Journal of Parallel and Distributed Computing 168 (2022) 108–119

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Towards scalable and efficient Deep-RL in edge computing: A game-based partition approach

Hao Dai [a,b], Jiashu Wu [a,b], Yang Wang [a,b,*], Chengzhong Xu [c]

[a] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, Guangdong, 518055, China
[b] University of Chinese Academy of Sciences, Beijing, 100049, China
[c] Faculty of Science and Technology, University of Macau, Taipa, Macau, China

ARTICLE INFO

Article history:
Received 31 August 2021
Received in revised form 19 April 2022
Accepted 9 June 2022
Available online 16 June 2022

Keywords:
Mobile edge computing
Deep reinforcement learning
DNN partition
Game theory

ABSTRACT

Currently, most edge-based Deep Reinforcement Learning (Deep-RL) applications have been deployed in the edge network, however, their mainstream studies are still short of adequate considerations on its limited compute and bandwidth resources. In this paper, we investigate the near on-policy of actions taking in distributed Deep-RL architecture, and propose a "hybrid near on-policy" Deep-RL framework, called *Coknight*, by leveraging a game-theory based DNN partition approach. We first formulate the partition problem into a variant of knapsack problem in device-edge setting, and then transform it into a potential game with a formal proof. Finally, we show the problem is NP-complete whereby an efficient distributed algorithm based on the potential game theory is developed from device perspective to achieve fast and dynamic partitioning. *Coknight* not only significantly improves the resource efficiency of the Deep-RL but also allows the inference to enforce the scalability of the actor policy. We prototype the framework with extensive experiments to validate our findings. The experimental results show that with the premise of a rapid convergence guarantee, *Coknight*, compared with *Seed-RL*, can reduce GPU utilization by 30% while providing large-scale scalability.

© 2022 Elsevier Inc. All rights reserved.

1. Introduction

Deep reinforcement learning (Deep-RL) [39,42], which integrates deep neural network (DNN) into a framework of reinforcement learning (RL) to help software agents learn how to reach their goals, has been widely deployed on various mobile or stationary devices, such as smartphones, wearables, smart cameras, and other intelligent objects to provide diverse digital services [31,40]. However, given the inherent constrains in energy consumption, chip size and compute capacity, these devices are usually SoC-based and characterized by limited performance [18,41], which as a result cannot fully exploit the potentials of Deep-RL. To address these issues, cloud center is usually introduced to develop a "cloud center + mobile device" architecture in traditional solutions. However, due to their high QoS requirements, this architecture has to struggle to make the trade-offs between network bandwidth, computation efficiency, and latency overhead for DNN-based service acceleration. Therefore, a new type of computation paradigm—mobile edge computing (MEC), which extends the capabilities of cloud platform to the edge of the network—was widely studied in recent years to exploit these capabilities to facilitate various computations [6,7,21,37]. As such, given the high requirements on compute capacities for DNN training and inference, it is of utmost important to conduct resource-efficient and scalable Deep-RL based on the MEC solution [30,34].

For the performance optimization of DNN, most works focus on reducing the computation overhead subject to the constraints on compute resources on devices. Some current mainstream methods typically tackle the issue via model compression [13,25], parameter quantization [14], model pruning [11,33], model early exit [35,36], and so on. These technologies are characterized by separating the training and inference of deep learning into different phases and focusing squarely on the performance optimization of the inference. In particular, the models are finely trained in servers, and the quantization and/or pruning are performed on the pre-trained model. Since Deep-RL implements model training through interaction with environments, it's necessary to perform corresponding actions in a certain environment to evaluate the new strategies to be used. Thus the inference and training are interleaved and inseparable in Deep-RL, which makes it infeasible for the above optimization methods to use in large-scale problem [8]. Much worse,

* Corresponding author at: Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, Guangdong, 518055, China.

E-mail addresses: hao.dai@siat.ac.cn (H. Dai), js.wu@siat.ac.cn (J. Wu), yang.wang1@siat.ac.cn (Y. Wang), czxu@um.mo.edu (C. Xu).

https://doi.org/10.1016/j.jpdc.2022.06.006
0743-7315/© 2022 Elsevier Inc. All rights reserved.

the interactions with the environment (including inference and action execution) often have high overheads, and consequently become the bottleneck of the whole system.

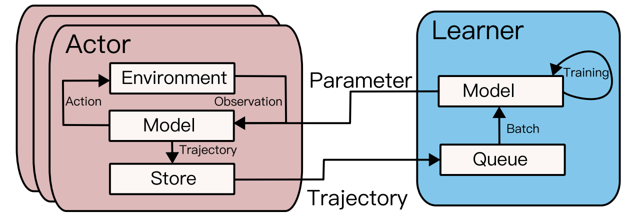
On the other hand, given the performance gap between GPU and CPU, the service acceleration through edge servers with GPU tends to be a promising approach. For instance, compared with CPU, the processing time is less than one tenth when the same DNN inference task is executed on GPU. As such, the current mainstream frameworks often support large-scale distributed Deep-RL in a CPU-GPU collaborative manner, such as *GA3C* [3], *DD-PPO* [38], *IMPALA* [9], *Seed-RL* [10], and so on. All these frameworks put the entire training and even the entire inference on the GPU for acceleration and leave the interaction with the environment on the CPU. Unfortunately, these frameworks are not designed for the edge networks as they lack the consideration of the limited edge resources. Although the migration of all tasks to the edge servers is feasible, it is not cost-efficient as it would cause wastage of network bandwidth and device computation resources, making it difficult to scale to large-scale clusters.

To address these issues, we design and implement a scalable and efficient Deep-RL framework, called *Coknight*, based on the idea of DNN partition [20,24] with the aid of the edge network to support intelligent applications deployed in resource-constrained mobile devices. DNN partition is an often-used method for cooperative inference acceleration between edge servers and mobile devices by flexibly splitting a DNN to both sides [20,24]. However, compared to existing partition methods which either adopt heuristic methods [24,28,33] or graph-based methods [20], Coknight bears some distinct features, which are short of adequate considerations in the existing studies: 1) Coknight is fairly efficient by accomplishing the partition in quadratic time; 2) Coknight is highly scalable by distributing the computation across all the devices; 3) Coknight is fully dynamic by adapting to the changes of the number of mobile devices. Moreover, to the best of our knowledge, we are the first to integrate the DNN partition into the Deep-RL framework to accelerate the model training.

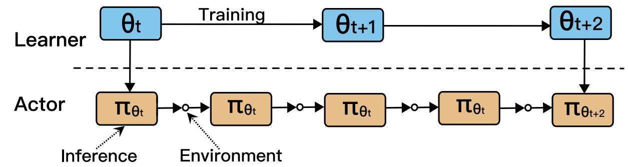
We design Coknight to achieve these features by overcoming the demerits of both *IMPALA* [9] and *Seed-RL* [10], two typical Deep-RL architectures designed for edge computing. To this end, we first formulate the partition problem into a variant of knapsack problem [19,26] in the device-edge setting, and then transform the problem into a multi-player game and prove it is a potential game. Finally, we show the problem is NP-complete whereby an efficient distributed algorithm based on the potential game theory is developed from device perspective to achieve fast and dynamic partitioning [5,17,23]. With Coknight, we can not only significantly improve the resource efficiency of the Deep-RL but also allow the inference to enforce the scalability of the actor policy. We prototype the framework with extensive experiments to validate our findings.

In summary, we make the following contributions:

- We design a collaborative large-scale actor-learner framework, called *Coknight*, that integrates DNN partition with Deep-RL to jointly accomplish the model inference between mobile device and edge server. We thus achieve a concept of "hybrid near on-policy" that can improve the resource efficiency and the parallelism of multiple actors with a rapid convergence guarantee.
- We formulate the partition problem into a variant of knapsack problem in the device-edge settings and prove its NP-Completeness. Then we transform the problem into a multi-player game and prove it is a potential game. According to the proofs, we further design a dynamic distributed algorithm with time complexity of $O(\mathcal{K}^2\mathcal{N}^2)$ for the DNN partition problem in *Coknight*, where $\mathcal{K}$ is the number of layers of DNN and $\mathcal{N}$ is the number of devices.
- We implement the *Coknight* framework and conduct extensive experiments on Atari Games [12] and a large-scale simulation study to validate its scalability and efficiency.



(a) *IMPALA* architecture



(b) off-policy of *IMPALA*

Fig. 1. The architecture of *IMPALA* (a) and the off-policy model of *IMPALA* (b). Data (model parameters and trajectories) transmission between the actor and the learner is asynchronous and timing.

The remainder of the paper is organized as follows. Section 2 introduces the background and the motivation of our work. After that, we go through the architecture of *Coknight* detailed in Section 3 and cover its dynamic partition algorithm in Section 4. We present the experimental results in Section 5 and discuss some related works in Section 6, which is followed by the conclusion in Section 7.

2. Background and motivation

In this section, we introduce some classic Deep-RL frameworks that inspire our proposed method. Most Deep-RL methods in the early era usually work in a simulator-based environment. Thus early Deep-RL operates in a synchronized single-agent architecture in the "online policy" manner, in which the agent leverages the latest trained policy to take action. In such architecture, the model can converge apace with few interactions. However, this single-agent design can barely scale out as a large amount of data training incurs significant overheads, compromising its overall performance.

To this end, a more general asynchronous architecture – actor-learner architecture – has been extensively used [2,32]. As opposed to the single-agent architecture, in this architecture, two agents are employed to work asynchronously – an actor is responsible for interacting with the environment and generating trajectories, while a learner conducts model training by updating the policy based on the generated trajectories. This so-called "off-policy" method dramatically improves the parallelism of the agents, which in turn boosts the efficiency of the Deep-RL. However, the convergence of this method is affected by the frequency of the model updates, which could require a large number of communication resources for the trajectories and parameter transmissions.

IMPALA. As shown in *Fig. 1(a)*, *IMPALA* [9] is a typical actor-learner architecture where the model training and inference occur in the learner and the actor, respectively. With an asynchronous inference mechanism, *IMPALA* can be easily scaled out. However,

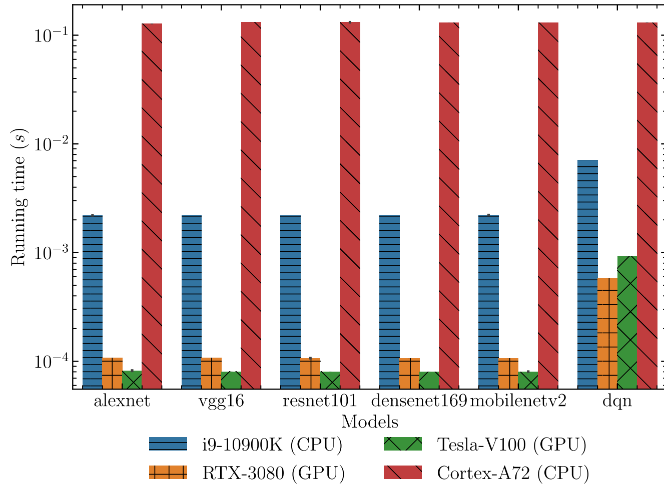
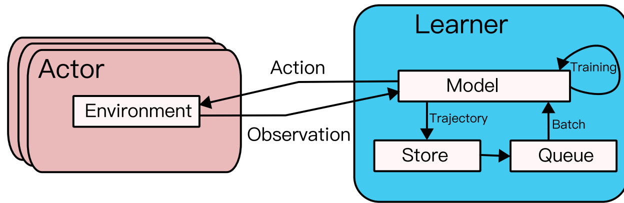
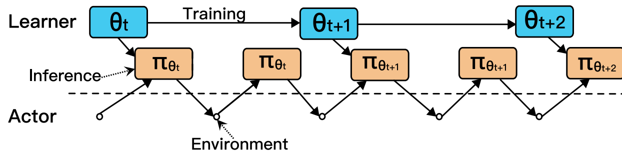


Fig. 2. Comparison of multiple classic models inference time on different CPU/GPU.



(a) *Seed-RL* architecture



(b) "near on-policy" of *Seed-RL*

Fig. 3. The architecture of *Seed-RL* (a) and the "near on-policy" of *Seed-RL* (b). Data (observations and actions) transmission between actor and learner is synchronous and real-time.

its shortcomings are also obvious: 1) the latency of CPU and GPU presents an order of magnitude gap, as shown in *Fig. 2*, and thus *IMPALA* cannot effectively benefit from the GPUs in the learner; 2) the adopted off-policy method usually leads to a large number of epochs when training Deep-RL model, as shown in *Fig. 1(b)*.

Seed-RL. To overcome these shortcomings, researchers proposed another framework called *Seed-RL* [10]. As shown in *Fig. 3(a)*, both the inference and training of *Seed-RL* work in the learner, making full use of GPU to accelerate the entire Deep-RL training process. Meanwhile, as illustrated in *Fig. 3(b)*, all the inferences in *Seed-RL* are performed using the same DNN parameters in the learner, resulting in a so-called *near on-policy* which converges within fewer epochs than *IMPALA*. *Seed-RL* bears some similarities with synchronous training methods because of the concentrated computations and the parallel environment. Thus, it is very efficient when using a simulator for training in the cloud center.

Motivation. Unfortunately, when we tried to adopt *Seed-RL* in the edge network, we found some crucial issues: 1) bandwidth resources in the edge network are always scarce, and the observation data are usually low-dimensional and redundant (e.g., images and voices), which could result in a high cost in data uploads; 2)

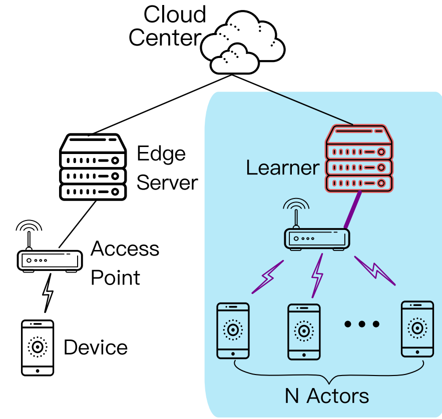


Fig. 4. In a typical edge computing architecture, multiple devices are connected to a specific edge server.

all the computations are performed in the learner, which requires massive computation resources on the edge server and becomes bottleneck-prone.

Since *IMPALA* and *Seed-RL* have troubles in resource inefficiency and weak scalability, respectively, we intend to combine their advantages to address these issues. With the guarantee of convergence efficiency, we can schedule computational tasks in an adaptive way based on the available compute and communication resources to improve the parallelism of actors and the resource efficiency of the edge servers. We will provide details of the approach in the next section.

3. Coknight and DNN partition

This section introduces the design details of our proposed actor-learner framework, *Coknight*, focusing on a formal description of the DNN partition problem inside.

3.1. Architecture overview

As illustrated in *Fig. 4*, a MEC network consists of an edge server and a set $D = \{d_1, d_2, ..., d_{\mathcal{N}}\}$ of $\mathcal{N}$ devices. Devices communicate with the edge server over WiFi, 5G, or dedicated links. The edge server has certain resource constraints, including compute capacity, bandwidth, etc. Let $\mathbb{R}_c$ and $\mathbb{R}_b$ be the total amount of the edge server's computation and bandwidth resources, respectively. With this model, we intend to formalize a DNN as a set of tasks that consume computation and bandwidth resources, and allocate them between devices and servers.

Fig. 5(a) is an example of an image classification neural network composed of a total of 11 layers. The type and scale of calculation for each layer are marked on the figure, including 9 3×3-CNN layers and 2 FC layers. In fact, all DNNs can be formulated into a similar architecture, with each layer performing a corresponding tensor calculation and exporting the results to the next layer of the network. Since the essence of DNN inference is a forward calculation with input tensors, the computation of each layer is proportional to the scale of input tensors. We can measure the calculation of each layer using the number of floating-point operations (FLOPs) [29].

We can define a $\mathcal{K}$-layer DNN as a sequence $L = (l_1, l_2, ..., l_{\mathcal{K}})$, l_k represents the kth layer of the neural network. Let c_k denote the FLOPs of l_k, which is a function taking tensor size as input. Indeed, the FLOPs are associated with not only the input tensor size but also the type of layer. The details of the formula calculation for FLOPs can be found in [29]. In short, for a certain DNN, the amount

Journal of Parallel and Distributed Computing 168 (2022) 108–119

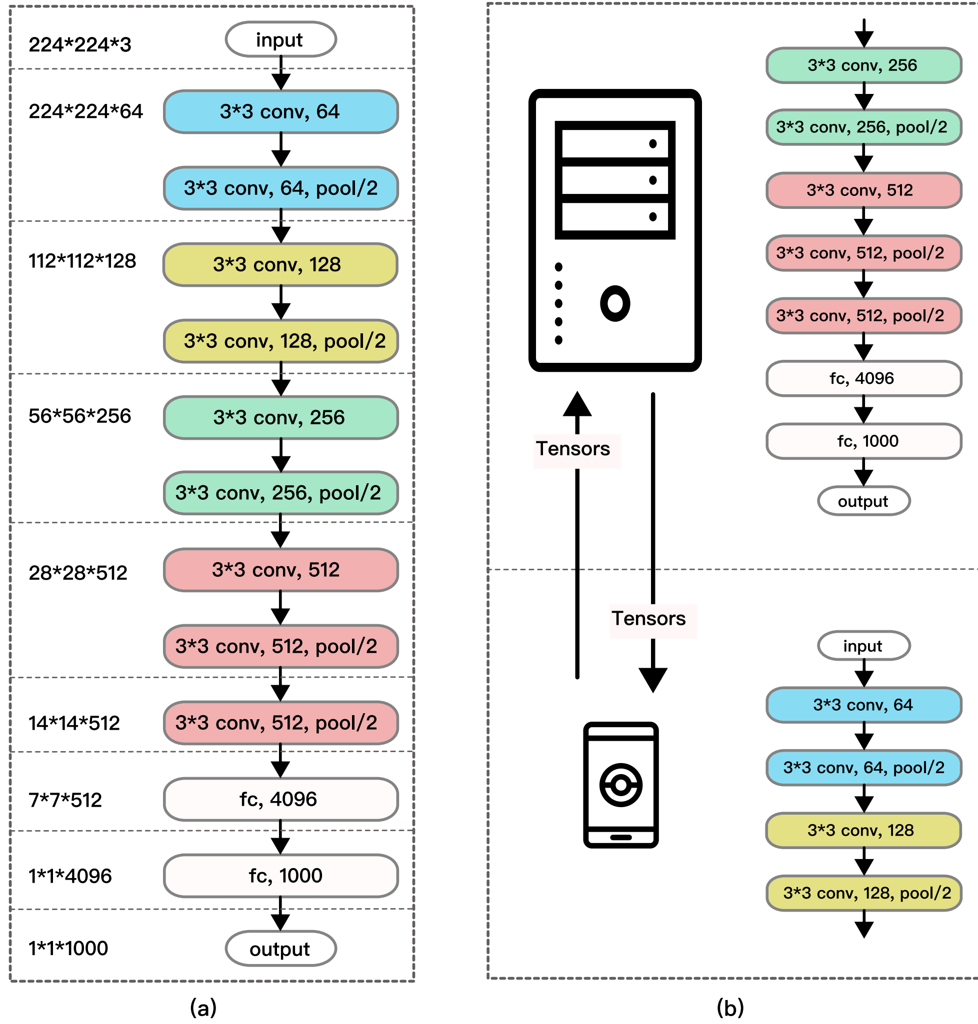


Fig. 5. An example of a deep neural network (a) and a DNN partition between device and edge server (b).

of computation under the same input scale C can be calculated as follows:

$$C = \sum_{k=1}^{\mathcal{K}} c_k \tag{1}$$

It is worth noting that the input tensor of each layer is predefined in the DNN, that is, the transmission of the network can also be portrayed in advance. Therefore, we can reduce resource consumption by partitioning the DNN between the edge server and the device. Meanwhile, since the computing speed of the edge server is faster than that of the device, we can accelerate the inference by offloading part of the task from the device to the edge server. As illustrated in *Fig. 5(b)*, the 11-layer DNN is partitioned into two parts. The first four layers are executed on the device, and the output tensors are transferred to the edge server, where the remaining seven layers are calculated. After the inference task is complete, the edge server sends results back to the device side.

Since the output tensor of the hidden layer is generally smaller than the raw data in most DNNs, the benefits of partition are not only reducing the computation overhead but also minimizing the bandwidth consumption, which means a trade-off between the computation and communication. As shown in *Fig. 6(a)*, an appropriate partition would benefit the overall inference time when the network bandwidth is sufficient. It can fully exploit the bandwidth to achieve equivalent performance to inference on GPU. In contrast, in the case of a low-bandwidth network environment, since only tensors of the hidden layer are sent to the edge server, the partition approach is even better than putting all the layers on the edge server, as shown in *Fig. 6(b)*. We leverage DNN partition in the proposed framework "*Coknight*" to attain resource efficiency, scalability, and latency reduction.

The design of the *Coknight* framework is shown in *Fig. 7(a)*. The main difference between *Coknight* and *IMPALA&Seed-RL* is that the inference task is not accomplished in a single site, rather, it is executed cooperatively between the actor and the learner through the partition. In such a way, we can achieve acceleration by striking a balance between reducing computation on low-speed devices and minimizing network bandwidth consumption.

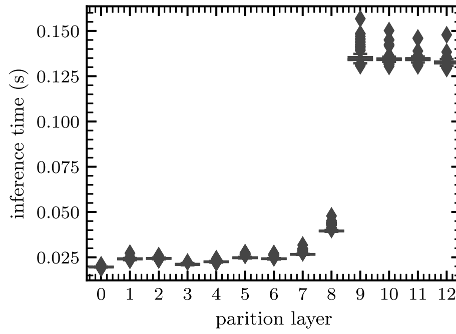
Meanwhile, the policy of the action taken in *Coknight* is also noteworthy. As shown in *Fig. 7(b)*, in an inference task, the shallow network may use the outdated model parameters while the deeper network leverages the latest parameters. Note that the action a selected by the original strategy $\pi(\boldsymbol{\theta}_t)$ based on observation s is:

$$a = \pi(s, [\theta_t^0, \theta_t^1, \theta_t^2, \theta_t^3, \dots, \theta_t^k]) \tag{2}$$

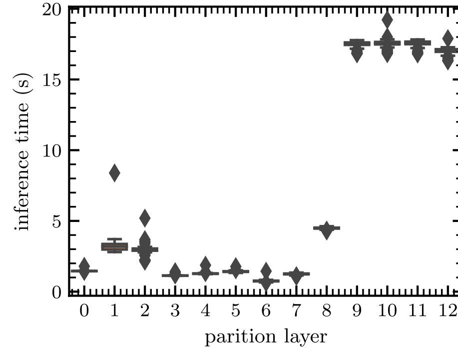
while the action selected by the new strategy in *Coknight* is:

$$a' = \pi(s, [\theta_t^0, \theta_t^1, \theta_t^2, \theta_{t+1}^3, \dots, \theta_{t+1}^k]). \tag{3}$$

We call this strategy "*hybrid near on-policy*". Since the shallow network is usually composed of some networks whose parameters are

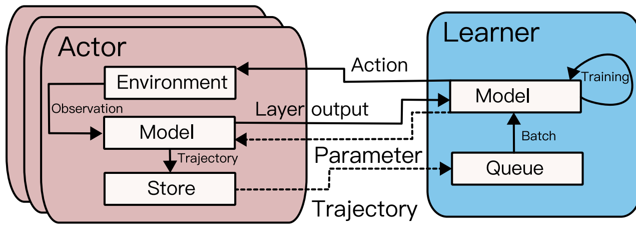


(a) Intel $i9$-$10900K$ + wired network without bandwidth limitation

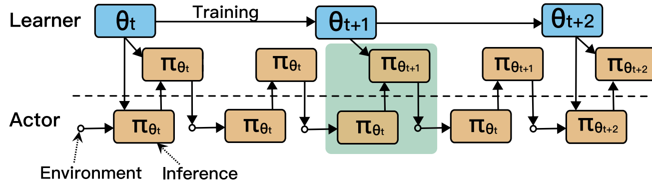


(b) Cortex-$A72$ + 4M wireless bandwidth

Fig. 6. Inference time distribution with different partition layers.



(a) *Coknight* architecture



(b) "hybrid near on-policy" of *Coknight*

Fig. 7. The architecture of *Coknight* (a) and the "hybrid near on-policy" of *Coknight* (b). Layer outputs and actions transmission between actor and learner are synchronous and real-time, while the parameter and trajectories are asynchronous and timing.

relatively stable during the training (e.g., CNN), it is reasonable to consider that the inference model in *Coknight* is similar to that in *Seed-RL*.

It is worth noting that in a general case when resources are sufficient resources as in Seed-RL, *Coknight* tends to perform the inference tasks on the learner. Only when resource bottlenecks occur will *Coknight* perform the partition on some devices. For those partitioned devices, we need to upload both intermediate tensor data and regular observation data to the edge server for model updates. Meanwhile, it is unnecessary to pay attention to the model and trajectory upload issues in those non-partitioned devices (all the layers are on the server). Therefore, the bandwidth consumption of *Coknight* will not increase exponentially compared to *Seed-RL*, since this upload operation is usually performed at a relatively low frequency.

With an appropriate partition of the DNN layers, *Coknight* can execute the front part of the DNN inference on devices and upload the rest to the edge server to complete the task. Hence, a crucial issue of *Coknight* is how to find the optimal partition of the network for each device actor, which is formally analyzed in the sequel.

3.2. Partition problem formulation

More generally, we assume a $\mathcal{K}$-layer DNN, and leverage a variable $y_i (0 \le y_i \le \mathcal{K})$ to indicate that the DNN is partitioned on y_ith layer on device d_i (i.e., actor i). Let $y_i = 0$ denote that the entire DNN model is executed on the edge server, while $y_i = \mathcal{K}$ represents that the whole DNN model is calculated on the device.

Let $C_d(y_i)$ and $C_e(y_i)$ denote FLOPs calculated on the device and the edge server (i.e., learner) with decision y_i, respectively, which can be formulated as the follows.

$$C_d(y_i) = \sum_{k=1}^{y_i} c_k; \qquad C_e(y_i) = \sum_{k=y_i+1}^{\mathcal{K}} c_k \tag{4}$$

We adopt *floating-point operations per second* (FLOPS) [29] to characterize the performance of devices and edge servers. We denote the FLOPS of the edge server as f_e. Since the devices are diverse, we use f_d^i to represent the FLOPS of the device d_i. Therefore, the calculation time of a partition task on device d_i and on the edge server are represented by $t_d^i(y_i)$ and $t_e^i(y_i)$, respectively, and can be calculated as follows:

$$t_d^i(y_i) = \frac{C_d(y_i)}{f_d^i}; \qquad t_e^i(y_i) = \frac{C_e(y_i)}{f_e} \tag{5}$$

In addition to the calculation time, we also need to pay attention to the data transmission time, which is proportional to the output tensor size of each layer, which is denoted as $b_k (0 \le k \le \mathcal{K})$, where b_0 represents the input tensor size, and $b_{\mathcal{K}}$ is the output tensor size. Suppose the transmission cost per unit tensor of each link is the same, denoted as tr, the transmission time is then the ratio of b_k and tr. In summary, the total response time $T_i(y_i)$ of a partition task on d_i can be formulated as follows:

$$T_i(y_i) = \begin{cases} t_e^i(y_i) + \frac{b_0 + b_{\mathcal{K}}}{tr} & y_i = 0 \\ t_d^i(y_i) + t_e^i(y_i) + \frac{b_{y_i} + b_{\mathcal{K}}}{tr} & 1 \le y_i \le \mathcal{K} - 1 \\ t_d^i(y_i) & y_i = \mathcal{K} \end{cases} \tag{6}$$

Obviously, when the resources (both compute and bandwidth) are not limited, there will be an optimal partition that can minimize the response time of the entire task.

$$y_i^* = \mathop{\arg\min}_{y_i \in \{0,1,\ldots,\mathcal{K}\}} (T_i(y_i)). \tag{7}$$

Here, y_i^* is the optimal partition decision that minimizes the response time.

It is reasonable to suppose that a device could perform a series of DNN inference tasks for the long term in Deep-RL. Such workloads consist of continuous tasks with the same input size and model. Therefore, from the perspective of computational load, we can treat the series of tasks as a whole task $\mathcal{W}$, consuming the same computation and communication resources continuously. For such an ongoing task $\mathcal{W}$, the resources allocated by the edge server to the device are monopolized, once these resources are earmarked for d_i, they would not be shared or conflicted with other devices.

Our goal is to minimize the latency of all the devices, that is, to reduce $T_i(y_i)$ of all the devices. This problem can be modeled as a constrained optimization problem (COP) as follows:

$$minimize \sum_{i=1}^{\mathcal{N}} T_i(y_i) \tag{8}$$

subject to:

$$0 \le y_i \le \mathcal{K} \tag{9}$$

$$\sum_{i}^{\mathcal{N}} C_e(y_i) \le \mathbb{R}_c \tag{10}$$

$$\sum_{i}^{\mathcal{N}} \delta b_{y_i} \le \mathbb{R}_b. \tag{11}$$

Here, δ is a proportional coefficient between the size of the tensor and the bandwidth it consumes. Based on the formulation of the multi-actor DNN partition problem, we will analyze its property and design the corresponding algorithm in the next section.

4. DNN partition algorithm design

Given the problem definition, in this section, we first conduct its complexity analysis in *Coknight*, thereby deriving an efficient algorithm based on the potential game theory [17].

4.1. Problem complexity

Firstly, we analyze the hardness of the problem to determine whether a polynomial-time optimal algorithm could be designed.

Theorem 1. *The multi-actor DNN partition (MADP) problem we formalized is NP-complete.*

Proof. First, this problem is apparently in NP as any solution to its decision form can be validated in polynomial time. Then, we conduct a reduction from MADP to the classical NP-complete knapsack problem (KP) [19,26] in polynomial time, to show the hardness of MADP.

Given an instance of KP whose definition is as follows:

$$maximize \sum_{i=1}^{n} x_i v_i \tag{12}$$

subject to:

$$\sum_{i=1}^{n} x_i w_i \le C \tag{13}$$

$$x_i = \{0, 1\}; 1 \le i \le n \tag{14}$$

For the reduction purposes, we transform the original minimization of MADP into an equivalent maximization goal. We intend to minimize the task inference time on each device by controlling the partition layer, that is, to maximize the benefits brought by the partition.

Through the observation, we can find that when the task is executed locally on the device, the constrained server computation and bandwidth are not consumed. Therefore, we design a utility function u_i for device d_i based on $y_i = \mathcal{K}$:

$$u_i(y_i) = T_i(y_i) - T_i(\mathcal{K}); 0 \le y_i \le \mathcal{K}, 1 \le i \le \mathcal{N} \tag{15}$$

We let a decision variable $x_{ij} = \{0, 1\}$ indicates whether the device d_i decides to perform the j-th partition. Since each device can only choose one partition layer, we need to make the following constraints on x_{ij}:

$$\sum_{j=0}^{\mathcal{K}} x_{ij} \le 1 \tag{16}$$

Combining with the formulas above, the equivalent maximization problem can be formalized as follows:

$$maximize \sum_{i=1}^{\mathcal{N}} \sum_{j=0}^{\mathcal{K}} x_{ij} u_i(j) \tag{17}$$

subject to:

$$\sum_{j=0}^{\mathcal{K}} x_{ij} \le 1 \tag{18}$$

$$\sum_{i=1}^{\mathcal{N}} \sum_{j=0}^{\mathcal{K}} x_{ij} C_e(j) \le \mathbb{R}_c \tag{19}$$

$$\sum_{i=1}^{\mathcal{N}} \sum_{j=0}^{\mathcal{K}} x_{ij} \delta b_j \le \mathbb{R}_b \tag{20}$$

$$x_{ij} = \{0, 1\}; 0 \le j \le \mathcal{K}, 1 \le i \le \mathcal{N} \tag{21}$$

According to this formulation, we introduce how to construct an instance of MADP to handle any KP problem. Let $\mathcal{N} := n$, $\mathbb{R}_c := C$ and $\mathbb{R}_b := +\infty$, we can align the constraints with KP. Then we let $x_{i0} := v_i$, and add a dummy item $x_{i1} := 0$. With this trivial transformation, it is obvious that any inputs of KP can be adopted to MADP. That is, the MADP solution can be used to solve KP. Correspondingly, any KP solution can also solve a particular form of MADP. To sum up, the multi-actor partition problem can be confirmed NP-complete.

4.2. Dynamic algorithm

Although many algorithms (e.g., branch and bound method, greedy, etc.) are designed for such an NP-complete problem, considering the devices would gradually enter or leave the edge network, we intend to develop an algorithm handling the dynamic increase and decrease of devices. For this purpose, we adopt game theory in the algorithm design.

Firstly, we formalize this problem as a game:

$$G = < \vec{\mathcal{N}}, (S_i)_{i \in \vec{\mathcal{N}}}, (u_i)_{i \in \vec{\mathcal{N}}} > \tag{22}$$

here, $\vec{\mathcal{N}}$ represents a set of $\mathcal{N}$ players; $\mathcal{S}_i$ denotes the strategy set of each player i, which is the same as $\{yi \mid 0 \le y_i \le \mathcal{K}\}$ in the partition problem; u_i is the utility function of each player, consistent with its definition in *Eq. (15)*, combined with resource constraints additionally. We suppose every player is selfish and intends to maximize its utility function u_i. And we will prove that Pure Nash Equilibrium (PNE) exists in G so that a concise and efficient local search algorithm could be adopted.

Theorem 2. *The game G is an exact potential game.*

Proof. Assuming there is a global utility function Φ, which indicates the time reduction of the entire system compared with the latency that all devices inference locally. Let s_i denote the strategy chosen by player i, and s_{-i} represent the strategy selected by other players except i. Given a set of player strategies $\vec{S} = \{s_1, s_2, ..., s_{\mathcal{N}}\}$, the definition of Φ is as follows:

$$\Phi(\vec{S}) = \sum_{i \in \vec{S}} u_i(s_i). \tag{23}$$

When player i changes its own strategy from s_i to s'_i, and other players' strategies remain s_{-i}, there is the following equation:

$$u_i(s_i, s_{-i}) - u_i(s'_i, s_{-i}) = \Phi(s_i, s_{-i}) - \Phi(s'_i, s_{-i}); \quad \forall s_i, s'_i \in S_i, s_{-i} \in S_{-i}. \tag{24}$$

Thus, G is a potential game with the potential function Φ, and hence always admits PNE in finite improvements.

Since there exists PNE in the G, we can attain the PNE through finite steps of improvement. Hence, the heuristic algorithm can theoretically achieve the optimal after a limited number of iterations. However, what we need is a dynamic polynomial-time algorithm so that devices can participate in the system fleetly. Apparently, heuristic algorithms cannot meet our requirements.

Consequently, an intuitive, dynamic and efficient algorithm is that player i calculates its best response according to the strategy already selected by other players, that is:

$$s_i^* = \arg\max_{s_i \in S_i} u(s_i, s_{-i}). \tag{25}$$

It is a straightforward greedy algorithm, which we named "incremental greedy algorithm." The algorithm has high computational efficiency with an unsatisfactory approximation ratio. Besides, according to the principle of "hybrid near on-policy," we prefer to keep the same partition strategy for different actors to leverage the same hybrid model for inference, thus accelerating the convergence of training.

To this end, we design a resource fairness algorithm named "dynamic efficient fairness partition algorithm," which intends to make all actors carry out resource competition equally. The basic idea is that the newly added actor forms a pair with every actor who has already participated in *Coknight*, and performs the Pareto Improvement in duos. The whole algorithm is shown in *Algorithm 1*.

On the one hand, *DEFP* is an incremental algorithm apparently. Moreover, this algorithm could be decremental because it can handle the release of resources when the device goes offline, simply by selecting a player to re-execute the algorithm. On the other hand, *DEFP* is a fair algorithm for all devices since each player has made Pareto Improvement with other players pairwise to compete for resources fairly. Thus, the algorithm can work in a distributed manner, with each actor communicating directly with another actor to compete for resources. Furthermore, *DEFP* is a polynomial-time algorithm, and we will give an approximate time complexity analysis of *Algorithm 1* as follows:

Theorem 3. *Given the $\mathcal{N}$-players and $\mathcal{K}$-strategies in game G, the time complexity of* DEFP *is upper bounded by $O(\mathcal{K}^2\mathcal{N}^2)$.*

Proof. Firstly, the complexity of the best response in *Eq. (25)* is roughly $O(\mathcal{K})$ (line 6). And the number of strategies in $S_{\mathcal{N}}$ is $\mathcal{K}$, hence the complexity of lines $5-12$ is $O(\mathcal{K}^2)$.

Algorithm 1: Dynamic Efficient Fairness Partition (*DEFP*) algorithm.

```
Input: S(s_1, s_2, ..., s_{N-1}): A set of strategies that players {1, 2, ..., N-1}
       have adopted;
       S_N : A set of available strategies of player N;
Output: S(s_1, s_2, ..., s_N): A set of strategies that players {1, 2, ..., N} have
        adopted
1  for (player i, s_i) in S(s_1, s_2, ..., s_{N-1}) do
       /* Pair Pareto Improvement                                   */
2      utility_of_pair_i_n ← 0;
3      available_resource ← the sum of resources occupied by player i and
       player N;
4      s_i^temp, s_N^temp ← the strategy of on-device computing independently;
5      for s_N^j in S_N do
           /* Best Response                                         */
6          s_i^* ← calculate the best response for player i based on Eq. (25);
           /* Pareto Improvement                                    */
7          if u_i(s_i^*) + u_N(s_j) ≥ utility_of_pair_i_n then
8              utility_of_pair_i_n ← u_i(s_i^*) + u_N(s_j);
9              s_i^temp ← s_i^*;
10             s_N^temp ← s_N^j;
11         end
12     end
       /* update players strategy                                   */
13     s_i ← s_i^temp;
14     s_N ← s_N^temp;
15 end
```

Since it is necessary to traverse all the players to form pairs (line 1), the complexity of running the *DEFP* algorithm once is $O(\mathcal{K}^2\mathcal{N})$. For $\mathcal{N}$ players, it is necessary to repeat the *DEFP* algorithm $\mathcal{N}$ times to construct the final strategy set. To sum up, the total complexity of solving $\mathcal{N}$-players and $\mathcal{K}$-strategies problem is $O(\mathcal{K}^2\mathcal{N}^2)$.

5. Performance studies

To validate our findings, we implemented *Coknight* based on *PyTorch* and gRPC to evaluate its performance. Firstly, we constructed a small-scale physical testbed to validate the model convergence. Afterward, we conducted a large-scale simulation based on physical test data to validate the scalability and efficiency. The simulator efficiently implemented the proposed algorithms and the model upon which the algorithms are built.

5.1. Experimental setup

By following the settings of *IMPALA* [9], we implemented the V-trace algorithm and conducted experiments on the Atari suite [4]. Besides, we also took the hyper-parameters and optimization procedures from [9].

Datasets: We took the Atari games as the test environments. The size of each frame in games is $3 \times 84 \times 84$. And we aggregated the reward of the game as part of the input, so the input tensor shape should be $4 \times 84 \times 84$. Our goal is to maximize the game's final score, which is reflected in the cumulative sum of all rewards in one round of the game.

Baselines: As for physical experiments, we treated *IMPALA* [9] and *Seed-RL* [10] as baselines. Both methods are variations implemented based on *Coknight* ($y_i = 0$ or $y_i = \mathcal{K}$). Hence the performance is marginal different from the original version, but the overall architecture is the same. Each set of experiments would train 6,000,000 steps to obtain stable scores.

And for the simulation comparison of *DEFP* algorithm, we evaluated it against mixed integer linear programming (*MILP*, i.e., optimal solution, obtained by the solver), incremental greedy (*IGreedy*), and offline greedy (*OGreedy*) as baselines. We make a scalability analysis of the algorithm concerning the number of actors.

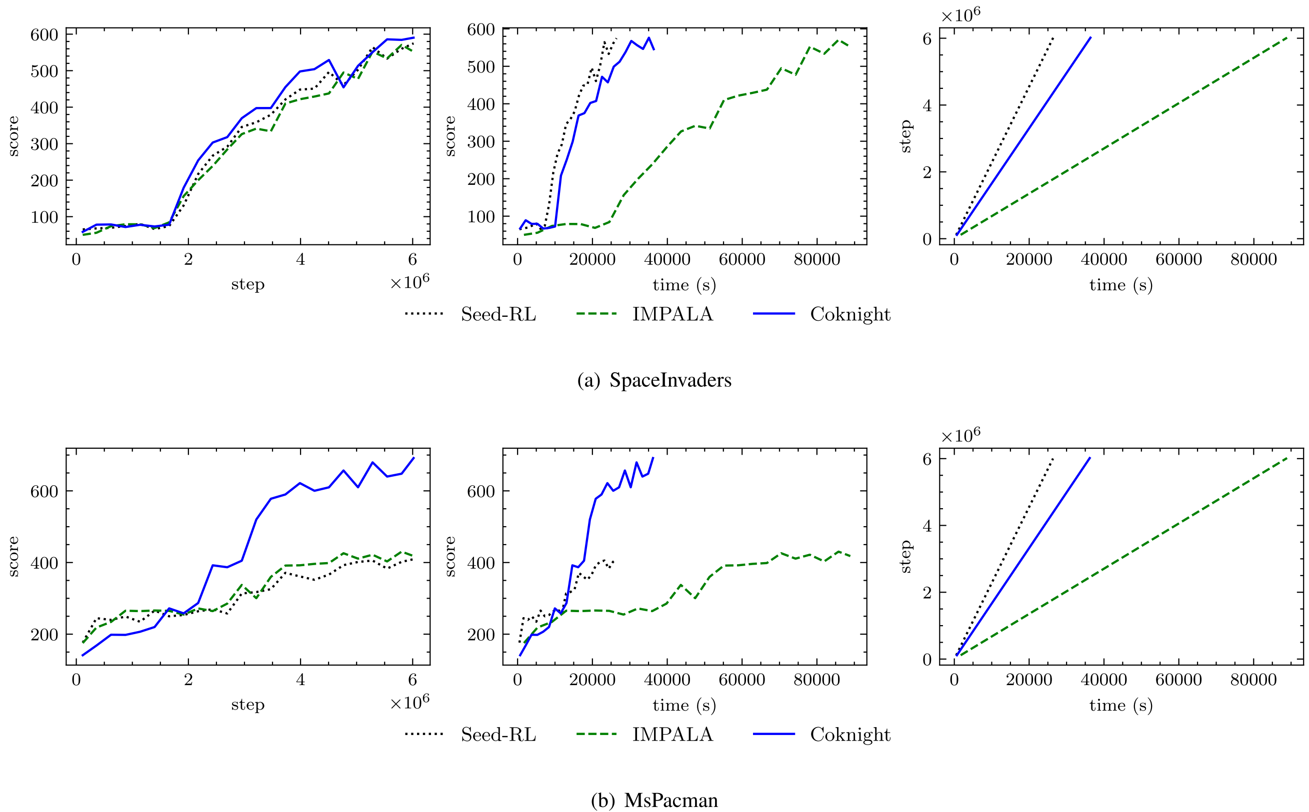


Fig. 8. Performance comparison of the three frameworks. The figure from left to right is the convergence rate, convergence time, and running time, respectively.

Evaluation Platform: The experiments were conducted with *Python* 3.7.9, *Pytorch* 1.7.1, gRPC 1.32.0 and *gym* 0.18.0. As for the hardware of the platform, we performed experiments in three types of heterogeneous edge devices similar to [15]. We took 1× Tesla-V100 32G as an edge server (learner), as well as 4×40 cores Intel(R) Xeon(R) E5-2630 v4 @2.20 GHz servers and Raspberry Pi 4B as the mobile devices (actors). In addition, we leveraged switches to build a wired edge network. We deployed *Coknight*'s actors on CPU (Intel/ARM) servers, while the leaner was deployed on the GPU server, with peer-to-peer communication to Actors via gRPC. In the initial phase, each actor will run the partition algorithm to establish communication with leaner. In addition, to validate the performance of *Coknight* in the edge network, we constrained the bandwidth of the subnetwork through switches.

5.2. Numerical results

We investigated an empirical experiment to validate the convergence and resource efficiency of *Coknight*, *IMPALA* and *Seed-RL* at first. After that, we compared the scalability of *Coknight* and *Seed-RL* by varying the number of actors. All methods adopted the same hyper-parameters, including the number of actors, batch size, buffer size, learning rate, etc. To make *Coknight* execute the partition strategy, we limit the network bandwidth of *Coknight* to 10 MB/s, which is a practical setting as most real-life WiFi. Furthermore, we provided a large-scale simulation to validate the scalability of the proposed DNN partition algorithm.

5.2.1. Convergence efficiency

Firstly, we studied the impact on convergence rate and total running time between *Coknight* and other frameworks in two different game experiments: *SpaceInvaders* and *MsPacman*. *Fig. 8* illustrates the performance comparison of the three frameworks with the same number of actors ($n = 12$). Regarding the convergence rate (left figure), we can find that due to the impact of the off-policy, *IMPALA* obtains the lowest score in the same step (i.e., convergence rate). It is worth noting that the score of *Coknight* is slightly higher than that of *Seed-RL*. We speculate that the shallow network of the hybrid near on-policy remains relatively stable, which leads to a particular regularization effect on the model and improves its reward. Meanwhile, as shown in *Fig. 8* on the right, driving the same number of actors in an experimental environment with almost unlimited network bandwidth, *Coknight*'s running time is slightly lower than *Seed-RL* while still significantly improves compared to *IMPALA*. Combining these two results, we can find from the central figures that *Coknight* can obtain scores equivalent to *Seed-RL* when running after the same time. What's more, in MsPacman Game, we found that the reward of *Coknight* surpasses that of *Seed-RL*. These results actually indicate that the hybrid model in *Coknight* is relatively more robust.

5.2.2. Resource efficiency

Further, we focused on the resource usage of the learner during the entire system running time, including GPU usage, memory usage, and bandwidth usage. We monitored the usage of resources during the entire training process and made statistics on the resource utilization in each period. The numerical results are illustrated in *Fig. 9*, and the X-axis is the utilization ratio of resources, such as GPU utilization and bandwidth. While the Y-axis is the density distribution of statistics, the larger it is, the longer the resource utilization remains at the corresponding x value during training. As shown in the diagrams of *Fig. 9*, the GPU utilization of *Coknight* is significantly reduced to approximately 60%, while *Seed-RL* requires more than 90%. In another word, *Coknight* saves around 30% of GPU utilization than *Seed-RL*, with almost the same performance guarantee. What needs to be explained is that the reduced part of the computation is not evaporated but allocated to actors' CPUs. Another concern is that the bandwidth consumption

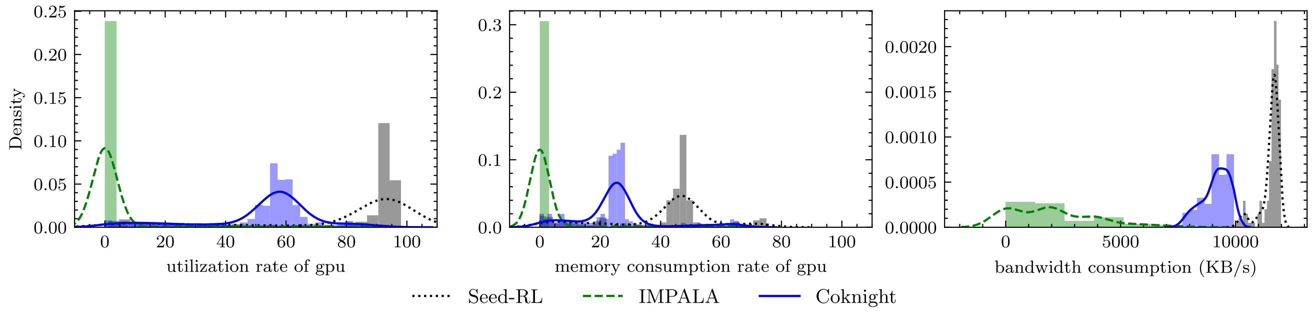


Fig. 9. The resource usage of the learner. From left to right are GPU usage, memory usage, and bandwidth consumption.

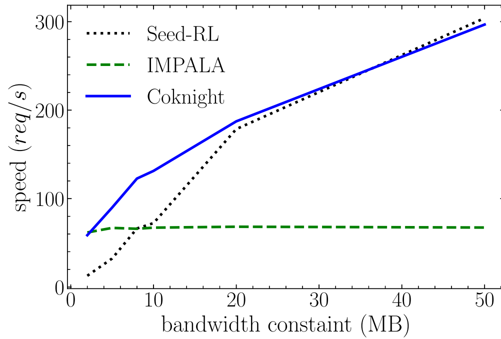


Fig. 10. How the performance of baselines varies with the bandwidth constraint.

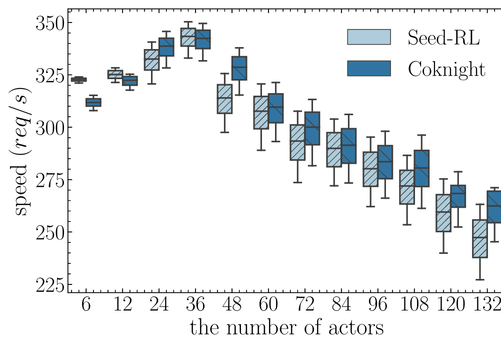


Fig. 11. How the performance of *Coknight* and *Seed-RL* is relatively changed concerning the number of actors.

of *Seed-RL* is proximate 12 MB/s. If the bandwidth of *Seed-RL* is limited to 10 MB/s following *Coknight*, it is reasonable to believe that the performance will decrease moderately. To further study the impact of bandwidth on performance, we changed the number of actors to 6 and then varied the bandwidth constraint from 50 MB to 2 MB through switches. The baselines performance varying with different bandwidths are shown in Fig. 10. It is worth noting that *Seed-RL*'s performance declined significantly as bandwidth decreased, while *Coknight*'s declined more moderately and eventually came into line with *IMPALA*'s. This indicates that *Coknight* performs better on low bandwidth networks.

5.2.3. Scalability

With the observation above, we further investigated the scalability of *Coknight* concerning the number of actors. Since the overall running time of *Seed-RL* in *Fig. 8* is less than that of *Coknight*, we compared the speed of *Coknight* with *Seed-RL* in the same wired network. The speed we defined means the number of inference requests served per second (req/s). The numeral results are shown in *Fig. 11*. We can find that *Seed-RL* is slightly faster than *Coknight* at a small scale (with less than 36 actors). The reason is that *Coknight* needs additional bandwidth for model synchronization between the edge server and devices. With the increase in the number of actors, both the performance of *Seed-RL* and *Coknight* are declined. The difference is that *Coknight* has relatively shown better performance than *Seed-RL*. This observation is highly consistent with our expectation that the partition strategy started to work, balancing the computing and bandwidth requirements of the increased actors. In short, *Coknight* shows satisfying performance and high scalability toward large-scale actors.

Afterward, based on the resource usage and latency data collected in the physical system, we conducted a large-scale simulation experiment to validate the scalability of the algorithm. To measure these algorithms' relative performance, we evaluated them by three indicators: 1) total latency reduction; 2) strategy variance; 3) latency reduction variance. Total latency reduction refers to the decreased time relative to running locally when all actors take one step, and it is our most concerned optimization goal. Strategies variance and reduction latency variance are used to measure the difference between the strategies executed by different actors. Remember that we aim to make all actors adopt the same partition as much as possible to avoid inconsistent hybrid models.

To simulate a real-world scenario, we compared two types of resource environments: 1) elastic server: resources can be elastically increased, such as AWS-based edge servers; 2) non-elastic server: the physical machine environment whose resources are relatively fixed. *Fig. 12* illustrates the numerical results of the simulation, and in general, the performance of the proposed algorithm (*DEFP*) is quite similar to the optimal solution (*MILP*) on each indicator. In the case of the elastic server, the total latency reductions of *DEFP* and *MILP* are both increasing as the number of actors increasing, and can observe a clear near-linear trend. For greedy algorithms, the deduction is much lower than *DEFP* and *MILP*. Noticeably, because of the unreasonable allocation of resources by *OGreedy* and *IGreedy*, both tend to choose to execute locally, so the variance is relatively slighter than others. It's worth noting that despite the approximate performance, *DEFP* has a minor variance than *MILP*, which is one of the features of *DEFP*. The same phenomenon is established in the non-elastic case. What's more, it is valuable that *DEFP* gradually achieves the same results as optimal solutions with constant resources. It validates our view that this issue is a potential game so that we can achieve the optimal solution through finite steps improvement.

6. Related work

In this section, we overview some related work from two aspects — distributed deep reinforcement learning and DNN inference acceleration — to show how the Coknight framework is distinct from the existing works.

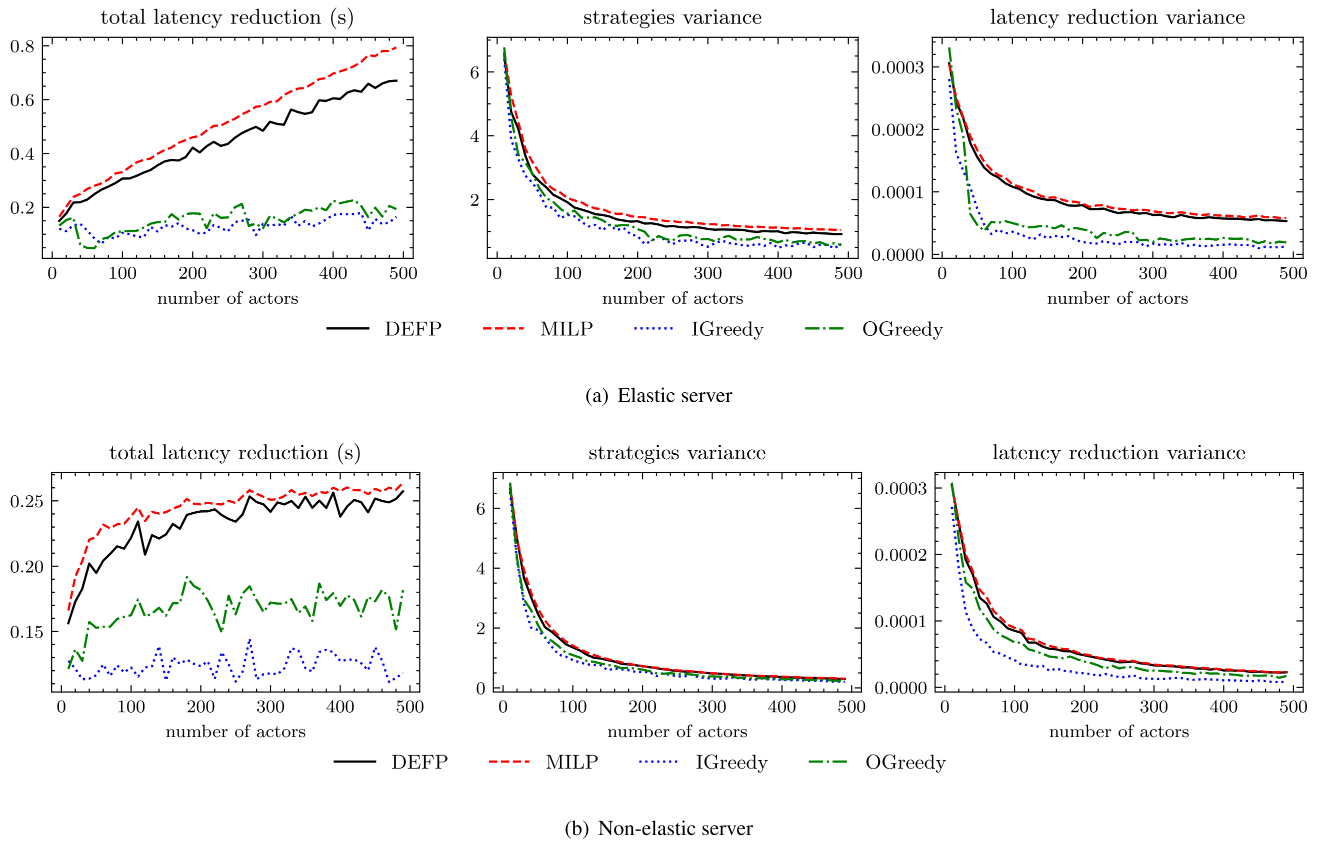


Fig. 12. In the case of elastic and non-elastic, the performance comparison between algorithms.

6.1. Distributed deep reinforcement learning

Some classic actor and learner-based Deep-RL frameworks such as A3C [27] are distributively deployed by decomposing the model into actor and learner to scale out the parallelism of the Deep-RL. On the downside, the off-policy introduced by their distributed architecture leads to a severe problem that massive epochs are required to converge. To address the off-policy issue, Batch A2C [1] implements the synchronization mechanism by alternately performing the inference and training. Although this on-policy method improves the convergence rate, the synchronization method does not make full use of the computation resources of both actors and learners, resulting in a significant delay in the entire system. Therefore, *IMPALA* [9] proposes the v-trace algorithm, which makes Deep-RL converge apace with an asynchronous actor-learners architecture. This algorithm is also adopted in the proposed *Coknight* framework. This algorithm dramatically improves the availability of the asynchronous distributed Deep-RL. However, the inference in *IMPALA* is mainly performed on the actors' CPU, which introduces a huge latency. To fully exploit the superiority of the high-speed float operation on GPU, *Seed-RL* [10] puts both the inference and training procedures on the GPUs for the speedup purpose.

Although *Seed-RL* seems promising, when we adopted it to the edge computing, it apparently lacks attention to the limited bandwidth and computation of the edge server. Therefore, we attempted to find a way to make full use of the customized limited computation and bandwidth resources for the edge system, as well as to achieve an on-policy strategy for fast convergence guarantee. Since a large amount of resource consumption in Deep-RL comes from the inference phase, we considered adopting the current mainstream methods on inference acceleration to implement the proposed framework.

6.2. DNN inference acceleration

There are a significant number of methods on DNN inference acceleration, such as model compression [16], model pruning [11,13], model quantization [14,22], etc. The acceleration techniques in these algorithms are typically achieved by modifying the model parameters, which violate the on-policy principle — the models of training and inference are expected to be consistent. Therefore, we leveraged the DNN partition to allocate resources and accelerate the inference in *Coknight*, which does not need to modify any parameters of the DNN model. Neurosurgeon [24] is one of the pioneering works with this technology, which is creatively proposed to reduce the latency and energy consumption of the inference through the partition. However, Neurosurgeon can only deal with chain-topology DNNs and do nothing for others with complex structures. In contrast, ECDI [20] is presented to formalize the neural network structure into a DAG (direct acyclic graph) whereby a minimum cut-based method is proposed to find the optimal partition decision. Although there is a solid exploration in graph theory, ECDI only handles the naive two-part partition in the single edge and cloud manner and lacks consideration of multiple edges. Thus, another DINA [28] method applies partition to fog computing, distributing a task to multiple fog nodes and proposing a more fine-grained division of the tensors. In addition to these works, to take advantage of each other, partition and pruning are combined to attain inference acceleration in [33], which further explores the availability of the partition method.

Unfortunately, these previous works [20,28,33] do not meet the requirements of the proposed architecture. There are two main reasons: the first is that these methods are optimized for a single task and are not applicable for the multi-tenant scenario. The other reason is that these methods require numerous offline iterations, while the scenarios we support are dynamic and distributed.

Another important innovation is that all these methods apply DNN partition to the traditional deep learning inference stage, and we embed it in the Deep-RL framework. To the best of our knowledge, we are the first to conduct the multiple actor-learner framework based on the DNN partition.

7. Conclusions

In this paper we presented *Coknight*, a scalable and efficient framework for deep reinforcement learning by exploiting the DNN partition techniques in the edge environment. We first formulated the partition problem into a variant of knapsack problem in device-edge setting, and showed it is NP-complete, and then transformed it into a multi-player game. Next, we proved the game is a potential game whereby an efficient distributed algorithm is developed for fast and dynamic partitioning. Finally, we prototyped the *Coknight* framework and conducted extensive experiments to show its performance superiority in terms of resource efficiency and scalability over the selected existing works.

Coknight improves the efficiency of Deep-RL by making a trade-off between CPU utilization, GPU utilization, and network bandwidth cooperatively. This partition-based approach flexibly improves task efficiency and task parallelism concerning limited network bandwidth. However, although *Coknight* reduces the real-time bandwidth requirement, it transmits trajectories to the learner offline, resulting in massive bandwidth consumption. Simultaneously, the current single learner architecture of the *Coknight* still suffers from the constrained GPU resources. To improve scalability further, the designs with multi-leaner architecture as those in federated learning and gossip training, are worth further studying concerning parameter aggregation. In addition, how to train on partitioned tensors instead of complete trajectories is also a focus of future works.

CRediT authorship contribution statement

Hao Dai and Jiashu Wu conceived of the presented idea. Yang Wang developed the theory and performed the computations. Chengzhong Xu verified the analytical methods and supervised the findings of this work. Hao Dai developed the prototype, and Jiashu Wu carried out the experiments. Hao Dai wrote the manuscript with support from Yang Wang and Chengzhong Xu. All authors discussed the results and contributed to the final manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by Key-Area Research and Development Program of Guangdong Province (2020B010164002), Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0015/2019/AKP, and National Natural Science Foundation of China (61672513).

References

[1] C. Alfredo, C. Humberto, C. Arjun, Efficient parallel methods for deep reinforcement learning, in: RLDM, 2017, pp. 1–6.
[2] M. Assran, J. Romoff, N. Ballas, J. Pineau, M. Rabbat, Gossip-based actor-learner architectures for deep reinforcement learning, in: NeurIPS, 2019.
[3] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, J. Kautz, GA3C: GPU-based A3C for Deep Reinforcement Learning, Technical Report, 2019.
[4] M.G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: an evaluation platform for general agents, J. Artif. Intell. Res. (2013).
[5] A. Boukerche, S. Guan, R.E.D. Grande, Sustainable offloading in mobile cloud computing, ACM Comput. Surv. 52 (2019) 1–37.
[6] J. Chen, X. Ran, Deep learning with edge computing: a review, Proc. IEEE 107 (2019).
[7] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, M. Pavone, S. Katti, Cellular network traffic scheduling with deep reinforcement learning, in: AAAI, 2018, pp. 766–774.
[8] S. Dalton, I. Frosio, M. Garland, Accelerating Reinforcement Learning Through GPU Atari Emulation, 2019.
[9] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, B. Yotam, F. Vlad, H. Tim, I. Dunning, S. Legg, K. Kavukcuoglu, IMPALA: scalable distributed deep-RL with importance weighted actor-learner architectures, in: ICML, 2018.
[10] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, M. Michalski, SEED RL: scalable and efficient deep-rl with accelerated central inference, in: ICLR, 2020.
[11] D. Gao, X. He, Z. Zhou, Y. Tong, K. Xu, L. Thiele, Rethinking pruning for accelerating deep inference at the edge, in: KDD, 2020, pp. 155–164.
[12] X. Guo, S. Singh, H. Lee, R. Lewis, X. Wang, Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning, in: NeurIPS, 2014.
[13] Y. Guo, A. Yao, Y. Chen, Dynamic network surgery for efficient DNNs, in: NeurIPS, 2016.
[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in: ICML, PMLR, 2015, pp. 1737–1746.
[15] R. Han, D. Li, J. Ouyang, C.H. Liu, G. Wang, D. Wu, L.Y. Chen, Accurate differentially private deep learning on the edge, in: IEEE TPDS, 2021.
[16] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding, in: ICLR, 2016.
[17] Q. He, G. Cui, X. Zhang, F. Chen, S. Deng, H. Jin, Y. Li, Y. Yang, A game-theoretical approach for user allocation in edge computing environment, IEEE Trans. Parallel Distrib. Syst. 31 (2020) 515–529.
[18] T.M. Ho, K.K. Nguyen, Joint server selection, cooperative offloading and handover in multi-access edge computing wireless network: a deep reinforcement learning approach, IEEE Trans. Mob. Comput. 1233 (2020).
[19] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, J. ACM (1974).
[20] C. Hu, W. Bao, D. Wang, F. Liu, Dynamic adaptive DNN surgery for inference acceleration on the edge, in: Proceedings - IEEE INFOCOM 2019-April, 2019, pp. 1423–1431.
[21] G. Huang, C. Luo, K. Wu, Y. Ma, Y. Zhang, X. Liu, Software-defined infrastructure for decentralized data lifecycle governance: principled design and open challenges, in: ICDCS, IEEE, 2019, pp. 1674–1683.
[22] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, in: NeurIPS, 2016.
[23] Y. Jiang, Y. Hu, M. Bennis, F.C. Zheng, X. You, A mean field game-based distributed edge caching in fog radio access networks, IEEE Trans. Commun. 68 (2020) 1567–1580.
[24] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, L. Tang, Neurosurgeon: collaborative intelligence between the cloud and mobile edge, ACM SIGPLAN Not. 52 (2017) 615–629.
[25] H. Mao, W.J. Dally, Deep compression: compressing deep neural, in: ICLR, 2016.
[26] S. Martello, P. Toth, Algorithms for Knapsack Problems, North-Holland Mathematics Studies, 1987.
[27] V. Mnih, A.P. Badia, L. Mirza, A. Graves, T. Harley, T.P. Lillicrap, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: ICML, 2016.
[28] T. Mohammed, C. Joe-Wong, R. Babbar, M.D. Francesco, Distributed inference acceleration with adaptive DNN partitioning and offloading, in: Proceedings - IEEE INFOCOM 2020-July, 2020, pp. 854–863.
[29] P. Molchanov, S. Tyree, T. Karras, T. Aila, J. Kautz, Pruning convolutional neural networks for resource efficient inference, in: ICLR, 2017.
[30] K. Poularakis, J. Llorca, A.M. Tulino, I. Taylor, L. Tassiulas, Joint service placement and request routing in multi-cell mobile edge computing networks, in: Proceedings - IEEE INFOCOM 2019-April, 2019, pp. 10–18.
[31] Y. Qian, R. Wang, J. Wu, B. Tan, H. Ren, Reinforcement learning-based optimal computing and caching in mobile edge network, IEEE J. Sel. Areas Commun. 38 (2020) 2343–2355.
[32] G. Qu, Y. Lin, A. Wierman, N. Li, Scalable Multi-Agent Reinforcement Learning for Networked Systems with Average Reward, 2020.
[33] W. Shi, Y. Hou, S. Zhou, Z. Niu, Y. Zhang, L. Geng, Improving device-edge cooperative inference of deep learning via 2-step pruning, in: INFOCOM WKSHPS, 2019.
[34] F. Tang, Y. Zhou, N. Kato, Deep reinforcement learning for dynamic uplink/downlink resource allocation in high mobility 5G HetNet, IEEE J. Sel. Areas Commun. 8716 (2020) 1–10.
[35] S. Teerapittayanon, B. McDanel, H.T. Kung, Branchynet: fast inference via early exiting from deep neural networks, in: ICPR, IEEE, 2016, pp. 2464–2469.
[36] S. Teerapittayanon, B. McDanel, H. Kung, Distributed deep neural networks over the cloud, the edge and end devices, in: ICDCS, 2017, pp. 328–339.
[37] X. Wang, Y. Han, V.C. Leung, D. Niyato, X. Yan, X. Chen, Convergence of Edge Computing and Deep Learning: A Comprehensive Survey, 2020.
[38] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, D. Batra, Decentralized distributed ppo: solving pointgoal navigation, in: ICLR, 2020.

[39] R. Xie, X. Jia, K. Wu, Adaptive online decision method for initial congestion window in 5G mobile edge computing using deep reinforcement learning, IEEE J. Sel. Areas Commun. 38 (2020) 389–403.
[40] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, X. Liu, DeepWear: adaptive local offloading for on-wearable deep learning, IEEE Trans. Mob. Comput. 19 (2020) 314–330.
[41] S. Yu, B. Dab, Z. Movahedi, R. Langar, L. Wang, A socially-aware hybrid computation offloading framework for multi-access edge computing, IEEE Trans. Mob. Comput. 19 (2020) 1247–1259.
[42] Y. Zhan, J. Zhang, An incentive mechanism design for efficient edge learning by deep reinforcement learning approach, in: Proceedings - IEEE INFOCOM 2020-July, 2020, pp. 2489–2498.

Hao Dai received the M.Sc. degree in Communication and Electronic Technology from Wuhan University of Technology, in 2017. He is currently working toward the PhD degree in the Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interests include deep learning, reinforcement learning, mobile edge computing systems.

Jiashu Wu received BSc. degree in Computer Science and Financial Mathematics & Statistics from the University of Sydney, Australia (2018), and M.IT degree in Artificial Intelligence from the University of Melbourne, Australia (2020). He is currently pursuing his Ph.D at the University of Chinese Academy of Sciences (Shenzhen Institute of Advanced Technology, CAS). His research interests include big data and cloud computing.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China (1989), and the MSc. and PhD. degrees in computer science from Carlton University (2001) and University of Alberta, Canada (2008), respectively. He is currently with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, as a full professor and with Xiamen University, China as an adjunct professor. His research interests include service and cloud computing, programming language implementation, and software engineering. He is an Alberta Industry R&D Associate (2009-2011), and a Canadian Fulbright Scholar (2014-2015).

Chengzhong Xu obtained B.Sc. and M.Sc. degrees from Nanjing University in 1986, and 1989, respectively, and a Ph.D. degree from the University of Hong Kong in 1993, all in Computer Science and Engineering. Currently, he is a Chair Professor of Computer Science and the Dean of Faculty of Science and Technology, University of Macau, China. His recent research interests are in cloud and distributed computing, systems support for AI, smart city and autonomous driving. He has published more than 400 papers in journals and conferences. He serves on a number of journal editorial boards and the Chair of IEEE TCDP from 2015 to 2020. He is a fellow of the IEEE.