A Co-Scheduling Framework for DNN Models on Mobile and Edge Devices with Heterogeneous Hardware

Zhiyuan Xu, Dejun Yang, Chengxiang Yin, Jian Tang, *Fellow, IEEE*,
Yanzhi Wang, Guoliang Xue, *Fellow, IEEE*

Abstract—With the emergence of more and more powerful chipsets and hardware and the rise of Artificial Intelligence of Things (AIoT), there is a growing trend for bringing Deep Neural Network (DNN) models to empower mobile and edge devices with intelligence such that they can support attractive AI applications on the edge in a real-time or near real-time manner. To leverage heterogeneous computational resources (such as CPU, GPU, DSP, etc) to effectively and efficiently support concurrent inference of multiple DNN models on a mobile or edge device, we propose a novel online Co-Scheduling framework based on deep REinforcement Learning (DRL), which we call COSREL. COSREL has the following desirable features: 1) it achieves significant speedup over commonly-used methods by efficiently utilizing all the computational resources on heterogeneous hardware; 2) it leverages emerging Deep Reinforcement Learning (DRL) to make dynamic and wise online scheduling decisions based on system runtime state; 3) it is capable of making a good tradeoff among inference latency, throughput and energy efficiency; and 4) it makes no changes to given DNN models, thus preserves their accuracies. To validate and evaluate COSREL, we conduct extensive experiments on an off-the-shelf Android smartphone with widely-used DNN models to compare it with three commonly-used baselines. Our experimental results show that 1) COSREL consistently and significantly outperforms all the baselines in terms of both throughput and latency; and 2) COSREL is generally superior to all the baselines in terms of energy efficiency.

Index Terms—Mobile Computing, Edge Computing, Deep Learning, Deep Reinforcement Learning, On-device DNN Inference

1 Introduction

Over the past few years, Deep Learning (DL) [29], [12] (e.g., Deep Neural Networks (DNNs)) has become the *de facto* approach for a variety of tasks (such as image classification, face recognition, object detection, action recognition, machine translation, etc) in multiple domains including computer vision and Natural Language Processing (NLP). With the emergence of more and more powerful chipsets and hardware and the rise of Artificial Intelligence of Things (AIoT), there is a growing need for bringing DNN models to empower mobile and edge devices with intelligence such that they can support attractive AI applications, such as flower recognition on a smartphone, voice-activated digital assistant, Driver Monitoring System (DMS) and Advanced Driver Assistance System (ADAS), in a real-time or near real-time manner.

However, deploying large DNN models onto resource-limited devices is quite challenging since most commonly-used DNNs have very complex architectures with a huge number of layers and parameters (e.g. ResNet152 [16] and Faster-RCNN [50]) and thus are known to be computationally intensive [53] and slow (inference with most DNNs cannot be real-time or near real-time even with a powerful GPU server); while most mobile and edge devices have very limited computing power and resources. Recently, research and development efforts have been made to overcome this challenge in both industry and academia. First, a few DL frameworks have been developed particularly for supporting DNN inference over mobile devices. For example, Google launched Tensorflow Lite [57], which extends the widely-used Tensorflow [1] to mobile devices. Facebook released PyTorch Mobile [47], which is a mobile-optimized framework for on-device DNN and quantized DNN inference. These frameworks aim to provide special (i.e., mobile-version) implementation of operators needed by DNNs to support and accelerate their inference on mobile devices. Another line of research targeted at this challenge from the application/model perspective by compressing DNN models via weight pruning, quantization and knowledge distillation. For example, in a pioneering work, Han *et al.* [14] presented a simple algorithm to compress DNN models by pruning unimportant weights. Later, Wen *et al.* [59] proposed a structured pruning method, which has been shown to be more effective. Commonly-used 8-bit quantization can reduce the size of a DNN model (using 32-bit float numbers to represent its weights) by 4x without any loss on accuracy [4]. Very extreme methods, such as binary neural networks [23], [32], use only 1-bit to represent each weight,

- *Zhiyuan Xu, Chengxiang Yin, and Jian Tang (corresponding author) are with Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244 USA. E-mail: {zxu105, cyin02, jtang02}@syr.edu. Dejun Yang is with the Department of Computer Science, Colorado School of Mines, Golden, CO 80401 USA. Email: djyang@mines.edu. Yanzhi Wang is with Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA. E-mail: yanz.wang@northeastern.edu. Guoliang Xue is with the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287 USA. E-mail: xue@asu.edu.*
- *This research was supported in part by NSF grants 1704662 and 1704092. The information reported here does not reflect the position of the government.*

which turn out to be very effective in terms of compression and acceleration, but usually suffer from substantial accuracy loss. Hinton *et al.* [19] introduced an interesting method called knowledge distillation, which trains a small student network using both ground truth (a.k.a hard) labels and the results given by a large teacher network (a.k.a soft labels). All these three model compression approaches have been widely used in commercial AI applications. We, however, overcome this challenge from a whole new angle by jointly scheduling multiple DNNs over heterogeneous hardware, which is completely complementary to those deep learning frameworks and model compression techniques introduced above, i.e., it can be used together with them to further optimize and accelerate DNN inference.

Currently, it is very common for a mobile/edge device to be equipped with heterogeneous hardware, such as CPU, GPU, DSP and emerging AI chipsets [39] (e.g. TPU, NPU, VPU, etc). However, scant attention has been paid to studying how to fully harness their power by making them work collaboratively to speed up DNN inference in the context of mobile and edge computing. Furthermore, we consider a practical scenario, in which multiple (independent or correlated) DNN models work together on a mobile/edge device. For example, we may use three different but correlated DNNs in a mobile application: one for face detection, one for facial expression recognition and another for facial landmarks estimation. However, some related works [69], [63] only addressed the inference of a single DNN model, which represents a rather simplistic case.

When a user runs a DNN model on a mobile device, its CPU undertakes all inference workload by default [57], [47] without the help of GPU or other available hardware, which is obviously not efficient (See Sections 2 and 4). In a cloud with powerful GPU servers (e.g., NVIDIA T4 [44]), it is a common practice to simply offload all DNN models to GPUs for high-throughput and low-latency processing. However, computational hardware on a mobile/edge device usually has much less computing power than those GPUs designed particularly for DNNs in a cloud. For example, the GPU on a Google's Pixel 2 XL smartphone has a limited computing power of only 567 GFLOPs [2]; while an NVIDIA T4 GPU offers a much higher computing power of $8,100$ GFLOPs [44]. Moreover, mobile/edge GPU may have to undertake other major tasks (such as graphics); while its cloud counterpart is usually used only for DNNs. Hence, it may not always be wise to distribute all DNN inference workload to GPUs on a mobile/edge device. In addition to latency and throughput, energy efficiency has always been another major concern for mobile and edge computing since most mobile devices and some edge devices are battery-powered with limited energy resources. Moreover, as mentioned above, AI applications with DNNs are computationally intensive thus energy hungry. So energy efficiency should be another design goal, which, however, may further complicate the scheduling problem. In short, the DNN scheduling problem studied here is not trivial at all.

In this paper, we aim to develop a co-scheduling framework, which jointly schedules multiple DNN models over heterogeneous hardware with the objective of achieving low-latency, high-throughput and energy-efficient inference. First of all, we perform a preliminary empirical study to gain some insights about the right direction for DNN scheduling over heterogeneous hardware by running some simple experiments on a Google's Android-based Pixel smartphone. We make several interesting findings from our preliminary study: 1) The current practice utilizing single hardware (a CPU or GPU) for DNN inference is inefficient; and a better way is to make the CPU and GPU work concurrently by co-scheduling tasks on both of them. 2) A straightforward scheduling method with a pre-defined fixed policy, e.g., round-robin, is neither high-throughput friendly nor energy-efficient; hence, designing a low-latency, high-throughput and energy-efficient co-scheduling algorithm for DNN inference is quite challenging.

Motivated by these findings, we design and implement a novel online Co-Scheduling framework based on deep REinforcement Learning (DRL), which we call COSREL. The main advantage of COSREL is its ability to leverage heterogeneous computational resources (e.g., CPU, GPU, DSP, etc) to effectively and efficiently support concurrent inference of multiple DNN models on a mobile or edge device. As demonstrated by our experimental results, COSREL can improve the throughput up to 1.8x, reduce the inference latency up to 88%, and consume up to 51% less energy in the scenario with multiple DNN models deployed on the device. Particularly, COSREL has several desirable features. First of all, COSREL achieves significant speedup over the commonly-used methods by efficiently utilizing all heterogeneous computational hardware. Second, COSREL leverages emerging DRL to make dynamic and wise online scheduling decisions for DNN models based on system runtime state. Third, COSREL is capable of making a good tradeoff among latency, throughput and energy efficiency. Last but not the least, COSREL makes no change to given DNN models, and thus preserves their accuracies. In addition, training a DRL agent for on-device inference is challenging (see Section 3.3). We propose a novel device-server co-training algorithm, which makes a device and a server work collaboratively and efficiently to train the DRL agent of COSREL. We summarize our contributions in the following:

1) We conduct a preliminary empirical study for inference with a simple DNN model on an off-the-shelf smartphone, and make several interesting findings, which can serve as a guidance for the design of an efficient co-scheduling algorithm.
2) We present the design and implementation of a novel co-scheduling framework, COSREL, which has several desirable features. Moreover, we propose a novel device-server co-training algorithm to train its DRL agent.
3) We well justify the effectiveness and superiority of COSREL by extensive experiments on off-the-shelf mobile devices with widely-used DNN models.

The rest of the paper is organized as follows. We present our preliminary study in Section 2. The proposed framework, COSREL, is presented in Section 3. We present experimental results and the corresponding analysis in Section 4. We present a literature review about related work in Section 5 and conclude the paper in Section 6.

2 PRELIMINARY STUDY

In this section, through a preliminary study, we discuss the problems of the current practice, which undertakes all inference tasks on single hardware, and the challenges associated with online co-scheduling on mobile and edge devices with heterogeneous hardware.

In a typical AI application scenario (e.g., image classification), a developer first designs a DNN model for this application, trains the model in servers with training data, and then converts the model to fit into mobile/edge devices using an on-device inference framework (such as Tensorflow Lite [57]). When deploying the DNN model to devices, the current practice is to specify a particular hardware, CPU (by default) or GPU (if available), to execute the DNN inference. However, this kind of single-hardware solution does not take advantage of heterogeneous computational resources. To utilize all available computational resources to support DNN inference, we can design a straightforward scheduling solution, e.g., evenly distributing inference tasks to different hardware in a round-robin manner, which, however, is shown to be inefficient. We conducted some experiments on an off-the-shelf smartphone, Google's Pixel 2 XL [45], which runs Android 10 on Qualcomm Snapdragon 835 CPU and Adreno 540 GPU. We used a simple DNN model for continuous image classification on this device, which mainly consists of a few convolutional layers and fully-connected layers. The basic information of this DNN model is summarized in Table 1, where #Cov is the number of convolutional layers with four $3\times3\times32$ and two $3\times3\times64$ filters, respectively, #MaxP is the number of max pooling layers, and the #FC is the number of fully connected layers with 1,000 neurons each.

#Input	#Cov	#MaxP	#FC	#Output
$244 \times 244 \times 3$	6	3	2	1000

TABLE 1: The basic information of a simple DNN

In our experiments, images kept arriving at a rate of 25 Frames Per Second (FPS), which is a typical sampling rate of a smartphone camera. For a given observation period, our goal is to complete as many inference tasks as possible and in the meanwhile, maintain a low inference latency. Here, each inference *task* refers to the process of computing the output from an input image using a given DNN. At the beginning of each decision epoch, the scheduler computes an assignment for all the following inference tasks that will arrive within this epoch. To show the performance of different scheduling methods, we used the following three metrics for comparisons: 1) *throughput*: the number of completed tasks during an observation period; 2) *inference latency (or simply latency)*: the elapsed time from the arrival of a task to the end of the inference; and 3) *energy efficiency index*: the average amount of energy consumption per task. We calculated the throughput and the average inference latency of completed tasks. Since there is no available Android API to directly measure the battery consumption for only one task, we measured the energy drop of the smartphone battery within a certain period using the Android API *BatteryManage* and computed the corresponding energy efficiency index. The same measurement method has been used in [34]. Similar energy efficiency metrics have also been used in the context of cloud computing [26] and in the context of wireless communications [43], [37].

In our experiments, we evaluated three scheduling methods: running all inference tasks only on the CPU (labeled as CPU), only on the GPU (labeled as GPU), and a straightforward round-robin-based co-scheduling method (labeled as Round-Robin). We set the duration of each decision epoch to 200ms and the duration of an observation period to 10s in our experiments. We ran the experiments for 100 observation periods and present the average values in terms of the three metrics in Figures 1, 2, and 3. We make several interesting findings from these results:

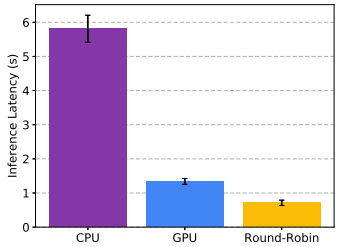


Fig. 1: Average inference latency

Finding 1: Co-scheduling DNN tasks over all computational hardware can significantly reduce the inference latency. The current practice usually leverages single hardware, CPU or GPU, for inference, which turns out to be inefficient due to under-utilization of available computational resources. As we can see from Figure 1, by fully leveraging all computational resources on the device by co-scheduling tasks, even a simple straightforward round-robin co-scheduling algorithm significantly reduces the average inference latency. Specifically, compared to the CPU-only and GPU-only methods, the round-robin method significantly reduces the average inference latency by 87.7% and 46.9%, respectively. This is because it distributes the workload to both the CPU and GPU, which improves resource utilization and thus reduces latency. Note that the GPU-only solution yields unsatisfactory latency because as mentioned above, mobile GPU has much less computing power than those GPUs designed particularly for DNN inference in a cloud; and moreover, GPU is a batch-processing hardware, which usually leads to high throughput but long latency. This finding leads us to believe that co-scheduling tasks over all computational hardware can significantly reduce the inference latency.

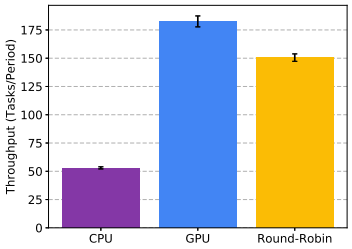


Fig. 2: Throughput

Finding 2: A straightforward co-scheduling method does not yield satisfying throughput. Even though it has been demonstrated that task co-scheduling can utilize both the CPU and GPU to reduce DNN inference latency, a straightforward method, such as Round-Robin, does not yield satisfying throughput, which is shown in Figure 2. Specifically, compared to the GPU-only method, the round-robin algorithm produces 17.5% less throughput on average. This is because heterogeneous hardware usually results in different inference times, distributing too much workload to low-speed hardware (e.g., CPU) may hurt the overall performance. Straightforward scheduling methods, such as Round-Robin, usually follow a pre-defined fixed policy and ignore system state at runtime, which likely leads to unsatisfactory performance too. Hence, we can learn that it is necessary to carefully design an intelligent scheduling algorithm, which can fully utilize all computational resources, and make wise online scheduling decisions based on runtime system state with the objective of achieving low-latency and high-throughput inference.

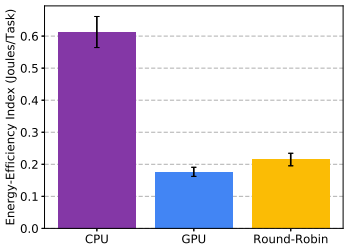


Fig. 3: Energy efficiency index

Finding 3: Achieving energy efficiency is non-trivial. As we can see from Figure 3, the straightforward co-scheduling method, Round-Robin, cannot well balance the performance and energy consumption, and thus leads to 21.6% more energy consumption per task, compared to the GPU-only method. The CPU-only solution is not energy-efficient either. This is because even though it may lead to a low energy dropping rate, it tends to spend a long time to process DNN inference tasks, which is not efficient in terms of energy efficiency index. Therefore, when designing a co-scheduling algorithm for DNN inference, we should carefully and explicitly address energy efficiency in our design.

3 Design and Implementation

In this section, we first present an overview of COSREL and then describe the details of its design and implementation.

3.1 Overview

Motivated by the findings described above, we propose a novel online co-scheduling framework called COSREL. Figure 4 illustrates the architecture of COSREL, which consists of two components:

1) *State Monitor*: It periodically collects runtime state of the system, and reports it to the DRL agent for decision making.
2) *DRL Agent*: It is the core of COSREL, which takes the runtime state as input and applies a DRL-based algorithm to compute a co-scheduling solution.

COSREL works as follows: given a well-trained DRL agent, at each decision epoch, based on the runtime state received from the state monitor, it derives a co-scheduling solution and deploys it to the system. We will discuss how to train the DRL agent in Section 3.3. The desirable features of COSREL is summarized in the following:

1) *Full Utilization of Heterogeneous Hardware:* COSREL fully utilizes all computational resources on heterogeneous hardware to support on-device inference of DNN models.
2) *DRL-based Online Co-Scheduling:* Based on DRL, COSREL makes dynamic and wise online co-scheduling decisions with consideration for system state at runtime.
3) *Good Tradeoff among Latency, Throughput and Energy Efficiency:* COSREL can achieve low-latency, high-throughput and energy-efficient DNN inference by setting the reward of its DRL agent properly.
4) *User/Model Transparency:* COSREL is transparent to users and DNN models, i.e., it makes no change to given DNN models and thus preserves their accuracies.
5) *Complementariness to existing DL frameworks and Model Compression Techniques:* As mentioned above, COSREL can work together with any existing DL framework and/or any model compression technique to further accelerate DNN inference.

We summarize major notations in Table 2 for quick reference.

TABLE 2: Major Notations

Notation	Description
$\mathcal{M}$	A set of DNN models
M	The number of DNN models
$\mathcal{N}$	A set of computational hardware
N	The number of computational hardware
$\mathbf{s}_t, \mathbf{a}_t, r_t$	The state, action and reward at decision epoch t
x_{ij}	The throughput of DNN model i on hardware j
y_{ij}	The average inference latency of DNN model i on hardware j
$Q(\cdot)$	The deep Q-Network (DQN)
$Q'(\cdot), Q''(\cdot)$	The DQN clone and its target network
$\boldsymbol{\theta}, \boldsymbol{\theta}', \boldsymbol{\theta}''$	The sets of weights of $Q(\cdot)$, $Q'(\cdot)$ and $Q''(\cdot)$

3.2 DRL-based Co-Scheduling

In this section, we present the design and implementation of the proposed DRL agent. We consider the following co-scheduling problem: given a set of DNN models $\mathcal{M}$ and a set of computational hardware $\mathcal{N}$, the co-scheduling problem seeks a co-scheduling solution, which assigns each DNN model to one of the computational hardware. Formally, the co-scheduling solution is given by an $M \times N$ matrix $\mathbf{a}$, where $M = |\mathcal{M}|$, $N = |\mathcal{N}|$, and each element $a_{ij} = 1$ denotes that DNN model i is assigned to computational hardware j; otherwise, $a_{ij} = 0$. Note that in

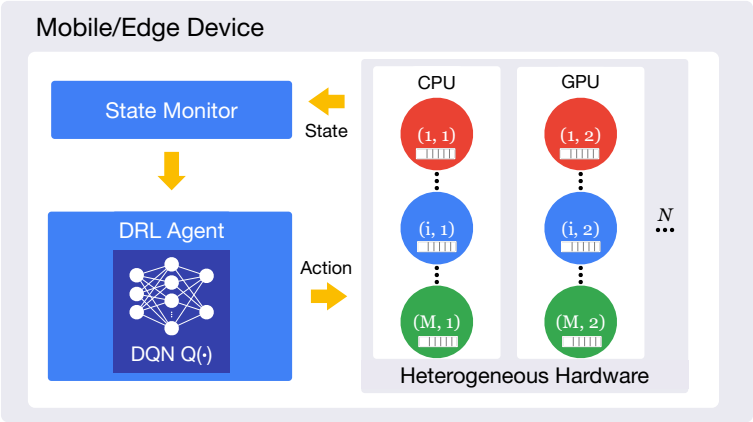


Fig. 4: The architecture of COSREL

a real system, each DNN model is loaded to all hardware in advance. At each decision epoch, *assigning a DNN model i to hardware j means that the inference tasks associated with DNN model i are inserted into the queue corresponding to model-hardware pair (i, j) for processing.* At runtime, we run a thread for each model-hardware pair (i, j), which keeps picking inference tasks (arrive one by one) from the corresponding queue and running them on hardware i. All these threads run concurrently in parallel. In the example illustrated by Figure 4, $N = 2$ and hardware 1 and 2 are the CPU and GPU respectively; each circle represents a thread corresponding to a model-hardware pair (i, j), which maintains a queue with tasks (of DNN model i) assigned to hardware j.

In COSREL, we propose to employ a DRL agent to solve the above scheduling problem. At each decision epoch t, the DRL agent observes system state $\mathbf{s}_t$; then based on its current control policy $\pi(\cdot)$ and state $\mathbf{s}_t$, the DRL agent computes a scheduling solution $\mathbf{a}_t$. In COSREL, we apply a Deep Q-Network (DQN) [40] $Q(\cdot)$ to deriving the control policy. Basically, DQN (i.e., $Q(\cdot)$) is a DNN that takes the current system state and an action (i.e., a scheduling solution) as input and outputs a continuous value, which is called Q-value. The Q-value can be considered as a score which tells how well to take an action $\mathbf{a}$ at state $\mathbf{s}_t$. The control policy π is defined as selecting action $\mathbf{a}_t$ with the highest Q-value at state $\mathbf{s}_t$, i.e., $\pi(\mathbf{s}_t) : \mathbf{a}_t := \text{argmax}_{\mathbf{a}} Q(\mathbf{s}_t, \mathbf{a})$. We present the definitions of state and action for the DRL agent in the following.

State: the state of the DRL agent is basically the runtime statistics, which are collected periodically from the device. In our design, the state consists of the following three features: 1) the length of the task queue of each model-hardware pair; 2) the inference time of each model on every computational hardware; 3) the resource usage (in percentage) of each computational hardware.

Note that as mentioned above, each inference *task* refers to the process of computing the output from input (e.g., an image) using a given DNN. So the length of a task queue is the number of tasks waiting in that queue for being processed by the corresponding hardware. We only keep necessary features from the runtime statistics in our design such that we will not introduce significant overhead to the DRL agent. We have tried different combinations of the available features and found that the above three features are sufficient for capturing the essence of the system state at runtime.

Action: An action of the DRL agent is rather straightforward, which is defined as a co-scheduling solution $\mathbf{a}$, which is a $M \times N$ matrix. Each of its element $\mathbf{a}_{ij} = \{0, 1\}$ specifies if model i is *assigned* to hardware j as described above.

Reward: COSREL is so flexible that it can accommodate different application-specific and/or device-specific needs by setting its reward function properly. There are usually two cases: the first case emphasizes performance (in terms of latency and throughput); and the second case cares about both performance and energy efficiency.

Specifically, for those real-time AI applications (such as object detection, object tracking, etc) or for those edge devices with continuous power supply, the major concern of the design is certainly performance (in terms of latency and throughput). In this case, we need to maximize the system performance by jointly addressing both latency and throughput in the reward function. In our design, instead of directly maximizing throughout (which likely leads to significant unfairness) or minimizing latency, we choose to follow the widely-used α-fairness model [6], [54], [60], [61] to address both latency and throughput, and the reward is defined accordingly as:

$$r = \sum_{i=1}^{M} \sum_{j=1}^{N} (U(x_{ij}) - \rho \cdot U(y_{ij})), \tag{1}$$

where $U(\cdot)$ is a utility function, x_{ij} and y_{ij} are the throughput and the average inference latency of DNN model i on the hardware j during the last decision epoch, respectively; and ρ is a scaling factor used to balance their relative importance and $\rho := 0.1$ in our implementation. According to the α-fairness model, the utility function is $U_\alpha(x) = (\frac{x^{1-\alpha}}{1-\alpha})$. For $\alpha > 0$, $U_\alpha(x)$ increases monotonically with x. α can be used to tradeoff fairness and performance. In our design, we choose to set $\alpha := 1$, then $U(x) = \log x$, which is considered to make a good tradeoff between performance and fairness thus has been widely used for resource allocation.

In addition, for those mobile devices (such as smartphones, pads and wearable devices) that are usually battery-

powered, energy consumption is one of major concerns and should play a key role in the reward function. In this case, we can define another reward function, which includes both performance-related factors (i.e., throughput and latency) and energy consumption. Formally, the reward is defined as:

$$r = \sum_{i=1}^{M} \sum_{j=1}^{N} (U(x_{ij}) - \rho \cdot U(y_{ij}) - \sigma \cdot e) \quad (2)$$

where e is the energy consumption since the last sampling; and σ is a scaling factor and $\sigma := 0.01$ in our implementation. Note that our framework provides an ability to express different optimization objectives via choice of the reward function (e.g., throughput, latency, and energy consumption). In practice, users could select different reward functions, like the ratio of these two utility functions. It is believed that using such reward functions would achieve similar performance as long as they have the same optimization objectives, i.e., maximizing the throughput and minimizing the latency.

Implementation Details: We implemented COSREL on Android 10 [3], which is one of the most popular OSs for mobile and edge devices. To collect the runtime statistics of the system, including necessary information for state and reward, we developed a state monitor, which runs as daemon process in the Android system, periodically broadcasting those information. The DRL agent registers the broadcasting and receives the information from the state monitor. The information will be used as the input to the DQN in the DRL agent for decision making and to form transition samples for training (which will be discussed next). We used Tensorflow Lite [57] for our implementation, which allows us to perform DNN inference using different types of computational hardware on the Android-based device, including CPU and GPU. Tensorflow Lite uses the $Interpreter$ class to warp all the functions needed by DNN inference, including configuring the hardware, loading DNN models, executing operations with input data, and accessing results. It uses a CPU to support inference of DNN models by default. To perform inference on a GPU, we need to select the $GpuDelegate$ option when initializing the $Interpreter$. To fully leverage all the computational resources, we warped each DNN model as a separate thread, which maintains a task queue and multiple $Interpreter$ (each of them is configured for certain hardware). At runtime, DNN inference tasks are submitted to the task queue with input data, and our DRL agent tells which hardware are used to perform the corresponding DNN inference. COSREL keeps running all threads to process the corresponding tasks from their task queues.

3.3 Device-Server Co-Training

A common practice for using a DNN model (e.g., a Convolution Neural Network (CNN)) on a mobile/edge device is to train the model on a (or multiple) server with given training data, and then deploy it on the device only for inference. However, this approach does not work for a DRL agent, i.e., it cannot be well trained only on a server in an offline manner, since data (i.e., transition samples) for training the agent need to be collected continuously via interactions with the mobile/edge device. On the other hand, it is also difficult to train a DRL agent only on a mobile/edge device. As we all know, complex mathematical operations (e.g., calculating gradients and backpropagation) are needed to be performed during the training process, which, however, are not supported by most mobile DL frameworks such as TensorFlow Lite and Pytorch Mobile. Moreover, training a DRL agent only on a mobile/edge device may take a long time to converge due to its very limited computational resources.

Hence, we propose to let the server and mobile/edge device to work collaboratively and efficiently to train the DRL agent, which we call *device-server co-training*. During the co-training process, the on-device DRL agent periodically sends transition samples to the server, where the training algorithm performs backpropagations and updates the DQN of the DRL agent.

For the completeness of the presentation, we formally present the device-server co-training algorithm as Algorithm 1. According to this algorithm, we have an on-device DRL agent with its DQN $Q(\cdot)$ on the device; and a clone $Q'(\cdot)$ of the on-device DQN and its corresponding target network $Q''(\cdot)$ on the server. First the algorithm randomly initializes the weights of $Q(\cdot)$, and copies its weights to its clone $Q'(\cdot)$ and the target network $Q''(\cdot)$ (Lines 1 and 2). We choose a random action as the starting point. After executing the scheduling action, the state transits to $\mathbf{s}_{t+1}$, and the agent observes the reward r_t from the system. Then the DRL agent sends transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ to the server for updating the DQN clone $Q'(\cdot)$(on the server).

On the server, whenever receiving a transition sample from the device, the server stores it into its replay buffer. Then the algorithm samples a mini-batch of transition samples from the replay buffer and then calculates the target value with a discount factor γ (Line 13). In our implementation, $\gamma := 0.99$. The loss function used for training is defined as the mean square error of the current output of DQN and the target value (Line 14). The DQN clone $Q'(\cdot)$ can then be updated using any training algorithm (such as Stochastic Gradient Descent (SGD)) on the server. Note that the learning objective here is to find the best policy π that maximizes the cumulated discounted reward over a long decision period. We apply a soft-update to slowly updating the target network $Q''(\cdot)$ with a scaling factor τ (Line 15) and $\tau := 0.01$ in the implementation. The target network and experience replay buffer here are both used to improve learning stability [40], [31]. Similar as in [40], we apply the ϵ-greedy policy for exploration (Line 16). Specifically, we take an random action with ϵ probability, otherwise, we take an action derived by the control policy π. ϵ decays as the training progresses. Note that we choose an action using the ϵ-greedy policy on the server and send it to the device for execution because the server has the most updated DQN $Q'(\cdot)$. To minimize the communication overhead, instead of synchronizing $Q(\cdot)$ with $Q'(\cdot)$ in every step, we do it only once in the end (Line 20). In every step, we only need to send the action generated using the ϵ-greedy policy to the DRL agent on the device for execution. Since the DRL agent does not need the target network to derive an action, it is deployed only on the server. The interactions between the device and the server are only needed during

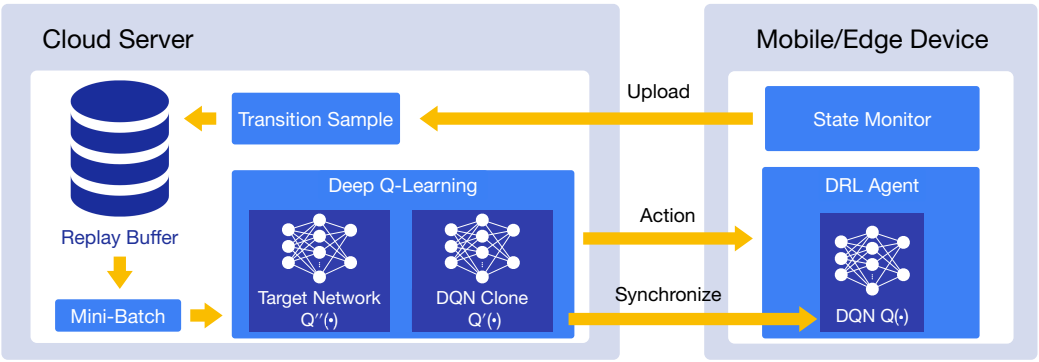


Fig. 5: Device-server co-training

Algorithm 1: Device-Server Co-Training

Input: The number of training epochs T, the size of a mini-batch K, the exploration ratio ϵ, replay buffer $\mathcal{B}$, DQN $Q(\cdot)$ with weights $\boldsymbol{\theta}$, its clone $Q'(\cdot)$ with weights $\boldsymbol{\theta}'$, and its target network $Q''(\cdot)$ with weights $\boldsymbol{\theta}''$.

Randomly initialize the weights $\boldsymbol{\theta}$ of DQN $Q(\cdot)$;
$\boldsymbol{\theta}' := \boldsymbol{\theta}$; $\boldsymbol{\theta}'' := \boldsymbol{\theta}$;
foreach *observation period* **do**
  Send a random action $\mathbf{a}_1$ to the DRL agent;
  $t := 1$;
  while *decision epoch* $t < T$ **do**
    /**Execution on the device**/
    Receive action $\mathbf{a}_t$ from the server;
    Execute the action and observe the reward r_t;
    Receive system state $\mathbf{s}_{t+1}$ from the state monitor;
    Send transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ to the server;
    /**Training on the server**/
    Receive the transition sample and store it into replay buffer $\mathcal{B}$;
    Sample K transition samples $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from $\mathcal{B}$;
    Compute the target value for each transition sample $y_i := r_i + \gamma \max_{\mathbf{a}} Q''(\mathbf{s}_{i+1}, \mathbf{a})$;
    Update the weights $\boldsymbol{\theta}'$ of the DQN clone with the loss function $\mathcal{L} = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q'(\mathbf{s}_i, \mathbf{a}_i))^2$;
    Update the weights of target network $\boldsymbol{\theta}'' := \tau\boldsymbol{\theta}' + (1-\tau)\boldsymbol{\theta}''$;
    Select an action with the ϵ-greedy policy

$$\mathbf{a}_{t+1} := \begin{cases} \text{a random action} & \text{with } \epsilon \text{ probability;} \\ argmax_a Q'(\mathbf{s}_{t+1}, \mathbf{a}) & \text{otherwise;} \end{cases}$$

    Send action $\mathbf{a}_{t+1}$ to the DRL agent;
  end
end
Synchronize DQN $Q(\cdot)$ with its clone $Q'(\cdot)$: $\boldsymbol{\theta} := \boldsymbol{\theta}'$;

the training process. Once the DRL agent is well trained, it makes scheduling decisions based only on its own DQN $Q(\cdot)$.

Implementation Details: We implemented the server-side training algorithm presented above as a training server using Flask [9] and PyTorch [48] and deployed it on a desktop computer. As mentioned above, the DRL agent and the device-side training algorithm were implemented using TensorFlow Lite. The training server also works as a web server, waiting for HTTP requests from the DRL agent on the device. At the beginning of each decision epoch, DRL agent constructs a POST request, fills in a transition sample in the JSON format, and sends it to the server. Upon receiving the request, the server parses the JSON string and stores the transition sample into its replay buffer. During the training, the device continuously interacts with the cloud server through uploading transition samples and downloading actions. The weights synchronizing only happens at the end of training. Therefore, the co-training communication overhead only depends on the decision frequency and the size of a transition sample (i.e., a pair of state, action, reward, and new state value). As we may notice, the size of both state and action is linearly increasing with the number of deployed DNN models on the device, which is usually in the range of one to several. Due to the resource limitation of devices, it is infeasible to deploy too many DNN models on the device. In practice, the users could adjust the decision frequency according to their application scenarios. For example, for the energy-sensitive case, they can set a longer decision epoch to reduce communication overhead; for the performance-first case, a shorter decision epoch can lead to more precise scheduling.

In our implementation, we chose a commonly-used two-layer fully-connected neural network (with ReLu as the activation function) to serve as the DQN, which includes 256 and 128 neurons in the first and second layer respectively. We applied the Adam [27] algorithm to updating the weights of DQN, and set the learning rate to 0.005. While we believe a finer-grain setting for different application scenarios, like more complex neural network or more layers/neurons, might potentially enhance the learning ability of DQN, leading to a better performance, it also leads to extra inference and training overload.

4 Performance Evaluation

In this section, we first present the common experimental setup and then introduce the three testing scenarios. After that, we present the experimental results and the corresponding analysis.

4.1 Experimental Setup

We performed extensive experiments on an off-the-shelf device, Google Pixel 2 XL running Android 10, which has Qualcomm Snapdragon 835 CPU and Adreno 540 GPU [45]. For device-server co-training, the server side program ran on a desktop with Ubuntu 18.04, Intel quad 2.6GHz CPU and 32 GB RAM. To evaluate the performance of different scheduling algorithms, we used average inference latency, throughput and energy efficiency index as the metrics, which have all been introduced in Section 2.

For a comprehensive evaluation, we compared COSREL with three widely-used baselines:

1) **CPU-only (labeled as "CPU"):** This method only uses CPU for the inference of DNN models.
2) **GPU-only (labeled as "GPU"):** This method only uses GPU for the inference of DNN models.
3) **Round-Robin (labeled as "Round-Robin"):** This method evenly distributes DNN inference tasks to CPU and GPU in a round-robin manner.
4) **Basic COSREL (labeled as "COSREL-P"):** This is COSREL, whose DRL agent was trained using the basic reward function (Equation (1)).
5) **Energy-efficient COSREL (labeled as "COSREL-E"):** This is COSREL, whose DRL agent was trained using the energy-efficient reward function (Equation (2)).

4.2 Experimental Results and Analysis

To evaluate the performance of COSREL, we designed three different testing scenarios. In the first scenario, we deployed a single DNN model on the device to evaluate whether COSREL can fully utilize all computational hardware to speed up the DNN inference. In the second scenario, we deployed two different DNN models on the device, aiming to evaluate whether COSREL can effectively and efficiently utilize heterogeneous computational resources to support multiple DNN models. In the third scenario, we had two real applications, facial expression recognition and facial landmarks detection, which consist of three correlated DNN models. We want to evaluate if COSREL can still work well in such a complex environment with multiple correlated DNN models and heterogeneous computational hardware. In all these three scenarios, we set a decision epoch to 200ms and an observation period to 10s, which are similar as in our preliminary study, and we trained the DRL agent for $20,000$ decision epochs. When completing the training, we tested COSREL for 100 observation periods. Then we calculated the averages of throughput, average inference latency, energy efficiency index, and their corresponding standard deviations (shown by error bars in Figures 6, 8, and 9). We found these settings are sufficient for the DRL agent to observe the system state, make the action decision, and learn a good control policy in all these scenarios. While a shorter epoch length or larger number of training epochs may lead to a finer-grain scheduling or better performance, the system will suffer from extra overhead of co-training communication.

Scenario 1: In this scenario, we deployed a widely-used DNN model, i.e., MobileNet [22] to the smartphone. We set the arrival rate of images to 53FPS. From the results shown in Figure 6, we can make the following observations:

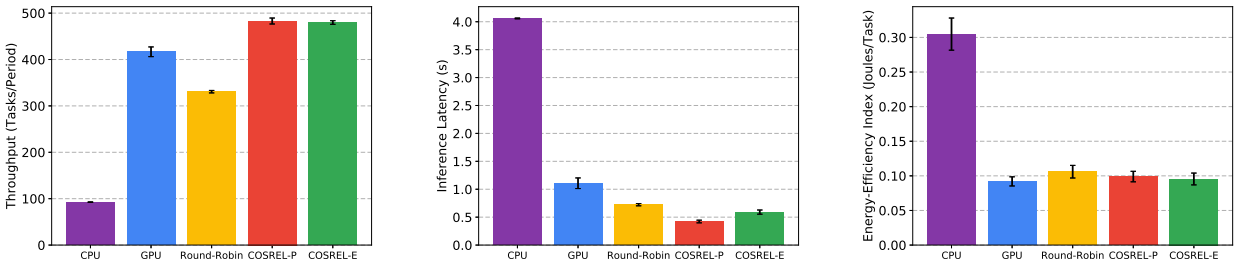
1) It is not surprising to see that the CPU-only method always has the worst performance. This is because CPU is designed for general-purpose computing, but the inference of DNN models needs a lot of floating-point vector/matrix calculations. During the experiment, when the CPU-only method was used, we observed that a long queueing delay dominated the inference latency. Hence, compared to a GPU, it usually takes more time for a CPU to execute the inference of DNN models. Specifically, compared to CPU-only method, the GPU-only method reduces the average inference latency by 72.72% and boosts the throughput by over 3.5x on average.

2) Although GPU can accelerate the inference of DNN models, the GPU-only method, however, is not efficient either. When too many inference tasks are assigned to the GPU, the length of its queue increases and then the inference latency rises sharply. By co-scheduling tasks on both the CPU and GPU, we can fully utilize all the computational resources, and avoid abusing the GPU. As we can see from Figure 6(b), even a simple straightforward co-scheduling method, Round-Robin, can help reduce the average inference latency. Specifically, the round-robin method reduces the average inference latency by 82.18% and 34.65% respectively on average, compared to the CPU-only method and the GPU-only method.

3) Although heterogeneous hardware can potentially speed up DNN inference, without a carefully-designed co-scheduling algorithm, we may fail to harness the real power of co-scheduling and parallel computing. As we can see from Figure 6(a), the round-robin method delivers lower throughput than the GPU-only method. This observation confirms that it is necessary to design an intelligent co-scheduling algorithm, which can fully utilize all computational resources based on the system state at runtime.

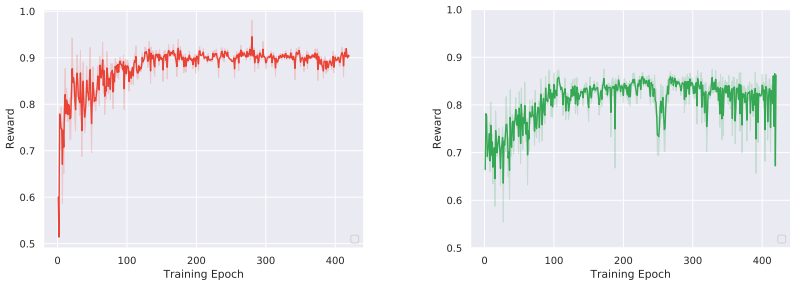
4) Although the goal of COSREL-P is not directly set to minimize the average inference latency or to maximize the throughput, COSREL-P still delivers satisfying performance in terms of both metrics. Specifically, compared to the CPU-only and GPU-only, and round-robin methods, COSREL-P significantly improves the throughput by 4.2x, 15.92%, and 46.06% respectively on average; and significantly reduces the average inference latency by 89.61%, 61.89%, and 41.68% respectively. These results well justify the effectiveness and superiority of COSREL.

5) As we can see from Figure 6(c), both COSREL-P and COSREL-E perform well in terms of energy efficiency. Even though energy efficiency is not directly addressed in its reward function, COSREL-P can still offer satisfying performance. Specifically, compared to the CPU-only and round-robin methods, COSREL-P leads to 67.52% and 6.63% less energy consumption per task respectively on average. As energy is explicitly addressed in its reward function, COSREL-E achieves better energy efficiency than



(a) Throughput (b) Average inference latency (c) Energy efficiency index

Fig. 6: Scenario 1: the performance of different methods with a single DNN model



(a) The reward of COSREL-P (b) The reward of COSREL-E

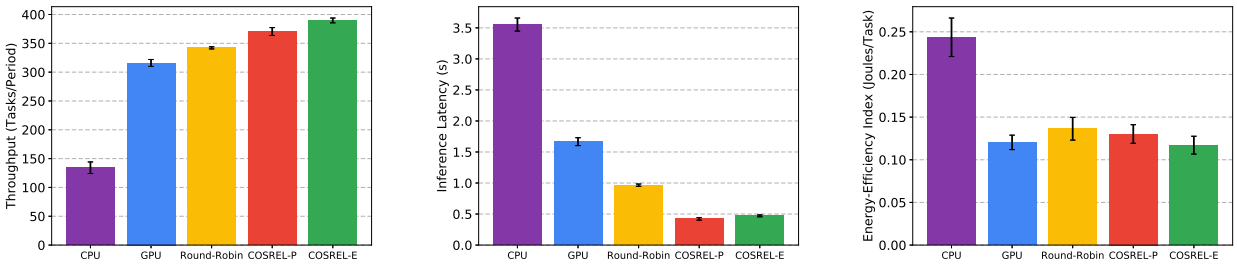
Fig. 7: Scenario 1: the reward of COSREL during the device-server co-training

COSREL-P with almost the same throughput and a slight increase on average inference latency. These results illustrate that our proposed framework is flexible and can be tuned to accommodate different needs. If energy consumption is the major concern (e.g., battery-powered mobile devices), we can use COSREL-E. Otherwise, if the performance has the highest priority, we can choose COSREL-P. Note that in this simple scenario, the GPU-only method offers slightly better energy efficiency than COSREL. However, in the next two scenarios (that are more realistic and complicated), COSREL leads to noticeable improvements over the GPU-only method in terms of energy efficiency index, which can be observed from Figures 8 and 9.

6) We show how the reward of the DRL agent improves during the training in Figure 7. Note that we normalized the reward values for better presentation. At the very beginning, the DRL agent has a high probability to randomly explore possible actions instead of taking actions derived by the DQN. Therefore, we observed large fluctuations on the reward. As the training progresses, the DRL agent gradually finds good solutions that lead to larger rewards. Finally, the reward converges to relatively high values, which indicates that the DRL agent is well-trained and ready for online deployment. Note that although the DRL agent stabilizes after a certain period, there are still small fluctuations on the reward. This is because the system is highly dynamic. Specifically, the inference time of each DNN model and the workload on each hardware are highly time-variant, which may affect the decision making of COSREL. However, the fluctuations are quite insignificant, which indicates that COSREL can consistently produce good co-scheduling solutions once it stabilizes.

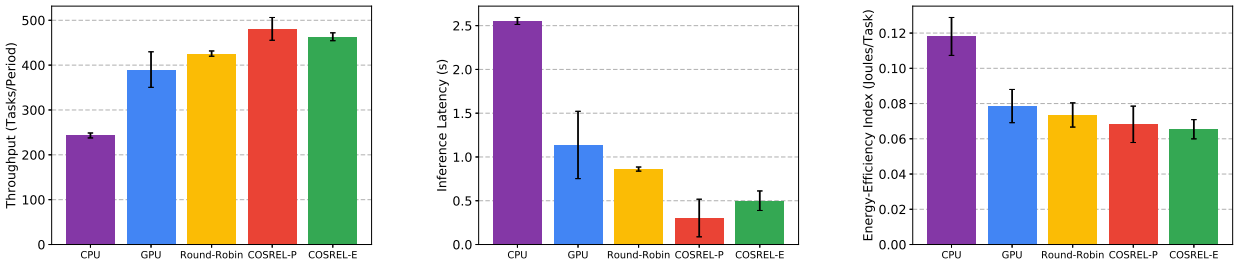
Scenario 2: It is quite common to have multiple DNN models on a device for supporting different DL applications. In this scenario, we deployed two different widely-used DNN models (i.e., MobileNet [22] and SqueezeNet [24]) on the smartphone. Since they ran on the same device, they competed for the computational resources in the hardware. We set the arrival rate of images to 25FPS in this scenario. The corresponding results are presented in Figure 8. As we can see, in this scenario, both COSREL-P and COSREL-E deliver much better performance than all the baselines in terms of both throughput and latency. Specifically, on average, compared to the CPU-only, GPU-only and round-robin methods, COSREL-P improves the throughput by 1.8x, 17.24% and 8.27% respectively; and it reduces the average inference latency by 88.10%, 74.63% and 56.22%. As for energy consumption, COSREL-E turns out to be the best solution. Specifically, compared to the three baselines and COSREL-P, COSREL-E consumes 51.91%, 2.71%, 14.10%, 10.05% less energy per task respectively on average. These results illustrate that COSREL is capable of handling the co-scheduling problem in the case of multiple DNN models.

Scenario 3: In this scenario, we tested the performance of COSREL in a more complicated case with three correlated models. Specifically, we conducted the experiments with two practical applications: facial expression recognition [56] and facial landmarks detection [55]. In the facial expression recognition application, a face detection DNN model first takes an image as input and detects the human face shown in the image. Then a facial expression recognition model is applied to classifying the detected face into 7 categories of expressions, including angry, disgust, fear, happy, sad, surprise and neutral. In the facial landmarks detection appli-



(a) Throughput (b) Average inference latency (c) Energy efficiency index

Fig. 8: Scenario 2: the performance of different methods with multiple DNN models



(a) Throughput (b) Average inference latency (c) Energy efficiency index

Fig. 9: Scenario 3: the performance of different methods with multiple correlated DNN models

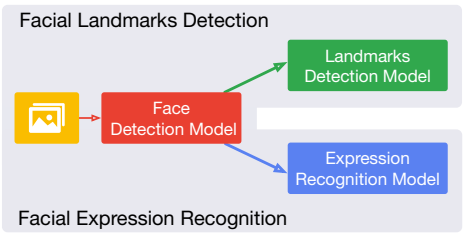


Fig. 10: The structure of the two correlated AI applications

cation, a face detection model is first applied to detecting the human face from an input image. Then a facial landmarks detection model is used to detect 5 key points from the detected face, including the positions of left and right eyes, nose, left corner and right corner of the mouth. Obviously, two applications share a common face detection model. We chose MobileNet-SSD [33]) as the backbone network for face detection to support these two applications. The structure of the models used in this scenario is illustrated in Figure 10. Through the model decomposition and layer-wise model sharing, we constructed a processing pipeline, avoiding duplicate inference with the backbone network and thus improving the overall efficiency for inference. We set the arrival rate of images to 25FPS in this scenario. From the results shown in Figure 9, we can make the following observations:

1) As expected, COSREL-P offers the best performance in terms of both throughput and latency in this scenario. Specifically, on average, COSREL-P significantly improves the throughput by 97.74%, 23.26%, 12.95% respectively and significantly reduces the average inference latency by 88.14%, 73.38%, 64.87% respectively over the three baselines. These results well justify the superiority of COSREL-P in terms of throughput and latency.

2) Even though COSREL-P does not explicitly consider energy efficiency, it still outperforms all the baseline methods. For example, COSREL-P uses 42.24% less energy per task than the CPU-only method on average. By sacrificing throughput and latency a little bit, COSREL-E further improves the energy efficiency by 4.09% over COSREL-P. Note that COSREL-E still outperforms all the baselines in terms of throughput and latency. Specifically, COSREL-E improves the average throughput by 90.50%, 18.75%, 8.82% respectively and reduces the average inference latency by 80.40%, 55.99%, 41.92% respectively.

These results demonstrate that both COSREL-P and COSREL-E work very well in the complicated case with multiple correlated DNN models.

5 Related Work

In this section, we present a comprehensive review for related work and point out the differences.

Inference Frameworks for DNNs: Extensive efforts have been made to develop efficient DL frameworks to support on-device DNN inference in both industry and academia. Two widely-used frameworks are Tensorflow Lite [57] presented by Google and PyTorch Mobile [47] presented by Facebook. Both of them support a number of mathematical operations needed by the inference of common DNN models, and they can be deployed on a wide range of mobile and

edge devices with heterogeneous hardware (e.g., CPU, GPU and DSP). Other examples include NCNN [58] presented by Tencent, MNN [42] presented by Alibaba, and TVM presented by Chen *et al.* [7], which focus on a smaller subset of frequently-used mathematical operations. However, they have a more optimized and lightweight design, which can further improve the inference efficiency.

Model Compression for DNNs: To reduce the computational complexity of DNN models while still preserving their high accuracies, DNN model compression techniques, such as weight pruning, quantization and knowledge distillation, have also attracted a lot of research attention recently. In a pioneer work [14], Han *et al.* proposed to reduce the storage and computation required by a DNN model by pruning its unimportant connections (i.e., weights). In a follow-up work [15], they proposed to further compress DNN models with weight pruning, trained quantization and huffman coding. Other important pruning methods include structured pruning [59], [38], [67], channel pruning [17] and AutoML-based pruning [18], [36]. Quantization has been widely used as an effective and direct approach to reduce the size of a DNN model by reducing the number of bits required to represent its weights. Representative quantization methods include the most extreme binary neural networks [23], [32], [49], ternary weight networks [30] and conservative low-bit quantization (e.g., 4-bit and 8-bit) [4], [21], [46] Besides, knowledge distillation represents another effective but different method for DNN model compression. Representative works along this line include [5], [19], [51], [65].

These related works tackled the challenge of deploying DNN models on mobile and edge devices from the application/model perspective. However, the proposed approach overcomes this challenge from the system perspective, which is transparent to DNN models and is complementary to these model compression methods.

Runtime Optimization for DNNs on Mobile and Edge Devices: Recently, there has been a growing interest on runtime optimization for DNNs on mobile and edge devices. Lane *et al.* [28] proposed DeepX, a software accelerator for DL model executions on mobile devices. DeepX first performs a runtime layer compression on a given DNN model to control the memory computation and energy consumption, then it decomposes the DNN model into unit-blocks of various types and schedules them on heterogeneous hardware (e.g., CPU and GPU) for efficient on-device inference. Yao *et al.* [64] presented a unified approach called DeepIoT to compress DNN models for sensing applications. DeepIoT compresses neural network structures into smaller dense matrices by finding the minimum number of non-redundant hidden elements, such as filters and dimensions required by each layer, while keeping the performance of sensing applications the same. Fang *et al.* [8] proposed NestDNN, which prunes a DNN model into a set of descendent models, each of which offers a unique resource-accuracy trade-off. At runtime, it dynamically selects a DNN model with the best resource-accuracy tradeoff to fit available resources in the system. Liu *et al.* [35] proposed a usage-driven selection framework, called AdaDeep, to automatically select a combination of compression techniques for a given DNN model, which leads to an optimal balance between user-specified performance goals (e.g., latency and energy) and resource constraints.

Another line of research has addressed the problem of scheduling DNN models among mobile/wearable devices, edge computing nodes and cloud servers. Han *et al.* [13] evaluated a variety of model optimization techniques to balance the resource usages in terms of memory, computation and accuracy. Then they proposed a framework called MCDNN to automatically optimize DNN models while conforming to the resource specification and assign DNN models to run either on a cloud or on a mobile device. Kang *et al.* [25] presented Neurosurgeon, a system that automatically partitions DNN models at the granularity of layers and assigns them to run on a mobile device or cloud for the best latency and energy consumption tradeoff. Zhao *et al.* [68] proposed ECRT, an edge computing system for real-time object tracking on resource-constrained devices. By intelligently partitioning DNN models into two parts, which are executed locally on an IoT device or on an edge server, ECRT minimizes the power consumption of IoT devices while meeting the user requirement on end-to-end delay. Xu *et al.* [62] proposed DeepWear, which focuses on applying DNN models on wearable devices. DeepWear offloads DNN inference tasks from a wearable device to its paired hand-held device through local network connectivity (e.g., Bluetooth). Zeng *et al.* [66] proposed Boomerang, an on-demand cooperative DNN inference framework for edge systems in an Industrial Internet of Things (IIoT) environment. Boomerang first reshapes the amount of DNN computation via an early-exit mechanism to reduce the total runtime of DNN inference, and then it segments the DNN model between IIoT devices and an edge server to achieve the DNN inference immediacy.

In addition, Georgieve *et al.* [11] presented LEO, a scheduler designed to maximize the performance of multiple continuous mobile sensing applications by making use of the domain-specific signal processing knowledge to distribute sensor processing tasks to heterogeneous computational resources (e.g., CPU, GPU, DSP and cloud). Zhou *et al.* [69] proposed S^3DNN, a system that supports DNN-based real-time object detection workloads on GPUs in a multi-tasking environment, while simultaneously improving real-time performance, throughput, and GPU resource utilization. Yang *et al.* [63] proposed a framework to improve the number of simultaneous camera streams for object detection on embedded devices without significantly increasing per-frame latency or reducing per-stream accuracy by applying a combination of techniques, including parallelism, pipelining, and the merging of per-camera images.

Unlike those methods proposed in [8], [28], [35], [64], which applied model compression techniques to reduce the model size and speedup model inference, COSREL represents a complementary and transparent solution, which does not change DNN models thus preserves their accuracies. Unlike [13], [25], [62], [66], [68], which addressed the problem of offloading DNN models (or part of models) to the cloud, we, however, focus on the on-device inference without the help of cloud servers, and consider a mathematically different problem of scheduling tasks on heterogeneous hardware. In addition, we study DNN inference on mobile and edge devices in general here, while some related

works [11], [63], [68], [69] targeted at specific applications or models with application-specific goals and requirements.

6 Conclusions

In this paper, we presented COSREL to fully leverage computational resources on heterogeneous hardware using DRL to effectively and efficiently support concurrent inference of multiple DNN models on a mobile or edge device. COSREL has several desirable features, including full utilization of heterogeneous hardware, DRL-based online co-scheduling, good tradeoff among latency, throughput and energy efficiency, user/model transparency, and complementariness to existing DL frameworks and model compression techniques. We also proposed a novel and efficient device-server co-training algorithm for COSREL. We implemented COSREL on an off-the-shelf Android smartphone, and conducted extensive experiments with various testing scenarios and widely-used DNN models to compare it with three commonly-used baselines. It has been shown by the experimental results that 1) COSREL consistently and significantly outperforms all the baselines in terms of both throughput and latency; and 2) COSREL is generally superior to all the baselines in terms of energy efficiency. Although DRL seems a promising technique for enabling co-scheduling for DNN models on mobile and edge devices with heterogeneous hardware, there are still many barriers to its real-world application. For example, training with less data is always a desired feature in practical edge and mobile scenarios, which is also an active research topic in the DRL community. Some new techniques, like transfer learning [70] and meta learning [10], can help to further improve the training efficiency and adaptability of DRL through learning from previous knowledge instead of learning from scratch. Moreover, our device-server co-training framework can be further extended to support distributed training with multiple devices. More efficient learning frameworks, like Asynchronous Advantage Actor Critic (A3C) [41] and distributed prioritized experience replay [20], can be applied to improve the wall-clock training time and final performance. Besides, our COSREL follows the typical DRL setting that aims to learn a best policy to maximize the cumulated discounted reward (i.e., a weighted sum of throughput, latency, and energy consumption) over a long decision period. However, it is still an open issue to simultaneously optimize multiple correlated or even conflicting objectives in DRL, which is known as Multi-Objective Reinforcement Learning (MORL) [52]. These topics are beyond the scope of this paper, we leave them for future research.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, and *et al.*, Tensorflow: large-scale machine learning on heterogeneous distributed systems, *arXiv preprint*, 2016, arXiv:1603.04467.

[2] Adreno: en.wikipedia.org/wiki/Adreno

[3] Android 10: android.com/android-10/

[4] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, Scalable methods for 8-bit training of neural networks, *NeurIPS'18*, pp. 5145–5153.

[5] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, Learning efficient object detection models with knowledge distillation, *NeurIPS'17*, pp. 742–751.

[6] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, PCC: Re-architecting congestion control for consistent high performance, *USENIX NSDI'15*, pp. 395–408.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, and *et al.*, {TVM}: an automated end-to-end optimizing compiler for deep learning, *arXiv preprint*, 2015, arXiv:1512.01274.

[8] B. Fang, X. Zeng, and M. Zhang, NestDNN: resource-aware multi-tenant on-device deep learning for continuous mobile vision, *ACM MobiCom'18*, pp. 115–127.

[9] Flask: palletsprojects.com/p/flask/

[10] C. Finn, P. Abbeel, and S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks, *ICML'17*, pp. 1126–1135.

[11] P. Georgiev, N. Lane, K. Rachuri, and C. Mascolo, LEO: scheduling sensor inference algorithms across heterogeneous mobile processors and network resources, *ACM MobiCom'16*, pp. 320–333.

[12] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning, *MIT Press*, 2016.

[13] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, MCDNN: an execution framework for deep neural networks on resource-constrained devices, *ACM MobiSys'16*.

[14] S. Han, J. Pool, J. Tran, and W. Dally, Learning both weights and connections for efficient neural network, *NeurIPS'15*, pp. 1135–1143.

[15] S. Han, H. Mao, and W. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, *ICLR'16*.

[16] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, *IEEE CVPR'16*, pp. 770–778.

[17] Y. He, X. Zhang, and J. Sun, Channel pruning for accelerating very deep neural networks, *IEEE ICCV'17*, pp. 1389–1397.

[18] Y. He, J. Lin, Z. Liu, H. Wang, L. Li, and S. Han, AMC: AutoML for model compression and acceleration on mobile devices, *ECCV'18*, pp. 784–800.

[19] G. Hinton, O. Vinyals, and J. Dean, Distilling the knowledge in a neural network, *arXiv preprint*, 2015, arXiv:1503.02531.

[20] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. V. Hasselt, and D. Silver, Distributed prioritized experience replay, *ICLR'18*.

[21] L. Hou and J. Kwok, Loss-aware weight quantization of deep networks, *ICLR'18*.

[22] A. 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*, 2017, arXiv:1704.04861

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, Binarized neural networks, *NeurIPS'16*, pp. 4107–4115.

[24] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer, SqueezeNet: alexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size, *arXiv preprint*, 2016, arXiv:1602.07360

[25] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, Neurosurgeon: collaborative intelligence between the cloud and mobile edge, *ACM SIGARCH'17*, Vol. 45, No. 1, 2017, pp. 615–629.

[26] G. Katsaros, J. Subirats, J. Fitó, J. Guitart, P. Gilet, and D. Espling, A service framework for energy-aware monitoring and VM management in clouds, *Future Generation Computer Systems*, Vol. 29, No. 8, 2013, pp. 2077–2091.

[27] D. Kingma and J. Ba, Adam: a method for stochastic optimization, *ICLR'15*.

[28] N. 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, *ACM/IEEE IPSN'16*, pp. 1–12.

[29] Y. LeCun, Y. Bengio, snd G. Hinton, *Nature*, Vol. 521, No. 7553, 2015, pp. 436–444.

[30] F. Li, B. Zhang and B. Liu, Ternary weight networks, *NIPS'16*.

[31] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, Continuous control with deep reinforcement learning, *ICLR'16*.

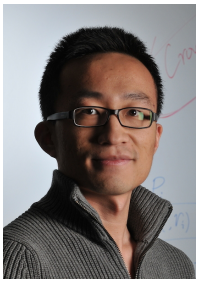
[32] X. Lin, C. Zhao, and W. Pan, Towards accurate binary convolutional neural network, *NeurIPS'17*, pp. 345–353.

[33] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, and A. Berg, SSD: single shot multibox detector, *ECCV'2016*, pp. 21–37.

[34] L. Liu, S. Isaacman, A. Bhattacharjee, and U. Kremer, POSTER: exploiting approximations for energy/quality tradeoffs in service-

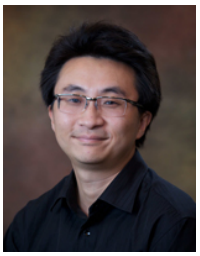
based applications, *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 156–157.
[35] Liu, Sicong and Lin, Yingyan and Zhou, Zimu and Nan, Kaiming and Liu, Hui and Du, Junzhao, On-demand deep model compression for mobile devices: a usage-driven model selection framework, *ACM MobiSys'18*, pp. 389–400.
[36] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, AutoCompress: an automatic DNN structured pruning framework for ultra-high compression rates, *AAAI'20*.
[37] X. Lu, P. Wang, D. Niyato, D. Kim, and Z. Han, Wireless networks with RF energy harvesting: a contemporary survey, *IEEE Communications Surveys & Tutorials*, Vol. 17, No. 2, 2014, pp. 757–789.
[38] J. Luo, J. Wu, and W. Lin, ThiNet: a filter level pruning method for deep neural network compression, *IEEE ICCV'17*, pp. 5058–5066.
[39] Neural Processor: en.wikichip.org/wiki/neural_processor
[40] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, and *et al.*, Human-level control through deep reinforcement learning, *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533.
[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, *ICML'16*, pp. 1928–1937.
[42] X. Jiang, H. Wang, Y. Chen, and *et al.*, MNN: a universal and efficient inference engine, *arXiv preprint*, 2020, arXiv:2002.12418.
[43] D. Ng, E. Lo, and R. Schober, Wireless information and power transfer: energy efficiency optimization in OFDMA systems, *IEEE Transactions on Wireless Communications*, Vol. 12, No. 12, 2013, pp. 6352–6370.
[44] NVIDIA-T4: nvidia.com/en-us/data-center/tesla-t4/
[45] Google Pixel 2 XL: en.wikipedia.org/wiki/Pixel_2
[46] A. Polino, R. Pascanu, and D. Alistarh, Model compression via distillation and quantization, *ICLR'18*.
[47] PyTorch Mobile: pytorch.org/mobile/
[48] PyTorch: pytorch.org/
[49] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, XNOR-Net: imagenet classification using binary convolutional neural networks, *ECCV'16*, pp. 525–542.
[50] S. Ren, K. He, R. Girshick, and J. Sun, Faster R-CNN: towards real-time object detection with region proposal networks, *NeurIPS'15*, pp. 91–99.
[51] A. Romero, N. Ballas, S. Kahou, A. Chassang, C. Gatta, and Y. Bengio, Fitnets: hints for thin deep nets, *ICLR'14*.
[52] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley, A survey of multi-objective sequential decision-making, *Journal of Artificial Intelligence Research*, Vol. 48, 2013, pp. 67–113.
[53] S. Shi, Q. Wang, P. Xu, and X. Chu, Benchmarking state-of-the-art deep learning software tools, *IEEE International Conference on Cloud Computing and Big Data*, 2016, pp.99-104.
[54] R. Srikant, The mathematics of Internet congestion control, *Springer Science & Business Media*, 2012.
[55] Y. Sun, X. Wang, and X. Tang, Deep convolutional network cascade for facial point detection, *IEEE CVPR'13*, pp. 3476–3483.
[56] Y. Tang, Deep learning using linear support vector machines, *arXiv preprint*, 2013, arXiv:1306.0239.
[57] Tensorflow Lite: tensorflow.org/lite
[58] Tencent, NCNN: github.com/Tencent/ncnn
[59] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, Learning structured sparsity in deep neural networks, *NeurIPS'16*, pp. 2074–2082.
[60] K. Winstein and H. Balakrishnan, TCP ex Machina: computer-generated congestion control, *ACM SIGCOMM'13*, pp. 123–134.
[61] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu and D. Yang, Experience-driven networking: a deep reinforcement learning based approach, *IEEE INFOCOM'18*, pp. 1871–1879.
[62] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, DeepWear: adaptive local offloading for on-wearable deep learning, *IEEE Transactions on Mobile Computing*, Vol. 19, No. 2, 2019, pp. 314–330.
[63] M. Yang, S. Wang, J. Bakita, T. Vu, F. Smith, J. Anderson, and J. Frahm, Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: addressing an industrial challenge, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 305–317.
[64] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, DeepIoT: compressing deep neural network structures for sensing systems with a compressor-critic framework, *ACM SenSys'17*, pp. 1–14.
[65] S. Zagoruyko and N. Komodakis, Paying more attention to attention: improving the performance of convolutional neural networks via attention transfer, *ICLR'17*.
[66] L. Zeng, E. Li, Z. Zhou, and X. Chen, Boomerang: on-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things, *IEEE Network*, Vol. 33, No. 5, 2019, pp. 96–103.
[67] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, A systematic DNN weight pruning framework using alternating direction method of multipliers, *ECCV'18*, pp. 184–199.
[68] Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, ECRT: an edge computing system for real-time image-based object tracking, *ACM SenSys'18*, pp. 394–395.
[69] H. Zhou, S. Bateni, and C. Liu, S^3DNN: supervised streaming and scheduling for GPU-accelerated real-time DNN workloads, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 190–201.
[70] Z. Zhu, K. Lin, and J. Zhou, Transfer learning in deep reinforcement learning: A survey, *arXiv preprint*, 2020, arXiv:2009.07888.

Zhiyuan Xu is currently pursuing the Ph.D. degree at the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, USA. He received the B.E. degree in School of Computer Science and Engineering from University of Electronic Science and Technology of China, Chengdu, China, in 2015. He was an exchange student in 2013 at Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. He was a visiting student in 2015 at Dalhousie University, Halifax, NS, Canada. His current research interests include deep reinforcement learning, communication networks, and edge computing.



Dejun Yang (M'13–SM'19) received the B.S. degree in computer science from Peking University, Beijing, China, in 2007, and the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, USA, in 2013. He is currently an Associate Professor of computer science with ColoradoSchool of Mines, Golden, CO, USA. His research interests include Internet of things, networking, and mobile sensing and computing, with a focus on the application of game theory, optimization, algorithm design, and machine learning to resource allocation, security, and privacy problems. He has received the IEEE Communications Society William R. Bennett Prize in 2019 (Best Paper Award for IEEE/ACM TON and IEEE TNSM in the previous three years), and the Best Paper Awards at the IEEE GLOBECOM (2015), the IEEE MASS (2011), and the IEEE ICC (2011 and 2012), as well as the Best Paper Award Runner-up at the IEEE ICNP (2010). He is the TPC Vice Chair of information systems for the IEEE INFOCOM 2020, a Student Travel Grant Co-Chair for INFOCOM 2018–2019, and was the Symposium Co-Chair for the International Conference on Computing, Networking and Communications (ICNC) 2016. He currently serves as an Associate Editor for the IEEE INTERNET OF THINGS JOURNAL.

Chengxiang Yin is currently pursuing the Ph.D. degree at the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, USA. He received his bachelor degree from School of Information and Electronics at Beijing Institute of Technology, Beijing, China, in 2016. His research interests include Machine Learning and Computer Vision.



Jian Tang (F19) is a Professor in the Department of Electrical Engineering and Computer Science at Syracuse University, an IEEE Fellow and an ACM Distinguished Member. He received his Ph.D degree in Computer Science from Arizona State University in 2006. His research interests lie in the areas of AI, IoT, Wireless Networking, Mobile Computing and Big Data Systems. Dr. Tang has published over 160 papers in premier journals and conferences. He received an NSF CAREER award in 2009. He also received several best paper awards, including the 2019 William R. Bennett Prize and the 2019 TCBD (Technical Committee on Big Data) Best Journal Paper Award from IEEE Communications Society (ComSoc), the 2016 Best Vehicular Electronics Paper Award from IEEE Vehicular Technology Society (VTS), and Best Paper Awards from the 2014 IEEE International Conference on Communications (ICC) and the 2015 IEEE Global Communications Conference (Globecom) respectively. He has served as an editor for several IEEE journals, including IEEE Transactions on Big Data, IEEE Transactions on Mobile Computing, etc. In addition, he served as a TPC co-chair for a few international conferences, including the IEEE/ACM IWQoS'2019, MobiQuitous'2018, IEEE iThings'2015. etc.; as the TPC vice chair for the INFOCOM'2019; and as an area TPC chair for INFOCOM 2017-2018. He is also an IEEE VTS Distinguished Lecturer, and the Chair of the Communications Switching and Routing (CSR) Technical Committee of IEEE ComSoc.

Yanzhi Wang is currently an assistant professor in the Department of Electrical and Computer Engineering at Northeastern University. He has received his Ph.D. Degree in Computer Engineering from University of Southern California (USC) in 2014, and his B.S. Degree with Distinction in Electronic Engineering from Tsinghua University in 2009. Dr. Wang's current research interests are the energy-efficient and high-performance implementations of deep learning and artificial intelligence systems. Besides, he works on the application of deep learning and machine intelligence in various mobile and IoT systems, medical systems, and UAVs, as well as the integration of security protection in deep learning systems. His works have been published in top venues in conferences and journals (e.g. ASPLOS, MICRO, HPCA, ISSCC, AAAI, ICML, ICLR, ECCV, ACM MM, CCS, VLDB, FPGA, DAC, ICCAD, DATE, LCTES, INFOCOM, ICDCS, Nature SP, etc.), and have been cited for around 4,000 times according to Google Scholar. He has received four Best Paper Awards, has another seven Best Paper Nominations and two Popular Papers in IEEE TCAD. His group is sponsored by the NSF, DARPA, IARPA, AFRL/AFOSR, and industry sources.

Guoliang Xue (F11) is a Professor of Computer Science and Engineering at Arizona State University. He received the Ph.D degree in Computer Science from the University of Minnesota in 1991. His research interests span the areas of QoS provisioning, machine learning, wireless networking, network security and privacy, crowdsourcing and network economics, Internet of Things, smart city and smart grids. He has published over 300 papers in these areas, many of which in top conferences such as INFOCOM, MOBICOM, NDSS and top journals such as IEEE/ACM ToN, IEEE JSAC, IEEE TDSC, and IEEE TMC. He has received the IEEE Communications Society William R. Bennett Prize in 2019 (best paper award for IEEE/ACM TON and IEEE TNSM in the previous three years). He was a keynote speaker at IEEE LCN'2011 and ICNC'2014. He was a TPC Co-Chair of IEEE INFOCOM'2010 and a General Co-Chair of IEEE CNS'2014. He has served on the TPC of many conferences, including ACM CCS, ACM MOBIHOC, IEEE ICNP, and IEEE INFOCOM. He served on the editorial board of IEEE/ACM Transactions on Networking and the Area Editor of IEEE Transactions on Wireless Communications, overseeing 13 editors in the Wireless Networking area. He is an IEEE Fellow, and the Steering Committee Chair of IEEE INFOCOM.