MDLdroidLite: a Release-and-Inhibit Control Approach to Resource-Efficient Deep Neural Networks on Mobile Devices

Yu Zhang, *Student Member, IEEE,* Tao Gu, *Senior Member, IEEE, Member, ACM,* and Xi Zhang

Abstract—Mobile Deep Learning (MDL) has emerged as a privacy-preserving learning paradigm for mobile devices. This paradigm offers unique features such as privacy preservation, continual learning and low-latency inference to the building of personal mobile sensing applications. However, squeezing Deep Learning to mobile devices is extremely challenging due to resource constraint. Traditional Deep Neural Networks (DNNs) are usually over-parametered, hence incurring huge resource overhead for on-device learning. In this paper, we present a novel on-device deep learning framework named MDLdroidLite that transforms traditional DNNs into resource-efficient model structures for on-device learning. To minimize resource overhead, we propose a novel Release-and-Inhibit Control (RIC) approach based on Model Predictive Control theory to efficiently grow DNNs from tiny to backbone. We also design a *gate-based* fast adaptation mechanism for channel-level knowledge transformation to quickly adapt new-born neurons with existing neurons, enabling safe parameter adaptation and fast convergence for on-device training. Our evaluations show that MDLdroidLite boosts on-device training on various PMS datasets with 28× to 50× less model parameters, 4× to 10× less floating number operations than the state-of-the-art model structures while keeping the same accuracy level.

Index Terms—Mobile Deep Learning, Deep Neural Networks, Dynamic Optimization Control, Resource Constraint.

1 Introduction

With the rapid development of wearable and mobile devices such as wristbands, EEG headsets, smartwatches and smartphones, recent years have witnessed rapid increase in the demand of Personal Mobile Sensing (PMS) applications, ranging from activity recognition [1], continual personal health monitoring [2], to private mental contexts understanding [3]. Through these devices, PMS applications are able to exploit rich contexts from personal sensing data that may be privacy sensitive. Machine learning (ML) plays a vital role in interpreting and making sense to sensor data. However, most of traditional machine learning techniques require manual and heavy feature engineering. Deep Learning (DL) offers automated feature extraction capability, the ability of scaling with data, and superior model generalization, hence has created tremendous opportunities to achieve breakthroughs in a higher level of accuracy and robustness [4].

A DL application works in two phases–training and inference. Existing Deep Neural Networks (DNNs) require heavy computation resources specially for training, beyond the capability of wearable and mobile devices. As a result, most of the solutions *offload* training workloads by transmitting sensor data from devices to clouds [5] or edge servers [6] and download pre-trained models [4] on devices for inference. However, real-world PMS applications have many intrinsic properties which create several open questions [7], [8]. Firstly, sensor data in PMS applications are highly *privacy-sensitive* as they contain motion and biological contexts of an individual. Transferring personal sensor data from devices to clouds or edge severs may raise severe privacy concerns. Secondly, due to the dynamic nature of sensor data, *i.e.*, unreliable and brittle over time, PMS applications are highly *user-specific* (*i.e.*, personal preferences or health conditions) and can be easily affected by *local scenario changes* (*i.e.*, long-term behavior changes, stationary-to-movement changes or ambient environment condition changes) [4], [7], hence *continual* training or adaptation is crucial to maintain model generalization and robustness. Thirdly, PMS applications such as gesture recognition [9] and fall detection [1] require real-time responses, hence *low-latency* in model inference is critical. Furthermore, since sensor data are naturally less interpretable than images or texts [4], collecting and labeling a large amount of sensor data with diverse real-world scenarios may be impractical. In fact, there is a lack of public available datasets and most of sensor data collections are done privately for specific PMS applications.

Existing approaches deploy pre-trained models (*i.e.*, training done in clouds) for on-device inference to avoid privacy violation [10], but these models may suffer from performance issues when applied to different users in different environments due to the problem of "one size fits all". Although transfer learning can be applied to user-specific model adaptation, several real-world limitations exist [11]: 1) the transferred model performance may be inferior due to *domain-shift* caused by target sensor data dynamics; 2) the transfer process is limited to specific source models, which may not be universally applied to different applications; 3) the model adaptation is completed by transmitting user-specific sensor data to clouds, which will violate privacy.

Mobile Deep Learning (MDL) has been recently advocated as an appealing on-device learning solution for privacy-preserving PMS applications [8], [12]. MDL

- *Y. Zhang and X. Zhang are with the School of Computer Science and IT, RMIT University, Melbourne, VIC 3000, Australia (E-mail: zac.lhjzyzzoo@gmail.com; zaibuer@gmail.com).*
- *T. Gu is with the Department of Computing, Macquarie University, NSW 2109, Australia (E-mail: tao.gu@mq.edu.au).*

promises to offer unique features to enable strict *privacy preservation* (*i.e.*, *zero* data transmission), *continual training* and *low-latency inference* on mobile devices. Thanks to data augmentation techniques [13] which can easily augment the collected user-specific sensor data to an adequate level for real-world applications, hence as a universal solution, a MDL framework with on-device training from scratch is essential for PMS applications.

Most of the existing works study model inference on mobile devices [10]. Few studies [14], [15] relate to on-device training which is more challenging because the resources required for training can easily go beyond the capacity of commodity mobile devices. Google's Federated Learning (FL) [12] as a well-known MDL framework aims to enable on-device training, but it is still at an early stage and the performance is much limited by the *resource constraint* on mobile devices. The resource overhead of on-device training therefore seems to be the main obstacle, and the performance of on-device training may be largely limited by the resources on mobile devices. Study [16] reveals that, as one of the underlying impediments, DNNs are originally designed as complex structures involving millions of parameters surprisingly, resulting in huge memory footprints, a large number of floating number operations (FLOPs), and the risk of overfitting. In essence, existing studies focus on training accuracy as the first priority yet less resource consideration for training. The accuracy-first approach will potentially result in *resource-inefficient* structures for mobile devices. Besides, most of DNN structures rely on hand-crafted model configuration with manual hyperparameter tuning on specific datasets, hence this process can be very costly and less dynamic when adapted to new datasets [17].

To reduce heavy model parameters in DNNs, model *pruning* has been proposed to achieve a lightweight structure with less resources used [18]. Study [17] indicates that a latest pruning technique can reduce 90% of the model parameters and FLOPs with little accuracy drop. Although pruning techniques have been successfully applied in mobile scenarios, they mainly focus on pruning a pre-trained, over-parameterized DNN (*i.e.*, a full-sized model configuration with redundancy) to a backbone structure (*i.e.*, much less resource overhead) for inference only, but not training [18]. Besides, since a pruned structure may be potentially over- "fitted" and primarily *fixed* on stationary datasets, the model is structurally limited to on-device *continual training*, which may cause serious learning forgetting issue to degrade model performance [16], [17], [19]. Moreover, due to the lack of hardware or libraries for sparsification support on off-the-shelf mobile devices, existing pruning-pipelines may not lead to actual compression or resource reduction [18]. The conventional training of DNNs initializes with millions of parameters in the first place may easily overwhelm the limited memory on mobile devices. We ask a fundamental question why we train DNNs with large redundant parameters from the beginning. This leads to our intuition of training DNNs from tiny to backbone, *i.e.*, small to big, eliminating the pruning process. This new approach, *i.e.*, a "growth" approach, may avoid heavy redundancy in computation resources, hence potentially fits in mobile devices.

Moving along this direction, Continuous Growth (CG) has recently been proposed in several works [16], [20], [21] that can continually search an efficient DNN structure with less redundancy and adaptable to different datasets and model configurations. CG combines both constructive (*e.g.*, adding neurons, channels or layers) and destructive (*i.e.*, pruning) structure learning. It starts training from a small-sized model configuration, and grows continually to reach the full size or the size bounded by a fixed resource budget, then pruning its model size down for inference. Although CG has not been shown its feasibility of training DNNs on mobile devices, the idea of growing DNNs from a small size can be promisingly used to continually build *resource-efficient* DNN structures for on-device training and inference. However, two critical challenges exist when applying CG on mobile devices. Firstly, the growth strategy in CG is simple and inefficient (*e.g.*, linear or near-exponential), hence it may still lead to a relatively large or over-parameterized model. In addition, since CG grows all layers of a model with the same growth rate, the model structure may contain large redundancy between layers. Although pruning may bring down the size, this process is inefficient in practice and currently unsupported on commodity mobile devices. Furthermore, CG controls the growth by pre-setting a fixed resource threshold, *i.e.*, resource budget, however it cannot handle dynamic resource changes on mobile devices in reality.

Secondly, CG adopts knowledge transfer (KT) to fast adapt new-borns (*i.e.*, new added neurons, channels or layers) by transferring existing learned parameters, which effectively saves resources [22]. However, when applied to resource-constrained devices, CG does not guarantee training convergence during growth. The convergence rate (*i.e.*, training loss rate) is a nontrivial metric to indicate the speed of training and a strong indicator to resource usage on mobile devices. Hence, such *slow convergence* in CG (Section §2) severely degrades the training performance on commodity smartphones and leads to inevitable resource overhead.

Our Approach To address the limitations of CG and move towards MDL, in this paper we present MDLdroidLite[1], a novel on-device DL framework to support privacy-preserving PMS applications. MDLdroidLite is able to fully operate DL on commodity smartphones for both training and inference. This capability is essentially achieved by our proposed dynamic fast-grow control to transform traditional DNNs into resource-efficient model structures running on mobile devices with negligible extra cost. In addition, given different datasets and DNN configurations, MDLdroidLite can dynamically control the growth of DNNs and train a dataset simultaneously in a resource-efficient way (*i.e.*, less model parameters, memory footprints, FLOPs and fast new-borns adaptation).

Our challenges are two-fold. To optimize DNNs for mobile devices, we propose a novel Release-and-Inhibit Control (RIC) approach that efficiently manages the growth of DNN model structure in layer-level (*i.e.*, each layer can grow independently). Different from CG [16], [20] that grows DNNs inefficiently to overparameter and prunes later to the backbone, our key idea is to manage the growth wisely

1. https://github.com/CPS-MDL/MDLdroidLite

from tiny to backbone so that we can avoid large redundant resource overhead. We specifically design a resource-constrained controller, named *RIC-grow*, based on the Model Predictive Control (MPC) theory, to manage the growth of DNNs in a single trajectory. In addition, we propose a layer-level, *compete-decay* growth model (§3.1.3) to predict the optimal grow-value for each grow-step, which efficiently assists the controller to make decisions (*e.g.*, whether growing or not and how many neurons). Conceptually, our approach works similar to human brain's hypothalamus that produces **Releas**ing and **Inhibit**ing hormones to help human body grow healthily [23]. Built upon RIC-enabled DNNs, MDLdroidLite can facilitate efficient training and inference on commodity smartphones with significantly-reduced resource overhead.

As aforementioned, slow convergence in CG is caused by the large variance of neurons after each growth. MDLdroidLite aims to minimize the variance for fast convergence by adapting new-born neurons quickly with existing neurons. To achieve, we design a *gate-based* fast adaptation mechanism for channel-level knowledge transformation in each layer, namely *RIC-adaption* pipeline. Different from CG, RIC-adaption pipeline uses a variance optimization function for efficient adaptation. *Safe* parameter adaptation is achieved by two proposed techniques–*three-step* distance-based selective parameter adaptation (DSPA) (§3.2.1) and gate-based coordination unit (GCU) (§3.2.2). Systematically, we first employ a cosine similarity-based parameter selection function to select a group of existing neurons that has a small variance. Next, we apply a model weight scaling function to scale down the selected parameters to new-born neurons for preserving the current loss. We then use a layer-to-layer mapping function to map new-born neurons of the subsequent layer in the same way to maintain the prior-subsequent layer shapes. Finally, we propose a *momentum-based* optimization function to minimize and coordinate the variance between the new-born and existing neurons using GCU. In a nutshell, RIC-adaption pipeline allows a notable fast convergence rate for each grow-step, hence speeding up on-device training.

We fully implement MDLdroidLite using two DL libraries, and conduct comprehensive evaluations on three off-the-shelf Android smartphones using 4 PMS datasets and 2 standard image datasets. MDLdroidLite outperforms existing parameter adaptation methods by speeding up training convergence 2.84× to 4.88×. The backbone models in MDLdroidLite achieve parameter reduction by 28× to 50×, FLOPs reduction by 4× to 10× over a full-sized model on PMS datasets while keeping the same accuracy level. In MDLdroidLite+, we notably improve the system performance in terms of growth convergence stability and resource-accuracy efficiency.

Our main **contributions** are summarized as follows.

- To the best of our knowledge, MDLdroidLite presents the first on-device structure learning framework that enables resource-efficient DNNs on off-the-shelf mobile devices, capable of building the privacy-preserving PMS applications.
- We propose a novel Release-and-Inhibit Control (RIC) approach, particularly a *compete-decay* model-based resource-constrained controller to manage the efficient growth of DNNs.
- We design a *gate-based* fast adaptation mechanism, *i.e.*, RIC-adaption pipeline, to efficiently adapt new-born neurons with existing neurons for fast convergence.
- We evaluate MDLdroidLite on Android smartphones with a number of DNNs using real-world PMS datasets. Results indicate that MDLdroidLite makes DNN model structure resource-efficient for on-device training and inference, outperforming the state-of-the-arts.

Implication MDLdroidLite moves an important step towards the promising MDL paradigm, and bridges the gap between DL and PMS applications for mobile devices. With on-device learning, MDLdroidLite will facilitate the building of a wide range of privacy-preserving PMS applications. In addition, with continual learning, MDLdroidLite will generate personalized models directly on smartphones, improving interactivity efficiency with individuals. While this paper primarily focuses on enabling DL on mobile devices for PMS applications, to further extent, MDLdroidLite can be applied to other embedded and Internet of Things (IoT) devices for intelligent edge systems and IoT applications.

2 Motivation

To discover the limitations of existing DL solutions for PMS applications, we conduct four preliminary experiments for on-device training and inference to motivate our proposal. The results are summarized as follows.

- Training with full-sized DNNs is very costly on resource-constrained smartphones, and may not be practical if no efficient solutions introduced.
- Applying pre-trained models to PMS applications may have severe poor performance. The underlying impediment is that the data from different users are naturally heterogeneous in reality, *e.g.*, non-independent and identically distributed (non-IID) personal data problem [24], and the generalization of the pre-trained models may be highly affected by this problem.
- Due to local scenario change (*e.g.*, stationary-to-movement change) in reality, new or unseen data may have severe impact on the performance of existing models (*e.g.*, pre-trained models). To tackle the poor model performance, continual on-device training may potentially work in PMS applications.
- The existing CG solutions (*i.e.*, NeST [16] and CGaP [20]) suffer from a notable slow convergence issue and incur significant resource overhead, which may not be practical for resource-constrained training on mobile devices.

Please refer to our conference paper [25] for the details of these experiments.

3 MDLdroidLite Framework

In this section, we present the system architecture of MDLdroidLite shown in Fig. 1 and its workflow shown in Fig. 2. We also describe the proposed RIC approach in detail including RIC-grow and RIC-adaption pipeline.

3.1 Release-and-Inhibit Control Approach

Conventionally, a DNN structure with specific dataset is optimized manually by domain experts on the basis of trial-and-error. Towards automatic DNN structure optimization,

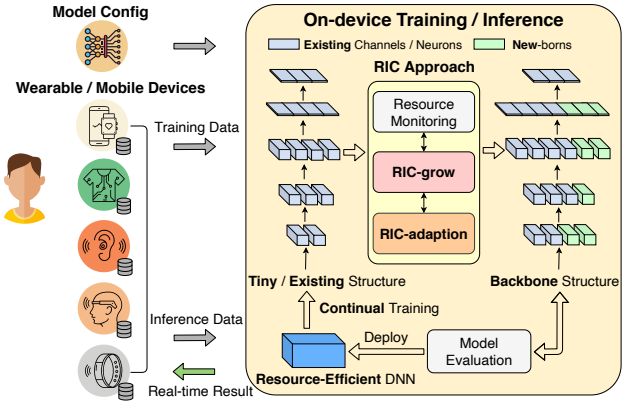


Fig. 1: MDLdroidLite Architecture.

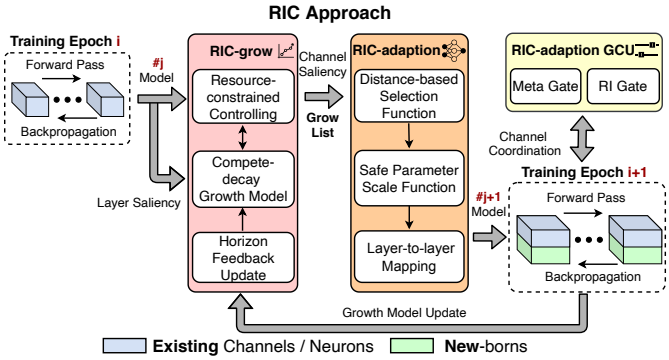


Fig. 2: MDLdroidLite Workflow.

Neural Architecture Search (NAS) has been recently proposed to utilize searching (*i.e.*, evolutionary algorithm based grid search [26]) and controlling (*i.e.*, Reinforcement Learning (RL) based structure control [27]) approaches to achieve efficient DNN structures. However, these approaches either suffer from intractable searching space or heavy extra training of control models, leading to tremendous computational cost [28], which are impractical for mobile devices.

MPC as a model-based control technique has been proposed to intuitively optimize the dynamic system states (*i.e.*, system's future actions) with a set of control constraints leveraging a finite-horizon formulation [29]. Due to the nature of less computation required for model tuning in MPC, yielding notable control performance, it has been widely applied in autonomous vehicle and mobile robotic domains to solve real-world control problems. Inspired from constructive structure learning, our basic idea is to introduce MPC-based dynamic growth control to transform traditional DNNs into resource-efficient structures for on-device learning.

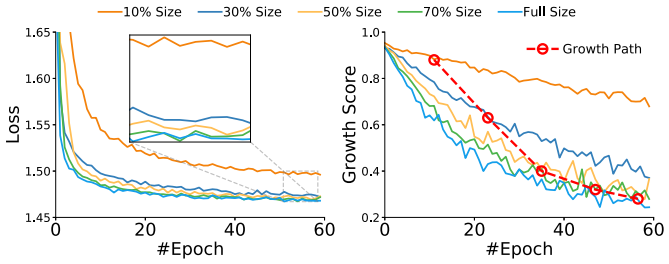
Towards training $\mathcal{T}(\cdot)$ a typical feed-forward DNN, the underlying optimization problem is to minimize the batch loss $\mathcal{L}$ measuring between the outputs transformed from input data $\mathbf{x}$ and the given labels y (*i.e.*, $\mathbf{x}, y \in \mathcal{D}$), as shown below.

$$\min_{\mathbf{W},b} \mathcal{T}(\mathcal{D},\phi) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\mathbf{W}^{\intercal}\mathbf{x}+b,y), \quad s.t.\ \mathbf{W},b \geq 0 \qquad (1)$$

where $\mathcal{D}$ and ϕ denote a given *dataset* and its *full-sized* model configuration, respectively, and m represents the batch size. Besides, both $\mathbf{W}$ as *weights* and b as *bias* are the model parameters. For each hidden layer $l = \{l_i, i = 1, 2, ..., L+1\}$

TABLE 1: Model configuration specifications

DNNs	Configuration (Type/Stride/Padding/BN)	Tiny Structure
LeNet [30]	**Conv1**$/S1 \in$ **[1, 20]** $\rightarrow Pool/S2 \rightarrow$ **Conv2**$/S1 \in$ **[1, 50]** $\rightarrow Pool/S2 \rightarrow$ **FC** $\in$ **[1, 500]** $\rightarrow Output$	[2-5-10-Output]
MobileNet [31]	**Conv1**$/S2 \in$ **[1, 32]** $\rightarrow$ **ConvDW1**$/S1 \in$ **[1, 32]** $\rightarrow$ **ConvP1**$/S1 \in$ **[1, 64]** $\rightarrow$ **ConvDW2**$/S2 \in$ **[1, 64]** $\rightarrow$ **ConvP2**$/S1 \in$ **[1, 128]** $\rightarrow$ **ConvDW3**$/S1 \in$ **[1, 128]** $\rightarrow$ **ConvP3**$/S1 \in$ **[1, 256]** $\rightarrow AvgPool \rightarrow Output$	[3-6-12-25-Output]
VGG-11 [32]	**Conv1**$/S1/P1/BN \in$ **[1, 64]** $\rightarrow Pool/S2 \rightarrow$ **Conv2**$/S1/P1/BN \in$ **[1, 128]** $\rightarrow Pool/S2 \rightarrow$ **Conv3**$/S1/P1/BN \in$ **[1, 256]** $\rightarrow$ **Conv4**$/S1/P1/BN \in$ **[1, 256]** $\rightarrow Pool/S2 \rightarrow$ **Conv5**$/S1/P1/BN \in$ **[1, 512]** $\rightarrow$ **Conv6**$/S1/P1/BN \in$ **[1, 512]** $\rightarrow Pool/S2 \rightarrow$ **Conv7**$/S1/P1/BN \in$ **[1, 512]** $\rightarrow$ **Conv8**$/S1/P1/BN \in$ **[1, 512]** $\rightarrow Pool/S2 \rightarrow Output$	[6-12-25-25-50-50-50-50-Output]



(a) (b)

Fig. 3: (a) Correlation between loss reduction and structure growth; (b) Non-linear correlation between structure growth and growth score in a layer.

in the DNN, the input channel size as *fan_in* and output channel size as *fan_out* of each layer l is defined as I^l and O^l in ϕ, respectively. Hence, the structure **shape** of l denotes as $I^l \times O^l \times KH^l \times KW^l$ in a convolution layer, and $I^l \times O^l$ in a fully-connected layer, where KH and KW denote kernel height and kernel width, respectively. When relating to resource usage, a large shape of l represents a large number of model parameters, memory footprints and FLOPs. In short, a DNN model structure is simply defined as an array of O^l, *e.g.*, array [20-50-500-10] represents a full-sized LeNet as shown in Table 1.

To observe the correlation between loss minimization and structure growth, we train a LeNet on MNIST with reduced scales (*e.g.*, 10%, 30%, 50%, 70%) of the full size shown in Fig. 3a. The result in the zoom-in figure indicates that the loss is monotonically decreased when the structure size increases. Our observation is that the reduction rate of loss gradually reduces as the structure size increases, even the loss of $50\%\,Size$ is nearly equal to that of $Full\,Size$, resulting in the same level of accuracy. Intuitively, large resources can be saved if the growth of model structure is properly controlled to a "fit" size without accuracy drop. To this end, we propose a novel RIC approach to optimize the DNN structure under resource constraints and speed up on-device training in layer-level. Specifically, RIC includes two components–RIC-grow which controls structure growth by *competing* the growth values of *release* and *inhibit* decisions (*i.e.*, structure *growing* or structure *staying* respectively) in each layer, and RIC-adaption pipeline which enables fast training convergence after growth.

To formulate our problem, we define that $A^l = 0, 1, ..., O^l$ is a set of control actions representing the *growth number* in O^l (*i.e.*, the *fan_out* of each layer in full-sized ϕ).

The current *fan_out* of each layer is denoted as $o \in O^l$, and each action is denoted as $a \in A^l$, where a is limited up to o (*i.e.*, maximum growth of double times o) for effective parameter transfer adaptation. Especially, $\mathcal{I}$ is denoted as *inhibit* decision if $a = 0$, and $\mathcal{R}$ is denoted as *release* decision if $a > 0$. Also, we let θ denote a structure array of O^l, and π denote a control action array of A^l. The *growth step* is defined as t in a set of $T = 1, 2, ..., N$ representing each training epoch, where N is a given maximum epoch. Practically, we optimize the structure θ at each grow-step t. Thus, our growth control optimization is formulated as follows.

$$\begin{aligned} &\arg\min_{\pi_t} \min_{\mathbf{W},b} \mathcal{T}(\mathcal{D}, \theta^\circ), \quad s.t.\ \theta \in \phi, \pi_t \in \mathcal{RIC}(t) \\ &\theta^* = \theta^\circ + \sum_{t=1}^{N} \pi_t, \quad \pi_t = [a_t^{l_1}, a_t^{l_2}, ..., a_t^{l_L}], a_t^l \in [0, o_t^l] \end{aligned} \tag{2}$$

where $\mathcal{RIC}(\cdot)$ denotes a dynamic control *constraint model*. In formal terms, given a tiny structure θ° and a specific dataset $\mathcal{D}$ on device, our main objective in Eq. 2 is to transform the structure to a resource-efficient backbone structure θ^* controlled by a set of growth control decisions π_t (*i.e.*, subject to $\mathcal{RIC}(t)$) through grow-step N along with the training objective $\mathcal{T}(\cdot)$ simultaneously. Since MDLdroidLite is an on-device structure learning framework, we focus mainly on controlling the growth of layers' *fan_out* O^l in a resource-efficient way.

3.1.1 Resource-Constrained RIC-grow

In MDLdroidLite, the control *constraint model* $\mathcal{RIC}$ in Eq. 2 is defined as a Markov tuple (O, T, A, P, R), where the *fan_out* of each layer O^l and the grow-step T are discrete state variables, P^l is a stochastic state transition model, and R is a resource-constrained reward function shown in Eq. 5. The control model state $s \in S^l$ is denoted as a pair of (t, o), s° denotes the initial state at each grow-step, and $s' = (t', o')$ denotes a *future* state transiting from the state s by control action a, where $o' = o + a$ and $t' = t + 1$. Given K as the size of finite-horizon *time window*, RIC-grow optimizes the control actions within the horizon area $t \to t + K$ at each grow-step. As a result, the layer-level control decision value function $\mathbf{V}^l(s)$ is formulated based on Bellman equation [33] as:

$$\mathbf{V}^l(s) = \max_{a \in \{\mathcal{R}, \mathcal{I}\}} [Bernoulli(p^l, s, a, s')(\mathbf{R}^l(s, a, s') + \gamma \mathbf{V}^l(s'))] \tag{3}$$

where γ denotes a value discount factor (*e.g.*, 0.5 as default). To reduce the computation cost of $\mathbf{V}^l(s)$, we employ a $Bernoulli(\cdot)$ [34] as default transition model to randomly dropout some actions during recursive optimization, where p^l is a pre-set probability for all actions (*e.g.*, 0.5 as default). Practically, RIC-grow makes each optimized control decision by solving $\max\{\mathbf{V}^l(s)|\mathcal{R}, \mathbf{V}^l(s)|\mathcal{I}\}$ to achieve the "fit" size for each layer, as shown in Algorithm 1.

In training process, RIC-grow is called after each training iteration. The growth of each layer $l_i \in L$ is individually controlled by $\mathbf{V}^l(s)$ with a given time window size K and a list of size o control actions, hence the time complexity of RIC-grow is theoretically represented as $\mathcal{O}(LKo^2)$. In practice, as both time window size K and layer size L are initialized as constant, the time complexity can be simplified to $\mathcal{O}(n^2)$. In addition, since applying $Bernoulli(\cdot)$ with a 0.5 dropout rate can effectively reduce the size of control actions, the actual time complexity of RIC-grow will remain as efficient for on-device learning.

3.1.2 Growth Cost Constraints

In the control decision value function, we consider two typical resource constraints of DNNs, the number of model parameters (*i.e.*, size of neurons) and the number of FLOPs for each layer, which may actually affect on-device memory footprints, battery consumption, training time and inference latency. We associate a $Flops^l(s)$ function [35] with a $Size^l(s)$ function to dynamic calculate the growth cost of each layer representing $\mathcal{C}^l(s)$ shown as:

$$\mathcal{C}^l(s) = (1 - \beta) Flops^l(s) + \beta Size^l(s) \tag{4}$$

where β denotes a normalization coefficient between both functions. Since the convolution layer mainly contributes the number of FLOPs (*e.g.*, 82.55% in a full-sized LeNet), and the fully-connected layer has a larger parameter size (*e.g.*, 94.06% in a full-sized LeNet), we set β to 0.2 in convolution layers and 0.8 in fully-connected layers to normalize the total growth cost. We hence propose the resource-constrained reward function $\mathbf{R}^l(s, a, s')$ formulated as a growth *state-value* function $\mathbf{G}^l(s, a, s')$ constrained by a related growth *state-cost* function $\mathbf{C}^l(s, a, s')$ formulated as:

$$\begin{aligned} \mathbf{R}^l(s, a, s') = \frac{\mathbf{G}^l(s, a, s')}{\mathbf{C}^l(s, a, s')} &= \frac{\left|\mathcal{G}^l(s) - \mathcal{G}^l(s')\right|}{\frac{\mathcal{C}^l(s')}{\mathcal{C}^l(s)} + \mathcal{P}^l} \\ &= \frac{\left|\mathcal{G}^l(s) - \mathcal{G}^l(s')\right|}{1 + \eta^l e^{d(s', s^\circ)} (\frac{\mathcal{C}^l(s')}{\mathcal{C}^l(s)} - 1)} \end{aligned} \tag{5}$$

where the growth *state-value* is calculated as a growth value difference between states, in which the growth values are predicted using a *compete-decay* growth model $\mathcal{G}^l(s)$ in Eq. 8. Also, the growth *state-cost* is extended as a scale of the incremental cost between states using the growth cost function $\mathcal{C}^l(s)$ in Eq. 4 with $\mathcal{P}^l$ denoted as a penalty function detailed in Section 3.1.5.

3.1.3 Compete-decay Growth Model

Since RIC-grow optimizes structure growth based on a dynamic model-based MPC, the growth model $\mathcal{G}^l(s)$ in Eq. 5 plays a vital role to ensure the resource-constrained control performance. To build the structure growth model, we first apply saliency metric analysis (*e.g.*, L1/L2 weight normalization) [36] to approximate the importance of different structure components (*i.e.*, layers and channels), and heuristically predicting the growth values after both *release* and *inhibit* control actions. Since the saliency metric is widely used in pruning tasks with fair performance [37], it is an efficient way to track the dynamic changes of layers and channels during the training optimization in Eq. 2. Different from the analysis of correlation between loss and specific component removal in pruning, we mainly analyze the direction of weights' gradient changes for each component through the loss reduction aiming to predict potential structure growth values within a finite-horizon time window. Based on L1 normalization of component weights [36], we combine the weights' gradients processed in Back-propagation (BP) to compute dynamic saliency metric of layers and channels, formulating as *layer* saliency score $\mathcal{S}^l$ and *channel* saliency score $\mathcal{CS}_i^l$ in Eq. 6, hence the direction

vector of weights' gradients changes is represented as a set of saliency scores. To formulate the correlation between a set of saliency score and loss reduction in each layer, we utilize a *cosine* similarity function as $Cosim(\cdot)$ [38] to normalize both vectors, marking as growth score GS^l formulated as $GS^l = Cosim(\mathcal{S}^l_{1:n}, \mathcal{L}_{1:n})$.

$$\mathcal{S}^l = \sum_{i=0}^{O}(\mathcal{CS}^l_i) = \sum_{i=0}^{O}(\sum_{j=0}^{I}\sum_{kh=0}^{KH}\sum_{kw=0}^{KW}\left|\frac{\partial \mathcal{L}(\mathbf{W}^\intercal \mathbf{x}+b, y)}{\partial \mathbf{W}^l}\mathbf{W}^l\right|) \quad (6)$$

where n denotes a unit size of direction vector (*e.g.*, 5 as default).

To explore the correlation between structure growth and the proposed growth score in each layer, we next present the preliminary results using the five trained LeNets on MNIST. In Fig. 3b, the growth scores of five layer structures not only demonstrate as monotonic *non-linear decay* along with loss reduction, marking as *Decay-Stay* (DS) (*i.e.*, growth score decay with structure staying), but also show a vertically monotonic decay along with increase of the structure sizes. Especially, the decay rate of growth scores vertically slows down as layer structure size increases, which also represents as monotonic *non-linear decay* about growth score, marking as *Decay-Grow* (DG) (*i.e.*, growth score decay with structure growing). The results indicate that layer structure is able to be *individually* controlled based on its dynamic growth score to efficiently save resources. Conceptually, the red dash line presents a potential structure growth path about growth score by transiting growth state every 12-epoch. Based on the observation, the way of using both *non-linear decay* (*i.e.*, DS and DG) can empirically cover both horizontal and vertical growth state transitions in RIC-grow, hence the structure growth model can be theoretically proposed as a composition of both non-linear models, named a *compete-decay* growth model as $\mathcal{G}^l(s) = \mathcal{F}(DS^l(s), DG^l(s))$, helping RIC-grow controller make each $\mathcal{R}|\mathcal{I}$ decision by competing the predicted growth values of *Decay-Grow* and *Decay-Stay* in layer-level.

To solve both $DS^l(s)$ and $DG^l(s)$, we utilize a typical decay exponential model $D^l(x) = ab^x + c$ to represent, where $a \in (0, 100)$, $b \in (0, 1)$, $c \in (0, 10)$, and x denotes a set of sample growth scores. Practically, we apply non-linear regression by solving mean square error (MSE) [39] to fit the proposed models in Eq. 7. Since RIC-grow collects growth scores GS^l of each layer at each grow-step, the model parameters (*i.e.*, $a, b, and\, c$) of $DS^l(s)$ can be continually tuned to ensure the performance. However, since the structure is controlled as a single instance, the model parameters of $DG^l(s)$ may not be able to converge due to insufficient growth scores between state transition. For this, we propose a *Triplet Decay Array* (TDA) formulated as $TDA^l = [DS^l(s(t, o-a_2)), DS^l(s(t, o-a_1)), DS^l(s(t, o))]$ to record *three* latest DS^l with the growth states transited by *release* actions (*i.e.*, two past states and current state with different O^l), aiming to provide sufficient growth scores between state transition to fit $DG^l(s)$ with minimal resource cost. The composed formulations are shown as follows.

$$\arg\min_{a,b,c} \sum_{i=1}^{3}(DG^l(s) - TDA^l[i])^2 \quad (7)$$

$$\mathcal{G}^l(s) = [DG^l_t(s) \quad ... \quad DG^l_{t+K}(s)] \quad (8)$$

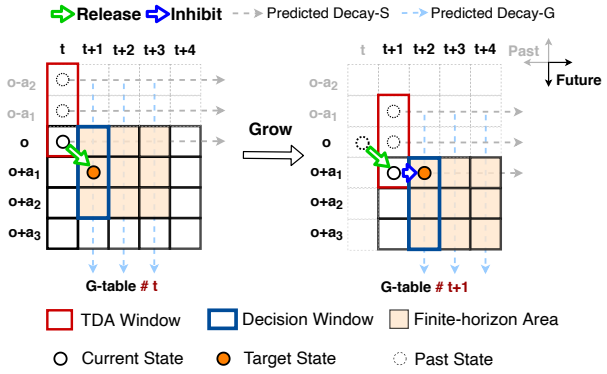


Fig. 4: RIC-grow control workflow in a layer.

where the model parameters of $DG^l(s)$ are tuned using TDA. Practically, RIC-grow requires to take two *release* actions at the beginning of training as *warm-up* to set-up TDA^l and $DG^l(s)$. As a result, the proposed *compete-decay* growth model $\mathcal{G}^l(s)$ is composed of multiple $DG^l(s)$ within the $t + K$ time finite-horizon area. For each grow-step, the outputs of $\mathcal{G}^l(s)$ can be represented as a growth value matrix named *G-table* to contribute in control decision value optimization in Eq. 3. Fig. 4 demonstrates the workflow of RIC-grow using the proposed growth model in a layer. In addition, both TDA and growth model are updated by the latest growth scores as *feedback* to ensure the growth value prediction performance, and TDA window moves to the latest growth state before next control decision.

3.1.4 RIC Convergence

Since existing CG works [16], [20], [21] have proved the training convergence (*i.e.*, monotonic loss reduction guarantee) during the DNN structure growth, our work is based on the ground to offer a layer-level resource-constrained growth control. For the idea of our RIC approach, keeping *inhibit* represents a standard learning process (*i.e.*, no structure growth) with a stable loss reduction, while performing *release* achieves a faster loss reduction due to the structure growth, but the *decay rate* of loss reduction monotonically turns to small until nearly "flat" when reaching a global minimum (*i.e.*, loss convergence). Besides, since the reduction of growth score GS^l by both actions refers to the loss reduction and performs as similar, the growth *state-value* $\mathbf{G}^l(s, a, s')$ in Eq. 5 also turns to converge along with the loss convergence. Therefore, constrained by the growth *state-cost* $\mathbf{C}^l(s, a, s')$ in the reward function, the control decision value function $\mathbf{V}^l(s)$ in Eq. 3 performs as *convex* and converges to keeping *inhibit* (*i.e.*, $a = 0$), as shown below.

$$\frac{\mathbf{G}^l(s^*, a, s')}{\mathbf{C}^l(s^*, a, s')} + \gamma \mathbf{V}^l(s') \leq \mathbf{G}^l(s^*, 0, s') + \gamma \mathbf{V}^l(s'), a \in [0, o] \cap p^l \quad (9)$$

where s^* denotes an optimized state (*i.e.*, achieving a resource-efficient backbone structure θ^*) when RIC is converged.

3.1.5 Dynamic Penalty Function

The penalty function $\mathcal{P}^l$ in Eq. 5 can efficiently work to reinforce constraints and avoid redundant growth. However, our previous penalty function is designed as a fixed penalty regulator with manual set-up, which may not handle dynamic growth scenarios well. Since growth values are

dynamically predicted by the growth model $\mathcal{G}^l(s)$ in Eq. 8, our observation is that the *error rate* of the prediction for next state s' may exponentially increase along with the increase of spatial distance between s' and s° during finite-horizon optimization, yielding sub-optimal control performance and unstable growth convergence. To achieve a fair control performance, we first design an exponential sub-equation $e^{d(s',s^\circ)}$ to efficiently reinforce the growth cost constraint, aiming to mitigate the effect of the *error rate* shown in Eq. 5, where $d(s', s^\circ)$ denotes as the distance between s' and s°, and η^l denotes as the penalty regulator to enlarge or reduce penalty effect for each layer. To stabilize growth convergence, we next extend penalty regulator η^l to a *momentum-based* dynamic function in Eq. 10, aiming to gradually *enlarge* the penalty effect for a stable growth performance.

$$\eta^l(t+1) = \eta^l(t) + \xi\, ReLu(\Delta\mathcal{G}^l(t-1) - \Delta\mathcal{G}^l(t)), \quad \eta \in (0, 1] \quad (10)$$

where ξ denotes a coefficient (*e.g.*, 0.05 by default). The increment of η^l represents the penalty effect enlargement. Since *growth acceleration* (*i.e.*, the gradient between prior and current growth value gradients) is a fair metric to indicate growth convergence performance, our intuition is that we dynamically enlarge η^l along with the changes of the growth acceleration, regulating the penalty effect to enhance convergence. In Eq. 10, we define that once the growth acceleration turns positive (*i.e.*, the growth turns to converge), η^l will be increased by the acceleration to enlarge the penalty effect until being stable. We also apply a $ReLu$ function to filter the condition where the growth acceleration presents as negative.

3.1.6 Reverse TDA Update

Since TDA^l records *three* latest DS^l to provide sufficient growth *gradients* between state transition to fit $DG^l(s)$ in Eq. 7, it is critical to update TDA^l to ensure the growth predication performance of growth model $\mathcal{G}^l(s)$. We discover some limitations in our previous *feedback* update mechanism of TDA^l. After making growth decisions at the t iteration, the states of all layers transit to the next $t+1$ iteration. We then train the model with m batch data, and collect the latest growth scores $GS^l_m(t+1)$ as *feedback* to update TDA^l, but the states of the *three* DS^l in TDA^l still stay at t. Although we can use $GS^l_m(t+1)$ to update the third record $DS^l(t)$ when taking a $\mathcal{I}$ decision or tune a new $DS^l(t+1)$ when taking a $\mathcal{R}$ decision, the prior two records may not be able to update due to different *fan_out* o^l (*i.e.*, different layer structures). Through our experiments, we observe that the state transition *gradients* may become larger over iterations due to the lack of a complete TDA^l update, leading to sub-optimal growth convergence (*i.e.*, the model may "over" grow). Alternatively, we can force to completely update all records in TDA^l by $GS^l_m(t+1)$, but the result presents that the *gradients* may be tuned smaller or even vanishing, yielding insufficient growth to achieve a degraded accuracy. To fully update TDA^l and reserve the state transition *gradients*, we propose a reverse TDA update mechanism based on a *gradient-scale* function formulated as:

$$GS^l_m(t+1, \tilde{o}) = \min\{\frac{DS^l(t+1, \tilde{o})}{DG^l(t+1, o)}, \frac{\overline{GS}^l_m(t, \tilde{o})}{\overline{GS}^l_m(t+1, o)}\}\, GS^l_m(t+1, o) \quad (11)$$

where $\tilde{o}$ denotes the layer *fan_out* in a prior record in TDA^l. Since we target to update prior records with different *fan_out*, the basic idea of the mechanism is: 1) the *gradient-scale* function is used to "transform" the latest growth scores $GS^l_m(t+1, o)$ to fit a prior record using the predicted gradient scale at the t+1 iteration (*i.e.*, $DS^l(t+1, \tilde{o})/DG^l(t+1, o)$); 2) the mechanism will reversely update the next prior record using the function based on the transformed growth scores, aiming to reserve the *gradients* between records in TDA^l. In addition, since both $DS^l(s)$ and $DG^l(s)$ are monotonically *non-linear decay*, the mean of the transformed growth scores should be less than that of the prior collected growth scores as ground truth, hence the *gradient-scale* should be not greater than the mean gradient scale between the prior and latest collected growth scores (*i.e.*, $\overline{GS}^l_m(t, \tilde{o})/\overline{GS}^l_m(t+1, o)$ as an upper bound). Thus, we define that the *gradient-scale* is the minimum between the predicted and collected scale in Eq. 11.

3.1.7 Layer-to-layer Growth Constraint

RIC-grow is designed to enable layer-level DNNs growth control, and each layer based on its growth model $\mathcal{G}^l(s)$ can individually grow in a resource-efficient way. Since runtime *low-latency* is a critical efficiency metric of on-device inference for real-world PMS applications, RIC-grow aims to optimize DNN structure with minimal FLOPs to achieve an efficient on-device inference. However, through our experiments, we observe that the optimized structure may achieve sub-optimal FLOPs performance due to the lack of layer-to-layer growth constraint. Since the prior layer growth will *indirectly* lead to the subsequent layer growth to keep the layer-to-layer input shapes, individually calculating the FLOPs of each prior layer growth as constraint may not be able to represent that of the layer-to-layer growth, yielding inferior growth performance and extra latency for on-device inference. To optimize the DNN structure with minimal FLOPs, we propose a layer-to-layer cost function on top of layer-level growth control, which is formulated as:

$$\arg\max_{\pi_t'} \frac{\sum_{i=1}^{L} \mathcal{G}^{l_i}(s)}{\frac{Flops(\theta_t + \pi_t)}{Flops(\theta_t)}},\ s.t.\ \pi_t = [a_t^{l_1}, a_t^{l_2}, ..., a_t^{l_L}], a_t^{l_i} \in \{a, 0\} \quad (12)$$

where θ_t denotes the current DNN structure, and π_t denotes an optimized set of all layer growth actions $a_t^{l_i}$. Since the layer-level growth control in Eq. 3 outputs a π_t at each iteration, we "re-apply" the $\mathcal{R}|\mathcal{I}$ decision for each action $a_t^{l_i}$ (*i.e.*, each action either holds to grow or drops to 0) to rearrange π_t as different growth action combinations. Our objective is to find the "best" combination $\pi_t{}'$ to fit the cost function in Eq. 12 with the maximum decision value. Since the sum of the all layers' growth values is re-subject to the layer-to-layer growth FLOPs, all "best" combinations through growth can eventually contribute to a resource-efficient model structure with minimal FLOPs, resulting in *low-latency* on-device inference.

3.2 RIC-adaption Pipeline

The aforementioned issue of slow training convergence may incur considerable resource overhead on mobile devices. To further understand this issue, we use the proposed channel saliency score $\mathcal{CS}^l_i$ in Eq. 6 to analyze the two existing

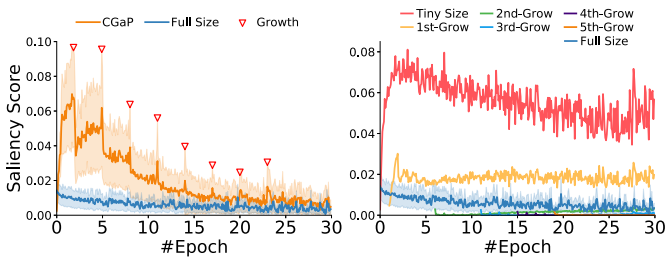


Fig. 5: Variance of neuron saliency scores in a layer after each growth: (a) Spiking and enlarged variance using CGaP; (b) Large variance in channel-level using NeST.

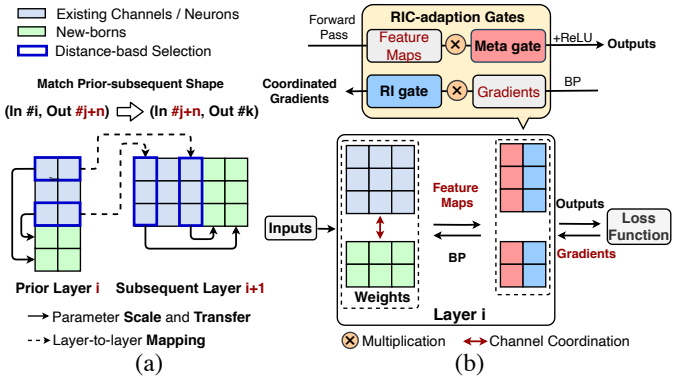


Fig. 6: RIC-adaption pipeline in channel-level: (a) Layer-to-layer parameter selection, scale and mapping; (b) Existing-to-new channel coordination in a layer.

CG methods and compare them with the full-sized model. Fig. 5a shows that the score variance using CGaP [20], representing as the shadow area on the line, is much larger than training a full-sized model around each growth mark, and the mean of the scores is way larger than that of the full-sized model especially at early-middle time stages.

One of our key observations reveals that the *variance* of channel saliency scores in each layer presents a notable effect of enlargement on both CG methods. Also, the channel saliency scores of CGaP continually demonstrate the strong *spiking* issue throughout the model growth, resulting in the degraded *spiking loss*. Such *unsafe* loss reduction leads to a serious *accuracy drop* after each growth. Similarly, Fig. 5b presents a detailed channel saliency score comparison after each growth using NeST [16]. The results indicate that the new-born channels of each growth perform much less importance, leading to large score variance between channels. Theoretically, the increased risk of large variance between channels in CG methods has a strong impact to the distribution of model parameters, and makes the model highly sensitive to input, resulting in unstable loss reduction (*i.e.*, *spiking loss*) and poor model generalization (*i.e.*, accuracy drop) to slow down the convergence rate [38], [40]. Thus, the existing KT adaptation approaches in CG largely increases the risk of large variance in channel-level, yielding unsafe parameter adaptation and slow training convergence under resource-constrained conditions.

Batch Normalization (BN) [41] has been widely applied in various DNN structures to speed up training, but it may not work well on resource-constrained mobile devices. To evaluate resource efficiency of BN on mobile devices, we train both MobileNet on HAR and VGG-11 on CIFAR-10 with different batch size conditions. Our observation is that applying BN on device causes extra resource overhead for both training and inference, and accuracy highly relies on the batch size, *e.g.*, 93.01% with batch-32 but 96.09% with batch-64 using MobileNet on HAR. Although BN works well on large-scale VGG-11, it may not be resource-efficient to small-scale MobileNet, *e.g.*, MobileNet on HAR can still achieve a fair accuracy on 95.55% without using BN.

To address the aforementioned issues and achieve resource efficiency for KT adaptation, we propose RIC-adaption pipeline with two unique techniques–*three-step* distance-based selective parameter adaptation (DSPA) and Gate-based Coordination Unit (GCU). Our basic idea is to first safely adapt *existing-to-new* (*i.e.*, transferring existing parameters to new-born parameters) in growth, and then gradually coordinate the variance of *existing-to-new* within the next training epoch.

3.2.1 *Three-step Selective Parameter Adaptation*

To safely adapt *existing-to-new* without accuracy drop in each growth, we propose three parameter adaptation methods–parameter selection, scale, and mapping between prior-subsequent layer. Firstly, we select the number of existing channels $\mathbf{W}^l_{Select}$ with a small variance and mean of saliency scores, and prepare to transfer them to new-born channels. Secondly, we propose a safe parameter scale function formulated in Eq. 13 to preserve the current loss reduction without accuracy drop. Combining with the parameter selection, the parameters of new-born channels $\mathbf{W}^l_{New}$ can be transferred by Eq. 14, where $\mathcal{N}$ denotes a small amount of noise initialized by a uniform distribution U, and μ sets as 1. Thirdly, we map selected channel indexes in prior layer to subsequent layer to select the related channels, and transfer using the same way in Eq. 14 in subsequent layer.

$$Scale^l = \frac{\sqrt{I^l}\, Std(\mathbf{W}^l_{Ex})}{\sqrt{3}} \tag{13}$$

$$\mathbf{W}^l_{New} = \frac{\mathbf{W}^l_{Select}}{Scale^l} + \mathcal{N}^l, \quad \mathcal{N} \in U[-\mu, \mu], \mathbf{W}^l_{Select} \in \mathbf{W}^l_{Ex} \tag{14}$$

where $\mathbf{W}^l_{Ex}$ denotes the parameters of existing channels. As a result, Fig. 6a illustrates the workflow of these three methods. Please refer to our conference paper [25] for the detailed design of the three-step DSPA.

3.2.2 *Gate-based Coordination Unit*

To coordinate the variance of *existing-to-new* after growth, we design GCU as a separate matrix followed up each layer shown in Fig. 6b to enable fast convergence and speed up training. In Eq. 6, the channel saliency score consists of channel weights and gradients, therefore we decompose the variance of the scores coordination into two tasks–weight variance reduction and gradient based variance coordination. To handle weight variance reduction, we apply *meta gate* as weight regulators (*i.e.*, $Meta^l_{Ex}$ and $Meta^l_{New}$) to gradually reduce the weight variance between existing and new-born channels in forward pass. Besides, we employ *RI gate* as a *gradient* regulator (*i.e.*, RI^l_{Ex} and RI^l_{New}) to perform gradient based variance coordination. We first design the coordination objective function as $\min_{RI^l} \left| \mathcal{CS}^l_{Ex} - \mathcal{CS}^l_{New} \right|$.

We next propose a *momentum* function in Eq. 15, where m is batch size, to improve the minimization as fast and stable. In practice, both RI^l_{Ex} and RI^l_{New} are dynamically optimized to minimize the objective function using Eq. 16 within the current training epoch, resulting a stable gradient descent with fast training convergence.

$$\mathcal{V}^l(m) = \lambda \mathcal{V}^l(m-1) + (1-\lambda)(\mathcal{CS}^l_i(m) - \mathcal{CS}^l_i(m-1)) \quad (15)$$

$$RI^l(m+1) = RI^l(m) + \hat{\alpha}\mathcal{V}^l(m), \quad RI^l \in [\frac{1}{2\,Scale^l}, 2\,Scale^l] \quad (16)$$

where $\hat{\alpha}$ denotes a signed unit vector, and λ denotes a momentum coefficient (*e.g.*, 0.95 by default). Please refer to our conference paper [25] for the detailed design of GCU.

In short, the RIC algorithm is illustrated in Algorithm 1.

Algorithm 1 RIC Approach Algorithm

Initialize: training dataset $\mathcal{D}$, the number of training iteration N, the size of finite-horizon time window K
procedure RIC-GROW($\mathcal{D}$, N, K)
  $\theta \leftarrow$ Build a tiny model θ° or re-load an existing model
  while Training iteration $t < N$ **do**
    for Batch iteration m **do**
      Call *RIC-GATE*($m \in \mathcal{D}$, θ, List $Scale$) to train a batch
      List $GS \leftarrow$ Collect all layers' growth scores
    for Layer $l_i \in L$ **do**
      if TDA^l *warm-up* **then**
        Reverse update TDA^l using List GS by Eq. 11
        Update the growth model $\mathcal{G}^l(s)$ by Eq. 7 and 8
        $\pi_t \leftarrow$ optimized growth action $a^{l_i}_t$ by Eq. 3
        Update the penalty regulator η^l by Eq. 10
    if $\pi_t \neq \varnothing$ **then**
      $\pi_t \leftarrow$ optimized layer-to-layer growth by Eq. 12
    else
      $\pi_t \leftarrow$ a set of small growth-unit for *warm-up*
    Call *RIC-ADAPTION*(θ, π_t) to model growth;
  Save resource-efficient model θ^* for on-device inference
procedure RIC-GATE($m \in \mathcal{D}$, θ, List $Scale$)
  for Layer $l_i \in L$ **do**
    Calculate feature maps $\mathbf{x}^l$ by Forward pass
    if l_i growth **then**
      $\mathbf{x}^l \leftarrow Meta^l\,\mathbf{x}^l$ then uniformly decay $Meta^l_{Ex}$
  Calculate the loss $\mathcal{L}$
  for Reverse layer $l_i \in L$ **do**
    Calculate gradients g^l by BP
    if l_i growth **then**
      Minimize variance using RI^l_{Ex} and RI^l_{New} by Eq. 16
procedure RIC-ADAPTION(θ, π_t)
  for Layer $l_i \in L$ **do**
    if $a^{l_i}_t \in \pi_t > 0$ **then**
      Select $a^{l_i}_t$ existing channels "close" to the lowest $\mathcal{CS}^l_i$
      Calculate $Scale^l$ by Eq. 13 and List $Scale \leftarrow Scale^l$
      Transfer to $\mathbf{W}^l_{New}$ by Eq. 14 and $\mathbf{W}^l_{Ex} \leftarrow \mathbf{W}^l_{Ex}/Scale^l$
      Map to subsequent layer and transfer by Eq. 14

4 Evaluation

In this section, we first describe the implementation of MDLdroidLite. We then design three sets of experiments to comprehensively evaluate the performance of MDLdroidLite on several commodity smartphones using a range of datasets. The first set evaluates the performance of RIC-adaption pipeline compared with existing parameter transfer adaptation baselines. The second set compares the performance of MDLdroidLite with existing CG and search methods, also evaluates the improvements of MDLdroidLite+. The third set examines the resource-accuracy efficiency of MDLdroidLite+ in real-world use scenarios.

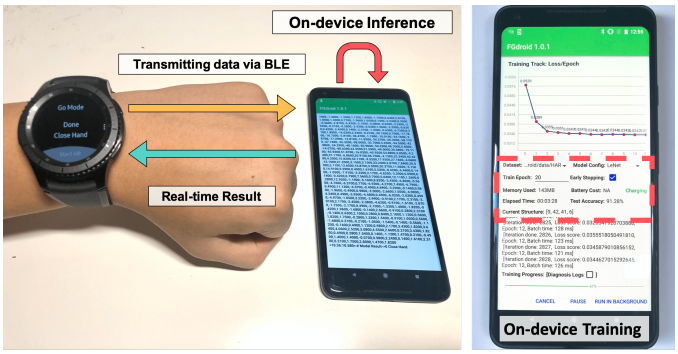


Fig. 7: MDLdroidLite screenshot: (a) Smartwatch gesture input and on-device inference; (b) On-device training.

TABLE 2: PMS & Image dataset specifications

Datasets	Type	Task	Class	Sample	Rate	#-C	#-H×W	#-KH×W	#-TR	#-TE
sEMG [43]	EMG	GR	6	14695	50Hz	8	1×100	1×12	41.3MB	10.3MB
MHEALTH [44]	IMU	HBM	9	3255	50Hz	23	1×100	1×12	41.9MB	18MB
HAR [45]	IMU	ADLs	6	10299	50Hz	9	1×128	1×14	67.8MB	27.2MB
FinDroidHR [9]	IMU&HR	GR	6	2520	100Hz	7	1×150	1×14	15.6MB	2.6MB
MNIST [30]	IMG	IR	10	70000	NA	1	28×28	5×5	15MB	2.5MB
CIFAR-10 [46]	IMG	IR	10	60000	NA	3	32×32	3×3	113MB	23MB

4.1 MDLdroidLite Implementation

We implement MDLdroidLite based on two DL libraries– DL4J version 1.0.0-SNAPSHOT and PyTorch version 1.4.0. Specifically, we modify the source code of training flow and building DNN structure for both DL libraries. We apply our implementation on three off-the-shelf smartphones purchased in the past four years shown in Table 3. To simplify the usage scenario of MDLdroidLite with different model configurations, we implement the RIC approach as a separate layer. After loading a model configuration, MDLdroidLite will easily add a RIC layer between each hidden layer (*i.e.*, convolution layer and fully-connected layer) and subsequent BN or ReLU layer to enable both RIC-grow and RIC-adaption pipeline. To demonstrate the use of MDLdroidLite in real-world PMS applications and evaluate its end-to-end performance, we develop an application to recognize hand gestures using smartwatch based on the work done in [9]. In practice, MDLdroidLite applies an early stop strategy [42] that can stop the training early if no more higher resource-accuracy trade-off within an N-epoch countdown to avoid the risk of overfitting and redundant training.

4.2 Experimental Set-up

To evaluate MDLdroidLite, we select four PMS datasets (three public and one self-collected) and two well-known image datasets representing image recognition (IR) for standardized effectiveness tests in Table 2. The four PMS datasets are selected for various PMS applications, *e.g.*, activity for daily living (ADLs) recognition, health behavior monitoring (HBM), and gesture recognition (GR). We use three off-the-shelf smartphones with different resource capacities shown in Table 3 to evaluate the resource efficiency

TABLE 3: Mobile device specifications

Device	Year	ROM	RAM	CPU	Battery	OS
Huawei nova 6 SE	2019	128GB	8GB	Kirin 810	4200mAh	Android 10
Google Pixel 2 XL	2017	64GB	4GB	Snapdragon 835	3520mAh	Android 8.1.0
Google Pixel	2016	32GB	4GB	Snapdragon 821	2770mAh	Android 8.1.0
Samsung Gear S3	2016	4GB	768MB	Exynos 7 Dual 7270	380mAh	Tizen 4.0.0.4

of MDLdroidLite in reality, in which the screen battery drain is excluded in the results.

DNNs Selection In principle, the proposed RIC approach can be applied to optimize any DNN structures on devices. To fairly evaluate the performance of resource-accuracy efficiency, we employ three well-known DNN structures from small- to large-scale (*i.e.*, LeNet, MobileNet and VGG-11), detailed in Table 1. In particular, since MobileNet employs a set of depthwise separable convolutions (*i.e.*, ConvDW and ConvP in Table 1), it intrinsically achieves much less computational complexity (*i.e.*, FLOPs) comparing to normal DNNs, hence applying MobileNet is widely popular on mobile devices for resource-efficient purpose. Also, recent study [47] employs MobileNet to solve sensing tasks on mobile devices, achieving a fair resource-accuracy performance. Practically, we scale down the layer number of a standard MobileNet to fit sensor data.

Hyper-parameters We select Adam [48] as the default stochastic gradient descent optimization, and set a fixed learning rate to 0.0005. We set batch size with 64 for PMS datasets, and 100 for image datasets. The model parameters and noise are randomly initialized following a uniform distribution in $[-1, 1]$. We also apply a 2-epoch countdown early stop strategy [42]. Practically, we set each tiny structure as 10% of full-sized convolution layer and 2% of full-sized fully-connected layer shown in Table 1. We report top-1 accuracy throughout the evaluation.

4.3 RIC-adaption Pipeline Performance

We first evaluate the performance of RIC-adaption pipeline in terms of the variance of *existing-to-new* minimization, safe parameter adaptation, fast convergence rate, and time-to-accuracy efficiency.

Baselines We select three state-of-the-art parameter adaptation methods in CG as our baselines.

- **NeST-bridge** [16] uses a bridging-gradient transformation function to adapt new-born parameters in fully-connected layers, and utilizes trial-and-error to randomly generate parameters in convolution layers (*e.g.*, we set 10 trails per growth).
- **CGaP-select** [20] uses a saliency-based selective parameter adaptation method, transferring new-born parameters by picking up existing parameters with the highest saliency scores.
- **Net2WiderNet** [21] is based on a standard random duplication function with a safe compensation scale design for *existing-to-new*.

For a standardized effectiveness comparison, we employ a LeNet on MNIST to evaluate RIC-adaption pipeline running on a Pixel 2XL smartphone without constant power charging. Our experiments run with the same growth rate of 0.6 (*i.e.*, 60% per growth) and the same growth phase (*i.e.*, every 3-epoch growth) of CGaP-select to reach the full size. We run each experiment 5 times, and train each model with 30-epoch.

We conduct four experiments to compare RIC-adaption pipeline with the above baselines. The overall results present that RIC-adaption pipeline guarantees a safe parameter adaptation and outperforms the baselines with a minimized variance of *existing-to-new* in the growth. In addition, RIC-adaption pipeline achieves fast convergence by minimizing the channel-level variance during model growth, which speeds up on-device training convergence by 2.84× to 4.88× over the baselines. Please refer to our conference paper [25] for the detailed results about these experiments.

4.4 MDLdroidLite Performance

We now examine the performance of MDLdroidLite in terms of growth control fine-tuning, on-device time-to-accuracy structure efficiency, and on-device DL resource reduction using a range of datasets. To evaluate the improvement of MDLdroidLite+, we conduct several on-device experiments, including growth convergence stability, time horizon fine-tuning, and resource-accuracy run-time performance.

Baselines We select two state-of-the-art CG methods, one simplified NAS and one state-of-the-art pruning approaches as our baselines.

- **NeST** [16] is a linear growth approach. The model structure continually grows with a fixed phase until reaching the full size.
- **CGaP** [20] presents an exponential growth using a fixed growth rate. It also involves a simple pre-setting resource budget in growth.
- **S-search** [49] is a simplified NAS approach using Evolutionary algorithm with randomly parameter initialization. It performs a single-path search to select a candidate with the highest accuracy from a small amount of population.
- **Pruning** [37] is applied as an effectiveness baseline to compare the resource efficiency of MDLdroidLite. It is used for pruning a pre-trained model outside of smartphones to achieve a backbone structure with a fair accuracy.
- **Full Size** represents a conventional way of training DNNs with a full-sized model configuration shown in Table 1.

For benchmarking, we train MobileNet on three PMS datasets (*e.g.*, HAR, MHEALTH, and sEMG), LeNet on HAR, and LeNet on MNIST, respectively, on a Pixel 2XL smartphone without constant charging to compare the performance of MDLdroidLite with the baselines. Due to the lack of pruning support on off-the-shelf smartphones, the comparison of the structure growth performance is done on NeST and CGaP. To ensure the efficiency performance of S-search on device, we implement it as a linear candidate selection (*i.e.*, maximum two candidates are simultaneously active in memory) to avoid intensive memory use. For each experiment, we start training on a fully-charged smartphone and record its actual battery consumption throughout the experiment.

4.4.1 *Growth Control Fine-tuning*

In this experiment, we evaluate the growth control performance of MDLdroidLite with five different sizes of time horizon (TH) (*e.g.*, from TH-1 to TH-5) in terms of the accuracy-to-FLOPs efficiency and the control resource overhead (*e.g.*, time and battery consumption). The results show that different TH sizes may lead to a better resource-efficient DNN structure (*e.g.*, TH-4 for LeNet on MNIST), but the control resource overhead may increase as the size of TH increases. Practically, the size of TH should be safely managed

in a small range to assist dynamic control depends on actual resource budgets. Please refer to our conference paper [25] for the detailed results about this experiment.

4.4.2 On-device Time-to-accuracy Structure Efficiency

We continue evaluating the time-to-accuracy performance of MDLdroidLite by comparing to the baselines. We employ LeNet on both HAR and MNIST to measure the differences of using the same model with different datasets. We run each experiment 5 times. When 2-epoch countdown early stopping is applied, training will be stopped as soon as the accuracy is achieved. The results present that MDLdroidLite achieves a superior time-to-accuracy efficiency over the baselines, *e.g.*, training LeNet on HAR in MDLdroidLite outperforms the baselines by 2.9×, 2.4×, 3.13× and 4.6× faster over CGaP [20], NeST [16], S-search [49] and a full-sized model, respectively. Please refer to our conference paper [25] for the detailed results about this experiment.

4.4.3 On-device DL Resource Reduction

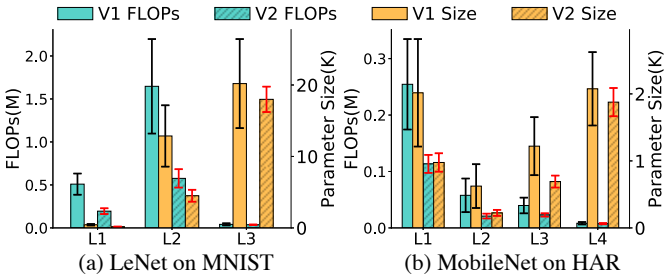
In this experiment, we quantify on-device resource reduction. The results show that MDLdroidLite achieves significant resource reduction on both model FLOPs and parameters, *e.g.*, the "grow" LeNet on MNIST reduces model parameters and FLOPs by 12× and 2.65×, respectively, over a full-sized model. We further measure the specific resource reduction, and the results present that the backbone models in MDLdroidLite achieves the lowest memory footprints and battery consumption, comparing to the baselines. Please refer to our conference paper [25] for the detailed results about this experiment.

4.4.4 Growth Convergence Stability

In this experiment, we evaluate the growth convergence stability of MDLdroidLite+ marked as V2 to quantify the improvements, comparing to MDLdroidLite marked as V1. Since we use fixed penalty regulators by manual set-up in V1, the growth convergence usually performs as unstable and sensitive to different set-ups. To evaluate the effectiveness of the proposed dynamic penalty function in V2, We apply 4 different penalty regulators (*e.g.*, from 0.1 to 0.4) to train each model 4 times up to the same accuracy level to examine the differences between V1 and V2. Fig. 8 reports a layer-level resource breakdown (*e.g.*, FLOPs and model size). The results present that V2 achieves a notable resource reduction especially in convolution layers of both models, *e.g.*, 2.6× and 2.24× FLOPs reduction on average in layer-1 of both models, respectively. Also, V2 performs a more stable growth convergence than V1 indicated by the range of error bars, *e.g.*, Fig. 8a shows that the standard deviation of both FLOPs and model size on V2 is significantly reduced by 5.14× and 5.15× over that on V1 in layer-2, respectively. In short, MDLdroidLite+ outperforms MDLdroidLite to achieve better growth convergence stability, and we henceforth set the penalty regulator to 0.4 as default.

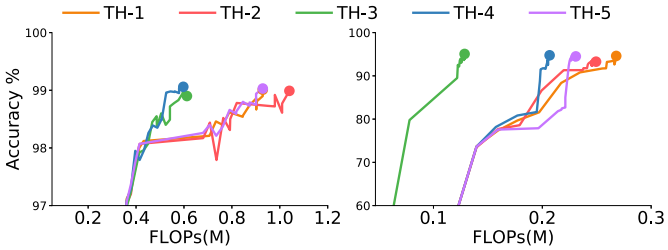
4.4.5 Time Horizon Fine-tuning

Since we propose both dynamic penalty and layer-to-layer cost functions in MDLdroidLite+ to further optimize resource efficiency, we next re-evaluate whether the modifications affect the performance of the growth control fine-tuning. We hence re-apply the five different TH sizes to



(a) LeNet on MNIST (b) MobileNet on HAR

Fig. 8: Growth convergence stability of MDLdroidLite+.



(a) LeNet on MNIST (b) MobileNet on HAR

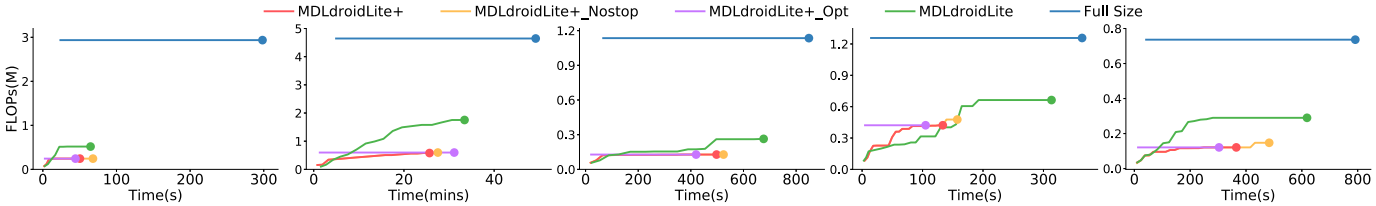
Fig. 9: Time horizon fine-tuning for MDLdroidLite+.

evaluate accuracy-to-FLOPs efficiency in MDLdroidLite+. Fig. 9 reports that using TH-4 on LeNet and TH-3 on MobileNet still achieve the best growth results, *e.g.*, using TH-4 on LeNet in MDLdroidLite+ achieves the best accuracy of 99.06% but with 3.1× FLOPs reduction comparing to that in MDLdroidLite. The results demonstrate that the modifications improve the growth resource efficiency with little impact on the TH fine-tuning. Thus, we consistently use the same TH settings in MDLdroidLite+.

4.4.6 Resource-accuracy Run-time Performance

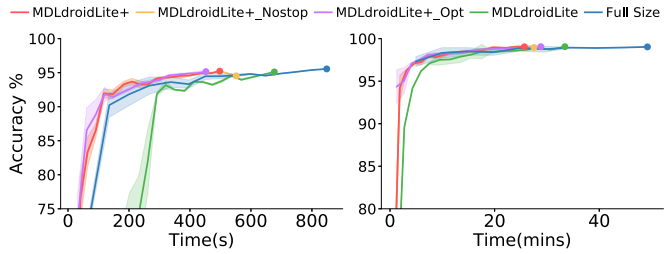
We now examine both time-to-accuracy and time-to-FLOPs improvements of MDLdroidLite+ using a range of datasets in run-time. We fairly train different models on different datasets to achieve their best accuracy, and record the run-time resource changes during the growth. To examine whether the early stopping affects the learning performance of MDLdroidLite+, we apply a baseline without early stopping applied throughout the training (*i.e.*, 20-epoch as a default completion), marked as MDLdroidLite+_Nostop. In addition, to investigate the training performance of MDLdroidLite+ comparing to that of using the optimized structure (*i.e.*, the output of the RIC pipeline) for training from scratch, we assume to define a baseline that trains the acquired resource-efficient models from scratch without using the proposed pipeline, marked as MDLdroidLite+_Opt.

Time-to-FLOPs Run-time Performance Fig. 10 shows the run-time time-to-FLOPs performance on four datasets. From the results, we obverse that MDLdroidLite+ achieves a considerable FLOPs and training time reduction over MDLdroidLite, *e.g.*, MobileNet on HAR in MDLdroidLite+ reduces FLOPs and training time by 2.06× and 1.36× over that in MDLdroidLite, respectively, achieving an accuracy of 95.22%. In addition, the time-to-FLOPs line of MDLdroidLite+ presents a faster growth convergence (*i.e.*, the line becomes stable earlier) than that of MDLdroidLite. Furthermore, the results show that MDLdroidLite+_Nostop achieves an identical (*e.g.*, remaining as 0.24M and



(a) LeNet on HAR (b) LeNet on MNIST (c) MobileNet on HAR (d) MobileNet on MHEALTH (e) MobileNet on sEMG

Fig. 10: Time-to-FLOPs run-time performance of MDLdroidLite+.



(a) MobileNet on HAR (b) LeNet on MNIST

Fig. 11: Time-to-accuracy performance of MDLdroidLite+.

0.13M FLOPs for both LeNet and MobileNet on HAR, respectively) or slightly higher FLOPs (*e.g.*, 1.03× higher for LeNet on MNIST) comparing to MDLdroidLite+, but the training time of MDLdroidLite+_Nostop is indeed slower. Hence, applying early stopping has a minor effect to the structure learning performance of MDLdroidLite+ while effectively saving training time in reality. By comparison with MDLdroidLite+_Opt, the results present that MDLdroidLite+ runs slightly slower to achieve the best accuracy on all PMS datasets, *e.g.*, 77.74s, 28.41s, and 61.75s slower than MDLdroidLite+_Opt for MobileNet on HAR, MHEALTH, and sEMG, respectively. Since larger sized models (*e.g.*, optimized models) may achieve a faster training convergence throughout the training, training these models from scratch may perform a better resource-accuracy efficiency performance than training the seed models by the proposed pipeline. Interestingly, we observe that MDLdroidLite+ trains faster than MDLdroidLite+_Opt for LeNet on MNIST (*e.g.*, 5.48 mins faster), in which RIC-adaption pipeline in MDLdroidLite+ may effectively contribute to a fast training convergence during the growth.

Time-to-accuracy Run-time Performance We continue to examine the time-to-accuracy performance of MDLdroidLite+. We re-employ both MobileNet on HAR and LeNet on MNIST, and manage each training 5 times with 2-epoch countdown early stopping applied. Fig. 11a shows that training MobileNet on HAR in MDLdroidLite+ achieves the best result, *e.g.*, takes 497.67s to achieve an accuracy of 95.22% on average, speeds up training by 1.36× over that in MDLdroidLite. Similarly, Fig. 11b reports that LeNet on MNIST in MDLdroidLite+ takes 25.67 mins to reach an equivalent accuracy of 99.04% on average, which is 1.3× faster than that in MDLdroidLite. In addition, from the results, we observe that MDLdroidLite+_Nostop performs a lower accuracy for both models comapring to MDLdroidLite+, *e.g.*, achieves a 0.68% and 0.08% accuracy drop on average for both MobileNet on HAR and LeNet on MNIST, respectively. Thus, applying early stopping in MDLdroidLite+ can efficiently terminate on-device training in time without accuracy drop. Furthermore, since MDLdroidLite+_Opt runs faster than MDLdroidLite+ on PMS datasets, MDLdroidLite+ achieves a slightly lower accuracy when MDLdroidLite+_Opt completes (*e.g.*, 0.24% lower on average for MobileNet on HAR). However, when MDLdroidLite+ completes, the achieved accuracy is even higher than that in MDLdroidLite+_Opt (*e.g.*, 0.07% higher on average for MobileNet on HAR). Besides, the results demonstrate that LeNet on MNIST in MDLdroidLite+ takes less training time to reach the same accuracy as that in MDLdroidLite+_Opt with no drop. Thus, the proposed pipeline not only efficiently optimizes DNN structures on devices, but also guarantees a fair accuracy performance.

This experiment demonstrates that the improved design in MDLdroidLite+ further optimizes model structure (*i.e.*, less resource overhead) with an equivalent or higher accuracy.

4.5 Real-world Use Scenarios

We next evaluate the resource-accuracy efficiency of MDLdroidLite+ in different use scenarios using a range of datasets and different DNN models on commodity smartphones.

4.5.1 *Continual Training Scenarios*

In this experiment, we evaluate the resource-accuracy efficiency of MDLdroidLite+ in continual training scenarios. Since MDLdroidLite+ aims for a universal resource-constrained approach towards MDL, it not only works for training from scratch, but also can be applied to continually train a pre-trained model to recover the model accuracy and robustness in the wild. Although existing transfer learning with Pruning (TP) may be strongly limited to build pre-trained models for PMS applications (*e.g.*, strong domain-shift, lack of related source model or public dataset), we specifically prepare the pre-trained models by TP outside of smartphones as baselines in this experiment. We conduct this experiment in two real-world continual training scenarios [50]–domain-incremental (*i.e.*, input-distribution changes due to sensor data dynamics) and class-incremental (*i.e.*, learning new classes over time) scenarios. In both scenarios, we perform *two-stage* training, *e.g.*, stage-1 and 2 represent the conditions before and after the increments, respectively. Specially, we apply MDLdroidLite+ on top of TP (*e.g.*, TP+RIC) in stage-2 to evaluate the recovery performance for the learning forgetting issue [19].

Domain-incremental Scenario Since we re-apply the stationary-to-movement change in Section 2 as a domain-incremental scenario (*i.e.*, sensor data change from indoor

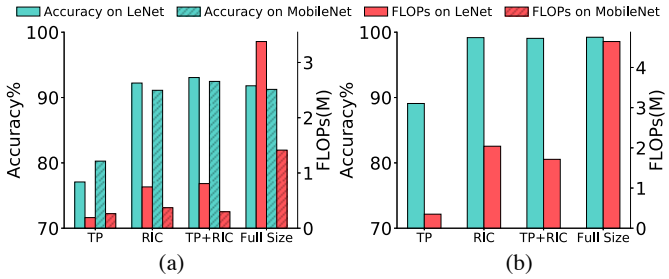


Fig. 12: Resource-accuracy performance of MDLdroidLite+. (a) FinDroidHR in domain-incremental scenario; (b) MNIST in class-incremental scenario.

sitting to outdoor running), we first train both LeNet and MobileNet on the indoor sitting data by TP, MDLdroidLite+, and full-sized model to achieve the state-of-the-art accuracy (*e.g.*, 98.19% and 98.34% on average, respectively) in stage-1. We next add the outdoor running data as a "domain" increment to continually train the models in stage-2. Fig. 12a presents the results of stage-2. Although both models by TP achieve the lowest FLOPs (*i.e.*, structure fixed as minimal in stage-1), the accuracy notably decreases due to learning new running data, *e.g.*, 14.72% and 10.98% drop, respectively, comparing to full-sized models. The results using MDLdroidLite+ show that not only the accuracy of both models remains at the same level after re-learning, but also the FLOPs of both models is managed much smaller than that of full-sized models. Due to the structure growth of MDLdroidLite+, though the FLOPs of both models by TP+RIC increase in a small amount, the accuracy is recovered and even higher than that of MDLdroidLite+.

Class-incremental Scenario We next select 5 out of 10 classes of MNIST to train LeNet in stage-1, and then add the rest of 5 classes for continual training in stage-2 to set up a class-incremental scenario. In Fig. 12b, we achieve similar results as above that the accuracy by TP drops by 10.14% with the minimal FLOPs, but it is recovered back to 99.06% using MDLdroidLite+ (*e.g.*, FLOPs increases by 1.36M in stage-2) without learning forgetting.

Through both real-world continual training scenarios, we obverse that TP is limited to continually learn new data or classes due to the fixed structure, hence it causes the learning forgetting issue to degrade model accuracy and robustness in the wild. This experiment demonstrates MDLdroidLite+ is capable to remain or recover the on-device model performance in continual training scenarios.

4.5.2 *Real-world Application Performance*

To evaluate the real-world performance of MDLdroidLite, we develop a hand gesture recognition application shown in Fig. 7a, and apply MobileNet for resource-efficient purpose in this experiment. We first ask the subject to collect a training dataset that contains 6 gestures and 120 data instances per gesture. Annotation is done manually using the application on smartwatch. We then apply three data augmentation techniques [13] to augment data on device: 1) *adding noise* is to randomly add noise generated by the IID (*i.e.*, independent and identically distributed) distribution to original sensor data; 2) *drift* is to drift the value of sensor data from its original values randomly and smoothly; 3) *pool* is to reduce the temporal resolution of sensor data while keeping the length.

For the updated results in MobileNet on Pixel 2XL, the backbone model in MDLdroidLite+ achieves a slightly higher accuracy of 98.63% with 1.9× less parameters, and 1.59× less FLOPs than that in MDLdroidLite. In addition, the training time, battery consumption, and batch latency of the backbone model in MDLdroidLite+ are reduced by 1.24×, 1.27×, and 1.18×, respectively. Thus, this case study shows that MDLdroidLite+ further boosts on-device training and inference with the resource-efficient backbone model for real-world PMS applications.

To compare the performance of MDLdroidLite with the conventional ML methods on the collected dataset, we employ both SVM and Random Forest with a fully hand-crafted feature engineering [9] as baselines. We also select 17 out of 182 features to improve the classification performance for both ML methods. Since the conventional ML methods require a hand-crafted training process outside of devices, we may not be able to compare the run-time on-device training performance with MDLdroidLite, hence we evaluate the training accuracy as the metric. From the results, we observe that both ML methods achieve the inferior accuracy comparing to MDLdroidLite, *e.g.*, 94.24% using SVM and 90.6% using Random Forest. Thus, DL-based methods not only ideally work for on-device learning towards the promising MDL paradigm due to the automated feature extraction capability, but also present superior accuracy performance to fit for real-world sensing tasks.

4.5.3 *On-device Resource-accuracy Efficiency*

We now summarize the resource-accuracy efficiency results of both MDLdroidLite and MDLdroidLite+ on three smartphones, in terms of parameters, FLOPs, memory, time, battery, and batch latency (B-latency). The experiments are done using small- to large-scale model configuration (*i.e.*, LeNet, MobileNet and VGG-11) on all six datasets.

For the results in LeNet on PMS datasets, the backbone models in MDLdroidLite achieve 28× to 50× on parameters reduction, 4× to 10× FLOPs reduction, 1.83× to 4.96× speedup on average over the full-sized model. Comparing to MDLdroidLite, the backbone models in MDLdroidLite+ reduce parameters from 1.42× to 1.73× and FLOPs from 1.23× to 3.17× with no accuracy drop, *e.g.*, the structure of the backbone model on HAR turns down to [5-13-25-6] with a higher accuracy of 94.97%. For the results in MobileNet on PMS datasets, the parameters, FLOPs, and training time of the backbone models in MDLdroidLite are reduced by 4× to 7×, 2× to 7×, and 1.27× to 1.56× over the full-sized model, respectively. In addition, the backbone models in MDLdroidLite+ achieve further improvement over MDLdroidLite, *i.e.*, parameters reduction from 1.9× to 2.21×, FLOPs reduction from 1.57× to 2.42×, and training time reduction from 1.24× to 2.35×. Since training a large-scale VGG-11 on device is quite costly, it cannot achieve a fair accuracy due to battery drain. However, a backbone VGG-11 in MDLdroidLite can safely achieve an accuracy of 75%+ with 1328mAh battery consumption. In MDLdroidLite+, the backbone VGG-11 on CIFAR-10 achieves a higher accuracy of 76.14% with less battery consumption on the Pixel 2XL.

TABLE 4: On-device resource-efficient study using LeNet.

Dataset	Method	Accuracy	Parameter	FLOPs	Memory	Pixel 2XL			Pixel			Nova 6SE			Structure
						Time	Battery	B-latency	Time	Battery	B-latency	Time	Battery	B-latency	
sEMG	Full Size [1]	87.8%	394.82K	2.0M	165M	208s	57mAh	164ms	290s	79mAh	210ms	310s	85mAh	166ms	[20-50-500-6]
	S-search [49]	86.03%	177.52K	1.22M	161M	142s	39mAh	131ms	197s	54mAh	168ms	211s	58mAh	132ms	[16-42-261-6]
	MDLdroidLite	86.47%	14.59K	0.5M	148M	92s	25mAh	114ms	128s	35mAh	146ms	137s	37mAh	115ms	[12-24-25-6]
	MDLdroidLite+	**87.46%**	**9.7K**	**0.34M**	**139M**	**77s**	**21mAh**	**110ms**	**107s**	**29mAh**	**141ms**	**114s**	**31mAh**	**111ms**	**[10-15-28-6]**
	Pruning [37]	87.08%	41.01K	0.94M	152M	-	-	118ms	-	-	151ms	-	-	119ms	[17-38-52-6]
HAR	Full Size [1]	94.6%	570K	2.9M	175M	298s	119mAh	154ms	415s	166mAh	197ms	444s	178mAh	156ms	[20-50-500-6]
	S-search [49]	95.13%	153K	1.8M	168M	203s	81mAh	135ms	283s	113mAh	173ms	302s	121mAh	136ms	[18-45-140-6]
	MDLdroidLite	94.46%	15.38K	0.52M	148M	65s	26mAh	114ms	90s	36mAh	146ms	96s	39mAh	115ms	[10-16-33-6]
	MDLdroidLite+	**94.97%**	**8.89K**	**0.24M**	**140M**	**50s**	**20mAh**	**107ms**	**70s**	**28mAh**	**137ms**	**75s**	**30mAh**	**108ms**	**[5-13-25-6]**
	Pruning [37]	94.9%	24.72K	0.66M	152M	-	-	122ms	-	-	156ms	-	-	123ms	[12-18-50-6]
MHEALTH	Full Size [1]	96.11%	401.52K	2.74M	197M	78s	23mAh	171ms	109s	32mAh	219ms	116s	34mAh	173ms	[20-50-500-11]
	S-search [49]	95.7%	84.49K	1.76M	187M	163s	47mAh	142ms	227s	66mAh	182ms	243s	70mAh	143ms	[18-40-112-11]
	MDLdroidLite	95.19%	9.68K	0.5M	168M	72s	21mAh	126ms	101s	29mAh	161ms	108s	31mAh	127ms	[7-15-25-11]
	MDLdroidLite+	**95.5%**	**6.83K**	**0.34M**	**155M**	**66s**	**19mAh**	**111ms**	**92s**	**27mAh**	**142ms**	**98s**	**28mAh**	**112ms**	**[5-11-25-11]**
	Pruning [37]	94.67%	21.5K	0.63M	179M	-	-	128ms	-	-	164ms	-	-	129ms	[8-22-48-11]
FinDroidHR	Full Size [1]	98.89%	694.26K	3.38M	183M	55s	19mAh	133ms	76s	27mAh	170ms	81s	28mAh	134ms	[20-50-500-6]
	S-search [49]	98.61%	175.85K	1.76M	158M	61s	21mAh	117ms	85s	30mAh	150ms	91s	32mAh	118ms	[15-46-132-6]
	MDLdroidLite	98.01%	17.56K	0.27M	151M	30s	10mAh	105ms	42s	15mAh	134ms	44s	16mAh	106ms	[5-15-39-6]
	MDLdroidLite+	**98.61%**	**10.63K**	**0.22M**	**146M**	**18s**	**6mAh**	**104ms**	**25s**	**9mAh**	**133ms**	**26s**	**9mAh**	**105ms**	**[5-11-31-6]**
	Pruning [37]	98.05%	18.86K	0.65M	152M	-	-	111ms	-	-	142ms	-	-	112ms	[12-18-30-6]
MNIST	Full Size [1]	99.08%	431.08K	4.9M	256M	49mins	1095mAh	280ms	68mins	1524mAh	358ms	73mins	1629mAh	283ms	[20-50-500-10]
	S-search [49]	98.95%	219.71K	3.9M	242M	154mins	2819mAh	205ms	214mins	3923mAh	262ms	228mins	4195mAh	207ms	[20-45-269-10]
	MDLdroidLite	99.06%	38.06K	1.84M	197M	33mins	765mAh	166ms	47mins	1065mAh	212ms	50mins	1138mAh	168ms	[13-33-50-10]
	MDLdroidLite+	**99.04%**	**20.96K**	**0.58M**	**176M**	**26mins**	**571mAh**	**142ms**	**36mins**	**795mAh**	**182ms**	**38mins**	**850mAh**	**143ms**	**[5-25-43-10]**
	Pruning [37]	99.01%	35.8K	2.04M	188M	-	-	168ms	-	-	215ms	-	-	170ms	[16-29-50-10]

TABLE 5: On-device resource-efficient study using MobileNet.

Dataset	Method	Accuracy	Parameter	FLOPs	Memory	Pixel 2XL			Pixel			Nova 6SE			Structure
						Time	Battery	B-latency	Time	Battery	B-latency	Time	Battery	B-latency	
sEMG	Full Size [1]	87.15%	50.31K	0.74M	161M	791s	265mAh	186ms	1148s	384mAh	322ms	886s	297mAh	212ms	[32-64-128-256-6]
	S-search [49]	84.23%	8.39K	0.34M	155M	1153s	386mAh	164ms	1671s	560mAh	284ms	1291s	432mAh	187ms	[24-38-34-68-6]
	MDLdroidLite	85.97%	7.56K	0.29M	145M	618s	207mAh	136ms	897s	300mAh	235ms	693s	232mAh	155ms	[20-35-40-52-6]
	MDLdroidLite+	**86.47%**	**3.75K**	**0.12M**	**127M**	**365s**	**122mAh**	**125ms**	**530s**	**177mAh**	**216ms**	**409s**	**137mAh**	**142ms**	**[10-12-31-45-6]**
	Pruning [37]	85.1%	5.63K	0.22M	137M	-	-	133ms	-	-	230ms	-	-	152ms	[18-17-38-47-6]
HAR	Full Size [1]	95.55%	51.72K	1.14M	171M	848s	305mAh	191ms	1230s	443mAh	330ms	950s	342mAh	218ms	[32-64-128-256-6]
	S-search [49]	95.24%	15.51K	0.49M	158M	868s	313mAh	173ms	1259s	453mAh	299ms	972s	350mAh	197ms	[19-34-63-126-6]
	MDLdroidLite	95.11%	7.37K	0.26M	149M	676s	243mAh	136ms	981s	353mAh	235ms	758s	273mAh	155ms	[10-30-48-58-6]
	MDLdroidLite+	**95.22%**	**3.33K**	**0.13M**	**120M**	**498s**	**179mAh**	**125ms**	**722s**	**260mAh**	**216ms**	**557s**	**201mAh**	**142ms**	**[6-12-31-39-6]**
	Pruning [37]	95.72%	6.21K	0.2M	146M	-	-	136ms	-	-	235ms	-	-	155ms	[9-16-35-86-6]
MHEALTH	Full Size [1]	97.86%	57.35K	1.26M	176M	364s	138mAh	175ms	528s	201mAh	303ms	408s	155mAh	200ms	[32-64-128-256-11]
	S-search [49]	97.64%	25.61K	0.78M	163M	582s	221mAh	167ms	844s	321mAh	289ms	652s	248mAh	190ms	[23-41-77-153-11]
	MDLdroidLite	97.85%	13.65K	0.66M	153M	313s	119mAh	150ms	454s	173mAh	260ms	351s	133mAh	171ms	[20-48-51-53-11]
	MDLdroidLite+	**97.94%**	**6.78K**	**0.42M**	**131M**	**133s**	**51mAh**	**124ms**	**194s**	**74mAh**	**215ms**	**149s**	**57mAh**	**141ms**	**[15-17-23-39-11]**
	Pruning [37]	97.84%	13.15K	0.6M	150M	-	-	150ms	-	-	260ms	-	-	171ms	[20-26-43-91-11]
FinDroidHR	Full Size [1]	99.16%	50.37K	1.41M	187M	155s	51mAh	248ms	224s	74mAh	429ms	173s	57mAh	283ms	[32-64-128-256-6]
	S-search [49]	98.61%	33.21K	1.03M	168M	119s	39mAh	225ms	173s	57mAh	389ms	134s	44mAh	256ms	[28-52-101-202-6]
	MDLdroidLite	98.61%	7.16K	0.27M	150M	99s	33mAh	167ms	144s	48mAh	289ms	111s	37mAh	190ms	[10-20-61-54-6]
	MDLdroidLite+	**98.63%**	**3.76K**	**0.17M**	**134M**	**80s**	**26mAh**	**141ms**	**116s**	**38mAh**	**244ms**	**90s**	**30mAh**	**161ms**	**[8-14-35-41-6]**
	Pruning [37]	98.61%	6.46K	0.32M	148M	-	-	175ms	-	-	303ms	-	-	200ms	[16-17-34-83-6]

TABLE 6: On-device resource-efficient study using VGG-11.

Dataset	Method	Accuracy	Parameter	FLOPs	Memory	Pixel 2XL			Pixel			Nova 6SE			Structure
						Time	Battery	B-latency	Time	Battery	B-latency	Time	Battery	B-latency	
CIFAR-10	Full Size	71.99%	9.23 M	306.42 M	650M	164mins	3316mAh	1971ms	-	-	3410ms	167mins	3382mAh	2247ms	[64-128-256-256-512-512-512-512-10]
	MDLdroidLite	74.9%	0.35M	19.52M	300M	68mins	1377mAh	445ms	80mins	1611mAh	770ms	69mins	1405mAh	507ms	[18-37-77-77-86-86-86-86-10]
	MDLdroidLite+	**76.14%**	**0.28M**	**11.96M**	**275M**	**56mins**	**1128mAh**	**344ms**	**65mins**	**1320mAh**	**595ms**	**57mins**	**1151mAh**	**392ms**	**[15-30-49-49-86-86-86-86-10]**
	Pruning [37]	80.11%	0.44M	22.73M	310M	-	-	465ms	-	-	804ms	-	-	530ms	[15-51-70-84-109-107-89-87-10]

Although pruning performs a better resource-accuracy efficiency on VGG-11, the pruning process takes nearly 5.4 hours on a laptop with a NVIDIA GeForce GTX 1080 Ti GPU, which is impractical to run on smartphones. The complete results are presented in Table 4 5 6, in which "-" denotes either no available on-device result or running out of resource budget.

5 Discussion and Future Work

Task and Model Structure Complexity Traditional model structure is usually made to be deep and large-scale to solve the complex tasks (*i.e.*, large size of inputs and number of classes) for high accuracy purpose [17], [18], especially in computer vision and image tasks. However, most of real-world PMS applications require much smaller input size (*i.e.*, limited sampling rate up to 100Hz and channels due to energy consumption on devices) and class number than image tasks, which is naturally less task complexity. In particular, to remain at the same task complexity level with the selected PMS datasets, we select CIFAR-10 instead of CIFAR-100 as a fair baseline. Hence, the high-accuracy requirement in PMS applications is usually not placed at the first priority. Instead, resource efficiency is more critical in the sensing domain. To match the sensing task complexity, lightweight or shallow models usually achieve better resource-accuracy performance than large-scale models in practice [47], [51], because they effectively avoid overfitting issues. Although training large-scale models on device may not be efficient enough, *e.g.*, training VGG-11 on CIFAR-10 in our evaluation, but MDLdroidLite moves the first step, which opens numerous possibilities to advanced on-device DL applications. We will extend the capability of MDLdroidLite to support more state-of-the-art model configurations in our future work.

Insufficient Training Data Data augmentation effectively solves the problem of insufficient training data. However, the problem may still exist in real-world PMS applications, especially at the bootstrapping stage. In this case, MDLdroidLite can start with a pruned pre-trained model, and continually fine-tune the model structure over time to maintain the optimal performance. Alternatively, existing MDL frameworks such as FL [12] and MDLdroid [8] can be incorporated to perform collaborative learning leveraging multiple users. Although MDLdroidLite may not currently work with these frameworks due to the dynamic structure, the challenge of heterogeneous model aggregation will be

addressed in our future work.

Lifelong Learning Scenario With the capability of on-device continual training in MDLdroidLite, lifelong learning [22] can be achieved on devices. However, to address the learning forgetting issue in lifelong learning scenarios, the model structure of MDLdroidLite may eventually grow to be large over time, leading to resource overhead on devices. We may apply a maximum resource constraint threshold to bound structure size, but it may fail to avoid the learning forgetting due to constrained structure. We plan to address this challenge in our future work.

6 Related Work

Constructive Structure Adaptation A constructive approach is able to grow and expand DNNs from a small structure. Recent works combine both constructive and destructive approaches into CG, but their efficiency seriously relies on pruning. NeST [16] proposes a linear CG approach by continually growing the layer width, but it is costly due to the random trial-and-error used to grow channels in convolution layers, and importantly a linear growth without control yields inferior training performance. Similarly, CGaP [20] shows a near-exponential growth with a saliency-based selective KT function, but the growth strategy is arbitrary and it could easily run into overparameter which can be too expensive for on-device training. Different from these works, MDLdroidLite follows the idea of structure growth but wisely controls a single trajectory growth through resource-constrained optimization to transform a traditional DNN structure to a resource-efficient DNN for mobile devices.

Knowledge Transfer Adaptation Existing CG methods enable fast parameter adaptation using KT but suffer from slow convergence under resource-constrained conditions. Net2net [21] proposes a standard random duplication function but with a safe compensation scale in each subsequent layer for rapid knowledge transformation. NeST [16] designs a bridging-gradient transformation function to help the neurons growth in fully-connected layers, but the newborn neurons identified as being inactive without coordination may present a weak contribution to slow down training. Differently, CGaP [20] employs a saliency-based selective duplication to achieve higher accuracy, but the training loss presents a degraded spiking after each transformation, hence the loss is notably unstable to yield inferior convergence performance. In contrast, leveraging RIC-adaption pipeline, MDLdroidLite not only safely adapts new-born neurons using a three-step DSPA, but also designs a GCU to minimize the variance between new-born and existing neurons, resulting in fast convergence after each grow-step to significantly speed up on-device training.

7 Conclusion

This paper presents a novel on-device structure learning framework that enables resource-efficient DNNs on mobile devices. MDLdroidLite is able to perform on-device training from scratch, continual learning to support personalized, privacy-preserving PMS applications. Moreover, MDLdroidLite achieves efficient on-device training and inference performance for most of the state-of-the-art DNNs.

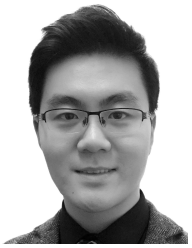
Acknowledgment

This work is supported by Australian Research Council (ARC) Discovery Project grants DP180103932 and DP190101888.

References

[1] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, "A survey of mobile phone sensing," *IEEE Communications Magazine*, 2010.
[2] R. Miotto, F. Wang, S. Wang, and X. Jiang, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, 2017.
[3] D. C. Mohr, M. Zhang, and S. M. Schueller, "Personal sensing: Understanding mental health using ubiquitous sensors and machine learning," *Annual review of clinical psychology*, 2017.
[4] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu, and Y. Liu, "Deep learning for sensor-based human activity recognition: Overview, challenges and opportunities," 2020.
[5] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *MobiSys '16*, 2016.
[6] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *MECOMM '18*, 2018.
[7] Z. Yu, H. Du, F. Yi, Z. Wang, and B. Guo, "Ten scientific problems in human behavior understanding," *CCF TPCI*, 2019.
[8] Y. Zhang, T. Gu, and X. Zhang, "Mdldroid: a chainsgd-reduce approach to mobile deep learning for personal mobile sensing," in *IPSN '20*, 2020.
[9] Y. Zhang, T. Gu, C. Luo, V. Kostakos, and A. Seneviratne, "Findroidhr: Smartwatch gesture input with optical heartrate monitor," *IMWUT '18*, 2018.
[10] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surv.*, 2020.
[11] L. Zhang, "Transfer adaptation learning: A decade survey," *CoRR*, 2019.
[12] J. Konecný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *CoRR*, 2016.
[13] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. F. Wang, and J.-B. Huang, "A closer look at few-shot classification," in *ICLR'19*, 2019.
[14] Y. Chen, S. Biookaghazadeh, and M. Zhao, "Exploring the capabilities of mobile devices supporting deep learning," in *HPDC '18*, 2018.
[15] A. Hard, K. Rao, R. Mathews, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated learning for mobile keyboard prediction," *CoRR*, 2018.
[16] X. Dai, H. Yin, and N. K. Jha, "Nest: A neural network synthesis tool based on a grow-and-prune paradigm," *TC*, 2019.
[17] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *ICLR '19*, 2019.
[18] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *ICLR*, 2019.
[19] Z. Li and D. Hoiem, "Learning without forgetting," *TPAMI*, 2018.
[20] X. Du, Z. Li, and Y. Cao, "Cgap: Continuous growth and pruning for efficient deep learning," *CoRR*, 2019.
[21] T. Chen, I. Goodfellow, and J. Shlens, "Net2net: Accelerating learning via knowledge transfer," 2015.
[22] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," in *ICLR '18*, 2018.
[23] E. E. Müller, V. Locatelli, and D. Cocchi, "Neuroendocrine control of growth hormone secretion," *Physiological Reviews*, 1999.
[24] W. Y. B. Lim, C. Nguyen, H. Dinh Thai, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, 2020.
[25] Y. Zhang, T. Gu, and X. Zhang, "Mdldroidlite: a release-and-inhibit control approach to resource-efficient deep neural networks on mobile devices," in *Sensys '20*, 2020.
[26] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *CoRR*, 2018.
[27] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *AAAI '17*, 2017.

[28] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *ICML '18*, 2018.
[29] T. Tran, L. Marsh, and R. Hunjet, "Reinforcement learning with model predictive control - recent development," 2019.
[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.
[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, 2017.
[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv*, 2014.
[33] N. Meuleau, E. Benazera, R. I. Brafman, E. A. Hansen, and Mausam, "A heuristic search approach to planning with continuous resources in stochastic domains," *J. Artif. Int. Res.*, 2009.
[34] L. Debnath and K. Basu, "A short history of probability theory and its applications," *International Journal of Mathematical Education in Science and Technology*, 2015.
[35] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *ICCV '17*, 2017.
[36] K. Persand, A. Anderson, and D. Gregg, "Composition of saliency metrics for channel pruning with a myopic oracle," 2020.
[37] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *ICRL '17*, 2017.
[38] L. Chunjie, Z. jianfeng, W. lei, and Y. Qiang, "Cosine normalization: Using cosine similarity instead of dot product in neural networks," 2017.
[39] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature," *Geoscientific Model Development*, 2014.
[40] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, 2010.
[41] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML'15*, 2015.
[42] T. Angles, R. Camoriano, A. Rudi, and L. Rosasco, "Nytro: When subsampling meets early stopping," 2016.
[43] S. Lobov, N. Krilova, I. Kastalskiy, V. Kazantsev, and V. Makarov, "Latent factors limiting the performance of semg-interfaces," *Sensors*, 2018.
[44] O. Banos, R. Garcia, J. A. Holgado-Terriza, M. Damas, H. Pomares, I. Rojas, A. Saez, and C. Villalonga, "mhealthdroid: A novel framework for agile development of mobile health applications," in *Ambient Assisted Living and Daily Activities*, L. Pecchia, L. L. Chen, C. Nugent, and J. Bravo, Eds., 2014.
[45] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *ESANN'13*, 2013.
[46] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.
[47] M. Nutter, C. H. Crawford, and J. Ortiz, "Design of novel deep learning models for real-time human activity recognition with mobile phones," in *IJCNN '18*, 2018.
[48] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, 2016.
[49] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," 2020.
[50] G. M. van de Ven and A. S. Tolias, "Three scenarios for continual learning," *CoRR*, 2019.
[51] A. Shrestha and J. Dang, "Deep learning-based real-time auto classification of smartphone measured bridge vibration data," *Sensors*, 2020.



Yu Zhang received the B.S. degree in Software Engineering from Southwest University for Nationalities, China. He is currently pursuing the Ph.D. degree in computer science at RMIT University, Australia. His research interests include mobile computing, on-device machine learning, wireless sensor network, embedded systems, Internet of Things and big data analytics.

Tao Gu is currently a Professor in Department of Computing at Macquarie University, Australia. His research interests include Internet of Things, ubiquitous computing, mobile computing, embedded AI, wireless sensor networks, and big data analytics. He is currently serving as an Editor of IMWUT, an Associate Editor of TMC and IoT-J. Please find out more information at https://taogu.site.

Xi Zhang received the B.S. degree from the Beijing Jiaotong University Haibin College of Computer Science, China in 2014, the M.S. degree from Monash University of Information Technology, Australia in 2018, and He is currently pursuing the Ph.D. degree of Computer Science in RMIT University, Australia. His research interests include Internet of Things, mobile computing and machine learning.