AdaDeep: A Usage-Driven, Automated Deep Model Compression Framework for Enabling Ubiquitous Intelligent Mobiles

Sicong Liu, *Student Member, IEEE,* Junzhao Du, *Member, IEEE*, Kaiming Nan, Zimu Zhou, *Member, IEEE*, Hui Liu, *Member, IEEE*, Atlas Wang, *Member, IEEE*, Yingyan Lin, *Member, IEEE*

Abstract—Recent breakthroughs in Deep Neural Networks (DNNs) have fueled a tremendously growing demand for bringing DNN-powered intelligence into mobile platforms. While the potential of deploying DNNs on resource-constrained platforms has been demonstrated by DNN compression techniques, the current practice suffers from two limitations: 1) merely stand-alone compression schemes are investigated even though each compression technique only suit for certain types of DNN layers; and 2) mostly compression techniques are optimized for DNNs' inference accuracy, without explicitly considering other application-driven system performance (e.g., latency and energy cost) and the varying resource availability across platforms (e.g., storage and processing capability). To this end, we propose AdaDeep, a usage-driven, automated DNN compression framework for systematically exploring the desired trade-off between performance and resource constraints, from a holistic system level. Specifically, in a layer-wise manner, AdaDeep automatically selects the most suitable combination of compression techniques and the corresponding compression hyperparameters for a given DNN. Furthermore, AdaDeep also uncovers multiple novel combinations of compression techniques. Thorough evaluations on six datasets and across twelve devices demonstrate that AdaDeep can achieve up to $18.6\times$ latency reduction, $9.8\times$ energy-efficiency improvement, and $37.3\times$ storage reduction in DNNs while incurring negligible accuracy loss. Furthermore, AdaDeep also uncovers multiple novel combinations of compression techniques.

---✦---

1 INTRODUCTION

THERE is a growing trend to bring machine learning, especially deep neural networks (DNNs) powered intelligence to mobile devices [1]. Many smartphones and handheld devices are integrated with intelligent user interfaces and applications such as hand-input recognition (*e.g.,* iType [2]), speech-based assistants (*e.g.,* Siri), face recognition enabled phone-unlock (*e.g.,* FaceID). New development frameworks targeted at mobile devices (*e.g.,* TensorFlow Lite) have been launched to encourage novel DNN-based mobile applications to offload the DNN inference to local mobile & embedded devices. In addition to smartphones, DNNs are also expected to execute on-device inference on a wider range of mobile and IoT devices, such as wearables [3] (*e.g.,* Fitbit wristbands) and smart home infrastructures (*e.g.,* Amazon Echo). The diverse applications and the various mobile platforms raise a challenge for DNN developers and users: *How to adaptively generate DNNs for different resource-constrained mobile & embedded platforms to enable on-device DNN inference, while satisfying the domain-specific application performance requirements?*

- *Sicong Liu, Junzhao Du, Kaiming Nan, and Hui Liu are with the School of Computer Science and Technology, Xidian University, Xi'an, China; Zimu Zhou is with School of Information Systems, Singapore Management University, Singapore; Atlas Wang is with the Department of Computer Science and Engineering, Texas A&M University, Texas, USA; Yingyan Lin is with the Department of Electrical and Computer Engineering, Rice University, Texas, USA.*
  E-mail: {Sicong Liu, Kaiming Nan}@stu.xidian.edu.cn; {dujz, liuhui}@xidian.edu.cn; zimuzhou@smu.edu.sg; atlaswang@tamu.edu; yingyan.lin@rice.edu.
- *Corresponding Author: Junzhao Du.*

Generating DNNs for mobile mobile & embedded platforms is non-trivial because many successful DNNs are computationally intensive while mobile & embedded devices are usually limited in computation, storage and power. For example, LeNet [4], a popular DNN for digit classification, involves 60k weight and 341k multiply-accumulate operations (MACs) per image. AlexNet [5], one of the most famous DNNs for image classification, requires 61M weights and 724M MACs to process a single image. It can become prohibitive to download applications powered by those DNNs to local devices. These DNN-based applications also drain the battery easily if executed frequently.

In view of those challenges, DNN compression techniques have been widely investigated to enable the DNN deployment on mobile & embedded platforms by reducing the precision of weights and the number of operations during or after DNN training with desired accuracy. And consequently, they shrink the computation, storage, latency, and energy overhead on a target platform [6], [7]. Various categories of DNN compression techniques have been studied, including weight compression [8] [9] [10] [11], convolution decomposition [12] [13] [14], and special layer architectures [15] [16]. However, there are two major problems in existing DNN compression techniques:

- Most DNN compression techniques aim to provide an *one-for-all* solution without considering the diversity of application performance requirements and platform resource constraints. A single compression technique to reduce either model complexity or process latency may not suffice to meet complex user demands on the generated DNNs. Both the selection of DNN compression techniques and the configura-

tion of DNN compression hyperparameters should be *on-demand*, *i.e.*, adapt to the requirements and constraints on accuracy, computation, storage, latency, and energy imposed by developers and platforms.
- Most DNN compression techniques are *manually* selected and configured through experience engineering, while the design criteria remain a black-box to non-expert end developers. An *automatic* compression framework that allows user-defined criteria will benefit the development of DNN-powered mobile applications for diverse domain tasks.

This paper presents AdaDeep, a framework that automatically selects the compression techniques and the corresponding hyperparameters on a layer basis. It adapts to different user demands on application-specified performance requirements (*i.e.*, accuracy and latency) and platform-imposed resource constraints (*i.e.*, computation, storage, and energy budgets). To integrate these complex user demands into AdaDeep, we formulate the tuning of DNN compression as a constrained hyperparameter optimization problem. In particular, we define the DNN compression techniques (*e.g.,* weight compression and convolution decomposition techniques listed in § 7.2.1) as a new coarse-grained hyperparameter of DNNs. And we regard the compression hyperparameters (*e.g.,* the width multiplier and the sparsity coefficient enumerated in § 7.3.1) as the fine-grained hyperparameters of DNNs. However, it is intractable to obtain a closed-form solution, due to 1) the large numbers of the coarse-grained hyperparameter, *i.e.*, combinations of DNN compression techniques, 2) the infinite search space of the fine-grained hyperparameters, *i.e.*, compression hyperparameters, and 3) the varying platform resource constraints. Alternatively, AdaDeep applies a two-phase deep reinforcement learning (DRL) optimizer. Specifically, it involves a deep Q-network (DQN) optimizer for compression technique selection, and a deep deterministic policy gradient (DDPG) optimizer for the corresponding compression hyperparameter search. The two optimization phases are conducted interactively to provide a heuristic solution.

We implement AdaDeep with TensorFlow [17] and evaluate its performance over six different public benchmark datasets for DNNs on twelve different mobile devices. Evaluations show that AdaDeep enables a reduction of $1.7\times$ - $37.3\times$ in storage, $0.8\times$- $18.6\times$ in latency, $1.1\times$- $9.8\times$ in energy consumption, and $0.8\times$- $6.8\times$ in computational cost, with a negligible accuracy loss ($< 2.1\%$) for various datasets, tasks, and mobile platforms.

The main contributions of this work are as follows.

- To the best of our knowledge, this is the first work that integrates the selection of both compression techniques and compression hyperparameters into an automated hyperparameter tuning framework, and balances the varied user demands on performance requirements and platform constraints.
- We propose a two-phase DRL optimizer to automatically select the best combination of DNN compression techniques as well as the corresponding compression hyperparameters, in a layer-wise manner. AdaDeep extends the automation of DNN architecture tuning to DNN compression.
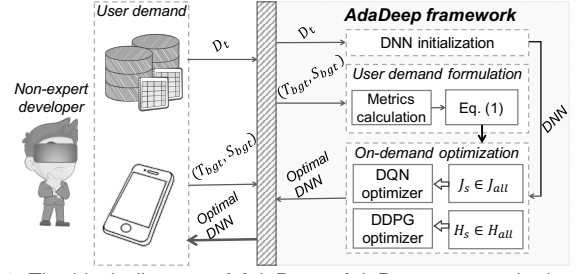

Fig. 1. The block diagram of AdaDeep. AdaDeep accepts both system performance requirements and platform resource constraints from users (*e.g.,* DNN application developers), and then automatically generates a DNN that balances these requirements and constraints.

- Experiments show that the DNNs generated by AdaDeep achieve much improved performance, as compared to existing compression techniques under various user demands (datasets, domain tasks, and target platforms). AdaDeep also uncovers some novel combinations of DNN compression techniques suitable for mobile applications.

A preliminary version of AdaDeep has been published in [18]. This work further develops [18] with the following three new contributions. First, a new DRL optimizer is proposed for fully automating the solving process of the constrained DNN compression problem (see Eq. (1)). Improving upon the one-agent based DQN for both the conv and fc layers, we develop a two-phase DRL optimizer for solving the constrained DNN compression problem in Eq. (1). In particular, in the first phase AdaDeep leverages the separate DQN agents for conv and fc layers to select the optimal combination of compression techniques in a layer-wise manner (see § 5), and then employs a DDPG optimizer in the second phase to search suitable compression hyperparameters for the selected compression techniques (refer to § 6). Second, all the experiments in [18] have been updated using the new DRL optimizer to extensively validate its effectiveness. Three, we have conducted experiments in additional model and dataset (*i.e.*, ResNet [19] on CIFAR-100 [20]) for evaluating AdaDeep in more diverse settings.

In the rest of this paper, we present AdaDeep's framework in § 2, and formulate user demands on performance and resource cost in § 3. We present the overview of the automated two-phase DRL optimizer in § 4, and elaborate the design of these two types of optimizer in both § 5 and § 6. We evaluate AdaDeep's performance in § 7, review the related work in § 8, discuss limitations and future directions in § 9, and finally conclude this work in § 10.

2 OVERVIEW

This section presents an overview of AdaDeep. From a system-level viewpoint, AdaDeep automatically generates the most suitable *compressed* DNNs that meet the performance requirements and resource constraints imposed by end developers and the target deployment platforms.

AdaDeep consists of three functional blocks: *DNN initialization*, *user demand formulation*, and *on-demand optimization* (Figure 1). The *DNN initialization* block selects an initial DNN model for the *on-demand optimization* block from a pool of state-of-the-art DNN models (§ 7.1). The *user demand formulation* block quantifies the DNN's performance and cost (§ 3), which are then input into the *on-demand*

optimization block as the optimization goals and constraints. The *on-demand optimization* block takes the initial DNN model and the optimization goals to automatically select the optimal DNN compression techniques and compression hyperparameters that maximize the system performance while satisfying cost budgets (§ 5).

Mathematically, AdaDeep aims to solve the following constrained optimization problem.

$$\underset{J_s \in J_{all}, H_s \in H_{all}}{argmax} \quad \mu_1 N(A - A_{min}) + \mu_2 N(E_{max} - E)$$
$$\text{s.t.} \quad T \le T_{bgt}, \ S \le S_{bgt}, \quad (1)$$

where A, E, T and S denote the measured accuracy, energy cost, latency and storage of a given DNN running on a specific mobile platform. User demands are expressed as a set of goals and constraints on A, E, T and S. Specifically, A_{min} and E_{max} are the minimal testing accuracy and maximal energy cost acceptable by the user. The two goals on A and E are combined by importance coefficients μ_1 and μ_2. $N(x)$ is a normalization operation, *i.e.*, $N(x) = (x - x_{min})/(x_{max} - x_{min})$. We denote T_{bgt} and S_{bgt} as the user-specified latency and storage budgets. The metrics A and S can be directly determined by the DNN architecture, while E and T are also platform-dependent. However, all of them can be tuned by applying different DNN compression techniques and compression hyperparameters. In summary, AdaDeep aims to select the best compression techniques J_s from the set of all possible combinations J_{all} and search the optimal compression hyperparameter H_s from the set of selective hyperparameter values H_{all}, according to the user-demands on performance and resource budgets. For completeness, the set J_{all} should be the permutations and combinations of discrete layer compression techniques at convolutional (conv) layers and fully-connected (fc) layers, defined as $J_{all} = Conv_{m_1}^{n_1} Fc_{m_2}^{n_2}$. Here, m_1 and m_2 are the number of optional compression techniques at conv and fc layers, respectively; n_1 and n_2 represent the number of conv and fc layers to be compressed, respectively; and the set H_{all} is a continuous real-value space.

We maximize A, minimize E while constrain S and T within the user-specified budgets, because we assume that accuracy is the most important performance metric, and the energy efficiency is in general more important than storage and latency for the power-sensitive mobile applications. AdaDeep can also integrate other optimization problem formulations.

Technically, AdaDeep faces two challenges.

- It is non-trivial to derive the runtime performance A and S, and the platform-dependent overhead E and T of a DNN. In § 3, AdaDeep proposes a systematic way to calculate these variables and associates them to the parameters of a DNN and the given platform. We apply the state-of-the-art estimation models and modify them to suite the software/hardware implementation considered in our work. Evaluations show that the proposed estimation models can achieve the same ranking as the measured one on the real-world deployment platforms.
- It is intractable to obtain a closed-form solution to the optimization problem in Eq.(1). AdaDeep employs the deep reinforcement learning (DRL) based opti-

TABLE 1
Summary of some frequently used symbols.

Symbols	Descriptions
A, E, T, C, S	DNN performance on accuracy, energy, latency, computation, storage
$T_b gt, S_b gt$	latency and storage budgets
$\mu_1, \mu_2, \mu_3, \mu_4$	Lagrange multiplier to balance performance requirements and constraints
$W_{1f}, W_{1c}, W_2, W_3, C_1$ C_2, C_3, L_1, L_2, L_3	Ten mainstream DNN compression techniques from three categories

mization process to solve it (see § 4, § 5, and § 6). Although DRL is a well-known optimization technique, its application in automated DNN architecture and hyperparameter optimization is emerging [21]. We follow this trend and apply two types of layer-wise DRL optimizer, *i.e.*, deep Q-network (DQN) and deep deterministic policy gradient (DDPG), in the context of user-demand DNN compression.

We summarize some symbols in Table 1, which are frequently used in this paper.

3 USER DEMAND FORMULATION

This section describes how we formulate the user demand metrics, including accuracy A, energy cost E, latency T and storage S, in terms of DNN parameters and platform resource constraints. Such a systematic formulation enables AdaDeep to predict the most suitable *compressed* DNNs by user needs, *before* being deployed to mobile devices.

Accuracy A. The inference accuracy is defined as:

$$A = prob(\hat{d}_i = d_i), i \in D_{mb} \quad (2)$$

where $\hat{d}_i$ and d_i denote the classifier decision and the true label, respectively, and D_{mb} stands for the sample set in the corresponding mini-batch.

Storage S. We calculate the storage of a DNN using the total number of bits associated with weights and activations [22]:

$$S = S_f + S_p = |\mathcal{X}| B_a + |\mathcal{W}| B_w \quad (3)$$

where S_f and S_p denote the storage requirement for the activations and weights, $\mathcal{X}$ and $\mathcal{W}$ are the index sets of all activations and weights in the DNN. B_a and B_w denote the precision of activations and weights, respectively. For example, $B_a = B_w = 32$ bits in TensorFlow [17].

Computational Cost C. We model the computational cost C of a DNN as the total number of multiply-accumulate (MAC) operations in the DNN. For example, for a fixed-point convolution operation, the total number of MACs is a function of the weight and activation precision as well as the size of the involved weight and activation vectors [23].

Latency T. The inference latency of a DNN executed in mobile devices strongly depends on the system architecture and memory hierarchy of the given device. We referred to the latency model in [24] which has been verified in hardware implementations. Specifically, the latency T is derived from a synchronous dataflow model, and is a function of the batch size, the storage and processing capability of the deployed device, as well as the complexity of the algorithms, *i.e.*, DNNs.

Energy Consumption E. The energy consumption of evaluating DNNs include computation cost E_c and memory access cost E_m. The former can be formulated as the total energy cost of all the MACs, *i.e.*, $E_c = \varepsilon_1 C$, where ε_1 and C denote the energy cost per MAC operation and the total number of MACs, respectively. The latter depends on the storage scheme when executing DNNs on the given mobile device. We assume a memory scheme in which all the weights and activations are stored in a Cache and DRAM memory, respectively, as such a scheme has been shown to enable fast inference execution [25] [26] [27]. Hence E can be modeled as:

$$E = E_c + E_m = \varepsilon_1 C + \varepsilon_2 S_p + \varepsilon_3 S_f \qquad (4)$$

where ε_2 and ε_3 denote the energy cost per bit when accessing the Cache and DRAM memory, respectively. To obtain the energy consumption, we refer to a energy model from a state-of-the-art hardware implementation of DNNs in [26], where the energy cost of accessing the Cache and DRAM memory normalized to that of a MAC operation is claimed to be 6 and 200, respectively. Accordingly:

$$E = \varepsilon_1 \cdot C + 6 \cdot \varepsilon_1 \cdot S_p + 200 \cdot \varepsilon_1 \cdot S_f \qquad (5)$$

where ε_1 is measured to be 52.8 pJ for mobile devices.

Summary. The user demand metrics (A, S, T and E) can be formulated with parameters of DNNs (*e.g.*, the number of C, the index sets of all activations $\mathcal{X}$ and weights $\mathcal{W}$) and platform-dependent parameters (*e.g.*, the energy cost per bit). The parameters of DNNs are tunable via DNN compression techniques and compression hyperparameters. Different mobile platforms vary in platform parameters and resource constraints. Hence it is desirable to *automatically* select appropriate compression techniques and compression hyperparameters to optimize the performance and resource cost for each application and platform.

Note that it is difficult to precisely model the platform-correlated user demand metrics, *e.g.*, E and T, since they are tightly coupled with the platform diversity. However, the ranking of the DNNs costs derived by the above estimation models is consistent with the ranking of the actual costs of these DNNs measured on the real-world deployment platforms. As will be introduced in the next section, the proposed AdaDeep framework is generic and it can easily integrate other advanced estimation models.

4   ON-DEMAND OPTIMIZATION USING DRL

We leverage deep reinforcement learning (DRL) to solve the optimization problem in Eq.(1). Specifically, two types of DRL optimizers are employed to automatically select compression techniques and the corresponding hyperparameters (*e.g.*, compression ratio, number of inserted neurons, and sparsity multiplier) on a layer basis, in the goal of maximizing performance requirements (*i.e.*, A and E) while satisfying users' demands on cost constraints (*i.e.*, S and T).

Figure 2 shows the two-phase DRL optimizer designed for the automated DNN compression problem. The first phase leverages two DQN agents for conv and fc layers to select a suitable combination of *compression techniques* in a layer-wise manner. A DDPG optimizer agent is used in the
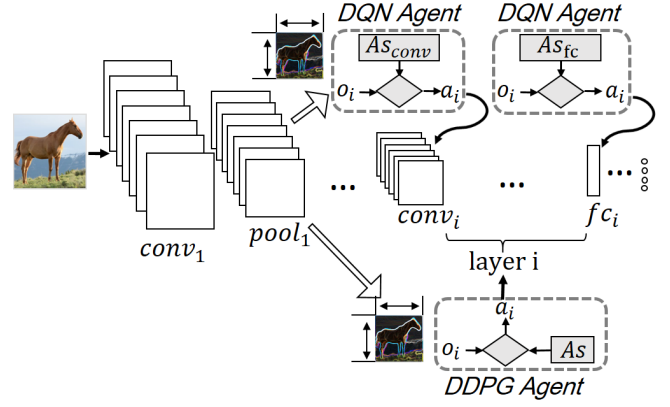

Fig. 2. The proposed two-phase DRL optimization for Eq.(1). It takes performance requirements and cost constraints as its input, automatically selects compression techniques and the corresponding compression hyperparameters in a layer-wise manner by using the DQN and DDPG agents, and outputs an optimally compressed DNN.

second phase to select *compression hyperparameter* from a continuous real-value space for the selected compression techniques at different layers. The two optimization phases are conducted interactively. During the DQN-based optimization phase, the hyperparameters at different compressed layers are fixed as the values estimated by DDPG agent. In the DDPG-based optimization phase, hyperparameter search is performed based on the compression techniques selected by DQN.

DQN and DDPG are two typical DRL methods to handle complex input, action and rewards to learn the controller agent. In the literature of DRL, a *policy* π refers to a specific mapping from *state* o to *action* a. A reward function $R(o, o', s)$ returns the gain when transitioning to state o' after taking action a in state o. Given a state o, an action a and a policy π, the action-value (a.k.a. the Q function) of the pair (o, a) under π is defined by the *action-value*, which defines the expected reward for taking action a in state o and then following policy π thereafter. The DQN agent iteratively improves its Q-function by taking actions, observing the reward and next state in the environment, and updating the estimate. Once the DQN agent is learned, the optimal policy for each state o can be decided by selecting a with the highest Q-value. As for the DDPG agent, it involves an actor-critic framework to combine the idea of DQN and Policy Gradient. Policy Gradient seeks to optimize the policy space directly, that is, an actor network learns the deterministic policy π to select action a at state o. And a value-based critic network is to evaluate potential value Q of policy (o, a) estimated by the actor network. We propose to adopt the DRL, *i.e.*, DQN and DDPG, for automated DNN compression in AdaDeep for the following reasons:

- Both DQN and DDPG agents enable automatic decision based on the dynamically detected performance and cost. And they are suited for non-linear and non-differentiable optimization.
- The DNN to be compressed and the DQN or DDPG agent can be trained jointly end-to-end [28]. Because the DQN/DDPG engent employs the neural network architecture, therefore can participant the feed-forward and back-propagation operations of the DNNs to be compressed. And the output of DQN
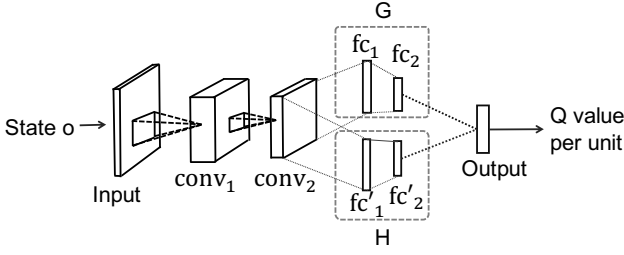
Fig. 3. The dueling DQN architecture for conv and fc layers.

and DDPG is the decision signal to control the selection of compressed techniques and hyperparameters.

- The DRL-based optimizer provide both capability and flexibility in DNN compression. Within the framework of DQN, we can easily add or delete selective compression techniques by simply adding branch sub-networks (*i.e.*, actions), and figure out the mapping function of the complex optimization problem's input and results. And DDPG can also expand or narrow the value region (action space) without affecting other components of this framework.

To apply DRL to the DNN compression problem, we need to *(i)* design the reward function to estimate the immediate reward and future reward after taking an action; *(ii)* design the definition of DRL's state o and action a in the context of DNN compression; and *(iii)* design the DRL architecture and training algorithm with tractable computation complexity. We will elaborate them in § 5 and § 6. We note that the proposed two-phase DRL optimizer, *i.e.*, DQN- and DDPG-based optimizer, are still heuristic. Hence they cannot theoretically guarantee a globally optimal solution. However, as we will show in the evaluations, the proposed optimizer outperform exhaustive or greedy approaches in terms of the performance of the compressed DNNs.

5 DQN OPTIMIZER FOR LAYER-WISE COMPRESSION TECHNIQUE SELECTION

5.1 Design of Reward Function

To define the reward function R according to the optimization problem Eq.(1), a common approach is to use the Lagrangian Multiplier [29] to convert the constrained formulation into an unconstrained one:

$$R = [\mu_1 Norm(A - A_{min}) + \mu_2 Norm(E_{max} - E)$$
$$+ \mu_3 Norm(T_{bgt} - \frac{C}{P}) + \mu_4 Norm(S_{Cache} - S_p)] \quad (6)$$

where μ_1, μ_2, μ_3 and μ_4 are the Lagrangian multipliers. It merges the objective (*e.g.*, A and E) and the constraint satisfaction (how well the T and E usages meet budgets). However, maximizing Eq.(6) rather than Eq.(1) will cause ambiguity. For example, the following two situations lead to the same objective values are thus indistinguishable: *(i)* poor accuracy and energy performance, with low latency/storage usage; and *(ii)* high accuracy and energy performance, with high latency/storage usage. Such ambiguity can easily result in a compressed DNN that exceeds the user-specified latency/storage budgets.

To avoid such ambiguity, we define two loss functions for the objective gain and the constraint satisfaction, respectively. We borrow the idea of dueling DQN [30] to separate the state-action value function and the state-action advantage function into two parallel streams (see Figure 3). The two streams share conv layers with parameters ω which learn the representations of states. And then they joint two columns to separately generate the state-action objective gain value G, with weight parameter β, and the state-action constraint satisfaction value H, with weight parameter η. The two columns are finally aggregated to output a single state-action value Q. We define a novel Q value:

$$Q(o, a; \omega, \beta, \eta) = G(o, a; \omega, \beta) + H(o, a; \omega, \eta) \quad (7)$$

The network G and H comes with their corresponding reward functions R_1 and R_2:

$$R_1 = \mu_1 Norm(A - A_{min}) - \mu_2 Norm(E_{max} - E)$$
$$R_2 = \mu_3 Norm(T_{bgt} - \frac{C}{P}) + \mu_4 Norm(S_{Cache} - S_p) \quad (8)$$

After taking an action, we observe the rewards R_1 for G and R_2 for H, and use their interaction and balance to guide the selection of compression techniques.

Algorithm 1 DQN optimizer for Layer-wise Compression Technique Selection and Combination

---

1: **procedure** DQN(D_t, $Budgets$, As_conv, As_fc)
2: 　　Initialize Λ, DNN
3: 　　Initialize DQN's predict Q with random ω, β, η
4: 　　Initialize DQN's target $\overline{Q}$ with weights $\overline{\omega}, \overline{\beta}, \overline{\eta}$
5: 　　**for** episode in range(1000) **do**
6: 　　　　**while** layer i is to be compressed **do**
7: 　　　　　　observe state o_i at layer i
8: 　　　　　　**if** i is conv layer **then**
9: 　　　　　　　　select a_i from As_conv at o_i by Q value ($\epsilon - greedy$)
10: 　　　　　　**else if** i is fc layer **then**
11: 　　　　　　　　select a_i from As_fc at o_i by Q value ($\epsilon - greedy$)
12: 　　　　　　**end if**
13: 　　　　　　layer $i + +$
14: 　　　　**end while**
15: 　　　　forward DNN to compute Reward R_{1t}, R_{2t}
16: 　　　　broadcast R_t to be the reward of all o_i
17: 　　　　store transmission (o_i, a_i, R_t, o_{i+1}) of each layer in reply memory Λ
18: 　　　　$\overline{Q}_t = R_{1t} + R_{2t} + \gamma Q(o', argmaxQ(o', a'; \omega_i, \beta_i, \eta_i); \overline{\omega}, \overline{\beta}, \overline{\eta})$
19: 　　　　perform greedy descent to update DQN's ω on loss of random mini-batches replay:
20: 　　　　　$L(\omega) = \mathbb{E}_{(o,a,R,o') \sim \Lambda}(\overline{Q}_t - Q(o_i, a_i; \omega_i, \beta_i, \eta_i))^2$
21: 　　　　every num steps reset $\overline{Q} = Q$
22: 　　**end for**
23: **end procedure**

5.2 The DQN Optimizer for Compression Technique Selection and Combination

The proposed layer-wise DQN optimizer for compression technique selection and combination is outlined in Algorithm 1. Table 2 explains the contextual definitions of the

TABLE 2
The DQN terms explained in the context of DNN compression technique selection.

DQN Terms	Contextual Meanings for DNN compression
State $o_i \sim Os$	Input feature size to DNN layer i
Action $a_i \sim As$	Selective compression techniques for DNN layer i
Reward function R	Optimization gain G & constraints satisfaction H
Q value $= \gamma \sum R$	Potential optimization gain & constraints satisfaction
Training loss function	Difference between the true Q value and the estimated Q value of DQN

DQN terms in our compression technique selection problem. For each layer i, we observe a state o_i. Two agents are employed for two types of DNN layers (*i.e.*, conv and fc layers), which respectively regard the optional compression techniques at conv and fc layers as their action space As_{conv} and As_{fc}. For each layer/state o_i, we select a random action with probability ϵ and select the action with largest Q value by $1 - \epsilon$ probability ($\epsilon = 0.001$ by default). Repeating the above operation layer by layer, we forward the entire DNN to compute a global Reward $R_t = R_{t1} + R_{t2}$, and regard it as the reward of each states o_i.

To build a DQN with weight parameters ω, β and η, we optimize the following loss function iteratively. At iteration t, we update $Q(o, a; \omega_t, \beta_t, \eta_t)$.

$$L(\omega_t) = \mathbb{E}_{(o,a,R,o') \sim \Lambda}[(\overline{Q}_i - Q(o, a; \omega_t, \beta_t, \eta_t))^2] \quad (9)$$

with the frozen Q value learned by the target network [31]:

$$\overline{Q}_t = R_{1t} + R_{2t} + \gamma Q(o', \max Q(o', a'; \omega_t, \beta_t, \eta_t); \overline{\omega}, \overline{\beta}, \overline{\eta}) \quad (10)$$

We adopt the standard DQN training techniques [30] and use the update rule of SARSA [32] with the assumption that future rewards are discounted by a factor γ [33] of the default value 0.01. And we leverage experience replay to randomly sample from a memory Λ, to increase the efficiency of DQN training.

6 THE DDPG OPTIMIZER FOR COMPRESSION HYPERPARAMETER SEARCH

We employ a DDPG optimizer to automatically search the proper compression hyperparameters for layer compression techniques from a continuous action space [34]. Its contextual definitions of state o and reward R are the same as that in the DQN optimizer (see Table 2).
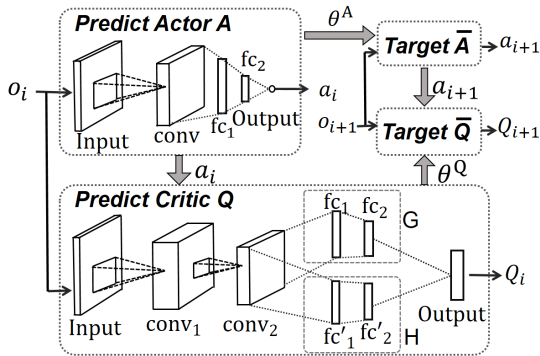

Fig. 4. The architecture of the DDPG optimizer for each layer/state o_i.

Action Space A_s for Hyperparameter Search. The compression hyperparameters considered in this work include the compression ratio in a weight pruning [9], the number of inserted neurons by weight factorization [8], [10], and

the sparsity multiplier in a convolution decomposition [12], [13]. Note that we search hyperparameters from a continuous action space for its effectiveness. To simplify implementation and reduce the training time, we transfer all of the above compression hyperparameters into a "ratio", whose value space is mapped into $[0, 1]$, so that we only need one DDPG agent to select action a from the same action space $A_s \sim [0, 1]$ for all compressed layers. We defer the transformation details from compression hyperparameters to the ratio to § 7.3.2.

Figure 4 shows the architecture of the proposed DDPG optimizer. It follows a actor-critic framework to concurrently learn the actor network A and the value-based critic network Q. The actor A gets advice from the critic Q that helps the actor decide which actions to reinforce during training. Meanwhile, the DDPG makes uses of double actor networks and critic networks to improve the stability and efficiency of training [35]. The architecture of $\overline{A}$ and $\overline{Q}$ is the same as A and Q with frozen parameters. We adopt a same dueling DQN architecture (see Figure 3) to build the critic network Q and $\overline{Q}$, which separates the reward into objective gain G and constraint satisfaction H (refer to § 5.1). And we establish the actor network, expressing the deterministic state-action function, through several conv and fc layers with parameters θ^A.

Algorithm 2 illustrates the DDPG optimizer for compression hyperparameter search. For each compressed layer i, it observes a state o_i and leverages the DDPG's predict actor network A to estimate the deterministic optimal action a_i with truncated normal distribution noise ϵ [34]. Repeating above operations, it forwards the DNN network to compute a global reward R_{1t} and R_{2t}, which is broadcast to each layer/state o_i. Then the predict critic network Q estimates the state value Q_i of the current state o_i and of the action a_i estimated by the actor A.

To train such DDPG optimizer, we optimize the actor network A at iteration t via the policy gradient function:

$$\nabla_{\theta^A} \approx \frac{1}{N} \sum_t \nabla_a Q(o_t, A(o_t); \theta^Q) \nabla_{\theta^A} A(o_t; \theta^A) \quad (11)$$

And we train the critic network Q by optimizing the loss function L from both the random reply memory and the output of the actor and the critic networks:

$$y_t = R_{1t} + R_{2t} + \gamma \overline{Q}(o_{t+1}, \overline{A}(o_{t+1}; \overline{\theta^A}))$$
$$L = \frac{1}{A} \sum_t (y_t - Q(o_t, a_t; \theta^Q))^2 \quad (12)$$

where y_t is computed by the sum of immediate reward R_{1t} and R_{2t} and the outputs of the frozen actor $\overline{A}$ and critic $\overline{Q}$.

7 EVALUATION

This section presents evaluations of AdaDeep across various mobile applications and platforms.

Algorithm 2 DDPG optimizer for Compression Hyperparameter Search

1: **procedure** DDPG(D_t, $Budgets$, As)
2:　　Initialize Λ, $DNN_{compressed}$
3:　　Initialize predict actor network $A(\theta^A)$ and target $\overline{A}(\overline{\theta^A})$
4:　　Initialize predict critic network $Q(\theta^Q)$ and target $\overline{Q}(\overline{\theta^Q})$
5:　　**for** episode in range(1000) **do**
6:　　　　**while** layer i is to be compressed **do**
7:　　　　　　observe state o_i at layer i
8:　　　　　　select a_i from As at o_i by A with noise ϵ
9:　　　　　　layer $i++$
10:　　　　**end while**
11:　　　　forward DNN network to compute Reward R_{1t}, R_{2t}
12:　　　　broadcast R_t to be the reward of all o_i
13:　　　　store transmission $(o_t, a_t, R_{1t}, R_{2t}, o_{t+1})$ of all states in Λ
14:　　　　update A using policy gradient:
15:　　　　$\nabla_{\theta^A} \approx \frac{1}{N} \sum_t \nabla_a Q(o_t, A(o_t); \theta^Q) \nabla_{\theta^A} A(o_t; \theta^A)$
16:　　　　set $y_t = R_{1t} + R_{2t} + \gamma \overline{Q}(o_{t+1}, \overline{A}(o_{t+1}; \overline{\theta^A}))$
17:　　　　update Q on loss of random mini-batches replay:
　　　　$L = \frac{1}{A} \sum_t (y_t - Q(o_t, a_t; \theta^Q))^2$
18:　　　　every num steps reset $\overline{Q} = Q$ and $\overline{A} = A$
19:　　**end for**
20: **end procedure**

7.1 Experiment Setup

We first present the settings for our evaluation.

Implementation. We implement AdaDeep with Tensor-Flow [17] in Python. The *compressed* DNNs generated by AdaDeep are then loaded into the target platforms and evaluated as Android projects executed in Java. Specifically, AdaDeep selects an initial DNN architecture from a pool of three state-of-the-art DNN models, including LeNet [4], AlexNet [5], ResNet [19], and VGG [36], according to the size of samples in D_t. For example, LeNet is selected when the sample size is smaller than 28×28, otherwise AlexNet, VGG, or ResNet is chosen. Standard training techniques, such as stochastic gradient descent (SGD) and Adam [37], are used to obtain weights for the DNNs.

Evaluation applications and DNN configurations. To evaluate AdaDeep, we consider six commonly used mobile tasks. Specifically, AdaDeep is evaluated for hand-written digit recognition (D_1: MNIST [38]), image classification (D_2: CIFAR-10 [39], D_3: CIFAR-100 [20] and D_4: ImageNet [40]), audio sensing application (D_5: UbiSound [41]), and human activity recognition (D_6: Har [42]). According to the sample size, LeNet [4] is selected as the initial DNN structure for D_1, D_2, D_5 and D_6, ResNet-56 is choosen for D_3, while AlexNet [5] and VGG-16 [36] are chosen for D_4.

Mobile platforms for evaluation. We evaluate AdaDeep on **twelve commonly used mobile and embedded platforms**, including six smartphones, two wearable devices, two development boards and two smart home devices, which are equipped with varied processors, storage, and battery capacity.

7.2 Layer Compression Technique Benchmark

In our experiment, we study the performance differences of the state-of-the-art DNN compression techniques in terms of user demand metrics, *i.e.*, accuracy A, storage S, latency T, and energy cost E. For this benchmark, we use the default compression hyperparameters (*e.g.*, k in both W_{1f} and W_2) for a fair comparison.

7.2.1 Benchmark Settings

We apply ten mainstream compression techniques from three categories, *i.e.*, weight compression (W_{1f}, W_2, W_3, W_{1c}), convolution decomposition (C_1, C_2, C_3), and special architecture layers (L_1, L_2, L_3), to a 13-layer AlexNet (input, conv$_1$, pool$_1$, conv$_2$, pool$_2$, conv$_3$, conv$_4$, conv$_5$, pool$_3$, fc$_1$, fc$_2$, fc$_3$ and output) [5] and compare their performance evaluated on CIFAR-10 dataset (D_2) [39] on a RedMi 3S smartphone. The details of them are as follows.

- W_{1f}: insert a fc layer between fc$_i$ and fc$_{(i+1)}$ layers using the singular value decomposition (SVD) based weight matrix factorization [10]. The neuron number k in the inserted layer is set as $k = m/12$, where m is the number of neurons in fc$_i$.
- W_2: insert a fc layer between fc$_i$ and fc$_{(i+1)}$ using sparse-coding, another matrix factorization method [8]. The k-basis dictionary used in W_2 is set as $k = m/6$, where m is the neuron number in fc$_i$.
- W_3: prune fc$_1$ and fc$_2$ using the magnitude based weight pruning strategy proposed in [9]. It removes unimportant weights whose magnitudes are below a threshold (*i.e.*, 0.001).
- L_3: replace the fc layers, fc$_i$ and fc$_{i+1}$, with a global average pooling layer [16]. It generates one feature map for each category in the last conv layer. The feature map is then fed into the softmax layer.
- W_{1c}: insert a conv layer between conv$_i$ and pool$_i$ using SVD based weight factorization [10]. The numbers of neurons k in the inserted layer by SVD $k = m/12$, where m is the neuron number in conv$_i$.
- C_1: decompose conv$_i$ using convolution kernel sparse decomposition [14]. It replaces a conv layer using a two-stage decomposition based on principle component analysis.
- C_2: decompose conv$_i$ with depth-wise separable convolution [13]. The width multiplier $\alpha = 0.5$.
- C_3: decompose conv$_i$ using the sparse random technique [12] and we set the sparsity coefficient $\theta = 0.75$. The technique replaces the dense connections of a small number of channels with sparse connections between a large number of channels for convolutions. Different from C_2, it randomly applies dropout across spatial dimensions at conv layers.
- L_1: replace conv$_i$ by a Fire layer [15]. A Fire layer is composed of a 1×1 conv layer and a conv layer with a mix of 1×1 and 3×3 conv filters. It decreases the sizes of input channels and filters.
- L_2: replace conv$_i$ by a micro multi-layer perceptron embedded with multiple small kernel conv layers (Mlpconv) [16]. It approximates a nonlinear function to enhance the abstraction of conv layers with small (*e.g.*, 1×1) conv filters.
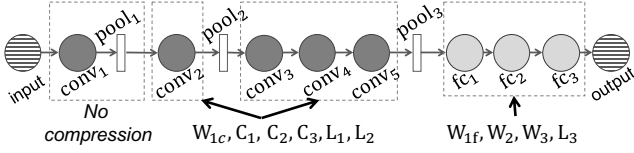
Fig. 5. An illustration of the locations that different layer compression techniques are applied to AlexNet.

The parameters (k in W_{1f}, W_{1c} and W_2, the depth multiplier α in C_2, the sparse random multiplier θ in C_3) are empirically optimized by comparing the performance on the layer where the compression technique is applied.

As shown in Figure 5, compression techniques W_{1f}, W_2, W_3 and L_3 can be applied to the fc layers (fc$_1$, fc$_2$ and fc$_3$), while W_{1c}, C_1, C_2, C_3, L_1 and L_2 are employed to compress the conv layers (conv$_2$, conv$_3$, conv$_4$ and conv$_5$). For each layer compression technique, we load the compressed DNN on smartphone to process the test data 10 times, and obtain the mean and variance of the inference performance and resource cost, considering the varied workload of the device at different test times.

7.2.2 Performance of Single Compression Technique

To illustrate the performance of different compression techniques, we compare their compressed DNNs in terms of the evaluation metrics (A, S_p, S_f, T and E), over both the initial layer that they are applied to (see Figure 6) and the entire initial network, *i.e.*, AlexNet (see Figure 7). First, we can see that overall these mainstream compression techniques are quite effective in trimming down the complexity of the initial network, with a certain accuracy loss ($0.3\% - 10.2\%$) or accuracy gain ($0.5\% - 2.4\%$). For example, the compression techniques W_3 and L_3 reduce S_p by about $150 - 203MB$, while W_{1c}, C_1, C_2, C_3, L_1 and L_2 reduce S_p to be less than $10MB$. Second, as expected, compressing the fc layers (W_{1f}, W_2, W_3, and L_3) results in a higher S_p reduction, while compressing the conv layers (W_{1c}, C_1, C_2, C_3, L_1 or L_2) lead to a larger C reduction. This is due to the common observation in DNNs that the conv layers consume dominant computational cost while the fc layers account for most of the storage cost. Third, most of the considered compression techniques affect the S_f only in the order of KB, thus we only consider S_p for the storage cost in following experiments.

Summary. The performance of different categories of compression techniques on the *same* DNN varies. Within the same category of compression techniques, the performance also differs. There is no a single compression technique that achieves the best A, S, T and E. To achieve optimal overall performance on different mobile platforms and applications, it is necessary to combine different compression techniques and tune the compression hyperparameters according to the specific usage demands.

7.2.3 Performance of Blindly Combined Compression Techniques

In this experiment, we compare the performance when blindly combining two compression techniques, tested on a RedMi 3S smartphone (Device 1) using the AlexNet model and CIFA-10 dataset (D_1). Specifically, one of the four techniques to compress the fc layers fc$_1$ and fc$_2$ (*i.e.*, W_f, W_2,

W_3 or L_3) is combined with one of the six techniques to compress the conv layer conv$_2$ (*i.e.*, W_{1c}, C_1, C_2, C_3, L_1 or L_2), leading to a total of 24 combinations. Among them, the $W_{1c} + W_2$, $L_1 + L_3$ and $L_2 + L_3$ combinations have been introduced in the prior works named SparseSep [8], SqueezeNet [15] and NIN [16], respectively.

Table 3 summarizes the results. We leverage the compressed AlexNet using the W_3 technique as a baseline. In particular, it achieves a detection accuracy of 79.9% and requires a parameter storage of $6.09MB$, an energy cost of $30.72mJ$, and a detection latency of $189ms$. First, compared with the compressed model using W_3, some combinations of compression techniques, *e.g.*, C_2+W_3 and C_2+L_3, reduce more than $48mJ$ of E, decrease T by $103ms$, and dramatically cut down S_p by more than $18MB$, while incurring only 2.4% accuracy loss. While some combinations might perform worse than a single compression technique, *e.g.*, $W_{1c}+L_3$ and C_3+W_3 incur over 28% accuracy loss. Second, the combination of L_1+W_3 achieves the best balance between system performance and resource cost.

Summary. Some combinations of two compression techniques can dramatically reduce the resource consumption of DNNs than using a single technique. Others may lead to performance degradation. Furthermore, the search space grows exponentially when combining more than two techniques. These results demonstrate the need for an automatic optimizer to select and combine compression techniques.

7.3 Performance of DRL Optimizer

This section tests the performance of the DDPG and DQN optimizer in hyperparameter search and compression technique selection, and evaluates the collaborative two optimizers.

7.3.1 Hyperparameters Learned by DDPG Optimizer

We first describe the compression hyperparameters needed for our benchmark compression techniques, and present how we transform various hyperparameters to a "ratio" so that they can share a single DDPG agent with the same action space $As \sim [0, 1]$. As in § 7.2.1, we apply ten mainstream layer compression techniques at different conv and fc layers. Note that only some of them need extra compression hyperparameters. In particular, we consider the following "ratio" hyperparameters, whose optional value can be normalized as a percentage within the real-value region $[0, 1]$:

- ratio of the number of neurons inserted between fc_i and $fc_{(i+1)}$ layer to the number of neurons at fc_i by W_{1f} technique.
- ratio of the number of neurons inserted between $conv_i$ and $conv_(i+1)$ layer to the number of neurons at $conv_i$ layer by W_{1c} technique.
- ratio of the number of k-basis dictionary inserted between fc_i and $fc_(i + 1)$ layer to the number of neurons at fc_i layer by W_2.
- ratio of the neuron number at layer i used W_3 to neuron number in original DNN layer i.
- width multiplier (a percentage) in C_2.
- sparsity coefficient (a percentage) in C_3.

Table 4 presents the performance of the DDPG optimizer on hyperparameter search and provides a referential hyperparameter setup in the compressed AlexNet CIFAR-10 (D2)
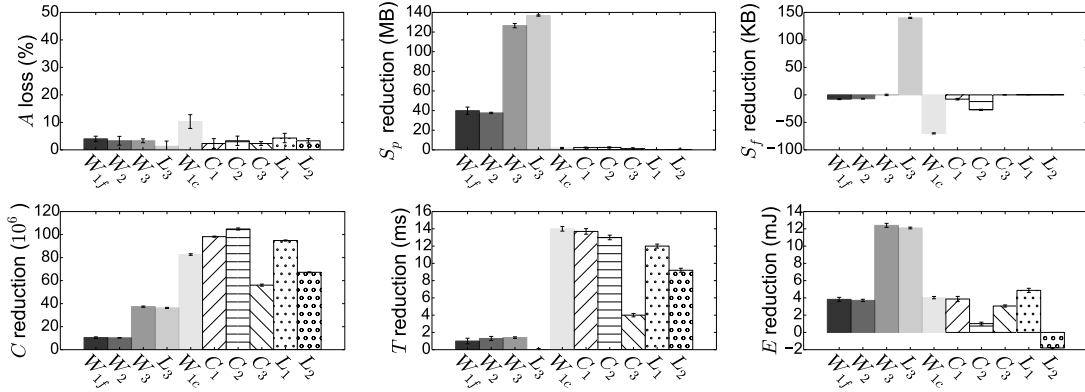
Fig. 6. Performance of different layer compression techniques minus by the *initial layer* that they are applied to, in terms of accuracy A, storage (S_p, S_f), computational cost C, latency T, and energy cost E. The Y-axis denotes the accuracy loss (%) over the initial AlexNet and the cost reduction over the *initial layer* that they are applied to.
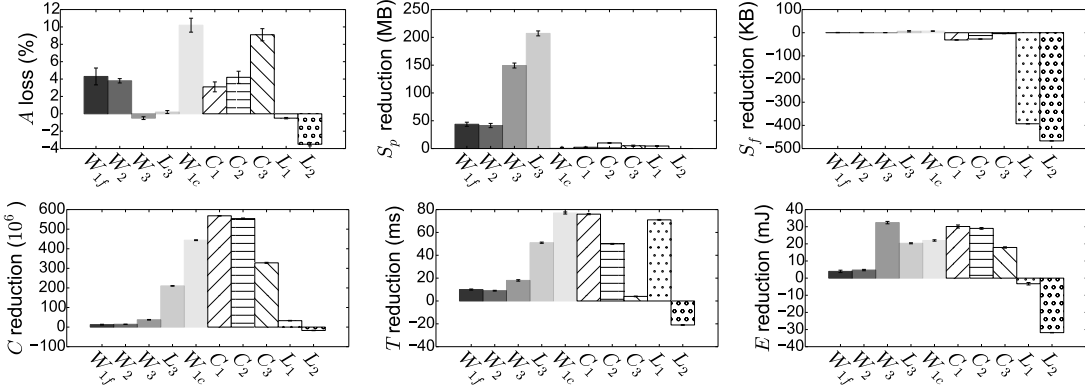

Fig. 7. Performance of different layer compression techniques normalized over the *entire* AlexNet in terms of accuracy A, storage (S_p, S_f), computational cost C, latency T, and energy cost E. Y-axis denotes the accuracy loss (%) and the cost reduction over the original *entire* AlexNet.

TABLE 3
Performance of combining two compression techniques to compress both the fc layers and the conv layers, evaluated on a RedMi 3S smartphone (Device 1) using the AlexNet model and CIFAR-10 dataset (D_1).

Compression technique	Measured accuracy & cost				Compression technique	Measured accuracy & cost			
	$A(\%)$	$S_p(MB)$	$T(ms)$	$E(mJ)$		$A(\%)$	$S_p(MB)$	$T(ms)$	$E(mJ)$
C_1+W_{1f}	74.2	15.3	180	62.8	L_1+W_{1f}	79.8	16.2	194	33.7
C_1+W_2	75.1	12.3	189	65.2	L_1+W_2	78.1	15.3	189	34.4
C_1+W_3	77.6	23.2	132	63.48	L_1+W_3	84.8	1.1	86	13.9
C_1+L_3	75.4	0.68	102	52.9	L_1+L_3 [15]	87.1	1.6	257	78.2
C_2+W_{1f}	72.4	15.3	129	33.1	L_2+W_{1f}	86.4	17.4	305	108.4
C_2+W_2	71.8	14.9	130	31.3	L_2+W_2	86.9	17.1	312	100.1
C_2+W_3	81.8	2.9	124	14.8	L_2+W_3	88.7	10.6	266	51.6
C_2+L_3	81.5	0.7	98	16.9	L_2+L_3 [16]	87.1	1.8	126	113.4
C_3+W_{1f}	59.3	16.7	236	43.4	$W_{1c}+W_{1f}$	78.4	16.1	139	36.1
C_3+W_2	57.5	15.7	210	42.7	$W_{1c}+W_2$ [8]	79.2	16.4	147	39.3
C_3+W_3	53.2	3.2	60	21.7	$W_{1c}+W_3$	61.2	2.7	143	20.8
C_3+L_3	77.3	1.4	84	26.8	$W_{1c}+L_3$	56.2	1.2	27	22.9

TABLE 4
The optimal hyperparameter at each compressed layer using different compression techniques, found by the proposed DDPG optimizer on [AlexNet, CIFAR-10(D2)].

Layer	Hyperparameters of compression technique					
	W_{1f}	W_{1c}	W_2	W_3	C_2	C_3
conv1	-	-	-	-	-	-
conv2	-	0.39	0.21	0.89	0.44	0.99
conv3	-	0.73	0.25	0.78	0.64	0.28
conv4	-	0.98	0.24	0.82	0.98	0.97
conv5	-	0.93	0.22	0.89	0.82	0.01
fc1	0.74	-	0.91	0.86	-	-
fc2	0.19	-	0.89	0.81	-	-
fc3	-	-	-	-	-	-

using different layer compression techniques. The first conv layer and final fc layer are not compressed. W_{1f}, W_{1c} and W_2 conduct weight factorization at conv and fc layers using an inserted layer with 19% to 74% neurons. W_3 prunes the weights of both conv and fc layers by the compression ratio of 21% to 89%. C_2 and C_3 decompose conv layers by the sparsity multiplier ranging from 1% to 99%.

Summary. The optimal hyperparameters of the single compression technique at different layers differ. The search space is large when searching the optimal hyperparameters for multiple layers. To balance the compression performance and the searching cost, an automated layer-wise hyperparameter search optimizer is necessary.

7.3.2 Performance Comparison of Optimizer

This experiment is to evaluate the advantage of both the proposed *DQN optimizer* and *DDPG optimizer* when searching for the optimal compression combination as well as

hyperparameters. To do so, we compress [LeNet, MNIST] and [AlexNet, CIFAR-10] using the DQN optimizer, the two-phase DRL optimizer and two baseline optimization schemes and evaluate the resulted DNNs on a RedMi 3S smartphone (Device 1). The accuracy loss (%) and the cost reduction (×) are normalized over the compressed DNNs using the W_3 technique.

- **Exhaustive optimizer**: This scheme exhaustively test the performance of all combinations of two compression techniques (similar to § 7.2.3), and select the best trade-off on the validation dataset of MNIST, *i.e.*, the one that yields the largest reward value defined by Eq. (12). The selected one is L_2+L_3, *i.e.*, *Fixed*, in both the cases of LeNet on MNIST and AlexNet on CIFAR-10. The selected combination $L_2 + L_3$ does not have tunable hyperparameters.
- **Greedy optimizer**: It loads the DNN layer by layer and selects the compression technique that has the largest reward value defined by Eq. (12), in which both μ_1 and μ_2 are set to be 0.5. Also, when T or S violate the budget T_{bgt} or S_{bgt}, the optimization terminates. The compression hyperparameters layer compression techniques are fixed by the default optimal value (similar to § 7.2.1).
- **DQN optimizer**: It compresses the DNN using the DQN optimizer as described in § 5. We set the scaling coefficients in Eq. (8) to be $\mu_1 = 0.6$ and $\mu_2 = 0.4$ considering that the battery capacity in RedMi 3S is relatively large and thus the energy consumption is of lower priority, and we set $\mu_3 = 0.5$ and $\mu_4 = 0.5$ in Eq. (8) because their corresponding constraints (*i.e.*, C and S_p) are equally important. The same as in the Greedy search within this subsection. The compression hyperparameters of layer-wise compression techniques are also set as the default optimal value (similar to § 7.2.1).
- **DDPG plus DQN optimizer**: It further leverages the DDPG optimizer to tune the compression hyperparameters of the DNN compressed by above DQN optimizer. The setup of scaling coefficients ($\mu_1 = 0.6, \mu_2 = 0.4, \mu_3 = 0.5, \mu_4 = 0.5$) is the same as that in the DQN optimizer within this subsection.

Table 5 summarizes the best performance achieved by the above four optimizers. We can see that the networks generated by DQN and DDPG optimizer achieve better overall performance in terms of storage S_p, latency T, and energy consumption E, while incurring negligible accuracy A loss (0.1% or 2.1%), compared to those generated by the other two baseline optimizers. In particular, compared with the DNN compressed by W_3, the best DNN from the Greedy optimizer only reduces S_p by $4.6\times$ and $2.2\times$ in [LeNet, MNIST] (case 1) and [AlexNet, CIFAR-10] (case 2), respectively. In contrast, the best DNN from the Exhaustive optimizer, *i.e.*, *Fixed*, can reduces S_p by $23.9\times$ and $3.5\times$, respectively. DQN optimizer cuts down $24.5\times$ and $2.5\times$ of S_p, while DDPG plus DQN optimizer achieves a maximum reduction of $28.5\times$ and $4.6\times$ on S_p in two cases. Second, the network from the proposed DQN and DDPG plus DQN optimizers are the most effective in reducing the latency ($> 2.3\times$) in both cases, while those from the two baseline optimizers may result in an increased T in some cases.

For example, the DDPG plus DQN optimizer reaches the maximum reduction of T by $3.8\times$ in case 1, and the DQN optimizer sharply reduces T by $2.5\times$ in case 2. The network from the Greedy optimizer increases T by $0.6\times$ in case 1 and the one from the Exhaustive optimizer introduces an $0.7\times$ extra T in case 2. Third, when comparing the energy cost E, *Fixed* is the least energy-efficient (reduce E by only $1.1\times$ over the DNN compressed by W_3), while those from the DQN, the DDPG plus DQN, and the Greedy optimizers achieve an reduction of $1.4\times$ to $2.8\times$, respectively. Meanwhile, the accuracy loss from the two baseline optimizers ranges from 0.1% to 2.3%, while those from DQN plus DDPG optimizer achieves the best accuracy (only a 0.2% degradation in case a and even a 4.9% gain in case 2). Finally, as for the training time, the DDPG and DQN optimizers require a shorter, or equal, or longer time compared with the exhaustive and Greedy optimizers (refer to § 7.4.2).

Summary. The proposed DDPG and DQN optimizers attain the best overall performance in both experiments. Both DDPG plus DQN and DQN optimizers outperform the other two schemes for DNN compression in terms of the storage size, latency, and energy consumption while incurring negligible accuracy in diverse recognition tasks. This is because the run-time performance metrics (A, S, T and E) and the resource cost (S and T) of the whole DNN network are systematically included in the reward value and adaptively feedback to the layer-wise compression technique selection or hyperparameter search process.

7.4 Performance of AdaDeep

In this subsection, we test the end-to-end performance of AdaDeep over six tasks and on twelve mobile platforms. Furthermore, to show the flexibility of AdaDeep in adjusting the optimization objectives based on the user demand, we show some examples of the choices on the scaling coefficients in Eq. (8).

7.4.1 AdaDeep over Different Tasks

In this experiment, AdaDeep is evaluated on all the six tasks/datasets using a RedMi 3S smartphone (Device 1). We set the scaling coefficients in Eq. (8) to be the same as those for the DRL optimizer in § 5.3.1, *i.e.*, $\mu_1 = 0.6$ and $\mu_2 = 0.4$, $\mu_3 = 0.5$ and $\mu_4 = 0.5$. In addition, we assume a Cache storage budget of 2 MB and a latency budget of 10 ms.

Performance. Table 6 compares the performance of the best DNNs generated by AdaDeep on the six tasks in terms of accuracy loss, storage S_p, computation C (total number of MACs), latency T and energy cost E, normalized over the DNNs compressed using W_3. Compared with their initial DNNs, DNNs generated by AdaDeep can achieve a reduction of $1.7\times$ - $37.3\times$ in S_p, $0.8\times$ - $3.1\times$ in C, $0.8\times$ - $18.6\times$ in T, and $1.1\times$ - $4.3\times$ in E, with a negligible accuracy loss ($< 0.4\%$) or even accuracy gain ($< 4.9\%$).

Summary. For different compressed DNNs, tasks, and datasets, the combination of compression techniques found by AdaDeep also differs. Specifically, the combination that achieves the best performance while satisfying the resource constraints is C_3+W_3 for Task 1 (on MNIST initialized using LeNet), L_1+W_3 for Task 2 (on CIFAR-10 initialized using AlexNet), L_1+W_3 for Task 3 (on CIFAR-100 initialized using

TABLE 5
Performance of the best DNN generated by the DQN, the two-phase DRL optimizer (*i.e.*, DDPG plus DQN), and two baseline optimizers are tested on a RedMi 3S smartphone using LeNet on MNIST (case 1) and AlexNet on CIFAR-10 (case 2). The accuracy loss % and the cost reduction ($\times$) are normalized over the corresponding DNN compressed using W_3.

Optimizer	Compared to the compressed LeNet on MNIST (case 1)				Compared to the compressed AlexNet on CIFAR-10 (case 2)			
	A loss	S_p	T	E	A loss	S_p	T	E
Exhaustive	0.1%	23.9$\times$	2.7$\times$	1.1$\times$	-7.2%	3.5$\times$	0.7$\times$	1.2$\times$
Greedy	2.3%	4.6$\times$	0.6$\times$	2.7$\times$	0.3%	2.2$\times$	1.2$\times$	1.9$\times$
DQN	0.4%	24.5$\times$	3.1$\times$	2.4$\times$	-2.6%	2.5$\times$	2.5$\times$	1.4$\times$
DDPG plus DQN	0.2%	28.5$\times$	3.8$\times$	2.8$\times$	-4.9%	4.6$\times$	2.3$\times$	1.8$\times$

TABLE 6
Performance of AdaDeep evaluated on different datasets using a RedMi 3S smartphone (Device 1), normalized over the corresponding DNNs compressed using W_3. **The compression techniques marked by '*' are the novel combinations that have not been proposed in related studies**.

Task	Compression techniques&hyperparameters	Compare to the DNN compressed by W_3				
		S_p	C	T	E	A loss
1.MNIST (LeNet)	$*C_3(0.96, 0.24) + W_3(0.85)$	1.8$\times$	1.5$\times$	1.8$\times$	1.3$\times$	-2.5%
2.CIFAR-10 (AlexNet)	$L_1 + W_3(0.78, 0.82)$	4.6$\times$	3.1$\times$	2.3$\times$	1.8$\times$	-4.9%
3.CIFAR-100 (ResNet-56)	$L_1 + W_3(0.48, 0.52)$	1.7$\times$	1.9$\times$	1.1$\times$	1.2$\times$	-0.1%
4.ImageNet (AlexNet)	$*L_2 + C_2(0.88, 0.81) + L_3$	18.5$\times$	2.3$\times$	3.6$\times$	1.4$\times$	-1.2%
5.ImageNet (VGG)	$*L_2 + C_1 + L_3$	37.3$\times$	2.3$\times$	18.6$\times$	4.1$\times$	0.2%
6.Ubisound (LeNet)	$*C_3(0.83, 0.31) + L_3$	3.2$\times$	1.9$\times$	1.6$\times$	1.1$\times$	0.4%
7.Har (LeNet)	$L_1 + W_3(0.76)$	2.1$\times$	0.8$\times$	0.8$\times$	1.5$\times$	-2.6%

ResNet-56), $L_2+C_2+L_3$ for Task 4 (on ImageNet initialized using AlexNet), $L_2+C_1+L_3$ for Task 5 (on ImageNet initialized using VGG), C_3+L_3 for Task 6 (on Ubisound initialized using LeNet), and L_1+W_3 for Task 7 (on Har initialized using LeNet), respectively. We can see that although the combination of compression techniques found by AdaDeep cannot always outperforms a single compression techniquein in all metrics, it achieves a better overall performance in terms of the five metrics according to the specific user demands.

7.4.2 AdaDeep over Different Mobile Devices

This experiment evaluates AdaDeep across twelve different mobile devices using LeNet and UbiSound (D_4) as the initial DNN and evaluation dataset, respectively. The performance achieved by the initial DNN is as follows: $A = 95.1\%$, $S_p = 25.2$ MB, $C = 28,324,864$, $T = 31$ ms, and $E = 4.3$ mJ.

Different devices have different resource constraints, which lead to different performance and budget demands and thus require different coefficients $\mu_1 \sim \mu_4$ in Eq. (8). Specifically, we empirically optimize $\mu1 \sim \mu4$ for different devices to be: $\mu_2 = max\{\frac{4000-E_{battery}}{4000}, 0.6\}$, $\mu_1 = 1 - \mu_2$, $\mu_4 = max\{\frac{8-S_{Cache}}{8}, 0.6\}$, and $\mu_3 = 1 - \mu_4$.

Performance. Table 7 summarizes the generated compression combinations as well as compression hyperparameters by AdaDeep and the corresponding preformance. For twelve different resource constraints, DNNs generated by AdaDeep, which are initiated with the same DNN model, can reduce parameter size by $3.4\times$ - $28.1\times$, computation cost by $1.6\times$ - $6.8\times$, latency by $1.1\times$ - $3.1\times$ and energy cost by $1.1\times$ - $9.8\times$, respectively, while incurring a negligible accuracy loss ($\leq 2.1\%$). The optimal combinations of compression techniques found by AdaDeep differ from device to device. Furthermore, AdaDeep finds some combinations that work the best for a given mobile platform yet have not been proposed by previous works (*e.g.*, C_1+W_3 for Device 1, C_2+W_3 for Devices 3, 4 and 5, C_3+W_3 for Device 11).

The training process of AdaDeep includes three intertwined phases: training the regularized DNN, re-training (such as in L_1, L_2, and L_3) or fine-tuning (such as in W_3) DNN for compression, and training the DRL (*i.e.*, DQN and DDPG) optimizer. Because the training time of the

regularized DNN is standard, we only quantify the total training time required by the DNN compression and the DRL based selection on different tasks, which is 3 hours on [MNNIST, LeNet], 10 hours on [CIFAR-10, LeNet], 6.5 hours on [CIFAR-10, AlexNet], 16 hours on [CIFAR-100, ResNet], 3.5 hours on [Ubisound, LeNet], 2 hours on [Har, LeNet], and 15 hours on [ImageNet, AlexNet], respectively, using two HP Z400 workstations with two GEFORCE GTX 1060 GPU cards.

Summary. Overall, AdaDeep can automatically select the proper combinations of compression techniques that meet diverse demands on accuracy and resource constraints within 3.5 to 15 hours. We find that the optimal compression strategy differs over tasks and across mobile devices, and there is no one-fit-all compression technique for all tasks and mobile devices. AdaDeep is able to adaptively select the best compression strategy given diverse user demands. It also uncovers some combinations of compression techniques not proposed in previous works. Also, the sensitivity of the performance metrics to different resources may vary for different choices of the scaling coefficients ($\mu_1 \sim \mu_4$).

8 RELATED WORK

Our work is closely related to the following research.

8.1 Automatic Hyperparameter Optimization

Hyperparameters of DNNs, such as the number of layers and neurons, the size of filters and the model architecture, are crucial to the inference accuracy. Common hyperparameter tuning techniques can be categorized into parallel search, such as grid search [43] and random search [44], and sequential search, *e.g.*, Bayesian optimization [45]. The grid and random search approaches search blindly and thus are usually time-consuming. Bayesian approaches [46] [47] [48] automatically optimize hyperparameters, but is slow due to the sequential operations.

Inspired by state-of-the-art automatic hyperparameter optimization techniques, compressing DNNs can be viewed as a hyperparameter tuning process. AdaDeep is the first to treat compression technique as a tunable coarse-grained hyperparameter. And it provides a systematic method to

TABLE 7
Performance of AdaDeep on different devices using the UbiSound dataset (D_4), normalized over the corresponding initial DNNs. **The compression techniques marked by '*' are the combinations that have not been proposed in related studies**.

Device	Compression techniques & hyperparameters	Compare to initial DNN				
		S_p	C	T	E	A loss
1. Xiaomi Redmi 3S	$C_1+W_3(0.81)$	12.1×	2.1×	1.6×	1.1×	0.9 %
2. Xiaomi Mi 5S	$C_2(0.41, 0.48, 0.77, 0.65)+L_3$	27.1×	3.6×	2.1×	1.2×	1.8 %
3. Xiaomi Mi 6	$*C_2(0.65, 0.68, 0.97, 0.65)+W_3(0.83)$	13.1×	5.6×	1.9×	1.6×	1.1%
4. Huawei pra-al00	$*C_2(0.63, 0.66, 0.96, 0.85)+W_3(0.81)$	12.7×	6.8×	1.4×	1.8×	1.2%
5. Samsung note5	$*C_2(0.63, 0.68, 0.94, 0.83)+W_3(0.81)$	12.8×	4.1×	1.6×	1.8×	1.2%
6. Huawei iP9	$C_1+W_3(0.82)$	13.0×	1.6×	1.6×	1.7×	0.9%
7. Sony watch SW_3	$C_2(0.73, 0.86, 0.98, 0.86)+W_2(0.89)$	6.4×	2.1×	1.5×	9.8×	1.6%
8. Huawei watchH2P	L_2+L_3	27.8×	3.6×	3.1×	8.3×	2.1%
9. firefly-rk3999	$L_1+W_3(0.83)$	13.2×	5.6×	2.6×	1.2×	1.8%
10. firefly-rk3288	$C_2(0.63, 0.68, 0.97, 0.85)+W_{1f}(0.21)$	3.4×	4.8×	1.1×	1.3×	0.7%
11. Xiaomi box 3S	$*C_3(0.89, 0.48, 0.95, 0.12)+W_3(0.84)$	14.1×	4.1×	1.4×	1.1×	1.2%
12. Huawei box	L_1+L_3	28.1×	1.6×	2.8×	1.2×	1.9%

automatically search the most suitable coarse-grained hyperparameter (*i.e.*, compression technique) and the fine-grained compression hyperparameters.

8.2 DNN Compression

The success of machine learning in mobile and IoT applications has stimulated the adoption of more powerful DNNs in mobile and embedded devices [49], [50], [51], [52], [53]. Compression is a commonly employed technique to trim down the complexity of DNNs, which can be performed by reducing the weight precision, or the number of operations, or both [6]. Various DNN compression techniques have been proposed, including weight compression [8] [9] [10], convolution decomposition [12] [13] [14], and compact architectures [15] [16]. However, existing efforts investigate a one-for-all scheme, *e.g.,* reducing DNN complexity using one compression technique, and do not consider the diversity of user demands on performance and resource cost. Our experiment results show that there is no single compression technique work well for diverse user demands.

AdaDeep enables an automatic selection of the best combination of compression techniques to balance the application-driven system performance and the platform-imposed resource constraints. Specifically, AdaDeep supports automatic selection from three categories of mainstream DNN compression techniques, and automatic configuration of compression hyperparameters.

8.3 Run-time DNN Optimization

Orthogonal to DNN compression, DNNs can also be optimized at run-time to reduce their resource utilization and unnecessary overhead on energy, latency, storage or computation. MCDNN [54] pre-evaluates a set of compressed models with different execution cost and selects one for each DNN that maximizes the accuracy given total cost constrains of multi-programmed DNNs. However, it only presents two cost reduction algorithms. LEO [55] designs a low power unit resource scheduler to maximize energy efficiency for the unique workload of different tasks on heterogenous computation resources. DeepX [10] designs a set of resource control algorithms to decompose DNNs into different unit-blocks for efficient execution on heterogeneous computation resources. EIE [56] is a dedicated accelerator to execute sparse NN.

The above run-time optimization techniques can be applied on top of the compressed DNN generated by AdaDeep

to further improve the efficiency of DNN execution on mobile devices. For example, the current version of AdaDeep only leverages the CPU on mobile platforms for DNN execution. The scheduler proposed in [55] and [10] can be combined when extending AdaDeep to mobile platforms with heterogeneous resources. With proper hardware support, the sparse NN output by AdaDeep can also be executed faster using the accelerator in [56].

8.4 Automatic Control Techniques using DRL

Deep reinforcement learning (DRL) is widely applied in automatic-play games to learn actions at different states that maximize a given reward function [33]. For example, Mnih *et al.* [33] propose to learn control policies from complex sensory inputs using a deep Q-network (DQN). Liu *et al.* [28] leverage DQN to dynamically select parts of a NN to execute according to different input resolution so as to improve computational efficiency of multi-objective optimization problems. Achiam *et al.* [57] solve the constrained optimization problem with DRL by replacing the objective and constraints with approximate surrogate, *i.e.,* lower bound on policy divergence. However, the required operation of inverting the divergence matrix is in general impractically expensive. Bello *et al.* [58] present a framework to tackle the combinatorial optimization of sequential problems with DRL and recurrent DNN. David *et al.* [35] apply deterministic policy gradient to choose action from continuous action space.

To the best of our knowledge, AdaDeep is the first work to leverage DQN and DDPG for DNN compression technique selection as well as compression hyperparameter optimization, considering both application-driven system performance and platform constraints.

8.5 Automated DNN Architecture Optimization

An emerging topic for the deep learning community is to automate the engineering process of deep model architectures: using recurrent networks and reinforcement learning to generate the model descriptions of deep models [21], or by transferring architectural building blocks to construct scalable architectures on larger datasets [59]. Those methods are purely data-driven, with deep architectures composed with the goal to maximize the expected accuracy on a validation set. Lately, a handful of exploratory works have emerged to correlate the model composition with domain knowledge. For example, Andreas *et al.* [60] constructed and learned modular networks, which composed collections

of jointly-trained neural "modules" into deep networks for question answering, to simultaneously exploit the representational capacity of deep networks and the compositional linguistic structure of questions. Devin *et al.* [61] proposed a similar modular network by decomposing robotic policies into task-specific and robot-specific modules, to facilitate multi-task and multi-robot policy transfer. However, none of those previous efforts have correlated their efforts with DNN compression and energy efficiency.

8.6 AutoML for DNN Compression

Automated machine learning (AutoML) aims at providing effective system to free non-experts from selecting the right algorithm or hyperparameter at hand. AutoML systems like Auto-WEKA [62] and Auto-skelearn [63] leverage Bayesian optimization method to search the best classifier given the datasets. Auto-Net [64] leverages the tree-based Bayesian method to tune DNN hyperparameters without human intervention. AMC [34] comes up with a continuous compression ratio control strategy with DDPG agent to find the redundancy. AdaNet [65] adaptively learn both the DNN structure and its weights. NetAdapt [34] is an automatic tool to gradually reduce the number of filters of a DNN for resource consumption reduction.

So far AutoML systems do not yet simultaneously support DNN architecture and hyperparameter optimization for DNN compression. AdaDeep extends the automation of DNN architecture selection and hyperparameter optimization to include DNN compression, that considerably trading off among both user-defined requirements and platform-imposed constraints.

9 DISCUSSIONS

In this section, we point out several limitations of AdaDeep in this work for future research.

Finer-grained Compression Optimization. AdaDeep is built upon a set of predefined compression techniques. It searches for the best combination of compression techniques and the corresponding hyperparameters for each layer via reinforcement learning. Hence the optimization space is constrained by the granularity of the predefined compression techniques. Integration of other categories of compression techniques will expand the action space and potentially result in better compressed DNNs. Randomization techniques such as layer skipping or re-ordering may also facilitate finer-grained model compression optimization.

Extensions to Other Layer Types and Processors. In this work, we mainly design AdaDeep based on compression techniques for dense and convolutional layers. Since compression techniques for recurrent layers are gaining increasing attention [66], nne next step is to extend our optimization framework to also support recurrent layers. Furthermore, although our evaluations include experiments with twelve different mobile devices, we mainly evaluate the performance of different methods on devices with merely CPUs. With the increasing popularity of GPUs installed on off-the-shelf mobile devices, it remains an interesting question how AdaDeep performs on those mobile devices.

DRL Optimization Speedup. Despite its effectiveness in optimizing the compression technique combination and

hyperparameters, DRL can consume considerable time due to the large search space and the sophisticated optimization procedure. We anticipate that conditional search by setting search conditions based on activation and previous prediction at different layers/compression techniques will accelerate the optimization process.

10 CONCLUSION

This paper presents AdaDeep, a usage-driven and automated DNN compression and optimization framework that selects the most suitable combination of compression techniques and the corresponding compression hyperparameters to balance diverse user-specified performance goals and device-imposed resource constraints. We systematically formulate user demands on performance requirements (*e.g.,* accuracy, latency) and resource constraints (*e.g.,* storage and energy budgets) into a unified optimization problem. And we leverage two types of DRL optimizors, *i.e.,* a DQN based optimizer and a DDPG based optimizer, to effectively find the feasible combination of compression techniques and the corresponding compression hyperparameters in a layer-wise manner. Evaluations on six widely used tasks and twelve different devices show that there is no one-fit-all compression technique or hyperparameter configuration that meets the diverse user demands. AdaDeep also figures out some novel combinations of compression techniques unexplored in previous work. AdaDeep is the first to model DNN compression as an automated hyperparameter tuning process, that automates the selection of the coarse-grained hyperparameters (*e.g.,* compression techniques) and the fine-grained compression hyperparameters (*e.g.,* compression ratio and sparsity coefficient) of DNNs.

REFERENCES

[1] Y. Han, X. Wang, V. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *arXiv preprint arXiv:1907.08349*, 2019.
[2] Z. Li, M. Li, P. Mohapatra, J. Han, and S. Chen, "itype: Using eye gaze to enhance typing privacy," in *Proceedings of INFOCOM*, 2017.
[3] Y. Liu and Z. Li, "itype: Using eye gaze to enhance typing privacy," in *Proceedings of INFOCOM*, 2018.
[4] Y. LeCun, "Lenet," 2017, https://goo.gl/APBzd5.
[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of NIPS*, 2012.
[6] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *arXiv preprint arXiv:1703.09039*, 2017.

[7] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *arXiv preprint arXiv:1905.10083*, 2019.

[8] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of SenSys*, 2016.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proceedings of ICLR*, 2016.

[10] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of IPSN*, 2016.

[11] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 5363–5372. [Online]. Available: http://proceedings.mlr.press/v80/wu18h.html

[12] S. Changpinyo, M. Sandler, and A. Zhmoginov, "The power of sparsity in convolutional neural networks," *arXiv preprint arXiv:1702.06257*, 2017.

[13] 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," *arXiv preprint arXiv:1704.04861*, 2017.

[14] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of CVPR*, 2015.

[15] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[16] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proceedings of ICLR*, 2014.

[17] Google, "Tensorflow," 2017, https://goo.gl/j7HAZJ.

[18] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of MobiSys*. ACM, 2018, pp. 389–400.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of CVPR*, 2016, pp. 770–778.

[20] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[21] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[22] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 3007–3016. [Online]. Available: http://proceedings.mlr.press/v70/sakr17a.html

[23] Y. Lin, S. Zhang, and N. R. Shanbhag, "Variation-tolerant architectures for convolutional neural networks in the near threshold voltage regime," in *Proceedings of SiPS*, 2016.

[24] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for fpga-based convolutional neural networks," in *Proceedings of FPL*, 2017, pp. 1–8.

[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of ISCA*, 2016.

[26] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of CVPR*, 2017.

[27] M. Xu, F. Qian, and S. Pushp, "Enabling cooperative inference of deep learning on wearables and smartphones," *arXiv preprint arXiv:1712.03073*, 2017.

[28] L. Liu and J. Deng, "Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution," in *Proceedings of AAAI*, 2018.

[29] K. Ito and K. Kunisch, *Lagrange multiplier approach to variational problems and applications*. SIAM, 2008.

[30] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2016.

[31] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *Proceedings of AAAI*, 2016.

[32] H. Van Seijen, H. Van Hasselt, S. Whiteson, and M. Wiering, "A theoretical and empirical analysis of expected sarsa," in *Proceedings of ADPRL*, 2009.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *Proceedings of NIPS Workshops*, 2013.

[34] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of ECCV*, 2018, pp. 784–800.

[35] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of ICLR*, 2015.

[37] D. Kingma and J. Ba, "Adam:a method for stochastic optimization," in *Proceedings of ICLR*, 2015.

[38] Y. LeCun, "The mnist database of handwritten digits," https://goo.gl/t6gTEy, 1998.

[39] A. Krizhevsky, N. Vinod, and H. Geoffrey, "The cifar-10 dataset," https://goo.gl/hXmru5, 2014.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of CVPR*, 2009.

[41] L. Sicong, Z. Zimu, D. Junzhao, S. Longfei, J. Han, and X. Wang, "Ubiear: Bringing location-independent sound awareness to the hard-of-hearing people with smartphones," *Journal of IMWUT*, 2017.

[42] UCI, "Dataset for human activity recognition," https://goo.gl/m5bRo1, 2017.

[43] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proceedings of NIPS*, 2011.

[44] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, 2012.

[45] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of NIPS*, 2012.

[46] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable bayesian optimization using deep neural networks," in *Proceedings of ICML*, 2015, pp. 2171–2180.

[47] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves." in *Proceedings of IJCAI*, 2015.

[48] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust bayesian neural networks," in *Proceedings of NIPS*, 2016.

[49] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of UbiComp*, 2015, pp. 283–294.

[50] X. Zheng, J. Wang, L. Shangguan, Z. Zhou, and Y. Liu, "Design and implementation of a csi-based ubiquitous smoking detection system," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3781–3793, 2017.

[51] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "Predictivenet: An energy-efficient convolutional neural network via zero prediction," in *Proceedings of ISCAS*, 2017.

[52] Y. Wang, T. Nguyen, Y. Zhao, Z. Wang, Y. Lin, and R. Baraniuk, "Energynet: Energy-efficient dynamic inference," in *Thirty-second Conference on Neural Information Processing Systems (NIPS 2018) Workshop*, 2018.

[53] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 5363–5372.

[54] 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 *Proceedings of MobiSys*, 2016.

[55] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proceedings of MobiCom*, 2016.

[56] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of ISCA*, 2016.

[57] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," *Proceedings of ICML*, 2017.

[58] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2017.

[59] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv preprint arXiv:1707.07012*, 2017.

[60] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Deep compositional question answering with neural module networks," *arXiv preprint arXiv:1511.02799*, vol. 2, 2015.

[61] C. Devin, A. Gupta, T. Darrell, P. Abbeel, and S. Levine, "Learning modular neural network policies for multi-task and multi-robot transfer," in *Proceedings of ICRA*, 2017.

[62] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka," *The Journal of Machine Learning Research*, 2017.

[63] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 2962–2970.

[64] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter, "Towards automatically-tuned neural networks," in *Workshop on Automatic Machine Learning*, 2016.

[65] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang, "Adanet: Adaptive structural learning of artificial neural networks," in *Proceedings of ICML*. JMLR. org, 2017.

[66] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 4451–4458.
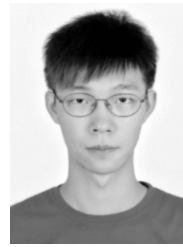
Sicong Liu received the BS and MS degrees from Xidian University in 2013 and 2016, respectively. From 2017 to 2018, she was a visiting scholar at Rice University. She is currently a Ph.D. student with the School of Computer Science and Technology, Xidian University. Her research interests include mobile computing system, mobile and embedded deep learning design, and automated deep model optimization.

Junzhao Du received the BS, MS, and Ph.D. degrees from School of Computer Science at Xidian University in 1997, 2000, and 2008, respectively. He is currently a professor and PhD advisor at Xidian University. His research interests include mobile computing, cloud computing, and IoT systems. He is the member of ACM/IEEE, senior member of CCF, and vice secretary of ACM Xian Chapter.
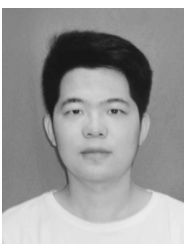
Kaiming Nan received the BS degree in School of Software from Xidian University in 2017. He is currently a MS student at Xidian University. His research interests include mobile computing, mobile and embedded data collection, and energy consumption optimization in mobile deep learning.

Zimu Zhou is currently an Assistant Professor of Information Systems at Singapore Management University. He received the B.E. degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2011, and the Ph.D. degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, in 2015. From 2016 to 2019, he was a Post-Doctoral Researcher at the Computer Engineering and Networks Laboratory, ETH Zurich, Zurich, Switzerland. His research interests include mobile and ubiquitous computing.

Hui Liu received the BS, MS, and PhD degrees from School of Computer Science at Xidian University in 1998, 2003, and 2011, respectively. She is currently an associate professor at Xidian University. Her research interests includes big data analysis, task scheduling, and mobile computing. She is the member of ACC, IEEE, and CCF.
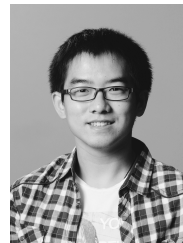
Zhangyang (Atlas) Wang is an Assistant Professor of Computer Science and Engineering (CSE), at the Texas A&M University (TAMU). During 2012-2016, he was a Ph.D. student in the Electrical and Computer Engineering (ECE) Department, at the University of Illinois at Urbana-Champaign (UIUC). Prior to that, he obtained the B.E. degree at the University of Science and Technology of China (USTC), in 2012. He was a former research intern with Microsoft Research (summer 2015), Adobe Research (summer 2014), and US Army Research Lab (summer 2013). Dr. Wang's research has been addressing machine learning, computer vision, as well as their interdisciplinary applications, using advanced feature learning and optimization techniques. He has co-authored over 80 papers, and has published 2 books and 1 invited chapter. He has been granted 3 patents, and has received over 20 research awards and scholarships.

Yingyan Lin is an Assistant Professor in the Department of Electrical and Computer Engineering (ECE) at Rice University. She received a Ph.D. degree in ECE from the University of Illinois at Urbana-Champaign in 2017. From 2007 to 2009, she worked at China's National Research Center for Integrated Circuits in Wuhan, where she designed three analog and mixed-signal circuit IPs for large panel displays that were acquired by TOSHIBA Microelectronics Corporation in Japan. She was the recipient of a Best Student Paper Award at the 2016 IEEE International Workshop on Signal Processing Systems (SiPS 2016), the 2016 Robert T. Chien Memorial Award at UIUC for Excellence in Research, and was selected as a Rising Star in EECS by the 2017 Academic Career Workshop for Women at Stanford University. Her research focuses on embedded machine learning, which is to explore algorithm-, architecture-, and circuit-level techniques for enabling powerful yet power hungry machine learning systems to be deployed in resource-constrained platforms.