DNNOff: Offloading DNN-Based Intelligent IoT Applications in Mobile Edge Computing

Xing Chen, *Member, IEEE*, Ming Li, Hao Zhong, *Member, IEEE*, Yun Ma, *Member, IEEE*, and Ching-Hsien Hsu, *Senior Member, IEEE*

***Abstract*—A deep neural network (DNN) has become increasingly popular in industrial Internet of Things scenarios. Due to high demands on computational capability, it is hard for DNN-based applications to directly run on intelligent end devices with limited resources. Computation offloading technology offers a feasible solution by offloading some computation-intensive tasks to the cloud or edges. Supporting such capability is not easy due to two aspects: *Adaptability:* offloading should dynamically occur among computation nodes. *Effectiveness:* it needs to be determined which parts are worth offloading. This article proposes a novel approach, called DNNOff. For a given DNN-based application, DNNOff first rewrites the source code to implement a special program structure supporting on-demand offloading and, at runtime, automatically determines the offloading scheme. We evaluated DNNOff on a real-world intelligent application, with three DNN models. Our results show that, compared with other approaches, DNNOff saves response time by 12.4–66.6% on average.**

***Index Terms*—Computation offloading, deep neural networks (DNNs), intelligent Internet of Things (IoT) application, mobile edge computing (MEC), software adaption.**

I. Introduction

RECENT years have witnessed the remarkable improvements of a deep neural network (DNN) . As the core machine learning technique [1], the DNN has been applied in industrial Internet of things (IoT) scenarios such as computer vision [2] and self-driving cars [3]. Meanwhile, more and more trained deep learning models have been deployed on intelligent end devices, such as wearable devices [4], vehicles [5], and unmanned aerial vehicles [6]. In this article, we call such trained models as DNN-based intelligent IoT applications.

Due to limited resources about computation and storage, complex DNN-based applications cannot be directly run on intelligent end devices. One feasible solution is to offload all or part of computational tasks to the cloud with sufficient resources [7], [8]. More specifically, DNNs are divided by the granularity of neural network layers [9]. Thus, some computation-intensive neural network layers can be offloaded to the cloud for execution, while other simpler neural network layers are processed locally.

However, the network communication between end devices and the cloud is likely to cause significant execution delay, and it seriously affects the user experience. To address this delay problem, mobile edge computing (MEC) has been introduced [10]. The mobile edges provide computing capabilities in close proximity to end devices and enable the execution of highly demanding applications in end devices while offering significantly lower latencies. Although MEC provides new opportunities to offload DNN-based applications among end devices, the cloud, and nearby edges, the prior approaches do not consider how to offload them in the new environment. On the one hand, as the environment is constantly changing, the offloading scheme of the DNN model shall be flexible for the need of adaptation. On the other hand, an offloading scheme shall make tradeoffs between the reduced execution time and the network delay, when it determines which layers will be offloaded and where to offload them, based on the changes of environment.

To fully release the potential of offloading, an offloading mechanism shall support on-demand changes for DNN-based applications and shall enable the execution of some parts of the DNN model on different computing nodes (including end devices, cloud, and edge servers). Afterward, there needs to be an efficient estimation model, which can determine which of its layers shall be offloaded. In summary, our main research questions are: 1) How to design a mechanism to support the automatic offloading of DNN-based applications in the MEC environment? 2) How to build an estimation model to determine the optimal offloading schemes? After the above questions are carefully handled, the problem of offloading can be reduced to a traditional optimization problem [11].

Q1 Manuscript received February 22, 2021; revised March 17, 2021 and
Q2 March 29, 2021; accepted April 18, 2021. This work was supported in part by the National Natural Science Foundation of China under Grant 62072108 and in part by the Natural Science Foundation of Fujian
Q3 Province for Distinguished Young Scholars under Grant 2020J06014. Paper no. TII-21-0800. *(Corresponding author: Hao Zhong.)*

Xing Chen and Ming Li are with the College of Mathematics and Computer Science and the Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou 350118, China (e-mail: chenxing@fzu.edu.cn; N190327047@fzu.edu.cn).

Hao Zhong is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: zhonghao@sjtu.edu.cn).

Yun Ma is with the Institute for Artificial Intelligence, Peking University, Beijing 100871, China, and also with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: mayun@pku.edu.cn).

Ching-Hsien Hsu is with the Department of Computer Science and Information Engineering, Asia University, Taichung 41354, Taiwan, and also with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi 621301, Taiwan (e-mail: robertchh@gmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TII.2021.3075464.

Digital Object Identifier 10.1109/TII.2021.3075464

To address the aforementioned questions, we present a novel approach called DNNOff, which supports offloading DNN-based applications in the MEC environment. This article makes the following major contributions:

1) an offloading mechanism that enables DNN-based applications to be offloaded automatically and dynamically in the MEC environment. To achieve this, DNNOff translates a DNN-based application to a target program that is easier to offload;
2) an effective model to predict the latency of offloading schemes. DNNOff first extracts the structure and parameters of the DNN model and then uses a random forest regression model to predict the execution cost of each layer. Based on the prediction model, DNNOff can determine which parts shall be moved to MEC servers; and
3) an evaluation on a real-world DNN-based application with AlexNet, VGG, and ResNet models. Our results show that DNNOff reduces the response time by 12.4–66.6% for complex DNN-based applications.

The rest of this article is organized as follows. Section II reviews the related work. Section III presents our approach, and Section IV evaluates it on a real-world application. Section V discusses some issues about applicability. Section VI introduces industrial applications. Finally, Section VII concludes this article.

II. Related Work

Mobile devices are generally limited to storage space, battery life, and computing power [12]. To improve the performance of mobile applications, computation offloading has become the most widely used technology. MCC improves the performance of applications by sending computing-intensive components from end devices to the cloud . These applications are partitioned at different granularities, such as method, thread, and class. For example, MAUI [13] supports offloading at the granularity of methods. It allows annotating which parts of a program can be offloaded to the cloud and makes offloading decisions at runtime. CloneCloud [14] is a thread-based computation offloading framework, and it modifies virtual machines to support seamless offloading of threads to the cloud. DPartner [15] can offload classes, and it uses a proxy mechanism to access class instances. Furthermore, it calculates the coupling of classes and divides them into two sets. The two sets are deployed on the end device and the cloud server, respectively. However, MCC has inherent limitations, namely, long latency between end devices and remote clouds. Hence, MEC has been proposed, in which the service of cloud is increasingly moving toward nearby edges [16]. AndroidOff [17] supports mobile applications with the offloading capability at the granularity of objects for MEC. It provides the mechanism to offload an object-oriented application and determine which parts shall be offloaded. However, the proposed works above cannot apply to DNN-based applications.

Computation offloading for DNN-based applications is further advanced in recent years. Neurosurgeon [9] showed that large amounts of data produced by DNN models should be uploaded to the cloud via wireless network, leading to high latency

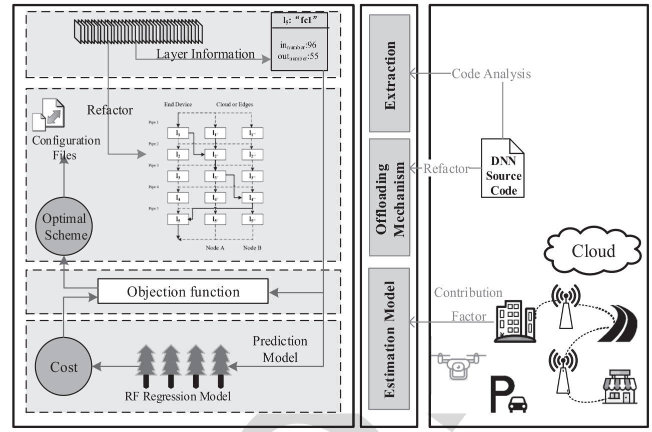

Fig. 1. Overview of DNNOff.

and energy consumption. For the sake of better performance and energy efficiency of modern DNN-based applications, Neurosurgeon designed a light weight scheduler to partition DNN-based applications automatically between end devices and the cloud at the granularity of neural network layers. Edgent [18] is a framework that automatically and intelligently selects the best partition point of a DNN model to satisfy the requirement on the execution latency. Compared with Neurosurgeon, Edgent can offload computation-intensive DNN layers to the remote server at a low transmission overhead, namely, nearest computation node. Liu *et al.* [19] proposed an image recognition framework based on the DNN in the MEC environment and realized the food image recognition system by employing an edge-computing-based service infrastructure. It allows the system to overcome some inherent limitations of the traditional MCC paradigm, such as high latency and energy consumption. Zhou *et al.* [20] proposed a robust mobile crowd sensing framework in the MEC environment. It can reduce the service delay with edge-computing-based local processing. The above approaches assume that end devices use a single remote server for computation offloading and cannot make efficient use of dispersed and changing computing resources in the MEC environment.

III. Approach

Fig. 1 presents the overview of DNNOff. For the nodes, we use rectangles to denote its components and circles to denote its internal data. For the edges, red ones denote data flows, and blue ones denote requests. DNNOff has three main components, namely, extraction, offloading mechanism, and estimation model. First, the extraction component extracts the structure and parameters of a DNN model (see Section III-A). Second, the offloading mechanism translates a DNN model to a target program that enables offloading (see Section III-B) and deploys it on end devices and remote servers where offloading may occur. Finally, the estimation model component deployed on the end device synthesizes an optimized offloading scheme to execute different parts of the target program on proper locations, based on the DNN network structure information and the surrounding MEC environment (see Section III-C). Moreover, the estimation model will update the offloading decision when the surrounding

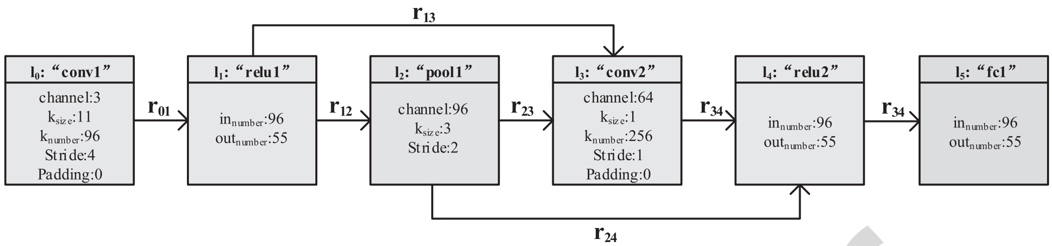

Fig. 2. Example of the DNN model.

MEC environment changes. In Fig. 1, a DNN-based application and its MEC environment are presented on the right.

A. Extracting Structure for the DNN Model

Fig. 2 shows an example of the DNN model. A DNN model consists of layers. In Fig. 2, layers are represented as squares in different colors. In particular, the yellow one represents a convolution layer, which translates an image to a feature map with learned filters. The blue one represents an activation layer, which is a nonlinear function. The function accepts a feature map and generates an output with the same dimension. The purple one represents a pooling layer, which can be defined as a general pooling, an average pooling, or a max pooling. The green one represents a fully connected layer, which calculates the weighted sum of the inputs by learned weights. The top of square is the name of layer, such as "conv1" and "relu1," and the bottom of square is the parameters of layer. For example, "channel:3" denotes that the corresponding value of the parameter "channel" is "3." The black arrow represents the data flow. DNNOff first extracts the structure of a DNN model, and the structure includes the parameters of each layer and the data flow between layers. Its definitions are as follows.

Definition 1 (DNN model structure): A DNN model structure is a directed graph $G_D = (L, R)$ representing data transmissions between layers of a DNN D, where $L = \{l_1, l_2, \ldots, l_n\}$ is the set of layers of D and R is the set of data flow edges. Each edge $r_{ij} \in R$ represents a data flow from l_i to l_j.

Definition 2 (DNN layer information): A layer consists of type and parameters as $l_i =< type, feature >$, where $type$ is the type of the layer and $feature$ denotes the set of features of the layer.

In general, the DNN-based application stores its trained model in the configuration file, such as prototxt of Caffe2.[1] Our approach takes this file as the input and gets the DNN model graph $G_D = (L, R)$ through code analysis.

B. Offloading Mechanism for the DNN Model

First, we translate an original application to a target program, and the translated target program follows the pipe-and-filter style. In this style, DNN layers are modeled as filters that receive and send data, and data flows between two layers are modeled as pipes that transmit the intermediate results. Second, we propose a "Pipe" engine to determine which neural network layer shall be offloaded.

[1][Online]. Available: https://caffe2.ai/

```
# Source Structure of DNN program        # Target Structure of DNN program

DP(InitData):                            DP(InitData, InitL, EndL) :
    l0 = InitData                            CurrentL = InitL
    l1 = l1.type(l0, l1.feature)             l_InitL-1 = InitData
    l2 = l2.type(l1, l2.feature)             CurrentL = Pipe(1)
    ......                                   ......
    li = li.type(li-1, li.feature)           CurrentL = Pipe(i) ------------ Pipe_i
    ......                                   if CurrentL == i:
    ln = ln.type(ln-1, ln.feature)             li = li.type(li-1, li.feature)
                                               CurrentL++                      l_i
    return n.blobs[ln].data[0]               if CurrentL-1 == EndL:
                                               Output = n.blobs[l_EndL].data[0]
                                               return Output
                                             ......
```

Fig. 3. Translation of a DNN program.

1) Target Program: We abstract a DNN program using the pipe-and-filter architecture style, based on which we propose a design pattern to support adaptive offloading in MEC.

A DNN program is essentially a data flow software architecture [21]. Each layer can be regarded as a filter, and the data transmission between layers can be regarded as a pipe. In a typical DNN program, each filter performs the calculation of a layer, whereas the pipe uses the result of the preceding layer as the input data of the succeeding layer.

In order to support adaptive offloading of DNN applications, the pipe should decide whether to transfer the data to the local filter or to the remote filter for the successive computation tasks. The filter should decide whether to perform the current computation task (calculation of the current layer) or directly return the results.

The left-hand side of Fig. 3 shows the source program of a DNN-based application. It starts from the first layer and receives the initial data (i.e., an image). The result of each layer, namely the intermediate result, is hidden, and the output of the last layer is the return value. The statement, $l_i = l_i.type(l_{i-1}, l_i.feature)$, indicates that the l_i layer takes the result of l_{i-1} as its input. The right-hand side of Fig. 3 presents the translated target program. It uses "Pipe" functions to connect each layer, such that the DNN model can be offloaded at the granularity of layers.

2) Code Translation: Our translation has three steps.

Step 1 (Adding the parameters such as "InitL" and "EndL" after "InitData"): For a given program, DNNOff automatically adds "InitL" and "EndL" into the list of parameters. The two parameters represent the labels of the initial and the last layers, respectively. In addition, DNNOff adds "$CurrentL$," which denotes the label of the layer to be executed. Meanwhile, "InitData" is assigned to the result of $l_{\mathrm{InitL}-1}$, and used as an input to l_{InitL}. Here, when the "DP" program runs, the layers between l_{InitL} and l_{EndL} shall be executed.

Algorithm 1: Pipe.

Input: m—the label of the "Pipe" function
Output: $CurrentL$—the label of the layer to be executed
Declare:
$config$—the offloading scheme that records execution locations of each layer;
$EndL$—the label of the last layer that is executed at Local;
l_i—the result of the ith layer

1: **if** $m < CurrentL$ **then**
2:   **return** $CurrentL$
3: **end if**
4: **if** $m == CurrentL$ and $config[m] == Local$ **then**
5:   **return** $CurrentL$
6: **end if**
7: **if** $m == CurrentL$ and $config[m]$!= $Local$ **then**
8:   $k \leftarrow calculate\ the\ label\ of\ the\ next\ layer\ that$
9:       $shall\ be\ executed\ at\ Local$
10:   **if** $k == Null$ **then**
11:     $k \leftarrow EndL + 1$
12:   **end if**
13:   $l_{k-1} \leftarrow remote(l_{CurrentL-1},\ CurrentL,\ k-1)$
14:   $CurrentL \leftarrow k$
15:   **return** $CurrentL$
16: **end if**

Step 2 (Adding a "Pipe(i)" function before each layer l_i): This function determines whether the layer l_i shall be offloaded (see Section III-B3 for details).

Step 3 (Adding two if statements to check each layer l_i): The first statement is "$ifCurrentL == i$," where l_i represents the ith layer. It checks whether the layer l_i is to be executed currently. The second statement is "$ifCurrentL - 1 == EndL$." If the layer l_{EndL} has been executed, its result shall be returned, and the layers after l_{EndL} are skipped.

3) Computation Offloading at Runtime: At runtime, the "Pipe" functions connect each layer that can be executed locally or remotely, according to the offloading scheme. Algorithm 1 shows how the "Pipe" function works. Pipe(m) denotes the pipe between the layers l_{m-1} and l_m, $config$ is the offloading scheme that records execution locations of each layer, and $CurrentL$ denotes the label of the layer to be executed.

When $m < CurrentL$, it indicates that the layer l_m has been executed and does not need to be repeated (lines 1–3). Therefore, the layer l_m will be skipped.

When $m == CurrentL$ and $config[m] == Local$ (Local is a keyword, representing the local node), it means that the layer l_m is to be locally executed (lines 4–6). Therefore, l_m is executed and the value of $CurrentL$ is added by 1.

When $m == CurrentL$ and $config[m]$!= $Local$, it means that the layer l_m is to be remotely executed (lines 7–15). Then, we calculate the label of the next layer that shall be executed at local (line 8), and if k does not exist, we assign "$EndL$+1" to it (lines 9–11). Finally, we run the program $DP(l_{CurrentL1}, CurrentL, (k-1)$ on the remote node according to $config[m]$ and assign its result to l_{k-1} (line 12).

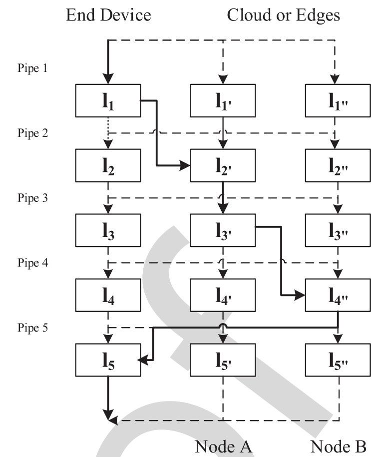

Fig. 4. Proposed design pattern of DNN programs.

Fig. 4 shows the example of adaptive offloading of the five-layer DNN, which is executed on three computation nodes. Layers l_1 and l_5 are to be executed on end device, layers l_2 and l_3 are to be executed on Node A, and layer l_4 is to be executed on Node B. First, the DNN program $DP(InputData, 1, 5)$ runs on end device, while $CurrentL$ is 1 and $EndL$ is 5, and l_0 is set to $InputData$; $Pipe(1)$ and l_1 are executed, as $config[1]$ is end device; $Pipe(2)$ is executed and the remote service is invoked, as $config[2]$ is Node A. Second, the DNN program $DP(l_i, 2, 4)$ runs on the Node A; $Pipe(1)$ and l_1 are skipped, as $CurrentL$ is 2; $Pipe(2)$, l_2, $Pipe(3)$, and l_3 are executed in sequence, as $config[2]$ and $config[3]$ are both Node A; $Pipe(3)$ is executed and the remote service is invoked, as $config[4]$ is Node B. Third, the DNN program $DP(l_3, 4, 4)$ runs on the Node B; $Pipe(1)$, l_1, $Pipe(2)$, l_2, $Pipe(3)$, and l_3 are all skipped, as $CurrentL$ is 4; $Pipe(4)$ and l_4 are executed as $config[4]$ is Node B; then, $CurrentL$ is 5 and thus return the calculation result to the DNN program on end device. Finally, l_5 is executed on end device and the output is produced.

C. Estimation Model for the Offloading Scheme

1) Predicting Cost With Random Forest Regression: The execution time of each layer is an essential factor in the estimation model. If the layer l_i is executed on the node n_k, we define its execution cost as follows.

Definition 3 (Execution cost): $Cost_{n_k}^{l_i} =< time, datasize >$: $time$ denotes the execution time from setting input data to generating output data, which depends on the performance of the execution node, while $datasize$ denotes the amount of data transmission, which is a fixed value obtained by the extraction component.

With the number of layers and the diversity of computing nodes, it is difficult to get execution time of each layer on each computing node. Thus, we used the random forest regression to build prediction models for different layer types and computing nodes, which is to predict $Cost_{n_k}^{l_i}.time$. The RF regression model is proposed by Brieman [22] and is proved to carry out the nonlinear relation between the variables. It is a nonlinear model-building tool, which is widely used in classification [23] and prediction [24].

TABLE I
FACTORS THAT CAN INFLUENCE THE OFFLOADING DECISION

Symbol	Description
L	the set of layers in DNN model, $L = \{l_1, l_2, \cdots, l_n\}$
N	the set of compute nodes including ED, NE and RC, $n_k \in N$
P^{l_i}	the set of parent nodes of layer l_i
DEP	the set of offloading schemes, $DEP = \{dep(l_1), dep(l_2), \cdots, dep(l_n)\}$
$v_{n_i n_j}$	the data transmission rate between n_i and n_j
$rtt_{n_i n_j}$	the round-trip time between n_i and n_j
t_i	the moment after the execution of layer l_i
$T_d(l_k, l_m)$	the data transmission time between layer l_k and layer l_m
$T_e(l_i)$	the execution time of layer l_i
$T_{response}$	the response time of DNN-based application

Definition of the prediction model:

$$Y = \text{predict}(X) \tag{1}$$

$$\begin{aligned} X_{\text{conv}} &= (channel, k_{\text{size}}, k_{\text{number}}, stride, padding) \\ X_{\text{pooling}} &= (channel, k_{\text{size}}, stride) \\ X_{\text{relu}} &= (in_{\text{number}}, out_{\text{number}}) \\ X_{fc} &= (in_{\text{number}}, out_{\text{number}}). \end{aligned} \tag{2}$$

We use the dataset of history data to train the prediction model, which is collected from DNN applications, including Alexnet [25], VGG16, VGG19 [26], ResNet-50, and ResNet-152 [27]. The RF regression prediction model is represented as Equation (1). The input(X) depends on the type of layers as Equation (2), and the layer types include convolution layer, pooling layer, activation layer, and fully connected layer.

2) Contributory Factor: In this subsection, we introduce a context model that describes the environment (e.g., computation nodes) and the factors that affect the offloading decision.

The context architecture consists of an end device (ED), several nearby edges (NE), and a remote cloud (RC). We use a graph to present this network $G_C = (N, E)$, where N denotes a set of compute nodes, including end device and remote servers, and E represents a set of communication links among nodes $n_i \in N$. Each edge$(n_i, n_j) \in E$ is associated with a data transmission rate $v_{n_i n_j}$ and a round-trip time $rtt_{n_i n_j}$ between n_i and n_j. A typical offloading scenario is as follows: The data are generated on the end device (the only n_{ED}), and the layers can be offloaded to nearby edges (some nodes of n_{NE}) or the remote cloud (n_{RC}).

Table I shows our factors for estimating an offloading scheme. Among them, $n_k \in N, v_{n_i n_j}$ and $rtt_{n_i n_j}$ are defined before. We next introduce $DEP = (dep(l_1), dep(l_2), \ldots, dep(l_n))$, where DEP denotes the offloading scheme. Each $l_i \in L$ is executed on a computation node $dep(l_i) \in N$. Let $T_e(l_i)$ denote the execution time of l_i and let $T_d(l_k, l_m)$ denote the data transmission time between layer l_k and layer l_m. The response time of application can be represented by T_{response}, which is equal to the moment after the execution of the last layer (t_n). In addition, an objective function is constructed to calculate T_{response} and estimate the offloading scheme.

3) Objective Function: Our objective function makes predictions, based on contributory factors. In particular, based on the factors in Section III-C2, we construct the objective function as shown in Equation (3). Here, we consider that a scheme is optimal, if its objective value is the smallest. As Table I shows that t_i is the moment after the execution of layer l_i, the total response time is obtained when the last layer l_n is executed

Algorithm 2: Calculation of Response Time.

Input: P^{l_i}—the set of parent nodes of the layer l_i
Output: t_n—the response time of an offloading scheme
Declare:
l_i—the ith layer
t_i—the moment after the execution of the layer l_i;
$t_{\max}$—the maximum sum of the time at the moment after the execution of each parent layer with the addition of the
8: transmission time between two layers;
9: $T_d(l_k, l_m)$—the data transmission time between the layer
l_k and the layer l_m;
$T_e(l_i)$—the execution time of the layer l_i

```
function currentTime P^{l_i}, l_i
    for each p_j^{l_i} ∈ P^{l_i} do
    if t_{p_j^{l_i}} not calculated then
        t_{p_j^{l_i}} ← currentTime(P^{p_j^{l_i}}, p_j^{l_i})
    end if
    t_max ← max{t_max, t_{p_j^{l_i}} + T_d(p_j^{l_i}, l_i)}
  end for
  t_i ← t_max + T_e(l_i)
  return t_i
end Function
t_0 ← 0
t_n ← currentTime(P^{l_n}, l_n)
return t_n
```

$$T_{\text{response}} = t_n \tag{3}$$

$$t_i = \max\left\{t_{p_j^{l_i}} + T_d(p_j^{l_i}, l_i)\right\} + T_e(l_i), \forall p_j^{l_i} \in P^{l_i} \tag{4}$$

$$T_e(l_i) = Cost^{l_i}_{dep(l_i)}.time \tag{5}$$

$$T_d(p_j^{l_i}, l_i) = \frac{Cost^{p_j^i}.datasize}{v_{dep(p_j^i)dep(l_i)}} + rtt_{dep(p_j^i)dep(l_i)}. \tag{6}$$

The description of Equation (4) is expounded as follows: First, the moment before the execution of current layer l_i is calculated as the moment after the execution of previous layer ($t_{p_j^{l_i}}$) with the addition of the transmission time between two layers ($T_d(p_j^{l_i}, l_i)$). Second, according to the characteristic of the DNN, the current layer can only be executed when all branches from previous layers have already been executed. Hence, t_i includes the execution time of layer l_i, and the maximum sum of the time at the moment after the execution of each parent layer with the addition of the transmission time between two layers. Among them, the execution time of layer l_i is represented as Equation (5) ($Cost^{l_i}_{dep(l_i)}.time$ is mentioned in Definition 3) and the transmission time with previous layer is represented as Equation (6).

TABLE II
DEVICE CONTEXTS IN DIFFERENT LOCATIONS

	Community	Traffic Road	Parking Lot	Store	Cloud
E1	–	RTT = 30 ms V = 1 Mb/s	–	RTT = 30 ms V = 1 Mb/s	RTT = 50 ms V = 800 Kb/s
E2	RTT = 30 ms V = 1 Mb/s	RTT = 60 ms V = 700 Kb/s	–	–	RTT = 80 ms V = 500 Kb/s
Cloud	RTT = 150 ms V = 200 Kb/s	RTT = 150 ms V = 200 Kb/s	RTT = 150 ms V = 200 Kb/s	RTT = 150 ms V = 200 kb/s	–

As a result, given an offloading scheme, the calculation of response time is shown in Algorithm 2. According to line 11 of Algorithm 2, we first initialize the value of t_0. Then, we use the "currentTime" function to calculate t_n recursively according to line 12. The calculation principle of the "currentTime" function corresponds to Equation (4).

IV. EVALUATION

We implemented DNNOff and conducted evaluations to explore the following research questions.

- (RQ1) To what degree does DNNOff improve performance of DNN-based applications (see Section IV-A)?
- (RQ2) How does DNNOff perform in cost prediction of each neural network layer (see Section IV-B)?
- (RQ3) How much extra overhead does DNNOff introduce (see Section IV-C)?

For RQ1, our results show that DNNOff saved 12.4–66.6% response time compared with other approaches. For RQ2, DNNOff achieved high accuracy for predicting execution time in different layer types and computing nodes. For RQ3, the overhead of our offloading mechanism is acceptable.

A. RQ1 Improvement Over the State of the Art

1) Experimental Settings:

a) Network environment: The network context consists of four computation nodes: one end devices and three remote servers. We simulate four locations, which are named community, traffic road, parking lot, and store. Table II lists the connections among our computation nodes. The column and the row of a cell denote the round-trip time and the data transmission rate between computation nodes. We utilize the network simulation tool Dummynet[2] to control the available bandwidth. A smaller rtt and a higher v denotes a better signal strength.

b) Devices: We take three desktop computers to emulate the Elastic Compute Service (ECS) and edge servers $E1$ and $E2$. The ECS is equipped with a 3.6-GHz 16-core CPU and 16-GB RAM, server $E1$ is equipped with a 2.5-GHz eight-core CPU and 8-GB RAM, and server $E2$ is equipped with a 3.0-GHz eight-core CPU and 8-GB RAM. We further use a smartphone to act as the end device, and the end device is equipped with a 2.2-GHz CPU and 4-GB RAM.

c) Application: We use a real-world DNN-based image recognition application in the evaluation. It is written in Python and powered by the Caffe2 deep learning framework.

We mainly concern with three models, which are the core of the DNN-based application, including AlexNet, VGG16, and ResNet-50. The most complex model is ResNet-50, while AlexNet is the simplest one. The inference latency and recognition accuracy are increasing as the model is more complex.

d) Compared approaches: In our evaluation, we compared DNNOff with four other approaches.

The original application is executed on end device, without any offloading.

Neurosurgeon [9] selected the best DNN partition point and sent the remaining DNN layers from end device to the cloud.

Edgent [18] is similar to Neurosurgeon [9], but offloads computation-intensive DNN layers to the remote server at a low transmission overhead, namely, nearest computation node.

For *the ideal plan*, it has to get execution time of each layer on each computing node and choose the fastest one after executing all the schemes in reality. The ideal plan is infeasible in practice since it needs to get the execution time at different levels in advance and try all the possibilities. We introduce the ideal plan to illustrate how close DNNOff is to the ideal one.

e) Measurement: To show the effectiveness of DNNOff, we define the following metrics.

1) *Total response time:* We use the total response time as the metric for performance. To make a fair comparison, we pick ten different images from the video in each location and calculate their averages for comparison. Here, the start time is recorded when the image is input, and the end time is recorded when the recognition result is output. It includes local inference, data transmission, and remote inference. The less response time indicates better results.
2) *Local inference:* This is the time about inference process on the end device. The inference on remote servers is usually more efficient than end device.
3) *Remote inference:* This is the time about inference process on the remote servers.
4) *Data transmission:* This is the time to transmit the feature vectors result by the partitioned layers of DNN model, and it is often slow under poor network connection.

2) Results: The total response time consists of the inference time and the transmission time. Fig. 5 shows the time of compared approaches in the four locations. For each approach, the blue bar denotes the local inference time, the orange one denotes the data transmission time, while the gray one denotes the remote inference time.

Compared with the original application, DNNOff reduces the total response time by 30.4–66.6%. The result also shows that the more complex the model, the better the optimization of DNNOff. In general, the optimization of community is better than that of store, because community is closer to the better performing edge

[2][Online]. Available: http://info.iet.unipi.it/luigi/dummynet/

Fig. 5. Process of a DNN-based image recognition application. (a) Image recognition with the AlexNet model. (b) Image recognition with the VGG16 model. (c) Image recognition with the ResNet-50 model.

server, which can significantly reduce the reference time. In the traffic road, the ResNet-50 is optimized to 66.6% with DNNOff, since the data transfer volume between the layers in ResNet-50 is small and the location is connected to all remote servers, so that the offloading can alleviate the bottleneck of local inference time and, meanwhile, guarantee a lower data transmission time. It should be noted that the parking lot is only connected to the cloud server, so the performance improvement is not as obvious as that in other locations, but it can still reduce the time by 30.4–47.1%. Hence, DNNOff is still effective even if there are no edge servers.

Compared with Neurosurgeon, DNNOff reduces the total response time by 26.5–53.2%. The results show that DNNOff significantly outperforms Neurosurgeon in the traffic road with the VGG model. Because traffic road has the best network connection with remote servers, which provide more choices to DNNOff for offloading . While in the parking lot, DNNOff can keep the same performance as Neurosurgeon . Due to the poor network connection, multiple partitions will increase the

TABLE III
SAMPLE ITEMS

Sample No	channel	k_{size}	k_{number}	stride	padding	time(ms)
1	3	11	96	4	0	144
2	96	5	256	1	2	183
3	256	3	384	1	1	200
4	384	3	384	1	1	301

data transmission time instead. In this case, DNNOff makes the same offloading scheme as Neurosurgeon does.

Compared with Edgent, DNNOff reduces the total response time by 12.4–39.3%. Although Edgent considers the use of nearest computation node, DNNOff can cut the DNN at multiple points and execute different parts over the end device, edges, and the cloud.

Compared with the ideal plan, DNNOff can achieve comparable performance in different cases, and the performance gap between them is about 5%.

In summary, DNNOff saved 12.4–66.6% of the total response time compared with other approaches. Meanwhile, the results show that DNNOff achieves optimal/near-optimal performance of offloading.

B. RQ2 Accuracy for Cost Prediction of DNN Layers

1) Experimental Settings:

a) Model training: We use the dataset of history data to train the random forest regression prediction model, which is collected from DNN-based applications running on different computing nodes. In total, we collected the layer information about convolution layers, pooling layers, activation layers, and fully connected layers of 425 items, 320 items, 582 items, and 96 items, respectively. Table III shows some convolution layer items, which is collected on the end device, as an example. Column "Channel" lists the number of channels of convolution kernel. "k_{size}" and "k_{number}" list the size and the number of their filters, respectively. Columns "Stride" and "Padding" list the stride and the padding with which the filters are being applied. The inputs (X) include the channel, k_{size}, k_{number}, stride, and padding. The output (time) is denoted as the predicted value of layer latency. Based on the dataset, we randomly split the data items into two categories: 70% for training the prediction model and 30% for testing the quality of our model.

b) Measurement: We regard root-mean-square error (RMSE) and R-squared (R^2) as the evaluation measures of the prediction model

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{t=1}^{N}(\text{observed}_t - \text{predicted}_t)^2} \tag{7}$$

$$R^2 = 1 - \frac{\sum(\text{observed}_t - \text{predicted}_t)^2}{\sum(\text{observed}_t - \text{mean}_t)^2}. \tag{8}$$

RMSE is the sample standard deviation of the differences between predicted and observed values. R^2 is commonly used to evaluate the quality of regression models. They are calculated according to Equations (7) and (8).

2) Results:

Table IV shows the accuracy of the random forest regression prediction model. It illustrates the RMSE and

TABLE IV
RMSE AND R-SQUARED OF THE PREDICTION MODEL ON THE TEST SET

	RMSE		R-Squared	
Convolution Layer	Device	0.289 ms	Device	0.91
	Edge1	0.155 ms	Edge1	0.93
	Edge2	0.131 ms	Edge2	0.93
	Cloud	0.098 ms	Cloud	0.94
Pooling Layer	Device	1.210 ms	Device	0.78
	Edge1	0.845 ms	Edge1	0.82
	Edge2	0.799 ms	Edge2	0.83
	Cloud	0.524 ms	Cloud	0.83
Activation Layer	Device	0.058 ms	Device	0.69
	Edge1	0.029 ms	Edge1	0.70
	Edge2	0.022 ms	Edge2	0.72
	Cloud	0.012 ms	Cloud	0.75
Fully-connected Layer	Device	4.951 ms	Device	0.57
	Edge1	2.098 ms	Edge1	0.62
	Edge2	1.589 ms	Edge2	0.66
	Cloud	1.248 ms	Cloud	0.66

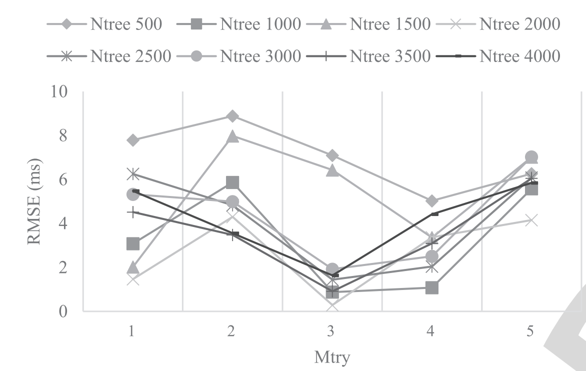

Fig. 6. Optimization of random forest parameters using RMSE.

R^2 results for predicting in different layer types and computation nodes. For RMSE, the smaller RMSE indicates the better model's fitting degree [28], [29]. For R^2, an acceptable value of R^2 is greater than 0.5 [30], and the closer to 1, the better the model is. Table IV shows that the RMSE of the model is small and R^2 is greater than 0.5, illustrating that the prediction model is acceptable. And the high accuracy of prediction model lays the foundation for scheme estimation.

In addition, there are two parameters in random forest: $Ntree$, the number of regression trees grown based on a bootstrap sample of the collected layers, and $Mtry$, the number of different predictors tested at each node. The two parameters ($Ntree$ and $Mtry$) are optimized based on the RMSE of calibration. Take the training of convolution layers on the end device as an example. $Ntree$ values from 500 to 4000 with intervals of length 50 were tested, and $Mtry$ was tested from 1 to 5. The results of random forest parameters ($Ntree$ and $Mtry$) are shown in Fig. 6, which clearly indicates that random forest parameters affect the error of prediction. The optimization was done using the calibration dataset ($n = 297$) and RMSE. The result $Ntree = 2000$ and $Mtry = 3$ yielded the lowest RMSE (0.289 ms). In this case, we chose $Ntree = 2000$ and $Mtry = 3$ as the best parameters.

C. RQ3 Extra Overhead

1) Experimental Settings:

a) Setting: We use a simple AlexNet [25] application with 24 layers, which is a state-of-the-art DNN for image

TABLE V
FIVE OFFLOADING SCHEMES FOR ALEXNET

Layers	1-8	9-15	16-24
Scheme 1	Device	Cloud	
Scheme 2	Device	Edge 1	
Scheme 3	Device	Edge 2	
Scheme 4	Device	Edge 1	Cloud
Scheme 5	Device	Edge 2	Cloud

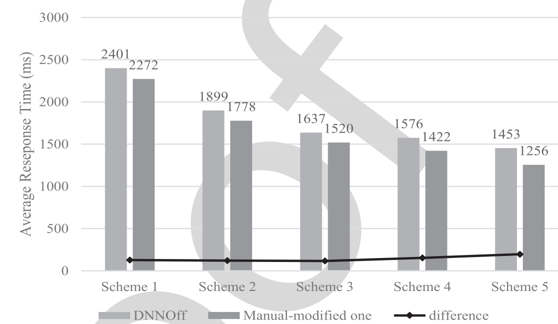

Fig. 7. Overhead of DNNOff and manual-modified one.

classification, and simulate five typical offloading schemes, which represent device-cloud, device-edge, and device-edge-cloud offloading, as shown in Table V.

b) Compared approaches: We evaluate the overhead of DNNOff by comparing the performance of the adaptive offloaded application with the manual-modified offloaded application for five typical offloading schemes. The adaptive offloaded application is dynamically offloaded according to the offloading scheme, which is supported by our framework. The manual modified one is implemented by separating the code according to the offload scheme case by case.

2) Results: We run the application in the five typical offloading schemes and, respectively, record their average response time, as shown in Fig. 7. We can see that the response time of DNNOff is similar to the manual modified one, but with an overhead of 120–150 ms. The slight increase of response time (under 10%) is due to the condition statements of pipes that are needed to go through for each layers execution in our framework. For instance, the overheads in cases 1–3 are all about 120 ms because the cutoff points of three offloading scheme are the same. The overhead in cases 4 and 5 are both 150 ms because there are two cutoff points in each offloading scheme, for which more condition statements need to be executed. Overall, the overhead is acceptable.

V. DISCUSSION

Some issues about applicability need to be further discussed.

A. Online Decision

For online decision, DNNOff uses the estimation model to calculate the response time given an offloading scheme, based on which the problem of online decision can be reduced to a traditional optimization problem. Some algorithms can be used to reduce overhead. For instance, it takes about minutes to determine the offloading decision for the genetic algorithm, while it just takes milliseconds to determine for the greedy algorithm; considering the performance and overhead of two algorithms, they are suitable to work in different situations [31]. However, this study mainly focuses on supporting DNN-based applications with the offloading capability in an MEC environment, and the issue above is orthogonal to the problem in this study. For future work, some state-of-the-art algorithms can be introduced to enhance our framework, such as deep reinforcement learning [32].

B. Energy Saving

Complex applications usually have many computation-intensive tasks and consume a great deal of energy. Although the battery capacity of end devices keeps growing continuously, it still cannot keep pace with the growing requirements of intelligent applications. Computation offloading is a popular technique to help reduce the energy consumption of intelligent application as well as improve its performance [10]. Because of space limitation, this article mainly focuses on performance improvement by offloading. For future work, energy consumption can be introduced to the objective function (see Section III-C) of our framework, wherein energy consumption reduced by offloaded computing and extra energy consumption caused by communication should be both considered [13], [14], [33].

VI. Industrial Applications

Recently, unmanned aerial/ground vehicles have begun to be applied in industrial IoT scenarios [34], such as patrolling the forest and delivering meals. These intelligent IoT applications have to rely on computer vision, whose cores are large-scale and complex DNNs, and thus, they commonly require sufficient resources and lead to high energy consumption. In the MEC environment, some computationally complex DNN layers are offloaded to the cloud or edges, while other tasks with simpler DNN layers are processed locally. This paradigm can improve performance of DNN-based intelligent IoT applications.

DNNOff first automatically translates the DNN-based application to a target program that is easier to offload. As the unmanned vehicle shifts during the day, its context (e.g., locations, network conditions, and available mobile edges) keeps changing. When the context changes, DNNOff synthesizes an optimal scheme for the intelligent application and then offloads its DNN layers according to the scheme.

VII. Conclusion

The DNN has become increasingly popular in intelligent IoT applications. Due to limited resources about computation and storage on end devices, complex DNN-based applications cannot be directly run on end devices. Although many researchers have considered partitioning DNN models between end devices and the cloud, we believe that it can completely release the potential of offloading DNN if an application can be partitioned at more cut-points and determine which parts shall be offloaded to MEC servers. With this insight, this article presents DNNOff, a novel approach that supports offloading DNN-based applications in MEC. DNNOff can enable a DNN-based application to run its different parts over the end device, the cloud, and edges and automatically determine the offloading scheme based on cost estimation. We evaluate DNNOff on a real-world intelligent IoT application with three DNN models. Results show that DNNOff can significantly reduce the response time.

References

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[2] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, "A guide to convolutional neural networks for computer vision," in *Synthesis Lectures on Computer Vision*, San Rafael, CA, USA: Morgan & Claypool, 2018, pp. 1–207.

[3] V. B. Cardoso *et al.*, "A large-scale mapping method based on deep neural networks applied to self-driving car localization," in *Proc. Int. Joint Conf. Neural Netw.*, 2020, pp. 1–8.

[4] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "DeepWear: Adaptive local offloading for on-wearable deep learning," *IEEE Trans. Mobile Comput.*, vol. 19, no. 2, pp. 314–330, Feb. 2020.

[5] F. Yang, J. Li, T. Lei, and S. Wang, "Architecture and key technologies for internet of vehicles: A survey," *J. Commun. Inf. Netw.*, vol. 2, no. 2, pp. 1–7, 2017.

[6] S. Jeong, O. Simeone, and J. Kang, "Mobile edge computing via a UAV-mounted cloudlet: Optimization of bit allocation and path planning," *IEEE Trans. Veh. Technol.*, vol. 67, no. 3, pp. 2049–2063, Mar. 2018.

[7] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1492–1499.

[8] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li, "Cost-driven offloading for DNN-based applications over cloud, edge, and end devices," *IEEE Trans. Ind. Informat.*, vol. 16, no. 8, pp. 5456–5466, Aug. 2020.

[9] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2017, pp. 615–629.

[10] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[11] O. Munoz, A. Pascual-Iserte, and J. Vidal, "Optimization of radio and computational resources for energy efficiency in latency-constrained application offloading," *IEEE Trans. Veh. Technol.*, vol. 64, no. 10, pp. 4738–4755, Oct. 2015.

[12] A. Yousafzai, A. Gani, R. M. Noor, A. Naveed, R. W. Ahmad, and V. Chang, "Computational offloading mechanism for native and android runtime based mobile applications," *J. Syst. Softw.*, vol. 121, pp. 28–39, 2016.

[13] E. Cuervo *et al.*, "MAUI: Making smartphones last longer with code offload," in *Proc. Int. Conf. Mobile Syst., Appl. Services*, 2010, pp. 49–62.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. Conf. Comput. Syst.*, 2011, pp. 301–314.

[15] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," *ACM Sigplan Notices*, vol. 47, no. 10, pp. 233–248, 2012.

[16] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surv. Tut.*, vol. 19, no. 4, pp. 2322–2358, Oct–Dec. 2017.

[17] X. Chen, J. Chen, B. Liu, Y. Ma, Y. Zhang, and H. Zhong, "AndroidOff: Offloading android application based on cost estimation," *J. Syst. Softw.*, vol. 158, 2019, Art. no. 110418.

[18] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun.*, 2018, pp. 31–36.

[19] C. Liu *et al.*, "A new deep learning-based food recognition system for dietary assessment on an edge computing service infrastructure," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 249–261, Mar./Apr. 2018.

[20] Z. Zhou, H. Liao, B. Gu, K. M. S. Huq, S. Mumtaz, and J. Rodriguez, "Robust mobile crowd sensing: When deep learning meets edge computing," *IEEE Netw.*, vol. 32, no. 4, pp. 54–60, Jul./Aug. 2018.

[21] P. G. Whiting and R. S. Pascoe, "A history of data-flow languages," *IEEE Ann. Hist. Comput.*, vol. 16, no. 4, pp. 38–59, Winter 1994.

[22] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[23] Z. Chai and C. Zhao, "Enhanced random forest with concurrent analysis of static and dynamic nodes for industrial fault classification," *IEEE Trans. Ind. Informat.*, vol. 16, no. 1, pp. 54–66, Jan. 2020.

[24] I. A. Ibrahim, M. Hossain, and B. C. Duck, "An optimized offline random forests-based model for ultra-short-term prediction of PV characteristics," *IEEE Trans. Ind. Informat.*, vol. 16, no. 1, pp. 202–214, Jan. 2020.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations*, 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[28] R. Silhavy, P. Silhavy, and Z. Prokopova, "Analysis and selection of a regression model for the use case points method using a stepwise approach," *J. Syst. Softw.*, vol. 125, pp. 1–14, 2017.

[29] M. Khodayar, O. Kaynak, and M. E. Khodayar, "Rough deep neural architecture for short-term wind speed forecasting," *IEEE Trans. Ind. Informat.*, vol. 13, no. 6, pp. 2770–2779, Dec. 2017.

[30] H. Jahangir *et al.*, "A novel electricity price forecasting approach based on dimension reduction strategy and rough artificial neural networks," *IEEE Trans. Ind. Informat.*, vol. 16, no. 4, pp. 2369–2381, Apr. 2020.

[31] Z. Chen, J. Hu, X. Chen, J. Hu, X. Zheng, and G. Min, "Computation offloading and task scheduling for DNN-based applications in cloud-edge computing," *IEEE Access*, vol. 8, pp. 115537–115547, Jun. 2020.

[32] Y. Liu, H. Yu, S. Xie, and Y. Zhang, "Deep reinforcement learning for offloading and resource allocation in vehicle edge computing and networks," *IEEE Trans. Veh. Technol.*, vol. 68, no. 11, pp. 11 158–11168, Nov. 2019.

[33] J. Wang, Y. Wang, D. Zhang, and S. Helal, "Energy saving techniques in mobile crowd sensing: Current state and future opportunities," *IEEE Commun. Mag.*, vol. 56, no. 5, pp. 164–169, May 2018.

[34] T. Yang, Z. Jiang, R. Sun, N. Cheng, and H. Feng, "Maritime search and rescue based on group mobile computing for unmanned aerial vehicles and unmanned surface vehicles," *IEEE Trans. Ind. Informat.*, vol. 16, no. 12, pp. 7700–7708, Dec. 2020.

Xing Chen (Member, IEEE) received the B.S. and Ph.D. degrees from Peking University, Beijing, China, in 2008 and 2013, respectively.

Since 2020, he has been a Professor with Fuzhou University, Fuzhou, China, where he is also the Deputy Director of the Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing and leads the Systems research group. His current projects cover the topics from self-adaptive software, computation offloading, model-driven approach, and so on. He has authored or coauthored more than 50 journal and conference articles. His research interests include software systems and engineering approaches for cloud and mobility.

Dr. Chen was awarded two First Class Prizes for Provincial Scientific and Technological Progress, separately, in 2018 and 2020.

Ming Li received the B.S. degree in computer science and technology in 2019 from Fuzhou University, Fujian, China, where he is currently working toward the M.S. degree in computer technology with the College of Mathematics and Computer Science.

Since September 2019, he has also been a part of the Fujian Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University. His current research interests include system software and edge computing.

Hao Zhong (Member, IEEE) received the Ph.D. degree from Peking University, Beijing, China, in 2009.

After graduation, he worked as an Assistant Professor with the Institute of Software, Chinese Academy of Sciences, and became an Associate Professor in 2012. From 2013 to 2014, he was a Visiting Scholar with the University of California, Davis, CA, USA. Since 2014, he has been an Associate Professor with Shanghai Jiao Tong University, Shanghai, China. His research interests include software engineering, with an emphasis on empirical software engineering and mining software repositories.

Dr. Zhong is a recipient of the ACM SIGSOFT Distinguished Paper Award 2009, the Best Paper Award of the 2009 IEEE/ACM International Conference on Automated Software Engineering, and the Best Paper Award of the 2008 Asia-Pacific Software Engineering Conference. His Ph.D. dissertation was nominated for the distinguished Ph.D. dissertation award of China Computer Federation. He is a Member of the ACM.

Yun Ma (Member, IEEE) received the B.S. and Ph.D. degrees from the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, in 2011 and 2017, respectively.

He is currently a Postdoctoral Researcher with the School of Software, Tsinghua University, Beijing. Currently, he focuses on synergy between the mobile and the Web, trying to improve the mobile user experience by leveraging the best practices from native apps and Web apps. His research interests include mobile computing, Web technologies, and service computing.

Ching-Hsien Hsu (Senior Member, IEEE) is currently a Chair Professor and the Dean of the College of Information and Electrical Engineering, Asia University, Taichung, Taiwan. He is also a Professor with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan. He has authored or coauthored 200 papers in top journals such as IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, *ACM Transactions on Multimedia Computing, Communications, and Applications*, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, IEEE SYSTEM, and IEEE NETWORK, top conference proceedings, and book chapters in these areas. He has been acting as an Author/Co-Author or an Editor/Co-Editor for ten books from Elsevier, Springer, IGI Global, World Scientific, and McGraw-Hill. His research interests include high-performance computing, cloud computing, parallel and distributed systems, big data analytics, and ubiquitous/pervasive computing and intelligence.

Prof. Hsu is a Fellow of the Institution of Engineering and Technology.