight model and compatible hardware? The first crucial step is to check lightweight models' specifications and hardware compatibility. For example, depthwise separable convolutions have huge MAC and high RAM requirements. It is, therefore, imperative to employ a network on hardware that considers both RAM and storage capacity. To this end, Fan et al. [50] redesign the depthwise separable convolution and channel shuffle modules to be hardware-friendly on FPGA. Moreover, to minimize the inference time and to support deployment on a small target device, replacing multiplication with shift or add operations can effectively reduce the total parameters and MACs/FLOPs. Thus, ShiftNet or AdderNet series can be a good choice, since they require smaller parameters and MACs. Within these two series, AddressNet-20 gives the best performance. For target devices with relatively more storage, such as mobile phones or GPUs, models with higher accuracy are preferred for a better user experience. EfficientNetV2-L can thus be considered, since it achieves the highest Top-1 accuracy. However, it is important to note that the EfficientNet series costs a disproportionately higher number of parameters and MACs, which limits the application under low-end devices. Another way to achieve a better tradeoff model is to apply fundamental compression methods such as pruning, quantization, and NAS [29, 189] (see Section 3) to adjust the architecture. This can be an efficient technique to reduce MACs/FLOPs, parameters, and inference time.

Some lightweight methods, such as SqueezeNet and ShuffeNet, may not be able to take full advantage of GPU-accelerated performance due to the lack of customized designs [200]. Additionally, if pruning is applied to a network, like the CondenseNet series, then the network structure might be irregular, preventing the target device from supporting it. In such a scenario, parallelism requires specifically designed computing hardware. Fortunately, customized hardware can be designed to fit a lightweight model. For instance, Um et al. [200] note that CIM is incompatible with AdderNet, because it cannot predict details of an absolute difference nor reuse the computation results. Therefore, they designed a novel ADA-CIM processor offering low-cost sign prediction and higher energy efficiency.

2.3 Transformer-based Series

Transformer models are widely used in NLP [202] and have recently obtained promising results in computer vision tasks [132, 133, 243]. Figure 3 shows the architecture of a typical vision transformer. Transformers are notable for having a significant drawback in that they require a large number of parameters and a high MAC to maintain their performance, which results in a significant amount of time needed for both the training and inference phases, particularly when the input sequence is long. Additionally, the computation and network structures inside transformers are more complex than those of CNNs. The huge number of FLOPs and parameters make practical inference and hardware deployment more difficult. To bridge the gap between transformers and real-world applications, efficient transformers will be discussed in the following sub-sections.

Table 2. The Complexity of Efficient Transformers [208]

Model	Complexity per Layer	Sequential Operation
Transformer [202]	$O(N^2)$	$O(N)$
Sparse Transformer [31]	$O(N\sqrt{N})$	$O(1)$
Linformer [208]	$O(N)$	$O(1)$
Reformer [108]	$O(Nlog(N))$	$O(log(N))$

2.3.1 Lite Attention Module. To address the heavy MAC and huge computation requirements in the self-attention layers, **Long-Short Range Attention (LSRA)** [219] was proposed to extract the global and local features separately, alleviating the attention computations in the **feed-forward network (FFN)**. Child et al. [31] effectively exploit stride and fixed operations to form a sparse connectivity matrix. Linformer [208] decomposes self-attention into several low-rank matrices using linear projection, reducing the complexity of self-attention from N^2 to N, where N denotes the sequence length. Choromanski et al. [36] propose a linear self-attention mechanism based on the **FAVOR+ (fast attention with positive orthogonal random features)** approach to construct an approximate softmax operation. FAVOR+ enables unbiased estimation of self-attention with low estimation variance, reducing spatial and temporal complexity. Reformer [108] utilizes locality-sensitive hashing to replace dot product operations in attention. It directly decreases the computation requirements from N^2 to $Nlog(N)$, allowing longer sequence inputs to be considered. In addition, Reformer employs a reverse residual layer [60] to save GPU memory by L times (number of layers). Unlike traditional residuals, a reverse residual layer does not require activation data to be stored in each layer. The complexity of these efficient transformers is depicted in Table 2.

In addition, transformers stack many FFNs to obtain better-integrated features. Specifically, an FFN is a series of linear transformations that require a lot of calculations due to its dense connections. To tackle this issue, Mehta et al. [142] introduce **grouped linear transformations (GLTs)**, which incorporate the concept of group convolution to make the transformer block more lightweight. Facing the same shortcoming from the group convolutions (as presented in Section 2.1.3), the **hierarchical group transformation (HGT)** [143] aims to enhance the information flow between groups using a split layer and a skip connection operation. DeLighT [141] exploits GLTs to make feature dimensions wider and deeper, making it possible to use single-head attention instead of multi-head attention. This technique decreases the computation cost in the attention operation from $d_m N^2$ to $d_o N^2$, where d_m and d_o are the input dimension and output dimension, respectively.

2.3.2 Token Sparsing. **Vision transformer (ViT)** [44] is the earliest work that applied transformers to solve an image classification task. It first splits an image into several patches and flattens them so it can be passed in as an embedding sequence input to the transformer architecture. As the resolution of images in ImageNet is 224×224, their tokens require significantly more computation compared to other datasets with smaller resolutions, such as CIFAR-10 and CIFAR-100 (32×32).

To address this, T2T-ViT [237] observes that image splitting in transformers causes a loss of local relationships between tokens, since there is no overlap between the tokens. Hence, they employ soft unfolding to combine the surrounding spatial tokens into high-dimensional manifolds, enabling smaller MLP sizes and increasing memory efficiency.

An extensive study on transformers [147] demonstrates that transformers are robust to patch drops, with only a slight decrease in accuracy when patches suffer from distortion or occlusions. DynamicViT [158] integrates a prediction module between transformer blocks to mask the less significant tokens. The prediction module is a binary decision mask in the range (0,1) that measures

the importance of tokens. EViT [122] computes attentiveness scores from class tokens and other tokens and keeps top-K tokens, representing the highest positive correlation to the prediction. A-ViT [232] adaptively changes the number of tokens at different depths based on the complexity of the input image to reduce the inference time in ViT.

2.3.3 Lightweight Hybrid Models. Due to the long-range dependence property inherent in attention mechanisms, transformer networks outperform CNN in accuracy. However, a transformer network lacks strong inductive biases [38, 63, 133], making it difficult to train and requires extra data augmentation and heavy regularization to maintain performance [197]. However, CNN extracts features based on sliding windows, resulting in stronger inductive biases, which make models easier to train and have better generalizability. Interestingly, the aggregation of CNN and transformer networks [47, 175, 217, 220] produces versatile and powerful models. Since the hybrid models would have many parameters, DeiT [197] applies KD and achieves better accuracy with less latency than CNN under comparable parameters. To improve data efficiency and simplify model complexity, the student model, a ViT model, added a distillation token to provide insight into the inductive biases of a CNN-based teacher model. MobileViT [144] points out that transformer-based networks perform worse than CNN networks under similar parameters because they are still bulky. Mobile-ViT employs MobileNetV2 [167] as the CNN backbone to obtain inductive biases and replaces the MBconv block in MobileNetV2 with a MobileViT block with unfolding and folding operations, which can compute long-range dependencies like a transformer. Similarly, MobileFormer [27] devises a parallel structure consisting of CNNs and transformers to achieve feature fusion. Inductive bias and the ability to capture global features are incorporated via two-way cross-attention.

2.3.4 Discussion and Summary. Recent transformer models focus on lighter and more powerful architectures. This observation is apparent from Table 3, where many recent transformers, such as T2T-ViT [237] and DymViT-LVit [158], are shown to achieve higher accuracy with significantly fewer parameters and lower FLOPs than the original ViT and ResNet-based CNNs. Specifically, we split the discussion into three subsections with bold headings.

VIT & KD transformer. Inspired by Reference [85], several papers [22, 125, 197] apply KD to distill the inductive bias from the CNN-based teacher models to the transformer-based student models. For example, the design of DeiT-B [197] architecture integrates a CNN-based teacher model, a RegNetY-16G [157], and a transformer-based student model, ViT-B [44]. Results show that DeiT-B outperforms all the models in terms of Top-1 accuracy, achieving an accuracy of 84.5%. Despite their stronger abilities, transformer-based student models require a large network to maintain their performance, since they are harder to converge than CNN models [38].

VIT & CNN hybrid transformer. To overcome the shortcomings of the KD-based transformer models, the hybrid models [27, 38, 144] utilize both the convolution and transformer layers in the network. By doing so, they can obtain stronger inductive bias, leading to better convergence during training. Thus, hybrid models typically have fewer FLOPs and parameters. For example, Mobile-Former-96M [27] achieves the lowest FLOPs of 0.096 G while MobileViT-XS [144] has the lowest parameters, which is 2.3M. These hybrid models are extremely lightweight, but sometimes efficiency is achieved at the expense of accuracy, as we can observe from their performance in Table 2. For instance, MobileVit-XS has roughly half the total parameters of MobileViT-S, its counterpart, but its accuracy has significantly dropped by 3.6%. Another noteworthy observation shows that although Mobile-Former-96M achieves the lowest FLOPS, its parameter size was doubled, and accuracy is 2.0% lower compared to MobileVit-XS. This demonstrates that there is not always a correlation between FLOPs and total parameters and that lowering FLOPs appears to have a greater impact on accuracy than lowering parameters.

Table 3. Comparison of Lightweight Transformer Models on the ImageNet Dataset

Categories	Model	Image (Size)	Params. (M)	Throughput (image/s)	FLOPs(G)	ImageNet Top-1
CNN	ResNet50 [77]	224×224	25.5	–	4.13	76.2
	ResNet101 [77]	224×224	44.6	–	7.9	77.4
	ResNet152 [77]	224×224	60.2	–	11.0	78.3
	RegNetY-16GF [157]	224×224	84.0	334.7	–	82.9
ViT	ViT-B/16 [44]	384×384	86.6	85.9	17.6	77.9
	ViT-L/16 [44]	384×384	307.0	27.3	63.6	76.5
ViT & KD	DeiT-Ti [197]	224×224	5.0	2,536.5	–	72.2
	DeiT-Ti [197]	224×224	6.0	2,529.5	–	74.5
	DeiT-S [197]	224×224	22.0	936.2	4.6	81.2
	DeiT-B [197]	224×224	87.0	290.9	17.6	83.4
	DeiT-B [197]	384×384	87.0	85.8	17.6	**84.5**
ViT & Token Sparsing	T2T-ViT-14 [237]	224×224	21.5	–	5.2	81.5
	T2T-ViT-14 [237]	384×384	21.5	–	17.1	83.3
	T2T-ViT-19 [237]	224×224	39.2	–	8.9	81.9
	DymViT-LViT-S/0.5 [158]	224×224	26.9	–	3.7	82.0
	DymViT-LViT-M/0.7 [158]	224×224	57.1	–	8.5	83.8
	EViT-DeiT-S (k=0.5) [122]	224×224	22.0	4,385	3.0	79.5
	EViT-DeiT-S (k=0.7) [122]	224×224	22.0	**5,408**	2.3	78.5
	EViT-LViT-S (k=0.5) [122]	224×224	26.2	3,603	3.9	82.5
	EViT-LViT-S (k=0.7) [122]	224×224	26.2	2,954	4.7	83.0
	A-ViT-T [232]	224×224	5.0	3,400	0.8	71.0
	A-ViT-S [232]	224×224	22.0	1,100	3.6	78.6
ViT & CNN (Hybrid models)	**Mobile-Former-96M [27]**	224×224	4.6	–	**0.096**	72.8
	Mobile-Former-29[27]	224×224	11.4	–	0.294	77.9
	Mobile-Former-508M [27]	224×224	14.0	–	0.508	79.3
	MobileViT-XS [144]	224×224	**2.3**	–	0.7	74.8
	MobileViT-S [144]	224×224	5.6	–	–	78.4

We use **bold** to emphasize the models with the least parameters, highest throughput, lowest FLOPs, and best accuracy, with the corresponding values also underlined for enhanced readability.

VIT & Token sparsing transformer. Another series of efficient transformers [122, 147, 158, 232, 237] aims to prune the transformer structure efficiently via token sparsing. From the results, token sparsing-based models achieve a competitive accuracy with fewer parameters and FLOPs. It is worth noting that EViT-DeiT-S (k=0.7) [122] can reach the highest throughput, 5,408 images per second. Therefore, for a faster transformer-based model, such as accomplishing a real-time system, aggregating tokens into smaller amounts may provide a promising solution.

Due to their competitive accuracy and lightweight design [105, 135], lightweight transformer models are gaining popularity in a wide range of applications, such as edge AI and mobile AI; more details of efficient transformers can be found in References [74, 195].

3 FUNDAMENTAL METHODS IN MODEL COMPRESSION

In this section, we explore popular compression methods used in recent years and their improvements over time. These techniques encompass pruning [54, 76, 81, 92, 113], quantization [43, 49, 97], knowledge distillation [62, 85, 249], and neural architecture search [129, 215], which are widely adopted for designing efficient models. We further unveil a detailed exploration of each method, offering deeper insights that stem from their distinctive characteristics.
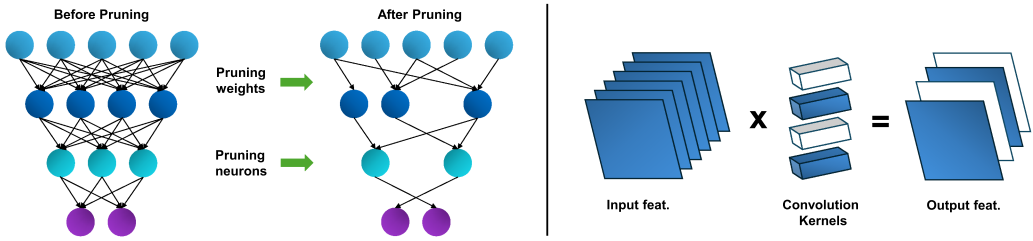
Fig. 4. Illustration of pruning methods: unstructured pruning (left) and structured pruning (right). Pruned components are shown in white color. Take note of the change in the pruned component's output dimensions.

3.1 Pruning

DL models frequently comprise numerous learnable parameters, requiring extensive training. Pruning methods aim to compress and expedite NNs by removing redundant weights. These pruning methods can be categorized as either unstructured or structured.

3.1.1 Unstructured Pruning. Unstructured pruning aims to identify and eliminate individual weights from the network, regardless of where they are located. This method imposes no restrictions or rules on weight trimming. Specifically, the nodes with the removed weights are not physically removed from the network; instead, the weights are set to zero. Since this operation results in numerous zero multiplications, models can be significantly compressed for faster inference. As illustrated in Figure 4 (left), unstructured pruning may cause the pruned network to have an irregular structure. Early works in pruning, such as Optimal Brain Damage [113] and Optimal Brain Surgeon [76], utilize second-order derivatives and Hessian matrices to assess the importance of weights in the network and subsequently prune them. While these methods demonstrate impressive performance, they demand substantial computational power.

To this end, Dong et al. [41] introduce a method that restricts the computation of second-order derivatives. This approach does not require the computation of the Hessian matrix over all parameters; instead, it focuses on specific layers of the model. Similarly, Frankle et al. [54] propose the lottery ticket hypothesis, where they attempt to find more manageable and pruned sub-networks while maintaining a performance comparable to the original network. In their approach, they prune the nodes, subsequently restoring the original pre-training initialization values of the untouched nodes, and repeat this cycle until a certain level of sparsity is achieved.

However, unstructured pruning can significantly reduce accuracy when weights are pruned during the training process before the network converges. Unfortunately, the pruned connections cannot be restored. To address this limitation, Guo et al. [67] introduce a splicing algorithm capable of recovering previously deleted connections that are discovered to be important at any point in time. Furthermore, Namhoon et al. [114] propose a single-shot network pruning approach in which they prune the network before the training begins. Instead of analyzing the model's final weights after training, they examine the response of the loss function to variance scaling during initialization. This innovative approach allows the network to be pruned just once before training, providing a more convenient and effective pruning method.

3.1.2 Structured Pruning. Structured pruning methods remove pruned components from a pre-trained network and preserve its regular structure, as shown in Figure 4 (right). Common structured pruning methods include filter pruning [79–81, 248] and channel pruning [83, 92, 153].

(1) Filter pruning. Most pruning approaches rely on the "smaller-norm-less-important" criterion, which involves pruning filters with lower norm values in the network [115, 231]. However,

Table 4. Comparison of Different Pruning Methods Using ResNet50 on the ImageNet Dataset

Type	Method (30%)	Baseline (%)	Pruned Acc. (%)	Pruned FLOPs (%)
-	ResNet50	76.15	–	-
Filter	SFP [80]	76.15	74.61 (−1.54)	41.8
	FPGM [81]	76.15	75.59 (−0.56)	42.2
	LFPC [79]	76.15	74.46 (−1.69)	60.8
	ASTER [248]	76.15	75.27 (−0.88)	**63.2**
Channel	CCP [153]	76.15	75.50 (−0.65)	48.8
	GFP [130]	76.79	**76.42 (−0.37)**	50.6
	SCP [106]	75.89	74.20 (−1.69)	54.3
	CATRO [92]	75.98	75.84 (−0.14)	45.8

The methods that achieve the highest percentage of pruned FLOPs are marked in **bold**.

He et al. [81] point out the limitations of this criterion-based approach. They propose a novel technique for calculating the Geometric Median of filters within the same layer. By doing so, they prune filters that make the most replaceable contribution instead of those with comparatively less contribution. Criterion-based pruning methods tend to reduce model capacity due to fixed pruning thresholds. To address this, He et al. [79] introduce learnable pruning thresholds for each layer using a differentiable criterion sampler, which can be updated during training. Additionally, Zhang et al. [248] propose an adaptive pruning threshold based on the sensitivity of the loss to the threshold value.

(2) Channel pruning. Channel pruning is another effective approach for reducing FLOPs and inference time, complementing filter pruning. He et al. [83] first implement channel pruning by focusing on eliminating redundant channels by evaluating the L1 norm. Peng et al. [153] take a different approach by using the Hessian matrix to model inter-channel dependencies and select channels using sequential quadratic programming. For more complex modules such as group convolutions and depthwise convolutions, Liu et al. [130] introduce a layer grouping mechanism to search for coupled channels automatically. The importance of these channels is calculated based on Fisher's information. CATRO [92] leverages feature space discrimination to assess the joint impact of multiple channels while consolidating the layer-by-layer impact of preserved channels.

3.1.3 Comparison of Pruning Methods. Table 4 displays the accuracy after pruning and the corresponding pruned FLOPs of the various structure pruning methods. While one might initially assume that the best-performing methods prune the highest number of FLOPs, in reality, we often perceive the "best" as those that effectively balance the tradeoff between pruned FLOPs and the associated drop in accuracy. For instance, while GFP attains the highest pruned accuracy, its reduction in FLOPs is limited to 50.6%. In contrast, ASTER removes the most FLOPs, yet its pruned accuracy falls short of being the best. In summary, filter and channel pruning methods can efficiently decrease the FLOPs while maintaining similar accuracy. We advocate choosing a pruning method that seamlessly integrates with the current network architecture, prioritizing ease of implementation. For example, if the network's feature map boasts over a thousand channels but only uses a few filters, then opting for channel pruning would be more beneficial.

3.2 Quantization

Pruning is an efficient way to compress the model. However, after pruning, the remaining weights, typically stored as full-precision 32-bit floating-point numbers (float32), still demand significant memory. To address this, quantization [64], a technique that allows parameters to be represented
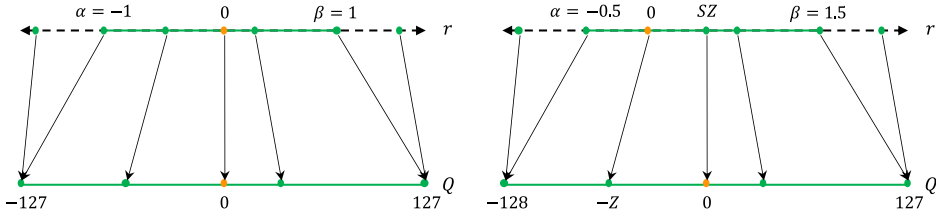
Fig. 5. Symmetric (left) and asymmetric (right) quantization representation [58]. Note that r represents the real value, S represents the real-valued scaling factor, and Z represents the integer zero point.

with reduced bit precision, becomes a desirable solution. Specifically, quantization maps weights and activations to a set of finite numbers through a calibration process that determines potential values using a symmetric or asymmetric representation. As depicted in Figure 5, both methods define a range $[\alpha, \beta]$, in symmetric quantization, $-\alpha = \beta$, whereas in the asymmetric quantization, $-\alpha \neq \beta$.

The calibration of this range, as outlined by Gholami et al. [58], falls into two categories: dynamic and static calibration. The first one is accurate but computationally demanding, as it computes $[\alpha, \beta]$ for each feature map. The latter is a computationally lighter alternative, because it calculates the range based on typical values after several iterations, albeit with less accuracy. Both dynamic and static calibration are pivotal for optimizing the quantization process.

Quantization theory has been applied to NN from various perspectives over time. For instance, Gupta et al. [70] introduce the use of fixed-point numbers during the model's training process to enhance the algorithm's noise tolerance. They also employ stochastic rounding as an alternative to the round-to-nearest strategy to counteract the adverse effects of fixed-point numbers. In another approach, Faghri et al. [49] introduce two adaptive quantization methods, **Adaptive Level Quantization (ALQ)** and **Adaptive Multiplier Quantization (AMQ)**, which update their compression method in parallel during training to quantize the gradients in data-parallel stochastic gradient descent adaptively. This adaptation aims to reduce communication costs between the processors. Last, Wang et al. [206] treat the quantization problem as a differentiable lookup operation. They jointly optimized both the network and the associated tables during training.

3.2.1 Half-precision and Mixed-precision Training. Mixed-precision training involves using lower-precision values while retaining full-precision values for crucial information [146]. For instance, in a notable series of works, HAWQ [43] implements an automatic approach based on the Hessian of the model to determine the optimal mixed-precision settings for weight values. Subsequently, the HAWQ-V2 model [42] introduces mixed-precision quantization for activation values. The HAWQ-V3 model [229] further improves it by focusing on integer-only quantization. Interestingly, Liu et al. [131] introduce a method that utilizes a linear combination of multiple low-bit vectors to approximate a full-precision vector. This approach achieves "mixed-precision training" with a single precision level by varying the number of vectors to approximate different weights.

3.2.2 Quantization Using Fewer Bits. In an early work by Banner et al. [9], the quantization of weights, activations, and most gradient streams in all layers of an NN is performed using 8-bit precision by replacing traditional batch-norm with ranged batch-norm layers. Another technique proposed by Wang et al. [207] allows matrix and convolutional operations to also be implemented using 8-bit numbers. Furthermore, there have also been methods that use ternary values to quantize an NN. In an important work done by Liu et al., TWN [128] manages to constrain weights to

Table 5. Comparison of Several Quantization Methods Using Different Levels of Precision to Compress a ResNet18 on the ImageNet Dataset

Method	Initial Accuracy. (%)	Quantized accuracy (%)	Precision
QIL [104]	70.2	70.1 (−0.1)	4-bit
[131]	69.8	61.7 (−8.1)	4-bit
LLT [206]	69.8	70.4 (+0.6)	4-bit
LLT [206]	69.8	69.5 (−0.3)	3-bit
HAWQ-V3 [229]	71.5	68.5 (−3.0)	MP
TWN [128]	65.4	61.8 (−3.6)	2-bit
TTQ [255]	69.6	66.6 (−3.0)	2-bit
XNOR-Net [159]	69.3	51.2 (−18.1)	1-bit
Least Squares [154]	69.6	63.4 (−6.2)	1-bit

+1, 0, and −1 values, achieving a 16× compression of the model. This idea is extended in TTQ [255], where the positive and negative weights use two different learnable scales, w_1 and w_2, resulting in possible values of $−w_1$, 0, and w_2.

More aggressive approaches have sought to reduce quantization levels further by implementing NN binarization. This approach uses binary values instead of floating-point or integer values for faster computations, lower memory usage, and reduced power consumption. Courbariaux et al.'s pioneering work [97] binarizes networks by restricting the weights and activations to either +1 or −1, determining the final values by evaluating the sign of the real values. Variations of this work include topologies such as XNOR-Net [159] and the Least Squares method [154], which introduce an additional activation layer after the binary convolutions.

3.2.3 Quantization Aware Training (QAT). In the early stages of quantization research, a prevalent approach was first to train an unquantized model, apply a quantization process, and then retrain or fine-tune the model to achieve an acceptable level of accuracy. This methodology, known as **Post-training Quantization (PTQ)**, proved to be an effective strategy for achieving significant compression, especially when the pre-trained model has ample representational capacity. The success of PTQ lies in its ability to balance compression gains and maintain satisfactory model accuracy, making it a pivotal technique in model optimization and deployment. However, quantization is a lossy process, which can lead to a significant drop in model accuracy. To address this issue, Jacob et al. [99] introduced QAT, a technique that computes inference-time quantization errors during the model training stage, allowing the model to become aware of these errors and make adjustments accordingly. This process simulates inference-time errors through a process known as FakeQuant.

Improvements to the core QAT technique have been explored by introducing learnable clipping scalars [33]. In a recent development, Sakr et al. [166] achieved state-of-the-art performance by identifying the MSE-minimizing clipping scalars and implementing 4-bit quantization.

3.2.4 Comparison of Quantization Methods. Table 5 compares the performance of quantization methods on the ImageNet dataset, emphasizing the tradeoff between compression and accuracy loss. Notably, binarized networks aiming for a 32× compression and speedup show significant accuracy drops. However, approaches with 4-bit quantization, except Reference [131], result in little loss of accuracy and can, therefore, be a good choice of precision for quantization. However, theoretical compression and speedup expectations may not align with actual results due to additional operations such as quantization and dequantization. This may explain why some works opt not to conduct an in-depth analysis of the quantized model size, although Reference [131] does provide
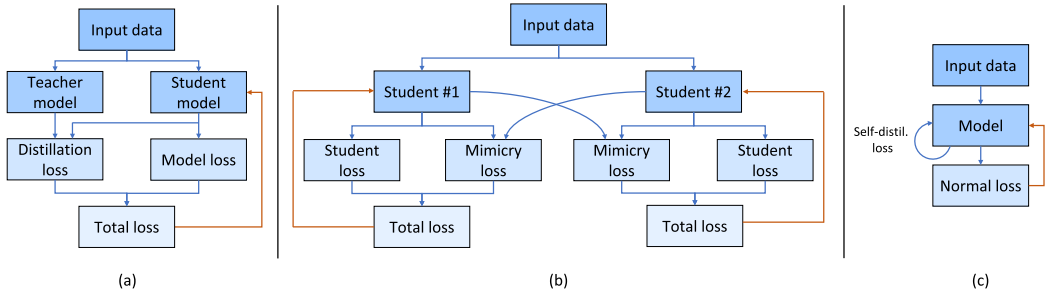
Fig. 6. (a) Offline distillation [85]. (b) Online distillation [249]. (c) Self-distillation [245]. We use orange lines to indicate the gradient update.

such an analysis and successfully achieves an approximately 8-fold reduction in the model size (42.56 MB to 5.37 MB).

3.3 Knowledge Distillation (KD)

KD is a model compression technique designed to transfer knowledge from a large network to a smaller one [62, 85]. Its simplest form is illustrated in Figure 6(a), where the larger model is referred to as the teacher and the smaller model as the student. In the approach proposed by Hinton et al. [85], the teacher model is initially trained to generate soft labels. Then, the training of the student model leverages ground-truth labels and the teacher's predictions on the same data. This combination enables the student to attain performance comparable to the teacher using fewer parameters.

KD algorithms can be categorized into three types: offline, online, and self-distillation, as illustrated in Figure 6. The key distinction lies in the teacher's definition and training strategy. For instance, in offline distillation, teacher and student training processes are performed sequentially, whereas in online distillation, the teacher can continue or initiate training alongside the student. However, in self-distillation, the student becomes its own teacher.

3.3.1 Offline Distillation. Most of the earlier KD works fall under the category of offline distillation. In offline distillation, a pre-trained teacher model is required, as seen in the case of the vanilla KD [85]. While offline distillation is relatively easy to implement, it comes with the unavoidable overhead of time and computational resources required to train a large teacher model first.

Various methods have been explored to enhance KD algorithms, including introducing alternative loss functions such as contrastive-based loss [196] and minimizing the maximum mean discrepancy between models [96]. Significant size disparities between teacher and student models can impact results, leading Zhao et al. [251] to redefine logit distillation by decoupling the influence of target and non-target classes. Lin et al. [125] address the semantic information gap in KD by dynamically distilling each pixel of the teacher features to all spatial locations of the student features, guided by a similarity measure from the transformer.

Recently, SimKD [19] proposed a straightforward distillation approach, reusing the teacher's classifier and aligning intermediate features with an L2 loss. SemCKD [20] involves student learning through feature embedding, preserving feature similarities in the intermediate layers of the teacher network.

3.3.2 Online Distillation. Offline distillation can be problematic when obtaining a pre-trained large teacher model is not feasible, rendering many of the previously mentioned methods unusable. Online distillation introduces an end-to-end training strategy that overcomes this limitation by

concurrently training the teacher and student networks, challenging the traditional concept of a "single large teacher" [66, 117, 249].

The **Deep Mutual Learning (DML)** algorithm, proposed in Reference [249], eliminates the need for a pre-trained teacher in the KD process, as depicted in Figure 6(b). Instead, this approach advocates simultaneous learning of a cohort of networks, with each network incorporating the predictions of the others in its loss functions. This change enables all networks in the cohort to benefit from each other's knowledge, even improving networks that are large enough to have acted as teachers in a conventional KD process. These large networks can enhance their results with knowledge distilled from other untrained, smaller networks. Further refinements of this approach have been made in References [66, 117]. Online distillation techniques can also incorporate adversarial concepts. Zhang et al. [242] propose an adversarial co-distillation approach that employs **Generative Adversarial Networks (GANs)** to explore "divergent examples" and enhance knowledge transfer.

Furthermore, online distillation has demonstrated notable efficacy in scenarios requiring generating pseudo labels for data. The widely adopted mean teacher framework [194] introduces the concept of employing two identical models; specifically, the teacher model has the same structure as the student model. The primary idea involves updating the teacher's weights through an **exponential moving average (EMA)** of the student's weights. In various unsupervised contexts [40, 236], this principle is leveraged to create pseudo labels for training the student via a supervised loss. Notably, each prediction made by the teacher model can be viewed as an ensemble incorporating the current and past iterations of the student model, rendering it inherently more robust and stable.

3.3.3 Self-distillation. As depicted in Figure 6(c), self-distillation techniques involve the process of KD, where a model distills knowledge from itself. In this scenario, during the training process, a single instance of the model simultaneously acts as both the teacher and student. Strategies in this distillation approach encompass using the same model saved at different epochs [224] and leveraging various model layers for self-instruction [87, 238].

Zhang et al. [245] pioneered self-distillation from deeper to shallower layers of the model. Their innovation improves results and reduces training time by eliminating the need for additional networks. Similarly, Hou et al. [87] harness knowledge transfer through attention maps from deeper layers. Yang et al. [224] use the weights of previous iterations for knowledge distillation instead of using deeper layers of the model. Kim et al. [107] elevate self-distillation with a sophisticated progressive framework, incorporating adaptive gradient rescaling for hard example mining.

In an important study, Yuan et al. [238] challenge the foundations of conventional KD by introducing the **Teacher-free KD (Tf-KD)**. They explore the intricate relationship between KD and **Label Smoothing Regularization (LSR)** techniques and suggest employing self-training or manually designed regularization terms for improving the student model's accuracy when faced with the difficulty of a powerful teacher model. Additionally, self-distillation methods have successfully been applied to domain adaptation tasks [179, 233].

3.3.4 Comparison of KD Methods. Table 6 compares several distillation methods and analyzes their respective outcomes on the CIFAR-100 dataset. These findings challenge the perception that offline distillation methods are outdated and too simplistic. For example, SimKD recently achieved state-of-the-art performance with a ResNet32 as the teacher and a ResNet8 as the student. Additionally, our analysis demonstrates the efficacy of online distillation, showcasing instances where a teacher can improve its own performance despite instructing a student with significantly lower accuracy. Notably, the WRN-28-10 achieves a 0.27% (78.69% to 78.96%) improvement even when paired with a ResNet32, which initially achieves nearly 10% (78.69% to 68.99%) less

Table 6. KD Methods Evaluated on the CIFAR-100 Dataset

Methodology	Algorithm	Teacher (baseline)	Student (baseline)	Improved Accuracy
Offline distillation	SimKD [19]	ResNet32 (79.42)	ResNet8 (73.09)	78.08 (4.99 ↑)
	SemCKD [20]	ResNet32 (79.42)	ResNet8 (73.09)	76.23 (3.14 ↑)
	SRRL [225]	ResNet32 (79.42)	ResNet8 (73.09)	75.39 (2.30 ↑)
	SemCKD [20]	ResNet32 (79.42)	WRN-40-2 (76.35)	79.29 (2.94 ↑)
Online distillation	DML [249]	WRN-28-10 (78.69)	WRN-28-10 (78.69)	80.28, 80.08 (1.39 ↑)
	DML [249]	WRN-28-10 (78.69)	ResNet32 (68.99)	78.96, 70.73 (1.74 ↑)
	FFSD [117]	ResNet56 (71.55)	ResNet32 (69.96)	75.78, 74.85 (4.90 ↑)
	KDCL [66]	WRN-16-2 (72.20)	ResNet32 (69.90)	75.50, 74.30 (4.40 ↑)
Self-distillation	SD [224]	–	ResNet32 (68.39)	71.29 (2.90↑)
	Tf-KD [238]	–	ResNet18 (75.87)	77.10 (1.23↑)
	PS-KD [107]	–	ResNet18 (75.82)	79.18 (3.36↑)
	Tf-KD [238]	–	ShuffleNetV2 (70.34)	72.23 (1.89↑)

↑ indicates an improvement over the baseline. Note: The pair of accuracies in the online distillation methods represent the teacher and student models' performances after distillation.

accuracy. Furthermore, self-distillation emerges as a promising strategy, necessitating only one model, exemplified by a ResNet18 achieving 3.36% gains through the PS-KD method, albeit not surpassing the improvements seen in other methods. To address this limitation, it is advisable to complement self-distillation with other forms of distillation or compression methods for enhanced performance. Ultimately, a comparison between methodologies is hard, as performance heavily depends on implementation details. Therefore, we advocate for adopting a strategy that is easier to implement and aligns most logically with the ongoing development objectives.

3.4 Neural Architecture Search (NAS)

Even if DL techniques excel in numerous tasks, it is true that they often depend heavily on human expertise to find the best tradeoff between performance and complexity. Optimizing a model can be exceptionally challenging due to a multitude of choices involving hyperparameters, network layers, hardware devices, and so on.

In response to this challenge, **Automated Machine Learning (AutoML)**, which aims to automatically build ML systems without much requirement for ML expertise and human intervention, is being extensively studied [78]. Several mature tools exist for AutoML applications, such as Auto-WEKA [109] and Auto-sklearn [51]. In this article, our primary focus is NAS, a crucial section of AutoML. The fundamental concepts of NAS are outlined as follows:

— **Search Space:** The search space encompasses the possible combinations of hyperparameters, including kernel size, channel size, convolution stride, depth, and more. A larger search space that covers a wider range of possibilities increases the likelihood of discovering a highly accurate model. However, a vast search space can lead to longer search times.

— **Search Algorithm:** This refers to the algorithm used to find the optimal combination within the search space. Common strategies include random search, grid search, **reinforcement learning (RL)** [189, 256], **evolutionary algorithms (EA)** [161, 223], and gradient optimization [129, 215]. An efficient search strategy can significantly reduce search time, especially in extensive search spaces.

— **Performance Evaluation Strategy:** This defines the criteria for selecting the neural architecture that maximizes specific performance metrics among all the models generated through NAS. Performance metrics, such as Top-1 or Top-5 scores for classification and
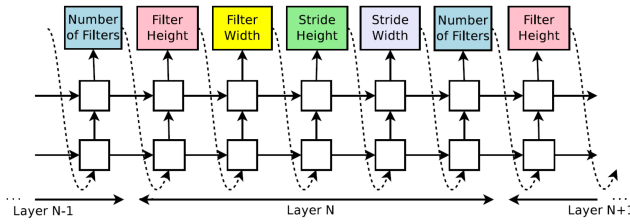
Fig. 7. NAS with RL [256].

average precision (AP) or F1 scores for object detection, reflect the suitability of the hyperparameter combinations for the given task.

In this section, we explore various approaches in the field of NAS, including RL-based NAS, EA-based NAS, Gradient-based NAS, and other related works, all based on different search algorithms.

3.4.1 RL-based NAS. In this pioneering work of adopting RL for NAS, Zoph et al. [256] utilize a **recurrent neural network (RNN)** controller (called an agent) to generate candidate hyperparameters for constructing child networks (environments). The child network then receives a score (reward) based on metrics such as accuracy and AP. The RNN controller updates itself according to the reward and refines the hyperparameters for the child network iteratively. A detailed process is illustrated in Figure 7. Moving forward, MnasNet [189] considers latency and employs RL to identify Pareto-optimal solutions that balance latency and performance. This approach also introduces a factorized hierarchical search space, which organizes the CNN into predefined blocks and explores different connections and operations within each block.

3.4.2 EA-based NAS. To enhance model performance, Real et al. [161] introduce an EA-based approach for NAS. This method continuously evolves model architectures. The evolution process begins with workers generating an initial set of models, forming what is known as a population. During the evolution step, two models are randomly selected from the population, and their accuracy on the validation set is evaluated. The weaker-performing model is removed from the population, while the better model becomes the parent model. In the mutation step, the parent model is duplicated, producing two identical copies. One of these copies is reintroduced into the population, while the other undergoes mutation to create a new model, referred to as the child model. Subsequently, the workers train and assess the child model's performance before adding it back to the population. This process is iteratively repeated, resulting in increasingly improved models within the population.

However, a random search approach within a large population can be highly inefficient when dealing with a vast search space. To address this concern, Sun et al. [182] develop an encoding mechanism that maps CNN features to numerical values. This enables the acceleration of the evolutionary process by using a CNN architecture as an input to the Random Forest. More recently, Xue et al. [223] proposed a queue mechanism to reduce the population and incorporate crossover and mutation operators to enhance the diversity of child networks.

3.4.3 Gradient-based NAS. The core concept of gradient-based NAS involves the transformation of a discrete search space into a continuous one, enabling the application of gradient descent techniques to discover optimal model architectures automatically. Inferring latency after each training is inefficient for the proposed NAS network, especially for research institutes with limited resources. Additionally, using gradient-based NAS methods is deemed more appropriate when formulating hardware-aware NAS approaches.
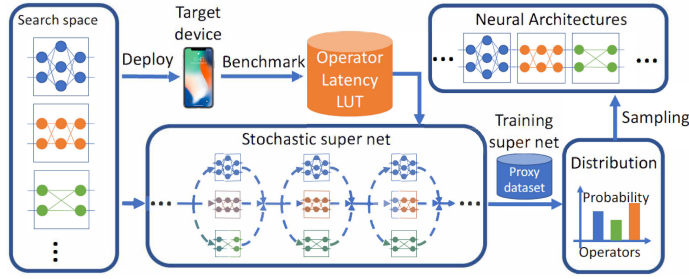
Fig. 8. The DNAS pipeline in FBNet [215].

DARTS [129] presents an efficient architecture search algorithm based on gradient descent that avoids black-box search problems. It converts structural parameters from discrete to continuous, making them differentiable. As a result, DARTS provides accurate, efficient, and differentiable NAS. Inspired by works such as MnasNet [189], DARTS [129], and NetAdaptV1 [227], FBNet [215] is a hardware-aware NAS breakthrough discovered through the **differentiable NAS (DNAS)** pipeline, depicted in Figure 8. In FBNet, nine distinct blocks are designed within the layer, and 22 layers are utilized to construct a stochastic supernetwork, which is optimized using **stochastic gradient descent (SGD)**. Additionally, FBNet devises a layer-wise search space, enabling each layer to select a different block. Furthermore, to reduce the layer-wise search space with lower latency, a latency lookup table is employed, and a latency-aware loss term is incorporated into the overall loss function, given by:

$$L(a, w_a) = CE(a, w_a) \cdot \alpha \log(LAT(a))^\beta, \tag{3}$$

where a and w_a denote the network architecture and network parameters for a specific device, while CE represents the cross-entropy loss. LAT stands for the latency of the architecture on the target device, which is determined using a lookup table. The parameters α and β serve as the magnitude of the overall loss function and the latency term, respectively. For further details and related work on FBNet, please refer to References [37, 204].

3.4.4 Other NAS-related Works. Numerous other NAS algorithms have been proposed. One example is the Symbolic DNN-Tuner [53], which introduces an automatic software system for determining optimal tuning actions following each network training session using probabilistic symbolic rules. The system comprises a module for data processing, search space exploration, and Bayesian optimization. The controller module manages the training process and decides the tuning actions. Besides finding the best combination from a vast search space, testing the proposed combination network is also time-consuming. Measuring the latency of the entire model on the target device each time can be highly inefficient.

To address this issue, NetAdaptV1 [227] employs an adaptive algorithm that considers energy consumption and memory usage, enabling it to respond more realistically to hardware constraints. The approach involves the creation of a layer-wise lookup table, as shown in Figure 9, simplifying the search complexity for a pre-trained network. In this setup, the latency of each layer is premeasured, and a lookup table is constructed to record latency based on the layer's structure. For instance, as illustrated in Figure 9, Layer 1 consists of 3 channels with 4 filters and a measured latency of 6 ms, and Layer 2 consists of 4 channels with 6 filters and a measured latency of 4 ms. The total latency is calculated as the sum of the latency for each layer, resulting in a total latency of 10 ms (6 + 4).

Moving forward, NetAdaptV2 [228] introduces **Channel-Level Bypass Connections (CBCs)**, which combine depth and layer width in the original search space to enhance the efficiency of both
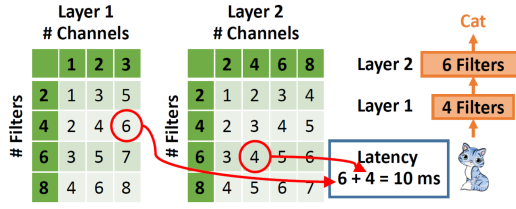
Fig. 9. Layer-wise lookup table [227].

training and testing. Moreover, Abdelfattah et al. [2] leverage pruning-at-initialization [114] and incorporate six zero-cost proxies for NAS proposal scoring. This innovative approach requires only a single minibatch of data and a single forward/backward propagation pass instead of full training, resulting in a more efficient NAS process.

3.5 Discussion and Summary

This section encapsulates a summary of the preceding discussion on model compression. Additionally, it provides valuable practical tips and guidance, aiming to offer actionable insights for effective implementation and application in relevant contexts.

Pruning. Although unstructured pruning methods [54, 113] have made significant strides in parameter reduction, their irregular structures frequently pose compatibility issues with hardware accelerators. Therefore, structure pruning [80, 81, 92] has emerged as a preferable alternative, primarily due to its regular structure. Notably, modern DL frameworks, such as PyTorch and TensorFlow, have integrated built-in functionalities that facilitate the seamless implementation of structure pruning. This streamlined integration enhances the ease and efficiency with which structure pruning techniques can be applied.

Quantization. When considering quantization, the choice of technique depends on the hardware environment where the model will be deployed. Hardware specifications play a critical role, turning quantization from an optional optimization into an imperative requirement. For instance, specific MCUs or edge TPUs exclusively support integer operations, making full integer quantization essential for model implementation. **TensorFlow Lite (TF-Lite)** [61] effectively addresses this need, reducing the model size by up to four times and significantly accelerating inference by more than three times. In hardware with low-power CPUs, an 8-bit integer quantization strategy is often recommended, as CPUs exhibit exceptional computational efficiency when handling integer operations instead of floating-point values. Notably, when using 16-bit float quantization, values are subsequently de-quantized back to 32-bit float representations during execution on the CPU. For a deeper analysis of hardware support for quantization and facilitating libraries, see Reference [121].

Knowledge distillation. KD techniques have significantly enhanced NNs by leveraging insights from other models. In practice, the offline KD process [85] can be effectively utilized when training a large model is viable. However, online distillation stands forth as a promising solution. For example, the DML process [249] has shown remarkable results without necessitating a pre-trained teacher model, making it adaptable to multi-GPU training with several small models. In situations characterized by a scarcity of labeled data or noisy labels, the mean teacher framework has emerged as a valuable and effective solution. Moreover, self-distillation and ongoing advancements in KD [125, 251] open numerous possibilities for exploration and offer different options for the definition of the teacher and student networks.

NAS. While both RL-based NAS [189] and EA-based NAS [182] have demonstrated their capacity to achieve impressive accuracy, it is important to note that their training demands extensive

resources and time, often spanning days or weeks and involving hundreds of GPUs. This resource-intensive nature has contributed to a relative decline in the number of studies in these areas. Therefore, when confronted with GPU limitations, gradient-based algorithms such as DARTS [129] and FBNet [215], which introduce continuity into the search space, can be considered. This approach significantly reduces the training time. Alternative options include approaches like "once for all" NAS [13], which tailor the extensive network into subnetworks optimized for different target devices. However, if ample computational resources are at hand, then RL-based and EA-based NAS methods are viable options, and they also offer superior performance compared to gradient-based NAS [162]. Additionally, when memory footprint, energy consumption, and latency are key considerations, the hardware-aware NAS concepts introduced by studies such as FBNet [215], NetAdapt [227], and NetAdaptV2 [228] may be particularly relevant.

Conclusion. In conclusion, model compression approaches have their strengths and limitations. Quantization is a relatively simple but proven effective compression technique in many cases. It is essential to first match the selected quantization approach with the specific hardware requirements for floating-point or integer values. In scenarios where hardware constraints permit, starting with a 16-bit float quantization is often a prudent initial step. If there is a need for more substantial model compression, then two viable options emerge. First, model pruning offers an effective solution, substantially reducing redundant network parameters while preserving performance integrity. This is particularly valuable when working with resource-constrained environments. Second, the KD framework proves advantageous, especially in scenarios with ample unlabeled data, as often encountered in applications like autonomous driving. The mean teacher structure, in particular, is a valuable tool for generating pseudo labels from unlabeled data, effectively incorporating this additional information into training and enhancing overall model performance. Finally, NAS can also be considered, particularly for tasks where it excels the most, such as image classification, where it can potentially discover optimal network architectures tailored to specific requirements. The choice among these approaches should be guided by the specific demands of the task and the available computational resources.

4 HARDWARE ACCELERATION OF DEEP LEARNING MODELS

With the advancements in GPUs, DL has risen to the forefront of artificial intelligence technology. DL models, such as CNNs, are computationally intensive. Hence, hardware acceleration is becoming imperative to render DL applications feasible and practical. In this section, we present an overview of prominent hardware accelerators of DL models. We then introduce typical dataflow and data locality optimization techniques, as well as widely adopted DL libraries. Finally, we discuss algorithms that employ a co-design approach for software/hardware deployment.

4.1 Hardware Architectures

Hardware accelerators for DL models encompass a range of options, including GPUs and CPUs based on temporal architecture, as well as FPGAs and ASICs rooted in spatial architecture. The basic components of a hardware accelerator are an **arithmetic logic unit (ALU)**, a control unit, and a local memory unit (cache unit). In the temporal architecture, the control and local memory units are centralized, and the **processing elements (PEs)** only contain the ALUs. Data is accessed sequentially from centralized memory to PEs, with no interactions between the PEs [16]. In contrast, spatial architecture entails PEs equipped with control units, ALUs, and local memory (register file). This allows independent data processing and direct communication between PEs.

4.1.1 Temporal Architecture. Temporal architectures are often adopted in general-purpose platforms, such as CPUs and GPUs, which are optimized for sequential tasks and parallel tasks, respectively.

Central processing unit (CPU). CPUs process input data into usable information output, executing calculations sequentially through serial computing. A recent CPU-based acceleration technique, SLIDE [17], which leverages C++ OpenMP to combine intelligent randomized algorithms with multi-core parallelism and workload optimization, demonstrates that employing smart algorithms on a CPU can potentially achieve better speed than using an NVIDIA-V100 GPU.

Graphics processing unit (GPU). GPUs are designed for parallel computation. Their architecture may consist of thousands of cores. Hence, GPUs excel at parallel computing, enabling them to process multiple instructions simultaneously, making them highly efficient for tasks that involve simple and repetitive computations. Given that DL models often entail extensive matrix addition and multiplication operations, GPUs have emerged as the primary accelerators for the development of DL. Their parallel processing capabilities make them instrumental in accelerating DL tasks.

4.1.2 Spatial Architecture. By utilizing PEs, spatial architectures often seen in FPGAs and **application-specific integrated circuits (ASICs)**, the necessity for repeated and redundant access to external memory is reduced, leading to lower energy consumption.

FPGAs. FGPAs consist of programmable logic blocks with logic gates capable of performing computations. Reprogrammable by nature, they can accelerate various DL structures effectively and better support pruning methods. Additionally, FPGAs can directly implement algorithms without any decoding and interpretation process. To enhance AI applications using FPGAs, Qi et al. [155] emphasize key concepts of parallel computing and demonstrate how these concepts can be implemented in FPGAs. Roggen et al. [163] successfully implement **digital signal processing (DSP)** algorithms, such as filter finite impulse response filters on FPGA platforms, thereby improving support for wearable computing. For more references on FPGA AI applications, consult References [148, 170].

ASICs. ASICs, customized for specific electronic systems, outperform FPGAs with superior speed, lower power consumption, and higher throughput. TPUs, prominent ASICs tailored for AI applications [103], excel in efficiently executing matrix operations, a pivotal capability advantageous in deep learning computations with prevalent expansive matrix multiplications. In a recent development, the newly introduced TPU-v3 can connect 1,024 TPU chips through a 2-D torus network [111]. This innovation enhances parallelism and enables execution on more TPU-v3 accelerator cores through spatial partitioning and weight update-sharing mechanisms. The supercomputer TPU-v4 [102] further elevates the capabilities by increasing the number of TPU chips to 4,096. TPU-v4 also introduces **optical circuit switches (OCSes)** that dynamically restructure their interconnection topology to improve scalability, accessibility, and utilization. As a result, TPU-v4 offers a 2.7 times improvement in performance/watt and a tenfold increase in speed compared to TPU-v3.

4.1.3 Discussion of CNN Accelerators. CPUs are generally not well-suited for training and inference of typical DL models due to low FLOPs performance. GPUs, which can support parallel computation with thousands of cores, excel in parallel computing and are widely adopted in various AI applications. However, GPUs are known for their high power consumption, rendering them unsuitable for edge devices and IoT applications. However, FPGAs and ASICs offer more energy-efficient acceleration options for edge AI applications. The choice between FPGAs and ASICs often depends on the specific requirements. FPGAs are preferred for AI products that require rapid development or are produced in small batches. ASICs are more suitable for AI products that undergo mass production, especially highly mature or customized ones. For projects with ample budget, TPUs can be the top choice. TPUs boast exceptional computational power,

making them ideal for handling extensive models with large batch sizes, such as the GPT-4 [150] and LLaMA [198], significantly reducing training and inference times.

4.2 Dataflow and the Data Locality Optimization

The computational complexity and data storage demands of CNNs pose significant challenges to computational performance and energy efficiency. These challenges are particularly pronounced in smaller devices with limited memory, including constrained on-chip buffers (SRAM) and off-chip memory (DRAM). To address these issues, optimizing dataflow is crucial for enhancing memory and energy efficiency. The dataflow process in deep models generally consists of three main steps. First, DL models are stored in off-chip memory, often referred to as external memory. Second, when convolution kernels are required, they are fetched from on-chip buffers. Finally, PEs are employed to execute the MACs.

4.2.1 Dataflow Types. Hardware accelerators of DL models have different types of dataflow based on their applications and can be categorized into pipeline-like dataflow [116, 127], DaDianNao-like dataflow [28, 136], Systolic-array-like dataflow [103, 212, 241], and streaming-like dataflow [45, 65].

Pipeline-like dataflow. In this dataflow, the input pixels (the pixels of the feature map) are passed on to individual PEs, and the model's weights (representing model parameters) are fixed on each PE. Notably, the partial sum is then forwarded to the subsequent PE. This approach offers substantial parallelism, facilitating the concurrent processing of data by multiple stages, thereby enhancing computational efficiency. However, tasks are executed sequentially, with each stage dependent on the completion of the previous one, potentially resulting in increased latency.

DaDianNao-like dataflow. In this dataflow, each PE can function like a neuron, processing input pixels in a way akin to an NN. Specifically, input pixels are routed to each PE, and the model's weights are embedded within each PE. The computed partial sums are then aggregated using an adder tree. This type of dataflow can accommodate different kernel sizes, making it capable of handling intricate and irregular model structures. However, this dataflow approach is energy-intensive and demands substantial hardware resources due to the model's complexity.

Systolic-array-like dataflow. This dataflow sequentially conveys input pixels and weights into the PEs, with PEs cascaded to enhance computational efficiency. Subsequently, an adder tree is employed to aggregate the partial sums. This dataflow approach optimizes the utilization of hardware resources, improves overall hardware efficiency, and mitigates timing issues in large designs. However, finding an appropriate mapping for CNNs onto a systolic array can be challenging.

Streaming-like dataflow. In this dataflow, input pixels are continuously sent to the following PE without pausing or needing intermediate storage, with weights being fixed on each PE. Subsequently, the adder tree accumulates the partial sums. This dataflow is particularly suitable for streaming data, such as audio and video processing, due to its high throughput and low latency. Nonetheless, applications requiring complex operations between stages or that rely on previous results may require additional processing and design. Figure 10 compares the types of dataflow.

4.2.2 Data Locality Optimization. CNNs deliver exceptional performance characterized by high throughput and energy consumption. However, their performance can be restricted by limited on-chip memory. Therefore, an effective locality optimization mechanism is essential. Data locality optimization focuses on devising a dataflow schedule that maximizes data reuse utilization and minimizes data movement. A prevalent approach involves applying loop transformation techniques, such as loop unrolling, loop tiling, and loop interchange, to optimize NN

Category	Input	Weight	PartialSum	Benefits	Limitations
Pipeline-like	Broadcast the pixel to each PE	Fixed on each PE	Flow into next PE	Flexible for various kernel size	Higher latency
Systolic-Array-like	Sequentially flow into PE	Sequentially flow into PE	1. Added up by the adder tree 2. Flow into vertical PE	Mitigate timing issues in synthesizing a large design	Not easy to find a feasible mapping
DaDianNao-like	Broadcast the pixel to each PE	Fixed on each PE	Added up by the adder tree	Flexible for various kernel size	Without kernel level parallelization
Streaming-like	Flow into next PE	Fixed on each PE	Added up by the adder tree	1. Minimize data movement 2. Achieve optimal energy efficiency	Not flexible for various kernel size

Fig. 10. A comparison of dataflow types [90]. PE stands for processing element.

deployment. These techniques help maximize hardware utilization and minimize memory traffic, addressing the limitations of on-chip memory constraints.

Loop unrolling [12, 95] is a method that involves expanding loop iterations into multiple sequential instructions. This technique significantly reduces the number of loop iterations in the CNN, resulting in faster CNN operations and improved hardware utilization through increased parallelization. However, it is important to note that loop unrolling may lead to code bloat, increased memory usage, and higher storage requirements, especially for larger CNN models.

Loop tilling [156, 176, 240] involves partitioning the input data into several blocks to enable parallel computations for CNN acceleration. For example, an original input data of size $224 \times 224 \times 3$ can be divided into smaller blocks of size $112 \times 112 \times 3$. These smaller blocks are processed sequentially to mitigate buffer loading and memory constraints. This technique effectively adapts to limited on-chip memory and significantly enhances cache locality. However, for modern accelerators, such as GPUs, where memory access patterns are already optimized for high throughput, loop tilling may add extra complexity without appreciable gains in performance.

Loop interchange [145, 222] involves changing the order of loops within a nested loop with the aim of improving data locality and extracting parallelism. Specifically, the order of the loops is optimized to allow each iteration of the outermost loop to utilize the same cache line, hence reducing memory access. Loop interchange can also accelerate CNN models by increasing the use of operators such as addition and multiplication. Notably, some algorithms have complex intrinsic properties and special meanings in their loop orders. Therefore, altering the loop order may yield meaningless results and reduce performance.

In this section, we introduce typical types of dataflow and provide an overview of various mechanisms for data locality optimization. More in-depth details can be found in References [56, 213].

4.3 Deep Learning Libraries

To facilitate the deployment of a DL model, it is also essential to use DL libraries that provide high-level APIs to simplify the implementation, design, and training of complex NNs. We introduce several popular DL libraries supporting GPU acceleration and the auto gradient system.

TensorFlow [1] supports static and dynamic graphs, allowing users to select the most suitable mode. With this flexibility, TensorFlow supports the research and development of custom DL models. Additionally, TensorFlow provides extensive APIs for DL model implementation. For instance, a TensorFlow model can be converted into a **TensorFlow-Lite (TF-Lite)** [39] model, a smaller, more efficient ML model format that can be run on mobile and edge devices.

PyTorch [152] is a framework renowned for its remarkable capacity to facilitate the creation of intricate models and the fine-tuning of NNs down to the minute details, making it a favored choice within the research community. Its simplicity, user-friendliness, and intuitiveness made it a go-to tool for prototyping DL models. However, there are certain deployment-related limitations with its API, which might restrict its application in certain real-life scenarios.

MXNet [24] is a library that provides optimized building blocks for implementing CNNs. It is specially tailored for Intel processors, offering vectorized and threaded support for CNNs on Intel CPUs and GPUs. Moreover, the MXNet framework provides interfaces in multiple languages, including Python, Scala, Java, Clojure, and R, making it convenient for cross-domain DL developers.

NVIDIA has been at the forefront of GPU hardware and software optimization for DL. cuDNN [30] is a highly optimized library specifically designed for DL networks, providing acceleration for DNN-related tasks. In addition to cuDNN, NVIDIA offers a range of DL libraries included in CUDA-X [149]. TensorRT [201], another NVIDIA library, optimizes inference on NVIDIA GPUs by applying layer and tensor fusion, kernel auto-tuning, and dynamic tensor memory optimizations.

Each DL library has unique strengths and caters to specific use cases, allowing practitioners to choose one that best suits their projects. To address the interoperability challenges between DL libraries, Microsoft and Facebook introduced **Open Neural Network Exchange (ONNX)** [52], an open standard for machine learning interoperability. With ONNX, models created in different libraries can be easily shared and executed. For instance, a PyTorch model can be run on an Android device by converting it into TensorFlow format, eliminating the need for model retraining.

4.4 Co-design of Hardware Architecture

In DL, acceleration solutions relying solely on software techniques are primarily limited by their dependence on the intrinsic capabilities of general-purpose processors, potentially struggling to exploit specialized hardware features designed for specific DL tasks fully. Conversely, hardware-only solutions may face limitations in flexibility and adaptability, as dedicated hardware is often tailored for specific tasks or architectures, making updates or adaptations to new DL models challenging without hardware modifications. This underscores the value of co-designing a hardware and software approach for resource-constrained environments, employing a holistic optimization strategy. This approach includes refining the DL algorithm, optimizing and compressing the model, efficient memory management, software kernel implementation, and hardware architecture design. This section discusses solutions that adopt a holistic approach to address challenges related to irregular memory accesses, enhance the handling of sparsity resulting from compression methods, and explore improved solutions within NAS algorithms.

In Section 3, we emphasize that many NN connections can be pruned effectively without substantial accuracy loss. However, in such models, only a subset of the NN's weights are active, and their locations are irregular or non-contiguous. Efficiently accessing these weights, especially when using hardware accelerators such as GPUs or TPUs, can be challenging due to the irregularity of weight locations. To tackle this issue, in earlier methods, like Cambricon-X [246], MAC operations utilize zero-weight connections and access required weights using sparse indices. However, irregular nonzero weight distribution caused issues such as indexing overhead, PE imbalances, and inefficient memory access. Later advancements, as seen in Cambricon-S [253], improve efficiency by enforcing regularity in filter sparsity through software/hardware integration.

Sparse-YOLO [211] introduces a dedicated sparse convolution unit tailored to handle quantized values and sparsity resulting from unstructured pruning techniques. Cho et al. [32] propose an acceleration technique for a quantized binary NN. This approach utilizes an array of PEs, with each PE responsible for computing the output of a specific feature map, implementing inter-feature map parallelism. Moreover, optimizing the storage of sparse weights post-pruning has been explored. Han et al. [75] show that these sparse weights can be compressed, reducing memory access bandwidth by around 20%–30%. SCNN [151] processes convolutional layers in their compressed format using an input stationary dataflow. This involves transmitting compressed weights and activations to a multiplier array, followed by a scatter network to add the scattered partial sums.
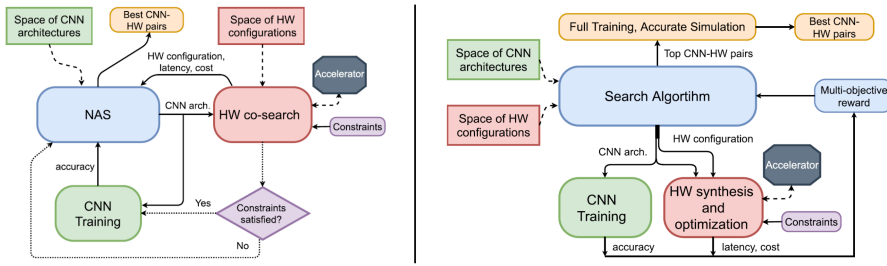
Fig. 11. Two different approaches for implementing NAS and hardware co-design [169].

In the NAS field, apart from the previously discussed hardware-aware NAS approaches that tailor models for specific hardware platforms, there are also co-designed solutions that initially remain hardware-agnostic. These co-designed systems seamlessly integrate hardware optimization within the NAS process, ensuring simultaneous hardware and DNN model optimization. Hardware settings can be explored in conjunction with DNN architectures using the same algorithm [34, 119, 254] or through an external search algorithm [126, 169].

As shown in Figure 11(a), the most direct approach for co-searching hardware and software settings involves creating CNN and accelerator pairs and evaluating the final model's performance. One can opt to train the CNN each time a new pair is tested or follow the approach of Chen et al. [25], where a supernet is employed to directly generate the weights of a DDN, and accuracy is assessed in a single testing run of the model. Figure 11(b) illustrates an alternative strategy employed by Lin et al. [126], where a hardware optimization algorithm takes a candidate CNN as input and optimizes the hardware accelerator to achieve specific objectives. The network is then trained and evaluated only if a viable hardware configuration is found. If no suitable hardware setting is identified, then the network remains untrained until a viable configuration is found. This strategy allows for the avoidance of training the CNN, which is the most complex phase of the co-design process.

In summary, the co-design of algorithms significantly improves compression and computational efficiency. However, these methods are inherently non-trivial and require in-depth exploration of software and hardware techniques.

5 CHALLENGE AND FUTURE WORK

In this survey, we explore the sophisticated domain of lightweight models, compression methods, and hardware acceleration, showcasing their advanced technological capabilities applicable across a broad spectrum of general applications. Nonetheless, deploying these models in resource-constrained environments continues to present substantial challenges. This section is dedicated to unveiling novel techniques in TinyML and LLMs for accelerating and applying DL models, focusing on unresolved issues that warrant further investigation.

5.1 TinyML

TinyML is an emerging technology that enables DL algorithms to run on ultra-low-end IoT devices that consume less than 1 mW of power. However, the extremely constrained hardware environment makes it challenging to design and develop a TinyML model. Low-end IoT devices predominantly employ MCUs due to their cost efficiency compared to CPUs and GPUs. However, MCU libraries, such as CMSIS-NN [112] and TinyEngine [124], are often platform-dependent, unlike GPU libraries such as PyTorch and TensorFlow, which offer cross-platform support. Consequently, the design focus of TinyML leans more toward specialized applications rather

than facilitating general-purpose research, potentially impeding the pace of overall research advancements.

MCU-based libraries. Due to the resource-constrained environments in TinyML, MCU-based libraries are often designed for specific use cases. For instance, CMSIS-NN [112], a pioneering work for MCU-based libraries developed on ARM Cortex-M devices, proposes an efficient kernel divided into NNfunctions and NNsupportfunctions. NNfunctions execute the main functions in the network, such as convolutions, poolings, and activations. NNsupportfunctions contain data conversions and activation tables. CMIX-NN [15] proposes an open-source mixed and low-precision tool that can support the model's weights and activation to be quantized into 8, 4, and 2 bits arbitrarily. MCUNet [124] presents a co-design framework tailored for DL implementation on commercially available MCUs. This framework incorporates TinyNAS to search for the most accurate and lightweight model efficiently. Additionally, it leverages the TinyEngine, which encompasses code generator-based compilations and in-place depthwise convolution, effectively addressing peak memory constraints. Moving forward, MCUNetV2 [123] introduces a patch-based inference mechanism that operates only on a small spatial region of the feature map, further reducing peak memory use. MicroNet [8] adopts **differentiable NAS (DNAS)** to search for efficient models with a low number of operations and supports the open-source platform **Tensorflow Lite Micro (TFLM)**. MicroNet achieves state-of-the-art results for all TinyMLperf industry-standard benchmark tasks, i.e., Visual Wake Words, Google Speech Commands, and Anomaly detection.

What hinders the rapid development of TinyML? Despite its progress, the growth of TinyML is hindered by several inherent key constraints, including resource constraints, hardware and software heterogeneity, and lack of datasets [160]. Extreme resource constraints, such as an incredibly small size of SRAM and less than 1 MB size flash memory, pose challenges in designing and deploying TinyML models on edge devices. Furthermore, due to hardware heterogeneity and a lack of framework compatibility, current TinyML solutions are tweaked for every individual device, complicating the wide-scale deployment of TinyML algorithms. Besides, existing datasets may not be suitable for TinyML architecture, as the data may not correspond to the data generation feature from external sensors of edge devices. A set of standard datasets suitable for training TinyML models is needed to advance the development of effective TinyML systems. These open research challenges need to be addressed before mass deployment on IoT and edge devices is possible.

5.2 Building Lightweight Large Language Models

LLMs have consistently exhibited outstanding performance across various tasks in the past two years [6, 199, 230]. LLMs hold significant potential for practical applications, especially when paired with human supervision. For instance, they can serve as co-pilots alongside autonomous agents or as sources of inspiration and suggestions. However, these models typically feature parameters at the billion scale. Deploying such models for inference generally demands GPU-level hardware and tens of gigabytes of memory, posing substantial challenges for everyday LLM utilization. For example, Tao et al. [193] find it hard to quantize generative pre-trained language models due to homogeneous word embedding and varied weight distribution. Consequently, transforming a large, resource-intensive LLM model into a compact version suitable for deployment on resource-constrained mobile devices has emerged as a prominent future research direction.

World-renowned enterprises have made significant strides in LLM deployment. In 2023, Qualcomm showcased the independent execution of the text-to-image model, Stable Diffusion [165], and the image-to-image model, ControlNet [244], on mobile devices, thereby accelerating the deployment of large models to edge computing environments. Google also introduced several versions of its latest universal large model, PaLM 2 [6], featuring a lightweight variant tailored

for mobile platforms. This development has created new opportunities for migrating large models from cloud-based systems to edge devices. However, certain large models still require several gigabytes of physical storage and runtime memory. Consequently, efforts are being directed towards achieving a memory footprint of less than 1 GB [160], signifying that significant work is still needed in this area. This section outlines some key initiatives for easing the implementation of LLMs in resource-constrained environments.

5.2.1 Pruning without Re-training. Recently, a substantial body of work has applied common DL quantization and pruning techniques to construct lightweight LLMs. Some approaches [218, 235] focus on implementing quantization, where numerical precision is greatly reduced. SparseGPT [55] demonstrates, for the first time, that large-scale **Generative Pre-trained Transformer (GPT)** models can be pruned to at least 50% sparsity in a single step, without any subsequent retraining, with minimal loss of accuracy. Following this, Wanda (Pruning by Weights and Activations) [180], specifically designed to induce sparsity in pre-trained LLMs, is introduced. Wanda prunes weights with the smallest magnitudes and does not require retraining or weight updates. The pruned LLM can be directly utilized, increasing its practicality. Notably, Wanda surpasses the established baseline of magnitude pruning and competes effectively with recent methods that involve extensive weight updates. These works set a significant milestone for future work in designing LLM pruning methods that do not require retraining.

5.2.2 Model Design. From a model design perspective, we can create lightweight LLMs from the very inception, focusing on reducing the number of model parameters. One promising avenue in this endeavor is prompt tuning, which seeks to optimize the LLMs' performance while maintaining efficiency and model size. A notable approach in this context is **Visual Prompt Tuning (VPT)** [101], which emerges as an efficient and effective alternative to the comprehensive fine-tuning of large-scale Transformer models employed in vision-related tasks. VPT introduces a mere fraction, less than 1%, of trainable parameters within the input space while maintaining the integrity of the model's backbone. Another noteworthy contribution is CALIP [68], which introduces parameter-free attention mechanisms to facilitate effective interaction and communication between visual and text features. It yields text-aware image features and visual-guided text features, contributing to the development of more streamlined and efficient vision-language models. In the near future, one promising avenue for advancing lightweight LLM design is the development of adaptive fine-tuning strategies. These strategies would dynamically adjust the model's architecture and parameters to align with specific task requirements. This adaptability ensures the model can optimize its performance for particular applications without incurring unnecessary parameter bloat.

5.2.3 Building Lightweight Diffusion Model. In recent years, denoising diffusion-based generative models, particularly those of the score-based variety [86, 174], have made notable strides in creating diverse and authentic data. However, the transition of the inference phase of a diffusion model to edge devices poses significant challenges. The inference phase reverses the transformation process to generate real data from Gaussian noise, commonly known as the denoising process. Moreover, when these models are compressed to reduce their footprint and computational demands, there is a potential risk of severe degradation in image quality. The compression process may need simplifications, approximations, or even the removal of essential model components, which could adversely affect the model's ability to reconstruct data from Gaussian noise accurately. Consequently, a critical concern emerges in balancing model size reduction with preserving high-quality image generation, thereby presenting a formidable challenge in developing diffusion models in resource-constrained scenarios.

In a very recent work, Shang et al. [171] introduce post-training quantization [14] into the field of diffusion model acceleration. When applied in a training-free manner, this quantization approach exhibits the capability to enhance the efficiency of the denoising process while simultaneously reducing the storage requirements for diffusion model weights, a critical component in the acceleration of diffusion models. Nevertheless, there remain numerous opportunities for improvement in this domain to achieve a tradeoff between high-quality and lightweight model solutions.

*5.2.4 Deployment of **Vision Transformers (ViTs)**.* Despite the increasing prevalence of lightweight ViTs, deploying ViT in hardware-constrained environments remains a persistent concern. According to Reference [210], ViT inference on mobile devices has a latency and energy consumption of up to 40 times higher than CNN models. Hence, without modification, mobile devices cannot support the inference of ViTs. The self-attention operations in ViTs need to compute the pair-wise relations between image patches, and the computations grow quadratically with the number of patches. Moreover, computation for FFN layers is more time-consuming than attention layers [210]. By removing the redundant attention heads and FFN layers, DeiT-Tiny can reduce latency by 23.2%, with negligible 0.75% accuracy loss.

Several works designed NLP models for embedded systems such as FPGAs [72, 73, 205]. More recently, DiVIT [120] and VAQF [181] proposed hardware-software co-designed solutions for ViTs. DiVIT proposes a delta patch encoding and novel differential attention at the algorithm level that leverages the patch locality during inference. In DiVIT, the design of a differential attention Processing Engine array with bit-saving techniques can calculate the delta with less computation and communicate with differential dataflow. Furthermore, the exponent operation is executed using a lookup table without additional computation and with minimal hardware overhead. VAQF first introduces binarization into ViTs, which can be used for FPGA mapping and quantization training. Specifically, VAQF can generate the required quantization precision and accelerator description for direct software and hardware implementation based on the target frame rate.

To enable the seamless deployment of ViTs in resource-constrained devices, we highlight two potential future directions:

(1) Algorithm optimizations. In addition to the design of efficient ViT models described in Section 2.3, the bottlenecks of ViTs should also be considered. For example, since MatMul operations cause a bottleneck in ViTs, these operations can be accelerated or reduced [210]. Additionally, integer quantization and improvement to operator fusion can be considered.

(2) Hardware Accessibility. Unlike CNNs, which are well-supported on most mobile devices and AI accelerators, ViTs do not have specialized hardware support. For instance, ViT fails to run on mobile GPUs and Intel NCS2 VPU. Based on our findings, some important operators are not supported on specific hardware. Specifically, on the mobile GPU, the concatenate operator requires a 4-dimensional input tensor in TFLiteGPUDelegate, but the tensor in ViTs is 3-dimensional. However, Intel VPU does not support LayerNorm, which exists in the architecture of transformers but is uncommon in CNN. Hence, hardware support for ViTs on resource-constrained devices warrants further investigation.

6 CONCLUSION

Recently, computer vision applications have increasingly prioritized energy conservation, carbon footprint reduction, and cost-effectiveness, highlighting the growing importance of lightweight models, particularly in the context of edge AI. This article conducts a comprehensive examination of lightweight **deep learning (DL)**, exploring prominent models such as MobileNet and Efficient transformer variants, along with prevalent strategies for optimizing these models, including pruning, quantization, knowledge distillation, and neural architecture search. Beyond providing a

detailed explanation of these methods, we offer practical guidance for crafting customized light-weight models, offering clarity through an analysis of their respective strengths and weaknesses.

Furthermore, we discussed hardware acceleration for DL models, delving into hardware architectures, distinct data flow types and data locality optimization techniques, and DL libraries to enhance comprehension of accelerating the training and inference processes. This investigation sheds light on the intricate interplay between hardware and software (co-design), providing insights into expediting training and inference processes from a hardware perspective. Finally, we turn our gaze toward the future, recognizing that the deployment of lightweight DL models in TinyML and LLM technologies presents challenges that demand the exploration of creative solutions in these evolving fields.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*. 265–283.

[2] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane. 2021. Zero-cost proxies for lightweight NAS. In *ICLR*.

[3] 2024. *Advances in Image Manipulation Workshop in Conjunction with ECCV 2022*. Retrieved from https://data.vision.ee.ethz.ch/cvl/aim22/

[4] D. Amodei and D. Hernandez. 2018. *AI and Compute*. Retrieved from https://openai.com/blog/ai-and-compute

[5] S. An, Q. Liao, Z. Lu, and J.-H. Xue. 2022. Efficient semantic segmentation via self-attention and self-distillation. *IEEE Trans. Intell. Transport. Syst.* 23, 9 (2022), 15256–15266.

[6] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, E. Chu, J. H. Clark, L. E. Shafey, Y. Huang, K. Meier-Hellstern, G. Mishra, E. Moreira, M. Omernick, K. Robinson, S. Ruder, Y. Tay, K. Xiao, Y. Xu, Y. Zhang, G. H. Abrego, J. Ahn, J. Austin, P. Barham, J. Botha, J. Bradbury, S. Brahma, K. Brooks, M. Catasta, Y. Cheng, C. Cherry, C. A. Choquette-Choo, A. Chowdhery, C. Crepy, S. Dave, M. Dehghani, S. Dev, J. Devlin, M. Díaz, N. Du, E. Dyer, V. Feinberg, F. Feng, V. Fienber, M. Freitag, X. Garcia, S. Gehrmann, L. Gonzalez, G. Gur-Ari, S. Hand, H. Hashemi, L. Hou, J. Howland, A. Hu, J. Hui, J. Hurwitz, M. Isard, A. Ittycheriah, M. Jagielski, W. Jia, K. Kenealy, M. Krikun, S. Kudugunta, C. Lan, K. Lee, B. Lee, E. Li, M. Li, W. Li, Y. Li, J. Li, H. Lim, H. Lin, Z. Liu, F. Liu, M. Maggioni, A. Mahendru, J. Maynez, V. Misra, M. Moussalem, Z. Nado, J. Nham, E. Ni, A. Nystrom, A. Parrish, M. Pellat, M. Polacek, A. Polozov, R. Pope, S. Qiao, E. Reif, B. Richter, P. Riley, A. C. Ros, A. Roy, B. Saeta, R. Samuel, R. Shelby, A. Slone, D. Smilkov, D. R. So, D. Sohn, S. Tokumine, D. Valter, V. Vasudevan, K. Vodrahalli, X. Wang, P. Wang, Z. Wang, T. Wang, J. Wieting, Y. Wu, K. Xu, Y. Xu, L. Xue, P. Yin, J. Yu, Q. Zhang, S. Zheng, C. Zheng, W. Zhou, D. Zhou, S. Petrov, Y. Wu. 2023. PaLM 2 technical report. Google. *arXiv preprint arXiv:2305.10403* (2023).

[7] A. Asperti, D. Evangelista, and M. Marzolla. 2021. Dissecting FLOPs along input dimensions for GreenAI cost estimations. In *LOD*. 86–100.

[8] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough. 2021. MicroNets: Neural network architectures for deploying TinyML applications on commodity microcontrollers. In *Annual Conference on Machine Learning and Systems*.

[9] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. 2018. Scalable methods for 8-bit training of neural networks. In *Annual Conference on Neural Information Processing Systems*.

[10] M. Bastian. 2024. *GPT-4 Has More than a Trillion Parameters - Report*. Retrieved from https://the-decoder.com/gpt-4-has-a-trillion-parameters/

[11] A. Berthelier, T. Chateau, S. Duffner, C. Garcia, and C. Blanc. 2021. Deep model compression and architecture optimization for embedded systems: A survey. *J. Sig. Process. Syst.* 93, 8 (2021), 863–878.

[12] M. Booshehri, A. Malekpour, and P. Luksch. 2013. An improving method for loop unrolling. *Int. J. Comput. Sci. Inf. Secur.* 11, 5 (2013), 73–76.

[13] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. 2020. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*.

[14] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer. 2020. ZeroQ: A novel zero shot quantization framework. In *CVPR*. 13169–13178.

[15] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini. 2020. CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices. *IEEE Trans. Circ. Syst. II* 67, 5 (2020), 871–875.

[16] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique. 2020. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access* 8 (2020), 225134–225180.

[17] B. Chen, T. Medini, J. Farwell, C. Tai, A. Shrivastava, et al. 2020. SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Annual Conference on Machine Learning and Systems*. 291–306.

[18] C.-Y. Chen, L. Lo, P.-J. Huang, H.-H. Shuai, and W.-H. Cheng. 2021. FashionMirror: Co-attention feature-remapping virtual try-on with sequential template poses. In *ICCV*. 13809–13818.

[19] D. Chen, J.-P. Mei, H. Zhang, C. Wang, Y. Feng, and C. Chen. 2022. Knowledge distillation with the reused teacher classifier. In *CVPR*. 11933–11942.

[20] D. Chen, J.-P. Mei, Y. Zhang, C. Wang, Z. Wang, Y. Feng, and C. Chen. 2021. Cross-layer distillation with semantic calibration. In *AAAI*, Vol. 35. 7028–7036.

[21] H. Chen, Y. Wang, C. Xu, B. Shi, C. Xu, Q. Tian, and C. Xu. 2020. AdderNet: Do we really need multiplications in deep learning? In *CVPR*.

[22] P. Chen, S. Liu, H. Zhao, and J. Jia. 2021. Distilling knowledge via knowledge review. In *CVPR*. 5008–5017.

[23] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Comput. Archit. News* 42, 1 (2014), 269–284.

[24] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. 2016. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPSW*.

[25] W. Chen, Y. Wang, S. Yang, C. Liu, and L. Zhang. 2020. You only search once: A fast automation framework for single-stage DNN/Accelerator co-design. In *DATE*. 1283–1286.

[26] W. Chen, D. Xie, Y. Zhang, and S. Pu. 2019. All you need is a few shifts: Designing efficient convolutional neural networks for image classification. In *CVPR*. 7241–7250.

[27] Y. Chen, X. Dai, D. Chen, M. Liu, X. Dong, L. Yuan, and Z. Liu. 2022. Mobile-Former: Bridging MobileNet and transformer. In *CVPR*. 5270–5279.

[28] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. 2014. DaDianNao: A machine-learning supercomputer. In *MICRO*. 609–622.

[29] Y. Chen, T. Yang, X. Zhang, G. Meng, C. Pan, and J. Sun. 2019. DetNAS: Neural architecture search on object detection. In *NIPS*. 4–1.

[30] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[31] R. Child, S. Gray, A. Radford, and I. Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[32] J. Cho, Y. Jung, S. Lee, and Y. Jung. 2021. Reconfigurable binary neural network accelerator with adaptive parallelism scheme. *Electronics* 10, 3 (2021), 230.

[33] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan. 2018. PACT: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085* (2018).

[34] K. Choi, D. Hong, H. Yoon, J. Yu, Y. Kim, and J. Lee. 2021. DANCE: Differentiable accelerator/network co-exploration. In *DAC*.

[35] F. Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *CVPR*. 1251–1258.

[36] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller. 2021. Rethinking attention with performers. In *ICLR*.

[37] X. Dai, A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda, and J. E. Gonzalez. 2021. FBNetV3: Joint architecture-recipe search using predictor pretraining. In *CVPR*. 16276–16285.

[38] Z. Dai, H. Liu, Q. V. Le, and M. Tan. 2021. CoAtNet: Marrying convolution and attention for all data sizes. In *NIPS*. 3965–3977.

[39] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes. 2021. TensorFlow Lite Micro: Embedded machine learning for TinyML systems. In *MLSys*. 800–811.

[40] J. Deng, W. Li, Y. Chen, and L. Duan. 2021. Unbiased mean teacher for cross-domain object detection. In *CVPR*. 4091–4101.

[41] X. Dong, S. Chen, and S. Pan. 2017. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *NIPS*.

[42] Z. Dong, Z. Yao, D. Arfeen, A. Gholami, M. W. Mahoney, and K. Keutzer. 2020. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. In *NIPS*. 18518–18529.

[43] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. 2019. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *ICCV*. 293–302.

[44] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*.

[45] L. Du, Y. Du, Y. Li, J. Su, Y.-C. Kuan, C.-C. Liu, and M.-C. F. Chang. 2017. A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things. *IEEE Trans. Circ. Syst. I* 65, 1 (2017), 198–208.

[46] S. Dubey, V. K. Soni, and B. K. Dubey. 2019. Application of microcontroller in assembly line for safety and controlling. *Int. J. Res. Analyt. Rev.* 6, 1 (2019), 107–111.

[47] S. d'Ascoli, H. Touvron, M. L. Leavitt, A. S. Morcos, G. Biroli, and L. Sagun. 2021. ConViT: Improving vision transformers with soft convolutional inductive biases. In *ICML*. 2286–2296.

[48] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li. 2021. DeepShift: Towards multiplication-less neural networks. In *CVPR*. 2359–2368.

[49] F. Faghri, I. Tabrizian, I. Markov, D. Alistarh, D. M. Roy, and A. Ramezani-Kebrya. 2020. Adaptive gradient quantization for data-parallel SGD. In *NIPS*. 3174–3185.

[50] Z. Fan, W. Hu, H. Guo, F. Liu, and D. Xu. 2021. Hardware and algorithm co-optimization for pointwise convolution and channel shuffle in ShuffleNet V2. In *SMC*. 3212–3217.

[51] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. 2019. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning*. Springer International Publishing, Cham, 113–134.

[52] 2024. *ONNX*. Retrieved from https://onnx.ai/

[53] M. Fraccaroli, E. Lamma, and F. Riguzzi. 2022. Symbolic DNN-tuner. *Mach. Learn.* 111, 2 (2022), 625–650.

[54] J. Frankle and M. Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*.

[55] E. Frantar and D. Alistarh. 2023. SparseGPT: Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774* (2023).

[56] Z. Fu, M. He, Z. Tang, and Y. Zhang. 2023. Optimizing data locality by executor allocation in spark computing environment. *Comput. Sci. Inf. Syst.* 20, 1 (2023), 491–512.

[57] J. Getzner, B. Charpentier, and S. Günnemann. 2023. Accuracy is not the only metric that matters: Estimating the energy consumption of deep learning models. In *ICLR*.

[58] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. 2022. A survey of quantization methods for efficient neural network inference. *LPCV*, 291–326.

[59] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. 2018. SqueezeNext: Hardware-aware neural network design. In *CVPRW*. 1638–1647.

[60] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. 2017. The reversible residual network: Backpropagation without storing activations. In *NIPS*.

[61] Google. 2023. *Post-training Quantization | TensorFlow Lite*. Retrieved from https://www.tensorflow.org/lite/performance/post_training_quantization

[62] J. Gou, B. Yu, S. J. Maybank, and D. Tao. 2021. Knowledge distillation: A survey. *Int. J. Comput. Vis.* 129, 6 (2021), 1789–1819.

[63] B. Graham, A. El-Nouby, H. Touvron, P. Stock, A. Joulin, H. Jégou, and M. Douze. 2021. LeViT: A vision transformer in ConvNet's clothing for faster inference. In *ICCV*. 12259–12269.

[64] R. M. Gray and D. L. Neuhoff. 1998. Quantization. *Trans. Inf. Theor.* 44, 6 (1998), 2325–2383.

[65] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. 2017. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 37, 1 (2017), 35–47.

[66] Q. Guo, X. Wang, Y. Wu, Z. Yu, D. Liang, X. Hu, and P. Luo. 2020. Online knowledge distillation via collaborative learning. In *CVPR*. 11020–11029.

[67] Y. Guo, A. Yao, and Y. Chen. 2016. Dynamic network surgery for efficient DNNs. In *NIPS*.

[68] Z. Guo, R. Zhang, L. Qiu, X. Ma, X. Miao, X. He, and B. Cui. 2023. CALIP: Zero-shot enhancement of clip with parameter-free attention. In *AAAI*, Vol. 37. 746–754.

[69] M. Gupta and P. Agrawal. 2022. Compression of deep learning models for text: A survey. *ACM Trans. Knowl. Discov. Data* 16, 4 (2022), 1–55.

[70] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. 2015. Deep learning with limited numerical precision. In *ICML*, 1737–1746.

[71] S. Gupta and B. Akin. 2020. Accelerator-aware neural network design using AutoML. In *Annual Conference on Machine Learning and Systems Workshop*.

[72] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong. 2020. Aˆ3: Accelerating attention mechanisms in neural networks with approximation. In *HPCA*. 328–341.

[73] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee. 2021. ELSA: Hardware-Software co-design for efficient, lightweight self-attention mechanism in neural networks. In *ISCA*. 692–705.

[74] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, Z. Yang, Y. Zhang, and D. Tao. 2023. A survey on vision transformer. *IEEE Trans. Pattern Anal. Mach. Intell.* 45, 1 (2023), 87–110.

[75] S. Han, H. Mao, and W. J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*.

[76] B. Hassibi, D. G. Stork, and G. J. Wolff. 1993. Optimal brain surgeon and general network pruning. In *ICNN*. 293–299.

[77] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.

[78] X. He, K. Zhao, and X. Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowl.-based Syst.* 212 (2021), 106622.

[79] Y. He, Y. Ding, P. Liu, L. Zhu, H. Zhang, and Y. Yang. 2020. Learning filter pruning criteria for deep convolutional neural networks acceleration. In *CVPR*. 2006–2015.

[80] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang. 2018. Soft filter pruning for accelerating deep convolutional neural networks. In *IJCAI*. 2234–2240.

[81] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang. 2019. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *CVPR*. 4340–4349.

[82] Y. He, X. Liu, H. Zhong, and Y. Ma. 2019. AddressNet: Shift-based primitives for efficient convolutional neural networks. In *WACV*. 1213–1222.

[83] Y. He, X. Zhang, and J. Sun. 2017. Channel pruning for accelerating very deep neural networks. In *CVPR*. 1389–1397.

[84] S. C. Hidayati, T. W. Goh, J.-S. G. Chan, C.-C. Hsu, J. See, L.-K. Wong, K.-L. Hua, Y. Tsao, and W.-H. Cheng. 2020. Dress with style: Learning style from joint deep embedding of clothing styles and body shapes. *Trans. Multim.* 23 (2020), 365–377.

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

[86] J. Ho, A. Jain, and P. Abbeel. 2020. Denoising diffusion probabilistic models. In *NIPS*. 6840–6851.

[87] Y. Hou, Z. Ma, C. Liu, and C. C. Loy. 2019. Learning lightweight lane detection CNNs by self attention distillation. In *ICCV*. 1013–1021.

[88] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. 2019. Searching for MobileNetV3. In *ICCV*. 1314–1324.

[89] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[90] L.-C. Hsu, C.-T. Chiu, K.-T. Lin, H.-H. Chou, and Y.-Y. Pu. 2020. ESSA: An energy-aware bit-serial streaming deep convolutional neural network accelerator. *J. Syst. Archit.* 111 (2020), 101831.

[91] J. Hu, L. Shen, and G. Sun. 2018. Squeeze-and-excitation networks. In *CVPR*. 7132–7141.

[92] W. Hu, Z. Che, N. Liu, M. Li, J. Tang, C. Zhang, and J. Wang. 2023. CATRO: Channel pruning via class-aware trace ratio optimization. *Trans. Neural. Netw. Learn. Syst.* (2023), 1–13. DOI : https://doi.org/10.1109/TNNLS.2023.3262952

[93] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger. 2018. CondenseNet: An efficient DenseNet using learned group convolutions. In *CVPR*. 2752–2761.

[94] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. 2017. Densely connected convolutional networks. In *CVPR*. 4700–4708.

[95] J.-C. Huang and T. Leng. 1999. Generalized loop-unrolling: A method for program speedup. In *ASSET*. 244–248.

[96] Z. Huang and N. Wang. 2019. Like what you like: Knowledge distill via neuron selectivity transfer. In *ICLR*.

[97] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. 2016. Binarized neural networks. In *NIPS*. 4114–4122.

[98] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. 2017. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. In *ICLR*.

[99] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2704–2713.

[100] Y. Jeon and J. Kim. 2018. Constructing fast network through deconstruction of convolution. In *NIPS*.

[101] M. Jia, L. Tang, B.-C. Chen, C. Cardie, S. Belongie, B. Hariharan, and S.-N. Lim. 2022. Visual prompt tuning. In *ECCV*. 709–727.

[102] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson. 2023. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *ISCA*. 1–14.

[103] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*. 1–12.

[104] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi. 2019. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *CVPR*. 4350–4359.

[105] B. Kang, X. Chen, D. Wang, H. Peng, and H. Lu. 2023. Exploring lightweight hierarchical vision transformers for efficient visual tracking. In *ICCV*. 9612–9621.

[106] M. Kang and B. Han. 2020. Operation-aware soft channel pruning using differentiable masks. In *ICML*. 7021–7032.

[107] K. Kim, B. Ji, D. Yoon, and S. Hwang. 2021. Self-knowledge distillation with progressive refinement of targets. In *ICCV*. 6567–6576.

[108] N. Kitaev, Ł. Kaiser, and A. Levskaya. 2020. Reformer: The efficient transformer. In *ICLR*.

[109] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. 2019. Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA. In *Automated Machine Learning*. Springer International Publishing, Cham, 81–95.

[110] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.

[111] S. Kumar, V. Bitorff, D. Chen, C. Chou, B. Hechtman, H. Lee, N. Kumar, P. Mattson, S. Wang, T. Wang, et al. 2019. Scale MLPerf-0.6 models on Google TPU-v3 pods. *arXiv preprint arXiv:1909.09756* (2019).

[112] L. Lai, N. Suda, and V. Chandra. 2018. CMSIS-NN: Efficient neural network kernels for ARM Cortex-M CPUs. *arXiv preprint arXiv:1801.06601* (2018).

[113] Y. LeCun, J. Denker, and S. Solla. 1989. Optimal brain damage. In *NIPS*.

[114] N. Lee, T. Ajanthan, and P. H. Torr. 2019. SNIP: Single-shot network pruning based on connection sensitivity. In *ICLR*.

[115] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. 2017. Pruning filters for efficient ConvNets. In *ICLR*.

[116] N. Li, S. Takaki, Y. Tomiokay, and H. Kitazawa. 2016. A multistage dataflow implementation of a deep convolutional neural network based on FPGA for high-speed object recognition. In *SSIAI*. 165–168.

[117] S. Li, M. Lin, Y. Wang, Y. Wu, Y. Tian, L. Shao, and R. Ji. 2023. Distilling a powerful student model via online knowledge distillation. *Trans. Neural. Netw. Learn. Syst.* 34, 11 (2023), 8743–8752.

[118] S. Li, M. Tan, R. Pang, A. Li, L. Cheng, Q. V. Le, and N. P. Jouppi. 2021. Searching for fast model families on datacenter accelerators. In *CVPR*. 8085–8095.

[119] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen. 2020. EDD: Efficient differentiable DNN architecture and implementation co-search for embedded AI solutions. In *DAC*. 1–6.

[120] Y. Li, Y. Hu, F. Wu, and K. Li. 2022. DiVIT: Algorithm and architecture co-design of differential attention in vision transformer. *J. Syst. Archit.* 128, C (2022), 102520.

[121] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[122] Y. Liang, G. Chongjian, Z. Tong, Y. Song, J. Wang, and P. Xie. 2021. EViT: Expediting vision transformers via token reorganizations. In *ICLR*.

[123] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han. 2021. MCUNetV2: Memory-efficient patch-based inference for tiny deep learning. In *NIPS*.

[124] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. 2020. MCUNet: Tiny deep learning on IoT devices. In *NIPS*. 11711–11722.

[125] S. Lin, H. Xie, B. Wang, K. Yu, X. Chang, X. Liang, and G. Wang. 2022. Knowledge distillation via the target-aware transformer. In *CVPR*. 10915–10924.

[126] Y. Lin, D. Hafdi, K. Wang, Z. Liu, and S. Han. 2020. Neural-hardware architecture search. In *NIPSWS*.

[127] Y.-J. Lin and T. S. Chang. 2017. Data and hardware efficient design for convolutional neural network. *IEEE Trans. Circ. Syst. I* 65, 5 (2017), 1642–1651.

[128] B. Liu, F. Li, X. Wang, B. Zhang, and J. Yan. 2023. Ternary weight networks. In *ICASSP*. 1–5.

[129] H. Liu, K. Simonyan, and Y. Yang. 2019. DARTS: Differentiable architecture search. In *ICLR*.

[130] L. Liu, S. Zhang, Z. Kuang, A. Zhou, J.-H. Xue, X. Wang, Y. Chen, W. Yang, Q. Liao, and W. Zhang. 2021. Group Fisher pruning for practical network compression. In *ICML*.

[131] X. Liu, M. Ye, D. Zhou, and Q. Liu. 2021. Post-training quantization with multiple points: Mixed precision without mixed precision. In *AAAI*, Vol. 35. 8697–8705.

[132] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong, F. Wei, and B. Guo. 2022. Swin transformer V2: Scaling up capacity and resolution. In *CVPR*. 12009–12019.

[133] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *ICCV*. 10012–10022.

[134] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun. 2019. Metapruning: Meta learning for automatic neural network channel pruning. In *ICCV*. 3296–3305.

[135] G. Luo, Y. Zhou, X. Sun, Y. Wang, L. Cao, Y. Wu, F. Huang, and R. Ji. 2022. Towards lightweight transformer via group-wise transformation for vision-and-language tasks. *Trans. Image. Process.* 31 (2022), 3386–3398.

[136] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen. 2016. DaDianNao: A neural network supercomputer. *IEEE Trans. Comput.* 66, 1 (2016), 73–88.

[137] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. 2018. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *ECCV*. 116–131.

[138] 2021. *Mobile AI Workshop 2021*. Retrieved from https://ai-benchmark.com/workshops/mai/2021/#challenges

[139] 2022. *Mobile AI Workshop 2022*. Retrieved from https://ai-benchmark.com/workshops/mai/2022/#challenges

[140] 2023. *Mobile AI Workshop 2023*. Retrieved from https://ai-benchmark.com/workshops/mai/2023/#challenges

[141] S. Mehta, M. Ghazvininejad, S. Iyer, L. Zettlemoyer, and H. Hajishirzi. 2021. DeLighT: Very deep and light-weight transformer. In *ICLR*.

[142] S. Mehta, R. Koncel-Kedziorski, M. Rastegari, and H. Hajishirzi. 2018. Pyramidal recurrent unit for language modeling. In *EMNLP*.

[143] S. Mehta, R. Koncel-Kedziorski, M. Rastegari, and H. Hajishirzi. 2020. DeFINE: Deep factorized input token embeddings for neural sequence modeling. In *ICLR*.

[144] S. Mehta and M. Rastegari. 2022. MobileViT: Light-weight, general-purpose, and mobile-friendly vision transformer. In *ICLR*.

[145] L. Mezdour, K. Kadem, M. Merouani, A. S. Haichour, S. Amarasinghe, and R. Baghdadi. 2023. A deep learning model for loop interchange. In *ACM SIGPLAN CC*. 50–60.

[146] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. 2018. Mixed precision training. In *ICLR*.

[147] M. M. Naseer, K. Ranasinghe, S. H. Khan, M. Hayat, F. Shahbaz Khan, and M.-H. Yang. 2021. Intriguing properties of vision transformers. In *NIPS*.

[148] A. Nechi, L. Groth, S. Mulhem, F. Merchant, R. Buchty, and M. Berekovic. 2023. FPGA-based deep learning inference accelerators: Where are we standing? *ACM Trans. Reconfig. Technol. Syst.* 16, 4 (2023), 1–32.

[149] NVIDIA. 2023. *NVIDIA CUDA-X: GPU Accelerated Libraries*. Retrieved from https://developer.nvidia.com/gpu-accelerated-libraries

[150] OpenAI. 2023. GPT-4 technical report. OpenAI. (2023).

[151] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Comput. Archit. News* 45, 2 (2017), 27–40.

[152] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NIPS*.

[153] H. Peng, J. Wu, S. Chen, and J. Huang. 2019. Collaborative channel pruning for deep networks. In *ICML*. 5113–5122.

[154] H. Pouransari, Z. Tu, and O. Tuzel. 2020. Least squares binary quantization of neural networks. In *CVPRW*. 698–699.

[155] Z. Qi, W. Chen, R. A. Naqvi, and K. Siddique. 2022. Designing deep learning hardware accelerator and efficiency evaluation. *Comput. Intell. Neurosci.* 2022 (2022).

[156] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *ACM FPGA*. 26–35.

[157] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár. 2020. Designing network design spaces. In *CVPR*. 10428–10436.

[158] Y. Rao, W. Zhao, B. Liu, J. Lu, J. Zhou, and C.-J. Hsieh. 2021. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *NIPS*. 13937–13949.

[159] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *ECCV*. 525–542.

[160] P. P. Ray. 2022. A review on TinyML: State-of-the-art and prospects. *J. King Saud Univ.-Comput. Inf. Sci.* 34, 4 (2022), 1595–1623.

[161] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. 2017. Large-scale evolution of image classifiers. In *ICML*. 2902–2911.

[162] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *Comput. Surv.* 54, 4 (2021), 1–34.

[163] D. Roggen, R. Cobden, A. Pouryazdan, and M. Zeeshan. 2022. Wearable FPGA platform for accelerated DSP and AI applications. In *PerComW*. 66–69.

[164] B. Rokh, A. Azarpeyvand, and A. Khanteymoori. 2023. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Trans. Intell. Syst. Technol.* 14, 6 (2023), 1–50.

[165] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *CVPR*. 10684–10695.

[166] C. Sakr, S. Dai, R. Venkatesan, B. Zimmer, W. Dally, and B. Khailany. 2022. Optimal clipping and magnitude-aware differentiation for improved quantization-aware training. In *ICML*. 19123–19138.

[167] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *CVPR*. 4510–4520.

[168] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni. 2020. Green AI. *Commun. ACM* 63, 12 (2020), 54–63.

[169] L. Sekanina. 2021. Neural architecture search and hardware accelerator co-search: A survey. *IEEE Access* 9 (2021), 151337–151362.

[170] K. P. Seng, P. J. Lee, and L. M. Ang. 2021. Embedded intelligence on FPGA: Survey, applications and challenges. *Electronics* 10, 8 (2021), 895.

[171] Y. Shang, Z. Yuan, B. Xie, B. Wu, and Y. Yan. 2023. Post-training quantization on diffusion models. In *CVPR*. 1972–1981.

[172] K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *ICLR*.

[173] S. Sinha. 2023. *State of IoT 2023: Number of Connected IoT Devices Growing 16% to 16.7 Billion Globally*. Retrieved from https://iot-analytics.com/number-connected-iot-devices/

[174] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole. 2021. Score-based generative modeling through stochastic differential equations. In *ICLR*.

[175] A. Srinivas, T.-Y. Lin, N. Parmar, J. Shlens, P. Abbeel, and A. Vaswani. 2021. Bottleneck transformers for visual recognition. In *CVPR*. 16519–16529.

[176] A. Stoutchinin, F. Conti, and L. Benini. 2019. Optimally scheduling CNN convolutions for efficient memory access. *arXiv preprint arXiv:1902.01492* (2019).

[177] E. Strubell, A. Ganesh, and A. McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *ACL*.

[178] Z. Su, L. Fang, W. Kang, D. Hu, M. Pietikäinen, and L. Liu. 2020. Dynamic group convolution for accelerating convolutional neural networks. In *ECCV*. 138–155.

[179] M. Sultana, M. Naseer, M. H. Khan, S. Khan, and F. S. Khan. 2022. Self-distilled vision transformer for domain generalization. In *ACCV*. 3068–3085.

[180] M. Sun, Z. Liu, A. Bair, and J. Z. Kolter. 2023. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695* (2023).

[181] M. Sun, H. Ma, G. Kang, Y. Jiang, T. Chen, X. Ma, Z. Wang, and Y. Wang. 2022. VAQF: Fully automatic software-hardware co-design framework for low-bit vision transformer. *arXiv preprint arXiv:2201.06618* (2022).

[182] Y. Sun, H. Wang, B. Xue, Y. Jin, G. G. Yen, and M. Zhang. 2020. Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. *IEEE Trans. Evolut. Computat.* 24, 2 (2020), 350–364.

[183] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. 2020. How to evaluate deep neural network processors: Tops/w (alone) considered harmful. *IEEE Solid-state Circ. Mag.* 12, 3 (2020), 28–41.

[184] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. 2017. Inception-v4, inception-ResNet and the impact of residual connections on learning. In *AAAI*.

[185] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *CVPR*. 1–9.

[186] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*. 2818–2826.

[187] A. Talwalkar. 2020. *The Push for Energy Efficient "Green AI."* Retrieved from https://spectrum.ieee.org/energy-efficient-green-ai-strategies

[188] J. Tan, L. Niu, J. K. Adams, V. Boominathan, J. T. Robinson, R. G. Baraniuk, and A. Veeraraghavan. 2019. Face detection and verification using lensless cameras. *Trans. Computat. Imag.* 5, 2 (2019), 180–194.

[189] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. 2019. MnasNet: Platform-aware neural architecture search for mobile. In *CVPR*. 2820–2828.

[190] M. Tan and Q. Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *ICML*. 6105–6114.

[191] M. Tan and Q. Le. 2021. EfficientNetV2: Smaller models and faster training. In *ICML*. 10096–10106.

[192] M. Tan and Q. V. Le. 2019. MixConv: Mixed depthwise convolutional kernels. In *BMVC*.

[193] C. Tao, L. Hou, W. Zhang, L. Shang, X. Jiang, Q. Liu, P. Luo, and N. Wong. 2022. Compression of generative pre-trained language models via quantization. In *ACL*.

[194] A. Tarvainen and H. Valpola. 2017. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *NIPS*.

[195] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler. 2021. Efficient transformers: A survey. *Comput. Surv.* 54, 4 (2021), 1–41.

[196] Y. Tian, D. Krishnan, and P. Isola. 2020. Contrastive representation distillation. In *ICLR*.

[197] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou. 2021. Training data-efficient image transformers & distillation through attention. In *ICML*. 10347–10357.

[198] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[199] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[200] S. Um, S. Kim, S. Kim, and H.-J. Yoo. 2021. A 43.1 TOPS/W energy-efficient absolute-difference-accumulation operation computing-in-memory with computation reuse. *IEEE Trans. Circ. Syst. II* 68, 5 (2021), 1605–1609.

[201] H. Vanholder. 2016. Efficient inference with tensorrt. In *GPU Technology Conference*.

[202] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *NIPS*.

[203] L. N. Viet, T. N. Dinh, D. T. Minh, H. N. Viet, and Q. L. Tran. 2021. UET-Headpose: A sensor-based top-view head pose dataset. In *KSE*. 1–7.

[204] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen, P. Vajda, and J. E. Gonzalez. 2020. FBNetV2: Differentiable neural architecture search for spatial and channel dimensions. In *CVPR*. 12965–12974.

[205] H. Wang, Z. Zhang, and S. Han. 2021. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. In *HPCA*. 97–110.

[206] L. Wang, X. Dong, Y. Wang, L. Liu, W. An, and Y. Guo. 2022. Learnable lookup table for neural network quantization. In *CVPR*. 12423–12433.

[207] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. In *NIPS*. 7686–7695.

[208] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768* (2020).

[209] X. Wang, M. Kan, S. Shan, and X. Chen. 2019. Fully learnable group convolution for acceleration of deep neural networks. In *CVPR*. 9049–9058.

[210] X. Wang, L. L. Zhang, Y. Wang, and M. Yang. 2022. Towards efficient vision transformer inference: A first study of transformers on mobile devices. In *WMCSA*. 1–7.

[211] Z. Wang, K. Xu, S. Wu, L. Liu, L. Liu, and D. Wang. 2020. Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2. *IEEE Access* 8 (2020), 116569–116585.

[212] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*. 1–6.

[213] M. E. Wolf and M. S. Lam. 1991. A data locality optimizing algorithm. In *PLDI*. 30–44.

[214] M. Wortsman, G. Ilharco, S. Y. Gadre, R. Roelofs, R. Gontijo-Lopes, A. S. Morcos, H. Namkoong, A. Farhadi, Y. Carmon, S. Kornblith, and L. Schmidt. 2022. Model soups: Averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *ICML*, 23965–23998.

[215] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer. 2019. FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search. In *CVPR*. 10734–10742.

[216] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer. 2018. Shift: A zero flop, zero parameter alternative to spatial convolutions. In *CVPR*. 9127–9135.

[217] H. Wu, B. Xiao, N. Codella, M. Liu, X. Dai, L. Yuan, and L. Zhang. 2021. CVT: Introducing convolutions to vision transformers. In *ICCV*. 22–31.

[218] X. Wu, C. Li, R. Y. Aminabadi, Z. Yao, and Y. He. 2023. Understanding INT4 quantization for transformer models: Latency speedup, composability, and failure cases. *arXiv preprint arXiv:2301.12017* (2023).

[219] Z. Wu, Z. Liu, J. Lin, Y. Lin, and S. Han. 2020. Lite transformer with long-short range attention. In *ICLR*.

[220] T. Xiao, P. Dollar, M. Singh, E. Mintun, T. Darrell, and R. Girshick. 2021. Early convolutions help transformers see better. In *NIPS*.

[221] H. Xie, M.-X. Lee, T.-J. Chen, H.-J. Chen, H.-I. Liu, H.-H. Shuai, and W.-H. Cheng. 2023. Most important person-guided dual-branch cross-patch attention for group affect recognition. In *ICCV*. 20598–20608.

[222] R. Xu, E. H.-M. Sha, Q. Zhuge, Y. Song, and H. Wang. 2023. Loop interchange and tiling for multi-dimensional loops to minimize write operations on NVMs. *J. Syst. Archit.* 135 (2023), 102799.

[223] Y. Xue, C. Chen, and A. Słowik. 2023. Neural architecture search based on a multi-objective evolutionary algorithm with probability stack. *IEEE Trans. Evolut. Comput.* 27, 4 (2023).

[224] C. Yang, L. Xie, C. Su, and A. L. Yuille. 2019. Snapshot distillation: Teacher-student optimization in one generation. In *CVPR*. 2859–2868.

[225] J. Yang, B. Martinez, A. Bulat, G. Tzimiropoulos. 2021. Knowledge distillation via softmax regression representation learning. In *ICLR*.

[226] L. Yang, H. Jiang, R. Cai, Y. Wang, S. Song, G. Huang, and Q. Tian. 2021. CondenseNet V2: Sparse feature reactivation for deep networks. In *CVPR*. 3569–3578.

[227] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. 2018. NetAdapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*. 285–300.

[228] T.-J. Yang, Y.-L. Liao, and V. Sze. 2021. NetAdaptv2: Efficient neural architecture search with fast super-network training and architecture optimization. In *CVPR*. 2402–2411.

[229] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. Mahoney. 2021. HAWQ-V3: Dyadic neural network quantization. In *ICML*. 11875–11886.

[230] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang, and X. Huang. 2023. A comprehensive capability analysis of GPT-3 and GPT-3.5 series models. *arXiv preprint arXiv:2303.10420* (2023).

[231] J. Ye, X. Lu, Z. Lin, and J. Z. Wang. 2018. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *ICLR*.

[232] H. Yin, A. Vahdat, J. Alvarez, A. Mallya, J. Kautz, and P. Molchanov. 2022. AdaViT: Adaptive tokens for efficient vision transformer. In *CVPR*, 10809–10818.

[233] J. Yoon, D. Kang, and M. Cho. 2022. Semi-supervised domain adaptation via sample-to-sample self-distillation. In *WACV*. 1978–1987.

[234] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin. 2020. ShiftAddNet: A hardware-inspired deep network. In *NIPS*. 2771–2783.

[235] C. Yu, T. Chen, and Z. Gan. 2023. Boost transformer-based language models with GPU-friendly sparsity and quantization. In *ACL*. 218–235.

[236] J. Yu, J. Liu, X. Wei, H. Zhou, Y. Nakata, D. Gudovskiy, T. Okuno, J. Li, K. Keutzer, and S. Zhang. 2022. Cross-domain object detection with mean-teacher transformer. In *ECCV*.

[237] L. Yuan, Y. Chen, T. Wang, W. Yu, Y. Shi, Z. Jiang, F. E. Tay, J. Feng, and S. Yan. 2021. Tokens-to-token ViT: Training vision transformers from scratch on ImageNet. In *ICCV*. 558–567.

[238] L. Yuan, F. E. Tay, G. Li, T. Wang, and J. Feng. 2020. Revisiting knowledge distillation via label smoothing regularization. In *CVPR*. 3903–3911.

[239] M. Yuan and Y. Lin. 2006. Model selection and estimation in regression with grouped variables. *J. R. Stat. Soc. B* 68, 1 (2006), 49–67.

[240] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *ACM FPGA*. 161–170.

[241] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong. 2018. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 38, 11 (2018), 2072–2085.

[242] H. Zhang, Z. Hu, W. Qin, M. Xu, and M. Wang. 2021. Adversarial co-distillation learning for image recognition. *Pattern Recog.* 111 (2021), 107659.

[243] H. Zhang, F. Li, S. Liu, L. Zhang, H. Su, J. Zhu, L. M. Ni, and H.-Y. Shum. 2023. DINO: DETR with improved DeNoising anchor boxes for end-to-end object detection. In *ICLR*.

[244] L. Zhang, A. Rao, and M. Agrawala. 2023. Adding conditional control to text-to-image diffusion models. In *ICCV*. 3836–3847.

[245] L. Zhang, J. Song, A. Gao, J. Chen, C. Bao, and K. Ma. 2019. Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In *ICCV*. 3713–3722.

[246] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *MICRO*. 1–12.

[247] X. Zhang, X. Zhou, M. Lin, and J. Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*. 6848–6856.

[248] F. Faghri, I. Tabrizian, I. Markov, D. Alistarh, D. M. Roy, and A. Ramezani-Kebrya. 2020. Adaptive gradient quantization for data-parallel SGD. *NIPS* 33 (2020), 3174–3185.

[249] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu. 2018. Deep mutual learning. In *CVPR*. 4320–4328.

[250] Z. Zhang, J. Li, W. Shao, Z. Peng, R. Zhang, X. Wang, and P. Luo. 2019. Differentiable learning-to-group channels via groupable convolutional neural networks. In *ICCV*. 3542–3551.

[251] B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang. 2022. Decoupled knowledge distillation. In *CVPR*. 11953–11962.

[252] D. Zhou, Q. Hou, Y. Chen, J. Feng, and S. Yan. 2020. Rethinking bottleneck structure for efficient mobile network design. In *ECCV*. 680–697.

[253] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *MICRO*. 15–28.

[254] Y. Zhou, X. Dong, B. Akin, M. Tan, D. Peng, T. Meng, A. Yazdanbakhsh, D. Huang, R. Narayanaswami, and J. Laudon. 2021. Rethinking co-design of neural architectures and hardware accelerators. *arXiv preprint arXiv:2102.08619* (2021).

[255] C. Zhu, S. Han, H. Mao, and W. J. Dally. 2017. Trained ternary quantization. In *ICLR*.

[256] B. Zoph and Q. V. Le. 2017. Neural architecture search with reinforcement learning. In *ICLR*.