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

Yu Zhang
RMIT University, Australia
zac.lhjzyzzoo@gmail.com

Tao Gu
Macquarie University, Australia
tao.gu@mq.edu.au

Xi Zhang
RMIT University, Australia
zaibuer@gmail.com

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.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Neural networks**; **Computational control theory**; • **Human-centered computing** → **Ubiquitous and mobile devices**.

Keywords

Mobile Deep Learning, Deep Neural Networks, Dynamic Optimization Control, Resource Constraint

ACM Reference Format:
Yu Zhang, Tao Gu, and Xi Zhang. 2020. MDLdroidLite: a Release-and-Inhibit Control Approach to Resource-Efficient Deep Neural Networks on Mobile Devices . In *The 18th ACM Conference on Embedded Networked Sensor Systems (SenSys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3384419.3430716

SenSys '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-7590-0/20/11…$15.00
https://doi.org/10.1145/3384419.3430716

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 [36], continual personal health monitoring [44], to private mental contexts understanding [45]. Through these devices, PMS applications are able to exploit rich contexts from personal sensing data that may be privacy sensitive. Machine learning plays a vital role in interpreting and making sense to sensor data. In particular, Deep Learning (DL) has created tremendous opportunities to achieve breakthroughs in a higher level of accuracy and robustness [35].

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 [24] or edge servers [38] and download pre-trained models [35] on devices for inference. However, real-world PMS applications have many intrinsic properties which create several open questions [60, 63]. 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. Numerous studies [32, 48] reveal that unexpected data leaking may lead to privacy violation. *Data privacy* protection has also been raised to the level of regulations and laws [57]. 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) [7, 22, 35, 60], hence *continual* training or adaptation is crucial to maintain model generalization and robustness. Thirdly, PMS applications such as gesture recognition [62] and fall detection [36] require real-time responses, hence *low-latency* in model inference is critical. Furthermore, since sensor data are naturally less interpretable than images or texts [7], 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 [11, 34], 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 [56, 61]: 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 [16, 17, 30, 63]. MDL promises to offer unique features to enable strict *privacy preservation* (*i.e.*, *zero* data transmission), *continual training* and *low-latency inference* on mobile devices [30, 63]. Thanks to data augmentation techniques [9, 54] 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 [34], *e.g.*, TensorFlow Lite [40] and optimization in [24]. Few studies [10, 25] 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 [55]. Google's Federated Learning (FL) [30] 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. The study [19] 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 [20].

To reduce heavy model parameters in DNNs, model *pruning* has been proposed to achieve a lightweight structure with less resources used [42]. The study [20] 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 to a backbone structure for inference only, but not training [12, 58]. 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 [14, 20, 39]. 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 [42, 64]. 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 [8, 14, 18, 29] 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 newborns (*i.e.*, new added neurons, channels or layers) by transferring existing learned parameters, which effectively saves resources [19, 59]. 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, 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 [14, 18] that grows DNNs inefficiently to overparameter and prunes later to the backbone, our key idea is to manage the growth wisely 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 [47]. 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.

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.

Table 1: Training resource overhead on Pixel 2 XL phone.

Model & Dataset	Battery	Memory	Parameter	FLOPs	Batch Size	Epoch
LeNet on MNIST	2250mAh	256MB	431K	4.59M	100	20
MobileNet on HAR	630mAh	160MB	55.2K	1.16M	32	20
VGG-11 on CIFAR-10	3320mAh	744MB	9.24M	306.12M	100	2

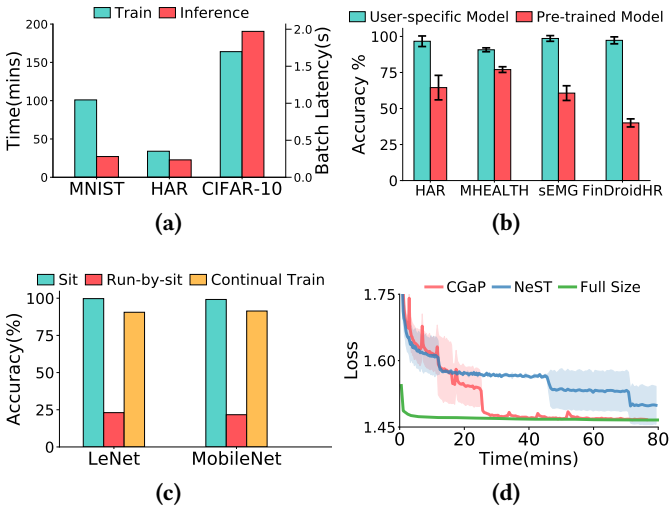


Figure 1: Preliminary results. (a) On-device training time and inference latency using the state-of-the-art DNNs; (b) Poor robustness using pre-trained model for individual; (c) Accuracy drop due to local scenario changes and recovery by continual training; (d) Slow convergence of existing CG methods.

2 Motivation

To discover the limitations of existing DL solutions for PMS applications, we conduct preliminary experiments for on-device training and inference.

Set-up We use four PMS datasets (*i.e.*, three public and one self-collected) and two image datasets, detailed in Table 3, to evaluate three DNN structures from small- to large-scale (*i.e.*, LeNet, MobileNet and VGG-11), detailed in Table 4. The study was done on an off-the-shelf Google Pixel 2XL smartphone, detailed in Table 5, which ran complete training and inference based on DL4J library.

Resource Bottleneck in On-device DL To understand the resource overhead of on-device training and inference, we train LeNet on MNIST, MobileNet on HAR and VGG-11 on CIFAR-10, respectively, with full-sized model configurations under no constant charging and memory-constrained conditions. From Fig. 1a, we observe that training is extremely slow due to heavy computation, *e.g.*, training a LeNet on MNIST takes more than 1.5 hours with a considerable 2250mAh out of 3520mAh battery drain, as shown in Table 1. Especially, the training of a large-scale VGG-11 on CIFAR-10 fails with

only two training iterations due to a complete battery drain. The memory footprints nearly reach a preset maximum threshold, resulting in an unsafe condition on the smartphone. With constant charging, the training can achieve an accuracy of over 80% by increasing iterations up to 10, but it takes more than 13.5 hours which is incredibly long. Thus, training with full-sized DNNs is very costly on resource-constrained smartphones, and may not be practical if no efficient solutions introduced.

Poor Performance of Pre-trained Model To study the performance of pre-trained model by user dependency in real-world PMS applications, we design two common training scenarios–a pre-trained model and a user-specific model based on LeNet, to train HAR, MHEALTH, sEMG, and FinDroidHR, respectively, as illustrated in Table 3. In this experiment, all users' data are manually labeled. For each dataset, we first pick the data contains ten users. We then train the user-specific model using the data from one user, and train the pre-trained model using the remaining users' data. After training, both models are evaluated on the test data for the specific user. The same process is repeated for each of the users. Fig. 1b shows that the pre-trained model results in a much lower accuracy than the user-specific model, *e.g.*, 32.16% and 37.91% accuracy drop on HAR and MHEALTH, respectively. In addition, the pre-trained models are less robust than the user-specific models when adapting to specific users. In summary, applying a pre-trained model to PMS applications has its intrinsic limitation due to user dependency.

Impact of Local Scenario Changes and Recovery Since the fact that most of PMS applications primarily work for stationary scenario (*e.g.*, indoor sitting), users may switch to movement scenario (*e.g.*, outdoor running) that the models may not cover in reality. We apply the stationary-to-movement change to quantify the impact of local scenario changes. We implement a real-world PMS application, *i.e.*, FinDroidHR [62], and collect a local dataset from two scenarios, *e.g.*, indoor sitting and outdoor running. We use both LeNet and MobileNet to train the models. Specifically, we first train the models on the indoor sitting dataset, and we evaluate the models for both scenarios when the local scenario is changed from indoor sitting to outdoor running. Fig. 1c shows a dramatic accuracy degradation due to a local scenario change, *e.g.*, 76.7% and 77.49% accuracy reduction on LeNet and MobileNet, respectively. This is probably due to new or unseen data generated as a result of local scenario change, which may have serve impact on the performance of existing models.

Continual training can be used to recover the existing model performance. Fig. 1c shows that with continual training the accuracy is well restored in both scenarios. This study shows that continual on-device training may potentially tackle the poor model performance caused by local scenario changes in PMS applications.

Slow Convergence in Existing CG To explore the training efficiency of CG on resource-constrained mobile devices, we implement two recent CG methods (*i.e.*, NeST [14] and CGaP [18]) and compare the training of a LeNet on MNIST from a tiny structure (*i.e.*, 10% of the full size) to that of a full-sized LeNet on MNIST. We run each experiment 5 times. Our result reveals that both methods suffer from a notable slow convergence issue and incur significant resource overhead. From Fig. 1d, we observe that the loss convergence time of both methods is way slower that of training a full-sized LeNet, *i.e.*, 20.68mins and 63.36mins slower using CGaP and NeST, respectively, to achieve a converged loss, and the lengthy training leads to inevitable battery drain. Due to slow convergence, both methods may fail to achieve a fair accuracy in case of low battery budget on smartphone. Although training a full-sized LeNet on MNIST incurs a large amount of memory footprints and FLOPs, it achieves a faster convergence rate to reach a fair accuracy at earlier time stage. In addition, CGaP presents a serious degraded *spiking loss* issue to slow down the training before getting converged, shown in Fig. 1d. Moreover, since NeST randomly generates the parameters of new-borns and adapts with the existing parameters in a costly trial-and-error way, the on-device loss convergence demonstrates as inferior to yield much more resource overhead. Hence, existing CG solutions may not be practical for resource-constrained training on mobile devices.

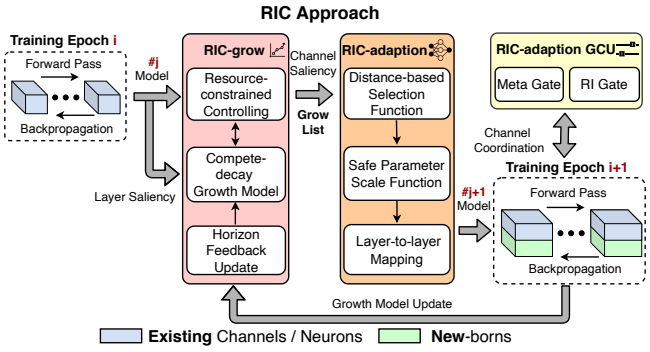


Figure 2: MDLdroidLite Workflow.

Table 2: Terminologies in MDLdroidLite

Terminology	Explanation
Model Configuration	State-of-the-art hand-crafted DNN structure configuration (*e.g.*, MobileNet) combining with a given dataset (*e.g.*, HAR) as the input of MDLdroidLite.
Training Epoch	Training iterations, running the proposed RIC approach after each iteration.
Over-parameterized Structure	Training a given dataset using a full-sized model configuration with redundancy.
Tiny Structure	Starting a structure with a tiny-sized model configuration (*e.g.*, 1% to 10% of full size).
Backbone Structure	A structure with much less resource overhead based on the given dataset as an output of MDLdroidLite.

3 MDLdroidLite Framework

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

For clarification, the terminologies involved in MDLdroidLite are summarized in Table 2.

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, Neural Architecture Search (NAS) has been recently proposed to utilize searching (*i.e.*, evolutionary algorithm based grid search [50]) and controlling (*i.e.*, Reinforcement Learning (RL) based structure control [5, 65]) approaches to achieve efficient DNN structures. However, these approaches either suffer from intractable searching space or heavy extra training of

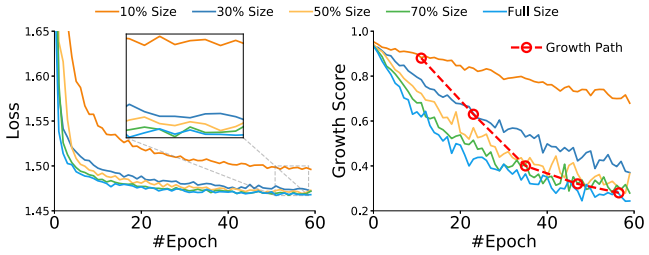


Figure 3: (a) Correlation between loss reduction and structure growth; (b) Non-linear correlation between structure growth and growth score in a layer.

control models, leading to tremendous computational cost [4], 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 [53]. 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 a typical feed-forward DNN, the underlying optimization problem is to minimize the batch loss measuring between the outputs and the given labels. For each hidden layer $l = \{l_i, i = 1, 2, ..., L+1\}$ 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, 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 addition, O^l_i denotes the i-th *channel* in a convolution layer, and the i-th *neuron* in a fully-connected layer. 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 4.

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, where the growth number is limited up to O^l for effective parameter transfer adaptation. Each action is denoted as $a \in A^l$, and current *fan_out* is denoted as $o \in O^l$. 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.

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

where $\mathcal{D}$ and ϕ denote a given dataset and its full-sized model configuration, respectively. $\mathcal{RIC}$ as a dynamic control constraint model starts from a tiny structure θ°, resulting in a resource-efficient backbone structure θ^* controlled by a set of resource-constrained π_t through grow-step N. Theoretically, RIC optimizes the structure and the training objective simultaneously. Since MDLdroidLite is an on-device structure learning framework, we focus mainly on controlling the growth of O^l in a resource-efficient way.

3.1.1 *Resource-Constrained RIC-grow*

In MDLdroidLite, the control constraint model $\mathcal{RIC}$ in Eq. 1 is defined as a Markov tuple (O, T, A, P, R), where the $O^l = 1, 2, ..., \phi^l$ and the T are discrete state variables, P^l is a stochastic state transition model, and R is a resource-constrained reward function shown in Eq. 4. 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 $t \rightarrow t+K$ at each grow-step. As a result, the layer-level control decision value function $\mathbf{V}^l(s)$ is formulated based on Bellman equation [43] as:

$$\mathbf{V}^l(s) = \max_{a \in \{\mathcal{R}, \mathcal{I}\}} [Bernoulli(p^l, s, a, s')(\mathbf{R}^l(s, a, s') + \gamma \mathbf{V}^l(s'))] \tag{2}$$

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 $Bernoulli(\cdot)$ [15] 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.

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 [41] with a $Size^l(s)$ function to dynamic calculate the growth cost of each layer representing $C^l(s)$ shown as:

$$C^l(s) = (1 - \beta) Flops^l(s) + \beta Size^l(s) \tag{3}$$

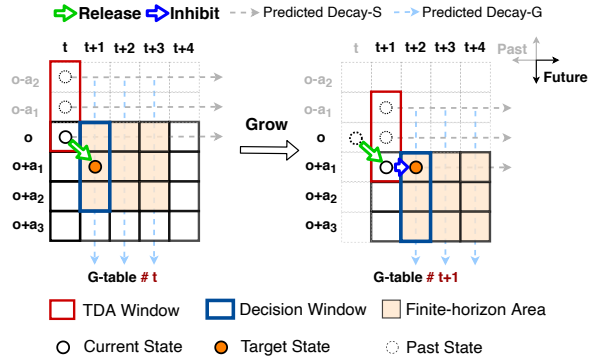


Figure 4: RIC-grow control workflow in a layer.

where β denotes a normalization coefficient between both functions. We hence propose the resource-constrained reward function $\mathrm{R}^l(s,a,s')$ formulated as a growth *state-value* function $\mathrm{G}^l(s,a,s')$ constrained by a related growth *state-cost* function $\mathrm{C}^l(s,a,s')$ formulated as:

$$\mathrm{R}^l(s,a,s') = \frac{\mathrm{G}^l(s,a,s')}{\mathrm{C}^l(s,a,s')} = \frac{\left|\mathcal{G}^l(s) - \mathcal{G}^l(s')\right|}{\frac{C^l(s')}{C^l(s)} + \mathcal{P}^l} \tag{4}$$

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. 7. Also, the growth *state-cost* is extended as a scale of the incremental cost between states using the growth cost function $C^l(s)$ in Eq. 3 with $\mathcal{P}^l$ denoted as a penalty function.

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. 4 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) [49] 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 [46], it is an efficient way to track the dynamic changes of layers and channels during the training optimization in Eq. 1. 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 [49], we combine the weights' gradients processed in Backpropagation (BP) to compute dynamic saliency metric of layers and channels, formulating as layer saliency score $\mathcal{S}^l$ and channel saliency score $\mathcal{CS}^l_i$ in Eq. 5, 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 $Cosim(\cdot)$ [13] to normalize both vectors, marking as growth score GS^l formulated as $GS^l = Cosim(\mathcal{S}^l_{1:n}, \mathcal{L}_{1:n})$.

$$\mathcal{S}^l = \sum_{i=0}^{O}(\mathcal{CS}^l_i) = \sum_{i=0}^{O}\Big(\sum_{j=0}^{I}\sum_{kh=0}^{KH}\sum_{kw=0}^{KW}\left|\frac{\partial \mathcal{L}(\mathbf{W}^\intercal \mathbf{x}+b,y)}{\partial \mathbf{W}^l}\mathbf{W}^l\right|\Big) \tag{5}$$

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

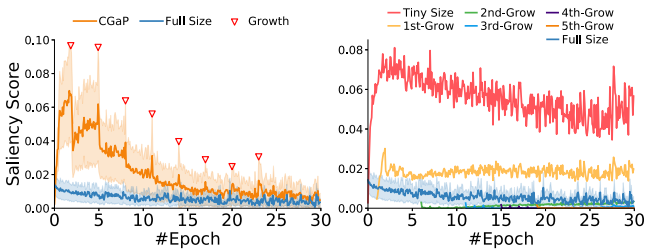
To explore the correlation between structure growth and the proposed growth score in each layer, we next present the preliminary results using the five trained LeNets on MNIST. In Fig. 3b, the growth scores of five layer structures not only demonstrate as monotonic *non-linear decay* along with loss 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) [6] to fit the proposed models. 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.

$$\arg\min_{a,b,c} \sum_{i=1}^{3}(DG^l(s) - TDA^l[i])^2 \tag{6}$$

$$\mathcal{G}^l(s) = \begin{bmatrix} DG^l_t(s) & \dots & DG^l_{t+K}(s) \end{bmatrix} \tag{7}$$

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. 2. 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.



(a) (b)

Figure 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.

3.2 RIC-adaption Pipeline

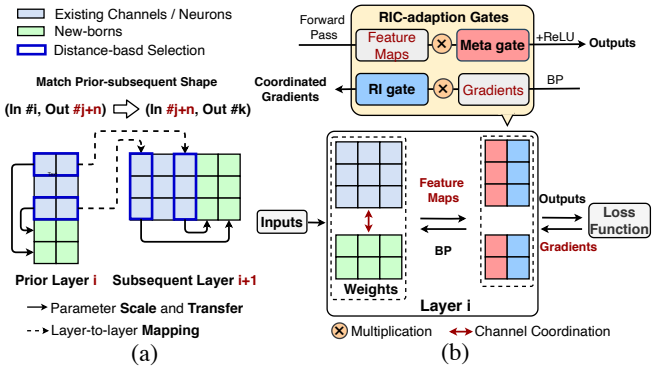
The aforementioned issue of slow training convergence may incur considerable resource overhead on mobile devices. To further understand this issue, we use the proposed channel saliency score CS_i^l in Eq. 5 to analyze the two existing CG methods and compare them with the full-sized model. Fig. 5a shows that the score variance using CGaP [18], 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 [14]. The results indicate that the new-born channels of each growth perform much less importance, leading to large score variance between channels. Thus, the existing KT adaptation approaches in CG largely increases the risk of large variance in channel-level [21], yielding unsafe parameter adaptation and slow training convergence under resource-constrained conditions.

Batch Normalization (BN) [28] 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*



(a) (b)

Figure 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.

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. Different from CGaP [18] that picks a number of channels with the highest saliency scores, we **firstly** reversely picks a channel with the lowest saliency score, and apply the cosine similarity function [13] to select the number of channels which are "close" to the lowest one. In this way, the selected group of channels presents a small variance and mean of the scores, hence transferring the selected "close" channels to new-born channels. We **secondly** apply a safe parameter scale function to preserve the current loss reduction without accuracy drop. Since the variance and mean of channel saliency scores in a full-sized model demonstrate being much smaller than that in existing CG methods shown in Fig. 5, the parameter distribution of a full-sized model is inherently more stable and safe. This is heuristically considered as "lower bound" to scale parameters down to the range. Rethinking the process from the beginning, we apply Kaiming [26] to initialize the model parameters. According to the Kaiming formulation $Std = \sqrt{\frac{3}{fan\_in}}$, the Std (*i.e.*, standard deviation) of initialized parameter distribution presents a decrease along with the increase of fan_in leading to small variance. Intuitively, since the model growth leads to the fan_in of layer increase (*i.e.*, O^l increase leads to I^{l+1} increase), we can dynamically mark each Std calculated by the increased fan_in as the "lower bound", scaling parameters to the Std to achieve small variance, which the scale function is formulated in Eq. 8.

$$Scale^l = \frac{\sqrt{I^l}\, Std(\mathbf{W}_{Ex}^l)}{\sqrt{3}} \tag{8}$$

$$\mathbf{W}_{New}^l = \frac{\mathbf{W}_{Select}^l}{Scale^l} + \mathcal{N}^l \quad \mathcal{N} \in U[-\mu, \mu], \mathbf{W}_{Select}^l \in \mathbf{W}_{Ex}^l \tag{9}$$

We hence propose to first remain the distribution of existing parameters to preserve loss reduction and avoid accuracy drop, and then scale the distribution of new-born parameters to the "lower bound" with small variance (*i.e.*, conceptually close to "inactive") to prepare for the channel coordination using GCU. Combining with the parameter selection, the parameters of new-born channels can be transferred by Eq. 9, where $\mathcal{N}$ denotes a small amount of noise. We **finally** map selected channel indexes in prior layer to subsequent layer to select the related channels, and then perform *existing-to-new* using the same way in Eq. 9 to maintain the relationship of

prior-subsequent layer in growth. In summary, Fig. 6a illustrates the workflow of these three methods.

3.2.2 *Gate-based Coordination Unit*

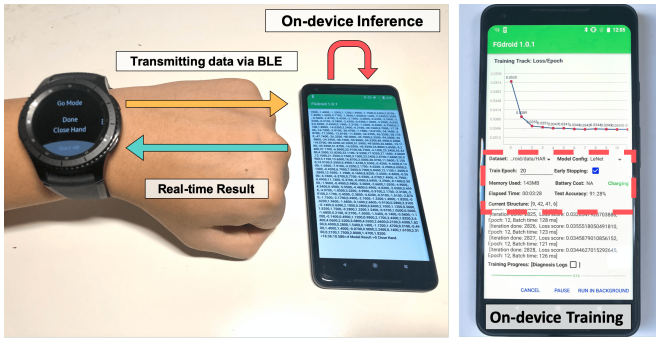
To coordinate the variance of *existing-to-new* after growth, we design GCU as a separate matrix followed up each layer shown in Fig. 6b to enable fast convergence and speed up training. In Eq. 5, the channel saliency score consists of channel weights and gradients, therefore we decompose the variance of the scores coordination into two tasks–weight variance reduction and gradient based variance coordination. GCU has two gates to handle both tasks, *i.e.*, *meta gate* as a *weight* regulator applied in forward pass, and *RI gate* as a *gradient* regulator applied in BP, resulting in minimizing the variance of *existing-to-new*.

Meta Gate Since new-born parameters are safely scaled close to "inactive" and existing parameters are remained to avoid accuracy drop in growth, the weight variance between existing and new-born channels is "temporarily" enlarged. To next minimize the weight variance, *meta gate* features as weight scale coefficients (*i.e.*, $Meta^l_{Ex}$ and $Meta^l_{New}$) to gradually coordinate between existing and new-born channels. Firstly, existing parameters will be scaled down to the same *Std* as new parameters using $Scale^l$ in Eq. 8, and both $Meta^l_{Ex}$ and $Meta^l_{New}$ are initialized as the $Scale^l$ and 1 respectively. Secondly, to functionally "remain" the existing parameters with no accuracy drop, $Meta^l_{Ex}$ is used to scale back the outputs (*i.e.*, *feature maps*) of the existing parameters, formulated as $Meta^l_{Ex}\,(\frac{\mathbf{W}^\intercal_{Ex}}{Scale^l}\mathbf{x} + b_{Ex})$. Conceptually, $Meta^l_{Ex}$ is coupled with the scaled existing parameters to not only physically reduce the weight variance between existing and new-born channels, but also functionally "remain" the outputs as identical with no impact to accuracy. Thirdly, since the $Scale^l$ is permuted to $Meta^l_{Ex}$ as a *weight* regulator, we can gradually turn down (*e.g.*, uniform decay) $Meta^l_{Ex}$ close to $Meta^l_{New}$ to minimize the weight variance, and collaborate with *RI gate* to dynamically minimize the variance of *existing-to-new* within the next training epoch.

RI Gate Since new-born parameters are safely scaled close to "inactive", the gradients of new-born channels present less contribution to the gradient descent. To gradually active the new-born parameters and coordinate with existing parameters for a minimal variance, *RI gate* works as gradient scale coefficients (*i.e.*, RI^l_{Ex} and RI^l_{New}) to dynamically "release" (*i.e.*, active) new-born's gradients and "inhibit" (*i.e.* turn down) existing gradients using a *momentum-based* variance minimization function. Theoretically, the *momentum-based* algorithm [51] improves optimization with stable and fast convergence, leading to efficient minimization for the variance of *existing-to-new* coordination. Firstly, we define the gradient scaling function as $g^{l*} = \frac{g^l}{RI^l}$, where g denotes channel gradients. Since the initial gradients of existing parameters using *meta gate* are larger than that of new-born parameters by $Scale^l$, we let RI^l_{Ex} and RI^l_{New} as $Scale^l$ and 1, respectively. Secondly, we set up the variance coordination objective function as $\min_{RI^l} \left|CS^l_{Ex} - CS^l_{New}\right|$. Thirdly, we propose a *momentum* function in Eq. 10, 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 variance coordination function using Eq. 11 within the current training epoch, resulting a stable gradient descent with fast training convergence.

$$\mathcal{V}^l(m) = \lambda\mathcal{V}^l(m-1) + (1-\lambda)(CS^l_i(m) - CS^l_i(m-1)) \tag{10}$$

$$RI^l(m+1) = RI^l(m) + \hat{\alpha}\mathcal{V}^l(m) \quad RI^l \in [\frac{1}{2\,Scale^l}, 2\,Scale^l] \tag{11}$$



(a) (b)

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

4 Evaluation

In this section, we first describe the implementation of MDLdroidLite. We then design two sets of experiments to comprehensively evaluate the performance MDLdroidLite. The first set evaluates the performance of RIC-adaption pipeline compared with existing parameter transfer adaptation baselines with the same growth rate, while the second set not only compares the performance of MDLdroidLite with existing CG and search methods, but also evaluates the resource efficiency of MDLdroidLite on several commodity smartphones using a range of datasets.

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 5. 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 [62] shown in Fig. 7.

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. 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). Practically, we scale down

Table 3: PMS & Image dataset specifications

Datasets	Type	Task	Class	Sample	Rate	#-C	#-H×W	#-KH×W	#-TR	#-TE
sEMG [33]	EMG	GR	6	14695	50Hz	8	1×100	1×12	41.3MB	10.3MB
MHEALTH [3]	IMU	HBM	9	3255	50Hz	23	1×100	1×12	41.9MB	18MB
HAR [1]	IMU	ADLs	6	10299	50Hz	9	1×128	1×14	67.8MB	27.2MB
FinDroidHR [62]	IMU&HR	GR	6	2520	100Hz	7	1×150	1×14	15.6MB	2.6MB
MNIST [37]	IMG	IR	10	70000	NA	1	28×28	5×5	15MB	2.5MB
CIFAR-10 [31]	IMG	IR	10	60000	NA	3	32×32	3×3	113MB	23MB

Table 4: Model configuration specifications

DNNs	Configuration (Type/Stride/Padding/BN)	Tiny Structure
LeNet [37]	**Conv1**/$S1$ ∈ **[1, 20]** → $Pool/S2$ → **Conv2**/$S1$ ∈ **[1, 50]** → $Pool/S2$ → **FC** ∈ **[1, 500]** → $Output$	[2-5-10-Output]
MobileNet [27]	**Conv1**/$S2$ ∈ **[1, 32]** → **ConvDW1**/$S1$ ∈ **[1, 32]** → **ConvP1**/$S1$ ∈ **[1, 64]** → **ConvDW2**/$S2$ ∈ **[1, 64]** → **ConvP2**/$S1$ ∈ **[1, 128]** → **ConvDW3**/$S1$ ∈ **[1, 128]** → **ConvP3**/$S1$ ∈ **[1, 256]** → $AvgPool$ → $Output$	[3-6-12-25-Output]
VGG-11 [52]	**Conv1**/$S1/P1/BN$ ∈ **[1, 64]** → $Pool/S2$ → **Conv2**/$S1/P1/BN$ ∈ **[1, 128]** → $Pool/S2$ → **Conv3**/$S1/P1/BN$ ∈ **[1, 256]** → **Conv4**/$S1/P1/BN$ ∈ **[1, 256]** → $Pool/S2$ → **Conv5**/$S1/P1/BN$ ∈ **[1, 512]** → **Conv6**/$S1/P1/BN$ ∈ **[1, 512]** → $Pool/S2$ → **Conv7**/$S1/P1/BN$ ∈ **[1, 512]** → **Conv8**/$S1/P1/BN$ ∈ **[1, 512]** → $Pool/S2$ → $Output$	[6-12-25-25-50-50-50-50-Output]

Table 5: 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

the layer number of a standard MobileNet to fit sensor data. We use three off-the-shelf smartphones with different resource capacities shown in Table 5 to evaluate the resource efficiency of MDLdroidLite in reality, in which the screen battery drain is excluded in the results.

Hyper-parameters We select Adam [51] 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 [2]. Practically, we set each tiny structure as 10% of full-sized convolution layer and 2% of full-sized fully-connected layer shown in Table 4. 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** [14] 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** [18] uses a saliency-based selective parameter adaptation method, transferring new-born parameters by picking up existing parameters with the highest saliency scores.
- **Net2WiderNet** [8] 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.

4.3.1 ***Channel-level Variance Minimization*** In this experiment, we evaluate the effectiveness of RIC-adaption pipeline in channel variance minimization. We also compare the channel variance of training a full-sized model as a standard "lower bound".

We investigate whether RIC-adaption pipeline performs fast and stable channel variance minimization throughout the entire model growth. Fig. 8a, b, and c report that the channel variance (*i.e.*, shadow area on the line of saliency score) using RIC-adaption pipeline is much smaller and more stable than all the baselines. Compared with the baselines at 9-epoch, RIC-adaption reduces channel variance 2.89×, 2.95×, and 5.13× on average over CGaP-select, Net2WiderNet, and NeST-bridge, respectively. Also, the results indicate that the mean of channel saliency scores using RIC-adaption pipeline is notably reduced in early growth stage to fast reach the "lower bound" at 5-epoch, which is 13-epoch, 20-epoch, 16-epoch faster than GaP-select, Net2WiderNet, and NeST-bridge, respectively. Hence, RIC-adaption pipeline outperforms the baselines with a minimized variance of *existing-to-new* in the growth.

4.3.2 ***Safe Parameter Adaptation*** We now examine whether RIC-adaption pipeline performs a safe parameter adaptation compared with the baselines. Fig. 8d plots a set of before- and after-growth accuracy comparison at 12-epoch. The results clearly show that the accuracy using RIC-adaption pipeline is effectively preserved through the growth with only 0.27% drop on average. Since the selected existing parameters are unsafely scaled in both CGaP-select and Net2WiderNet, the results reveal that the accuracy after growth is dropped by 34.77% and 22.63%, respectively. Interestingly, the accuracy in NeST-bridge presents a small drop by 8.84% after growth. It is because the new-born parameters are randomly generated in naturally inactive, which is relatively safe to remain the accuracy. Thus, RIC-adaption pipeline guarantees a safe parameter adaptation.

4.3.3 ***Fast Convergence Rate*** With the channel variance minimized, we now evaluate whether RIC-adaption pipeline can achieve a fast convergence rate over our baselines. Since the full-sized model inherently performs with fast convergence, it is marked as a "standard" convergence rate for benchmarking. Fig. 9a presents that RIC-adaption pipeline demonstrates a fastest convergence rate among the adaptation baselines, which reaches the "standard" rate at the earliest (*e.g.*, 11-epoch). In addition, since the channel variance is coordinated as stable, the loss reduction of using RIC-adaption pipeline also reports as stable with only minor *spiking loss*. Hence, RIC-adaption pipeline achieves fast convergence by minimizing the variance of *existing-to-new* during model growth.

4.3.4 ***Time-to-accuracy Efficiency*** We have shown RIC-adaption pipeline guarantees safe parameter adaptation with fast training convergence. It is now important to evaluate whether RIC-adaption pipeline can actually achieve superior time-to-accuracy efficiency (*i.e.*, less time cost when accuracy close to the "standard"). Fig. 9b presents that RIC-adaption pipeline achieves the best time-to-accuracy efficiency. The results report that the accuracy using RIC-adaption pipeline reaches the "standard" in 13.8mins, which speeds up training by 2.84×, 3.12×, and 4.88× on average over CGaP-select, Net2WiderNet, and NeST-bridge, respectively. In addition, the accuracy using RIC-adaption pipeline shows much more stable than the baselines.

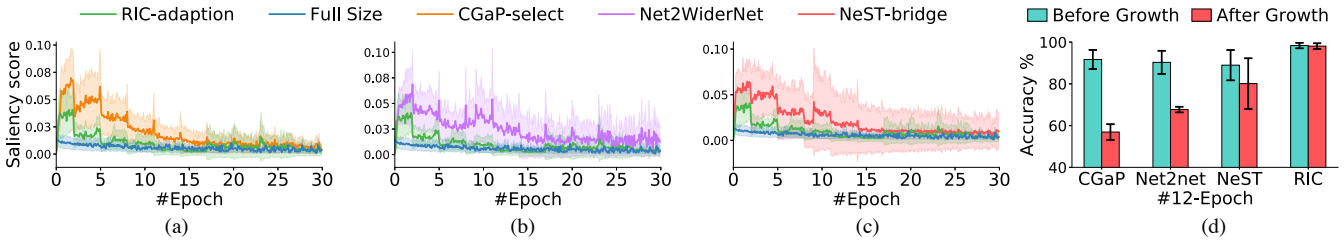


Figure 8: Channel variance minimization in growth and accuracy drop after growth.

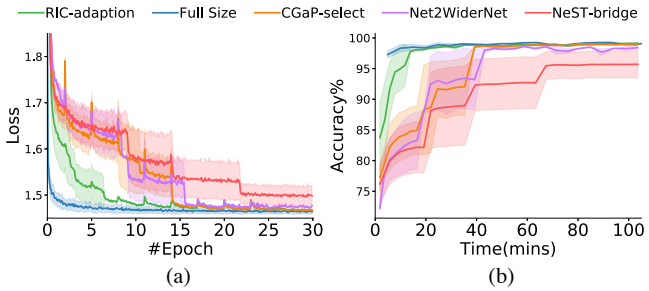


Figure 9: Time-to-accuracy efficiency of RIC-adaption.

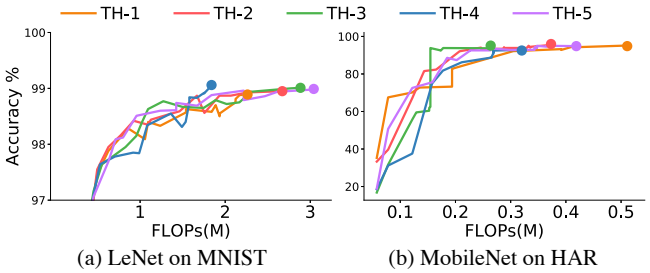


Figure 10: Accuracy-to-structure fine-tuning by time horizon.

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, on-device DL resource reduction, and on-device structure resource efficiency using a range of datasets.

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

- **NeST** [14] is a linear growth approach. The model structure continually grows with a fixed phase until reaching the full size.
- **CGaP** [18] presents an exponential growth using a fixed growth rate. It also involves a simple pre-setting resource budget in growth.
- **S-search** [23] 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.

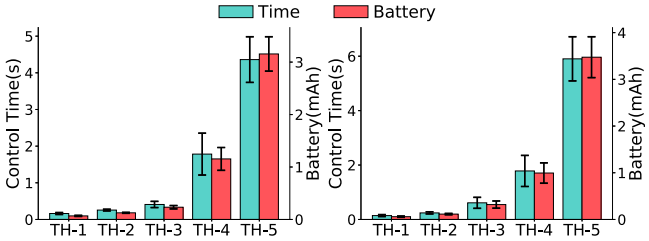
For benchmarking, we train MobileNet on HAR, LeNet on HAR, and LeNet on MNIST, respectively, on a Pixel 2XL smartphone without constant charging to compare the performance of MDLdroidLite with the baselines. Due to the lack of pruning support on off-the-shelf smartphones, the comparison of the structure 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 different sizes of time horizon (TH). Since MDLdroidLite uses the *compete-decay* growth model to dynamically optimize a recursive decision function, the size of TH is important for making an optimized growth decision. In addition, we fix the penalty regulator as default and set 20-epoch as the resource budget.

We apply five different TH sizes from TH-1 to TH-5 to evaluate accuracy-to-structure (*e.g.*, using FLOPs to represent structure) efficiency using the control decision optimization in MDLdroidLite. Fig. 10 plots the correlation between structure FLOPs and accuracy achieved in growth. Both results of TH-4 on LeNet and TH-3 on MobileNet present the optimal accuracy-to-structure, *e.g.*, using TH-4 on LeNet achieves the minimal FLOPs of 1.84M and the best accuracy of 99.06%. The results suggest that different TH sizes may lead to a better resource-efficient DNN structure. However, since our control optimization leverages a recursive-based function, the control cost may theoretically increase as the size of TH increases.

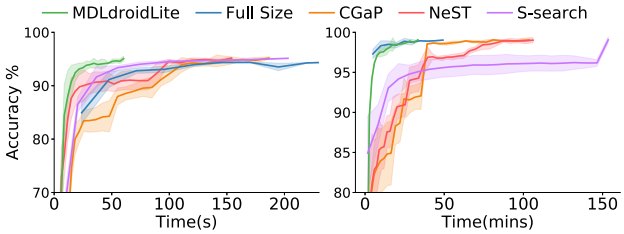
Fig. 11 reports the average resource overhead for making each control decision on device. The results indicate that with the size of TH increases, the control time and battery consumption of both models show a near-exponential increase. Especially, when using TH-5, the averaged control time is up to 4.36s for LeNet (*e.g.*, three layers growth) and 5.9s for MobileNet (*e.g.*, four layers growth). Since MDLdroidLite targets layer-level growth control, it also implies that the time cost may increase along with the number of model layers. Although a large size of TH may cause such "notable" resource overhead, it eventually depends on the accuracy-to-structure, *e.g.*, although the control time using TH-4 on LeNet is 1.78s on average, it still achieves the best accuracy-to-structure shown in Fig. 10a to save a large amount of resource on device. Practically, the size of TH may affect the optimized growth decision, and should be safely managed in a small range to assist dynamic control depends on actual resource budgets. Since different models on different datasets may need to fine-tuning with different size of TH for optimal accuracy-to-structure, we henceforth report the optimized results for simplicity.

4.4.2 ***On-device Time-to-accuracy Structure Efficiency*** We continue to examine the time-to-accuracy performance of MDLdroidLite to identify whether MDLdroidLite can transform a tiny structure to backbone structure in a resource-efficient way, comparing to our baselines. We employ LeNet on both HAR and MNIST to measure the differences of using the same model with different datasets. We



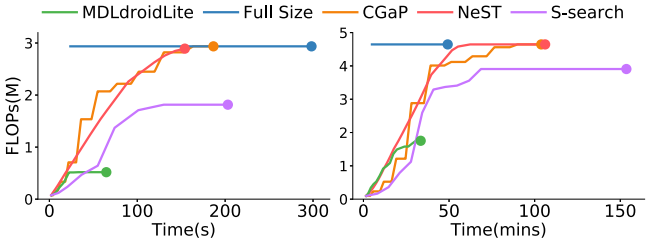
(a) LeNet on MNIST (b) MobileNet on HAR

Figure 11: Growth control resource by different time horizon.



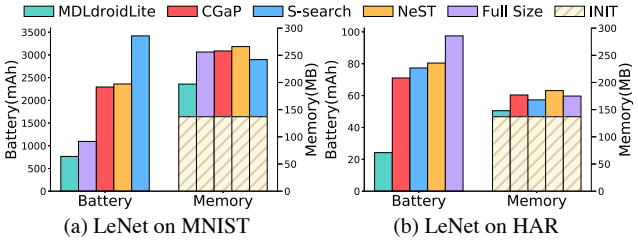
(a) LeNet on HAR (b) LeNet on MNIST

Figure 12: Time-to-accuracy efficiency comparison.



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

Figure 13: Time-to-FLOPs structure efficiency.



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

Figure 14: Battery and memory reduction comparison.

run each experiment 5 times. When 2-epoch countdown early stopping is applied, training will be stopped as soon as the accuracy is achieved.

Fig. 12a shows that MDLdroidLite achieves the best time-to-accuracy in training LeNet on HAR, *e.g.*, takes 64.80s to achieve an accuracy of 94.46% on average using a "grow" LeNet on HAR. The results demonstrate that MDLdroidLite outperforms the baselines by 2.9× and 2.4× faster over CGaP [18] and NeST [14], respectively. Also, MDLdroidLite speeds up training by 3.13× and 4.6× over S-search [23] and a full-sized model, respectively. Meanwhile, Fig. 12b reports the time-to-accuracy results of training LeNet on MNIST. Similarly, the "grow" LeNet on MNIST achieves the fastest (*e.g.*, 33.43mins) to reach the highest accuracy, which is 70.41mins, 72.68mins, 120.13mins and 15.78mins faster than CGaP [18], NeST [14], S-search [23], and a full-sized model, respectively. The results demonstrate that MDLdroidLite achieves a superior time-to-accuracy efficiency over the baselines.

4.4.3 ***On-device DL Resource Reduction*** In this experiment, we quantify the on-device resource reduction. Fig. 13 plots the dynamic increase of structure FLOPs along with elapsed time. 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. As a result, the structure of LeNet on MNIST scales down to [13-33-50-10], and that of LeNet on HAR is reduced to [10-16-33-6]. This experiment shows that, by our layer-level growth control, the structure of every single layer in MDLdroidLite can wisely "grow" to a much smaller size than that in S-search [23] with a fixed scale for all layer growth, *e.g.*, 6.1× on model parameters and 2.2× on FLOPs reduction in training LeNet on MNIST.

We further measure the specific resource reduction on memory footprints and battery consumption compared with all the baselines. Fig. 14 reports that both models using MDLdroidLite achieves the lowest memory footprints and battery consumption, *e.g.*, the battery consumption on the "grow" model can be reduced by 4× over a full-sized LeNet on HAR. Since S-search [23] is implemented to avoid intensive memory use, the single-path linear search shows a huge battery drain. For memory footprints, as the DL4J initial memory footprints (*e.g.*, loading libs or dependencies) are already roughly up to 137MB, the memory footprints for each one look tough, but the head parts of the memory bars in Fig. 14b indicate that the on-device memory footprints of "grow" LeNet on HAR are largely reduced by 3.4× over a full-sized model.

4.4.4 ***On-device Structure Resource Efficiency*** We now summarize the resource efficiency results of MDLdroidLite for three smartphones, in terms of parameters, FLOPs, memory, time, battery, and batch latency. The experiments are done using small- to large-scale model configuration (*i.e.*, LeNet, MobileNet and VGG-11) on all six datasets. For benchmarking with backbone structures, we select the state-of-the-art filter-based pruning technique [46], the full-sized model and S-search [23] as our baselines.

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. Also, the backbone models in MDLdroidLite are equivalent or outperformed to the pruned models, *e.g.*, the backbone model as [10-16-33-6] outperforms the pruned model as [12-18-50-6] on HAR. Besides, 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, 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. The complete results are omitted due to space constrained.

4.4.5 ***Real-world Performance of MDLdroidLite*** To evaluate the real-world performance of MDLdroidLite, we develop a hand gesture recognition application shown in Fig. 7a. 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 (*i.e.*, adding noise, drift, and pool) [54] to augment data on device, detailed in Table 3.

For the results in MobileNet on Pixel 2XL, the backbone model in MDLdroidLite achieves a state-of-the-art accuracy of 98.61% (*e.g.*, 99.16% in the full-sized model) with 7× less parameters, and 5.22× less FLOPs than the full-sized model. In addition, the training time, battery consumption, and batch latency of the backbone model in MDLdroidLite are efficiently reduced by 1.57×, 1.55×, and 1.49×, respectively. This case study demonstrates that MDLdroidLite can boost on-device training and inference with the resource-efficient backbone model for real-world PMS applications.

From the results we obtain, MDLdroidLite outperforms all the baselines in terms of safe parameter adaptation, fast training convergence, and on-device resource-efficiency, demonstrating the resource efficiency of MDLdroidLite for on-device DL. Leveraging the superiority in resource efficiency of MDLdroidLite, a large amount of training resources can be saved on mobile devices, *e.g.*, the battery consumption in MDLdroidLite is significantly reduced by 1527mAh and 1594mAh (*i.e.*, nearly half battery available) over CGaP [18] and NeST [14], respectively, shown in Fig. 14a. In addition, the three models (*i.e.*, MobileNet, LeNet, VGG-11) used in our evaluation represent small- to large-scale PMS applications in reality. The results show that MDLdroidLite can effectively handle different complexity levels of PMS applications and achieve the optimal model performance.

5 Discussion and Future Work

End-to-end Use Scenario MDLdroidLite aims for a universal resource-constrained approach towards MDL, hence theoretically it works for any on-device learning scenarios, *e.g.*, training from scratch, continual learning, and RL. Besides, due to the sensor data dynamics in real-world PMS applications, MDLdroidLite can incorporate with transfer learning to continually optimize model structure with new data to avoid learning forgetting [39]. Furthermore, although model structure optimization may involve a number of options (*e.g.*, depth size and input size), the focus of this paper is to manage the output size per layer for efficient on-device training and inference.

Model Structure Complexity Traditional model structure is usually made to be deep and complex for high accuracy, especially in computer vision. However, since PMS data naturally have less complexity than image data, 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. In our experience, some lightweight or shallow models may even work better 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.

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 [30] and MDLdroid [63], 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.

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 [14] 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 [18] 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 [8] proposes a standard random duplication function but with a safe compensation scale in each subsequent layer for rapid knowledge transformation. NeST [14] designs a bridging-gradient transformation function to help the neurons growth in fully-connected layers, but the new-born neurons identified as being inactive without coordination may present a weak contribution to slow down training. Differently, CGaP [18] 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.

Acknowledgments

We thank anonymous shepherd and reviewers for helpful comments. This work is supported by Australian Research Council (ARC) Discovery Project grants DP180103932 and DP190101888.

References

[1] Davide A., Alessandro G., Luca O., Xavier P., and J L Reyes-Ortiz. 2013. A Public Domain Dataset for Human Activity Recognition using Smartphones. In *ESANN'13*.

[2] Tomás Angles, Raffaello Camoriano, Alessandro Rudi, and Lorenzo Rosasco. 2016. NYTRO: When Subsampling Meets Early Stopping.

[3] Oresti B., Rafael G., Juan A. H., Miguel D., Hector P., Ignacio R., Alejandro S., and Claudia V. 2014. mHealthDroid: A Novel Framework for Agile Development of Mobile Health Applications. In *Ambient Assisted Living and Daily Activities*.

[4] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and Simplifying One-Shot Architecture Search. In *ICML '18*.

[5] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2017. Efficient Architecture Search by Network Transformation. In *AAAI '17*.

[6] T. Chai and R. R. Draxler. 2014. Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature. *Geoscientific Model Development* (2014).

[7] Kaixuan Chen, Dalin Zhang, Lina Yao, Bin Guo, Zhiwen Yu, and Yunhao Liu. 2020. Deep Learning for Sensor-based Human Activity Recognition: Overview, Challenges and Opportunities. (2020).

[8] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. 2015. Net2Net: Accelerating Learning via Knowledge Transfer. (2015).

[9] Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. 2019. A Closer Look at Few-shot Classification. In *ICLR'19*.

[10] Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. 2018. Exploring the Capabilities of Mobile Devices Supporting Deep Learning. In *HPDC '18*.

[11] Yanjiao Chen, Baolin Zheng, Zihan Zhang, Qian Wang, Chao Shen, and Qian Zhang. 2020. Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions. *ACM Comput. Surv.* (2020).

[12] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv* (2017).

[13] Luo Chunjie, Zhan jianfeng, Wang lei, and Yang Qiang. 2017. Cosine Normalization: Using Cosine Similarity Instead of Dot Product in Neural Networks. (2017).

[14] X. Dai, H. Yin, and N. K. Jha. 2019. NeST: A Neural Network Synthesis Tool Based on a Grow-and-Prune Paradigm. *TC* (2019).

[15] Lokenath Debnath and Kanadpriya Basu. 2015. A short history of probability theory and its applications. *International Journal of Mathematical Education in Science and Technology* (2015).

[16] Yunbin Deng. 2019. Deep Learning on Mobile Devices - A Review. *CoRR* (2019).

[17] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. 2020. On-Device Machine Learning: An Algorithms and Learning Theory Perspective. (2020).

[18] Xiaocong Du, Zheng Li, and Yu Cao. 2019. CGaP: Continuous Growth and Pruning for Efficient Deep Learning. *CoRR* (2019).

[19] Jiemin Fang, Yuzhu Sun, Kangjian Peng, Qian Zhang, Yuan Li, Wenyu Liu, and Xinggang Wang. 2020. Fast Neural Network Adaptation via Parameter Remapping and Architecture Search. In *ICLR '20*.

[20] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *ICLR '19*.

[21] Xavier Glorot and Y. Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track* (2010).

[22] Taesik Gong, Yeonsu Kim, Jinwoo Shin, and Sung-Ju Lee. 2019. MetaSense: Few-Shot Adaptation to Untrained Conditions in Deep Mobile Sensing. In *SenSys '19*.

[23] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2020. Single Path One-Shot Neural Architecture Search with Uniform Sampling. (2020).

[24] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *MobiSys '16*.

[25] Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated Learning for Mobile Keyboard Prediction. *CoRR* (2018).

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *ICCV '15*.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* (2017).

[28] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML'15*.

[29] Ozan Irsoy and Ethem Alpaydin. 2018. Continuously Constructive Deep Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* (2018).

[30] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated Optimization: Distributed Machine Learning for On-Device Intelligence. *CoRR* (2016).

[31] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images.

[32] J. Leon Kröger, Philip R., and T. Rahman B. 2019. Privacy Implications of Accelerometer Data: A Review of Possible Inferences. In *ICCSP '19*.

[33] Sergey L., Nadia K., Innokentiy K., Victor K., and Valeri A. M. 2018. Latent Factors Limiting the Performance of sEMG-Interfaces. *Sensors* (2018).

[34] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar. 2017. Squeezing Deep Learning into Mobile and Embedded Devices. *IEEE Pervasive Computing* (2017).

[35] Nicholas D. Lane and Petko Georgiev. 2015. Can Deep Learning Revolutionize Mobile Sensing?. In *HotMobile '15*.

[36] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. 2010. A survey of mobile phone sensing. *IEEE Communications Magazine* (2010).

[37] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[38] En Li, Zhi Zhou, and Xu Chen. 2018. Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy. In *MECOMM '18*.

[39] Z. Li and D. Hoiem. 2018. Learning without Forgetting. *TPAMI* (2018).

[40] TensorFlow Lite. 2020. Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite.

[41] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning Efficient Convolutional Networks Through Network Slimming. In *ICCV '17*.

[42] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2019. Rethinking the Value of Network Pruning. In *ICLR*.

[43] Nicolas Meuleau, Emmanuel Benazera, Ronen I. Brafman, Eric A. Hansen, and Mausam. 2009. A Heuristic Search Approach to Planning with Continuous Resources in Stochastic Domains. *J. Artif. Int. Res.* (2009).

[44] Riccardo Miotto, Fei Wang, Shuang Wang, and Xiaoqian Jiang. 2017. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics* (2017).

[45] David C Mohr, Mi Zhang, and Stephen M Schueller. 2017. Personal Sensing: Understanding Mental Health Using Ubiquitous Sensors and Machine Learning. *Annual review of clinical psychology* (2017).

[46] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *ICRL '17*.

[47] Eugenio E. Müller, Vittorio Locatelli, and Daniela Cocchi. 1999. Neuroendocrine Control of Growth Hormone Secretion. *Physiological Reviews* (1999).

[48] Greig Paul and James Irvine. 2014. Privacy Implications of Wearable Health Devices. In *SINCONF '14*.

[49] Kaveena Persand, Andrew Anderson, and David Gregg. 2020. Composition of Saliency Metrics for Channel Pruning with a Myopic Oracle. (2020).

[50] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2018. Regularized Evolution for Image Classifier Architecture Search. *CoRR* (2018).

[51] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *CoRR* (2016).

[52] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* (2014).

[53] Tri Tran, Luke Marsh, and Robert Hunjet. 2019. Reinforcement Learning with Model Predictive Control - Recent Development.

[54] Terry T. Um, Franz M. J. Pfister, Daniel Pichler, Satoshi Endo, Muriel Lang, Sandra Hirche, Urban Fietzek, and Dana Kulić. 2017. Data Augmentation of Wearable Sensor Data for Parkinson's Disease Monitoring Using Convolutional Neural Networks. In *ICMI '17*.

[55] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu. 2018. Deep Learning towards Mobile Applications. In *ICDCS'18*.

[56] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. Generalizing from a Few Examples: A Survey on Few-Shot Learning. *ACM Comput. Surv.* (2020).

[57] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *TIST* (2019).

[58] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. 2018. FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices. In *SenSys '18*.

[59] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. 2018. Lifelong Learning with Dynamically Expandable Networks. In *ICLR '18*.

[60] Zhiwen Yu, He Du, Fei Yi, Zhu Wang, and Bin Guo. 2019. Ten scientific problems in human behavior understanding. *CCF TPCI* (2019).

[61] Lei Zhang. 2019. Transfer Adaptation Learning: A Decade Survey. *CoRR* (2019).

[62] Yu Zhang, Tao Gu, Chu Luo, Vassilis Kostakos, and Aruna Seneviratne. 2018. FinDroidHR: Smartwatch Gesture Input with Optical Heartrate Monitor. *IMWUT '18* (2018).

[63] Yu Zhang, Tao Gu, and Xi Zhang. 2020. MDLdroid: a ChainSGD-reduce Approach to Mobile Deep Learning for Personal Mobile Sensing. In *IPSN '20*.

[64] Michael Zhu and Suyog Gupta. 2018. To prune, or not to prune: exploring the efficacy of pruning for model compression. (2018).

[65] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2017. Learning Transferable Architectures for Scalable Image Recognition. *CoRR* (2017).