On-NAS: On-Device Neural Architecture Search on Memory-Constrained Intelligent Embedded Systems

Bosung Kim
UNIST
(Ulsan National Institute of Science and Technology)
Ulsan, South Korea
bosung.k@unist.ac.kr

Seulki Lee
UNIST
(Ulsan National Institute of Science and Technology)
Ulsan, South Korea
seulki.lee@unist.ac.kr

ABSTRACT

We introduce *On-NAS*, a memory-efficient on-device neural architecture search (NAS) solution, that enables memory-constrained embedded devices to find the best deep model architecture and train it on the device. Based on the cell-based differentiable NAS, it drastically curtails the massive memory requirement of architecture search, one of the major bottlenecks in realizing NAS on embedded devices. On-NAS first pre-trains a basic architecture block, called *meta cell*, by combining n cells into a single condensed cell via *two-fold meta-learning*, which can flexibly evolve to various architectures, saving the device storage space n times. Then, the offline-learned meta cell is loaded onto the device and unfolded to perform online on-device NAS via 1) *expectation-based operation and edge pair search*, enabling memory-efficient partial architecture search by reducing the required memory up to k and $m/4$ times, respectively, given k candidate operations and m nodes in a cell, and 2) *step-by-step back-propagation* that saves the memory usage of the backward pass of the n-cell architecture up to n times. To the best of our knowledge, *On-NAS is the first standalone NAS and training solution fully operable on embedded devices* with limited memory. Our experiment results show that On-NAS effectively identifies optimal architectures and trains it on the device, on par with GPU-based NAS in both few-shot and full-task learning settings, e.g., even 1.3% higher accuracy on miniImageNet, while reducing the run-time memory and storage usage up to 20x and 4x, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**.

KEYWORDS

Mobile Computing, Neural Network, Neural Architecture Search

ACM Reference Format:
Bosung Kim and Seulki Lee. 2023. On-NAS: On-Device Neural Architecture Search on Memory-Constrained Intelligent Embedded Systems. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '23), November 12–17, 2023, Istanbul, Turkiye.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3625687.3625814

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SenSys '23, November 12–17, 2023, Istanbul, Turkiye
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0414-7/23/11…$15.00
https://doi.org/10.1145/3625687.3625814

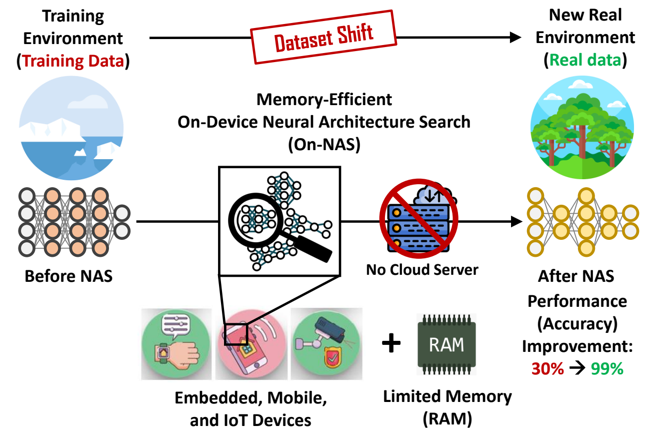

Figure 1: On-NAS enables memory-efficient neural architecture search (NAS) on the device, which effectively tackles the dataset shift problem intrinsic in many deep learning-based embedded, mobile, and IoT applications. It drastically curtails the huge memory (RAM) usage required by neural architecture search, which is one of the major bottlenecks of NAS, thus allowing memory-constrained embedded devices to update (adapt) their model architecture to new real data accordingly in a memory-efficient manner solely on the device without relying on external systems, e.g., cloud servers.

1 INTRODUCTION

As deep learning has made huge progress over the past decade [6, 45, 82, 87], various deep models are now actively deployed onto a wide range of embedded and mobile devices [51, 97]. Currently, the de facto way of bringing a deep model to those devices is to train a small-size deep model on a GPU machine offline and then deploy it onto the device primarily for inference tasks. However, the offline-train-and-deploy strategy suffers from a significant problem called *dataset shift* [74] that ruins the model performance, e.g., lower prediction accuracy, caused by *the disparity between the training and real data distributions* [64]. Since a real environment to which devices are deployed is likely to be different from the pre-defined training environment, the offline-trained deep model can hardly make correct inferences on new real environment data unless it is re-trained with real environment data online in a timely manner [1]. Considering that 1) most real environments keep evolving, and 2) their data distributions change accordingly, the performance and reliability of many deep learning applications, such as computer vision [96], NLP [39], and reinforcement learning [34], unable to be trained with real data, tend to deteriorate over time. Specifically, dataset shift is prevalent in embedded and mobile settings, as those devices are highly likely to keep running into dynamic environments and various users. Tab. 1 shows some examples of deep model performance degradation caused by dataset shift.

To solve the dataset shift problem, various on-device training techniques have been proposed to adapt (update) deep models to real environment data [30, 49, 94] in the wild. Although they have enabled resource-efficient and lightweight deep model training on the device, the model's performance, adaptability, and flexibility on new real data are still limited to some extent since they *only update the model's weight parameters* without adjusting the model architecture to new data, as the learning effect of weight update is significantly limited by the model architecture and capacity [17, 40].

Task (Dataset)	Metric	Training (ID)	Real (OOD)	Gap
CMNIST [46]	Avg Acc (%)	87.4	17.1	70.3
CIVILCOMMENTS [43]	Worst Group Acc (%)	92.2	56.0	36.2
CAMELYON17 [43]	Avg Acc (%)	93.2	70.3	22.9

Table 1: Deep models trained only with the pre-defined training data are likely to experience performance drops when encountering real data on the field of which distribution is different from that of the training data, i.e., in-distribution (ID) vs. out-of-distribution (OOD). More experimental details can be found in the original references [43, 46]. The tasks above are dedicated to simulating dataset shift.

Unlike the weight-only-update training, *Neural Architecture Search (NAS)* [22] finds the best-performing model architecture and weight parameters simultaneously for target data. Since *NAS optimizes not only the weight parameters but also the model architecture*, it is well known that NAS, in general, can better adapt to new data, compared to keeping the model architecture intact and only updating the weight parameters [22]. For instance, the weight-only-update YOLOv3 [76] and the NAS-based NAS-FPN [29] achieves an average precision of 22.1 and 48.3 on the COCO test-dev [54], respectively, exemplifying a huge performance gap between them. Furthermore, in perspective of addressing temporal dataset shift with NAS, AutoML4ETC [63] has recently shown state-of-the-art performance at encrypted traffic classification (ETC) tasks with NAS, showing a substantial amount of robustness toward temporal dataset shift.

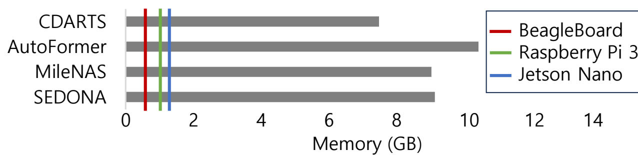

Figure 2: The memory requirement of NAS (CDARTS [98], AutoFormer [12], MileNAS [35], and SEDONA [71]) vs. the memory capacity of embedded platforms (BeagleBoard [16], Raspberry Pi 3 [70], and Jetson Nano [11]).

Although NAS enables deep models to better adapt to new data by composing the optimal network architecture for it, it comes at the cost of computing resources, especially *huge memory usage* [77, 99]. As NAS typically trains multiple candidate architectures simultaneously in search of the best one, the required memory space easily becomes to exceed that of the weight-only-update training that only deals with a single fixed model. Unlike the compute-intensive process of NAS, which has been alleviated by recent studies [10, 90, 95], the memory bottleneck remains the biggest obstacle to the practical implementation of NAS [95]. Considering the limited size of memory on resource-constrained embedded, mobile, and IoT devices, realizing NAS on such devices has been regarded as a formidable task so far despite its superior performance. Fig. 2 compares the memory requirement of state-of-the-art NAS and the memory capacity of commercial embedded platforms, clearly showing that existing NAS solutions can hardly run on memory-constrained embedded devices.

In this paper, we propose *On-NAS—a memory-efficient on-device neural architecture search (NAS)* which enables deep models running on a memory-constrained embedded system to adapt to new data within the tight memory budget based on gradient-based meta-learning [23] and differentiable architecture search [59]. It curtails the run-time memory usage and the storage requirement of NAS with three approaches: 1) *two-fold meta-learning*, 2) *expectation-based operation and edge pair search*, and 3) *step-by-step back-propagation*. Those proposed three methods allow a deep model to best learn new data on the device through *joint optimization of the model architecture* and weight parameters, providing *improved flexibility and adaptability in data adaptation* when compared to updating only the weight parameters, as illustrated in Fig. 1. With On-NAS, memory-constrained embedded devices become able to provide stable model performance in response to data changes (dataset shift) on their own without a manual effort of architecture design, leading such devices to be flexible and capable self-learning agents. To the best of our knowledge, *On-NAS is the first standalone on-device NAS entirely run on the device without relying on external systems*, unlike existing on-device NAS [3] that offload the NAS workloads as well as the device data to the cloud and then receive back the resulting model architecture.

We argue that On-NAS can tackle many dataset shift problems of intelligent embedded systems, especially for few-shot learning [67, 83], as On-NAS allows an agile architecture search with a reasonable number of architecture search steps on the device, enabled by the proposed meta cell pre-trained with multiple meta-learning tasks. Besides, differential architecture search allows efficient architecture adaptation by greatly reducing search time via gradient-based optimization. Furthermore, On-NAS can also apply to more severe dataset shift problems as future proof, including full-task learning where an optimal model architecture is found using the entire training dataset, which usually takes a larger number of search steps.

On-NAS can be best utilized when an intelligent embedded system, e.g., a robot, health care, military deployments, and infrastructures of rural areas, needs to adapt to the heterogeneity and/or change of data by itself [28, 33, 52] in a situation not allowing frequent deployments of newly-trained offline models as well as data transfer, possibly due to scarce network connectivity or privacy issues. For instance, underground power tunnels constructed 40m deep with minimum or no connectivity should be monitored for anomalies in power transmission lines. To monitor a wide range of incidents, including internal subsidence cracks [33], recent studies apply neural networks [33] for detection. Since such tunnels vary a lot in their structures, e.g., spatial difference, colors, etc., an offline-trained model can hardly make proper inferences on each tunnel as expected because of the disparate data distributions of each tunnel and their diverse deteriorations over time, e.g. aging, wear, and tear. In this case, On-NAS would be a feasible solution, as it is capable of adapting the model architecture and weight parameters to evolving or unseen data of each tunnel on the device without connections. Moreover, On-NAS leverages meta-learning, enabling itself to adapt faster to new data, when the number of newly acquired real data

samples with labels are not abundant, which can be interpreted as few-shot tasks.

We implement On-NAS on NVIDIA Jetson Nano [11] with PyTorch [68] and open-source it with an anonymous public git repository[1], which can be easily adapted to various embedded platforms. We evaluate On-NAS in two dataset shift scenarios with NAS benchmark tasks widely used to assess the performance of architecture search: 1) few-shot learning of miniImageNet [88] and Omniglot [48], and 2) full-task learning of CIFAR-10 [44] and CIFAR-100 [44]. The experiment results demonstrate that On-NAS reduces the run-time memory and storage usage of NAS up to 20x and 4x, respectively while successfully finding optimal architectures for various data configurations on the device, achieving competitive classification accuracy to existing GPU-running NAS consuming 20x more memory, e.g., 63.3% (On-NAS) vs. 62.0% (MetaNAS [23]) on 5-shot learning of miniImageNet.

2 BACKGROUND

2.1 Differentiable Architecture Search

Although neural architecture search (NAS) [19, 22, 37] can produce optimal architectures in various domains [7, 24, 29, 38, 101], they take a painfully long search time. For instance, a reinforcement learning-based NAS [103] takes 2,000 GPU days for an image task, implying they are infeasible for resource-limited embedded devices. To address this problem, many approaches have been suggested, including cell-based motifs [77] and differentiable architecture search (DARTS) [59] which takes 1.5 GPU days at same task.

In On-NAS, we take the combination of the *cell-based motif* [69] and *differentiable architecture search (DARTS)* [59] as the basic strategy. It allows efficient architecture search with cell-based structure and gradient descent with continuous relaxation [59], enabling us to achieve both a significant reduction in search time and superior performance, making it a practical choice for on-device NAS.

The super network of differentiable architecture search (DARTS) is a stack of multiple cells consisting of several nodes, $x^{(i)}$, in each, where the node $x^{(i)}$ is the summation of outputs computed through previous edges that take the previous nodes as input [59]. Each direct edge (i, j) between two nodes $x^{(i)}$ and $x^{(j)}$ contains a set of candidate operations $o^{(i,j)}$, e.g., convolution and pooling, to be included in the final architecture, which transforms $x^{(i)}$ by applying $o^{(i,j)}$ as:

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{1}$$

If we let O be a set of candidate operations, where each operation $o(\cdot) \in O$ is applied to $x^{(i)}$, the problem of categorical selection of operations is relaxed as follows:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x) \tag{2}$$

where the vector $\alpha^{(i,j)}$ of length $|O|$ denotes the operation mixing weight, also called *operation parameter* in this paper, for a pair of nodes (i, j). Thus, the task of architecture search is reduced to optimizing (learning) the operation parameter $\alpha = \{\alpha^{(i,j)}\}$ through the gradient descent [79]. From this, the final architecture is fixed by selecting the most-likely operation $\bar{o}^{(i,j)}$ from $o^{(i,j)}$=argmax$_{o \in O} a_o^{(i,j)}$, as shown in Fig. 3.

[1] https://github.com/eai-lab/On-NAS

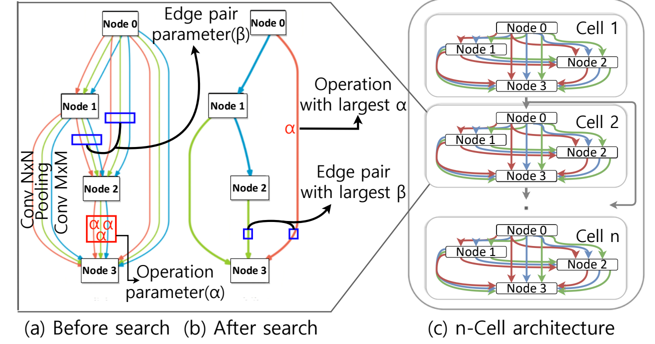

Figure 3: In differentiable architecture search (DARTS [59]), candidate operations, e.g., convolution and pooling, are deployed on edges in a repeated motif named "cell". The operation parameter α and the edge pair parameter β assigned to each operation and edge pair, respectively, are learned via the gradient descent to determine which operations and edge pairs to be included in the final architecture. (a) before optimizing (learning) α and β, (b) the final architecture after optimizing α and β, and (c) the cell-based architecture consisting of a stack of n cells as the basic building block. Figures are from [59].

Similar to the operation parameter α, another parameter β for each possible pair of m input nodes in a cell, called *edge pair parameter*, is introduced [23] to sparsify the combination of nodes used as inputs to the next node. Hence, by learning the edge pair parameter β along with α, optimal model architectures can be found after the discretization of architecture by gradually optimizing them simultaneously.

In sum, the task of differentiable architecture search is reduced to *find optimal α and β along with the weight parameter, denoted as w*, where a designated number of operations with higher α and edge pairs with higher β survive in the final architecture among vast search space, whereas the other remaining operations and edge pairs with lower α and β are pruned away.

2.2 Meta-Learning Architecture Search

Recently, the concept of meta-learning-applied architecture search [20, 23, 41] has been introduced to expand the advantage of meta-learning [65, 75], i.e., finding the best initial weight parameter set that can readily adapt to new datasets, to the domain of neural architecture search, i.e., *finding the best initial architecture* for faster and better model architecture adaptation. For instance, MetaNAS [23] optimizes the initial operation parameter α and edge pair parameter β from multiple tasks (datasets) using the gradient descent, along with the initial weight parameter w, prior to adapting to new data. As meta-learning-applied architecture search utilizes the gradient-based optimization similar to differentiable architecture search (DARTS), the idea of connecting them naturally becomes to work through the computational graphs. Therefore, a meta-learned architecture becomes more flexible to new situations with scarce or expensive data, making it a suitable pre-training strategy prior to on-device NAS.

In On-NAS, we utilize MetaNAS [23] and Reptile [66] for meta-learning-applied initial architecture setup (α, β, and w) as the preliminary step of on-device architecture search, considering its low computation complexity and simplicity.

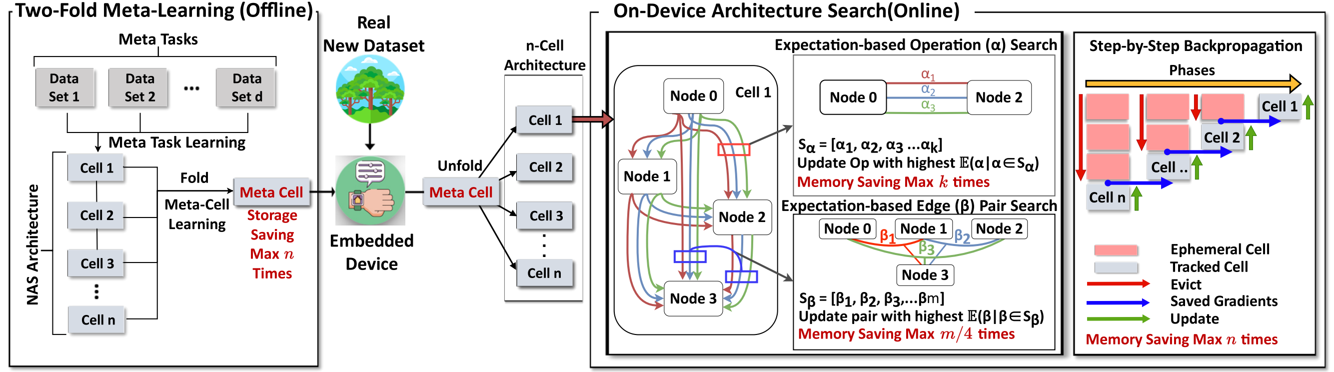

Figure 4: The overview of On-NAS: On-NAS first performs offline two-fold meta-learning over two axes, i.e., meta-task (dataset) learning and meta-cell learning, to pre-train a single meta cell, which combines n cells into one, saving the device storage usage n times. Next, given a real new dataset to adapt, a memory-efficient on-device architecture search is executed online by unfolding the meta cell and letting it evolve accordingly via the expectation-based operation (α) and edge pair (β) search over k candidate operations and m nodes in a cell, respectively, with step-by-step back-propagation, which collectively reduces the run-time memory requirement of architecture search up to $k \cdot m/4 \cdot n$ times.

2.3 Memory Burden of Architecture Search

Unfortunately, neural architecture search is considered one of the most resource-demanding as well as time-consuming tasks [61], as the search process usually entails multiple model training and validation sub-processes. Although differential architecture search (DARTS) [59] greatly reduces search time as described above, it suffers from a huge memory burden. The primary reason comes from that it needs to save intermediate outputs of all candidate operations [9], also known as activation [30, 99, 102], used to learn the operation parameter α in Eq. (2), edge pair parameter β, and the model weight parameter w through the back-propagation [20, 23].

For example, the super network [59, 100] composed for CIFAR-10 [44] is a stack of 8 cells, where each of the cells consists of 7 nodes, which makes 27 edges including 8 candidate operations each [59]. As a result, a total of $1,728 = 8 \times 27 \times 8$ outputs (tensors) should be saved in memory at run-time. Even worse, it is amplified proportionally to the mini-batch size used in the back-propagation, e.g., $55,296 = 1,728 \times 32$ tensors to be saved when the batch size is 32, which is 11 times larger than ResNet-152 [36] that requires saving roughly $4,864 = 152 \times 32$ tensors in memory for its training.

2.4 Search Space for Few-Shot Tasks

It is known that the performance of neural architecture search (NAS) significantly depends on the designated search space. Maintaining a balance between the search time and the optimal model architecture's performance necessitates careful consideration of the search space. To elaborate, a broader range of candidate operations and complex target architectures increases the likelihood of finding the optimal design. However, this approach inevitably requires a painfully-long search time, making such NAS algorithms practically unfeasible. In this paper, we leverage the well-established modular search space of MetaNAS [23], widely regarded as a suitable NAS technique for few-shot tasks, as shown by multiple prior works [20, 23, 57]. Few-shot tasks can be interpreted as an adaptation toward dataset with different distributions, aligning with numerous prior works to mitigate dataset shift through few-shot learning [60, 89, 91], which results in notable performance in few-shot classification tasks. We assume that the suggested search space of MetaNAS [23] would be suitable for our specific circumstances.

3 OVERVIEW

Fig. 4 depicts the overview of On-NAS, taking two steps: 1) *offline two-fold meta-learning* and 2) *online on-device architecture search.*

3.1 Two-Fold Meta-Learning

Two-Fold Meta-Learning. The first step of On-NAS is to pre-train a *single meta cell* offline via *two-fold meta-learning* that performs meta-learning over two axes: 1) *meta-task learning* and 2) *meta-cell learning*, as shown in the left side of Fig. 4. Unlike existing meta-learning NAS [23] that performs only meta-task learning over multiple tasks to boost the adaptability to new data, On-NAS additionally performs meta-cell learning over n cells and combines them into a single meta cell such that it readily evolves to any necessary optimal cell architecture accordingly during on-device architecture search. By having only one meta cell, not n cells, the device storage requirement reduces to n times, where n denotes the total number of cells in the model architecture, enabling it to fit the limited storage of the embedded device. To generate a meta cell that can effectively evolve to necessary architectures while folding it into a single cell conveying only $1/n$ information, On-NAS applies the *task coefficients* and *cell coefficients* to the meta-task and meta-cell learning, respectively. The offline-learned meta-cell is loaded onto the device and used as a starting backbone cell.

3.2 On-Device Architecture Search

The second step of On-NAS is to perform an online on-device architecture search from the pre-trained meta cell for real new data. It first unfolds the meta cell into n cells to construct the initial architecture and applies three memory-saving search algorithms: 1) *expectation-based operation search*, 2) *expectation-based edge pair search*, and 3) *step-by-step back-propagation*, as shown on the right side of Fig. 4.

Expectation-based Operation Search. In the cell-based architecture search [69], an edge connects two nodes (Eq. (1)) through k candidate operations. To determine which of them to be included in the final architecture, the relative probability [59] of each operation, represented by the *operation parameter* α in Eq. (2), is computed using differentiation. As α parameters of all operations are computed at the same time, it becomes to consume a large amount of memory. To reduce memory usage, On-NAS updates α parameters

only for a selected subset of operations at a time based on their expected value at the final search step obtained by estimating the gradients of remaining future search steps. It saves the run-time peak memory up to k times when a single operation is selected to be updated while allowing for optimal operation selection comparable to updating all α simultaneously.

Expectation-based Edge Pair Search. Among a set of possible edges that connect m nodes in a cell, On-NAS includes a pair of edges for each node as the final architecture [23], where each possible edge pair is weighted by the *edge pair parameter* β, representing the relative probability of each edge pair in architecture search. The edge pair parameter β works similarly to α for candidate operations, determining which edge pair will be included in the final architecture. To decrease the memory usage of edge pair search, On-NAS selects a subset of edge pairs and only updates the selected subset of β at a time based on their expectation computation similar to that of α. It allows finding the best edge pairs of the final architecture using up to $m/4$ times less memory.

Step-By-Step Back-Propagation. Finding an optimal architecture (α and β) along with the weight parameter (w) entails the back-propagation [93] computation. Since it requires saving the intermediate output (activation) of each layer in memory [95], the amount of memory needed for it easily becomes to exceed the limited capacity of embedded devices. To tackle this problem, On-NAS performs back-propagation for each cell separately one by one, starting from the last cell to the first one by maintaining the intermediate gradient required to calculate the chain of gradients between the cells and reusing the same memory space repeatedly in a similar manner to the re-materialization [31] and gradient check-pointing [13, 32]. Based on the repeated cell-based architecture, On-NAS first analyzes the entangled connections and then determines which gradients should be saved in memory in the computational graph of the back-propagation. By executing the back-propagation of each cell one at a time and reclaiming the memory space, the memory usage of a n-cell architecture is reduced up to $1/n$ while producing the exactly same back-propagation result.

4 TWO-FOLD META-LEARNING

As the preliminary step of On-NAS, the meta cell is pre-trained offline via *two-fold meta-learning* performed over two axes, i.e., 1) *meta-task learning* and 2) *meta-cell learning*. The two-fold meta-learning incorporates n cells into a *single meta cell* based on the *task and cell coefficients* that we propose to take into account the relative importance of each task and cell during the meta-learning for performance improvement and architecture search acceleration. Fig. 5 depicts the procedure of two-fold meta-learning for a single meta-epoch.

4.1 Task and Cell Coefficients

To generate a single meta cell that can readily adapt to new datasets, we apply the task and cell coefficients when aggregating the operation (α), edge pair (β), and weight (w) parameters of multiple tasks and cells during the two-fold meta-learning. By utilizing those two independent coefficients over two axes, it can learn more important tasks and cells with higher weights (coefficients) than less significant ones, which is crucial to attaining optimal initial architecture.

Task Coefficients. To represent the relative significance of the optimized model architecture for a task, we assume that a certain task is more crucial for finding the optimal initial architecture state than other tasks if it requires a longer Euclidean distance to get optimized. We define this distance representing the optimization strength of a task as the *task coefficient* and apply it to meta-task learning. The task coefficients promote the direction of parameter updates closer to more important tasks, yet further from the updates invoked by relatively less important tasks. Hence, they are expected to increase the performance of meta-learning-applied architecture search by reflecting the importance of each task accordingly while accelerating architecture search, unlike existing works [20, 23] that do not apply such coefficients.

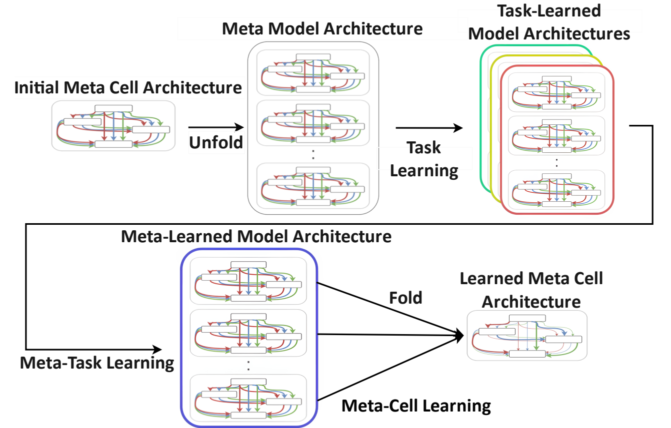

Figure 5: The illustration of two-fold meta-learning for a single meta-epoch: 1) the initial meta cell is unfolded to the n-cell meta architecture, 2) the meta architecture is updated with multiple task architectures (meta-task learning), and 3) the learned meta architecture is folded back to the single meta cell (meta-cell learning).

Let α_{meta}, β_{meta}, and w_{meta} be the meta operation, edge pair, and weight parameter, respectively, which we want to find through meta-task learning. Then, for the i-th task T_i among a total of d tasks to be meta-learned, its task operation, edge pair, and weight parameters, denoted as α_{T_i}, β_{T_i}, and w_{T_i}, respectively, and their task gradients over s search steps, denoted as $\mathcal{G}^{\alpha}_{T_i}$, $\mathcal{G}^{\beta}_{T_i}$, and $\mathcal{G}^{w}_{T_i}$, respectively, are defined as:

$$
\begin{aligned}
\alpha_{T_i} &= \alpha_{meta} - \lambda_\alpha \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial \alpha_{T_i}}, \ \mathcal{G}^{\alpha}_{T_i} \triangleq \lambda_\alpha \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial \alpha_{T_i}} = \alpha_{meta} - \alpha_{T_i} \\
\beta_{T_i} &= \beta_{meta} - \lambda_\beta \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial \beta_{T_i}}, \ \mathcal{G}^{\beta}_{T_i} \triangleq \lambda_\beta \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial \beta_{T_i}} = \beta_{meta} - \beta_{T_i} \\
w_{T_i} &= w_{meta} - \lambda_w \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial w_{T_i}}, \ \mathcal{G}^{\beta}_{w_i} \triangleq \lambda_w \sum_{j=1}^{s} \frac{\partial \mathcal{L}_{T_{i,j}}}{\partial w_{T_i}} = w_{meta} - w_{T_i}
\end{aligned}
\tag{3}
$$

where λ_α, λ_β, and λ_w is the learning rate of α_{T_i}, β_{T_i}, and w_{T_i}, respectively, and $\mathcal{L}_{T_{i,j}}$ is the loss of T_i at the j-th step.

As shown on the right part in Eq. (3), the gradients of the i-th task T_i , i.e., $\{\mathcal{G}^{\alpha}_{T_i}, \mathcal{G}^{\beta}_{T_i}, \mathcal{G}^{w}_{T_i}\}$, are efficiently computed by taking the distances between the meta parameters and the task parameters [66], i.e., $\{\alpha, \beta, w\}_{meta} - \{\alpha, \beta, w\}_{T_i}$. The distances between them represent the optimization strength of the i-th task T_i over the axes of $\{\alpha, \beta, w\}$, which is equivalent to the summation of the task gradients over s steps [66].

By normalizing the task gradients, $\{\mathcal{G}^{\alpha}_{T_i}, \mathcal{G}^{\beta}_{T_i}, \mathcal{G}^{w}_{T_i}\}$ in Eq. (3), with the softmax operation [4], the task coefficients of $\{\alpha, \beta, w\}_{T_i}$, denoted as $\{\mathcal{T}^{\alpha}_{T_i}, \mathcal{T}^{\beta}_{T_i}, \mathcal{T}^{w}_{T_i}\}$, are obtained as follows.

$$\mathcal{T}^{\alpha}_{T_i}, \mathcal{T}^{\beta}_{T_i}, \mathcal{T}^{w}_{T_i} = \frac{\exp(\mathcal{G}^{\alpha}_{T_i})}{\sum_{j=1}^{d}\exp(\mathcal{G}^{\alpha}_{T_j})}, \frac{\exp(\mathcal{G}^{\beta}_{T_i})}{\sum_{j=1}^{d}\exp(\mathcal{G}^{\beta}_{T_j})}, \frac{\exp(\mathcal{G}^{w}_{T_i})}{\sum_{j=1}^{d}\exp(\mathcal{G}^{w}_{T_j})} \quad (4)$$

The task coefficients, $\{\mathcal{T}^{\alpha}_{T_i}, \mathcal{T}^{\beta}_{T_i}, \mathcal{T}^{w}_{T_i}\}$ in Eq. (4), are used as the relative importance weight of T_i during two-fold meta-learning, along with the cell coefficients described below.

Cell Coefficients. In a similar way to computing the task coefficients, the *cell coefficients* among multiple cells can be derived for meta-cell learning. By considering cells at different locations in a n-cell architecture as different tasks that the cells have to be optimized, it becomes possible to calculate the relative coefficients among them.

We define the cell coefficients, $\{C^{\alpha}_{c_l}, C^{\beta}_{c_l}, C^{w}_{c_l}\}$, of the l-th cell c_l as the softmax normalization of the cell gradients, $\{\mathcal{G}^{\alpha}_{c_l}, \mathcal{G}^{\beta}_{c_l}, \mathcal{G}^{w}_{c_l}\}$, similar to Eq. (4), which is given by:

$$C^{\alpha}_{c_l}, C^{\beta}_{c_l}, C^{w}_{c_l} = \frac{\exp(\mathcal{G}^{\alpha}_{c_l})}{\sum_{j=1}^{n}\exp(\mathcal{G}^{\alpha}_{c_j})}, \frac{\exp(\mathcal{G}^{\beta}_{c_l})}{\sum_{j=1}^{n}\exp(\mathcal{G}^{\beta}_{c_j})}, \frac{\exp(\mathcal{G}^{w}_{c_l})}{\sum_{j=1}^{n}\exp(\mathcal{G}^{w}_{c_j})} \quad (5)$$

Here, the cell gradients, $\{\mathcal{G}^{\alpha}_{c_l}, \mathcal{G}^{\beta}_{c_l}, \mathcal{G}^{w}_{c_l}\}$, are the gradients with respect to the cell operation, edge pair, and weight parameter of the l-th cell c_l over s search steps, denoted as $\{\alpha_{c_l}, \beta_{c_l}, w_{c_l}\}$, computed with the distances from the meta parameters, i.e., $\{\alpha, \beta, w\}_{meta} - \{\alpha, \beta, w\}_{c_l}$, similarly in Eq. (3).

The cell coefficients, $\{C^{\alpha}_{c_l}, C^{\beta}_{c_l}, C^{w}_{c_l}\}$ in Eq. (5), strengthen the update of the cell that has been optimized far from the initial meta cell, with respect to the proportion of updates among the cells. By applying the cell coefficients that weigh the operation, edge pair, and weight parameters among the repeated cells, multiple cells are accelerated to converge into a single meta cell during two-fold meta-learning.

4.2 Two-fold Meta-Learning

By using the task and cell coefficients, *two-fold meta-learning* is performed to generate a single meta cell defined by the meta-cell operation, edge pair, and weight parameter, denoted as $\alpha_{meta-cell}$, $\beta_{meta-cell}$, and $w_{meta-cell}$, respectively, or $\{\alpha, \beta, w\}_{meta-cell}$ in short. Algorithm 1 summarizes the entire procedure of two-fold meta-learning.

It first constructs a n-cell meta architecture and the weight parameter, denoted as $\{\alpha, \beta, w\}_{meta}$, by stacking (unfolding) a pre-defined initial meta cell, $\{\alpha, \beta, w\}_{meta-cell}$, n times. Next, for each task T_i, the task architecture and weight parameters of T_i, i.e., $\{\alpha, \beta, w\}_{T_i}$, are composed (copied) from the meta architecture, $\{\alpha, \beta, w\}_{meta}$, and updated to minimize the task loss $\mathcal{L}_{T_{i,j}}$ over $1 \leq j \leq s$ steps via the gradient descent. After T_i has been optimized during the meta-task learning, the task coefficients for T_i, i.e., $\{\mathcal{T}^{\alpha}_{T_i}, \mathcal{T}^{\beta}_{T_i}, \mathcal{T}^{w}_{T_i}\}$, are computed as described in Eq. (4). Once finishing the meta-task learning, it proceeds to update the n-cell meta architecture and weight parameter, $\{\alpha, \beta, w\}_{meta}$, by summing up all the optimized task architectures and weight parameters, $\{\alpha, \beta, w\}_{T_i}$, based on the distance between the meta architecture and each task architecture [66], $\{\alpha, \beta, w\}_{meta} - \{\alpha, \beta, w\}_{T_i}$, as in Eq. (3), where each task is weighted by the task coefficients, $\{\mathcal{T}^{\alpha}_{T_i}, \mathcal{T}^{\beta}_{T_i}, \mathcal{T}^{w}_{T_i}\}$. After that, the cell coefficients for all $1 \leq l \leq n$ cells, i.e., $\{C^{\alpha}_{c_l}, C^{\beta}_{c_l}, C^{w}_{c_l}\}$, are computed as described in Eq. (5). Finally, the n-cell meta architecture $\{\alpha, \beta, w\}_{meta}$ is compressed (folded) back into a single meta cell as $\{\alpha, \beta, w\}_{meta-cell}$ through meta-cell learning by taking the distances between the meta cell and each cell of the meta architecture, i.e., $\{\alpha, \beta, w\}_{meta-cell} - \{\alpha, \beta, w\}_{meta_{c_l}}$, where each cell c_l is weighted by the cell coefficients, $\{C^{\alpha}_{c_l}, C^{\beta}_{c_l}, C^{w}_{c_l}\}$.

Algorithm 1 Two-Fold Meta-Learning

```
Input : Distribution over tasks P(T)
        Task learning rate λ_α, λ_β, λ_w
        Meta learning rate ξ_α, ξ_β, ξ_w
        Meta-cell learning rate δ_α, δ_β, δ_w
Result: Meta cell {α, β, w}_{meta-cell}
while not converged do
    {α, β, w}_{meta} ← stack({α, β, w}_{meta-cell})
    Sample tasks T_1 ... T_d from P(T)
    for all T_i do
        α_{T_i}, β_{T_i}, w_{T_i} ← α_{meta}, β_{meta}, w_{meta}
        for j ← 1, ..., s do
            α_{T_i} ← α_{T_i} − λ_α ∇_α L_{T_{i,j}}(α_{T_i}, β_{T_i}, w_{T_i})
            β_{T_i} ← β_{T_i} − λ_β ∇_β L_{T_{i,j}}(α_{T_i}, β_{T_i}, w_{T_i})
            w_{T_i} ← w_{T_i} − λ_w ∇_w L_{T_{i,j}}(α_{T_i}, β_{T_i}, w_{T_i})
        end
    end
    Compute {T^α_{T_i}, T^β_{T_i}, T^w_{T_i}} of T_i in Eq. (4)
    α_{meta} ← α_{meta} − ξ_α Σ_{T_i} T^α_{T_i}(α_{meta} − α_{T_i}) in Eq. (3)
    β_{meta} ← β_{meta} − ξ_β Σ_{T_i} T^β_{T_i}(β_{meta} − β_{T_i}) in Eq. (3)
    w_{meta} ← w_{meta} − ξ_w Σ_{T_i} T^w_{T_i}(w_{meta} − w_{T_i}) in Eq. (3)
    Compute {C^{alpha}_{c_l}, C^β_{c_l}, C^w_{c_l}} of c_l for all 1 ≤ l ≤ n in Eq. (5)
    α_{meta-cell} ← α_{meta-cell} − δ_α Σ_{c_l} C^α_{c_l}(α_{meta-cell} − α_{meta_{c_l}})
    β_{meta-cell} ← β_{meta-cell} − δ_β Σ_{c_l} C^β_{c_l}(β_{meta-cell} − β_{meta_{c_l}})
    w_{meta-cell} ← w_{meta-cell} − δ_w Σ_{c_l} C^w_{c_l}(w_{meta-cell} − w_{meta_{c_l}})
end
```

5 ON-DEVICE ARCHITECTURE SEARCH

The offline-learned meta cell is loaded onto the device and unfolded to construct the *initial n-cell architecture*. Then, a memory-efficient online on-device architecture search is performed for a target dataset through the *expectation-based operation and edge pair search*, and *step-by-step back-propagation*.

5.1 Expectation-based Architecture Search

Expectation-based Search. While differentiable architecture search [53, 59, 81, 100] achieves a search time reduction compared to the alternative methods [22], it still requires an excessive amount of memory. Since it utilizes the mixture of k candidate operations, e.g., convolution, pooling, etc., for every single edge in a cell, which is called *MixedOp* [59], weighted by the operation parameter α, intermediate outputs of all k operations should be stored in memory for the back-propagation, invoking the high memory consumption.

To enable memory-efficient differentiable architecture search, we propose the *expectation-based operation search*, which curtails the memory usage of k candidate operations by partially updating the selected subset of them. For each search step, it updates only

the selected q operations, where $q<k$, based on the expected value of the operation parameter α at the final search step f, which is denoted as α_f, estimated from the recent trend of gradients over the last t search steps.

At the j-th search step for $j<f$, where f is the final search step, the operation parameter α at the final search step f, α_f, is computed by the gradient descent given the loss $\mathcal{L}$ as:

$$\alpha_f=\alpha_{f-1}-\lambda_\alpha \frac{\partial \mathcal{L}_{f-1}}{\partial \alpha}=\alpha_j-\lambda_\alpha\left(\frac{\partial \mathcal{L}_j}{\partial \alpha}+\cdots+\frac{\partial \mathcal{L}_{f-2}}{\partial \alpha}+\frac{\partial \mathcal{L}_{f-1}}{\partial \alpha}\right) =\alpha_j-\lambda_\alpha \sum_{i=j}^{f-1} \frac{\partial \mathcal{L}_i}{\partial \alpha} \quad (6)$$

By assuming the gradient $\frac{\partial \mathcal{L}_i}{\partial \alpha}$ converges to some constant c (nearly zero) for all $i \leq f$ as f approaches to ∞, the expected value of α_f in Eq. (6), which is denoted as $\mathbb{E}[\alpha_f]$, can be estimated from the gradients over the last t search steps as:

$$\mathbb{E}[\alpha_f] = \mathbb{E}\Big[\alpha_j - \lambda_\alpha \sum_{i=j}^{f-1} \frac{\partial \mathcal{L}_i}{\partial \alpha}\Big] \simeq \alpha_j - \lambda_\alpha \frac{f-j}{t} \sum_{i=j-(t-1)}^{j} \frac{\partial \mathcal{L}_i}{\partial \alpha} \quad (7)$$

from that $\mathbb{E}\Big[\sum_{i=j}^{f-1} \frac{\partial \mathcal{L}_i}{\partial \alpha}\Big] \simeq \frac{f-j}{t} \sum_{i=j-(t-1)}^{j} \frac{\partial \mathcal{L}_i}{\partial \alpha}$ if $\frac{\partial \mathcal{L}_i}{\partial \alpha} \rightarrow c$. Even if $\frac{\partial \mathcal{L}_i}{\partial \alpha}$ does not converge to a constant c, the summation of the gradients over the last t steps is expected to help anticipate α_f by providing the recent trajectory in the parameter space, similarly working as the momentum factor [72].

Based on $\mathbb{E}[\alpha_f]$, which can be estimated at an arbitrary search step j as in Eq. (7), we select q operations having the highest $\mathbb{E}[\alpha_f]$ as a subset of k candidate operations in an edge and only update their α parameters for each search step, whereas those of non-selected operations are not updated.

While fully utilizing the forward execution of the entire architecture with all k operations, which allows for deriving stable model outputs, we store only the intermediate outputs of selected q operations in memory and discard those of the remaining $k-q$ operations, as the latter is not involved in the back-propagation computation. From this, maximum k times of memory can be saved when only one operation is selected for update, i.e., $q = 1$. By sampling a subset of operations based on $\mathbb{E}[\alpha_f]$ and only storing their intermediate outputs in memory, the partial update of MixedOp becomes possible without significant performance degradation of the architecture search, reducing both the memory usage and computation time required by the back-propagation.

Exploration vs. Exploitation. Although the expected value of the final operation parameter α, i.e., $\mathbb{E}[\alpha_f]$, can be effectively estimated, it might keep updating a similar subset of operations repeatedly, while the rest remains not selected for update, due to some potential induced bias of $\mathbb{E}[\alpha_f]$.

To tackle this problem, we incorporate the concept of *exploration and exploitation* [15] into the expectation-based operation search, which randomly samples q operations without considering $\mathbb{E}[\alpha_f]$ with the probability p, or alternatively, selects q operations based on $\mathbb{E}[\alpha_f]$ with the probability $1-p$ for each search step. Namely, the former explores new operations that may have lower $\mathbb{E}[\alpha_f]$ at the moment by updating their α, while the latter exploits $\mathbb{E}[\alpha_f]$, that reflects the trend of gradients known so far, to focus on the well-founded operations that are likely to be included in the final architecture.

Thus, with the exploration and exploitation being applied, a set of q operations O_q to be updated for each search step is composed from a set of all k candidate operations, O_k, as:

$$O_q=\begin{cases}\{o_1,\ldots,o_q\}\subseteq_R O_k & \text{if } B=1\\ \{o_1,\ldots,o_q\}\subseteq O_k \text{ s.t. } \mathbb{E}[\alpha_f^{o_u}]\geq\mathbb{E}[\alpha_f^{o_v}] \text{ for } u\leq q<v & \text{if } B=0\end{cases} \quad (8)$$

where $\subseteq_R$ denotes the random subset relation, $\mathbb{E}[\alpha_f^{o_u}]$ and $\mathbb{E}[\alpha_f^{o_v}]$ is the expected value of α at the final search step f for operation o_u and o_v, respectively, and B is a random variable from the Bernoulli distribution [86] as $B \sim Bern(p)$ taking the value 1 with probability p, and 0 with probability $1-p$.

Expectation-based Edge Pair Search. Given a set of edges connecting m nodes in a cell, where each edge contains k candidate operations on it, as shown in Fig. 3, we compose a set of possible pairwise edges and select a subset of them to be included in the final architecture. We apply the concept of pairwise edges to On-NAS as they are known to enhance the efficiency of architecture search and accelerate the optimization process [23]. To determine which edge pairs to be included in the final architecture, the edge pair parameter β is assigned to each possible pair of edges and updated over search steps in a similar way to the operation parameter α. Then, at the final search step f, a designated number of edge pairs with the highest β are chosen as the final architecture.

To enable memory-efficient edge pair search by enabling a partial update of the edge pair parameter β, we propose the *expectation-based edge pair search*, with which the expected value of β at the final search step f, denoted as $\mathbb{E}[\beta_f]$, is estimated similarly to the expected value of the operation parameter α, i.e., $\mathbb{E}[\alpha_f]$ in Eq. (7). Based on the expected value of β at the final step f, i.e., $\mathbb{E}[\beta_f]$, a set of g edge pairs $\mathcal{E}_g$ to be updated for each search step is composed from a set of all possible h edge pairs, $\mathcal{E}_h$, where $g < h$, as:

$$\mathcal{E}_g=\begin{cases}\{e_1,\ldots,e_g\}\subseteq_R \mathcal{E}_h & \text{if } B=1\\ \{e_1,\ldots,e_g\}\subseteq \mathcal{E}_h \text{ s.t. } \mathbb{E}[\beta_f^{e_u}]\geq\mathbb{E}[\beta_f^{e_v}] \text{ for } u\leq g<v & \text{if } B=0\end{cases} \quad (9)$$

where $\mathbb{E}[\beta_f^{e_u}]$ and $\mathbb{E}[\beta_f^{e_v}]$ is the expected value of β at the final search step f for edge e_u and e_v, respectively.

As the intermediate outputs of non-selected edge pairs are not stored in memory, further memory reduction is achieved on top of the expectation-based operation search. If only a single pair of edges is chosen to be updated for each node in a cell consisting of m nodes, i.e., g=$2(m-1)$ and h=$m(m-1)/2$, at most $m/4$ times of memory can be reduced. Combined together, the expectation-based operation and edge pair search reduce the memory usage of architecture search up to $k \cdot m/4$ times without a significant performance drop.

5.2 Step-By-Step Back-Propagation

To further reduce the memory consumption of on-device architecture search, we propose the *step-by-step back-propagation*, which optimizes the architecture and weight parameters (α, β, and w) of an individual cell discontinuously from each other with a reasonable cost of time by progressively storing the intermediate cell gradients in the computational chain, as a variant of re-materialization (gradient check-pointing) [31].

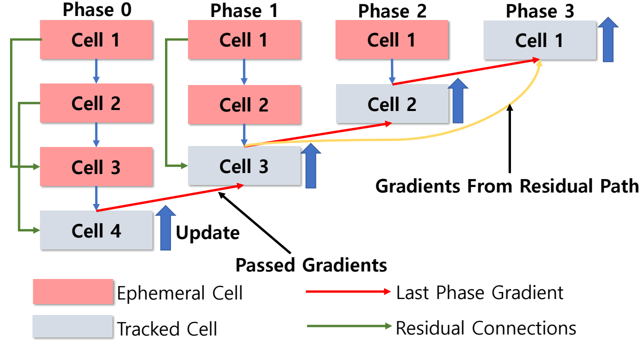

Figure 6: For a model architecture consisting of n cells, the peak memory consumption of the back-propagation is reduced to $1/n$ at maximum by evicting the intermediate outputs of all ephemeral cells except for the designated tracked cell, starting backward from the last cell. "Tracked cell" represents the designated cell to be updated at the current phase, and "ephemeral cell" represents the cell whose intermediate outputs are discarded from memory. "Last phase gradient" depicts that the required gradients are passed from the previous phase to the current phase to update the tracked cell.

In the cell motif-based architecture of On-NAS, a total of n cells are stacked and connected through their inputs and outputs with residual connections [59], considering the input and output of each cell as a computational node and their connection as an edge. Based on that, the step-by-step back-propagation updates the operation, edge pair, and weight parameters (α, β, and w) of each cell one by one (phase by phase) in the backward chain of gradient computation by utilizing the information passed from the previous phase.

As shown in Fig. 6, for the $(n-l)$th phase of the step-by-step back-propagation, the forward propagation is first executed from the first cell to the l-th designated cell (*tracked cell*), and then only the gradient of the tracked cell is computed and stored in memory, along with the cell output, while all the other outputs of preceding cells (*ephemeral cells*) acquired during the forward propagation are evicted from memory. Afterward, at the next $(n-l+1)$th phase, the operation, edge pair, and weight parameters (α, β, and w) of the designated $(l+1)$-th cell are updated by using the gradient and output stored at the previous $(n-l)$th phase. By storing (tracking) only a few intermediate gradients and cell outputs for each phase, requiring much less memory compared to saving all the intermediate outputs of all cells, each cell can be independently updated only using $1/n$ of memory. As the forward propagation is not fully executed except for the first phase, the time required by unnecessary forward propagation becomes minimized.

Given o_{c_l} as the output of the l-th cell c_l, the gradient of the loss $\mathcal{L}$ with respect to the weight parameters of the two consecutive $(l+1)$-th and l-th cell, $w_{c_{l+1}}$ and w_{c_l}, respectively, are calculated by the chain of the back-propagation [78] as:

$$\frac{\partial \mathcal{L}}{\partial w_{c_{l+1}}} = \frac{\partial \mathcal{L}}{\partial o_{c_{l+3}}} \left(\frac{\partial o_{c_{l+3}}}{\partial o_{c_{l+2}}} \frac{\partial o_{c_{l+2}}}{\partial o_{c_{l+1}}} + \frac{\partial o_{c_{l+3}}}{\partial o_{c_{l+1}}} \right) \frac{\partial o_{c_{l+1}}}{\partial w_{c_{l+1}}} \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial w_{c_l}} = \frac{\partial \mathcal{L}}{\partial o_{c_{l+2}}} \left(\frac{\partial o_{c_{l+2}}}{\partial o_{c_{l+1}}} \frac{\partial o_{c_{l+1}}}{\partial o_{c_l}} + \frac{\partial o_{c_{l+2}}}{\partial o_{c_l}} \right) \frac{\partial o_{c_l}}{\partial w_{c_l}} \tag{11}$$

Here, the first and second term is for the connection with the next cell and the residual connection with one after the next.

To compute the gradient of the l-th cell, $\frac{\partial \mathcal{L}}{\partial w_{c_l}}$ in Eq. (11), at the $(n{-}l{+}1)$-th phase, we store $\frac{\partial \mathcal{L}}{\partial o_{c_{l+2}}} = \frac{\partial \mathcal{L}}{\partial o_{c_{l+3}}} \frac{\partial o_{c_{l+3}}}{\partial o_{c_{l+2}}}$, $o_{c_{l+2}}$, and $o_{c_{l+1}}$ obtained from Eq. (10) in memory at the previous $(n{-}l)$-th phase and pass them to the next $(n{-}l{+}1)$-th phase. By doing that, when it proceeds to the next $(n{-}l{+}1)$-th phase, the gradient of the l-th cell, $\frac{\partial \mathcal{L}}{\partial w_{c_l}}$ in Eq. (11) can be computed from the passed $\frac{\partial \mathcal{L}}{\partial o_{c_{l+2}}}$, $o_{c_{l+2}}$, and $o_{c_{l+1}}$, in addition to o_{c_l} and w_{c_l} which can be obtained from the current $(n{-}l{+}1)$-th phase, resulting in the exactly same update to normal back-propagation consuming up to n times more memory.

6 EXPERIMENTS

Implementation. We implement the proposed On-NAS using PyTorch [68], the most popular deep learning framework, to provide development convenience and portability to various embedded platforms, which is publicly available at a git repository[1]. To evaluate the offline two-fold meta-learning, we use an NVIDIA RTX 3090 GPU. For the evaluation of online on-device architecture search, we deploy On-NAS onto NVIDIA Jetson Nano equipped with 2GB of unified memory [11], where 1GB of memory is occupied by the system, and the remaining 1GB of memory is left for On-NAS.

Evaluation. We first evaluate the effectiveness of the task and cell coefficients used in two-fold meta-learning, along with the storage usage of the meta cell. We next evaluate on-device architecture search and measure the memory consumption under two dataset shift scenarios, i.e., 1) few-shot learning with two widely used benchmark datasets (miniImageNet [88] and Omniglot [48]) and 2) full-task adaptation with two datasets (CIFAR-10 [44] and CIFAR-100 [44]) that are popularly used in the NAS community to assess the performance of architecture search from scratch.

Search Space. As established in previous works [23, 59], we opt for a 4-cell structure for our over-parameterized model, consisting of 2 normal cells and 2 reduction cells. To conduct fair comparisons, we also maintain consistency in other hyperparameter settings, including the number of nodes, edge connectivity, and candidate operations as established in previous work, MetaNAS [23]. To clarify, our set of candidate operations consists of *Conv3x3, DilatedConv3x3, Conv1x5-5x1, MaxPool3x3, AvgPool3x3, SepConv3x3 and SkipConnect.* Following MetaNAS [23], the cells have three intermediate nodes, where each node passes an output to the next nodes, resulting in a total of nine edges with *MixedOP* [59] for each cell. As we allow two different edges at most for the input of single nodes after the search, the total number of possible architecture configurations for each cell is equivalent to $2 \times 3^2 \times 7^6$, resulting in $2^2 \times 3^4 \times 7^{12}$ possible configurations considering normal cell and reduction cell for total model architecture, which is identical to previous work [23], comparable to other DARTS-based methods [59, 100].

Baselines. We compare the memory usage and performance of On-NAS against four state-of-the-art cell-based meta-learning-applied and/or differential architecture search methods, i.e., DARTS [59], MetaNAS [23], ProxylessNAS [10], and PC-DARTS [95].

6.1 Two-fold Meta-Learning

We first examine the performance of offline two-fold meta-learning by measuring the effectiveness of the task and cell coefficients given in Eq. (4) and Eq. (5) with the accuracy of the meta cell over meta epochs. Fig. 7 plots the test accuracy of 5-shot learning on miniImageNet [88] and Omniglot [48] over 15,000 meta epochs of GPU training with the four cases of two-fold meta-learning: 1) without coefficients, 2) task coefficients only, 3) cell coefficients only, and

4) both task and cell coefficients applied. It shows that applying both task and cell coefficients enables the meta cell to achieve the lowest loss (also the highest accuracy) and faster optimization for both miniImageNet and Omniglot by minimizing the loss more efficiently when compared to the cases without coefficients. For example, "both_coeff" of miniImageNet in Fig. 7 starts to touch the loss value of 1.20 around 5,000 epochs and finally arrives at 1.08, while "no_coeff" records 1.22 at its lowest. While utilizing the task and cell coefficient alone does not provide notable performance improvement, using them together gives a synergy in optimizing the meta cell as they complement each other and work as a momentum guiding the direction of the update towards the optimal point.

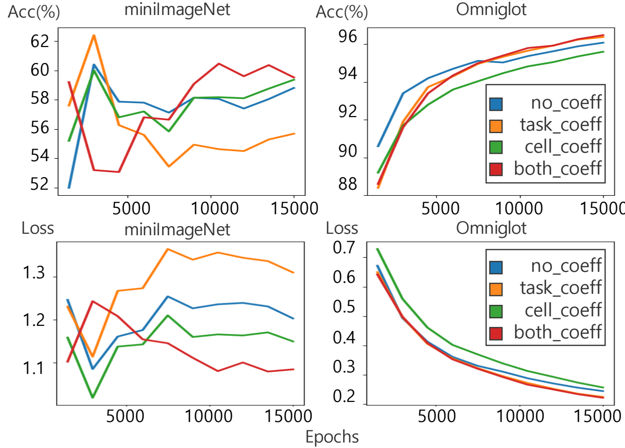

Figure 7: The performance of the meta cell pre-trained for 5-shot learning of miniImageNet [88] and Omniglot [48] over meta epochs. The trajectory of evaluation loss and accuracy demonstrates that two-fold meta-learning with both task and cell coefficients enables it to converge faster to the lowest loss (highest accuracy).

Fig. 8 shows examples of the normal and reduction cell [103], unfolded from a single meta cell learned by two-fold meta-learning for miniImageNet [88], where its search space, e.g., candidate operations, is predefined identically with MetaNAS [23]. Those two cells consist of separate operation and edge pair parameters (α and β) but with the same meta weight parameters (w) based on the weight sharing [69] implemented in On-NAS. As both α and β require only a trivial amount of storage, a single meta cell can be space-efficiently unfolded into the full model architecture and folded vice versa, by utilizing not only α and β but also w.

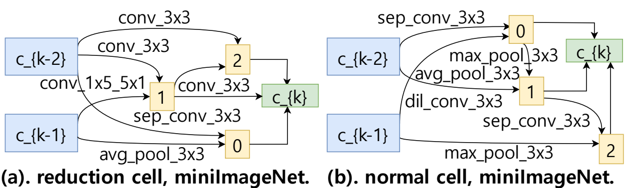

Figure 8: The visualization of two classes of cells in the form of a directed acyclic graph (DAG), i.e., the reduction and normal cell [103], unfolded from a single meta cell found by two-fold meta-learning of miniImageNet [88]. Blue and green boxes denote the cell input and output, respectively, and yellow boxes represent the nodes connected through the edges of operations, e.g., conv and pooling.

For instance, the four cells of miniImageNet [88] (2.6MB) in Fig. 8 are folded into a single meta cell (0.6MB), achieving 4.3x storage space saving. As the model architecture has more cells, more storage reduction is achieved, e.g., we observe 20x storage reduction (from 2,600MB to 128MB) in the scale-up DARTS-based model consisting of 20 cells, 7 nodes, and 256 conv channels [92].

6.2 On-Device Architecture Search

We next evaluate online on-device architecture search on two dataset shift settings, i.e., few-shot and full-task learning.

Few-Shot Learning. We measure the peak memory usage, model performance (test accuracy), and search time of on-device architecture search in few-shot learning settings, which requires a small number of architecture search steps, and compare them against three cell-based meta-learning-applied and/or differential architecture search methods, i.e., DARTS [59], MetaNAS [23], Proxyless-NAS [10], and PC-DARTS [95].

	NAS	Memory(MB)	Acc(%)	Time(Device/GPU)
miniImageNet				
5-shot	DARTS	4123	62.40±0.91	**X** / 5.2s
	MetaNAS	5100	62.00±0.87	**X** / 5.6s
	PC-DARTS	1700	59.67±1.06	**X** / 5.1s
	ProxylessNAS	2235	62.00±1.25	**X** / 20.1s
	On-NAS (Ours)	**243**	**63.33±0.11**	8m 15s / 27.7s
1-shot	DARTS	1616	45.67±2.84	**X** / 2.5s
	MetaNAS	1668	46.80±1.84	**X** / 2.5s
	PC-DARTS	567	42.73±1.59	1m 02s / 2.2s
	ProxylessNAS	753	41.19±1.14	**X** / 8.8s
	On-NAS (Ours)	**158**	**46.80±1.42**	3m 06s / 13.6s
Omniglot				
	NAS	Memory(MB)	Acc(%)	Time(Device/GPU)
5-shot	DARTS	534	98.82±0.16	4m 10s / 14.1s
	MetaNAS	564	98.83±0.22	4m 33s / 14.1s
	PC-DARTS	189	90.12±0.57	3m 10s / 13.9s
	ProxylessNAS	272	96.08±0.34	13m 46s / 48.1s
	On-NAS (Ours)	**26**	**98.98±0.10**	19m 26s / 1m 38s
1-shot	DARTS	192	87.10±0.51	1m 58s / 8.1s
	MetaNAS	202	85.62±0.21	2m 10s / 8.2s
	PC-DARTS	66	69.12±0.85	2m 21s / 8.1s
	ProxylessNAS	112	91.83±1.24	5m 41s / 25.4s
	On-NAS (Ours)	**25**	**90.22±0.85**	11m 10s / 56.0s

Table 2: The few-shot performance comparison among NAS methods. 'Device' and 'GPU' time indicates the required time for 10 search steps on Jetson Nano and RTX 3090 GPU, respectively. 'Device' shows the time required for architecture search on the target device, and 'X' indicates that the NAS algorithm is not able to run on the target device as lack of hardware (memory) resources.

Tab. 2 summarizes the experiment result of 5-way/20-way 5-shot and 1-shot adaptation on miniImageNet [88] and Omniglot [48]. Each task takes 10 architecture search steps for dataset adaptation on Jetson Nano with 1GB of free memory, starting from the unfolded meta cell that is offline pre-trained for 30,000 meta-epochs by two-fold meta-learning. All experiments are run under three independent trials with random seeds and tested with 100 sampled tasks per run, following the standard evaluation method of few-shot learning [23, 75].

As shown in Tab. 2, On-NAS achieves comparable or slightly superior performance over the four baselines, e.g., 1.3% higher accuracy than MetaNAS [23] with 20x less memory consumption. Although more time is required for architecture search, it enables NAS tasks to be performed on the device with limited memory, e.g., reducing 4,123MB memory to 243MB for 5-shot learning of DARTS on miniImageNet, which has been impossible with existing

miniImageNet						
	5-shot			1-shot		
q	Mem(MB)	Acc(%)	Time	Mem(MB)	Acc(%)	Time
1	361	63.50±0.99	8m 05s	132	44.87±0.81	3m 04s
2	431	63.33±0.11	8m 34s	157	46.87±1.64	3m 19s
3	505	62.12±1.53	9m 11s	189	44.40±1.72	3m 26s
4	576	61.33±0.92	9m 35s	201	50.53±2.78	3m 32s
5	645	62.25±0.79	10m 11s	233	46.80±2.61	3m 42s
6	713	61.83±0.99	10m 35s	257	50.87±1.23	3m 46s
7	787	62.02±0.87	11m 18s	277	48.47±1.16	3m 54s

Omniglot						
	5-shot			1-shot		
q	Mem(MB)	Acc(%)	Time	Mem(MB)	Acc(%)	Time
1	59	98.67±0.22	19m 23s	29	86.48±0.30	10m 58s
2	64	98.98±0.10	20m 03s	32	90.22±0.28	11m 18s
3	73	98.97±0.18	21m 23s	35	91.47±0.62	11m 51s
4	81	98.70±0.37	22m 03s	38	91.92±0.54	12m 21s
5	90	99.08±0.18	22m 50s	41	91.58±0.21	12m 35s
6	98	98.77±0.35	23m 15s	43	91.57±0.02	13m 09s
7	102	98.67±0.22	24m 00s	49	92.42±0.12	13m 33s

Table 3: The performance of On-NAS on 5-shot and 1-shot learning of miniImageNet [88] (5-way) and Omniglot [48] (20-way) with different numbers of operations (q) to be selected for the update of their operation parameter (α) during on-device architecture search.

NAS. Although the Omniglot experiment shows that the baselines could run on the device without applying On-NAS as their memory requirements are smaller than the device memory capacity (1GB), it implies that On-NAS can be utilized for on-device architecture search on a more resource-constrained low-end embedded platform, e.g., STM32 Cortex-M7 boards [5] with 32MB RAM, as memory consumption of architecture search is decreased 8x from 202MB (MetaNAS) to 25MB (On-NAS).

Memory Reduction. Fig. 9 provides the breakdown of memory reduction achieved by On-NAS. In 20x of total memory reduction, the expectation-based operation search, combined with the gradient accumulation [55, 85], contributes the most, i.e., 14x reduction, followed by the step-by-step back-propagation (1.4x) and expectation-based edge pair search (1.1x). It demonstrates that the real memory reduction closely matches our theoretical analysis with some implementation overheads increasing memory usage in practice. For instance, the expectation-based edge pair search in miniImageNet is expected to reduce the memory usage by up to 4x as the total number of cells in its architecture is four, i.e., n=4, and it actually reduces 1.4x of memory in the experiment, validating the bound of memory saving achieved by On-NAS.

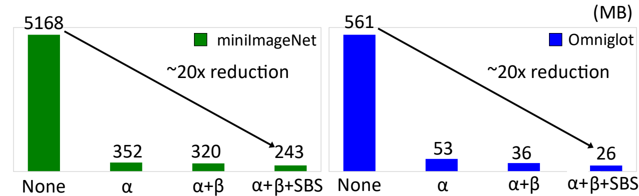

Figure 9: The efficacy of each memory-saving on-device architecture search method of few-shot learning on miniImageNet [88] and Omniglot [48]. α, β, and SBS denotes expectation-based operation search, edge pair search, and step-by-step back-propagation, respectively. By applying them all together, 20x memory saving is achieved.

Expectation-based Search. Tab. 3 summarizes the memory usage, model performance (test accuracy), and search time when different numbers of operations (q) are selected for the update at each search step during the expectation-based operation search. The result shows that almost similar accuracy is achieved in all numbers of operations, e.g., two operations record even 1.3% higher accuracy than the entire seven operations on 5-shot learning of miniImageNet while consuming 54.8% of memory and 16% less time. Fig. 10 provides statistics of the test accuracy in Tab. 3, showing that the overall performance of miniImageNet and Omniglot does not severely degrade even though the number of operations decreases, implying that the expectation-based search enhances the efficiency of operation search with a slight impact on the accuracy. Since the model performance tends to be maintained with small numbers of operations, while the memory consumption is reduced, On-NAS enables the user to adjust the number of operations according to the memory budget of the target device without significant performance drops, making On-NAS a memory-aware and flexible on-device NAS solution. Fig. 11 shows the final cell architecture found for few-shot learning of Omniglot [48] on the device.

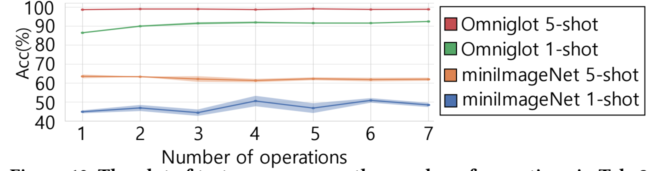

Figure 10: The plot of test accuracy over the number of operations in Tab. 3, including the mean and standard deviation, measured with three independent runs under the identical setting to Tab. 2.

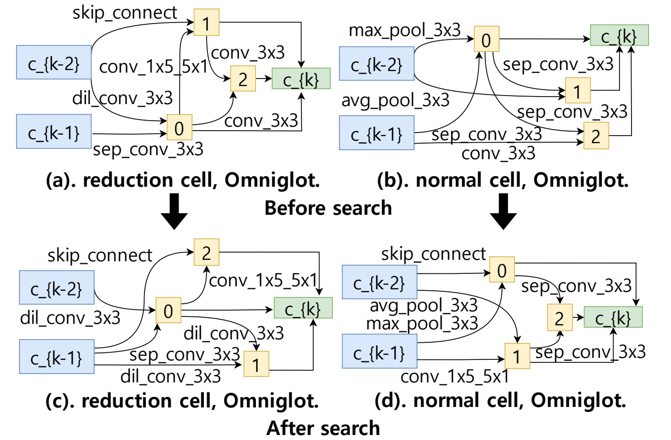

Figure 11: The visualization of the cell architectures (reduction and normal cell [103]) in the form of a directed acyclic graph for Omniglot [48]. (a) and (b): The initial architecture before online on-device architecture search. (c) and (d): The final architecture after online on-device architecture search.

Full-Task Adaptation. Considering the limited computing power of many embedded devices, On-NAS can be best utilized in practice for an environment where it finds the optimal model architecture by learning a relatively small number of data samples, as shown in the few-shot experiments given above. To assess the potentials and limits of On-NAS on more serious dataset shift scenarios where a large amount of data samples needs to be learned, we perform On-NAS for full-task adaptation required to find the optimal model architecture from scratch using all the available data samples.

Tab. 4 shows the memory usage, model performance (test accuracy), and search time of full-task adaptation to CIFAR-10 [44] and CIFAR-100 [44] over 391 × 50 and 391 × 100 search steps, which is much bigger than 10 search steps taken by few-shot learning on

miniImageNet. We observe that On-NAS executing the expectation-based operation search with two operations achieves 3.63% higher accuracy on CIFAR-10 than PC-DARTS [95], one of the state-of-the-art memory-efficient DARTS-based NAS. When five operations are updated instead of two, On-NAS achieves comparable performance to DARTS [59] on CIFAR-100, i.e., 56.58% vs. 55.76%, with 10x less memory being consumed (3,479MB vs. 325MB).

CIFAR-10			
NAS	Memory(MB)	Acc(%)	Time(Device/GPU)
DARTS	3460	87.47	**X** / 0.27 days
PC-DARTS	1070	77.83	**X** / 0.10 days
On-NAS (2 Ops)	**290**	**81.46**	2.21 / 0.17 days
On-NAS (5 Ops)	**397**	**86.03**	2.41 / 0.18 days
CIFAR-100			
NAS	Memory(MB)	Acc(%)	Time(Device/GPU)
DARTS	3479	56.58	**X** / 0.24 days
PC-DARTS	1169	50.73	**X** / 0.24 days
On-NAS (2 Ops)	**238**	**50.77**	4 / 0.31 days
On-NAS (5 Ops)	**325**	**55.76**	4.21 / 0.33 days

Table 4: Full-task adaptation performed from scratch on CIFAR-10 [44] and CIFAR-100 [44]. The performance of On-NAS is compared to the two baselines (DARTS [23] and PC-DARTS [95]). The search time is measured in GPU days on Jetson Nano and a server-grade GPU (RTX 3090). 'X' means the NAS could not run on the device.

Fig. 12 shows the contribution of each memory-saving method used in On-NAS on full-task adaptation. Similar to few-shot learning, 20x of memory is saved at maximum by the expectation-based operation and edge pair search, and step-by-step back-propagation, verifying that they also apply to more serious dataset shift problems that require searching for the optimal model architecture via full-task adaptation.

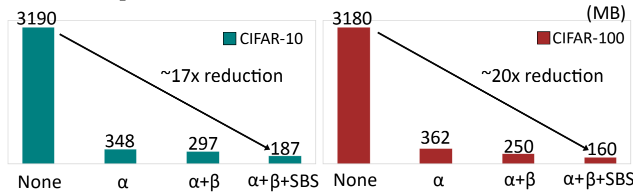

Figure 12: The efficacy of each memory-saving on-device architecture search method of full-task adaptation on CIFAR-10 [44] and CIFAR-100 [44]. α, β, **and SBS denotes expectation-based operation search, edge pair search, and step-by-step back-propagation, respectively. By applying them all, more than 17x of memory usage is reduced.**

Search Time. Full-task adaptation inevitably entails a large number of search steps (epochs), e.g., 391×50 steps (50 epochs) in our experiment, as it searches for the optimal architecture using the entire training dataset from scratch. Consequently, the amount of time spent by On-NAS becomes longer than GPU-running cases as provided in Tab. 4, e.g., 4 GPU days on CIFAR-100, due to the significant gap in computing power between a server-grade GPU and an embedded device, e.g., 35 TFLOPS (RTX 3090) vs. 472 GFLOPS (Jetson Nano). However, On-NAS seems to take a relatively short search time by effectively overcoming scarce resources of the device, i.e., 12x longer search time given 75x less computing power and 10x smaller memory when compared to using a GPU. Although computation (time) complexity is not the main topic of this paper, it indicates that On-NAS is not only memory-efficient but also compute-efficient. Nevertheless, it needs to be further investigated how to search model architectures in full-task adaptation with more reasonable time costs by lowering the search time complexity while keeping low memory consumption enabled by On-NAS.

7 LIMITATIONS AND FUTURE WORKS

Architecture Search Time. As shown in our experiments, On-NAS achieves competitive performance in few-shot learning on a device with limited memory by finding optimal model architectures over few steps within an acceptable period of time. However, when it comes to full-task learning that finds an optimal architecture using the entire training dataset from scratch, On-NAS inevitably entails a larger number of search steps, e.g., 600 epochs. Consequently, the amount of time spent by On-NAS becomes much longer when compared to using a high-end GPU, 4.21 days, in part prolonged by the step-by-step backpropagation. It could primarily be blamed on the significant gap in computing power between GPUs and embedded devices. Nevertheless, there exist trade-offs between the architecture search time and memory consumption in On-NAS, which should be further investigated to fully apply it to full-task learning with reasonable time costs. As a future work, a new approach that can decrease the search time complexity with low peak memory consumption is to be studied so that On-NAS can become feasible in both time- and memory-wise perspectives.

Online Meta Cell Learning. The proposed meta cell is generated from offline meta-learning of multiple datasets (tasks) and then loaded into the device, which enables fast and effective on-device architecture search for new real datasets, However, it may need to be updated (re-learned) on the device as real data distributions keep changing over time in practice. Yet the current framework requires server-side transfer to update the meta cell, we look forward to implementing meta-learning on the target device as a future work, to update the meta cell continuously. By continuously updating the meta cell with the real datasets on the device, it becomes able to better adapt to ever-changing data when compared to using the static meta cell not updated at all once loaded. One possible way of updating the meta cell on the device is to apply a continual learning method, such as EWC [42] or gradient-based sample selection [2], to prevent catastrophic forgetting [27, 47] of the meta cell. It allows maintaining the previously-learned data of the meta cell by reducing the risk of over-fitting to new data while gradually accumulating new information learned from new datasets. As more real datasets are aggregated to the meta cell over time, the adaptability of the meta cell to the continually-changing data gets improved, which is also expected to accelerate in finding optimal architectures.

Extension to Other NAS Algorithms. Based on the fact that we employ the Reptile [66] meta-learning algorithm, and cell-based architecture, utilized in previous works [53, 59, 100] which are currently regarded as state-of-the-art approaches achieving the best performance within a reasonable time on many NAS problems [14, 77], it is natural to extend On-NAS to other gradient-based meta-learning algorithms [26, 75, 80] and various other NAS techniques [18, 21, 50]. As meta-learning itself is less influential for the reduction of peak memory required, most of the gradient-based meta-learning algorithms can be seamlessly applied to current On-NAS. Nonetheless, there are NAS algorithms that are not based on cell-based structures or repeated motifs, which focus on the diversity of a module while compensating search time with performance that provides comparable results to the cell-based differentiable NAS. As the backbone of On-NAS is the cell-based differentiable NAS, it seems not obvious how to extend it to such NAS algorithms

with alternative structures. Considering the continuous improvement and development of NAS methods, the approaches that decrease the peak memory usage of NAS proposed in this paper need to apply to non-cell-based and non-differential NAS. Finding some common search procedures among NAS methods and analyzing their memory consumption patterns would help generalize On-NAS to a NAS-agnostic memory-efficient on-device NAS solution.

8 RELATED WORK

On-Device NAS. To the best of our knowledge, no existing works fully support neural architecture search (NAS) on the device. Instead, many on-device NAS [56, 62, 84] perform architecture search in a hybrid form with the collaboration of a cloud server by delegating the memory-consuming and compute-intensive architecture search process to the cloud and only evaluate the performance of the resulting architectures on the device. Hence, the network connectivity should be retained between the device and server during the search procedure, which can hardly address the dataset shift problem by the device itself when the network connection becomes unavailable. Unlike those so-called on-device NAS, On-NAS searches for the best architecture and weight parameter entirely on the device without the burden of connectivity, privacy issues, and data transfer bottlenecks, enabling a genuine architecture search on the device for the first time.

Efficient NAS. To lower the inefficiency of neural architecture search [22], the research domain of efficient NAS [61] has been facilitated in recent years [8, 58, 69, 103]. In particular, PC-DARTS [95] suggests partial connections of convolution channels to reduce the memory consumption of DARTS [59]. However, the scope of its partial search is limited only to the convolution where the channels are randomly sampled without any principle, unlike On-NAS that allows partial search on all types of operations as well as edge pairs based on the expectation estimation, saving much more memory , i.e., up to 4% (On-NAS) vs. 33% (PC-DARTS). FP-DARTS [90] utilizes two over-parameterized networks, each consisting of half of the candidate operations, and selectively controls the search path using a binary gate to reduce memory consumption. Similarly, ProxylessNAS [10] decreases the memory usage of NAS by binarizing possible paths. Although they can simplify the search path similar to On-NAS, they utilize only the selected paths in the forward execution, resulting in unstable model output, negatively affecting the search performance. On the contrary, On-NAS executes all forward paths only once, which enables fast and stable architecture search while reducing memory consumption in multiple facets, i.e., selectively determining a few operations and edge pairs on top of the step-by-step back-propagation. Moreover, all of the existing works are not designed for resource-constrained embedded devices, unlike On-NAS that enables systematic and flexible memory-efficient architecture search on embedded devices in accordance with their tight memory budgets.

Meta-Learning-applied NAS. To compensate for the generalization issue of deep models, gradient-based meta-learning has been proposed [25, 66, 75]. On that basis, On-NAS integrates gradient-based meta-learning into differential architecture search by combining Reptile [66] and DARTS [59] in a similar way to MetaNAS [23]. By doing so, it can set up the initial meta architecture from various task-specific architectures, enabling agile and better architecture adaptation to new data while lessening the burden of architecture search on the device. Unlike MetaNAS [23], On-NAS effectively condenses n cells into a single meta cell without significant performance compensation through two-fold meta-learning with the task and cell coefficients. Thus, the device storage required to employ the initial meta architecture of n cells decreases into $1/n$ so that the meta cell becomes fit to the tiny storage of the device. Alternatively, n meta cells trained with different domains would be deployed on the device to be accordingly utilized depending on the domain of new datasets (tasks), when given the same storage capacity.

On-Device Training. Recently, many approaches [19] have been proposed to train deep models on resource-constrained devices [49]. TinyTL [9] introduces on-device transfer learning by exploiting only the bias with the unique module while freezing the weight parameters. Another study [30] tries to enable memory-efficient training on the device by combining various memory-saving training techniques. Since on-device NAS can be seen as a repeated process of training multiple candidate architectures, one might wonder if it is possible to use existing on-device training methods as an alternative solution for on-device NAS. However, those on-device training techniques [9, 19, 30, 49, 73] only update the weight parameters of a fixed architecture, which cannot fully adapt to a new dataset as demonstrated in previous works [29, 43, 76]. For example, p-Meta [73] employed meta-learning and memory-efficient methodologies for on-device learning, similar to On-NAS. However, p-Meta focuses on choosing adaptation-critical layers and channels for memory efficiency, in contrast to On-NAS which focuses on memory efficiency over repeated motifs based on estimation of parameter optimization. Furthermore, it is noteworthy that p-Meta [73] is dedicated to adjusting model weights for new distribution, in contrast to our method which optimizes its architecture and weight parameters, with the utilization of neural architecture search algorithms. Also, neural architecture search is not a mere sum of model training, as it jointly optimizes the model architecture and weight parameter simultaneously, which is not only more challenging but also more resource-demanding. In this paper, we show that On-NAS would be a flexible and capable NAS solution for continuously-shifting field data by adapting both weight parameters and model architecture.

9 CONCLUSION

This paper proposes a memory-efficient on-device neural architecture search (NAS), which we call On-NAS, which drastically reduces the massive memory requirement of NAS on the device. Starting from the meta cell pre-trained through two-fold meta-learning proposed to condense multiple cells into one, On-NAS finds the optimal architecture for the target dataset on a resource-constrained embedded device with the expectation-based operation and edge pair search, and the step-by-step back-propagation. They collectively enable a memory-efficient standalone on-device NAS for the first time by solely performing NAS on the device without help from external systems. The evaluations demonstrate that On-NAS composes optimal model architectures competitive to GPU-based NAS while consuming 20x less memory and 4x less storage space for various dataset shifts on the device.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277383), Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01336, Artificial Intelligence graduate school support(UNIST)), and the Settlement Research Fund (1.210142.01) of UNIST (Ulsan National Institute of Science & Technology).

REFERENCES

[1] Rocío Alaiz-Rodríguez and Nathalie Japkowicz. 2008. Assessing the impact of changing environments on classifier performance. In *Advances in Artificial Intelligence: 21st Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2008 Windsor, Canada, May 28-30, 2008 Proceedings 21*. Springer, 13–24.

[2] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. 2019. Gradient based sample selection for online continual learning. *Advances in neural information processing systems* 32 (2019).

[3] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. 2021. A comprehensive survey on hardware-aware neural architecture search. *arXiv preprint arXiv:2101.09336* (2021).

[4] John Bridle. 1989. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems* 2 (1989).

[5] Geoffrey Brown. 2012. Discovering the STM32 microcontroller. *Cortex* 3, 34 (2012), 64.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Vinula Uthsara Buthgamumudalige and Torin Wirasingha. 2021. Neural architecture search for generative adversarial networks: A review. In *2021 10th International Conference on Information and Automation for Sustainability (ICIAfS)*. IEEE, 246–251.

[8] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791* (2019).

[9] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems* 33 (2020), 11285–11297.

[10] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[11] Stephen Cass. 2020. Nvidia makes it easy to embed AI: The Jetson nano packs a lot of machine-learning power into DIY projects-[Hands on]. *IEEE Spectrum* 57, 7 (2020), 14–16.

[12] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. 2021. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF international conference on computer vision*. 12270–12280.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[14] Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, and Arun K Somani. 2022. Neural architecture search for transformers: A survey. *IEEE Access* 10 (2022), 108374–108412.

[15] Melanie Coggan. 2004. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University* (2004).

[16] Gerald Coley. 2013. Beaglebone black system reference manual. *Texas Instruments, Dallas* 5 (2013), 2013.

[17] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. 2016. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913* (2016).

[18] Jiequan Cui, Pengguang Chen, Ruiyu Li, Shu Liu, Xiaoyong Shen, and Jiaya Jia. 2019. Fast and Practical Neural Architecture Search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[19] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. 2021. A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things* 2, 3 (2021), 1–49.

[20] Yadong Ding, Yu Wu, Chengyue Huang, Siliang Tang, Yi Yang, Longhui Wei, Yueting Zhuang, and Qi Tian. 2022. Learning to learn by jointly optimizing neural architecture and weights. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 129–138.

[21] Xuanyi Dong and Yi Yang. 2019. Searching for a Robust Neural Architecture in Four GPU Hours. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[22] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.

[23] Thomas Elsken, Benedikt Staffler, Jan Hendrik Metzen, and Frank Hutter. 2020. Meta-learning of neural architectures for few-shot learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12365–12375.

[24] Yang Fan, Fei Tian, Yingce Xia, Tao Qin, Xiang-Yang Li, and Tie-Yan Liu. 2020. Searching better architectures for neural machine translation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 28 (2020), 1574–1585.

[25] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*. PMLR, 1126–1135.

[26] Chelsea Finn, Kelvin Xu, and Sergey Levine. 2018. Probabilistic model-agnostic meta-learning. *Advances in neural information processing systems* 31 (2018).

[27] Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences* 3, 4 (1999), 128–135.

[28] Wei Gao. 2018. Integrated intelligent method for displacement prediction in underground engineering. *Neural Processing Letters* 47 (2018), 1055–1075.

[29] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. 2019. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 7036–7045.

[30] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 464–476.

[31] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

[32] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. *Advances in neural information processing systems* 29 (2016).

[33] Jianjun Gu, Liuchuang Wu, Jie Chen, Renli Cai, Hao Wan, Weimeng Shi, and Xiaoxiang Lv. 2022. Intelligent monitoring of subsidence cracks in underground power utility tunnel. In *Seventh Asia Pacific Conference on Optics Manufacture and 2021 International Forum of Young Scientists on Advanced Optical Manufacturing (APCOM and YSAOM 2021)*, Vol. 12166. SPIE, 848–852.

[34] Hirotaka Hachiya, Takayuki Akiyama, Masashi Sugiayma, and Jan Peters. 2009. Adaptive importance sampling for value function approximation in off-policy reinforcement learning. *Neural Networks* 22, 10 (2009), 1399–1410.

[35] Chaoyang He, Haishan Ye, Li Shen, and Tong Zhang. 2020. Milenas: Efficient neural architecture search via mixed-level reformulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 11993–12002.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[37] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems* 212 (2021), 106622.

[38] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*. 784–800.

[39] Jing Jiang and ChengXiang Zhai. 2007. Instance weighting for domain adaptation in NLP. ACL.

[40] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[41] Jaehong Kim, Sangyeul Lee, Sungwan Kim, Moonsu Cha, Jung Kwon Lee, Youngduck Choi, Yongseok Choi, Dong-Yeon Cho, and Jiwon Kim. 2018. Auto-meta: Automated gradient based meta learner search. *arXiv preprint arXiv:1806.06927* (2018).

[42] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[43] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, et al. 2021. Wilds: A benchmark of in-the-wild distribution shifts. In *International Conference on Machine Learning*. PMLR, 5637–5664.

[44] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[46] David Krueger, Ethan Caballero, Joern-Henrik Jacobsen, Amy Zhang, Jonathan Binas, Dinghuai Zhang, Remi Le Priol, and Aaron Courville. 2021. Out-of-distribution generalization via risk extrapolation (rex). In *International Conference on Machine Learning*. PMLR, 5815–5826.

[47] Dharshan Kumaran, Demis Hassabis, and James L McClelland. 2016. What learning systems do intelligent agents need? Complementary learning systems theory updated. *Trends in cognitive sciences* 20, 7 (2016), 512–534.

[48] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. 2019. The Omniglot challenge: a 3-year progress report. *Current Opinion in Behavioral Sciences* 29 (2019), 97–104.

[49] Seulki Lee and Shahriar Nirjon. 2020. Learning in the wild: When, how, and what to learn for on-device dataset adaptation. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. 34–40.

[50] Guohao Li, Guocheng Qian, Itzel C Delgadillo, Matthias Muller, Ali Thabet, and Bernard Ghanem. 2020. Sgas: Sequential greedy architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1620–1630.

[51] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. *IEEE network* 32, 1 (2018), 96–101.

[52] Ning Li, Hoang Nguyen, Jamal Rostami, Wengang Zhang, Xuan-Nam Bui, and Biswajeet Pradhan. 2022. Predicting rock displacement in underground mines using improved machine learning-based models. *Measurement* 188 (2022), 110552.

[53] Hanwen Liang, Shifeng Zhang, Jiacheng Sun, Xingqiu He, Weiran Huang, Kechen Zhuang, and Zhenguo Li. 2019. Darts+: Improved differentiable architecture search with early stopping. *arXiv preprint arXiv:1909.06035* (2019).

[54] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 740–755.

[55] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[56] Chia-Hsiang Liu, Yu-Shin Han, Yuan-Yao Sung, Yi Lee, Hung-Yueh Chiang, and Kai-Chiang Wu. 2021. FOX-NAS: Fast, On-Device and Explainable Neural Architecture Search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*. 789–797.

[57] Dongkai Liu, Jiaxing Li, Honglong Chen, Baodi Liu, Xiaoping Lu, and Weifeng Liu. 2022. EMAS: Efficient Meta Architecture Search for Few-Shot Learning. In *2022 IEEE 34th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 638–643.

[58] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[59] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[60] Jinlu Liu, Liang Song, and Yongqiang Qin. 2020. Prototype rectification for few-shot learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*. Springer, 741–756.

[61] Shiqing Liu, Haoyu Zhang, and Yaochu Jin. 2022. A survey on computationally efficient neural architecture search. *Journal of Automation and Intelligence* 1, 1 (2022), 100002.

[62] Bo Lyu, Hang Yuan, Longfei Lu, and Yunye Zhang. 2021. Resource-constrained neural architecture search on edge devices. *IEEE Transactions on Network Science and Engineering* 9, 1 (2021), 134–142.

[63] Navid Malekghaini, Elham Akbari, Mohammad A Salahuddin, Noura Limam, Raouf Boutaba, Bertrand Mathieu, Stephanie Moteau, and Stephane Tuffin. 2023. AutoML4ETC: Automated Neural Architecture Search for Real-World Encrypted Traffic Classification. *arXiv preprint arXiv:2308.02182* (2023).

[64] Jose G Moreno-Torres, Troy Raeder, Rocío Alaiz-Rodríguez, Nitesh V Chawla, and Francisco Herrera. 2012. A unifying view on dataset shift in classification. *Pattern recognition* 45, 1 (2012), 521–530.

[65] Alex Nichol, Joshua Achiam, and John Schulman. 2018. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999* (2018).

[66] Alex Nichol and John Schulman. 2018. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999* 2, 3 (2018), 4.

[67] Archit Parnami and Minwoo Lee. 2022. Learning from few examples: A summary of approaches to few-shot learning. *arXiv preprint arXiv:2203.04291* (2022).

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[69] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*. PMLR, 4095–4104.

[70] Raspberry Pi. 2015. Raspberry pi 3 model b. *online].(https://www. raspberrypi. org* (2015).

[71] Myeongjang Pyeon, Jihwan Moon, Taeyoung Hahn, and Gunhee Kim. 2021. Sedona: Search for decoupled neural networks toward greedy block-wise learning. In *International Conference on Learning Representations*.

[72] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

[73] Zhongnan Qu, Zimu Zhou, Yongxin Tong, and Lothar Thiele. 2022. p-meta: Towards on-device deep model adaptation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1441–1451.

[74] Joaquin Quinonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. 2008. *Dataset shift in machine learning*. Mit Press.

[75] Aravind Rajeswaran, Chelsea Finn, Sham M Kakade, and Sergey Levine. 2019. Meta-learning with implicit gradients. *Advances in neural information processing systems* 32 (2019).

[76] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[77] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34.

[78] Raul Rojas and Raúl Rojas. 1996. The backpropagation algorithm. *Neural networks: a systematic introduction* (1996), 149–182.

[79] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[80] Andrei A Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. 2018. Meta-learning with latent embedding optimization. *arXiv preprint arXiv:1807.05960* (2018).

[81] Richard Shin, Charles Packer, and Dawn Song. 2018. Differentiable neural network architecture search. (2018).

[82] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[83] Yisheng Song, Ting Wang, Subrota K Mondal, and Jyoti Prakash Sahoo. 2022. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *arXiv preprint arXiv:2205.06743* (2022).

[84] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. 2019. Single-path nas: Device-aware efficient convnet design. *arXiv preprint arXiv:1905.04159* (2019).

[85] Charles M Stein, Dinei A Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz G Fernandes. 2021. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurrency and Computation: Practice and Experience* 33, 11 (2021), e5786.

[86] James Victor Uspensky et al. 1937. Introduction to mathematical probability. (1937).

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[88] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. 2016. Matching networks for one shot learning. *Advances in neural information processing systems* 29 (2016).

[89] Hanhong Wang, Lin Qi, Yu Han, and Yun Lin. 2022. Prototypical Network for Few-Shot Signal Recognition. In *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 980–985.

[90] Wenna Wang, Xiuwei Zhang, Hengfei Cui, Hanlin Yin, and Yannnig Zhang. 2023. FP-DARTS: Fast parallel differentiable neural architecture search for image classification. *Pattern Recognition* 136 (2023), 109193.

[91] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[92] Yu Weng, Zehua Chen, and Tianbao Zhou. 2021. Improved differentiable neural architecture search for single image super-resolution. *Peer-to-Peer Networking and Applications* 14 (2021), 1806–1815.

[93] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[94] Yawen Wu, Zhepeng Wang, Yiyu Shi, and Jingtong Hu. 2020. Enabling on-device cnn training by self-supervised instance filtering and error map pruning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3445–3457.

[95] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. 2019. Pc-darts: Partial channel connections for memory-efficient architecture search. *arXiv preprint arXiv:1907.05737* (2019).

[96] Makoto Yamada, Leonid Sigal, and Michalis Raptis. 2013. Covariate shift adaptation for discriminative 3D pose estimation. *IEEE transactions on pattern analysis and machine intelligence* 36, 2 (2013), 235–247.

[97] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. Deepiot: Compressing deep neural network structures for sensing systems with

a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–14.

[98] Hongyuan Yu, Houwen Peng, Yan Huang, Jianlong Fu, Hao Du, Liang Wang, and Haibin Ling. 2022. Cyclic differentiable architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 1 (2022), 211–228.

[99] Haokui Zhang, Ying Li, Hao Chen, and Chunhua Shen. 2020. Memory-efficient hierarchical neural architecture search for image denoising. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 3657–3666.

[100] Miao Zhang, Steven W Su, Shirui Pan, Xiaojun Chang, Ehsan M Abbasnejad, and Reza Haffari. 2021. idarts: Differentiable architecture search with stochastic implicit gradients. In *International Conference on Machine Learning*. PMLR, 12557–12566.

[101] Pengyu Zhao, Kecheng Xiao, Yuanxing Zhang, Kaigui Bian, and Wei Yan. 2020. Amer: Automatic behavior modeling and interaction exploration in recommender system. *arXiv preprint arXiv:2006.05933* (2020).

[102] Yuekai Zhao, Li Dong, Yelong Shen, Zhihua Zhang, Furu Wei, and Weizhu Chen. 2021. Memory-efficient differentiable transformer architecture search. *arXiv preprint arXiv:2105.14669* (2021).

[103] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.