Design Automation for Fast, Lightweight, and Effective Deep Learning Models: A Survey

Dalin Zhang, Member, IEEE, Kaixuan Chen, Member, IEEE, Yan Zhao, Member, IEEE, Bin Yang, Senior Member, IEEE, Lina Yao, Senior Member, IEEE, and Christian S. Jensen, Fellow, IEEE

Abstract—Deep learning technologies have demonstrated remarkable effectiveness in a wide range of tasks, and deep learning holds the potential to advance a multitude of applications, including in edge computing, where deep models are deployed on edge devices to enable instant data processing and response. A key challenge is that while the application of deep models often incurs substantial memory and computational costs, edge devices typically offer only very limited storage and computational capabilities that may vary substantially across devices. These characteristics make it difficult to build deep learning solutions that unleash the potential of edge devices while complying with their constraints. A promising approach to addressing this challenge is to automate the design of effective deep learning models that are lightweight, require only a little storage, and incur only low computational overheads. This survey offers comprehensive coverage of studies of design automation techniques for deep learning models targeting edge computing. It offers an overview and comparison of key metrics that are used commonly to quantify the proficiency of models in terms of effectiveness, lightness, and computational costs. The survey then proceeds to cover three categories of the state-of-the-art of deep model design automation techniques: automated neural architecture search, automated model compression, and joint automated design and compression. Finally, the survey covers open issues and directions for future research.

Index Terms—deep learning, neural architecture search, lightweight model, model compression

I. INTRODUCTION

A. Background

Deep learning has achieved state-of-the-art performance at a multitude of tasks and has affected people’s lives in myriad areas, including recommendation systems [1], natural language understanding [2], and biomedical engineering [3]. Deep learning frees researchers from manually designing purposeful feature representations of objects by introducing multi-layer neural architectures capable of automatic feature extraction. This enables researchers to work at a higher level of abstraction, focusing on architecture engineering rather than on feature engineering and model building. The neural architectures of deep models tend to be increasingly intricate and complex, thus requiring substantial hardware resources for deployment. This may not be a problem when a powerful server is available, but important settings occur where this is not the case: 1) computing on mobile hardware, e.g., smartphones and tablet PCs; 2) computing on industrial hardware optimized for low deployment cost and low power consumption. Furthermore, although cloud computing may be available, it is often fundamentally unattractive to transfer data from edge devices to the cloud. On the one hand, such transfer may incur privacy, ownership, and consequent regulatory concerns, including for human-related data like audio and video data and utility consumption data [4]. On the other hand, data transfer incurs substantial latency due to low bandwidth “last mile” connectivity. Last but not least, deployment of complex models is expensive, due to the cost of specialized hardware and energy consumption, and it is also bad for the environment due to the carbon footprint of producing the required electricity [5].

Fortunately, research has demonstrated that deep learning models generally have large numbers of redundant parameters and computations that contribute to their performance [7], [8], so there is considerable room for reducing redundancies without compromising model accuracy. Hence, it is highly desirable and completely possible to design simplified deep learning models with reduced computational complexity, thus achieving lighter weight and more efficient deep models [4].

B. Design of Efficient Deep Learning Models

Research on building efficient deep learning models can be categorized into two categories: efficient network architecture design and model compression. In efficient network architecture design, the aim is to create compact neural modules and to connect these according to a carefully designed topology. The objective is to achieve efficient deep learning models with acceptable accuracy but small structures (low memory requirement) and low computational complexity (high speed). MobileNet [9] proposes an efficient network structure using a depthwise separable convolution module as the basic building block, bottleneck depth-separable convolution with residuals/inverted residual [10]. As a result, MobileNetV2 generally needs 30% fewer parameters, requires two times fewer operations and is about 30–40% faster on a Google Pixel phone while achieving higher accuracy than its predecessor. In

*By efficient deep learning models we mean deep learning models with low memory usage or low inference cost or latency.
architecture design space

In contrast to directly designing efficient architecture from a pool of basic building blocks, **model compression** aims at modifying a given neural model to reduce its memory and computational cost. **Pruning** is one such powerful technique that tries to remove unimportant components from a model. It is flexible in that it is possible to remove layers, neurons, connections, or channels. While pruning shrinks a model by removing redundant parts, **quantization** aims to reduce the number of bits required to represent model parameters. Most processors use 32 bits or more to store the parameters of a deep model. However, research estimates that the human brain stores information in a discrete format that uses 4–7 bits. Indeed, many efforts have been devoted to investigating using fewer bits to store model parameters to reduce memory and computational cost. Other techniques like knowledge distillation and tensor decomposition are also popular and effective at compressing deep models.

**C. Design Automation for Efficient Deep Learning Models**

Although remarkable progress has been achieved in building efficient deep learning models, initial proposals were heuristic rule-based and hand-tuned with inevitable limitations. First, building an efficient deep learning model still requires advanced prior knowledge and experience, making it difficult for beginners and even deep learning experts without domain knowledge to develop specialized models that meet given requirements. Second, as it is impossible to apply the same uniform model across diverse mobile platforms and tasks, enabling specializations that address such diversity is essential. Yet, it remains excessively time-consuming and inconvenient. Third, hand-crafted rules offer limited capabilities at utilizing hardware potentials fully, while satisfying the size and latency requirements. Different deep learning models can satisfy the same hardware constraints, and it is impractical to manually exhaust all possibilities.

Observations such as the above have prompted a multitude of studies of design automation techniques for efficient deep learning models. Fig. 1 shows a general automated design process. The design automation algorithm (i.e., **Automatic Designer**) applies a search strategy to find network architectures and compression in the predefined **Deep Learning Architecture Design Space**; next, a specific deep learning model is **Derived** through executing the identified operations and is then **Deployed** on target devices (e.g., CPU, GPU, or IoT devices); lastly, model performance metrics such as **accuracy**, **latency**, and **memory use** are estimated and provided to the Automatic Designer. In the next iteration, the Automatic Designer considers both the feedback and specialized constraints and takes a new design action to find a better model in the design space. This process is repeated until a satisfactory model is achieved.

Neural architecture search (NAS) aims to automate the design of neural networks that achieve the best possible accuracy. Next, more targeted studies that aim to automate the design of efficient neural networks build on generic NAS and involve the design of search spaces and the modification of the optimization objective from sole accuracy to both accuracy and efficiency. Ci et al. incorporate model latency into the optimization goal of their binarized design automation framework (i.e., ProxylessNAS) by means of a differentiable loss. Due to targeting optimized inference latency directly, ProxylessNAS can achieve efficient neural architectures 1.83 times faster than MobileNetV2 with the same level of top-1 accuracy. Notably, the search space of ProxylessNAS is based on the inverted residual blocks of different convolution sizes, as proposed by MobileNetV2. This implies that design automation is not only able to reduce human labor but even enables architectures that surpass handcrafted architectures.

In addition to studies that target the direct design of efficient neural networks, other studies target the automated compression of deep neural networks. For example, Xiao et al. design an automatic pruning approach that is based on learnable pruning indicators instead of pruning rules designed individually for specific architectures and datasets. This approach achieves superior compression performance on different widely-used neural models (e.g., AlexNet, ResNet, and MobileNet). Considering the progress in, and promise of, design automation for efficient deep learning models, it is a comprehensive and systematic survey is called for and holds the potential to accelerate future research.

**D. Key Contributions**

To the best of our knowledge, this is the first survey of state-of-the-art design automation methods that target fast, lightweight, and effective deep learning models. The key contributions of the survey are summarized as follows:

- We provide a comprehensive review of design automation techniques targeting fast, lightweight, and effective deep learning models. In doing so, more than 150 papers are covered, analyzed, and compared.
- We propose a new taxonomy of deep design automation methods from the perspectives of how to design, i.e., search, compression, pruning, and joint search and compression, and by what to design, i.e., the search space, the search strategy, and the performance estimation strategy; and by what to compress, e.g., tensors, knowledge, and representation. The detailed categories provide convenience to the readers in obtaining an overview of the literature and identifying a direction of interest.
- We summarize and compare the evaluation metrics that are used for both the obtained models and the design approaches. By means of the comparison, we emphasize the role of each metric and explicate the associated pros and cons.
- We discuss open issues and identify future directions on automated design and compression.
The remainder of the survey is organized as follows: Section II summarizes the evaluation metrics of efficiency (i.e., speed and lightness) and effectiveness of deep learning models. Section III covers studies of searching for efficient deep models, and Section IV then covers research on automated compression, and Section V considers studies of joint automated search and compression. Section VI presents research directions, and Section VII concludes the survey.

II. EVALUATION METRICS

Different evaluation metrics or objectives may lead to different or even opposite conclusions. Thus, it is indispensable to introduce and distinguish the relevant evaluation metrics before diving into the details of efficient models. In this section, we will introduce the evaluation metrics for measuring the efficiency and effectiveness of the obtained efficient models. These metrics are also critical to evaluating the effectiveness of a design automation approach. Furthermore, we introduce metrics for evaluating the cost of a design automation approach as well. We briefly summarize the characteristics including advantages and limitations of the commonly used evaluation metrics in Table I.

A. Device-agnostic Evaluation Metrics

The device-agnostic metrics can be calculated directly from the model architecture without real-world implementation on hardware. These metrics do not essentially reflect a model’s real performance that we care about [12], [27], [28].

1) FLoating-point OPerations (FLOPs): FLOPs is generally defined as the number of floating-point multiplication-add operations in a model for approximating the latency/speed or computation complexity [11], [12], [29]. There are also other commonly used analogous metrics, such as the number of multiply-add operations (MAdds) and the number of multiply-accumulate operations (MACs) [30], [27], [31]. However, one contention remains regarding the definition: whether multiplication-add should be considered as one or two operations. Some researchers argue that in many recent deep learning models, convolutions are bias-free and it makes sense to count multiplication and add as separate FLOPs [32]. Moreover, some non-multiplication or non-add operations require FLOPs in some implementations as well, such as an activation layer [33], [27]. Whether such operations should be counted into total FLOPs is also a dispute.

In addition to the inconsistency of the definition, there exist three main issues that induce the discrepancy between FLOPs and real latency. First, counting only FLOPs ignores some decisive factors that affect latency remarkably. One such factor is parallelism. With the same FLOPs, a model with a high degree of parallelism may be much faster than another model with a low degree of parallelism [34], [35]. Another important factor arises from the memory access cost. Since the on-device RAM is usually limited, data cannot be entirely
| Metrics | Characteristics | Advantages                                                                 | Limitations                                                                 |
|---------|----------------|----------------------------------------------------------------------------|----------------------------------------------------------------------------|
| **Device-agnostic Efficiency Evaluation Metrics** | **FLOPs** | • each to obtain  
• coarse approximation of latency | • inconsistent definitions  
• omitting the degree of computational parallelism  
• omitting the memory access cost  
• omitting the implementation library |
| | **Number of Parameters** | • easy to obtain  
• precise when measuring storage requirement | • omitting the computation caching for measuring memory  
• unable to reflect peak memory usage |
| **Device-aware Efficiency Evaluation Metrics** | **Latency** | • the real criterion that we care about  
• obtain through implementation on real hardware or prediction models  
• depend on different hardware platforms and implementation libraries | |
| | **Peak Memory Usage** | • the real criterion that we care about  
• obtain through implementation on real hardware or prediction models  
• reflect the peak usage that really matters | |
| **Effectiveness Evaluation Metrics** | **Accuracy/mAP/mIOU etc.** | • diverse and dependent on the targeting tasks  
• identical to those for evaluating normal deep learning models | |
| **Design Automation Cost Metrics** | **GPU Hours** | • used to evaluate the speed of a design automation method  
• depend on the number of GPUs used | |
| | **GPU Memory** | • used to evaluate the memory requirement of a design automation  
• grows linearly w.r.t. the size of the candidate set | |
loaded at one time and thus reading data from external memory is required. It has an increasing impact on the latency as the computation unit is getting stronger recently, thus becoming the bottleneck for latency. This intrinsic factor should not be simply neglected. Third, the implementation library of a deep learning model significantly influences its latency as well. For example, NVIDIA's cuDNN library provides different implementations of a convolution operation \[46\] that clearly require different amounts of FLOPs although the network architecture is identical. Some works also found that a smaller number of FLOPs could be even slower due to the library abstractions \[12\], \[13\].

2) Number of Parameters: It is usually required to fit all parameters of a neural network within on-chip memory to execute the model fast \[37\]. Thus, the number of parameters mainly constrains the memory requirement of a deep learning model. Although some mobile devices like smartphones have abundant memory, there are still a few mobile devices that have particularly scarce memory, such as microcontrollers that typically have 10's-100's of KB \[38\]. These memory-scarce devices are in demand in many fields due to their low prices. Thus, it is desirable to design a small-sized model that can fit into ‘tiny’ memory and the number of parameters is a common device-agnostic metric for evaluating such a requirement. Nevertheless, the number of parameters only accounts for a portion of memory usage; input data, computation caching (i.e., intermediate tensors produced at runtime), and network structure information take over a relatively larger portion of memory \[39\], \[40\]. In some extreme scenarios, only computation caching and active parameters (i.e., the parameters used for current computation) occupy memory. Some researchers propose to optimize the in-memory computation caching to reduce memory consumption but their performance largely depends on the network structure \[41\]. Thus, a model with a larger number of parameters is not certainly more memory-hungry than a model with a smaller number of parameters. On the other hand, model parameters account for a major component of storage/external memory (e.g., FLASH memory) usage.

B. Device-aware Efficiency Evaluation Metrics

Unlike device-agnostic metrics, device-aware metrics can reflect the computational cost that we really care about. They can be collected on real and target hardware platforms or approximated.

1) Latency: The latency is used to evaluate the running speed of a deep model making inferences. It is usually measured in the form of running time per inference (e.g., millisecond) \[42\] or inferences per unit time (e.g., batches/s or images/s) \[12\]. In practice, the precise latency is an average value computed on a large batch or several batches \[42\]. Some works collect this information through implementations on real devices including GPUs, TPUs, and CPUs \[25\], \[12\], \[43\], \[42\]. It should be noted that the reliability is unknown when evaluating a deep model’s latency not on its target mobile devices (e.g., smartphone CPUs) but on non-mobile devices (e.g., GPUs). In addition to directly measuring latency, some researchers try to approximate it \[23\], \[44\]. The lookup table is a latency approximation method that enumerates all the possible layers that a family of models can have along with the latency of each of the layers \[43\]. It is hardware and model family-specific and requires a significant amount of time to maintain a large and dynamic database. In addition, this simple summation of the latency of an individual does not take memory access cost and parallelism into consideration, and thus shows low precision. A latency prediction model is another latency approximation method that models the network latency as a function of network structures and/or hardware parameters \[23\], \[45\], \[44\]. This approach can make latency differentiable to be directly involved in an objective function for gradient-based optimization \[23\].

2) Memory Usage: Different from latency where we care about the total inference time of a sample/batch, the peak memory usage during inference is our major concern \[37\], \[46\]. As long as the peak memory usage is lower than the memory capacity, a model is able to run on the device. It is not necessary to keep memory usage as low as possible. Extreme scenarios with maximal memory saving are considered. The peak memory usage is dominated by the intermediate tensors (so-called activation metrics), so a small model (i.e., a small number of model parameters) doesn’t guarantee a low peak memory usage. For example, at similar ImageNet accuracy (70%), even though MobileNetV2 \[10\] reduces its model size by 4.6× compared to ResNet-18 \[47\], the peak memory requirement increases by 1.8× \[46\]. Thus, it is highly recommended to evaluate peak memory usage when targeting a device with quite constrained memory resources.

In addition to the peak memory usage, which considers the on-chip memory, storage/external-memory usage is also an essential evaluation metric. It mainly restricts the model size of which model parameters occupy the largest proportion. Therefore, the bit-precision of parameters has a crucial impact on the external-memory usage.

C. Effectiveness Evaluation Metrics

The metrics for evaluating the effectiveness of an efficient model are quite diverse and mainly dependent on the targeting tasks. Vision tasks, such as image recognition, object detection, and semantic segmentation, are such commonly used benchmark tasks \[12\], \[10\], \[48\]. Different tasks have different evaluation metrics: top-1 or top 5 accuracy for image recognition \[48\], mAP for object detection \[48\], and mIOU for semantic segmentation \[10\]. In addition to vision tasks, audio tasks, such as keyword spotting, are leveraged as an evaluation task \[46\]. However, accuracy is also used as the evaluation metric in this case.

D. Design Automation Cost Metrics

As the price of freeing human efforts, design automation normally demands an excessive computational cost that prohibits its wide deployment.
1) **GPU Hours:** GPU hours/days are metrics used to evaluate the time cost of a design automation method especially for a NAS-based method [49], [50]. The GPU days can be defined as:

\[
\text{GPU days} = N \times t, \tag{1}
\]

where \( N \) denotes the number of GPUs, and \( t \) denotes the number of days that are used for searching [51]. GPU hours have a similar definition. At the early stage, it requires several or even tens of days for searching [52], [49], while currently researchers have pushed the time to the magnitude of multiple hours [53].

2) **GPU Memory:** Although differentiable neural architecture search has reduced the cost of GPU hours considerably, it suffers from an intensive GPU memory cost. The consumption of GPU memory depends on the size of the candidate set for searching. Specifically, the required memory grows linearly w.r.t. the number of choices in a candidate set [54]. This issue restricts the search space size that prevents the capability of discovering novel and strong models. Some works have targeted this issue and achieved impressive progress [23]. Thus, it is important to involve GPU memory for a thorough evaluation.

### III. Search for Efficient Deep Learning Models

Driven by the growing demand for mobile applications, efficient deep learning models have gained explosive attention. A tremendous number of studies have been proposed to explore manually designed efficient neural architectures or modules and achieved impressive progress [29], [12], [10], [55], [56]. Though the notable success, it is challenging for human engineers to heuristically exhaust the design space to trade off accuracy and hardware constraints. Hardware-aware neural architecture search plays an influential role in advancing this field as it automates the design process to find an optimal solution. Similar to regular NAS [57], the hardware-aware NAS also has three components, i.e., search space, search strategy, and performance estimation strategy, but with additional freedom or constraints (Fig. 1 left bottom). In this section, we review recent achievements from these three aspects and summarize the main results in TABLE 1.

#### A. Efficient Search Space

A search space is the basis of NAS and determines what architectures NAS can discover in principle and their performance upper-limit [57]. A well-defined search space can not only accelerate the search process but also promote the searched model’s performance [49], [58]. Since some manually explored efficient neural architectures have achieved considerable advances, it is an intuitive yet practical idea to construct a search space with the heuristics of these handcrafted efficient structures, and leverage the NAS technology to automatically discover novel efficient models upon this search space. There are usually two components that define a search space: (i) operators that each layer executes, and (ii) a backbone that decides the topological connections of these layers.

#### 1) Operators: Operators, such as convolutional layers or fully connected layers, are basic building components for a deep learning model. A simple way to build a deep learning model is to directly stack several operators, such as VGG Net [59]. Other work connects several neural network layers to construct a motif (e.g., a residual block) and builds a model by repeating and arbitrarily connecting these motifs [60], [47], [57]. In this paper, we use the term "operator" to also represent a motif, which is usually used as a whole and regarded as a basic component of an end-to-end model.

As early-developed and widely-demonstrated neural architectures, the convolutional layer and its variants are the dominant operators of many deep learning models, especially those in the computer vision area. A standard convolution can be denoted as:

\[
\text{Conv}(W, X)_{(i,j)} = \sum_{m,n,k} w_{(m,n,k)} \cdot x_{(i+m, j+n)}, \tag{2}
\]

where \( W \) is the convolution kernel weight, \( X \) is the input feature maps to a convolution layer, \((i,j)\) is the coordinate of an output feature map, and \((m,n,k)\) is the coordinate of the convolutional kernel. NSGA-Net [61] utilizes standard convolutions as the operator and devotes to automatically determining their connections in a block-based manner. As it only seeks to optimize the connection of operators, limited efficiency is gained. Scheidegger et al. [62] and Lu et al. [63] also consider standard convolutions but, instead of optimizing the connections, they allow to search the convolution hyper-parameters such as the filter size, the stride and the number of filters.

In addition to the standard convolutions, extensive manually-designed variants have been demonstrated both efficient and effective. Therefore, it is intuitive to employ these architectures to construct an efficient search space. The depthwise separable convolution (DSConv), which can significantly reduce the number of parameters and computation and works as the primary module in MobileNet [9], is such a widely used efficient operator. As shown in Fig. 2a, a DSConv block is made up of two components: a depthwise convolution and a pointwise convolution. The depthwise convolutions (DWConv) applies a single filter per each input feature map (input depth) for spatial filtering:

\[
\text{DWConv}(W, X)_{(i,j)} = \sum_{m,n} w_{(m,n)} \cdot x_{(i+m, j+n)}. \tag{3}
\]

The pointwise convolution (PWConv), a simple \( 1 \times 1 \) convolution, is used to create a linear combination of the output of the depthwise convolutions for feature fusion:

\[
\text{PWConv}(W, X)_{(i,j)} = \sum_{k} w_k \cdot x_{(i,j)}. \tag{4}
\]

Thus, by combining DWConv and PWConv, the DSConv is denoted as:

\[
\text{DSConv}(W_p, W_d, X)_{(i,j)} = \text{PWConv}(W_p, \text{DWConv}(W_d, X)_{(i,j)}). \tag{5}
\]
LEMONADE [64] adopts both DSConvs and standard convolutions as the basic operators of its search space and supports increasing the number of filters and pruning filters to search efficient neural architectures. RENA [65] also includes DSConvs but allows to automatically decide not only their hyperparameters but also whether they should be used in each layer. In addition to the standard DSConv, ProxylessNAS [23] includes a variant, namely dilated depthwise separable convolution, in its search space as well. The search engine can choose between the standard one and the variant.

In MobileNetV2, a more advanced mobile convolution block, mobile inverted bottleneck convolutions (MBConv), is proposed [10] and soon becomes a popular operator in favour of an efficient search space. Fig. 2b illustrates the structure of MBConv. It is made up of three convolutions and one residual connection. First, a PWConv is applied to expand the input feature map to a higher-dimensional space so that non-linear activations (ReLU6) can better extract information. Then, a depthwise convolution is performed with $3 \times 3$ kernels and ReLU6 activations to achieve spatial filtering of the higher-dimensional feature maps. Furthermore, the spatially-filtered feature maps are projected back to a low-dimensional space with another pointwise convolution. Since the low-dimensional projection results in loss of information, linear activation is used after pointwise convolution. Finally, an optional residual connection (depending on whether the stride of the depthwise layer is 1) is added to combine the original input and the output of the low-dimensional projection. Note that the last two convolutions (depthwise convolution and pointwise convolution) are essentially a DSConv with dimension reduction. There are multiple works on purely using MBConv to construct an efficient search space but with different backbones or search strategies [65], [67], [68]. Xiong et al. [69] argue that DSConv is inexpensive based on FLOPs or the number of parameters, which are not necessarily correlated with the inference efficiency, so they propose a fused MBConv layer that fuses together the first pointwise convolution and the subsequent depthwise convolution into a single standard convolution. They achieve higher mAP and lower latency on EdgeTPU and DSP than the pure MBConv search space. Li et al. [70] provide an in-depth comparison between fused MBConv and MBConv and summarize that fused MBConv has higher operational intensity but higher FLOPs than MBConv depending on the shape and size of filters and activations. Therefore, they add both fused MBConv and MBConv into the search space to let the search strategy determine automatically.

In a more popular manner, MnasNet [48] upgrades MBConv with a squeeze and excitation (SE) module [71] after the DWConv for attentions on feature maps (as shown in Fig. 2c). Specifically, the S (squeeze) procedure first converts each individual feature map into a scalar descriptor using global average pooling so the input feature maps are converted into a vector $z \in \mathbb{R}^n$ with its k-th element calculated by:

$$z_k = \text{GlobalArgPool}(x_k) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} x(i, j, k),$$

where $x(i,j)$ is the $(i,j)$-th element of the $k$-th input feature map, which is of size $H \times W$. The E (excitation) procedure converts the S procedure’s output $z$ into a vector of activations $s$ using the gating mechanism of two fully connected layers:

$$s = \sigma(W_2 \cdot \text{ReLU}(W_1 \cdot z)), $$

where $\sigma(\cdot)$ is the sigmoid activation function. The final output of the SE module is $\tilde{X}$ with its $k$-th feature map denoted as:

$$\tilde{x}_k = \text{SE}(x_k) = s_k \cdot x_k.$$
and MBConv+SE) in the search space and let the search strategy choose automatically [81], [48]. The commonly used searchable parameters of the convolution operator family are kernel sizes, the number of output channels, expansion ratios (MBConv, MBConv+SE), and SE ratio (MBConv+SE). Although different works do not have exactly the same searching ranges, they are basically similar. For example, Stamoulis et al. [66] consider kernel sizes of {3, 5} and expansion ratios of {3, 6}; Fang et al. [67] consider kernel sizes of {3, 5, 7} and expansion ratios of {3, 6}; Cai et al. [77] consider kernel sizes of {3, 5, 7} and expansion ratios of {3, 4, 6}.

Besides the above convolution operators of the MobileNet family, the group convolution (GConv) and its variants [82], [83], [11] are also considered to be significant operators for constructing an efficient search space [84], [85], [86], [87]. Fig. 2d illustrates the structure of GConv, where we construct an efficient search space [84], [85], [86], [87], [88]. The feature maps and filters are divided into $G$ (in Fig. 2d) groups respectively: $X = \{X_1, X_2, ..., X_G\}$ and $W = \{W_1, W_2, ..., W_G\}$. In GConv, the convolution is only performed within each group so the output $\tilde{X}$ is denoted as:

$$\tilde{X} = \{W_1 \otimes X_1, W_2 \otimes X_2, ..., W_G \otimes X_G\},$$

(9)

where $\otimes$ is the convolution operation between two sets. In this way, GConv can not only reduce parameters and computation but also provide a simple way to model parallelism. Therefore, AlexNet [83] can be trained on multiple GPUs with only 3GB RAM each. Note that depthwise convolution is a special case of GConv with the number of groups being the same as the number of channels. A channel shuffle usually comes after GConv to enable inter-group communications [11]. Xu et al. [84] include GConv in their search space with the number of groups as searchable from 1 (standard convolution) to N (the number of input channels). FBNet [86] and RCAS [87] encompasses a new convolution operator in their search space by replacing the first and last pointwise convolution in MBConv with $1 \times 1$ GConv. This design expands MBConv but also allows the search strategy to automatically determine whether this expansion is needed. However, their allowed maximum group amount is quite small (2 for FBNet and 4 for RCAS). DPP-Net [85] and MONAS [88] also contain a variant of GConv, Learned Group Convolution (LGConv), which is the key operator of CondenseNet [89], in their search space. The LGConv prunes away unimportant filters with low magnitude weights further reducing the computational complexity on top of the GConv [82].

Some other studies use customized operators or non-convolution operators in their efficient search spaces. FasterSeg [90] proposes a zoomed convolution, where the input is sequentially processed with bilinear downsampling, standard convolution, and bilinear upsampling. The authors demonstrate that the zoomed convolution has 40% latency trim compared to a standard convolution on a GTX 1080i GPU. Different from previous works, which focus on 2D processing, Tang et al. [91] target 3D scenes. They propose sparse point-voxel convolutions and allow to search for channel numbers and network depth. However, as the computation of 3D CNN increases more significantly with increasing kernel sizes than 2D CNN, the authors keep the kernel size as a constant of 3. HR-NAS [92] also involves Transformer [93], [94] in addition to convolutions due to its recent success in computer vision [95], [96]. The authors design a lightweight Transformer that requires less computation when facing high-resolution images. Convolutional channels and Transformer queries are progressively reduced during the search. In order to facilitate high parallelism of convolution operators on TPUs and GPUs, Li et al. [70] add space-to-depth/batch into the search space to increase the depth and batch dimensions. The results show that even without the lowest FLOPs, their searched EfficientNet-X models are the fastest among compared model families on TPUs and GPUs. The main reason is that EfficientNet-X models strike a balance between FLOPs (lower is good for speed) and operational intensity (higher is good for speed).

In addition to parametric operators, non-parametric operators are critical components of an efficient search space as well. Pooling is usually coupled with convolutions in handcrafted CNN model family, and so does in convolution-based search space [87], [80]. It can help to reduce redundant information (favourable for accuracy) and computation (favourable for speed) without requiring additional parameters. Skip is another widely adopted operator that impacts the topology of achieved models. It can have two concepts: 1) drop skip directly feeding input to output without any actual computations, i.e., the entire layer is dropped and thus the model depth is reduced [66], [80], [67]; 2) residual skip providing an identity residual connection in parallel with another computation operator [64], [80]. The residual connection can be achieved by either concatenation or by addition [64]. Activation functions are another searchable non-parametric operators that impact both accuracy and speed [87], [70]. The computation cost of activation functions decreases when going deeper since the resolution of feature maps decreases [72]. In some research, the non-linearity is associated with convolution operators as a whole, and the operator together with its activation function is determined automatically [79].

2) Backbones: After defining the operators that a model can have, it is essential to determine how many operators there are and how these operators are connected, i.e., the backbone of a model. However, operators and backbones are not completely isolated, such as the skip operator, which removes layers or adds connections and can change a model's backbone. Depending on the connection topology, backbones can be roughly classified into two categories: chain-structured and multi-branch.

Chain-structured Backbones. The chain-structured backbone [57] is likely the earliest and simplest topological structure of a neural network. It directly stacks multiple operators in sequence. As shown in Fig. 3(a), the $i$th operator ($O_i$) takes the output of the $i$-1th operator ($O_{i-1}$) as input and its output serves as the input of the operator ($O_{i+1}$). Therefore, the topological connection of different operators is determined and not searchable. However, it is possible to search how many operations there are, e.g., via the drop skip. This simple backbone is a common practice of many handcrafted CNN models [83], [11], [9], [10].

An intuitive implementation is to stack operators layer
by layer [87], [77]. Motivated by the design principle of existing success [10], [11], the chain-structured backbone is more commonly implemented in a cell-based, a.k.a. block- or stage-based, manner [68], [81], [79], [91], [86], [67], where a backbone is composed of multiple chain-structured cells. The operators in the same cell can be either identical or diverse in hyperparameters but the same in type. In the sequence of cells, it is a usual principle that the input resolutions are reduced and widths are increased gradually. To achieve the reduced resolutions, the first or last operator in a cell usually has manually set strides and widths. This cell-based implementation is not only effective but also reduces the search space. Wu et al. [86] design a chain-structured backbone consisting of four searchable cells. There are 8 candidate convolution operators with various expansion rates, kernel sizes and numbers of groups, and a drop skip operator for searching. Different cells are separated by their input resolutions and widths, which are determined by manually set parameters. Yan et al. [81] also follow the same practice to construct their backbone but with more flexibility in searchable parameters. In these studies, it is usually required to predefine some parameters of the backbone, like the number of cells.

In comparison with setting backbone structure empirically, some work directly embraces the backbone of existing models as a starting point and then searches beyond that [76], [69], [88], [23], [66], [69]. This design principle relieves experts’ burden of search space design by reusing prior knowledge, which is important in NAS [97]. MONAS [88] relies on the backbone of a simplified version of AlexNet [83] to search the convolution filter sizes and amounts. Scheidegger et al. [62] investigate the backbone of MobileNetV2 [10] and allows to search the number of operators in each cell but all operators in a cell have the same settings except for the stride, which is used to modify the output resolution. The complexity reduction is mainly obtained by lowering the channel widths and reducing the number of topological repetitions. In addition to following the backbone of hand-crafted models, there is a trend to using the backbone of existing searched efficient models. MOGA [76] adopts MobileNetV3-large [72] as its backbone and keeps the same number and type of operators. It only searches the parameters of operators, like kernel sizes, expansion ratios for MBConv, and whether SE is enabled or not. Cai et al. [77] also adopts MobileNetV3, but they additionally provide the flexibility of searching the number of operators in each cell.

**Multi-branch Backbones.** While the chain-structured backbone is simple, it restricts the information flow to be sequential and single-path. Current hand-crafted architectures have suggested that a multi-branch architecture, which allows multi-path information flow and residual skip connections, works impressively better than a chain-structured architecture [98], [47]. As illustrated in Fig. [3](b), in the multi-branch backbone, an operator $O_i$ is allowed to accept the outputs from some of its previous operators (i.e., $O_1$, $O_2$ ... $O_{i-1}$) as the input but not necessarily takes all. This setting provides more degrees of freedom on network topology. Note that a chain-structured backbone with residual skip connections, where an operator $O_i$ must receive the output of its immediate previous operator $O_{i-1}$, is a special case of the multi-branch backbone.

The multi-branch backbone can be achieved by using an **insertion** operation in the search space [65]. However, searching for a multi-branch backbone as a whole is time-consuming and difficult to find an optimal structure. Similar to the chain-structured backbone, the cell-based principle is also widely adopted in designing a multi-branch backbone [80], [61], [68].
where the topology within a cell has a multi-branch structure. This results in a **Macro-micro Backbone** (Fig. 3 (c)) \[48, 61, 92, 84\]. In the macro-structure, *residual skips* are optional to provide the flexibility of multi-path information flow; in the micro-structure, operators are connected in the multi-branch fashion with the last operator (either parametric or non-parametric) assembling a single output of the cell. For example, MnasNet \[48\] predefines a backbone of 7 sequentially stacked cells and each cell contains sequentially stacked operators with a searchable amount, types, and parameters. It further allows searchable residual skip connections among operators within a cell. This multi-branch backbone simply augments the chain-structured backbone with residual skips, resulting in more flexibility of multi-path information flow but marginal research space expansion. NSGA-Net \[61\] and HR-NAS \[92\], in contrast, allow more general multi-branch connections within a cell. NSGA-Net supports searchable topology while HR-NAS sets fixed multi-branch connections and searches for discarded convolutional channels and transformer queries.

Different from searching for a customized structure, following existing success is also a favoured choice in designing multi-branch backbones \[70, 31, 80, 72\]. Li et al. \[70\] adopt the multi-branch backbone of the EfficientNet and only search operators. MnasFPN \[80\] constructs its search space based on the NAS-FPN(Lite) backbone and searches both multi-branch structures and operators in a cell for merging various resolutions. However, to reduce the search burden, it does not allow general connectivity patterns and applies limited merging connections. MobileNetV3 \[9\] uses the backbone of MnasNet \[48\] as the seed and proceeds layer-wise search on it. Scheidegger et al. \[62\] investigate the backbone of several existing models (DenseNet121 \[60\], MobileNetV2 \[10\], GoogLeNet \[99\], PNASNet \[100\], and ResNetXt \[82\]) and demonstrate that their approach is able to provide improved accuracy with hardware constraints and different backbones. While considering that searching both macro and micro structures overburdens the searching process, most works utilize the multi-branch macro backbone of existing models and only search the micro-structures \[23, 85, 88\]. ProxylessNAS \[23\] accepts the backbone of the residual PyramidNet \[101\], which has a residual skip connection every two operators, and replaces the original operators with their own tree-structured cells \[102\]. DPP-Net \[85\] selects the backbone of CondenseNet \[89\], which repeats an identical cell abundant times with both residual skip and chain connections, and only searches the operators in the cell. MONAS \[88\] also reuses the backbone of CondenseNet but it uses the same cell structure and searches the number of stages and growth rate.

Although the macro-micro backbone provides the highest flexibility and complexity among the above three categories, there is no evidence that it is the best choice. However, we note that no matter in which backbone category, cell-based implementation is the most common practice. This is due to the following considerations: i) an effective network usually has gradually shrinking resolutions as going deep, and a cell can have multiple operators on the same resolution to strengthen feature extraction; ii) previous experience with the manually designed network indicates that the cell-based structure is effective for deep learning; iii) the cell-based backbone provides high search efficiency by limiting the search space (e.g., the whole network can stack the same cell and a cell can have the same operator); iv) the cell-based backbone simplifies the network topology with inter- and intra-cell connectivity instead of random edges among all operators. Some recent works reveal that search performance may be impeded by these search space design biases \[103\]. Nevertheless, the choice of the search space principally regulates the difficulty of the search process. Current NAS algorithms are imperfect so more freedoms in the search space do not always lead to better resultant models but inversely overburden the search algorithms. With the continued improvement of NAS algorithms, it is essential to reduce design biases and construct a more general search space.

### B. Search Strategy

The search strategy describes how to explore a search space to find an optimal efficient network. Essentially, the search process is to find the top candidate networks regarding some evaluation metrics (e.g., accuracy and latency). However, it is computationally prohibitive to achieve these metrics of all candidate networks since it requires fully training a network to obtain its metrics. Therefore, the aim of a search strategy is to efficiently find top-ranking networks without exhaustively examining all candidate networks. In this section, we first conclude the *search algorithms* that describe how the search proceeds and then summarize how the hardware constraint is incorporated into the search process.

1) **Search Algorithms**: The most intuitive and easiest search algorithm is the *grid search*, which exhaustively explores the search space and evaluates every possible architecture. This approach works well for a small space \[68\] but is terribly inefficient for a large space due to the exponentially increased number of evaluations. Another problem with this type of method is that it only supports a bounded and discrete space and needs careful selection of the grid interval. *Random search* \[104\], on the other hand, samples neural architectures randomly from the search space. It can be used not only for a discrete space but also for a continuous space with a predefined distribution. This algorithm is superior to the grid search also when the search dimensions (e.g., kernel sizes, expansion ratios, and network depths) have different effects on the final performance. Random search and its simple variants are usually used when the performance of a candidate model is easy to obtain \[83, 75\]. Advanced search algorithms, like reinforcement learning and evolutionary search, are widely demonstrated better performance than random search \[49, 105, 106\]. In the following, we summarize the advanced search algorithms, including Bayesian optimization, evolutionary search, reinforcement learning, and differentiable methods, regarding design automation for efficient deep learning models.

**Bayesian Optimization (BayesOpt)** is an important approach for automated hyperparameter tuning and architecture search. Given a search space \( \mathbb{S} \) that contains a large set of
Evolutionary search
Differentiable search
Reinforcement learning

evaluation metric (e.g., accuracy), the goal of BayesOpt is to find an architecture \( a \in S \) that maximizes \( f(a) = \arg\max_{a \in S} f(a) \). However, it is expensive to achieve \( f(a) \) because it requires fully training the architecture \( a \) from scratch. Therefore, a statistical model, which is invariably a Gaussian process (GP), is used as a surrogate model. The more neural architectures (i.e., points of \( f(\cdot) \)) are evaluated, the less uncertainty the surrogate model has. Another important component of BayesOpt is the acquisition function for deciding which architectures are sampled and then evaluated at each iteration, which is often expected improvement (EI):

\[
EI(a) = \mathbb{E}_{\mathcal{N}}(f(a) - f(a^*), 0),
\]

where \( a \) is a sampling architecture and \( f(a^*) \) is the evaluation metric of the best architecture \( a^* \) so far, that is until \( t \)-th exploration. The general process of BayesOpt is shown in Fig. 4a, where multiple architectures are first randomly sampled from the efficient search space; the sampled architectures are then trained and evaluated for updating a GP model; the acquisition function is computed based on the updated GP model to decide which neural architectures should be sampled next; the process proceeds iteratively until certain conditions are met. The best architecture among all sampled architectures is usually output as the final model.

When additionally considering the hardware constraints, Multi-Objective Bayesian Optimization (MOBO) is commonly used \([107],[108],[109],[110]\), where GP is fitted for each objective independently, and Pareto-frontier is identified as the set of optimal trade-off solutions of the multiple objectives. Different MOBO approaches mainly differ in how to achieve the Pareto-frontier during the acquisition process. For example, Parsa et al. \([109]\) use a Gaussian distribution to estimate the Pareto-frontier function; Eriksson et al. \([110]\) use the Noisy Expected Hypervolume Improvement (NEHVI) acquisition function, which is a noise-tolerant and extended version of EI for the multi-objective setting, to sample intermediate Pareto-frontiers. Although BayesOpt has shown promising performance, especially in hyperparameter optimization, it is computationally expensive and challenging when handling a high-dimensional search space and multiple objectives \([111],[112]\).

**Evolutionary Search (ES)** is a long-thriving NAS approach, which is based on the concept of biological evolution. It manages to find an optimal solution by iteratively improving upon a population of candidate solutions according to a fitness function. As illustrated in Fig. 4b (the schematic of the basic ES process), a population of candidate networks is first initialized by randomly sampled from the efficient search space; then evolutionary operations, such as crossover (combination of two parents) and mutation (arbitrarily mutating operators with new ones from the search space), are applied to current population to generate next generation; lastly, all offsprings need to be tested against a fitness function, which is to select the most powerful candidate networks and update the population; the process proceeds repeatedly until stopping criteria are met. The best network in the last population is the final achieved model. The major benefit of evolutionary search is its flexibility in directly controlling the offspring generation process and population updating process \([105]\).

The hardware constraints are usually considered in the fitness function of ES in two ways: hard-constraint and soft-constraint, which we will present concretely in the flowing section III-B2. Thus, most studies directly use existing evolutionary algorithms with customised fitness function to incorporate hardware constraints \([78],[91],[81],[113],[77],[114],[105],[115]\). For example, FBNetV3 \([78]\) uses the adaptive genetic algorithms \([116]\) to realize adaptive probabilities of crossover and mutation, and customizes the fitness function with its proposed **accuracy predictor** and hard hardware constraints; OFA \([77]\) adopts the **regularized evolutionary search** \([117]\), which introduces an age property to favor younger generation during search; DONNA \([113]\), FairNAS \([118]\), and MOGA \([76]\) appeal to the famous NSGA-II \([119]\), which is a multi-objective evolutionary algorithm, to find the Pareto-optimal solution instead of the solution under hard constraints in previous studies. In contrast to directly using off-the-shelf ES algorithms, some researchers develop ES algorithms specific to the hardware efficient application \([61],[62],[64]\). LEMONADE \([64]\) is such a representative work, which handles various objectives differently considering that different objectives have different evaluation costs. Specifically, the
efficiency objective (e.g., FLOPs) is cheap to evaluate while the accuracy objective is much more expensive as it requires fully training the model at each iteration. The authors propose to first select the architectures that fulfill the Pareto front for the new generation and then only train and evaluate these networks. As traditional ES can only sample a limited amount of networks that satisfy hardware constraints, Scheidegger et al. [62] use ES algorithms to search sampling laws that can better cover the sub search space under specific constraints. Thus, better Pareto frontiers can be achieved. NSAG-Net [61] designs an additional exploitation step after the traditional evolutionary operating to learn and leverage the history of evaluated populations. Since the evaluated populations at each iteration rank relatively high regarding the fitness score, it is quite possible that the optimal model has similarities to the evaluated populations and thus extracting common patterns from the evaluated populations can accelerate convergence. Although the above studies manage to lighten the search process from the algorithm perspective, it is still costly due to the requirement of training each network of the new generation at each interaction.

**Reinforcement Learning (RL)** has achieved notable success and grabbed great attention in the NAS community since Zoph and Le’s work in 2017 [52]. The general framework of RL for NAS is illustrated in Figure 4c: the controller, usually an RNN network, is the core component, which generates actions to sample operators or hyperparameters from the efficient search space; a neural network will then be constructed using the sampled options and trained and evaluated to achieve a reward, which will be used to update the controller using a policy gradient method [120]. This process will be performed repeatedly until stopping criteria are met.

The hardware information is often considered in the RL reward function. Some studies thus directly use the existing RL-based NAS approach [52], but optimize with a customized reward function [88], [65]. Furthermore, most recent work mainly follows the advanced RL search algorithms in two representative studies, MnasNet [48] and TuNAS [106]. Specifically, [80], [72] follow MnasNet, and [69], [31] follow TuNAS. Similar to [52], MnasNet [48] also uses RNN as the controller while maximizing the expected reward using an advanced policy gradient method, Proximal Policy Optimization (PPO) [121], to alleviate the high gradient variance of the vanilla policy gradient method. TuNAS is based on ENAS [122] and ProxylessNAS [23]. It improves with warmup and channel masking techniques for better search robustness and scalability. Instead of building one candidate network at each training step, TuNAS encompasses all candidate networks into a supernet. Each path of the supernet represents a candidate network. In addition, TuNAS is not a sequential decision-making process and does not rely on an RNN controller; alternatively, it uses learnable probability distribution that spans over architectural choices as the RL controller. At each step, a candidate network is sampled from the distribution, and then the portion of the supernet correlated with the sampled network is trained; a reward is calculated with the sampled network to update the probability distribution i.e., the RL controller.

**Differentiable Search** is the most recently developed search paradigm that finds an optimal network by gradient descent [54]. As shown in Figure 4d, a network is as a directed acyclic graph consisting of an ordered sequence of $N$ nodes ($N = 4$ in Figure 4d). Each node $X_i$ is a latent representation and each directed edge represents a candidate operator $O_j$ that is applied to $X_i$. For example, in Figure 4d there are three candidate operators (i.e., $O_1$, $O_2$, and $O_3$) between node $X_0$ and $X_3$. In this way, the search process is formulated as an optimal path-finding problem. All outputs of the candidate operators from a node’s predecessors are summed with weights $\alpha_i$ to achieve the node representation: $X_j = \sum_{i=1}^{m} \alpha_j O_i(X_i)$, subject to $\alpha_i \geq 0, \sum_{i=1}^{m} \alpha_i = 1$. In Figure 4d edges with darker colours indicate larger weights. The weights $\alpha_i$ can be represented with a softmax function of architectural parameters $\beta_i$: $\alpha_i = \frac{\exp(\beta_i)}{\sum_{j=1}^{m} \exp(\beta_j)}$. The network weights ($w$) and architectural parameters ($\beta_i$) are trained alternatively with training data and validation data, respectively. This induces a bi-level optimization problem [54]:

$$\min_{\alpha} \min_{w, w'_{\alpha}} L(\alpha, w, w'_\alpha). \tag{11}$$

Therefore, unlike ES and RL, a differentiable search unifies model training and search into a joint procedure. As the whole network contains multiple parallel operators in each layer, it is called a supernet, and its subnet with one operator in each layer is a candidate network. The candidate operator with the highest associated weight is chosen to construct the final model. The training of the supernet is also the search process.

The hardware information can be considered in the loss function and optimized when training the architectural parameters. Various differentiable search algorithms are proposed to achieve better search efficiency and performance [79], [86], [23], [66], [92], [67]. FBNetV1 [86] replaces the softmax function with the Gumbel softmax function [123] to better represents the subnet sampling process, and thus reduce the performance gap between the supernet and subnet. HR-NAS [92] progressively discards the paths with low path weights during the search to increase the search efficiency. DenseNAS [67] splits the search procedure into two stages: the first stage optimizes the model weights only for enough epochs and the second stage alternatively optimizes the model weights and architectural parameters. This strategy alleviates the bias of fast convergence operators. Considering the small size of the search space of the conventional differentiable search due to its requirement of loading the whole supernet into memory, many studies propose novel differentiable search algorithms to reduce the memory footprint [23], [66], [79], [67]. ProxylessNAS [23] proposes to factorize the task of training of all paths into training two sampled paths, which have the highest sampling probabilities, and discard the other paths temporarily at each iteration. The architectural parameters of the two selected paths are updated during training and then rescaled after training to keep the path weights of unsampled paths unchanged. In this way, the memory requirement is decreased to the level of training two subnets. Different from previous papers, which use multi-path supernets and thereby have memory issues, Single-Path NAS [66] and FBNetV2 [79] develop masking mechanisms that train a supernet with the largest
hyperparameters (e.g., $7 \times 7$ kernel) and a mask/indicator function that determines whether to just use a small part (e.g., $5 \times 5$ or $3 \times 3$ kernel) of the largest hyperparameters.

Other types of search algorithms are also reported [87], [85]. Xiong et al. [87] propose a modified Cost-Effective Greedy algorithm for the submodular NAS process, where starting from an empty network, each block is filled iteratively with the highest marginal gain ratio regarding both accuracy and cost. DPP-Net [85] follows the progressive search algorithm in [100] to find the optimal operator layer by layer. Different from searching for a new model for each new resource budget, another way is to build a baseline model first and then scale it to obtain a family of models for various budgets [68], [70]. Specifically, they adopt the conventional RL-based hardware-aware search algorithm [48] to search for a small baseline model, and use a simple grid search to determine the best scaling coefficient (i.e., $\alpha$, $\beta$, $\gamma$) for network depth, width, and input size. A family of specialized models for different budgets can be obtained by scaling up the baseline model by $\alpha^N$, $\beta^N$, and $\gamma^N$ with $2^N$ times increased resource.

2) Hardware-constraint Incorporation Strategy: Despite some studies only using an efficient search space without considering hardware constraint [49], there are two strategies commonly used to consider hardware budget for finding optimal compact models. The first considers a specific hardware platform and treats its resources as hard constraint to build a specialized model that is the most accurate under the fixed constraint. In contrast, the second strategy is to directly search networks without considering specific hardware constraint. This strategy, dubbed soft-constraint incorporation, treats the model efficiency as an additional optimization objective and tries to find the Pareto frontiers. In this section, we summarize the design automation techniques from the above two perspectives.

Hard-constraint Incorporation can be easily adopted by evolutionary search [31], [91], [77], [78], [62] since the offspring generation and selection step can be directly controlled. The incorporation process is quite straightforward: when a set of candidate networks is generated (e.g., by evolutionary operating), their hardware costs are then tested, and the networks that do not meet the constraint are simply discarded. This strategy is also easy to implement for other search algorithms, which generate a set of candidate networks one time or at each iteration, such as random search [84], [75], grid search [68], and greedy search [87]. The RL-based search algorithms can also embrace the hard-constraint incorporation strategy but in a different way [65], [48], [88]. The general idea is to have only the accuracy in the reward function when the constraint is met; otherwise, both accuracy and the hardware objective are considered in the reward function. A typical reward formula $\mathcal{R}(\cdot)$ is proposed in MnasNet [48]:

$$ \mathcal{R}(a) = \begin{cases} ACC(a), & \text{if } C(a) \leq C_0 \\ ACC(a) \times (C(a)/C_0)^\beta, & \text{if } C(a) > C_0 \end{cases} $$ (12)

where $ACC(a)$ and $C(a)$ denote the accuracy and hardware cost (e.g., latency in [48]) of model $a$, respectively, and $C_0$ denotes the target cost constraint. $\beta < 0$ is the only tunable hyperparameter that controls the convergence speed. In [106], the authors empirically find that if $\beta$ is too large, the RL-controller will prefer to sample the architectures whose cost is significantly smaller than the target cost constraint. This will result in suboptimal models regarding accuracy.

Soft-constraint Incorporation does not target any specific constraint and is realized by adding an additional optimization objective to the accuracy objective. Since these objectives are competing, no unique optimal solution exists in the multi-objective space. Thus, Pareto frontiers are sought, especially by multi-objective BayesOpt and multi-objective evolutionary algorithms, and one specific model can be identified according to different application requirements [61], [109], [113], [76], [107], [63], [108], [113], [118]. For example, ChamNet [115] designs a multi-objective fitness function:

$$ \mathcal{F}(a) = ACC(a) - [\alpha H(C(a) - C_0)]^\beta, $$ (13)

where $C(a)$ and $C_0$ denote the hardware cost of model $a$ and the target constraint, respectively, $H$ is the Heaviside step function, and $\alpha$ and $\beta$ are positive constants. The searching objective is to maximize the fitness function: $a^* = \arg \max_a(\mathcal{F}(a))$. This fitness function guides the search process to find a model approaching the hardware target but without a hard guarantee. Some work also pursues the soft-constraint objective under a hard constraint with a two-phase approach, which first filters out models that do not satisfy the hard constraint and then performs multi-objective optimization [85], [62]. Instead of finding a set of solutions, some studies attempt to find the best tradeoff between efficiency and effectiveness. Two schemes are commonly used in this regard and mainly for the RL-based and differentiable algorithms: multiplication and linear combination. For the multiplication scheme, the effectiveness objective (e.g., accuracy) and efficiency objective (e.g., latency) are multiplied to construct the loss function [66] or reward function [72], [48], [80], [70]:

$$ \text{loss function: } \mathcal{L}(a) = \mathcal{L}_{CE}(w|a) \times \log(C(a))^\beta, $$ (14)

$$ \text{reward function: } \mathcal{R}(a) = ACC(a) \times (C(a)/C_0)^\beta. $$ (15)

The $\mathcal{L}_{CE}(w|a)$ is the cross-entropy loss of model $a$ with parameter $w$. The exponent coefficient ($\beta > 0$) for the loss function; $\beta < 0$ for the reward function) modulates the tradeoff between effectiveness and efficiency. Note that although there is a target constraint term $C_0$ in the reward function, this is still a soft constraint since this just pushes the cost to be lower than the target but without any guarantee. Alternatively, it is possible to keep running the search process until the hardware constraint is satisfied [72]. For the linear combination scheme, the two competing objectives are linearly combined to construct the loss function [66], [67], [79], [92], [23] or reward function [69], [106], [31]:

$$ \text{loss function: } \mathcal{L}(a) = \mathcal{L}_{CE}(w|a)+\beta \log(C(a)), $$ (16)

$$ \text{reward function: } \mathcal{R}(a) = ACC(a) + \beta(C(a)/C_0 - 1). $$ (17)

Similar to the multiplication scheme, the $\beta$ coefficient balances the effectiveness and efficiency. The log($\cdot$) function in the
hardware-related term in the loss function is used to scale the cost and are omitted in some studies \cite{79,23,92}. TuNAS \cite{106} empirically demonstrates that search results are robust to the exact value of \( \beta \). It shows that the same \( \beta \) value works great for different search spaces and hardware constraints. This \( \beta \) value-invariance alleviates the tedious tuning of \( \beta \) for new scenarios (e.g., new devices).

For non-differentiable search algorithms, the hardware constraint can be smoothly incorporated; by contrast, differentiable search algorithms require the cost term to be differentiable, which is intrinsically not. Therefore, specific cost estimation strategies are expected to resolve this contradiction, which we will introduce in Section III-C2.

C. Performance Estimation Strategy

The search process is managed by search strategies, which we have discussed in Section III-B and guided by the performance of candidate models, which we will discuss in this section. The performance in the context of hardware-efficient models has two aspects: task-related performance (e.g., accuracy) and hardware-related performance (e.g., latency). The most natural way to obtain a model’s performance is to fully train the model and evaluate it on validation data for task-related performance and deploy it on the target hardware for hardware-related performance (for direct metrics, e.g., latency) \cite{107,61}. However, it demands massive computation and time to fully train each candidate model from scratch and deploy them on target hardware. Therefore, efforts are made toward alleviating the performance estimation process. We categorize and discuss these efforts into two routes: one is to speed up training the process (section III-C1), and the other is to directly predict the performance without training (section III-C2). Since the hardware cost does not require training, studies of estimating hardware-related performance are reviewed in Section III-C2 as well.

1) Training Strategy: The hardware-aware NAS does not distinguish from conventional NAS regarding the training strategies of candidate models so the training strategy summarized in this section can also be applied to conventional NAS and vice versa. The simplest training strategy is to train each candidate model from scratch and use the sample-eval-update loop \cite{52} to achieve an effective search agent. This strategy is not only widely adopted by RL-based search \cite{48,80,72,65,49}, but also by BayesOpt \cite{107,108,109}, evolutionary search \cite{61,64,62}, and other algorithms \cite{70,68}. To alleviate the drawback of the straightforward training approach, some studies \cite{80,48,87} follow Zoph et al. \cite{49} to first train and search on proxy tasks, such as smaller datasets or fewer epochs, then transfer to the large-scale target task. The proxy task is a reduced version of the target task. Xiong et al. \cite{87} also propose lazy evaluation to further allow fewer evaluation. However, this is specific to its search strategy. An inevitable shortcoming of using a proxy task is that the model optimized on proxy tasks is not guaranteed to be favourable on the target task.

Another way of speeding up the training process is parameter sharing \cite{124}. The search space is represented as a single super Directed Acyclic Graph (DAG), dubbed a supernet, whose nodes are computation operators and edges illustrate the flow of information. Each candidate model is a subgraph/subnet of the supernet and candidate models can share their parameters if they share common operators. When the supernet is well trained (e.g., trained for enough epochs), a candidate model can directly inherit parameters from the supernet without any separate training. A more widely used supernet is to have each node represent a latent representation (e.g., feature maps of a CONV operator) and each directed edge represents some operation (e.g., a CONV operator). There are multiple edges (i.e., different operators) between every pair of nodes, and the search process is to determine which edge/edges should be retained between each pair of nodes. The common retained edges share parameters between different candidate models. Different studies differ in training the supernet. Some work \cite{88,109,114} first pre-trains the whole supernet, and then only trains a part (e.g., a subnet) of the supernet, which is selected by search algorithms. Specifically, TuNAS \cite{106} disables the RL controller at the first 25% of the search and only trains the parameters of the supernet. It randomly selects CNN filters and candidate operators to train with some probability \( p \), which is linearly decreased from 1 to 0 over the first 25% search. During the parameter sharing, TuNAS further shares the “submodule” parameters. For example, MBConv operators with different DWSConv kernel sizes are in different paths/edges, but they can share the PWConv weights in MBConv regardless of which DWSConv kernel is selected. In addition, TuNAS also applies channel masking for parameter sharing; it creates a convolution with the largest possible number of channels while simulating smaller channel numbers by choosing the first \( N \) channels while zeroing out the remaining ones. These techniques are well recognized and followed by many researchers \cite{31,69}.

Differentiable NAS naturally uses a parameter-sharing training strategy. It jointly trains the supernet’s parameters and the importance weights of each path/edge \cite{86}. Furthermore, some researchers \cite{67,92} propose to progressively discard the paths/edges with low importance during training to further accelerate the training process. This progressive shrinking strategy speeds up the training process significantly. A defect of the differentiable parameter sharing strategy is that the supernet consumes high GPU memory, which would grow linearly with regard to the number of candidate models. To fill this gap, the single-path training strategy is proposed to reduce the memory cost to the same level as training a single candidate model. The conventional parameter sharing strategy has different operators on different paths, even though these operators are of the same type but with different hyperparameters (e.g., different kernel sizes). In light of this observation, Single-Path NAS \cite{66} designs a superkernel \( w_k \) for the DWConv inside a MBConv to choose between a \( 3 \times 3 \) or a \( 5 \times 5 \) kernel:

\[
w_k = w_{3\times3} + \mathbb{I} (||w_{5\times5\times3,3}||^2 > t_k) \cdot w_{5\times5\times3,3}.
\]

(18)

It views the \( 3 \times 3 \) kernel as the inner core of the \( 5 \times 5 \) kernel, while zeroing out the weights of the outer shell. The \( 5 \times 5 \) kernel can be viewed as the summation of this inner core \( w_{3\times3} \).
and the outer shell \( w_{5 \times 5 \times 3 \times 3} \): \( w_{5 \times 5} = w_{1 \times 3} + w_{5 \times 5 \times 3 \times 3} \). The choice of the kernel size can be done using the indicator function \( \mathbb{1}(\cdot) \in \{0, 1\} \) in Equation (18), which is relaxed to be a sigmoid function \( \sigma(\cdot) \) to compute gradients. The \( t_k \) is a learned threshold. Likewise, the NAS decision of the expansion ratio \( e \in \{3, 6\} \) of MBConv can be represented as:

\[
w_{k, e} = \mathbb{1}(\|w_{k,3}\|^2 > t_{e=3}) \cdot (w_{k,3} + \mathbb{1}(\|w_{k,63}\|^2 > t_{e=6}) \cdot w_{k,63}),
\]

where \( w_{k,3} \) is the channels with expansion ratio \( e = 3 \), and can be viewed as the first half of the channels of the MBConv with expansion ratio \( e = 6 \), while zeroing out the second half of the channels \( w_{k,63} \). The first indicator function in the Equation (19) is used to decide whether to skip the MBConv layer. This NAS problem can then be formulated as a common single-level optimization problem:

\[
\min_w \mathcal{L}(w|t_k, t_e),
\]

as opposed to the bi-level optimization in Equation (11). This optimization is solved in a differentiable way. The memory consumption of the single-path NAS is of the same level as the largest candidate model. Nevertheless, this single-path strategy only validates searching for different hyperparameters of the same type of operators, especially for convolution-based operators. ProxylessNAS [23], on the other hand, still relies on a multi-path structure but only activates two paths at each training iteration, which are sampled with the highest learnable probabilities, and mask out all the other paths. FBNetV2 [79] combines the single-path and multi-path strategies. It also considers the smaller number of channels as part of the larger volume of channels via vector masks, each of which has ones in the first entries and zeros in the remaining entries; different vector masks are selected with Gumbel softmax weights; in this way, only the largest set of filters needs to be trained. For the resolution search, the authors propose to subsample smaller feature maps from the largest feature map, perform convolution, and enlarge with inserted zeros to the largest size. This design resolves the resolution mismatch problem during single-path training. Although FBNetV2 is still a multi-path structure, it reduces the network paths with the single-path strategy thus accelerating the training and reducing the memory cost.

Different from the joint optimization of gradient-based methods, evolutionary search or RL-based search decouples the training and search process into two sequential steps. Thus, it is unnecessary to use indicator functions or learnable weights in the training. The single-path training strategy becomes simpler. For example, SPOS [105] proposes to uniformly sample a single path from a supernet and makes it well trained. As a result, all candidate models are trained evenly and simultaneously. This strategy alleviates the co-adaptation problem of shared weights [125]. After training, subnets can be directly sampled from the supernet for search evaluation. Many papers [81], [91], [118], [76] then follow this simple yet effective strategy, and design novel techniques to improve fair sampling and training of candidate models [91], [118], [76]. A comprehensive study is FairNAS [118], [76] that ensures the parameters of every candidate operator be updated the same amount of times at any training iteration. Specifically, similar to SPOS, it randomly samples a candidate operator at each layer to form a subnet but differently, the chosen operators are not put back at the next sampling step. The sampling process continues until all operators are sampled. The sampled models are trained individually with back-propagation, but their gradients are accumulated to update the supernet’s parameters. This strategy can alleviate the ordering issue [118], where candidate models are trained with an inherent training order in SPOS. Though SPOS and FairNAS accelerate the training process and reduce the memory cost, it is conventionally required to retrain the searched model before deployment because the inherited parameters from the supernet are not specialized for a specific model. Several studies [75], [84], [77] attempt to form a single-training strategy that does not require any post training after searching. BigNAS [73], [84] employs a bunch of training techniques (e.g., sandwich rule, in place distillation, and batch norm calibration) [126], [75] to achieve a high quality supernet so the inherited parameters can work well for any subnets. OFA [77] is a more promising strategy that prevents interference between subnets and thus a derived model can be directly deployed. It trains candidate models from the largest size (i.e., largest kernel size, depth, and width) to the smallest size (i.e., smallest kernel size, depth, and width) progressively. When training smaller subnets, the authors keep the last layers or channels untouched and finetune the early ones from shared parameters. As for elastic kernel size, the authors train kernel transformation matrices to transform the center of the larger kernel into the smaller kernel. The OFA strategy ensures each candidate model has a specifically trained or finetuned part so that mitigates their interference.

2) Performance Predictor: Another strategy to speed up the training process is to estimate the accuracy of candidate models using an accuracy predictor [77], [78], [85], [115], [113]. Although it requires substantial [model, accuracy] pairs thus computation and time to construct an accuracy predictor, it is a one-time cost as the predictor can be re-used for multiple hardware constraints. In addition, the accuracy predictor can also be easily finetuned for new datasets. The input into the accuracy predictor is the representation of candidate models, which is often one-hot encoding of candidate operators and hyperparameters [77], [85] or continuous values of hyperparameters (e.g., for channel counts) [78], [115]. A different study is DONNA [113], which uses block-quality metrics derived from blockwise knowledge distillation as the input into the accuracy predictor. For the predictor architecture, some studies use neural networks (e.g., MLP [78], [77] or RNN [85]) while others use conventional learning models (e.g., linear regressor [113] or Gaussian Process regressor [115]). ChamNet [115] additionally adopts Bayesian optimization for model sampling to achieve better sampling efficiency and reliable prediction.

In addition to an accuracy predictor, predictors for hardware cost are also widely studied to reduce the huge communication expense between the target device and the model training machine. Furthermore, the hardware cost predictor is demonstrated high fidelity across platforms \((r^2 \geq 0.99)\).
Therefore, the cost predictor is popular and necessary. A broadly investigated approach is the latency \textit{Lookup Table (LUT)} \cite{80,86,67,66,114,70,115,77}, which records the runtime of each operator in the search space. The basic assumption is that the runtime of each operator is independent of other operators \cite{86,23}, so that the latency of an entire model $a$ can be estimated as the sum of the latency of each individual operator $O_i$:

$$LAT(a) = \sum_i LAT(O_i). \quad (21)$$

Researchers profile the target hardware and record the runtime for each candidate operator to estimate the latency of candidate models. Some research also considers the connectivity of operators and the communication overheads in sequential layers \cite{67,114,80}. Another possible way is to construct a regressor to estimate the hardware cost based on critical features of a candidate model \cite{107,23,115,69}. The performance regressor can be a linear regressor \cite{69,31}, a Gaussian Process regressor \cite{115}, or a neural network \cite{107}. Different from directly measuring the hardware cost and training a black-box performance regressor, two papers examine specific devices and derive the runtime \cite{108} or power consumption \cite{109} through theoretical analysis. Nevertheless, this approach requires the knowledge of specific devices and is inflexible to different devices.

The hardware cost predictor is essential for differentiable search algorithms. Since each candidate operator is to be selected by a binary indicator in a differentiable supernet, the latency of each candidate operator \cite{86}:

$$LAT(O) = \sum_i I_{l,i} LAT(O_{l,i}), \quad (22)$$

where $O_{l,i}$ and $I_{l,i}$ are the $i$th operator of the $l$th layer and its associated binary indicator, respectively. $LAT(O_{l,i})$ can be achieved through either LUT \cite{86,66,67} or performance regressor \cite{23}. However, the loss function \cite{14} and \cite{16} with indicator $I_{l,i}$ is not directly differentiable. To sidestep this problem, the indicator is relaxed to be a continuous variable computed via the \textit{Gumbel Softmax} function \cite{123} with learnable parameters:

$$I_{l,i} = \frac{\exp((\theta_{l,i} + g_i)/\tau)}{\sum_i \exp((\theta_{l,i} + g_i)/\tau)}, \quad (23)$$

where $\tau \in (0, 1)$ is the temperature parameter that controls the search efficiency and efficacy. Larger $\tau$ makes the indicator distribution smoother; smaller $\tau$ approaches discrete categorical sampling. Despite the value of $\tau$, the cost function \cite{14} and \cite{16} with the latency calculation \cite{22} and the relaxed indicator \cite{23} are differentiable. For single-path training, the latency calculation \cite{22} can be simplified \cite{23} as:

$$LAT(a) = \sum_i I_l \cdot LAT(O_l), \quad (24)$$

where $O_l$ is the selected operator at layer $l$ and $I_l$ is the path probability of operator $O_l$.

The cost of preparing a hardware cost predictor is minimal since it does not require training models. Only one forward pass of a test model is sufficient to record its cost.

IV. AUTOMATED COMPRESSION OF DEEP LEARNING MODELS

Since deep learning models have proved remarkable performance in a wide spectrum of problems, it is a natural idea to compress these well-established models for memory saving and compute acceleration and thus hardware-efficiency \cite{127}. This idea is fervently supported by the fact that the hand-crafted deep learning models are often over-parameterized \cite{128,7,8}. The goal of deep learning compression is to modify well-trained models for efficient execution without significantly compromising accuracy. Although related works are quite divergent, they can be broadly summarized into four categories \cite{128,127}: tensor decomposition, knowledge distillation, pruning, and quantization. In the past few years, above compression techniques have achieved great success while they crucially rely on domain knowledge, hand-crafted designs, and tremendous efforts for tuning. Recently, there is a growing demand and trend for automating the compression process on all the above four compression categories. In this section, we aim to provide a comprehensive review of recent research studies on automated compression of neural architectures with regard to the four compression categories.

A. Automated Tensor Decomposition

As the computation in neural networks is based on tensor operations, it is intuitive to compress tensors to squeeze and accelerate a neural network. The basic operation of tensor decomposition is the mode-$i$ product of a tensor with a matrix. For a $d$th-order tensor $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ and a matrix $B \in \mathbb{R}^{m \times n_i}(i \in \{1, 2, \ldots, d\})$, the mode-$i$ product between $\mathbf{X}$ and $B$ is $\mathbf{R} = \mathbf{X} \times_i B$, where $\mathbf{R} \in \mathbb{R}^{n_1 \times \ldots \times n_{i-1} \times m \times n_{i+1} \times \ldots \times n_d}$ is also a $d$th-order tensor. Given this definition, a $d$th-tensor $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ can be decomposed into one core $d$th-order tensor $\mathbf{G} \in \mathbb{R}^{r_1 \times r_2 \times \ldots \times r_d}$ and $d$ factor matrices $\mathbf{U}^{(i)} \in \mathbb{R}^{n_i \times r_i}(i \in \{1, 2, \ldots, d\})$:

$$\mathbf{A} \approx \mathbf{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \ldots \times_d \mathbf{U}^{(d)}. \quad (25)$$

This decomposition is called \textit{Tucker decomposition} \cite{129}, and the triple $(r_1, r_2, \ldots, r_d)$ is called the \textit{Tucker rank}. By selecting proper low ranks (i.e., $r_i < n_i$), the original tensor can be represented by lightweight decomposed pieces. The parameter compression ratio is:

$$P = \frac{\prod_{i=1}^d n_i}{\sum_{i=1}^d r_i n_i + \prod_{i=1}^d r_i}. \quad (26)$$

Given that the parameters of a neural network are in the form of tensors (e.g., the filter in a convolutional layer is a 4-way tensor), tensor decomposition can be naturally applied to model the parameters in a more efficient way. The speed-up ratio of a convolution layer is:

$$S = \frac{n_1 n_2 r_1 n_3 r_3 H W}{n_1 r_1 H' W' + n_2 r_2 H' W' + n_3 r_3 H W + n_4 r_4 H' W' + r_1 r_2 r_3 r_4 H' W'}. \quad (27)$$
TABLE II: Summary of Searching for Efficient Deep Learning Models. The “hardware” column indicates on which the achieved models are evaluated. The latency is reported per input; otherwise is specified in the relevant table cells (e.g., FPS). ∼ indicates the exact data is not reported in the paper and thus estimated from the reported figures. ‘-’ indicates unavailable records.

| Search strategy | Reference | Dataset | Hardware | Accuracy | Latency | FLOPs | # of Parameters |
|-----------------|-----------|---------|----------|----------|---------|-------|-----------------|
| BayesOpt+Soft   | MNIST     | Xilinx Zynq 7000 SoC ZC70 | acc: 99.49 | 64.74ms | -       | -     | -               |
|                 | Xilinx Zynq 7000 SoC ZC70 | acc: 99.46 | 105.54ms | -        | -       | -     | -               |
|                 | CIFAR10    | Xilinx Zynq 7000 SoC ZC70 | acc: 88.25 | 136.15ms | -       | -     | -               |
|                 | Xilinx Zynq 7000 SoC ZC70 | acc: 88.52 | 160.75ms | -        | -       | -     | -               |
| ESS-Soft        | CIFAR10    | -       | error: 3.85/7.52/5.90 | -   | -       | 1290M/535M/447M | 3.0M/3.3M/26.8M |
|                 | CIFAR10    | -       | error: 4.57/6.93/8.52/58 | -   | -       | 0.5M/1.1M/4.7M/13.1M |
|                 | ImageNet   | NVIDIA V100/Samsung S20 | Top1 acc: ~79/78.5 | ~20ms/16.25ms | - | - |
| RL+Soft         | ImageNet   | NVIDIA GV100 | Top1/Top5 acc: 73.9/9/2.8 | 11.8ms/11.1ms/101ms | - | 540M | 5.1M |
|                 | SNPE (Mi MIX3) | Top1/Top5 acc: 75.5/9/2.6 | 10.3ms/10.0ms/11ms | - | 240M | 5.5M |
|                 | MACE (Mi MIX3) | Top1/Top5 acc: 75.3/92.5 | 9.6ms/8.4ms/71ms | - | 221M | 5.4M |
|                 | Cityscapes | Top1 acc: 75.4 | 187ms/196ms | - | 312M | 3.9M |
|                 | Google Pixel 1 | Top1/Top5 acc: 75.2/92.5 | 78ms | - | 340M | 4.8M |
|                 | -          | Top1/Top5 acc: 75.6/92.7 | 84ms | - | 403M | 5.2M |
|                 | CIFAR10    | -       | Top1/Top5 acc: 76.7/90.3 | 100ms | - | 2.5M |
|                 | CIFAR100   | -       | error: 18.13 | - | - |
|                 | ImageNet   | Top1/Top5 acc: 75.5/9.1 | 72ms | - | 497M | 4.4M |
|                 | PTB        | Perplexity: 57.5 | - | - | 23M |
|                 | W2T        | Perplexity: 69.4 | - | - | 31M |
|                 | COCO       | Google Pixel 1 | mAP: 24.9/25.5 | 187ms/196ms | - | 900M/940M | 2.5M/2.6M |
|                 | ImageNet   | Top1 acc: 75.4 | 57.1ms | - | - |
|                 | COCO       | -       | Top1/Top5 acc: 76.8 | 106ms | - |
|                 | ImageNet   | Google Pixel 4CPU/GPU/ | Top1 acc: 74.9 | 25.2ms/4.7ms/3.3ms/2.2ms | 340M | 4.39M |
|                 | DSP/EdgeTPU | Top1 acc: 75.8 | 31.0ms/5.4ms/3.8ms/2.4ms | 433M | 4.91M |
|                 | COCO       | Google Pixel 1 CPU | mAP: 23.7/22.3 | 122ms/107ms/11ms | 510M/390M/450M | 3.85M/2.57M/4.21M |
|                 | Google Pixel 4 EdgeTPU | mAP: 25.5/25.424 | 7.6ms/6.8ms/7.4ms | 1530M/1760M/970M | 4.20M/4.70M/4.17M |
|                 | Google Pixel 4 DSP | mAP: 28.5/28.56/9 | 12.3ms/11.9ms/12.2ms | 2820M/3220M/1430M | 7.16M/15.9M/4.85M |
|                 | CIFAR10    | -       | error: 2.30/2.08 | - | - |
|                 | ImageNet   | Google Pixel 1 | Top1/Top5 acc: 74.6/92.2 | 78ms | - | 5.8M/5.7M |
|                 | Tesla V100 | Top1/Top5 acc: 75.1/92.5 | 5.1ms | - |
|                 | ImageNet   | Samsung Galaxy S8 | Top1 acc: 73.0/74.9 | 19.6ms/23.1ms/28.4ms | 249M/259M/375M | 4.3M/4.5M/5.5M |
|                 | ImageNet   | -       | Top1 acc: 68.3/71.2/76.1/77.2 | - | - | 56M/126M/230M/325M |
|                 | ImageNet   | Google Pixel 1 | Top1/Top5 acc: 74.9/92.2 | 78ms | - | - |
|                 | ImageNet   | TITAN-XP | Top1 acc: 76.1/71/74.6/75.3 | 28.9ms/13.6ms/15.4ms/17.9ms (per batch) | 479M/251M/341M/361M | - |
|                 | ImageNet   | Google Pixel 1 | Top1/Top5 acc: 74.9/91.8 | 78ms | - |
|                 | ImageNet   | Top1 acc: 76.6/77.3 | 267M/525M | 5.5M/6.4M |
|                 | COCO key-point | AP/Ap/Ap/Ap: 75.5/72.6/81.7/79.4 | - | 372M |
|                 | -          | AP/Ap/Ap/Ap: 65.7/62.5/71.4 | - | 358M |
|                 | Cityscapes | mIoU: 74.26/75.5 | 4910M/460M | 2.20M/3.85M |
|                 | ADE20K     | mIoU: 33.22/34.92 | 1430M/2190M | 2.49M/3.60M |
|                 | KITTI      | AP moderate/easy/hard: 78.4/97.6/75.53 | - | 15650M |
|                 | -          | AP moderate/easy/hard: 69.7/83.1/9674.89 | - | 3220M |

The exact data is not reported in the paper and thus estimated from the reported figures. '-' indicates unavailable records.
TABLE II: Summary of Searching for Efficient Deep Learning Models (cont.).

| Search strategy | Reference | Dataset | Hardware | Accuracy | Latency | FLOPs | # of Parameters |
|-----------------|-----------|---------|----------|----------|---------|-------|-----------------|
| Other+Soft       | [85]      | CIFAR10 | Titan X/ | error: 4.62 9ms/82ms/149ms | 63.5M 0.52M |
|                  |           |         | Maxwell 256/ | error: 4.78 5ms/75ms/210ms | 137M 1.00M |
|                  |           |         | ARM Cortex53 | error: 4.93 7ms/44ms/81ms | 270M 2.04M |
|                  |           |         | ARM Cortex53 | error: 4.84 9ms/85ms/145ms | 39.27M 0.45M |
|                  |           | ImageNet | Maxwell 256/ | Top1/Top5 error: 24.16/7.13 218ms/541ms | 9276M 77.16M |
|                  |           |         | ARM Cortex53 | Top1/Top5 error: 25.968/21 608ms/876ms | 523M 4.83M |
|                  | [81]      | VOT-19  | A10 Fusion PowerVR GPU/ | EAO/acc/robustness: 0.333/0.590/0.376 530M 1.97M |
|                  |           |         | Kirin 985 Mali-G77 GPU/ | AO/SR0.5: 0.611/0.710 52.6FPS/27.4FPS/ |
|                  |           |         | Snapdragon 845 GPU/ | P/P norm/AUC: 69.5/77.9/72.5 38.4FPS/43.5FPS/ |
|                  | [77]      | ImageNet | NVIDIA GTX1080Ti | mIoU: 60.3/63.7/66.4 89ms/110ms/259ms | 8900M/15000M/73800M 1.1M/2.6M/12.5M |
|                  |           | Google Pixel 1 | Top1 acc: 76.9/80.0 58ms/- | - 230M/955M /- |
|                  |               | ImageNet | Top1/Top5 acc: 79.1/94.5 7.7M/3.4M/29M/2.2M/4.0M | 95.81/94.04/94.82/93.16/95.02/95.64/95.18/93.65/93.07 3390M/1400M/6530M/890M/0.143M/0.047M/0.067M/0.425M/ |
|                  |               | COCO    | mAP: 30.53 2000m/5300m | 5.3M/10.6M |
|                  | [88]      | CIFAR10 | - | acc: ∼0.925∼0.85 - | ∼20M∼2M - |
|                  | [65]      | CIFAR10 | - | error: 3.84/3.7/2.9/3.0/2.2 - | - 7.7M/3.4M/2.0M/2.0M/4.8M |
|                  |           | Google speech commands |acc: 95.819/94.94/8293/16/95.0205/95.189/65/93.07 | - | 3390M/15000M/73800M/12570M/1000M/0.171M/0.733M/2.62M/0.674M/0.035M |
|                  | [75]      | ImageNet | Top1/Top5 acc: 76.5/78.9/79.5/80.9 242M/418M/586M/1040M | 4.5M/5.5M/6.4M/9.5M |
|                  |           | ImageNet | Top1/Top5 acc: 76.3/93.2 360M | 5.3M |
|                  |           | ImageNet | Top1/Top5 acc: 78.9/84.4 708M | 7.6M |
|                  |           | ImageNet | Top1/Top5 acc: 79.0/49.9 1000M | 9.2M |
|                  |           | ImageNet | Top1/Top5 acc: 81.195/5 1800M | 12M |
|                  |           | ImageNet | Top1/Top5 acc: 82.6/96.3 4200M | 19M |
|                  |           | ImageNet | Top1/Top5 acc: 83.3/86.7 9900M | 30M |
|                  |           | ImageNet | Top1/Top5 acc: 84.0/96.9 19000M | 43M |
|                  |           | ImageNet | Top1/Top5 acc: 84.4/97.1 37000M | 66M |
|                  | [68]      | ImageNet | Top1 acc: 76.5/78.9/79.5/80.9 242M/418M/586M/1040M | 4.5M/5.5M/6.4M/9.5M |
|                  |           | ImageNet | Top1 acc: 76.3/93.2 360M | 5.3M |
|                  |           | ImageNet | Top1 acc: 78.9/84.4 708M | 7.6M |
|                  |           | ImageNet | Top1 acc: 79.0/49.9 1000M | 9.2M |
|                  |           | ImageNet | Top1 acc: 81.195/5 1800M | 12M |
|                  |           | ImageNet | Top1 acc: 82.6/96.3 4200M | 19M |
|                  |           | ImageNet | Top1 acc: 83.3/86.7 9900M | 30M |
|                  |           | ImageNet | Top1 acc: 84.0/96.9 19000M | 43M |
|                  |           | ImageNet | Top1 acc: 84.4/97.1 37000M | 66M |
|                  | [69]      | ImageNet | TPUv3/GPUv100 | Top1 acc: 77.3 8.71ms/22.5ms | 910M 7.6M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 79.4 13.6ms/34.4ms | 1580M 10.4M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 86.0 15.7ms/45.5ms | 1890M 11.5M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 81.4 31.9ms/86.6ms | 4300M 16M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 83.0 64.8ms/149.2ms | 10400M 34M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 83.7 125.8ms/290.2ms | 22200M 60M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 84.4 258.1ms/667.2ms | 52000M 137M |
|                  |           | ImageNet | TPUv3/GPUv100 | Top1 acc: 84.7 396.1ms/847.7ms | 93000M 190M |
|                  | [87]      | CIFAR100 | - | Top1 acc: 76.194.0 8.7M | 1.92M |
|                  | [72]      | ImageNet | Top1/Top5 acc: 72.291.0 294M | 3.4M |
|                  |           | ImageNet | Top1/Top5 acc: 74.792.0 471M | 4.7M |
where $H \times W$ is the input feature map size, $H' \times W'$ is the output feature map size, and $n_1 \times n_2 \times n_3 \times n_4$ is the kernel size. Tucker decomposition is a widely applied approach to this aim by tensorization of neural network layers [130], [131]. Another commonly used tensorization approach is the Canonical Polyadic (CP) decomposition [132], which is a special case of the Tucker decomposition [133], [134]. After decomposition, it is usually required to fine-tune the tensorized network to recover the accuracy.

When tensorizing neural networks, the decomposition rank is the most important hyperparameter that needs to be carefully selected since it controls the compression-accuracy trade-off. A typical selection method is cross-validation, which is quite cumbersome for selecting a diverse range of ranks, so the common practice is to set the ranks of different layers to be the same. This simplification is coarse and sub-optimal. Therefore, automating the rank-selection process is crucial to determining optimal decomposition ranks. Kim et al. [130] propose to employ global analytic solutions for variational Bayesian matrix factorization (VBMF) [135] for automatic rank selection of Tucker decomposition. Publicly available tools can easily implement the VBMF. They perform full model compression including fully connected and convolutional layers and show that the accuracy degradation after compression can be well recovered by fine-tuning. However, Gusak et al. [136] claim that they find it difficult to restore the initial accuracy by fine-tuning with the global analytic VBMF ranks. Therefore, they use the global analytic solution of Empirical VBMF (EVBMF) to automatically select ranks. Instead of directly using the ranks achieved from EVBMF, the authors design a weakened rank for decomposition, which is larger than the extreme rank (i.e., obtained by EVBMF) and thus not optimal with a certain amount of redundancy after decomposition. The reason for this design is that this work proposes an iterative compression and fine-tuning strategy that gradually compresses the original model and restores the initial accuracy. Different from tensor decomposition on a well-trained network followed by fine-tuning in [130], [136], Hawkins et al. [137], [138] operate automatic rank determination and tensor decomposition along with the model training process. They propose a prior distribution mask, which can be optimized during training, over the decomposition core tensor to control the rank. If an element of the resultant posterior mask is smaller than a threshold, it will be regarded as zero and the rank will be automatically selected. The authors use a Bayesian inference method to train this low-rank tensorized model and prove its efficacy for various decomposition schemes. MARS [139] sets a factorized Bernoulli prior and optimizes the model via relaxed MAP (maximum a posteriori) estimation to achieve a binary mask to control the rank so the manually set threshold used in [137], [138] is avoided. It shows slightly worse in compression but better accuracy than [138]. Inspired by the success of reinforcement learning in making decisions, Javaheripi et al. [140], [141] propose a state-action-reward system to automatically select the optimal rank for each layer. The hardware cost and accuracy are both considered in the reward and the action of decomposing a layer with the highest rewarded rank is selected. Bayesian optimization can also be utilized to achieve the optimal ranks [142].

TABLE III summarizes the main results of automated tensor decomposition studies. Note that the latency is achieved with the hardware listed in the table, which is different from the hardware for training.

B. Automated Knowledge Distillation

Knowledge distillation (KD) is extended from knowledge transfer (KT) [143] by Ba and Caruana [144] to compress a cumbersome network (teacher) into a smaller and simpler network (student). This is done by making the student model mimic the function learned by the teacher model in order to achieve a competitive accuracy. It is later formally popularized by Hinton et al. [21] as a student-teacher paradigm, where the knowledge is transferred from the teacher to the student by minimizing the difference between the logits (features before the final softmax) of the teacher and student. In many situations, the performance of the teacher is almost perfect with a very high classification probability for the correct class and flat probabilities for the other classes. Therefore, the teacher is not able to provide much more information than the ground truth labels. Hinton et al. [21] introduce the concept of softmax temperature to transfer knowledge, which can better deliver the information of which classes the teacher find similar to the correct class. Formally, given the logits of the teacher model, the classification probability $p_i$ of the class $i$ is:

$$p_i = \frac{\exp\left(\frac{z_i}{\tau}\right)}{\sum_j \exp\left(\frac{z_j}{\tau}\right)},$$

where $\tau$ is the temperature parameter. It controls how soft the labels from the teacher are. The soft labels together with the ground truth labels are used to supervise a compact student model.

Vanilla knowledge distillation mostly focuses on transferring knowledge to a student model with a fixed small architecture, which is manually designed in advance. However, different teachers and tasks favour different student architectures, and hand-crafted architectures are prone to be sub-optimal. Considering these limitations, there is a growing trend to automate the architecture design of a student model [145], [146], [147], [148], [149], [150], [151]. The ground-truth labels are combined with the distillation labels to guide the automatic design process. AKDNet [145] proposes to search optimal student architectures for distilling a given teacher by RL-based NAS. It adopts the efficient search space of [48] and designs a KD-guided reward with a teacher network. KDAS-RL [147], AdaRec [150], and NAS-KD [151] leverage the differential NAS [54] to determine the optimal student structures. In addition, Bayesian optimization [149], [148] and evolutionary search [136] are also popular search strategies in the context of student architecture search.

Although the soft labels from the final output of a teacher have been demonstrated effective for transferring knowledge, some works argue that it is also helpful to mimic the teacher from the intermediate layers. AdaRec [150] and NAS-KD [151] try to minimize the difference between the intermediate features of the student and the teacher. Similarly, Li et al. [152]...
TABLE III: Summary of Automated Tensor Decomposition Studies. The “hardware” column indicates on which the achieved models are evaluated. The model name (e.g., AlexNet) in the “Accuracy” column indicates the original model before tensor decomposition. Some studies only report relative performance changes which are denoted as $n \times$ in the table, indicating the original model is $n$ times its compressed counterpart. The latency is reported per input. ‘-’ indicates unavailable records.

| Method             | Reference        | Dataset         | Hardware          | Accuracy    | FLOPs          | Latency          | # of Parameters |
|--------------------|------------------|-----------------|-------------------|-------------|----------------|------------------|-----------------|
| ARD-LU             | [126]            | MNIST           | MLP (Acc): 98.96/98.30/96.28/98.24 | -           | -              | -                | 7k/101k/4/8k    |
|                   |                  | IMDB            | DLRM (Acc): 87.61/87.79/53.30/88.95 | -           | -              | -                | 66/82/23/16k    |
| Criteo Ad Kaggle   |                  |                 | BiLSTM (Acc): 78.61/78.64/63.77/67.72 | 1.8x        | -              | -                | 56/4/17k/154/20k |
| ARD-HU             | [126]            | MNIST           | MLP (Acc): 97.98/98.30/97.04/98.23 | -           | -              | -                | 7k/91k/3/8k     |
|                   |                  | IMDB            | DLRM (Acc): 87.54/88.18/52/88.78 | -           | -              | -                | 68/5k/14k/14k   |
| Criteo Ad Kaggle   |                  |                 | BiLSTM (Acc): 78.57/78.62/63/87.73 | 1.7x        | -              | -                | 57/4/20k/16/6k  |
| CIFAR-10           | [135]            | -               | ResNet-110 (Acc): 90.04 | -           | -              | 7.4x             | -               |
| CIFAR-10           | [140]            | MNIST           | LeNet-5 (Acc): 99.0 | 1.19x       | -              | -                | 10x             |
| CIFAR-10           | [139]            | -               | ResNet-110 (Acc): 90.79/11.1 | -           | -              | 7.4x/5.5x        | -               |
| [141]              |                  | ARM-A57         | AlexNet (Top5 acc): 81.01/80.77/99.88 | 4.13(2.21/4.61) | - | - | 3.15M |
| [142]              |                  | -               | VGG-16 (Top5 acc): 90.05/99.61/88.89 | 49/37/23/68M | 72.15(42.55)/50.48 | - | 4.49M |

and MFAGAN [153] compress the student generator in Generative Adversarial Networks (GANs) via intermediate feature distillation and once-for-all NAS [77]. PPCD-GAN [154] also compresses the generator but with a teacher-guided learnable mask to automatically reduce the number of channels. Unlike searching for how to reduce, Kang et al. [155] and Mitsuno et al. [156] propose to search on how to increase. They set an extremely small student backbone at the start point, and augment operations [155] or additional layer channels [156] to the backbone during the search process. A large pre-trained teacher [156] or an ensemble teacher [155] is used to guide the search process. The motivation of these works is to alleviate the search burden with a start point and to maximize the distilled knowledge by optimally reducing the capacity gap between teacher and student.

A different direction is NAS-BERT [157], where the block-wise KD [158] is applied to train a supernet composing a bunch of candidate compact subnets. Then, the meta information (e.g., parameter, latency) of the candidate networks is summarized in a lookup table. Given certain hardware constraints, all candidate networks that satisfy the constraints are evaluated and the best performing one is selected as the final compressed model. The benefit of this strategy is that no retraining and researching are required for new hardware constraints.

Not only the student architecture is searchable, but also related hyperparameters (e.g., the temperature $\tau$ in Equation (27) [148]) can be included in the search process. AutoKD [148] uses the BayesOpt to simultaneously explore the optimal student structure and KD hyperparameters, temperature $\tau$ and loss weight $\alpha$, during the KD process.

We outline the major results of automated knowledge distillation papers in TABLE IV.

C. Automated Pruning

Neural network pruning, a.k.a. sparsification, attempts to reduce memory storage and computation cost by removing unimportant weights or neurons from the base network. According to the granularity, pruning can be roughly categorized into structured pruning and unstructured pruning. Unstructured pruning enforces weights or layers to be sparse by removing individual connections or neurons. Structured pruning, on the other hand, targets discarding the entire channels or layers. Although unstructured pruning can theoretically achieve a better accuracy-compression tradeoff, it is less compatible with existing deep learning platforms and hardware than structured pruning. Thus, structured pruning attracts more research focus.

A typical pruning procedure is to first identify and cut away unimportant weights or layers from an existing over-parameterized network and then fine-tune the pruned network to restore the accuracy. The first step is the key and largely relies on expert knowledge and hand-crafted heuristics, such as manually setting the criterion of parameter importance [159] and the pruning rate [160]. However, heuristic settings are typically based on trial and error and are sub-optimal so researchers actively seek automated solutions to alleviate human intervention during pruning. These efforts can roughly be categorized into three directions.

1) Reducing the number of hyperparameters (i.e., per-layer pruning rates) needed to be tuned [161].

2) zeng et al. [161] compute the importance of model parameters using the Taylor expansion of the loss function...
and prune unimportant ones. They introduce an auxiliary hyperparameter that controls the model shrinking proportion at each pruning iteration and show that this hyperparameter is insensitive to the final performance. In this way, they avoid carefully tuning the overall pruning rate. Li et al. [162] use the regularization approach, Alternating Direction Method Multipliers (ADMM), as the core pruning algorithm but substitute its hard constraint with the soft constraint-based formulation and solve the optimization problem with the Primal-Proximal solution. Their approach naturally does not require predefining the per-layer pruning rates and thus reduces the number of hyperparameters. Zheng et al. [163] introduce the normalized Hilbert-Schmidt Independence Criterion (nHSIC) from the information theory to measure the per-layer importance and derive the compression problem with constraints into a linear programming problem, which is convex and easily solved. In such a manner, only two hyperparameters are required and demonstrated robust to different values.

### 2) Searching for optimal pruning rates or magnitude thresholds for pruning

[26], [164], [165], [166], [167], [168], [169], [170], [171], [172], [173]. Note that the magnitude thresholds of weights can be naturally derived from pruning rates. AMC [26] and Auto-Prune [165] leverage deep reinforcement learning (DRL) to automatically determine the pruning rate of each layer and empirically prune the weights with the least magnitude. Arguing that DRL has an inherent incompatibility with the pruning problem, Liu et al. [164] suggest using the heuristic search technique, simulated annealing (SA), to search for the per-layer pruning rates and then use the ADMM to dynamically regularize network weights for structural pruning. They hypothesize that the layers with more weights can have higher compression rates. The authors also apply the ADMM to searching for the magnitude thresholds for the second-phase unstructured pruning after the first-phase structural pruning, Tung et al. [167] propose to employ a Bayesian optimization framework to search the pruning hyperparameters including the magnitude threshold. Some studies [168], [169] follow this work to make it constraint-aware [168] and applicable to pruning deeper networks [169]. Other than pruning full-precision neural networks, Guerra and Drummond

### TABLE IV: Summary of Automated Knowledge Distillation Studies. The “hardware” column indicates on which the achieved models are evaluated. The model name (e.g., Inception-ResNet-V2) in the “Accuracy” column indicates the teacher model; if the teacher is not specified, the teacher is searched during training. The latency is reported per input; otherwise is specified in the relevant table cells (e.g., s/batch). ‘-’ indicates unavailable records.

| Method | Reference | Dataset | Hardware | Accuracy | FLOPs | Latency | # of Parameters | Memory |
|--------|-----------|---------|----------|----------|-------|---------|-----------------|--------|
|        |           |         |          |          |       |         |                 |        |
|        | ImageNet  | Pixel 1 phone | Inception-ResNet-V2 (Top5 acc): 87.5/99.1/93.1 | - | 15ms/25ms/75ms | - | - |
|        | SST-2     | -       | BERT (Acc): 84.1 | - | - | 24M | - |
|        | Ag News   | -       | BERT (Acc): 90.3 | - | - | 16.9M | - |
|        | Market-1501 | -     | ResNet50 (mAP/Rank): 94.7/95.6 | - | - | 14.3M | - |
|        | CIFAR100  | -       | Inception-V3 (Acc): 81.2 | - | - | 4M | - |
|        | MIT67     | -       | DARTS (Acc): 76.0 | - | - | 6M | - |
|        | ImageNet  | -       | Inception-ResNet-V2 (Acc): 78.0 | - | - | 6M | - |
|        | GLUE      | TPUs/4i | IB-BERT$_{LARGE}$ (AvgAcc): 80.38/81.69 | - | 0.45ms/0.58ms | 20.6M/28.5M | - |
|        | SQAD      | -       | IB-BERT$_{LARGE}$ (F1): 88.4/88.1 | - | 0.59ms/0.49ms | 22.8M/20.6M | - |
|        | RetailRocket | -     | NextNet(MRR@5/HR@5/5/NDCG@5): 0.73450/79.643/75.0 | - | - | - | - |
|        | SOMusic   | -       | NextNet(MRR@5/HR@5@5/NDCG@5): 0.63430/73.151/0.6544 | - | - | - | - |
|        | ML-2K     | -       | NextNet(MRR@5/HR@5@5/NDCG@5): 0.44890/65.409/49.95 | - | - | - | - |
|        | GLUE      | -       | IB-BERT$_{LARGE}$ (SST-2/mAP/Rank): 82.8/86.9/81.2 | - | 42.4M | - | - |
|        |          |         |             | - | - | - | - |
|        | Cityscapes | -    | Pix2Pix (mIoU): 41.71 | 5450M | 0.89M | - | - |
|        |          |         |             | - | - | - | - |
|        | COCO-Stuff | -    | GauGAN (mIoU): 61.17 | 31200M | 20.2M | - | - |
|        |          |         |             | - | - | - | - |
|        | Set5     | NVIDIA V100 | MFANet (PSNR/LPIPS): 30.1600/0.571 | 8410M | 21.9ms | 0.55M | 0.52G |
|        | Set14    | NVIDIA V100 | MFANet (PSNR/LPIPS): 26.6900/1.113 | - | - | - | - |
|        | B100     | NVIDIA V100 | MFANet (PSNR/LPIPS): 25.3300/1.322 | - | - | - | - |
|        | Urban100 | NVIDIA V100 | MFANet (PSNR/LPIPS): 24.2300/1.132 | - | - | - | - |
|        | ImageNet | AMD Ryzen 9 3900X | BigGAN (IS/FID/LPIPS): 83.1/12.76/60.62 | 1600M | 2.05s/batch | 13.6M | - |
|        |          | NVIDIA GTX3090 | MFANet (PSNR/LPIPS): 33.42/25.06 | 35400M | 1.19s/batch | - | - |
|        | GLUE     | -       | BERT$_{LARGE}$ (MNLI/QQP/NQ/L-NLI inference): | - | 60M | - | - |
|        |          |         |             | - | - | - | - |
|        | SQADv1.1 | -       | BERT$_{LARGE}$ (EM/F1): 81.28/83 | - | - | - | - |
|        | SQADv2.0 | -       | BERT$_{LARGE}$ (EM/F1): 73.97/77.1 | - | - | - | - |
aim at automatically pruning quantized neural networks by a customized, rule-based pruning strategy with Bayesian optimization of layer-wise pruning ratios.

3) Learning the weight/channel importance [174], [24], [173], [176], [177]. Different from relying on heuristic weight/channel importance measurement like the weight magnitude, this direction struggles to learn the importance measurement through optimizing the final objective. Liu et al. [179] propose a slimming scheme that regularizes the scaling factors in batch normalization (BN) layers as a channel selection indicator to identify unimportant channels (or neurons). Then, they apply a threshold to the trained scaling factors for channel pruning. AutoPrune [24] and DAIS [174] are following works that rely on auxiliary learnable channel selection indicator for pruning. AutoPrune argues that decoupling the channel selection indicators and network parameters will help to stabilize the weight learning and make the pruning insensitive to hyperparameters. DHP [176] introduces an additional hypernetwork that can generate weights for the backbone network (i.e., the network to be pruned). The input of the hypernetwork is the latent vectors, which are attached to each layer of the backbone network. Binary channel masks will then be achieved by setting those latent vector elements, which are smaller than a threshold, to be zero otherwise to be one. In contrast to learning channel importance, ASBP [177] targets to automatically decide the binary importance of each bit of weight. This bit-level pruning has the finest granularity but is impractical to achieve a real reduction of resource consumption due to poor hardware support. To solve this problem, the authors target a specific sort of hardware, RRAM, which assembles multiple low-precision cells together as a crossbar to represent a high-precision data value. They then prune at the granularity of crossbar size, that is all cells in a crossbar share the same pruning strategy. The DL algorithm, deep deterministic policy gradient (DDPG), is used to search for the optimal set of bit-pruning strategies.

1) Unifying knowledge distillation and pruning: Since pruning and knowledge distillation both require a pre-trained model (i.e., the base model in pruning; teacher in KD) to guide the compression process, researchers naturally explore ways to unify these two techniques. One route is the two-step unifying scheme [178], [179]. TAS [178] is an early and representative work. It first searches for the shrunken width and depth of the pruned network with the help of learnable probability from the base model. The searched architecture is then trained from scratch as the student of a simple KD approach with the base model serving as the teacher. The other route is to unify pruning and KD in a single step [180], [181]. Gu and Tresp [181] introduce learnable channel gates into the base model and train the network weights and channel gates with a distillation-aware loss function. Thus, this scheme prunes the base model and distills its knowledge simultaneously and automatically. Yao et al. additionally search for both teacher and student to pursue the optimal distillation pair.

D. Automated Quantization

Unlike the above three compression techniques, which struggle to optimize network architectures, quantization appeals to reduce the representation precision of network weights and intermediate activation tensors. Neural networks are generally trained using the 32-bit floating-point precision; if we were to perform network inference in the 32-bit floating-point as well, MAC operations, data transfer, and data saving would have to be done all in 32-bit floating-point. Hence, using lower bit precision would substantially reduce the hardware overhead, including communication bandwidth, computation and memory usage [182]. For example, when moving from 32 to 8 bits, the memory cost and computation would decrease by a factor of 4 and 16 respectively. In addition, fixed-point computation is more efficient than its floating-point counterpart [182]. The most commonly explored quantization scheme, uniform quantization, converts a floating point tensor \( x = \{x_1, ..., x_N\} \) with range \((x_{\text{min}}, x_{\text{max}})\) into its integer coding \(x_q\) with range \([n, p]\) via the following definition [183]:

\[
x_q = \text{clamp}(\frac{x}{\Delta} + z; n, p),
\]

\[
\Delta = \frac{x_{\text{max}} - x_{\text{min}}}{p - n}, \quad z = \text{clamp}(-[\frac{x_{\text{min}}}{\Delta}] + z; n, p),
\]

\[
\hat{x} = (x_q - z)\Delta,
\]

where \(\Delta\) is the scale factor that specifies the step size of the quantizer; \(z\) is the zero-point that represents the real value zero without any precision loss; \(b\) is the quantization bitwidth; \([-\cdot]\) is the round-to-nearest operation. The \text{clamp}(\cdot) function is to truncate all values to fall between \(n\) and \(p\). Through this procedure, two sorts of errors are induced: a clipping error induced by the \text{clamp} function and a rounding error induced by the round-to-nearest operation \([-\cdot]\). The quantized tensor \(\hat{x}\) is then used for efficient but low precision computation. Since there are no theoretical correlations between the model accuracy and \(b\), simple try-and-error is only feasible in the scenario where all weights and activation tensors have the same precision. However, different layers have different redundancy and behaviours on both the hardware and the task. Therefore, automatically determining the bitwidth of weights and activations for each layer is stunningly attractive, and this mixed-precision feature has recently been supported by hardware manufacturers [184], [185].

One direction is to model the layer-wise bit-precision assignment as a reinforcement learning problem, where the action is to determine the bitwidth values [25], [186]. HAQ [25], [186] is such a representative work. It utilizes the DDPG agent to explore a continuous action space of \([0, 1]\), from which selected actions (i.e., real numbers) can be rounded into discrete bitwidth values for the weights and activations of each layer. The use of a continuous action space is based on the consideration that a discrete action space is not able to represent the relative order of bitwidth. The quantized model is then retrained one more epoch to recover the accuracy. Since the authors induce the hardware constraints by limiting the search space, they only consider minimizing the accuracy drop after quantization in the reward function. Another direction is to develop a differentiable framework that searches for the quantization bitwidth in the same way as searching for the
TABLE V: Summary of Automated Pruning Studies. The “hardware” column indicates on which the achieved models are evaluated. The model name (e.g., LeNet-300-100) in the “Accuracy” column indicates the original model before pruning; if the original model is not specified, it is searched during training. The latency is reported per input; otherwise is specified in the relevant table cells (e.g., FPS (frame per second)). \( \downarrow n\% \) indicates the compressed model decreases \( n \) per cent compared to its original model. \( n \times \) indicates that the original model is \( n \) times its compressed counterpart. '-' indicates unavailable records.

| Method      | Reference | Dataset | Hardware | Accuracy | FLOPs | Latency | # of Parameters | Memory |
|-------------|-----------|---------|----------|----------|-------|---------|----------------|--------|
|             |           | MNIST   |          |          |       |         |                |        |
| CIFAR-10    | [161]     | MNIST   |          | LeNet-300-100/LeNet-5(Top1 error): \( \uparrow 0.08/\uparrow 0.04 \) | -     | 77 \( \times \)200 \( \times \) | 5.4 \( \times \) | -      |
|             |           | CIFAR-10| Octacore | VGG16/ResNet-18/MobileNetV2(acc): 93.5/94.2/94.6 | 2.3/1271.54 | 283M/1219M | 15ms/17ms | -      |
| CIFAR-10    | [162]     | CIFAR-10| Google Pixel 1 | MobileNetV1/MobileNetV2(Top1 acc): 94.00/92.01/94.05 | 98.8M/20.8M/59.5M | - | - | - |
| CIFAR-10    | [163]     | CIFAR-10| Qualcomm Kryo 485 Octacore | VGG16/ResNet-18/MobileNetV2(acc): 93.21/93.81 | 8.8 \( \times \)12.2 \( \times \) | 2.7ms/1.45ms | - | - |
| CIFAR-10    | [164]     | CIFAR-10| Qualcomm Kryo 485 Octacore | VGG16/ReNet-18(Top1 acc): 92.06/92.60/93.66 | \( \downarrow 48.35\%/\downarrow 45.05\%/\downarrow 52.00\% \) | - | - | - |
| CIFAR-10    | [165]     | CIFAR-10| Qualcomm Kryo 485 Octacore | VGG16/ReNet-18(Top5 acc): 66.93/68.98/70.50 | \( \downarrow 46.35\%/\downarrow 55.17\%/\downarrow 55.29\% \) | - | - | - |
| CIFAR-10    | [166]     | CIFAR-10| Intel Core(TM) i7-4790 | CaffeNet(Top1 acc): 53.70 | 69.7ms/batch | - | - | - |
| CIFAR-10    | [167]     | CIFAR-10| NVIDIA MX250 | YOLOv3(mAP): 69.6 | 4458M | 16ms | 4.683M | - |
| CIFAR-10    | [168]     | CIFAR-10| Mobile Robot Detection (sim) | YOLOv3(mAP): 92.1 | 327M | 3ms | 0.298M | - |
| CIFAR-10    | [169]     | CIFAR-10| Mobile Robot Detection (real) | YOLOv3(mAP): 98.0 | 1811M | 8ms | 2.545M | - |
| CIFAR-10    | [170]     | CIFAR-10| Google Pixel 1 | MobileNetV1(Top1 acc): 46.3/69.1 | 6.01ms/74.9ms | - | - | - |
| CIFAR-10    | [171]     | CIFAR-10| Google Pixel 1 | MobileNetV2(Top1 acc): 70.9/30.0 | 61.6ms/53.5ms | - | - | - |
| CIFAR-10    | [172]     | CIFAR-10| Google Pixel 1 | MobileNetV3(Top1 acc): 77.0/78.5 | 225M/314M | 51ms | - | - |
network architecture [187], [188], [189], [190], [191], [192]. Given a network to be quantized, a supernet is first constructed by inserting multiple parallel quantization operations into every pair of adjacent layers of the network. Each inserted operation constitutes an edge from the $i$-th layer to the $(i+1)$-th layer and is associated with a learnable parameter. Multiple parallel edges/operations on the same level are of different quantization bitwidths and summarized by assembling all parallel edges/operations on the same level are of different quantization bitwidths and summarized by assembling all parallel edges/operations with the weights derived from the learnable parameters. Then, the network parameters and the quantization weights are jointly trained with the combination of accuracy and model size as the target. Wu et al. [192] and Xu et al. [189] simply try to diminish the model size while Yu et al. [188] and Wei et al. [191] consider hardware constraints in the training size as the target. Wu et al. [192] and Xu et al. [189] simply try to diminish the model size while Yu et al. [188] and Wei et al. [191] consider hardware constraints in the training 

A. Joint Search and Knowledge Distillation

For knowledge distillation, the intuitive joint automated design strategy is to first search for a powerful teacher model and then achieve a compressed student model by automated knowledge distillation. An improved strategy is FasterSeg [90], which searches for student and teacher alternatively so as to ensure optimal teacher-student matching pairs and effective knowledge distillation. Instead of searching for the entire teacher model, Guan et al. [194] target the feature distillation of the corresponding layer's feature in that group; then the features are weighted to guide the knowledge distillation of the corresponding layer. An improved strategy is FasterSeg [90], which searches for student and teacher alternatively so as to ensure optimal teacher-student matching pairs and effective knowledge distillation. Instead of searching for the entire teacher model, Guan et al. [194] target the feature distillation 

V. JOINT AUTOMATED DESIGN STRATEGIES

Neural architecture design and compression are divergent routines to achieve efficient deep learning models and a combination of these two strategies is an intuitive way to achieve even more efficiency than merely relying on either technique individually. The most natural way in this regard is a sequential pipeline: design then compress. Under the context of design automation, the pipeline is commonly recognized as a search-compress scheme. There are bare studies on combining tensor decomposition and NAS. We will thus concentrate on the other three compression techniques in this section. TABLE VII summarizes main references on joint automated design work.

### TABLE V: Summary of Automated Pruning Studies (cont.)

| Method                  | Reference | Dataset       | Hardware     | Accuracy                | FLOP       | Latency | # of Parameters | Memory       |
|-------------------------|-----------|---------------|--------------|-------------------------|------------|---------|-----------------|--------------|
| ResNet18/ResNet34/ResNet50 | [178]    | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [179]     | -             | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [180]     | -             | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [181]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [182]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [183]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [184]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [185]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |
| CIFAR-10                | [186]     | Tiny-ImageNet | MobileNetV2   | 51.91%/11.92%           | 51.63/32.45 | -       | -               | -            |

VII summarizes main references on joint automated design work.
TABLE VI: Summary of Automated Quantization Studies. The “hardware” column indicates on which the achieved models are evaluated. The model name (e.g., MobileNetV1) in the “Accuracy” column indicates the original model before quantization. The latency is reported per input. \( n \times \) indicates that the original model is \( n \) times its quantized counterpart. ‘-‘ indicates unavailable records.

| Method          | Reference | Dataset          | Hardware             | Accuracy               | Bitwidth | Latency                  | Model Size       |
|-----------------|-----------|------------------|----------------------|------------------------|---------|--------------------------|------------------|
| Automated       | [25], [186] | ImageNet        | MobileNetV1 (Top1 acc): 67.40/70.58/71.20 | 45.51ms/57.70ms/70.35ms | -       | -                        | -                |
| Quantization    |           |                  | MobileNetV1 (Top5 acc): 57.14/67.67/71.74 | 1.09MB/1.58MB/2.07MB    | -       | -                        | -                |
| BISMO on        |           |                  | MobileNetV2 (Top1 acc): 66.97/70.90/71.89 | 52.12ms/66.92ms/82.34ms | -       | -                        | -                |
| Xilinx Zynq-7020 |           |                  | MobileNetV1 (Top1 acc): 65.33/69.97/71.20 | 57.40ms/77.49ms/99.66ms | -       | -                        | -                |
| BISMO on        |           |                  | MobileNetV2 (Top5 acc): 36.60/38.37/39.08 | -                      | -       | -                        | -                |
| Xilinx VU9P     |           |                  | MobileNetV1 (Top1 acc): 65.33/69.97/71.20 | 73.97ms/88.94/100.20    | -       | -                        | -                |
| BitFusion       |           |                  | MobileNetV2 (Top1 acc): 67.01/69.45/71.85 | 7.86ms/11.09ms/19.98ms  | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 64.57/67.00/69.89 | -       | -                | MobileNetV2 (Top5 acc): 87.46/88.94/100.24 | -                      | -       | -                        | -                |
| MobileNetV1 (Top5 acc): 64.57/67.00/69.89 | -       | -                | -                    | -                      | -       | -                        | -                |
| MobileNetV2 (Top1 acc): 66.97/70.90/71.89 | -       | -                | ResNet50 (Top1 acc): 70.63/75.30/76.14 | 6.30MB/9.22MB/12.14MB   | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 66.97/70.90/71.89 | -       | -                | ResNet50 (Top5 acc): 89.93/92.45/92.89 | -                      | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 57.14/67.67/71.74 | -       | -                | Avg: 3.5/3.3/2.9       | -                      | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 57.14/67.67/71.74 | -       | -                | Avg: 3.8/2.9           | -                      | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 67.40/70.58/71.20 | -       | -                | ResNet50 (Top1 acc): 70.63/75.30/76.14 | 10.19×/10.74×/12.08×   | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 67.40/70.58/71.20 | -       | -                | ResNet50 (Top5 acc): 89.93/92.45/92.89 | 6.5MB/4.8MB            | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 87.85/89.89/99.95 | -       | -                | ResNet50 (Top1 acc): 70.63/75.30/76.14 | 3.8/2.9                | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 87.85/89.89/99.95 | -       | -                | ResNet50 (Top5 acc): 89.93/92.45/92.89 | 8.0MB/9.1MB            | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 67.40/70.58/71.20 | -       | -                | Avg: 3.5/3.3/2.9       | -                      | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 67.40/70.58/71.20 | -       | -                | Avg: 3.8/2.9           | -                      | -       | -                        | -                |
| MobileNetV1 (Top1 acc): 87.85/89.89/99.95 | -       | -                | ResNet50 (Top1 acc): 70.63/75.30/76.14 | 3.8/2.9                | -       | -                        | -                |
| MobileNetV2 (Top5 acc): 87.85/89.89/99.95 | -       | -                | ResNet50 (Top5 acc): 89.93/92.45/92.89 | 8.0MB/9.1MB            | -       | -                        | -                |

B. Joint Search and Pruning

Different from the search-compress scheme, pruning can be incorporated into neural architecture search as integrity. The aforementioned automated pruning approaches (in Section IV-C) decide the pruning strategy based on the weights of trained networks. However, a recent study [195] indicates that the essence of network pruning is the pruned structure instead of the weights inherited from the base network, which means that the pruning strategy can be determined before the model is trained. In light of this critical conclusion, the research direction turns to finding optimal pruned structures, e.g., the channel number in each layer, rather than to determining important channels from the base network. This objective is basically the same as NAS so numerous works emerge to use NAS to achieve pruning, where the pruning strategy and network structure are sampled together before training in the search process [196], [197], [198], [199], [200], [201]. However, different from pure NAS, the NAS-based pruning

1) requires an existing state-of-the-art model as the pruning base,
2) only searches the network hyperparameters (e.g., channel number, layer number) instead of the type of operators, and 3) is bounded by searching smaller hyperparameters than the base network. MetaPruning [196] is an early work in this direction, which attempts to find the optimal number of channels for each layer. It first constructs a PruningNet to generate weights.
for pruned networks so that fine-tuning is not required during search time; then it utilizes an evolutionary algorithm to search for the optimal combination of layers with different numbers of channels. The search procedure is highly efficient and no fine-tuning is obliged after searching. However, the search space is large especially when there are many channels in the base network. Some following works thus attempt to shrink the search space by only searching channel number ratios \([198]\) or using a clustering algorithm to cluster channels in each layer as the initial number of channels in the pruned network for later search \([199]\). Other works \([200], [201]\) on the other hand, seek to develop a performance estimator to accelerate the searching process. Overall, most NAS-based pruning process is analogous to searching optimal pruning rates, but without pruning low magnitude channels at each iteration, instead, it trains from scratch with the searched structure or quickly derives the performance with an additional performance estimator.

Alternative to unifying NAS and pruning by sampling the pruning strategy and model architecture before training, SpArSe \([202]\) selects the pruning strategy based on a trained model but optimizes the pruning hyperparameters and model architecture jointly. Specifically, it first samples a configuration of the model architecture (e.g., operators and connectivity) and hyperparameters of pruning algorithms (i.e., Sparse Variational Dropout (SpVD) and Bayesian Compression (BC)); it then trains the model and prunes it. After evaluation, it optimizes the sampling configuration with multi-objective Bayesian optimization. In this way, both structured and unstructured pruning strategies can be optimized together with the model architecture.

C. Joint Search and Quantization

1) Optimizing Search and quantization separately: Some researchers consider optimizing the two components separately and particularly focus on automating the architecture design process. For example, Cai et al. use the ProxylessNAS \([23]\) framework to firstly search for an efficient architecture and then perform quantization-aware fine-tuning to further compress the model \([203]\); Liu et al. \([204]\) include the quantization-friendly activation function ReLU6 in the search space to facilitate the following quantization; Peter et al. \([205]\) compare the performance of post-training quantization (PTQ) and quantization-aware training (QAT) and indicate that the PTQ performance drops rapidly when the bitwidth is smaller than 4 while QAT only has marginal drop even with 1-bit precision. This is due to the incompatibility between quantization and the network structure, which is primarily optimized for full precision.

In light of this, attentions are drawn to quantization-aware NAS, where the search space is based on quantized operators \([206], [207], [208], [209], [210]\). Most of these works are for binary neural networks (BNNs) because of their extreme computation and memory savings \([206], [207], [208], [209], [211]\). Shen et al. \([209]\) apply an evolutionary search to optimize the layer-wise channel expansion ratio for binarized VGG and ResNet. Phan et al. \([207]\) discover that binary DWConv has limited representation capability and thus attempt to search for the group number for each layer of the MobileNet via an evolutionary algorithm as well. BATS \([206]\) and BNAS \([208]\) concurrently emerge in the sense of differentiable quantization-aware NAS. They both design a BNNs-oriented search space discarding the separable convolution due to its failure during quantization; BNAS keeps convolutions and dilated convolutions, and BATS keeps group convolutions and dilated group convolutions. It is interesting to note that the BATS binarizes only activations during training but keeps weights at full precision and binarizes the weights after training with a marginal drop in accuracy (~1%). Instead of optimizing the architecture as a whole, Xu et al. \([211]\) probe layer-wise searching under the KD guidance for BNN object detectors.

The above work only automatically optimizes the neural architectures while using fixed quantization policies. As a result, the final model has a large possibility to be sub-optimal; e.g., the BNN with the searched architecture is not necessarily smaller and more accurate than a mixed-precision model.

2) Optimizing search and quantization jointly: Jointly optimizing the neural architecture and quantization policy can find the best combination of both worlds. The most intuitive joint approach is to construct a new search space that contains both structure and bitwidth choices, and the search process is the same as for traditional hardware-aware NAS \([212], [213]\). For example, JASQ \([212]\) employs a classical evolutionary search algorithm to explore both operators and cell-wise quantization bitwidths; Gong et al. \([213]\), instead, engage differentiable neural architecture search to explore a mixed-precision supernet of MBConv hyperparameters and bitwidths. Though simple and effective, this strategy is inefficient: the search space is enlarged by many times that JASQ requires three GPU days to search for a suitable model for each given resource budget.

To tackle this issue, some studies decouple the supernet training and search process so that no training is required in the search process \([105], [214], [215]\). The weight-sharing training paradigm is further adopted to reduce the training cost. SPOS \([105]\) designs a single path weight-sharing strategy that trains a supernet independently to the search process. In each iteration of the supernet training, only one single path of the supernet is activated and trained. The single-path activation is realized by uniformly sampling one candidate block at each block level of the supernet. The weights of the same block are shared among different supernet training iterations (i.e., different paths). In the search process, an evolutionary algorithm is performed on randomly sampled paths of the pre-trained supernet to find the optimal combination of architecture and block-wise bitwidths. As the supernet is well pre-trained, each architecture only performs inference during the evolutionary search. This SPOS strategy improves the efficiency of both the supernet training and search. APQ \([214]\) adopts the once-for-all \([77]\) supernet training strategy, which requires little time for retraining the searched model but is inefficient to support different bitwidths in the supernet. Therefore, the authors devise a quantization-aware accuracy predictor so that they can train the once-for-all supernet in full precision and derive the accuracy of a quan-
tized network via the accuracy predictor. However, it is hard and time-consuming to collect a large volume of [quantized network, accuracy] pairs for preparing the quantization-aware accuracy predictor due to lengthy quantization-aware training. To alleviate this issue, the authors first train a full precision accuracy predictor with abundant [full-precision network, accuracy] pairs from the supernet, and then fine-tune it with only a few [quantized network, accuracy] pairs to obtain its quantization-aware counterpart. Next, they then do a simple evolutionary search with the accuracy predictor to achieve a new model for a given resource budget. Instead of using the accuracy predictor to deal with the different bitwidths problem, Shen et al. [215] conceive the once quantization-aware (OQA) training strategy, where a set of quantized supernets with sequential bitwidths are created via the bit inheritance scheme: the $k-1$ bit supernet uses double quantization step size of its $k$ bit counterpart. The authors experimentally show that with just one epoch fine-tuning the lower-bit network with bit inheritance outperforms its counterpart with QAT from scratch. The final architecture is derived by a simple coarse-to-fine architecture selection procedure without any retraining. However, OQA only supports fixed precision quantization policies as well.

The most recent work, QFA [183], manages to allow joint mixed-precision quantization and architecture search without retraining. The authors propose a new quantization strategy, batch quantization, to stabilize the estimation of the scale factor so the mixed-precision supernet can be stably trained. They replace the progressive shrinking training strategy in the work [215] with a newly proposed two-stage training strategy reducing more than half training epochs.

VI. FUTURE DIRECTIONS

The design automation for efficient deep learning models is an exciting area yet still in its infant research phase. Existing work has covered almost its every research aspect and laid firm foundations for its future evolution. In this section, we discuss several future directions that are worthy to delve into.

A. Beyond Computer Vision

Most current works validate their performance on computer vision (CV) tasks, e.g., image classification and object detection. This is ascribable to two reasons: 1) the CV area is the most developed area regarding deep learning research that abundant resources are available and experimental protocol is well standardized; thus it requires only bare efforts to prepare comparison benchmarks; 2) the CV models are mainly based on the CNN family and much manual engineering has been devoted to devising efficient CNN models so it is relatively easy to establish an efficient search space or base models for compression. Nevertheless, CV is not the only field that expects efficient deep learning models; applications like IoT are more resource-constrained but seldom investigated regarding the design automation for efficient deep learning. Since these domains have dissimilar characteristics to CV, specific research is desired to go beyond the CV. Though some works have already been presented, like [146], [149], [189] on automatically compressing language models, these are very limited and more room for different applications and the deeper investigation remains to be explored.

B. Accelerating Search

While searched models are driven to be efficient, the search process is often time- and computation-intensive because vast numbers of candidate models need to be evaluated before choosing the best one. This is problematic as the search process is task- and/or hardware-specific and thus a new search is required when a new task or hardware arises. Accelerating the search process can not only facilitate the task/hardware adaptation but also save the energy consumed during the search. The weight-sharing paradigm [124], [54] can alleviate this problem but this is still too slow, especially for the hardware-aware settings. Using proxies is another strategy to accelerate the search process, which approximates different models’ performance without fully training them [117]. Most recently, zero-cost proxies for NAS have attained great interest and shown outstanding performance on some NAS benchmarks [216]. This strategy aims to rank network architectures without any training or even without seeing data [217], [216]. Though remarkable efforts have been made in accelerating conventional NAS, how these approaches can be efficiently and effectively applied to the hardware-aware NAS and automated compression is still an open question. For example, which proxies can be used for automated quantization? Or can we use the same proxies for different quantization bitwidths? Another direction for accelerating the search is to reduce the search space. For example, EfficientMet [68] fixes a baseline model and only searches the constant scale ratios of width/depth/resolution. Only a small grid search can produce extraordinary performance. An effective and reduced search space is based on extensive investigation and human knowledge and thus demands considerable effort in this direction.

C. Hardware Simulation

The feedback (e.g., latency and memory usage) from target devices is a necessity for exploring efficient deep learning models automatically. However, running an enormous amount of models on target devices is labour- and time-intensive, especially when the devices are general-purpose computing systems (e.g., MCU) instead of specific NN accelerators (e.g., TPUs). Furthermore, the frequent communication between the target devices and the devices, on which the design program runs, takes even more time. Therefore, it is desirable to have an accurate hardware simulator that simulates the behaviour of DNN hardware implementations. This simulator would accelerate the automated design process by running all models and obtaining all feedback on a powerful server. In addition, with a hardware simulator, beginners do not need to have cross-disciplinary knowledge in the hardware setting and compilation to collect hardware-cost data; the design process becomes easy to get started with. Although some works have reported hardware simulators like [213], [219], the precision and hardware information that can be provided are far from satisfactory. It is also attractive to have a general software
TABLE VII: Summary of Joint Automated Design Strategies. The “hardware” column indicates on which the achieved models are evaluated. The model name (e.g., MobileNetV1) in the “Accuracy” column indicates the base model used for search; if the original model is not specified, it is searched during training. The latency is reported per input; otherwise is specified in the relevant table cells (e.g., FPS). $\downarrow n\%$ indicates the compressed model decreases $n$ per cent compared to its original model. $\sim$ indicates the exact data is not reported in the paper and thus estimated from the reported figures. ‘-’ indicates unavailable records.

| Method | Reference | Dataset | Hardware | Accuracy | FLOPs | Latency | # of Parameters | Memory |
|--------|-----------|---------|----------|----------|------|--------|----------------|--------|
| CIFAR-100 | [205] | CIFAR-10 | - | (Acc): 79.74% | 1875.88 | - | - | - |
| CIFAR-10 | - | CIFAR-10 | (Acc): 85.41% | 4679.38 | - | 1.27M | 30M | 30M |
| ImageNet | - | ImageNet | NVIDIA GeForce GTX1080Ti | MobileNetV1(Top1 acc): 70.806 | 1.972 | ResNet50/Top1 error/Top5 error: 78.275/473.4 | 800M/2000M/100M |
| CIFAR-10 | [204] | CIFAR-10 | NVIDIA Titan Xp | MobileNetV1(Top1 acc): 71.087 | 493.6 | - | 0.48m | 30M | 17M |
| CIFAR-100 | [203] | CIFAR-100 | NVIDIA GeForce RTX 2080 Ti | MobileNetV2(Top1 acc): 73.271 | 364.5 | - | 0.63m | 47M | 29M |
| ImageNet | - | ImageNet | NVIDIA Tesla V100 | VGG16/ResNet34(acc): 69.42 | 4274.59 | - | - | - |
| CIFAR-10 | [198] | CIFAR-10 | NVIDIA Tesla V100 | VGG16/ResNet50(acc): 90.93% | 56.91 | - | - | - |
| CIFAR-100 | [197] | CIFAR-100 | NVIDIA Tesla V100 | ResNet56/ResNet110(acc): 83.30 | 100.56 | - | - | - |
| ImageNet | - | ImageNet | NVIDIA Tesla V100 | ResNet50/ResNet56/ResNet101/ResNet50/ResNet152(Top1 acc): 67.80/70.96/73.67/67.67/73.12 | 968.13M/2170.77M/1890.60M | - | - | - |
| CIFAR-10 | [196] | CIFAR-10 | NVIDIA Tesla V100 | ResNet56/ResNet110(acc): 83.30 | 100.56 | - | - | - |
| CIFAR-100 | [195] | CIFAR-100 | NVIDIA Tesla V100 | ResNet50/ResNet56/ResNet101/ResNet50/ResNet152(Top1 acc): 67.80/70.96/73.67/67.67/73.12 | 968.13M/2170.77M/1890.60M | - | - | - |
| ImageNet | - | ImageNet | NVIDIA Tesla V100 | ResNet56/ResNet110(acc): 83.30 | 100.56 | - | - | - |
| CIFAR-10 | [194] | CIFAR-10 | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |
| CIFAR-100 | [193] | CIFAR-100 | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |
| ImageNet | - | ImageNet | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |
| CIFAR-10 | [192] | CIFAR-10 | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |
| CIFAR-100 | [191] | CIFAR-100 | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |
| ImageNet | - | ImageNet | NVIDIA Tesla V100 | VGG16/ResNet50/ResNet110(Top1 acc): 93.37/03.65 | 4.54% | 78.32 | - | - |

| Method | Reference | Dataset | Hardware | Accuracy | FLOPs | Latency | # of Parameters | Memory |
|--------|-----------|---------|----------|----------|------|--------|----------------|--------|
| MNIST | - | MNIST | STJE2P/2413 | (acc): 96.9759 | 26 | 51.7% | 10.2% | 28.52ms/27.09ms/3.89ms | 1.3KB/0.7KB |
| CIFAR10 binar | - | CIFAR10 | STJE2P/2413 | (acc): 73.470 | 48 | - | 105.02 | 2.0KB |
| CIFAR10 binar | - | CIFAR10 | MC617 | (acc): 68 | - | 105.02 | 2.0KB |
| CIFAR10 binar | - | CIFAR10 | Chali-MC | (acc): 54.87 | - | 77.60 | 1.8KB |

| Method | Reference | Dataset | Hardware | Accuracy | FLOPs | Latency | # of Parameters | Memory |
|--------|-----------|---------|----------|----------|------|--------|----------------|--------|
| ImageNet | - | ImageNet | Google Pixel 2 | Top1 acc: 69.2 | 8 | 55ms | - | - |
| ImageNet | - | ImageNet | ARM Cortex-A72 | Top1 acc: 75.1 | 8 | ~100ms | - | - |
| ImageNet | - | ImageNet | Coral USB TPU | Top1 acc: 76.3 | 8 | ~5ms | - | - |
| ImageNet | - | ImageNet | Nvidia 2080 Ti | Top1 acc: 76.3 | 32 | 22ms | - | - |
| ImageNet | - | ImageNet | Google Speech | Top1/TOP5 acc: 66.187 | 1 | - | 15M | - |
| ImageNet | - | ImageNet | Top1/TOP5 acc: 69.8058 | 1 | - | 15M | - | - |
platform that offers the simulation of deploying DNNs on diverse and configurable devices.

D. Benchmarking

Despite tremendous results that have been reported to achieve state-of-the-art performance, most of these works are not based on the same protocol thus resulting in skewed comparison. First, some literature uses direct evaluation metrics while others use indirect ones. Superiority on indirect metrics does not necessarily mean more efficiency on real devices. Second, the comparison results can vary on different hardware or different implementation details on the same hardware. Finally, the dataset/task settings also influence the performance significantly. For example, even using the same dataset, different input resolutions would lead to different hardware consumption. It is thus imperative to benchmark these evaluation settings to resolve unfair comparisons and make this domain more reproducible. In addition, a unified benchmark dataset can also facilitate the comparison of design automation algorithms. Li et al. [220] recently develop a comprehensive benchmark dataset of hardware-aware NAS on three categories of hardware (e.g., commercial edge devices, FPGA, and ASIC). However, this benchmark dataset primarily aims to facilitate the development of search strategies, and therefore is relatively small and based on only two spaces. There so far lacks well-recognized benchmark settings and a thorough dataset in this domain.

Apart from the above less noticed and underdeveloped directions, other directions like developing novel efficient operators and competent design algorithms have been widely researched, but there is still large room to improve.

VII. Conclusion

This survey provides a thorough view of the design automation techniques for fast, lightweight and effective deep learning models. We analyze and summarize current studies into three categories: 1) by searching, 2) by compressing, and 3) by jointly. In addition, we conclude the evaluation metrics specific to efficient deep learning models. At the end of this work, we afford discussion of existing issues and future directions for both novices and experienced researchers.

REFERENCES

[1] S. Zhang, L. Yao, A. Sun, and Y. Tax, “Deep learning based recommender system: A survey and new perspectives,” ACM Computing Surveys, Vol. 52, no. 1, pp. 5:1–5:38, 2019.
[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in NAACL-HLT, 2019, pp. 4171–4186.
[3] D. Zhang, K. Chen, D. Jian, L. Yao, S. Wang, and P. Li, “Learning attentional temporal cues of brainwaves with spatial embedding for motion intent detection,” in ICDM, 2019, pp. 1450–1455.
[4] D. Chen and H. Zhao, “Data security and privacy protection issues in cloud computing,” in ICCSEE, 2012, pp. 647–651.
[5] E. Strubell, A. Ganesh, and A. McCallum, “Energy and policy considerations for deep learning in NLP,” in ACL, 2019, pp. 3645–3650.
[6] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, “Green ai,” Communications of the ACM, vol. 63, no. 12, pp. 54–63, 2020.
T. Yang, A. G. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, J. Li, H. Li, Y. Chen, Z. Ding, N. Li, M. Ma, Z. Duan, and D. Zhao, "Stochastic layer-wise precision training," *arXiv preprint arXiv:2108.08532*, pp. 1–10, 2021.

N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "Autocompress: An automatic DNN structured pruning framework for ultra-high compression rates," in *AAAI*, 2020, pp. 4876–4883.

S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X. Sun, "Auto-prune: automated dnn pruning and mapping for reram-based accelerator," in *ICS*, 2021, pp. 304–315.

B. Li, Y. Fan, Z. Pan, Y. Bian, and G. Zhang, "Automatic channel pruning with hyper-parameter search and dynamic masking," in *MM*, 2021, pp. 2121–2129.

F. Tung, S. Muralidharan, and G. Mori, "Fine-pruning: Joint fine-tuning and compression of a convolutional network with bayesian optimization," in *BMVC*, 2017, pp. 115.1–115.12.

C. Chen, F. Tung, N. Vedula, and G. Mori, "Constraint-aware deep neural network compression," in *ECCV*, 2018, pp. 409–424.

J. Mu, H. Fan, and W. Zhang, "High-dimensional bayesian optimization for cnn auto pruning with clustering and rollback," in *ECCV*, 2023, pp. 1–17.

J. Li, H. Li, Y. Chen, Z. Ding, N. Li, M. Ma, Z. Duan, and D. Zhao, "Aepc: Automatic block-wise and channel-wise network pruning via joint search," *arXiv preprint arXiv:2110.03858*, pp. 1–12, 2021.

L. Guerra and T. Drummond, "Automatic pruning for quantized neural networks," in *DICTA*, 2021, pp. 1–8.

T. Yang, A. G. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *ECCV*, 2018, pp. 289–304.

T. Yang, Y. Liao, and V. Sze, "Netadaptv2: Efficient neural architecture search with fast super-network training and architecture optimization," in *CVPR*, 2021, pp. 2402–2411.

Y. Guan, N. Liu, P. Zhao, Z. Che, K. Bian, Y. Wang, and J. Tang, "Dais: Automatic channel pruning via differentiable annealing indicator search," *IEEE Transactions on Neural Network and Learning Systems*, pp. 1–12, 2020.

Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *ICML*, 2017, pp. 2755–2763.

Y. Li, S. Gu, K. Zhang, L. V. Gool, and R. Timofte, "DHP: differentiable meta pruning via hypernetworks," in *ECCV*, 2020, pp. 608–624.

S. Qu, B. Li, Y. Wang, and L. Zhang, "ASBP: automatic structured bit-pruning for ram-based NN accelerator," in *DAC*, 2021, pp. 745–750.

X. Dong and Y. Yang, "Network pruning via transformable architecture search," in *NeurIPS*, 2019, pp. 759–770.

N. Xue and J. Xiao, "Network compression via cooperative architecture search and distillation," in *AHI*, 2021, pp. 42–43.

L. Yao, R. Pi, H. Xu, W. Zhang, Z. Li, and T. Zhang, "Joint-dtnas: Upgrade your detector with nas, pruning and dynamic distillation," in *CVPR*, 2021, pp. 10 175–10 184.

J. Gu and V. Tresp, "Search for better students to learn distilled knowledge," in *ECAI*, 2020, pp. 1159–1165.

M. Horowitz, "1. Computing’s energy problem (and what we can do about it)," in *ISSCC*, 2010, pp. 14–10.

H. Bai, M. Cao, P. Huang, and J. Shan, "Batchquant: Quantized-for-all architecture search with robust quantizer," in *NeurIPS*, 2021, pp. 1074–1085.

EENews, "Apple describes 7nm a12 bionic chips," 2018.

Nvidia, "Nvidia tensor cores," 2018.

W. Yang, Z. Zhang, P. Cui, S. Pan, and S. Han, "Hardware-centric autotuner for mixed-precision quantization," *International Journal of Computer Vision*, vol. 128, no. 8, pp. 2035–2048, 2020.

G. Lacey, G. W. Taylor, and S. Areibi, "Stochastic layer-wise precision in deep neural networks," in *UAI*, 2018, pp. 663–672.

H. Yu, Q. Han, J. Li, J. Shi, G. Cheng, and B. Fan, "Search what you want: Barrier panelty NAS for mixed precision quantization," in *ECCV*, 2020, pp. 1–16.

J. Xu, S. Hu, J. Yu, X. Liu, and H. Meng, "Mixed precision quantization of transformer language models for speech recognition," in *ICASSP*, 2021, pp. 7383–7387.

Q. Sun, L. Jiao, Y. Ren, X. Li, F. Shang, and F. Liu, "Effective and fast: A novel sequential single path search for mixed-precision quantization," *IEEE Transactions on Industrial Informatics*, pp. 1–13, 2022.

X. Wei, H. Chen, W. Liu, and Y. Xie, "Mixed-precision quantization for cnn-based remote sensing scene classification," *IEEE Geoscience and Remote Sensing Letters*, vol. 18, no. 10, pp. 1721–1725, 2021.
[220] C. Li, Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, C. Hao, and Y. Lin, “Hw-nas-bench: Hardware-aware neural architecture search benchmark,” in *ICLR*, 2019, pp. 1–14.