On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework

Sicong Liu
Xidian University
scliu007@gmail.com

Yingyan Lin
Rice University
yingyan.lin@rice.edu

Zimu Zhou
ETH Zurich
zzhou@tik.ee.ethz.ch

Kaiming Nan
Xidian University
nankaiming@stu.xidian.edu.cn

Hui Liu
Xidian University
liuhui@xidian.edu.cn

Junzhao Du
Xidian University
dujz@xidian.edu.cn

ABSTRACT

Recent research has demonstrated the potential of deploying deep neural networks (DNNs) on resource-constrained mobile platforms by trimming down the network complexity using different compression techniques. The current practice only investigate stand-alone compression schemes even though each compression technique may be well suited only for certain types of DNN layers. Also, these compression techniques are optimized merely for the inference accuracy of DNNs, without explicitly considering other application-driven system performance (e.g., latency and energy cost) and the varying resource availabilities across platforms (e.g., storage and processing capability). In this paper, we explore the desirable trade-off between performance and resource constraints by user-specified needs, from a holistic system-level viewpoint. Specifically, we develop a usage-driven selection framework, referred to as AdaDeep, to automatically select a combination of compression techniques for a given DNN, that will lead to an optimal balance between user-specified performance goals and resource constraints. With an extensive evaluation on five public datasets and across twelve mobile devices, experimental results show that AdaDeep enables up to 9.8x latency reduction, 4.3x energy efficiency improvement, and 38x storage reduction in DNNs while incurring negligible accuracy loss. AdaDeep also uncovers multiple effective combinations of compression techniques unexplored in existing literature.

CCS CONCEPTS

• Human-centered computing → Ubiquitous and mobile computing systems and tools;

KEYWORDS

deep learning; model compression; deep reinforcement learning

ACM Reference Format:


1 INTRODUCTION

There is a growing trend to bring machine learning, especially deep neural networks (DNNs) powered intelligence to mobile devices. Many smartphones and handheld devices are integrated with intelligent user interfaces and applications such as hand-input recognition (e.g. iType[30]), speech-based assistants (e.g. Siri), face recognition enabled phone-unlock (e.g. FaceID). New development frameworks targeted at mobile devices have also been launched (e.g. TensorFlow Lite) to encourage novel DNN-based mobile applications. In addition to smartphones, DNNs are also expected to execute locally on a wider range of mobile and IoT devices, such as wearables[34] (e.g. Fitbit wristbands) and smart home infrastructure (e.g. Amazon Echo). The diverse applications and the various mobile platforms raise a challenge for DNN developers and users: How to generate the DNN that meet the application performance requirements on the target resource-constrained mobile platforms?

Generating DNNs for mobile platforms is non-trivial because many successful DNNs are computationally intensive while mobile devices are usually limited in computation, storage and power. For example, LeNet [29], a popular DNN for digit classification, has 60k weights and 341k multiply-accumulate operations (MACs) per image. AlexNet [23], 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 has been widely investigated to reduce the precision of weights and the number of operations during or after DNN training so as to shrink the computation and the size of the original DNNs while remaining the desired accuracy [41]. Various DNN compression techniques have been proposed, including weight compression [6] [16] [25], convolution decomposition [7] [18] [32], and special layer architectures [20] [31]. However, there are two major problems in existing DNN compression techniques:

• Most DNN compression techniques aim to provide a one-for-all solution without considering the application performance requirements and the platform resource constraints. A single
This section presents an overview of varying resource constraints, it is intractable to obtain a closed-level viewpoint, AdaDeep.

In this paper, we propose AdaDeep, a framework that automatically selects a combination of DNN compression techniques to adapt to user-specified performance requirements and platform-imposed resource constraints on accuracy, latency, storage, and energy consumption. We define DNN compression techniques as a new high-level hyper-parameter of DNNs. To model various user demands, we formulate the tuning of DNN compression as a constrained hyperparameter optimization problem. Due to the large numbers of combinations of DNN compression techniques and the varying resource constraints, it is intractable to obtain a closed-form solution to the optimization problem. Inspired by the emerging trend to automate the engineering process of deep model architectures [50] [51], AdaDeep designs a reinforcement learning based optimizer to automatically and effectively solve the constrained optimization problem. We implement AdaDeep with TensorFlow [14] and evaluate its performance over five different public benchmark datasets for DNNs on twelve different mobile devices. Evaluations show that AdaDeep enables a reduction of $2.1 \times -38 \times$ in storage, $1.1 \times -19.8 \times$ in latency, $1.5 \times -4.3 \times$ in energy consumption, and $1.1 \times -9.8 \times$ in computational cost, with a negligible accuracy loss (< 2.1%) for various datasets, tasks, and mobile platforms.

The main contributions of this work are summarized as follows.

- To the best of our knowledge, this is the first work that integrates DNN compression into a hyperparameter tuning framework that aims to balance multiple user-defined requirements and platform resource requirements.
- We propose a reinforcement learning based optimizer to automatically select the best combination of DNN compression techniques. AdaDeep extends the automation of deep model architecture tuning to include DNN compression.
- Experiments show that the DNNs generated by AdaDeep achieve comparable performance to existing compression techniques under various user demands (datasets, tasks, and mobile platforms). AdaDeep also uncovers some combinations of compression techniques suitable for mobile platforms that have not been proposed in previous DNN compression works [6] [20] [31].

In the rest of this paper, we present an overview of AdaDeep in Section 2, formulate the user demands in terms of different metrics in Section 3, present the automatic optimizer in Section 4, and evaluate the performance of AdaDeep in Section 5. Finally we review related work in Section 6 and conclude this work in Section 7.

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 mobile or embedded platforms.

AdaDeep consists of three functional blocks: DNN initialization, user demand formulation, and on-demand optimization (see 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 (see Section 5.1). The user demand formulation block calculates performance and resource constraints imposed by users (see Section 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 constraints to automatically search for an optimal combination of DNN compression techniques that maximizes the system performance while satisfying the resource constraints (see Section 4).

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

$$\arg\max_{J_k \in \text{all}} \mu_1(N(A - A_{\min}) + \mu_2(N(\mu_{\max} - E)$$

subject to

$$T \leq T_{\text{bgt}}, S \leq S_{\text{bgt}},$$

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. Users demands are expressed as a set of constraints on accuracy, energy, latency and storage. Specifically, $A_{\min}$ and $E_{\max}$ are the minimal accuracy and maximal energy acceptable by the user. The two constraints are combined by coefficients $\mu_1$ and $\mu_2$. $N(x)$ is a normalization process, i.e., $N(x) = (x - x_{\min})/(x_{\max} - x_{\min})$ to transform accuracy and energy to the same scale. We denote $T_{\text{bgt}}$ and $S_{\text{bgt}}$ as the latency budget and the storage budget imposed by the target mobile platform. The accuracy $A$, energy cost $E$, latency $T$ and storage $S$ are determined by the DNN architecture as well as the target mobile platform. These variables can be tuned by applying different combinations of DNN compression techniques. The goal of AdaDeep is to select the best combination of compression techniques $J_k$ from the set of all possible combinations $J_{\text{all}}$ that satisfies the performance requirements and resource constraints.

We maximize the accuracy $A$ because it is the most important performance metric for a DNN. We minimize $E$ but only constrain
S within a threshold because mobile devices are usually battery-powered and many applications require continuous inference from deep models. That is, energy efficiency is in general more important than storage for mobile applications. We do not minimize T but limit it within a threshold because the T of many DNNs is acceptable (in the order of millisecond [26]).

Technically, AdaDeep faces two challenges.

- It is non-trivial to derive the accuracy A, energy cost E, latency T and storage S of a DNN. For example, there is still no universal consensus on the estimation models on energy consumption of DNNs on mobile devices. In Section 3, AdaDeep proposes a systematic way to calculate these variables and associates them to the parameters of a DNN and the given mobile platform. We apply the state-of-the-art estimation models and modify them to suit the software/hardware implementation considered in our work. Evaluations show that the models can achieve the same ranking as the measured one on the actual mobile device.

- It is intractable to obtain a closed-form solution to Eq.(1). AdaDeep employs deep reinforcement learning (DRL) based optimization to solve the problem (see Section 4). Although reinforcement learning is a well-known optimization technique, its combination with deep learning (i.e., DRL) and its applications in automatic deep neural network architecture optimization is emerging [50]. We follow this trend and apply a novel DRL structure in the context of automatic DNN model compression for mobile devices.

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 into various mobile devices.

Accuracy A. The inference accuracy is defined as

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

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 needed to run a DNN using the total number of bits associated with weights and activations:

$$S = S_f + S_p = |X| B_A + |W| B_w$$

where $S_f$ and $S_p$ denote the storage requirement for the activations and weights, $X$ and $W$ are the index sets of all activations and weights in the network, and $B_A$ and $B_w$ denote the precision of activations and weights, respectively. For example, $B_A = B_w = 32$ bits in TensorFlow [14].

Computational Cost C. We model the computational cost C of a DNN as the total number of multiply-accumulate (MAC) operations in the DNNs. 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 [45].

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 [45] 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 the total MACs, i.e., $E_c = \varepsilon_1 C$, where $\varepsilon_1$ 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 [8] [48] [47]. Hence E can be modeled as:

$$E = E_c + E_m = \varepsilon_1 C + \varepsilon_2 S_p + \varepsilon_3 S_f$$

where $\varepsilon_2$ and $\varepsilon_3$ denote the energy cost per bit when accessing the Cache and DRAM memory, respectively. To obtain the energy consumption, we refer to a model from state-of-the-art hardware implementation of DNNs in [48], 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:

$$E = \varepsilon_1 \cdot C + 6 \cdot \varepsilon_1 \cdot S_p + 200 \cdot \varepsilon_1 \cdot S_f$$

where $\varepsilon_1$ is measured to be 52.8 pJ for mobile devices.

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

Note that it is difficult to precisely model certain user demand metrics e.g. energy consumption since they are tightly coupled with the underlying hardware and may change from device to device. However, the ranking of the estimated costs of the DNNs derived by the models above is consistent with the ranking of the actual costs of these DNNs measured on mobile devices. As will be introduced in the next section, the proposed AdaDeep framework is generic and more advanced metric estimation models can be easily integrated.

4 ON-DEMAND OPTIMIZATION USING DEEP REINFORCEMENT LEARNING

We leverage deep reinforcement learning (DRL) to solve the optimization in Eq.(1). Specifically, a DQN is employed to learn the automatic agent for selection of hyper-parameters as well as compression techniques, to maximize the performance benefits (i.e., A.
and $E$) while satisfying users’ demands on the cost constraints (i.e., $S$ and $T$). Figure 2 shows the DRL optimizer designed for AdaDeep.

![DQN diagram](image)

**Figure 2: The proposed DRL optimizer for Eq.(1). It takes performance requirements and cost constraints as input, automatically selects compression techniques as well as other hyperparameters, and outputs an optimally compressed neural network. $C$, $M_p$ and $Q$ stand for the conv layer, the max-pooling layer and the DQN based optimizer, respectively.**

### 4.1 Basics for DRL and DQN

Reinforcement learning refers to a machine learning paradigm that learns an optimal policy, by trial and error, for sequential decision making problems [35]. The environment is typically formulated as a Markov decision process (MDP) and the reinforcement learning has been recently combined with deep learning, also known as deep reinforcement learning (DRL) to handle complex input, action and reward functions.

In the literature of MDP and DRL, a policy $\pi$ refers to a specific mapping from the state $o$ to the 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 $\pi$, the action-value (a.k.a. the $Q$ function) of the pair $(o, a)$ under $\pi$ is defined by the action-value, which defines the expected future discounted reward for taking action $a$ in state $o$ and then following policy $\pi$ thereafter. A deep Q-network (DQN) iteratively improves its $Q$-function estimate by taking actions in the environment, observing the reward and next state, and updating the estimate. DQN has the proven capability to find the optimal policy for any finite MDP [35]. Once a DQN is learned, the optimal policy for each state $o$ can be decided by selecting $a$ with the highest $Q$-value. Table 1 explains the contextual definitions of DQN terms in our compression problem.

We propose to adopt DQN for automatic DNN model compression in AdaDeep for the following reasons:

- DQN can implement automatic decision based on the dynamically detected performance and cost metrics.
- The DQN-controller, which is also a neural network architecture, is suited for the feed-forward and back-propagation when training the target regular DNN. The DQN’s output is a decision signal that controls which compressed models are selected and combined. In other words, the regular DNN and the DQN can be trained jointly end-to-end [33].
- DRL is suited for non-linear and non-differentiable optimization. Also, within the framework of DQN, we can add some more new compression techniques by simply adding branch sub-networks (optional actions), and figure out the function of the complex optimization problem’s input and results by the DQN mapping. Therefore a DQN-based optimizer provide both capability and flexibility in DNN compression.

To apply DQN in DNN compression, we need to (i) carefully design a reward function to utilize DQN to solve a constrained optimization problem; and (ii) design a DQN structure with tractable computation complexity. In this work we propose a novel DQN based optimizer for Eq.(1), which separates the reward of performance gain and constraint satisfaction into two streams by two parallel parts in the dueling DQN structure [46].

### 4.2 Design of Reward Function in AdaDeep

To define the reward function $R$ to optimize Eq.(1), a common approach is to use the Lagrangian Multiplier function [21] to first convert the constrained formulation into an unconstrained one:

$$
R = [\mu_1 \text{Norm}(A - A_{\text{min}}) + \mu_2 \text{Norm}(E_{\text{max}} - E) + \mu_3 \text{Norm}(T_{\text{avg}} - \frac{C}{P}) + \mu_4 \text{Norm}(S_{\text{cache}} - S_P)]
$$

where $\mu_1$, $\mu_2$, $\mu_3$ and $\mu_4$ are the Lagrangian multipliers. It merges the objective (e.g., accuracy, energy) and the constraint satisfaction (how well the latency and storage usages meet budget requirements). However, maximizing Eq.(6) rather than Eq.(1) can cause ambiguity, in the sense that the following two situations may lead to the same objective values and 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 latency/storage usage defined by end developers.

To avoid such ambiguity, we define two loss functions for the objective gain and for the constraint satisfaction, respectively. We adopt a dueling DQN architecture [46], which separates the state-action value function and the state-action advantage function (see Figure 3) into two parallel streams. The two stream share the convolutional (conv) layers with parameters $\omega$ which learn the representations of state, but are followed by two separate columns to generate state-action objective gain value $G$ with weight parameter $\beta$, and the state-action constraint satisfaction value $H$ with weight parameter $\eta$, respectively. The two columns are then aggregated to output an single state-action value $Q$. We define a novel $Q$ value:

$$
Q(o, a; \omega, \beta, \eta) = G(o, a; \omega, \beta) + H(o, a; \omega, \eta)
$$
The network $G$ and $H$ comes with their corresponding reward functions $R_1$ and $R_2$:

$$R_1 = \mu_1 \text{Norm}(A - A_{\text{min}}) - \mu_2 \text{Norm}(E_{\text{max}} - E)$$

$$R_2 = \mu_3 \text{Norm}(T_{\text{tgt}} - \frac{C}{p}) + \mu_4 \text{Norm}(S_{\text{Cache}} - S_p)$$

After taking an action, we can observe the reward $R_1$ for $G$, and $R_2$ for $H$. Their interaction and balance guide the selection process.

### Algorithm 1: DRL Optimizer Algorithm

**Input:** $D_t$, Budgets, $a \sim A$  
**Output:** $D_{\text{NNoptimal}}$

1. Initialize $O_s$, $A_s$, $D_{\text{NN}}$;
2. Initialize $DQN$’s predict $Q$ with random $\omega$, $\beta$, $\eta$;
3. Initialize $DQN$’s target $\bar{Q}$ with weights $\overline{\omega}$, $\overline{\beta}$, $\overline{\eta}$;
4. for episode in range(1000) do
   5. Select $a_t$ for $o_t$ by $Q$ value ($e = \text{greedy}$) and observe $R_t$;
   6. Store $(o_t, a_t, R_t, o_{t+1})$ in reply memory $\Lambda$;
   7. $Q_{\text{tgt}} = R_1 + R_2 + \gamma Q(o', \text{argmax} Q(o', a'; \omega, \beta, \eta)); \overline{\omega}, \overline{\beta}, \overline{\eta}$;
   8. Perform greedy descent iteratively to tune $DQN$’s $\omega$ on loss of random mini-batches replay:
      $L(\omega) = \mathbb{E}_{(o, a, r, o')} - \Delta (Q_{\text{tgt}} - Q(o, a; \omega, \beta, \eta))^2$; Every num steps reset $Q_{\text{tgt}} = Q$;
9. end

### 4.3 Design of DRL Optimizer in AdaDeep

The proposed DRL optimizer based on DQN is outline in Algorithm 1. We select random action with probability $\epsilon$ and select the action with largest $Q$ value by $1 - \epsilon$ probability ($\epsilon = 0.001$ by default). To build a $DQN$ with weight parameters $\omega$, $\beta$, and $\eta$, we optimize the following loss function at iteration $i$ to update $Q(o, a; \omega, \beta, \eta)$.

$$L(\omega) = \mathbb{E}_{(o, a, r, o')} - \Delta (Q_{\text{tgt}} - Q(o, a; \omega, \beta, \eta))^2$$

with the frozen $Q$ value learned by target network [43]:

$$Q_{\text{tgt}} = R_1 + R_2 + \gamma Q(o', \text{argmax} Q(o', a'; \omega, \beta, \eta)); \overline{\omega}, \overline{\beta}, \overline{\eta}$$

We adopt the standard DQN training techniques [46] and use the update rule of SARS [44] with the assumption that future rewards are discounted by a factor $\gamma$ [35] of the default value 0.1. The $DQN$ is further trained with random samples from reply memory $\Lambda$ to increase the efficiency of experience replaying.

Note that since DRL-based optimization is still heuristic, the proposed optimizer in AdaDeep cannot theoretically guarantee a global optimal solution. However, as we will show in the evaluations, the DRL-based optimizer outperforms exhaustive or greedy approaches in terms of the performance of the compressed DNNs.

### 5 EVALUATION

This section presents the evaluations of AdaDeep across various recognition tasks and mobile platforms.

#### 5.1 Experiment Setup

We first present the settings for our evaluation.

**Implementation.** We implement AdaDeep with TensorFlow [14] 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 from a pool of three state-of-the-art DNN models, including LeNet [29], AlexNet [23], and VGG [37], according to the size of samples in $D_t$. For example, LeNet is selected when the sample size is smaller than 28x28, otherwise AlexNet or VGG is chosen. Standard training techniques, such as stochastic gradient descent (SGD) and Adam [22], are used to obtain weights for the DNNs.

**Evaluation applications and datasets.** To evaluate the performance of the proposed AdaDeep, five commonly used mobile applications are considered, for which the corresponding benchmark datasets are summarized in Table 2. Specifically, AdaDeep is evaluated for hand-written digit recognition ($D_1$: MNIST [28]), image classification ($D_2$: Cifar10 [24] and $D_3$: ImageNet [9]), audio sensing application ($D_4$: Ubisound [36]), and activity recognition ($D_5$: Har [42]). According to the sample size (see Table 2), LeNet [29] is selected as the initial DNN for $D_1$, $D_2$, $D_4$, and $D_5$, while AlexNet [23] and VGG-16 [37] are chosen for $D_3$.

**Mobile platforms for evaluation.** We evaluate AdaDeep on twelve commonly used mobile 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 as elaborated in Table 3.

### 5.2 Layer Compression Technique Benchmark

In our experiment, the first step is to 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 $E$. The outcomes of this study indicates the need to suitably combine compression techniques for different user demands on performance and cost constraints, and will also serve as baselines for evaluating AdaDeep.

#### 5.2.1 Benchmark Settings

We apply ten mainstream compression techniques from three categories, i.e., weight compression
(W_{1f}, W_{2}, W_{3}, W_{1e}), 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) [23] and compare their performance evaluated on CIFAR-10 dataset (D_2) [24] on a RedMi 3S smartphone (Device 1 in Table 3). The details of the compression techniques are as follows.

- \(W_{1f}\): insert a fully-connected (fc) layer between fc_1 and fc_{i+1} layers using the singular value decomposition (SVD) based weight matrix factorization [25]. The number of neurons \(k\) in the inserted layer is set as \(k = m/12\), where \(m\) is the number of neurons in fc_1.
- \(W_{2}\): insert a fc layer between fc_1 and fc_{i+1} using sparse-coding, another matrix factorization method [6]. The k-basis dictionary used in \(W_{2}\) is set as \(k = m/6\), where \(m\) is the number of neurons in fc_1.
- \(W_{3}\): prune fc_1 and fc_2 using the magnitude based weight pruning strategy proposed in [16]. It removes unimportant weights whose magnitudes are below a threshold (i.e., 0.001).
- \(L_{1}\): replace the traditional fc layers, fc_1 and fc_{i+1}, with a global average pooling layer [31]. It generates one feature map for each category in the last conv layer. The feature map is then fed into the softmax layer.
- \(W_{1e}\): insert a conv layer between conv_1 and pool_1 using SVD based weight factorization [25]. The numbers of neurons \(k\) in the inserted layer by SVD is set as \(k = m/12\), where \(m\) is the number of neurons in conv_1.
- \(C_1\): decompose conv_1 using convolution kernel sparse decomposition [32]. 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 [18] and we set the width multiplier \(\alpha = 0.5\). It is a key technique of Google’s MobileNet [18], which decomposes the standard conv into a depth-wise convolution and a \(1 \times 1\) point-wise convolution.
- \(C_3\): decompose conv_i using the sparse random technique [7] 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_1 by a Fire layer [20]. A Fire layer is composed of a \(1 \times 1\) conv layer and a conv layer with a mix of \(1 \times 1\) and \(3 \times 3\) conv filters. It decreases the sizes of input channels and filters.
- \(L_{2}\): replace conv_1 by a micro multi-layer perceptron embedded with multiple small kernel conv layers (Mlpconv) [31]. It approximates a nonlinear function to enhance the abstraction of conv layers with small (e.g. \(1 \times 1\)) conv filters.

The parameters (i.e., \(k\) in \(W_{1f}, W_{1e}\) and \(W_{2}\), the depth multiplier \(\alpha\) in \(C_2\), the sparse random multiplier \(\theta\) in \(C_3\)) are empirically optimized by comparing the performance improvement on the layer where the compression technique is applied.

As shown in Figure 4, 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 \(W_{1e}, C_1, C_2, C_3, L_{1}\) 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 Device 1 to process the test data 10 times, and obtain the mean and variance of the inference.

### Table 2: Summary of the applications and corresponding datasets for evaluating AdaDeep.

<table>
<thead>
<tr>
<th>No.</th>
<th>Task</th>
<th>Dataset</th>
<th>Description</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_1</td>
<td>Digit</td>
<td>MNIST [28]</td>
<td>55,000 images, 10 classes</td>
<td>(28, 28, 1)</td>
</tr>
<tr>
<td>D_2</td>
<td>Image</td>
<td>CIFAR-10 [24]</td>
<td>60,000 images, 10 classes</td>
<td>(32, 32, 3)</td>
</tr>
<tr>
<td>D_3</td>
<td>Image</td>
<td>ImageNet [9]</td>
<td>a small version of ImageNet, 65,000 images, 5 classes</td>
<td>(120, 120, , 3)</td>
</tr>
<tr>
<td>D_4</td>
<td>Audio</td>
<td>UbiSound [36]</td>
<td>7,500 audio clips in wav, 9 classes</td>
<td>(40, 40, 1)</td>
</tr>
<tr>
<td>D_5</td>
<td>Activity</td>
<td>Har [42]</td>
<td>10,000 records of accelerometer and gyroscope, 7 classes</td>
<td>(33, 17, 1)</td>
</tr>
</tbody>
</table>

### Table 3: Summary of the varied resource constraints of the mobile platforms for evaluating AdaDeep.

<table>
<thead>
<tr>
<th>Type</th>
<th>Device</th>
<th>Processor</th>
<th>DRAM</th>
<th>Cache</th>
<th>Battery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smart phones</td>
<td>1. Xiaomi Redmi 3S</td>
<td>Qualcomm 430</td>
<td>3 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>4100 mAh</td>
</tr>
<tr>
<td></td>
<td>2. Xiaomi Mi 5S</td>
<td>Qualcomm B21</td>
<td>4 GB</td>
<td>L_2-Cache(1 MB)</td>
<td>3200 mAh</td>
</tr>
<tr>
<td></td>
<td>3. Xiaomi Mi 6</td>
<td>Qualcomm 835</td>
<td>6 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>3350 mAh</td>
</tr>
<tr>
<td></td>
<td>4. Huawei pra-al00</td>
<td>HiSilicon kirin655</td>
<td>3 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>3000 mAh</td>
</tr>
<tr>
<td></td>
<td>5. Samsung note5</td>
<td>samsung exynos7420</td>
<td>4 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>3000 mAh</td>
</tr>
<tr>
<td></td>
<td>6. HuaweiP9</td>
<td>HiSilicon kirin955</td>
<td>3 GB</td>
<td>L_2-Cache(4 MB)</td>
<td>3000 mAh</td>
</tr>
<tr>
<td>Wearable devices</td>
<td>7. Sony watch SW3</td>
<td>Quad-core cortexA7</td>
<td>512 MB</td>
<td>L_2-Cache(1 MB)</td>
<td>420 mAh</td>
</tr>
<tr>
<td></td>
<td>8. Huawei watchH2P</td>
<td>Snapdragon wear2100</td>
<td>768 MB</td>
<td>L_2-Cache(1 MB)</td>
<td>420 mAh</td>
</tr>
<tr>
<td>Boards</td>
<td>9. firefly-rk3999</td>
<td>Dual-core cortexA72</td>
<td>2 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>power-plug</td>
</tr>
<tr>
<td></td>
<td>10. firefly-rk3288</td>
<td>Dual-core cortexA17</td>
<td>2 GB</td>
<td>L_2-Cache(8 MB)</td>
<td>power-plug</td>
</tr>
<tr>
<td>Smart home</td>
<td>11. Xiaomi box 3S</td>
<td>Amlogic S905</td>
<td>2 GB</td>
<td>L_2-Cache(2 MB)</td>
<td>power-plug</td>
</tr>
<tr>
<td></td>
<td>12. Huawei box</td>
<td>Hisilicon hi3798M</td>
<td>2 GB</td>
<td>L_2-Cache(1 MB)</td>
<td>power-plug</td>
</tr>
</tbody>
</table>
A parameter storage of 6.09MB, an energy cost of 30.72mJ, and a detection latency of 189ms. Second, compared with the compressed model using W3, some combinations of layer compression techniques, e.g. C2+W5 and C2+L3, reduce more than 48mJ of the energy cost E, decrease the latency by about 103ms, and dramatically cut down the storage requirement Sb by more than 18MB, while incurring only 2.4% accuracy loss. On the other hand, some combinations, e.g. W1c+L3 and C3+W5, incur over 28% accuracy loss, and might perform worse than a single compression technique. Third, the combination of L1+W5 achieves the best balance between system performances and resource constraints.

Summary. Some combinations of two layer 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 the proper combination of compression techniques.

5.3 Performance of AdaDeep

In this subsection, we first evaluate the performance of AdaDeep’s core block DRL optimizer, and then test performance of AdaDeep over five different tasks and on twelve different mobile platforms. Furthermore, to show the flexibility of AdaDeep in adjusting the optimization objectives based on user demand, we show some examples of the choices on the scaling coefficients in Eq. (8).

5.3.1 Performance of the proposed DRL optimizer. This experiment is to evaluate the advantage of the proposed DRL optimizer when searching for the optimal compression combination. To do so, we compress [LeNet, MNIST] and [AlexNet, CIFAR-10] using the 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 (x) are normalized over the compressed DNNs using the W3 technique.

- **Exhaustive optimizer**: This scheme exhaustively test the performance of all combinations of two compression techniques (similar to Section 5.2.3), 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. (6). The selected one is L2+L3, i.e., Fixed, in both the cases of LeNet on MNIST and AlexNet on CIFAR-10.

- **Greedy optimizer**: It loads the DNN layer by layer and selects the compression technique that has the largest reward value defined by Eq. (6), in which both μ1 and μ2 are set to be 0.5. Also, when T or S violate the budget Tbgt or Sbgt, the optimization terminates.

- **DRL optimizer**: It compresses the DNN using the DRL optimizer as described in Section 4. We set the scaling coefficients in Eq. (8) to be μ1 = 0.6 and μ2 = 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 μ3 = 0.5 and μ4 = 0.5 in Eq. (8) because their corresponding constraints (i.e., C and Sb) are equally important. The same as in the Greedy search within this subsection.

Table 5 summarizes the best performance achieved by the above three optimizers. We can see that the networks generated by our
Figure 5: 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 compression techniques $W_{1f}$, $W_2$, $W_3$, and $L_3$ are applied to $fc_1$ and $fc_2$, while $W_{1c}$, $C_1$, $C_2$, $C_3$, $L_1$ and $L_2$ are applied to $conv_2$. The Y-axis denotes the accuracy loss (%) over the initial AlexNet and the cost reduction over the initial layer that they are applied to.

Figure 6: 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 $E$. The compression techniques $W_{1f}$, $W_2$, $W_3$, and $L_3$ are applied to $fc_1$ and $fc_2$, while $W_{1c}$, $C_1$, $C_2$, $C_3$, $L_1$ and $L_2$ are applied to $conv_2$. The Y-axis denotes the accuracy loss (%) and the cost reduction over the original entire AlexNet.

DRL optimizer achieve the best overall performance in terms of storage, latency, and energy consumption, while incurring negligible accuracy loss (0.1% or 2.1%), compared to those generated by the two baseline optimizers. In particular, compared with the DNN compressed by $W_3$, the best DNN from the Greedy optimizer only reduces the storage size $S_p$ by 4.6× and 2.2× in the cases of [LeNet, MNIST] and [AlexNet, CIFAR-10], respectively. In contrast, the best DNN from the Exhaustive optimizer, i.e., Fixed, can reduces $S_p$ by 23.9× and 3.5×, respectively, while that generated by the DRL optimizer achieves a maximum reduction of 28.5× and 4.6× on $S_p$, respectively, in the two cases. Second, the network from the DRL optimizer is also the most effective in reducing the latency (>2.3×) in both experiments, while those from the two baseline optimizers may result in an increased latency in some cases. For example, the network from the Greedy optimizer increases $T$ by 0.6× in the [LeNet, MNIST] experiment and the one from the Exhaustive optimizer introduces an 0.7× extra latency in the [AlexNet, CIFAR-10] experiment. Third, when comparing the energy cost, Fixed is the least energy-efficient (reduce the energy consumption by only 1.1× over the DNN compressed by $W_3$), while those from both the DRL and Greedy optimizers achieve an energy reduction of 1.8× to 2.8×, respectively. Meanwhile, the accuracy loss of the networks from
Table 4: 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 ($D_1$).

<table>
<thead>
<tr>
<th>Compression technique</th>
<th>Measured accuracy &amp; cost</th>
<th>Compression technique</th>
<th>Measured accuracy &amp; cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 + W_1f$</td>
<td>$A(%)$ 15.3 $S_p$, $E(mJ)$ 62.8</td>
<td>$L_1 + W_1f$</td>
<td>$A(%)$ 15.3 $S_p$, $E(mJ)$ 62.8</td>
</tr>
<tr>
<td>$C_1 + W_2$</td>
<td>$A(%)$ 12.3 $S_p$, $E(mJ)$ 65.2</td>
<td>$L_1 + W_2$</td>
<td>$A(%)$ 15.3 $S_p$, $E(mJ)$ 189 34.4</td>
</tr>
<tr>
<td>$C_1 + W_3$</td>
<td>$A(%)$ 23.2 $S_p$, $E(mJ)$ 63.48</td>
<td>$L_1 + W_3$</td>
<td>$A(%)$ 1.1 $S_p$, $E(mJ)$ 86 13.9</td>
</tr>
<tr>
<td>$C_1 + L_4$</td>
<td>$A(%)$ 0.68 $S_p$, $E(mJ)$ 52.9</td>
<td>$L_1 + L_4$ [20]</td>
<td>$A(%)$ 1.6 $S_p$, $E(mJ)$ 257 78.2</td>
</tr>
<tr>
<td>$C_2 + W_1f$</td>
<td>$A(%)$ 15.3 $S_p$, $E(mJ)$ 33.1</td>
<td>$L_2 + W_1f$</td>
<td>$A(%)$ 17.4 $S_p$, $E(mJ)$ 305 108.4</td>
</tr>
<tr>
<td>$C_2 + W_2$</td>
<td>$A(%)$ 14.9 $S_p$, $E(mJ)$ 31.3</td>
<td>$L_2 + W_2$</td>
<td>$A(%)$ 17.1 $S_p$, $E(mJ)$ 312 100.1</td>
</tr>
<tr>
<td>$C_2 + W_3$</td>
<td>$A(%)$ 2.9 $S_p$, $E(mJ)$ 14.8</td>
<td>$L_2 + W_3$</td>
<td>$A(%)$ 10.6 $S_p$, $E(mJ)$ 266 51.6</td>
</tr>
<tr>
<td>$C_2 + L_3$</td>
<td>$A(%)$ 0.7 $S_p$, $E(mJ)$ 16.9</td>
<td>$L_2 + L_3$ [31]</td>
<td>$A(%)$ 1.8 $S_p$, $E(mJ)$ 126 113.4</td>
</tr>
<tr>
<td>$C_3 + W_1f$</td>
<td>$A(%)$ 16.7 $S_p$, $E(mJ)$ 43.4</td>
<td>$W_{1c} + W_{1f}$</td>
<td>$A(%)$ 16.1 $S_p$, $E(mJ)$ 139 36.1</td>
</tr>
<tr>
<td>$C_3 + W_2$</td>
<td>$A(%)$ 15.7 $S_p$, $E(mJ)$ 42.7</td>
<td>$W_{1c} + W_{2f}$ [6]</td>
<td>$A(%)$ 16.4 $S_p$, $E(mJ)$ 147 39.3</td>
</tr>
<tr>
<td>$C_3 + W_3$</td>
<td>$A(%)$ 3.2 $S_p$, $E(mJ)$ 21.7</td>
<td>$W_{1c} + W_3$</td>
<td>$A(%)$ 2.7 $S_p$, $E(mJ)$ 143 20.8</td>
</tr>
<tr>
<td>$C_3 + L_3$</td>
<td>$A(%)$ 1.4 $S_p$, $E(mJ)$ 26.8</td>
<td>$W_{1c} + L_3$</td>
<td>$A(%)$ 1.2 $S_p$, $E(mJ)$ 27 22.9</td>
</tr>
</tbody>
</table>

Table 5: Performance of the best DNN generated by the DRL and baseline optimizers and tested on a RedMi 3S smartphone (Device 1) using LeNet on MNIST ($D_1$) and AlexNet on CIFAR-10 ($D_2$). The accuracy loss $\%$ and the cost reduction $\times$ are normalized over the corresponding DNN compressed using $W_3$.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Compared to the compressed LeNet</th>
<th>Compared to the compressed AlexNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive</td>
<td>$A$ loss: 0.1% $S_p$: 23.9 $T$: 2.7 $E$: 1.1 $\times$</td>
<td>$A$ loss: -7.2% $S_p$: 3.5 $T$: 0.7 $E$: 1.2 $\times$</td>
</tr>
<tr>
<td>Greedy</td>
<td>$A$ loss: 2.3% $S_p$: 4.6 $T$: 0.6 $E$: 2.7 $\times$</td>
<td>$A$ loss: 0.3% $S_p$: 2.2 $T$: 1.2 $E$: 1.9 $\times$</td>
</tr>
<tr>
<td>DRL</td>
<td>$A$ loss: 0.2% $S_p$: 28.5 $T$: 3.8 $E$: 2.8 $\times$</td>
<td>$A$ loss: -4.9% $S_p$: 4.6 $T$: 2.3 $E$: 1.8 $\times$</td>
</tr>
</tbody>
</table>

the two baseline optimizers ranges from 0.1% to 2.3%, while those from our proposed DRL optimizer have the best accuracy (only a 0.2% degradation in the [LeNet, MNIST] case and even a 4.9% gain in the [AlexNet, CIFAR-10] case). Finally, as for the training time, the DRL optimizer requires a shorter, or equal, or longer time compared with the exhaustive and Greedy optimizers (see Section 5.3.3 for examples of DRL’s required training time). Also, we recognize that it is impossible to compare the performance of the network generated by AdaDeep with that of the DNN compressed by the optimal combination among all possible combinations, because it is not practical to test all possible combinations of the considered compression techniques and identify the optimal one.

Summary. DRL optimizer outperforms the other two schemes 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 constraints ($S$ and $T$) are systematically included in the reward value and adaptively feedback to the DRL decision process.

5.3.2 AdaDeep over Different Tasks. In this experiment, AdaDeep is evaluated on all the five tasks/datasets (see Table 2) 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 five 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 $W_5$. Compared with their initial DNNs, DNNs generated by AdaDeep can achieve a reduction of 1.8$\times$ - 38$\times$ in $S_p$, 0.8$\times$ - 3.3$\times$ in $C$, 0.8$\times$ - 19.8$\times$ in $T$, and 1.1$\times$ - 4.3$\times$ in $E$, with a negligible accuracy loss ($< 1\%$) 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 $C_3 + W_5$ for Task 1 (on MNIST initialized using LeNet), $L_1 + W_5$ for Task 2 (on Cifar10 initialized using AlexNet), $L_2 + C_1 + L_3$ for Task 3 (on ImageNet initialized using AlexNet), $C_3 + C_2 + L_3$ for Task 4 (on ImageNet initialized using VGG), $C_3 + C_2 + C_3 + L_3$ for Task 5 (on Ubisound initialized using LeNet), and $L_1 + W_5$ for Task 6 (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 technique in all metrics, it achieves a better overall performance in terms of the five metrics according to the specific user demands.

5.3.3 AdaDeep over Different Mobile Devices. This experiment evaluates AdaDeep across twelve different mobile devices (see Table 3) 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.

As shown in Table 3, different devices have different resource constraints, which lead to different performance and budget demands.
Table 6: Performance of AdaDeep evaluated on different tasks/datasets using a RedMi 3S smartphone (Device 1), normalized over the corresponding DNNs compressed using $W_5$. The compression techniques marked by “*” are the combinations that have not been proposed in related studies.

<table>
<thead>
<tr>
<th>Task</th>
<th>Compression techniques</th>
<th>Compare to the DNN compressed using $W_5$ technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. MNIST (LeNet)</td>
<td>$C_3 + W_3$</td>
<td>1.8× 1.5× 1.8× 1.3× −2.5%</td>
</tr>
<tr>
<td>2. CIFAR-10 (AlexNet)</td>
<td>$L_1 + W_3$</td>
<td>4.6× 3.1× 2.3× 1.8× −4.9%</td>
</tr>
<tr>
<td>3. ImageNet (AlexNet)</td>
<td>$L_2 + C_1 + L_3$</td>
<td>18.2× 2.1× 3.7× 1.2× −1.2%</td>
</tr>
<tr>
<td>4. ImageNet (VGG)</td>
<td>$L_2 + C_2 + L_3$</td>
<td>38× 3.3× 19.8× 4.3× 0.1%</td>
</tr>
<tr>
<td>5. Ubisound (LeNet)</td>
<td>$C_3 + L_3$</td>
<td>3.2× 1.9× 1.6× 1.1× 0.4%</td>
</tr>
<tr>
<td>6. Har (LeNet)</td>
<td>$L_1 + W_3$</td>
<td>2.1× 0.8× 0.8× 1.5× −2.6%</td>
</tr>
</tbody>
</table>

and thus require different coefficients $\mu_1 \sim \mu_4$ in Eq. (8). Specifically, we empirically optimize $\mu_1 \sim \mu_4$ for different devices to be: $\mu_2 = \max\{\frac{2000 - E_{battery}}{4000}, 0.6\}$, $\mu_1 = 1 - \mu_2$, $\mu_4 = \max\{\frac{8 - S_{cache}}{8}, 0.6\}$, and $\mu_3 = 1 - \mu_2$.

**Performance.** Table 7 summarizes the generated compression combinations by AdaDeep and the performance of the corresponding compressed DNNs. For the twelve different resource constraints, DNNs generated by AdaDeep, which are initiated with the same DNN model, can reduce the parameter size by 3× - 12×, computation cost by 1.6x - 6.8x, latency by 1.1x - 3.1x and energy cost by 1.1x - 9.8x, 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 $L_1$, $L_2$, $L_3$) or fine-tuning (such as in $W_3$) DNN for compression, and training the DQN 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 DQN based selection on different tasks, which is 3 hours on [MNIST, LeNet], 10 hours on [CIFAR-10, LeNet], 6.5 hours on [CIFAR-10, AlexNet], 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 ($\mu_1 \sim \mu_4$).

6 RELATED WORK
Our work is closely related to the following research.

6.1 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 of DNNs. Common hyperparameter tuning techniques can be categorized into parallel search, such as grid search [5] and random search [4], and sequential search, such as Bayesian optimization [38]. The grid and random search approaches search blindly and thus are usually time-consuming. Bayesian approaches [39] [11] [40] automatically optimize the hyperparameters, but can still be slow due to the intrinsically sequential operations.

In general, DNNs need to be compressed either during or after training to be deployed on resource-constrained mobile devices, due to their high complexity. Selecting compression techniques can be viewed as a hyperparameter tuning process. Inspired by state-of-the-art automatic hyperparameter optimization techniques, AdaDeep is the first work to treat compression techniques as a new hyperparameter to be tuned during training.

6.2 DNN Compression
The success and popularity of machine learning in mobile and IoT applications [12] [19] [27] [49] has stimulated the adoption of more powerful DNNs in mobile and embedded devices. 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 [41]. Various DNN compression techniques have been proposed, including weight compression [6] [16] [25], convolution decomposition [7] [18] [32], and compact architectures [20] [31]. However, existing compression techniques investigate a one-for-all scheme, e.g. how to reduce DNN complexity using one compression technique, and do not consider the various resource constraints across different deployment platforms.

AdaDeep enables an automatic selection of the best combination of different compression techniques that 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.

6.3 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 [17] pre-evaluates a set of compressed models with different execution...
overhead 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 [13] designs a low power unit resource scheduler to maximize the energy efficiency for the unique workload of different tasks on heterogeneous computation resources. DeepX [25] designs a set of resource control algorithms to decompose DNNs into different unit-blocks for efficient execution on heterogeneous computation resources. EIE [15] 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 [13] and [25] 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 [15].

### 6.4 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 [35]. For example, Mnih et al. [35] propose to learn control policies from complex sensory inputs using a deep Q-network (DQN). Liu et al. [33] 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. [1] 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. [3] present a framework to tackle the combinatorial optimization of sequential problems with DRL and recurrent DNN.

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

### 6.5 Automatic DNN Architecture Optimization

An emerging topic of interest 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 [50], or by transferring architectural building blocks to construct scalable architectures on larger datasets [51]. 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. [2] 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. [10] 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.

### 7 CONCLUSION

This paper presents AdaDeep, an automatic optimization framework that selects a combination of compression techniques that balance diverse user-specified performance goals and device-imposed resource constraints. We systematically formulate the goals and constraints on accuracy, latency, storage and energy into a unified optimization problem, and leverage a deep reinforcement learning based strategy to effectively find the feasible combination of compression techniques. Evaluations on five widely used datasets and twelve different mobile devices show that there is no one-fit-all compression technique that meets the specific performance goals and resource constraints. It also figures out some combinations of compression techniques unexplored in previous DNN compression research. AdaDeep is the first work that models DNN compression as a new hyperparameter for automatic tuning. In the future,
we plan to investigate fully automatic hyper-parameter tuning to optimize DNNs suited for mobile and embedded platforms.

ACKNOWLEDGEMENTS

We are grateful for our shepherd Professor Nic Lane (University of Oxford) and the anonymous reviewers for their valuable comments and suggestions, Professor Lin Zhong (Rice University) for his useful feedback on an early version of this paper, Xin Wang and Yuheng Wei (Xidian University) for their help on implementing some of the baseline techniques, and Prof. Zhangyang Wang (Texas A&M University) for his useful discussion on Section 4. This work is supported in part by National Key Research & Development Program of China #2018YFB1003605, Natural Science Foundation of China (NSFC) #61472312, and Natural Science Foundation (NSF) Award #1611295.

REFERENCES