A DNN Inference Acceleration Algorithm in Heterogeneous Edge Computing: Joint Task Allocation and Model Partition

Lei Shi[1], Zhigang Xu[1(✉)], Yi Shi[2], Yuqi Fan[1], Xu Ding[3], and Yabo Sun[1]

[1] School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230009, China
xuzhig1995@163.com

[2] Intelligent Automation Inc., 15400 Calhoun Drive, Rockville, MD 20855, USA

[3] Institute of Industry and Equipment Technology, Hefei University of Technology, Hefei 230009, China

Abstract. Edge intelligence, as a new computing paradigm, aims to allocate Artificial Intelligence (AI)-based tasks partly on the edge to execute for reducing latency, consuming energy and improving privacy. As the most important technique of AI, Deep Neural Networks (DNN) has been widely used in various fields. And for those DNN based tasks, a new computing scheme named DNN model partition can further reduce the execution time. This computing scheme partitions the DNN task into two parts, one will be executed on the end devices and the other will be executed on edge servers. However, in a complex edge computing system, it is difficult to coordinate DNN model partition and task allocation. In this work, we study this problem in the heterogeneous edge computing system. We first establish the mathematical model of adaptive DNN model partition and task offloading. The mathematical model contains a large number of binary variables, and the solution space will be too large to be solved directly in a multi-task scenario. Then we use dynamic programming and greedy strategy to reduce the solution space under the premise of a good solution, and propose our offline algorithm named GSPI. Then considering the actual situation, we subsequently proposed the online algorithm. Through our experiments and simulations, we proved that our proposed GSPI algorithm can reduce the system time cost by at least 32% and the online algorithm can reduce the system time cost by at least 24%.

Keywords: Task allocation · Model partition · Edge computing · Edge intelligence

1 Introduction

As the most popular algorithm in deep learning, nowadays Deep Neural Networks (DNN) has achieved very good results in various fields, such as face recognition [1], autonomous driving [2], speech recognition [3], and so on. Especially

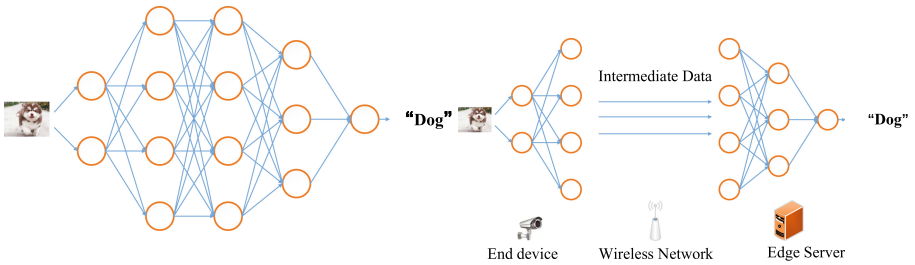
H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 349, pp. 237–254, 2021.
https://doi.org/10.1007/978-3-030-67537-0_15

with the advent of the Internet of Everything [4], the number of the embedded devices (such as mobile phones, wearables, smart cameras), which tend to do some DNN calculation jobs, are increased dramatically. However, these embedded devices are resource-constrained, and they do not have enough computing ability to run these DNNs. The traditional approach is based on cloud computing solution [5], which means the input data generated from end devices is uploaded to the cloud center for executing, and then results are sent back to the end devices after the DNN calculation finished on the cloud. But with this cloud-centric approach, a large amount of data needs to be transmitted to the cloud through the wide area network, and this will cause high latency and network congestion.

Fortunately, in recent years, a new computing paradigm called edge computing [6] prompts us some new ways to deal with these issues. In edge computing environment, computation tasks will be allocated to edge servers which are close to end devices for guaranteeing real-time performance. For example, in [7], authors proposed an edge computing framework based on video processing to deal with the high latency and network congestion problems in traditional cloud computing. In addition, edge computing has the potential to play a huge role in smart homes [8] and smart cities [9]. Many scholars have done researches on the resource scheduling and task allocations on edge computing. For example, in [10], authors proposed an online algorithm named OnDoc, which can satisfy the task scheduling requested deadline as much as possible. In [11], authors investigated a green mobile-edge computing (MEC) system with energy harvesting devices and developed an effective computation offloading strategy.

The concept of the edge intelligence was proposed in [12], which can be considered as the combination of the artificial intelligence and the edge computing. Edge intelligence has the characteristics that edge computing is close to the user, which can effectively solve the high latency problem caused by running DNN in cloud computing mode. In edge intelligence, an important research hotspot is to further accelerate DNN inference. Different researchers have done different works on this hotspot. Some studies aim at making lighter DNN models so that they can be deployed on resource-constrained devices. For example, in [13], authors proposed the method of compressing deep neural network with pruning, trained quantization and Huffman coding, and making DNN models easier to be deployed on embedded systems with limited hardware resources. In [14–17], scholars design lightweight networks by improving the structure of the network model. But making lighter DNN models may reduce the accuracy, so other studies aim at dividing the DNN inference model into two parts: one part is executed on the end device, and the other is executed on the edge server. We call this method as model partition [18]. For example, in [19], authors proposed a DeepWear model, which applied the DNN model partition to DNN-based wearable device applications, and achieved a good acceleration effect. In [20], based on BranchNet [21], authors proposed the method which combining the model partition and the model early exit, for providing the low-latency edge intelligence. In [22], authors proposed a partition method based on graph min-cut method,

and proposed the partition algorithms under the light workload and the heavy workload respectively. In [23], based on DNN model partition, authors further reduced the delay by encoding the feature of middle layer. In [24], authors proposed a cost-driven partition strategy for DNN-based applications over the cloud, edge and end devices, and the partition strategy effectively reduced the system cost within the corresponding deadline of each task. In [25], authors proposed the partition and offloading strategy which can make the optimal tradeoff between performance and privacy for battery-powered mobile devices. In Fig. 1, we show a simple DNN view (Fig. 1(a)) and its partition example (Fig. 1(b)).



(a) A simple DNN (b) Partition of the DNN

Fig. 1. A simple DNN & partition of the DNN

Previous work has made some contributions in the field of model partition. However, most of them only considered the situation of the single end device to single edge server pattern. In reality, the optimal partition point under the case of the single end device to single edge server pattern may not be suitable for the multi-device multi-server system. For example, the optimal partition point for the same task is different under the different system loads. So in this paper, different from other researchers' previous works, in addition to get an optimal solution, we also consider some good suboptimal partition points in our system. More concretely, tasks will adopt several different available partition points. For further analysis, to minimize time cost for tasks in the heterogeneous edge system, the process of the model partition and task allocation are inseparable. So we jointly consider model partitioning and task allocation, and try to design the algorithm to minimize the average time cost of all tasks.

Specifically, the main contributions of the paper are as follows: 1) We establish the mathematical model of adaptive DNN model partition and task offloading under the heterogeneous edge computing; 2) We use dynamic programming to solve complex problems and propose the Greedy Strategy for Progressive Inference (GSPI) algorithm; 3) Considering the actual situation, it is necessary to make the selection of the partition point and the allocation of tasks in the online situation, and we propose the corresponding online algorithm. Through simulation experiments, we find that using DNN model partition in

heterogeneous edge system can effectively reduce the time cost. And both offline and online algorithms can achieve good performance.

The rest of this paper is organized as follows: In Sect. 2, we introduce our system model and define our problem. In Sect. 3, we give the model partition algorithm and task allocation algorithm respectively. In Sect. 4, we give the simulation results and analyze them. In Sect. 5, we summarize this paper.

2 System Model and Problem Definition

2.1 DNN Model Partition and Network

Consider a two-dimensional network consists of u servers and n end devices. Denote $s_i(s_i \in S, i = 1...u)$ as one of the servers, and $e_i(e_i \in E, i = 1...n)$ as one of the end devices. As shown in Fig. 2, we consider the edge servers are heterogeneous, which means each server has a different calculation ability. While end devices are the same type, which means they have the same calculation ability. Consider end devices transmit data to edge servers via wireless network. Suppose end devices may have one or several calculation tasks in the whole scheduling time T. We divide T into h time slots, and define $\tau_i(\tau_i \in T, i = 1...h)$ as one of the time slot. In general, suppose the length of all time slot is the same, i.e., $\tau_1 = \tau_2 = ... = \tau_h$. Consider each task may start in a different time slot, and there are N tasks in the whole scheduling time T. Denote m_i^j as one of the task, where i means the task is from e_i, and j means the task starts from τ_j.

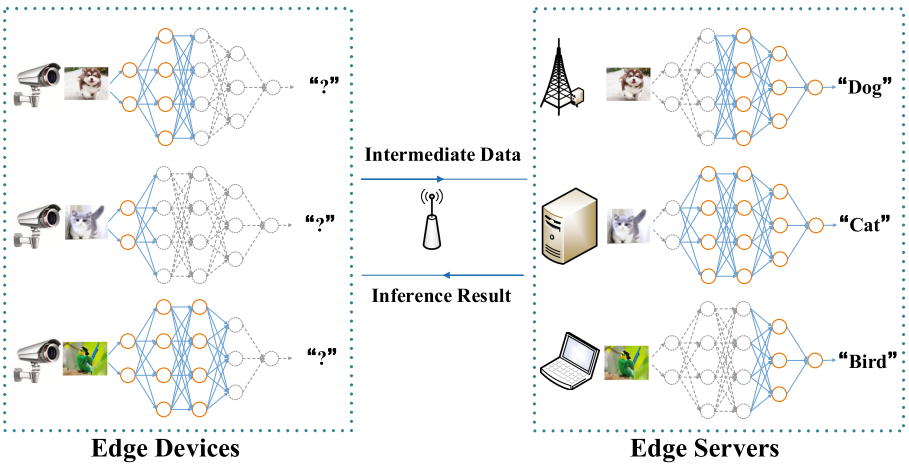


Fig. 2. Edge computing network with heterogeneous servers

Suppose all tasks use a same DNN model for calculating, and suppose this DNN model has v layers. Denote $r_i(r_i \in R, i = 1...v)$ as one of the layer, and denote d_0 as the initial input data and d_i as the output data for layer r_i.

For each server, only one task can be calculated at a time, which means when the server processes multiple tasks, the other tasks will not be executed until the previous task is completed.

By using model partition, the DNN calculation for each task can be divided into two parts. As shown in Fig. 1(b), the first part will be calculated on the end device, and the second part will be calculated on a server. Each task may use any partition point (as shown in Fig. 2). Suppose all devices can connect directly with all servers in this region. We want to decide each task's partition point and calculating server, so that all tasks can be calculated out as soon as possible.

2.2 Problem Formulation

Table 1. Notations

Variables	Meaning
B	The bandwidth of the system;
d_0	The input data of the DNN;
d_i	The output data of the layer r_i;
s_i	One of the edge server $i \in \{1, 2, 3, ..., u\}$;
e_i	One kind of end devices $i \in \{1, 2, 3, ..., n\}$;
τ_i	One of the time slots $i \in \{1, 2, 3, ..., h\}$;
r_i	One of the layer of the network $i \in \{1, 2, 3, ..., v\}$;
m_i^j	The task from the end device e_i at the j-th time slot;
N	The total number of the all tasks;
$t(m_i^j)$	The total time to complete the task m_i^j;
$t_d(m_i^j)$	The executing time of the end device part for task m_i^j;
$t_t(m_i^j)$	The transmission time of the intermediate data task m_i^j;
$t_w(m_i^j)$	The waiting time for executing the task m_i^j on the edge server;
$t_s(m_i^j)$	The executing time of the edge server part for task m_i^j;
t_D^α	The time cost for end device to execute the r_α layer of the DNN;
$t_{S_k}^\beta$	The time cost for edge server to execute the r_β layer of the DNN;
$J_{m_i^j}^k$	The executing time on the k-th edge server for task m_i^j;
$t_r(m_i^j)$	The time of returning result for task m_i^j;

For the task m_i^j, if the end device i does not generate a task at time slot j, then the task m_i^j does not exist. Here we use $E(m_i^j)$ to indicate whether there is a task m_i^j, and it can be expressed as (Table 1)

$$E(m_i^j) = \begin{cases} 1 : \text{the end device } i \text{ generates a task at time slot } j; \\ 0 : \text{otherwise.} \end{cases}$$

The total time to complete the task m_i^j can be expressed as the following

$$t(m_i^j) = t_d(m_i^j) + t_t(m_i^j) + t_w(m_i^j) + t_s(m_i^j) + t_r(m_i^j), \tag{1}$$

where $t_d(m_i^j)$ indicates the time cost for executing on the end device e_i, $t_t(m_i^j)$ indicates the time cost for data transmission to a server, $t_w(m_i^j)$ is the waiting time for execution, $t_s(m_i^j)$ is the execution time of task m_i^j on edge server, and $t_r(m_i^j)$ is the time for returning results. In the following we give the specific formulation for each item. We will first give the simple items for the first, the second, and the last item, then give the complex items for the forth and the third item.

For the first item $t_d(m_i^j)$, we use a binary variable $x_p(m_i^j)$ to indicate whether the task m_i^j is partitioned at the r_p layer of the DNN, i.e.

$$x_p(m_i^j) = \begin{cases} 1 : E(m_i^j) = 1 \text{ and } m_i^j \text{ is partitioned at the } r_p \text{ layer of the DNN;} \\ 0 : \text{otherwise.} \end{cases}$$

Note that $\sum_{p=1}^{v} x_p(m_i^j) = 1$. Then the first item $t_d(m_i^j)$ can be expressed as

$$t_d(m_i^j) = \sum_{p=1}^{v} \big(x_p(m_i^j) \cdot \sum_{\alpha=1}^{p} t_D^\alpha\big), \tag{2}$$

where t_D^α is the time cost for the end device to execute the r_α layer. Since all end devices are the same type, so different end devices have a same t_D^α.

For the second item $t_t(m_i^j)$, it can be calculated by

$$t_t(m_i^j) = \sum_{p=1}^{v} \big(x_p(m_i^j) \cdot \frac{d_p}{B}\big), \tag{3}$$

where d_p is the output data of layer r_p and B is the bandwidth.

For the last item $t_r(m_i^j)$, it is the time cost to return inference result and can be considered as a constant.

For the forth item $t_s(m_i^j)$, it will be decided by the server which executing task m_i^j eventually. Denote a binary scheduling variable $y_k(m_i^j)$ to indicate whether the task m_i^j is executed on the server s_k, i.e.,

$$y_k(m_i^j) = \begin{cases} 1 : E(m_i^j) = 1 \text{ and } m_i^j \text{ is executed on the server } s_k; \\ 0 : \text{otherwise.} \end{cases}$$

Apparent we have $\sum_{k=1}^{u} y_k(m_i^j) = 1$, and we have

$$t_s(m_i^j) = \sum_{k=1}^{u} \big(y_k(m_i^j) \cdot J_{m_i^j}^k\big), \tag{4}$$

where $J^k_{m_i^j}$ is the executing time on the k-th edge server for task m_i^j. $J^k_{m_i^j}$ can be expressed as

$$J^k_{m_i^j} = \sum_{p=1}^{v} \big(x_p(m_i^j) \cdot \sum_{\beta=p+1}^{v} t^{\beta}_{s_k}\big), \tag{5}$$

where $t^{\beta}_{s_k}$ is the time cost for edge server s_k to execute the r_β layer of the DNN.

For the third item $t_w(m_i^j)$, denote $AS(m_i^j)$ as the arriving time of the task m_i^j, we have

$$AS(m_i^j) = j + t_d(m_i^j) + t_t(m_i^j) = j + \sum_{p=1}^{v} \big(x_p(m_i^j) \cdot (\sum_{\alpha=1}^{p} t^{\alpha}_D + \frac{d_p}{B})\big). \tag{6}$$

Since when task m_i^j starts to run on the server, all tasks $m_{i'}^{j'}$ come to this server before $AS(m_i^j)$ should be accomplished. Here we use a binary variable $z_t^1(m_i^j)$ to judge the arrival time of a task, and $z^1_{AS(m_i^j)}(m_{i'}^{j'}) = 1$ means that the task $m_{i'}^{j'}$ arrives the edge server before the task m_i^j. For those tasks arriving at the edge server at the same time slot, we use another binary variable $z_t^2(m_i^j)$ to represent them. $z^2_{AS(m_i^j)}(m_{i'}^{j'}) = 1$ means that the task $m_{i'}^{j'}$ arrives the edge server at the same time slot as the task m_i^j. Note that in order to solve the execution order of the tasks satisfied $z_t^2(m_i^j)$, we adopt the first-generation-first-serving (FGFS) strategy for these tasks. These two binary variables are expressed as follows:

$$z_t^1(m_i^j) = \begin{cases} 1 : AS(m_i^j) < t; \\ 0 : \text{otherwise.} \end{cases}$$

$$z_t^2(m_i^j) = \begin{cases} 1 : AS(m_i^j) = t; \\ 0 : \text{otherwise.} \end{cases}$$

Then the third item $t_w(m_i^j)$ can be expressed as

$$\begin{aligned} t'_w(m_i^j) = \sum_{k=1}^{u} \Bigg(y_k(m_i^j) \cdot \Big(\sum_{j'=1}^{h} \sum_{i'=1}^{n} \big(y_k(m_{i'}^{j'}) z^1_{AS(m_i^j)}(m_{i'}^{j'}) \cdot (J^k_{m_{i'}^{j'}})\big)\Big) \\ + \Big(\sum_{j'=1}^{j-1} \sum_{i'=1}^{n} \big(y_k(m_{i'}^{j'}) z^2_{AS(m_i^j)}(m_{i'}^{j'}) \cdot (J^k_{m_{i'}^{j'}})\big)\Big)\Bigg) - \big(AS(m_i^j) - 1\big). \end{aligned} \tag{7}$$

In (7), it is composed of three parts. The first part is the total time cost of all tasks that arrive at the server before the task m_i^j, under the premise of being allocated to the same server as the task m_i^j. The second part is the total time cost of all tasks that arrive at the server at the same time as task m_i^j but were generated before the task m_i^j, under the premise of being allocated to the same server as the task m_i^j. The last part is the time that the system has been running

until the task m_i^j arrived at the server. Apparently the value of $t_w(m_i^j)$ needs to be positive, i.e:

$$t_w(m_i^j) = \max\{0, t'_w(m_i^j)\}. \tag{8}$$

Then we can get the average time of all tasks in the system,

$$t_{Average} = \frac{1}{N} \sum_{i=1}^{n} \sum_{j=1}^{h} t(m_i^j). \tag{9}$$

Then our problems can be formulated as,

$$\begin{array}{l} \min \quad t_{Average} \\ \text{s.t.}(2)(3)(4)(5)(6)(7)(8)(9) \\ \sum_{p=1}^{v} x_p(m_i^j) = 1 \quad (\forall i \in E, j \in T \& E(m_i^j) = 1) \\ \sum_{k=1}^{u} y_k(m_i^j) = 1 \quad (\forall i \in E, j \in T \& E(m_i^j) = 1). \end{array} \tag{10}$$

In (10), $E(m_i^j)$, d_i, d_p, and other symbols are all constants or determined values for specific network. $x_p(m_i^j)$ and $y_k(m_i^j)$ are binary variables. However, these binary variables almost appear in all items with different forms. These make the original problem model complex and hard to be solved directly. So for the problem to be solved in polynomial time, we need to give further analysis and find some way to reduce the complexity of the original problem.

3 Algorithms

In the last section, we give the original problem model and show that it is difficult to be solved directly. In this section, we will try to find some feasible solution for the whole network. Firstly, in order to get the best possible experimental result, we designed an offline algorithm. In the offline algorithm, all the $E(m_i^j)$ can be know in advance, that is, we know the generation time of all tasks. However, in a real network with multiple end devices and multiple edge servers, we may only know tasks that have been generated or are generating, but we do not know tasks that will be generated. In other words, for a time τ_k, we know all $E(m_i^j)$ where $(i = 1 \ldots n)$ and $(j \leq k)$, but we know nothing about $E(m_i^j)$ where $(j > k)$. That means in reality we need to design an online algorithm with only the knowledge about the history and the current tasks. In the following, we first design the offline algorithm in Subsect. 3.1. Then in Subsect. 3.2, we design the online algorithm.

3.1 Offline Strategy

For the offline strategy, suppose we know all $E(m_i^j)$. When considering all tasks directly, the solution space is infinite, since there are $(u * v)^N$ combinations of N task strategies. To design an offline algorithm in polynomial time, we must reduce the solution space. Here we consider two aspects to reduce the solution space.

First, for a DNN model with v layers, each layer may be the position of the partition point. However, we find that most of layers have no potential to be partition points. Because the intermediate data for these layers are too large, and partition in these layers may cause high transmission cost. So we need to design an algorithm to exclude these layers. We call the algorithm as the Partition-Points-Selection (PPS) algorithm. Through PPS algorithm, we can reduce the solution space size in a certain extent.

Second, based on the dynamic programming, we try to reduce the solution space further. The main idea is first generated tasks will be first handled. We call the designed algorithm based on this idea as the Greedy Strategy for Progressive Inference (GSPI) algorithm.

Now we give the details.

The PPS Algorithm

For the layers that have no potential to be partition points, we can use our Partition-Point-Selection algorithm to exclude them in advance, that is, we will select a few good partition points. Then in order to determine which partition points will be selected, we need to define a time standard *Latency*. Here, if a partition point is a good partition point, then it needs to satisfy that the total time cost of the task $t(m_i^j)$ is less than *Latency* without considering the waiting time $t_w(m_i^j)$. In this way, we will get a suitable set of partition points SP_u for each server s_u.

But it is worth noting that there are multiple servers with different computing capabilities in our system, and for a same task, the total cost time at the same partition point is different when different servers are selected. So for the whole system, the suitable partition points should meet latency requirements on all servers. Therefore, we need to use the common suitable partition points on all servers as the final set of suitable partition points SP.

The detailed steps are in Algorithm 1. After get the all available partition points, each task can choose any partition point in SP. Through Partition-Points-Selection algorithm, we can greatly reduce the choice of unreasonable partition points, thereby reducing the solution space.

The GSPI Algorithm

Now we discuss the GSPI algorithm. After using the PPS algorithm, we have shrunk the solution space for $x_p(m_i^j)$. However, there are still lots of variables of $x_p(m_i^j)$, $y_k(m_i^j)$ in (10). Though it is impossible to determine all these variables once, it is possible to determine them step by step by greedy. In detail, suppose we need to allocate a task generated in the first time slot. As the first generated task, it can select any possible partition point and can select any edge server without calculating $t_w(m_i^j)$, since $t_w(m_i^j) = 0$ now. We can calculate all values and select the best one as this task's allocating strategies. We can continue doing this calculation in the following time slot based on the strategies of tasks in the previous time slots. The most difficult calculating item in (10) is $t_w(m_i^j)$. But by using the GSPI algorithm, when we try to allocate a task m_i^j, all $t_w(m_{i'}^{j'})$ where

Algorithm 1. Partition-Points-Selection algorithm

Input:

S: Optional Server number; B: Bandwidth; V: the total layer for the network; d_i: the output data of the layer $ri, i \in \{1, 2, 3, ..., v\}$; t_d^i; executing time for the layer ri on the end device; $t_{s_u}^i$: executing time for the layer ri on the end server s_u, $u \in \{1, 2, 3, ..., N\}$; *Latency*:latency requirement for the system;

Output: the optional partition points SP in the system

for $u \in S$ **do**
  for $v \in V$ **do** $T \leftarrow \sum_{i=0}^{v} t_d^i + \frac{d_v}{B} + \sum_{k=v+1}^{V} t_{s_u}^k$
    if $Latency \geq T$ **then**
      Recode the partition point for server s_u in SP_u;
    end if
  end for
end for
$SP \leftarrow \bigcap_{u \in \{1,2,3,...,N\}} SP_u$
return SP

$(j' < j)$ have already been determined. And the solution space is been reduced further.

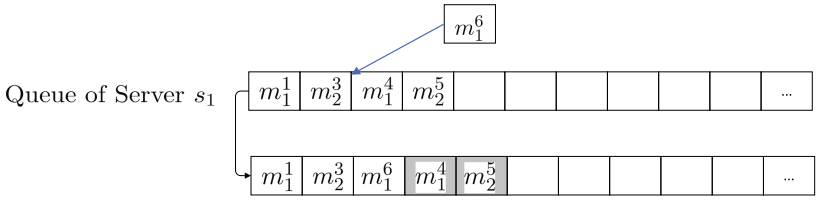


Fig. 3. Insert a task to the server's queue. A task with a gray background indicates that its strategy is improvable.

Notice that we may meet the situation that task m_i^j generated after task $m_{i'}^{j'}$, i.e., $(j > j')$, but it reach to the same edge server before $m_{i'}^{j'}$, i.e., $AS(m_i^j) < AS(m_{i'}^{j'})$. For example, in Fig. 3, the task m_1^6 is assigned to the server after the task m_1^4 and m_2^5, but will arrive before the task m_1^4, i.e. $AS(m_1^6) < AS(m_1^4) < AS(m_2^5)$. Obviously the strategies of the task m_1^4 and m_2^5 are improvable because m_1^6 was miss-considered when strategy center made their strategies. For this situation, the offline algorithm can remake the strategy for improving the performance. The task m_1^4 and m_2^5 will be removed from the queue of the server and their strategies will be remade. We will consider this situation in our GSPI algorithm.

Based on the above ideas, the GSPI algorithm can be summarized into the following three steps:

Algorithm 2. The GSPI Algorithm

Input: S:Optional servers; Q:Task Queue of edge servers; M:the tasks stack; J:cost Matrix; SP:Optional partition points.

Output: the partition and allocation strategy for the task in M.

```
1: while M!=Empty do
2:     M' ← M.pop();
3:     while M'! = NULL do
4:         for m_i^j ∈ M' do
5:             get the strategy according to arg min_{k∈S;p∈SP} t(m_i^j);
6:             insert the tasks into Q;
7:             update the statu of edge server Q.
8:         end for
9:     end while
10:    for m_i^j in Q do
11:        if the strategy of m_i^j is improvable then
12:            add m_i^j into Q'.
13:        end if
14:    end for
15:    for m_i^j in Q do
16:        remake the strategy for m_i^j.
17:    end for
18:    update Q
19: end while
```

Step 1: Initialization. Put all tasks into the task stack M in units on their generated time. The first generated task is at the top of the stack, and the last generated task is at the bottom of the stack.

Step 2: Get the Tasks' Strategies. Take out all tasks on the top of the stack M. Calculate their optimal strategies one by one. Note that when the strategy of one task is generated, the strategy-making center will update the status of each edge server.

Step 3: Check the improvable Strategies. After the strategies of all tasks for a stack frame are completed, we need to check whether there is any task with improvable strategy, such as the task m_1^4 and the task m_2^5 in Fig. 3. For these tasks with improvable strategies, they should be removed from the servers' queues, and the strategies should be remade.

The step 2 and step 3 will be repeated until the stack M is empty. The GSPI algorithm is shown in Algorithm 2. Line 3–9 are the process of a single time slot task, and line 10–17 are the process of checking strategies and remaking the strategies.

3.2 Online Strategy

For the online strategy, we only know tasks that have been generated and are being generated. And when the tasks are generated, we should make their strategies in real time. Here we propose a strategy-making architecture as shown in

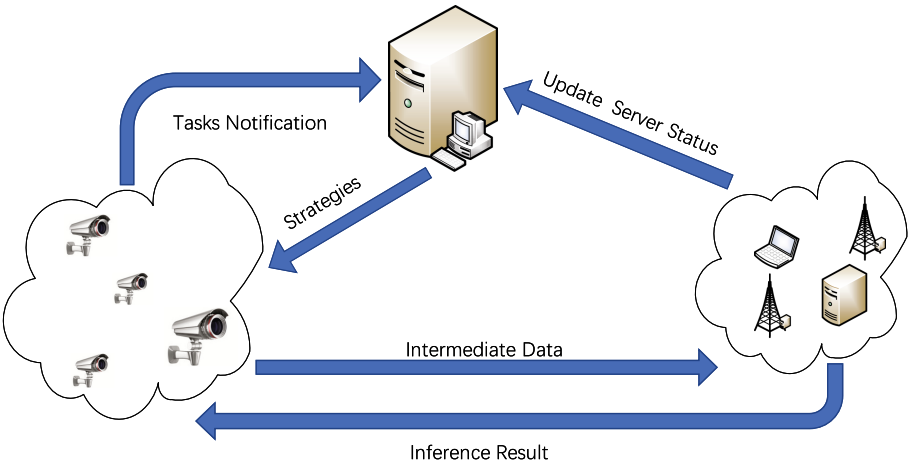


Fig. 4. Strategy model

Fig. 4. We use a much powerful edge server as the strategy-making center. As the end devices generate tasks, the strategy center will decide the tasks calculating strategies after receiving the notification. For the online strategy, the strategy center does not know the future tasks when making strategies. So formula (7) should be modified accordingly,

$$t'_w(m_i^j) = \sum_{k=1}^{u}\left(y_k(m_i^j)\cdot\left(\sum_{j'=1}^{j}\sum_{i'=1}^{n}\left(y_k(m_{i'}^{j'})\cdot(J^k_{m_{i'}^{j'}})\right)\right)\right)-\left(AS(m_i^j)-1\right). \quad (11)$$

Here the modified formula ignores the tasks generated after time slot j to meet the characteristics of real-time strategies-making.

Most steps of the online algorithm is the same as the offline algorithm. But unlike the offline algorithm, since we do not have the knowledge about the future tasks, and we need to calculate each task in-time, so we will not consider the situation that task m_i^j generated after task $m_{i'}^{j'}$ but reaches to the same edge server before $m_{i'}^{j'}$. That means, if this situation occurs, task $m_{i'}^{j'}$ will be added to the server stack directly and wait to be executed.

The heuristic algorithm of our online algorithm is shown in Algorithm 3. We use an auxiliary task queue Q' temporarily save the state of server queue. After the strategies for all tasks in a time slot are formulated, copy the status of Q' to Q. In this way, the strategy center can return the strategies of tasks at the same time.

3.3 Complexity Analysis

In this subsection, we will give the complexity analysis of three algorithms. For the Algorithm 1, there are u servers in the system, and the DNN model has v

Algorithm 3. Online Algorithm

Input: S:Optional servers; Q:Task Queue of edge servers; M_τ:Tasks generated at the current momentτ; J:cost Matrix; SP:Optional partition points.

Output: The strategies for the all tasks in M_τ and the updated Q.

1: $\kappa \leftarrow length(M_\tau)$
2: **if** $\kappa > 1$ **then**
3: $Q' \leftarrow Q$
4: **while** M_τ!=NULL **do**
5: $m_i^j \leftarrow M_\tau.pop()$
6: Get the best strategy according to $\underset{k\in S;p\in SP}{\arg\min}\ t(m_i^j)$;
7: Update the Q'.
8: **end while**
9: $Q \leftarrow Q'$
10: **else**
11: Get the best strategy according to $\underset{k\in S;p\in SP}{\arg\min}\ t(m_i^j)$;
12: **end if**
13: Update the Q.

layers. It takes $u * v$ times to calculate each possible combination, and $u * v$ can be treated as a constant. So the time complexity of Algorithm 1 is $O(1)$.

For the Algorithm 2, to get the strategy of a single task in line 5, we need to calculate the total time consumption of the task $t(m_i^j)$. From formula 1 and the corresponding expansion, we can get that except for $t_w(m_i^j)$, the other items are all constant items. To calculate $t_w(m_i^j)$, the formula 7 tell us that all tasks need to be considered. So for any task, the time complexity of computing $t_w(m_i^j)$ is $O(N)$, where N is the total number of the tasks. Then get the best strategy according to $\underset{k\in S;p\in SP}{\arg\min}\ t(m_i^j)$, the time complexity come to $O(a*N)$, where a is a constant related to the set of servers S and the set of available partition points SP. And there are N tasks, so the final time complexity is $O(a*N^2)$, which can be expressed as $O(N^2)$.

For the Algorithm 3, to get the strategy of a single task in line 6, it is the same as Algorithm 2, that is, the main complexity lies in $t_w(m_i^j)$. To calculate $t_w(m_i^j)$, the formula 11 tell us that all tasks generated before task m_i^j need to be taken into account, that is, to calculate the $t_w(m_i^j)$ of the N-th task, the strategies of the first N-1 tasks need to be considered. So for all tasks, their average time complexity of computing $t_w(m_i^j)$ is $O(N/2)$, where N is the total number of the tasks. Then get the best strategy according to $\underset{k\in S;p\in SP}{\arg\min}\ t(m_i^j)$, the time complexity come to $O(a*N/2)$, where a is a constant related to the set of servers S and the set of available partition points SP. Finally, there are N tasks in the entire process, so the above content has to be repeated N times, and the final time complexity is $O(a*N^2/2)$, which can be expressed as $O(N^2)$.

4 Simulation and Experiment

In this section, we introduced the related work of experiments and simulations. The DNN model used in our experiments is VGG16 [26]. The model is trained using the cifar-10 dataset, and the framework used is pytorch. All tasks in our system is the inference task based on VGG16. We use the computing power of a CPU to simulate the computing power of our end device, and use the GPUs on different computers as our servers. To get the constants in Sect. 2, such as t_D^{α}, $t_{S_k}^{\beta}$, d_i and so on. First we run a DNN inference on a CPU for 100 times and get the execution time of each layer of the DNN. Then we take the average result of these 100 experiments as our experimental data, i.e the value of t_D^{α}. The values of other constants are obtained by the same way. Notice that we set enough end devices and each end device have enough time to calculate the generated tasks, that is, there will be no waiting tasks on the device. We set 4 servers with computing capabilities in the system.

We first study these DNN-based tasks in Subsect. 4.1, and give experimental results of Partition-Points-Selection. Then based on these experimental results, we give further experimental setting. In Subsect. 4.2, we validate the proposed model and algorithm. We conducted experiments under different bandwidths and different number of tasks. Experiment result shows that compared to running the DNN in end-only mode, running the DNN in a multiple partition points mode can reduce system time cost by 32% on average. And compared to the server-only mode, the system time is reduced by 48% on average.

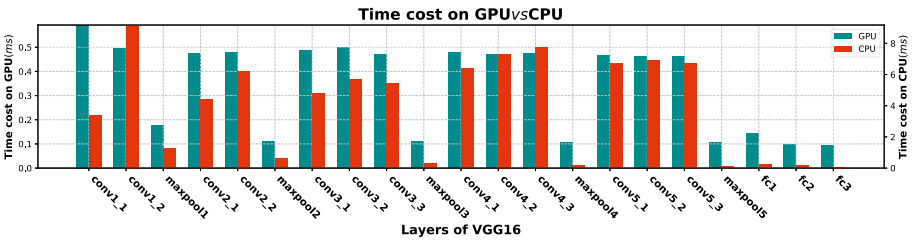


Fig. 5. Each layer's executing time of VGG16 on GPU and CPU

4.1 Results of Partition Points

First, we get the execution time of each layer of VGG16 on the end device and edge servers and get the amount of intermediate output data. Figure 5 shows the execution time of each layer of the model on CPU and one of the GPUs, and the time cost of running a task on the end device is more than ten times that on the edge server.

Then we get the total time cost of a DNN task at different partition points under the premise of ignoring the task waiting time. One of the result is shown

as Fig. 6, and finally we get 6 partition points. Compared with the original 20 partition points, this can greatly reduce the amount of calculation. The red stars in Fig. 6 indicate the selected partition points.

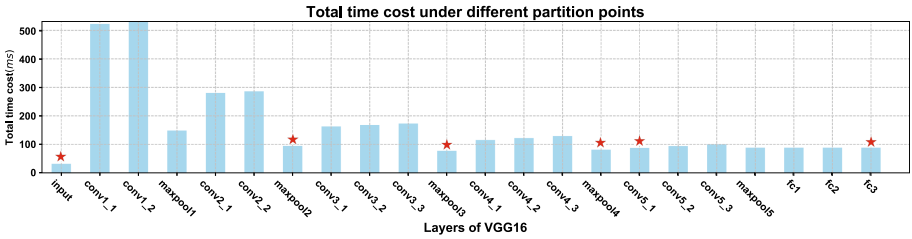


Fig. 6. The time cost of VGG16 at different partition points

Based on the above experiment result, we define 1 ms as one time slot. Then we use the time slot as our minimum time unit, and convert normal time representation into time slot representation. Table 2 shows the time cost for end device and edge servers at different partition points. The partition point $P1$ means the entire task will be completely uploaded to the server for calculation, and partition point $P6$ means that the entire task is calculated on the end device.

Table 2. Time cost of device and servers in different partition points

Partition schemes	End Device	Server1	Server2	Server3	Server4
P1	0	7	10	20	38
P2	25	5	7	13	27
P3	41	3	5	8	19
P4	62	2	3	4	9
P5	71	1	2	3	6
P6	88	0	0	0	0

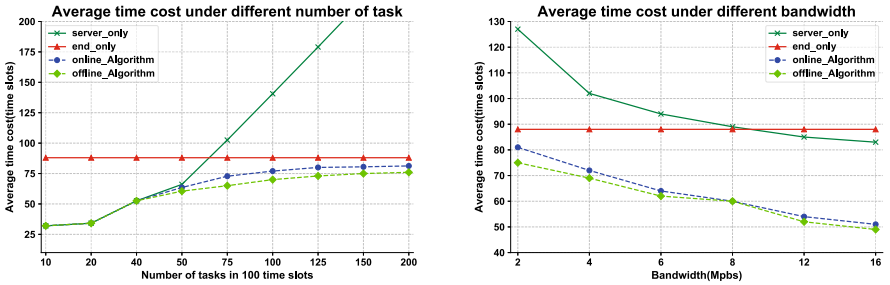
4.2 Experiment Results

Firstly we set time slots $h = 100$ and the bandwidth $B = 4$ Mbps. Then we set different number of tasks in 100 time slots, and run these tasks in end-only mode, server-only mode and our adaptive partition mode. The result is shown in Fig. 7(a).

When the number of tasks is relatively small, we can see the results of our algorithm are very close to the server-only mode. The reason for this is that when

there are few tasks, the waiting time of the queues on the servers are almost 0. This drives the system to run the entire task on the server. As the number of tasks gradually increases, the waiting time of the queue on the servers will increase greatly. As a result, the average task time cost in server-only mode will greatly increase. But our adaptive partition algorithm will push part of the task to be executed on the end device when the waiting time of the server is too long, so as to reduce the total time cost of the task.

According to our experimental results, compared to the end-only mode, our adaptive partition mode can reduce system time cost by 32% on average. And compared to the server-only mode, the system time is reduced by 48% on average. The result of our offline algorithm GSPI is always the best because it can not only adopt adaptive partition, but also can adjust the order of task execution. The online-algorithm we proposed also performs well. When the number of tasks is small, its performance is the same as the offline algorithm, and as the number of tasks increases, it cost at most 10% more system time than offline algorithms.



(a) Different number of tasks (b) Different bandwidth

Fig. 7. (a) The average time cost for different number of tasks. The bandwidth $B = 4$ Mbps. (b) The average time cost under different bandwidth. Number of tasks is set as 75 in 100 time slots.

And Fig. 7(b) shows that we set 75 tasks in 100 time slots and bandwidth B from 2 Mbps to 16 Mbps. From the result, we can know that higher bandwidth will produce less time cost. But the bandwidth does not affect the end-only mode, because no intermediate data needs to be uploaded to the edge server. It is worth noting that the adaptive partition mode we proposed performs better than the server-only mode at any bandwidth.

5 Conclusion and Future Work

In this paper, we studied the DNN model partition and task allocation in heterogeneous edge system. We first establish the mathematical model. The model contains a large number of binary variables, and the solution space will be too large to be solved directly. Then we proposed the adaptive DNN model partition

and task allocation algorithm named GSPI. Considering the real-time nature of the reality, we also proposed corresponding online algorithm. To the best of our knowledge, this is the first attempt to put the DNN model partition in a heterogeneous edge system, especially considering the waiting time of the server queue. Our simulation experiments proved that using model partition in heterogeneous edge systems can effectively reduce the time cost of DNN-based tasks.

This article is an attempt to introduce model partition into the heterogeneous edge system. In the future, we will consider that the end devices in the system have different computing capabilities, and the network environment from the device to the edge server is dynamically changing.

Acknowledgements. This article was supported by the National Key Research And Development Plan (Grant No. 2018YFB2000505), National Natural Science Foundation of China (Grant No. 61806067) and Key Research and Development Project in Anhui Province (Grant No. 201904a06020024).

References

1. Parkhi, O.M., Vedaldi, A., Zisserman, A.: Deep face recognition. In: BMVC (2015)
2. Chen, C., Seff, A., Kornhauser, A.L., Xiao, J.: DeepDriving: learning affordance for direct perception in autonomous driving. In: 2015 IEEE International Conference on Computer Vision (ICCV), pp. 2722–2730 (2015)
3. Chan, W., Jaitly, N., Le, Q.V., Vinyals, O.: Listen, attend and spell: a neural network for large vocabulary conversational speech recognition. In: 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 4960–4964 (2016)
4. Snyder, T., Byrd, G.: The internet of everything. Computer **50**(6), 8–9 (2017)
5. Pandey, P., Singh, S., Singh, S.: Cloud computing. In: ICWET (2010)
6. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J. **3**, 637–646 (2016)
7. Long, C., Cao, Y., Jiang, T., Zhang, Q.: Edge computing framework for cooperative video processing in multimedia IoT systems. IEEE Trans. Multimedia **20**, 1126–1139 (2018)
8. Deschamps-Sonsino, A.: Smarter Homes. Apress, New York (2018)
9. Alba, E., Chicano, F., Luque, G. (eds.): Smart-CT 2016. LNCS, vol. 9704. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39595-1
10. Liu, L., Huang, H., Tan, H., Cao, W., Yang, P., Li, X.-Y.: Online DAG scheduling with on-demand function configuration in edge computing. In: Biagioni, E.S., Zheng, Y., Cheng, S. (eds.) WASA 2019. LNCS, vol. 11604, pp. 213–224. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23597-0_17
11. Mao, Y., Zhang, J., Letaief, K.B.: Dynamic computation offloading for mobile-edge computing with energy harvesting devices. IEEE J. Sel. Areas Commun. **34**, 3590–3605 (2016)
12. Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., Zhang, J.: Edge intelligence: paving the last mile of artificial intelligence with edge computing. Proc. IEEE **107**, 1738–1762 (2019)
13. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. CoRR abs/1510.00149 (2015)

14. Howard, A.G., et al.: MobileNets: efficient convolutional neural networks for mobile vision applications. arXiv:abs/1704.04861 (2017)
15. Sandler, M., Howard, A.G., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: inverted residuals and linear bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4510–4520 (2018)
16. Zhang, X., Zhou, X., Lin, M., Sun, J.: ShuffleNet: an extremely efficient convolutional neural network for mobile devices. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 6848–6856 (2018)
17. Ma, N., Zhang, X., Zheng, H.-T., Sun, J.: ShuffleNet V2: practical guidelines for efficient CNN architecture design. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) Computer Vision – ECCV 2018. LNCS, vol. 11218, pp. 122–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01264-9_8
18. Kang, Y., et al.: Neurosurgeon: collaborative intelligence between the cloud and mobile edge. In: ASPLOS 2017 (2017)
19. Xu, M., Qian, F., Zhu, M., Huang, F., Pushp, S., Liu, X.: DeepWear: adaptive local offloading for on-wearable deep learning. IEEE Trans. Mob. Comput. **19**, 314–330 (2020)
20. Li, E., Zeng, L., Zhou, Z., Chen, X.: Edge AI: on-demand accelerating deep neural network inference via edge computing. IEEE Trans. Wireless Commun. **19**, 447–457 (2020)
21. Teerapittayanon, S., McDanel, B., Kung, H.T.: BranchyNet: fast inference via early exiting from deep neural networks. In: 2016 23rd International Conference on Pattern Recognition (ICPR), pp. 2464–2469 (2016)
22. Hu, C., Bao, W.S., Wang, D., Liu, F.: Dynamic adaptive DNN surgery for inference acceleration on the edge. In: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, pp. 1423–1431 (2019)
23. Ko, J.H., Na, T., Amir, M.F., Mukhopadhyay, S.: Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained Internet-of-Things platforms. In: 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), pp. 1–6 (2018)
24. Lin, B., Huang, Y., Zhang, J., Hu, J., Chen, X., Li, J.: Cost-driven off-loading for DNN-based applications over cloud, edge, and end devices. IEEE Trans. Industr. Inf. **16**, 5456–5466 (2020)
25. Shi, C., Chen, L., Shen, C., Song, L., Xu, J.: Privacy-aware edge computing based on adaptive DNN partitioning. In: 2019 IEEE Global Communications Conference (GLOBECOM), pp. 1–6 (2019)
26. Qassim, H., Feinzimer, D., Verma, A.: Residual squeeze VGG16. arXiv:abs/1705.03004 (2017)