Octopus: SLO-Aware Progressive Inference Serving via Deep Reinforcement Learning in Multi-tenant Edge Cluster

Ziyang Zhang[1](✉), Yang Zhao[2], and Jie Liu[1,2]

[1] School of Science and Technology, Harbin Institute of Technology, Harbin 150001, China
zhangzy@stu.hit.edu.cn

[2] International Research Institute for Artificial Intelligence, Harbin Institute of Technology, Shenzhen 518055, China
{yang.zhao,jieliu}@hit.edu.cn

Abstract. Deep neural network (DNN) inference service at the edge is promising, but it is still non-trivial to achieve high-throughput for multi-DNN model deployment on resource-constrained edge devices. Furthermore, an edge inference service system must respond to requests with bounded latency to maintain a consistent service-level objective (SLO). To address these challenges, we propose Octopus, a flexible and adaptive SLO-aware progressive inference scheduling framework to support both computer vision (CV) and natural language processing (NLP) DNN models on a multi-tenant heterogeneous edge cluster. Our deep reinforcement learning-based scheduler can automatically determine the optimal joint configuration of 1) DNN batch size, 2) DNN model exit point, and 3) edge node dispatching for each inference request to maximize the overall throughput of edge clusters. We evaluate Octopus using representative CV and NLP DNN models on an edge cluster with various heterogeneous devices. Our extensive experiments reveal that Octopus is adaptive to various requests and dynamic networks, achieving up to a 3.3× improvement in overall throughput compared to state-of-the-art schemes while satisfying soft SLO and maintaining high inference accuracy.

Keywords: Edge computing · Progressive inference · Deep reinforcement learning · Multi-tenant

1 Introduction

Recent advancements in deep learning and Internet of Things (IoT) have facilitated the development of various edge intelligence applications [25], such as autonomous driving [20] and augmented reality [14]. These applications utilize deep neural network (DNN) models to perform various complex tasks. However, it is non-trivial to deploy compute-intensive DNN models to IoT devices due to limited resources. In this case, edge computing [18] has emerged as a promising paradigm for providing low-latency inference services by deploying models to edge devices, which are in closer proximity to users than cloud servers [1].

As shown in Fig. 1, an edge inference service usually involves a multi-tenant environment [6] comprised of various IoT devices. These IoT devices send their inference requests to a nearby edge device, or in our case an edge cluster, on which computing resources are allocated among multiple tenants and DNN models. Existing edge inference serving systems adopt a wide range of approaches to process as many requests as possible, i.e., achieve high throughput on resource-constrained edge devices. For instance, DeepRT [23] adopts batching to provide soft real-time inference services. Edgent [11] leverages multi-exit DNN models for collaborative inference. MAEL [17] uses cross-processor scheduling to satisfy service level objectives (SLO) of various requests. Indeed, a high-throughput edge inference serving system needs to trade-off among inference accuracy, latency and throughput. However, none of the works mentioned above targets inference serving on an edge cluster, which poses new challenges in optimizing in multi-dimensional search spaces.

An edge cluster equipped with GPUs located close to user devices can be used to improve throughput of DNN inference serving. Furthermore, edge inference services must be flexible to accommodate SLO budget, heterogeneous hardware accelerator, and inference accuracy requirement. Thus, for an SLO-aware inference serving system on an edge cluster, the scheduler should be capable of dispatching inference requests from IoT devices to appropriate edge nodes, where multiple DNN models are deployed, to satisfy different SLOs while maintaining high inference accuracy.

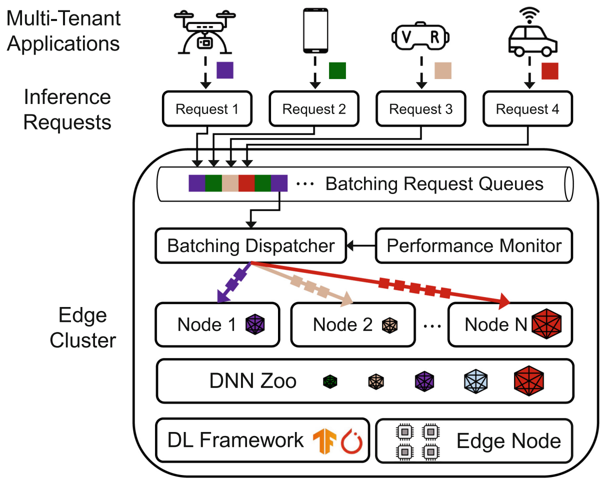


Fig. 1. Deep learning (DL) inference serving on a multi-tenant edge cluster.

To address these challenges, we propose Octopus, the first progressive inference serving system designed for a multi-tenant edge cluster, which aims at maximizing the overall throughput of the edge cluster while satisfying soft SLO budget and inference accuracy. Octopus adopts the multi-exit DNN inference approach [19], i.e., progressive inference, a mechanism that enables early exit at different points during DNN inference [10], given the request budget. The scheduler in Octopus utilizes deep reinforcement learning to efficiently schedule resources for inference requests. More precisely, Octopus automatically learns the optimal joint configuration of exit point, batch size, and node dispatching, in order to provide high-throughput progressive inference serving while taking into account SLO and accuracy budget. Additionally, the latency predictor in Octopus leverages an attention-based long short-term memory (LSTM) to achieve SLO awareness, and ensure bounded response latency for inference requests.

Overall, this paper makes the following contributions:

– We propose a novel multi-exit DNN-based progressive edge inference serving system, aiming to maximize the overall throughput of a heterogeneous edge cluster while satisfying SLO budget and maintaining high accuracy.
– We design a deep reinforcement learning-based scheduler that automatically co-optimizes a three-dimensional search space with batch size, exit point, and node dispatching to provide high-throughput inference services for multi-tenant edge intelligence applications.
– We implement a system prototype of Octopus on a heterogeneous edge cluster, deploying three representative CV and NLP DNN models. Extensive evaluations show that Octopus achieves up to 3.3× in overall throughput compared to state-of-the-art schemes, while maintaining high inference accuracy and low SLO violation rate below 5%.

The rest of this paper is organized as follows: Sect. 2 introduces related work. Section 3 illustrates the system architecture and formulates the optimization problem. Section 4 proposes an SLO-aware latency predictor. Section 5 details the design of the learning-based scheduler. Section 6 provides the system prototype and performance comparison. Section 7 summarizes our work.

2 Related Work

Edge inference services have recently attracted great attention among researchers. Prior work utilizes multi-exit DNN to efficiently share limited resources on edge devices. For instance, Delen [12] adopts multi-exit DNN to adaptively control inference requests with SLO, accuracy, and energy budget. Edgent [11] leverages an early exit mechanism to achieves collaborative inference between end devices and edge servers, balancing latency and accuracy. MAMO [3] proposes a bidirectional dynamic programming approach to determine the optimal exit point, and utilizes deep reinforcement learning to co-optimize resource allocation and model partitioning. However, none of these works provides SLO budget. In practice, edge devices must respond to inference requests within bounded latency, so as to provide QoS consistent with SLO budget.

Prior work also proposes various scheduling algorithms for single-device edge inference services. For instance, DeepRT [23] proposes a scheduler based on earliest-deadline-first (EDF) [4], which aims to provide soft real-time inference services. Jellyfish [16] leverages dynamic programming that adapt input data and

DNNs, so as to provide soft SLOs while maintaining high accuracy. HiTDL [22] proposes a latency-based performance model that considers resource availability, DNN exit points, and cross-DNN interference, in order to improve throughput while satisfying SLO. However, these works only provide limited inference services due to the lack of an efficient resource sharing.

Additionally, some research focus on scheduling heterogeneous multiprocessor to provide on-device edge inference. For instance, BlastNet [13] introduces a priority-driven algorithm for block-level scheduling across CPU-GPU processors. Similarly, Band [9] schedules the subgraphs of DNN model on heterogeneous multiprocessor to coordinate multi-DNN inference. MAEL [17] proposes a heterogeneous multiprocessor-aware scheduling strategy for edge devices equipped with CPU, GPU, and DSP, using a minimum-average-expected-latency algorithm to satisfy SLO while reducing energy consumption. Note that these works are orthogonal to Octopus, which can be used to further improve throughput.

3 System Architecture and Problem Formulation

In this section, we illustrate the workflow procedure of our proposed progressive inference serving framework for multi-tenant edge cluster, and formulate the scheduling problem as an optimization problem.

3.1 System Overview

Figure 2 shows an overview of the proposed Octopus system, which comprises multiple clients and an edge cluster with heterogeneous devices. When multiple clients ❶ send batch inference requests to the edge cluster via a network, the monitor ❷ in Octopus generates configuration files that specify the SLO budget and accuracy threshold for each request. Meanwhile, the latency predictor ❸ utilizes historical data to estimate the end-to-end latency of subsequent requests, thereby achieving SLO awareness (Sect. 4). The learning-based scheduler ❹ then learns the optimal batch size, exit point, and node dispatching for each request based on the collected request information and predicted latency (Sect. 5). Next, each edge node ❺ deploys multi-exit DNN models using the joint optimal configuration from the scheduler. Finally, the inference results ❻ are sent back to clients, thus completing an end-to-end inference request.

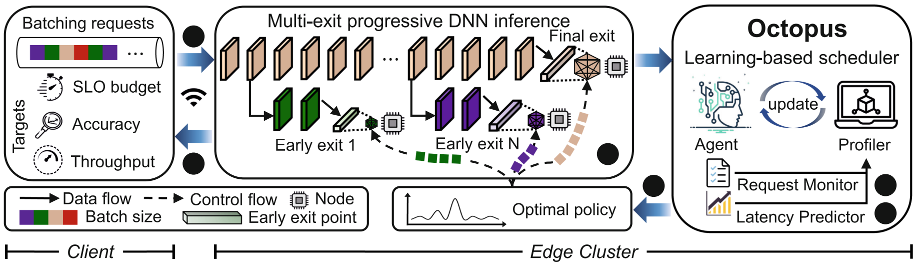


Fig. 2. Overview of Octopus system architecture.

3.2 Problem Formulation

Let $\mathbb{K} = \{1, 2, \ldots, K\}$ represent the set of inference requests. Each request $k \in \mathbb{K}$ has its input data size D_k, network bandwidth $\mathcal{B}_k$, expected accuracy ξ_k, request rate τ_k and SLO budget s_k. The edge cluster comprises multiple heterogeneous edge nodes, denoted as $\mathbb{N} = \{1, 2, \ldots, N\}$, The computility (i.e., floating point operations per second) and hardware clock frequency of each edge node $n \in \mathbb{N}$ are represented by C_n and f_n, respectively. The higher the hardware clock frequency, the bigger the computility. An ensemble of domain-specific DNN models (such as CV and NLP) forms a DNN Zoo, denoted as $\mathbb{M} = \{1, 2, \ldots, M\}$. The inference latency of each DNN model $m \in \mathbb{M}$ is related to the batch size b and exit point q. The set of batch sizes is denoted as $\mathbb{B} = \{1, 2, \ldots, B\}$. The larger the batch size, the higher the throughput. The set of exit points for a DNN model is denoted as $\mathbb{Q} = \{1, 2, \ldots, Q\}$. Each exit point $q (q \in \mathbb{Q})$ is sorted in ascending order according to inference latency. Note that the later the exit point, the higher the latency and the accuracy.

The end-to-end latency of the i-th request k comprises network latency and inference latency. More precisely, network latency is modeled as a function of input data size D_k and network bandwidth $\mathcal{B}_k$. Inference latency is related to the input data size D_k, the computility C_n and the clock frequency f_n of each edge node, and the batch size b_k, which can be formulated as:

$$t_k(b, n, m, q) = \frac{D_k}{\mathcal{B}_k} + \frac{D_k \cdot C_n}{f_k \cdot b_k}, \tag{1}$$

Inspired by [16], we introduce a binary decision variable $\psi_{b,n,m,q} \in \{0, 1\}$ to indicate whether a request k is dispatched to DNN m with exit point q and batch size b deployed on edge node n. We first model the throughput on a single edge node. Each edge node deploys multiple DNN models with varying exit points and batch sizes. The throughput of each node processing K inference requests can be formulated as:

$$rps_k(b, n, m, q) = \frac{K}{\sum_K t_k(b, n, m, q)} \cdot \psi_{b,n,m,q}, \tag{2}$$

The goal of Octopus is to maximize the overall throughput of the edge cluster while satisfying the SLO budget and maintaining high accuracy for each inference request. Based on the throughput of an individual edge node as defined in Eq. (2), the scheduling problem can be formulated as:

$$\min_{\psi} \sum_{n=1}^{N} rps_k(b, n, m, q) \cdot \tau_k \cdot \psi_{b,n,m,q} \tag{3}$$

$$s.t. \sum_{b=1}^{B} \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{q=1}^{Q} \psi_{b,n,m,q} = 1, \forall k \in K \tag{4}$$

$$ITA_k(b, n, m, q) \geq \xi_k, \forall k \in K \tag{5}$$

$$t_k(b, n, m, q) \leq s_k, \forall k \in K \tag{6}$$

$$\tau_k \cdot \psi_{b,n,m,q} \leq rps_k(b, n, m, q), \forall k \in K \tag{7}$$

$$\psi_{b,n,m,q} \in \{0, 1\}, \forall b \in B, \forall n \in N, \forall m \in M, \forall q \in Q \tag{8}$$

$$ModelSize(m_q) + PeakSize(m_q) + BufSize(m_q) \leq Memory_{avl}^{n} \tag{9}$$

where Eq. (3) defines the maximizing overall throughput as the optimization objective. Equation (4) ensures that each inference request can only be dispatched to a single edge node. $ITA_k(b, n, m, q)$ is the inference-to-accuracy of request k on DNN m_q deployed on node n with exit point q. Equation (5) specifies that $ITA_k(b, n, m, q)$ is higher than the accuracy budget ξ_k. Equation (6) enforces latency budget, that is, the end-to-end latency should not exceed the SLO budget. Equation (7) ensures that each edge node has enough resources to support batch inference. Equation (8) illustrates that the dispatch of request is a binary variable, meaning that a DNN model cannot be partitioned for distributed inference. Equation (9) considers the limited memory resources of edge nodes. Since multi-exit DNN models require intensive memory for inference [8], it is necessary to load the weight matrix $ModelSize(m_q)$, the intermediate feature matrix $PeakSize(m_q)$, and the buffer size $BufSize(m_q)$ into memory to speed up inference. The total memory requirement should not exceed the available memory of the edge device $Memory^n_{avl}$.

4 SLO-Aware Latency Predictor

Octopus predicts the latency of the current batch requests based on the previous batches. Prior work [5] has revealed that DNN inference is highly predictable. Importantly, there is a highly correlated temporal relationship between consecutive requests, such as video streams for object monitoring. Attention mechanism [21] and long short-term memory (LSTM) [7] have shown impressive effectiveness in predicting long- and short-term time series data, respectively. Inspired by [24], we adopt an attention-based LSTM as latency predictor, to achieve SLO-awareness for batch requests. The latency predictor aims to minimize the error between the predicted and actual latency for batch requests:

$$\min_{\sigma} \mathcal{L}(\hat{L}^t_b, L^t_b) = \min \textstyle\sum_b [(\hat{l}^t_b - l^t_b)^2] \tag{10}$$

$$s.t.\ \hat{L}^t_b = f(\{l^{t-N}_b, l^{t-N+1}_b, , l^{t-1}_b, \dots\}, \sigma) \tag{11}$$

where L^t_b is the actual latency of batch requests with batch size b at time slot t, and $\hat{L}^t_b$ is the corresponding predicted latency. l^t_b and $\hat{l}^t_b$ represent the actual and predicted latency of the b-th inference request in the batch requests, respectively. σ is the LSTM parameter, and N represents the number of previous batch requests used in the prediction network $f(\cdot)$.

As shown in Fig. 3, the attention-based LSTM-based latency predictor is composed of an encoder, an attention module and a decoder.

Encoder. The encoder is implemented using a two-layer LSTM. To accurately predict the latency of the b-th request in the current batch requests, the encoder takes as input the latency corresponding to the b-th request in the past N batches of requests, and encodes it into a feature map $\{Y_t\}$:

$$Y_t = f(Y_{t_1}, L^N_b) \tag{12}$$

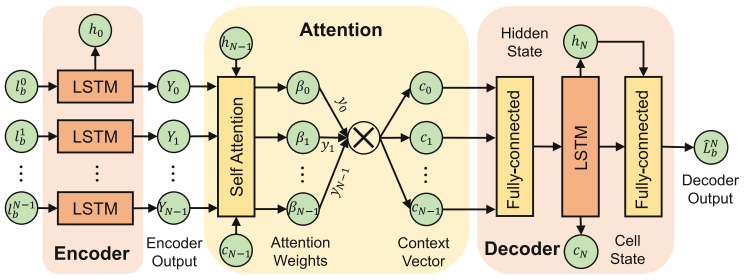


Fig. 3. Overview of the attention-based LSTM for latency predictor.

where $f(\cdot,\cdot)$ denotes the LSTM network.

Attention Module. We use an attention module with a fully-connected layer to evaluate the importance of encoded feature maps. The weight of the feature map generated by the attention module can be formulated as:

$$\mu_t = W_1 \tanh(W_2[Y; c_{N-1}; h_{N-1}]) \tag{13}$$

$$\beta(Attn)_t = \frac{\exp(\mu_t)}{\sum_t \exp(\mu_t)} \tag{14}$$

where $[Y; c_(N-1); h_(N-1)]$ represents the encoder output Y, the state vector $c_(N-1)$ of decoder unit, and the hidden state vector of decoder $h_(N-1)$. W_1 and W_2 are the weights that need to be optimized. $\beta(Attn)_t$ is the normalized weight of different feature maps. The context vector $c_(N-1)$ is used to evaluate the contribution of each feature map.

Decoder. The decoder is implemented using two fully-connected layers and an LSTM, which processes the context vectors.

5 Learning-Based Scheduler

Complexity Analysis: Octopus aims to find the optimal batch size, exit point and node assignment for each inference request to maximize the overall throughput of the edge cluster. The challenge with the three-dimensional scheduling space for Octopus is that scheduling decisions are affected by several interdependent variables. More precisely, the batch size and exit point depend on the computing resources of the allocated nodes to ensure SLO and accuracy. To represent the search space, suppose Q is the number of exit points in a DNN model, M is the number of DNN models to serve, and B is the number of batch sizes on each multi-exit DNN model. Therefore, there are a total of $B^M Q^M$ possible options to configure M DNN models. Since there are N edge nodes in an edge cluster, the complexity of the search space is as follows:

$$Total\ Search\ Space = O(NQ^M B^M) \tag{15}$$

Solving such a huge search space is non-trivial. Exhaustively search and heuristic-based approaches are unable to handle the problem in polynomial time. In contrast, deep reinforcement learning (DRL) considers the impact of current decisions on future outcomes by using Markov decision process (MDP), and learns the optimal policy to maximize cumulative returns, enabling it to be suitable for complex decision problems in multi-dimensional search spaces. Consequently, we propose leveraging DRL to automatically learn the optimal joint configuration of batch size, exit point, and node dispatching.

Markov Decision Process. In DRL, the agent continuously interacts with the environment and makes decisions via a policy, which is achieved using Markov decision process (MDP). Consequently, we first transform the scheduling problem in Eq. (3) into an MDP. An MDP can be represented as a three-tuple: state space $\mathcal{S}$, action space $\mathcal{A}$ and reward function r, which are described as follows:

- **State:** At each scheduling time slot t, the agent in DRL constructs a state $s_t (s_t \in \mathcal{S})$ to periodically collect the information of inference requests. We define the state s_t using four components: (I) Input data size D_t. (II) Bandwidth $\mathcal{B}_k$. (III) Request rate τ_k. (IV) Predicted latency $\hat{L}_b^{N+1}$.
- **Action:** The action represents a decision made by the agent based on the current state. We define the action as the choosing of the appropriate batch size b, exit point q, and edge node n for each multi-exit DNN model, which can be denoted as $a_t = (b, q, n)$.
- **Reward:** The agent aims to maximize the cumulative expected reward $\mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$, where $\gamma \in [0, 1]$ is a discount factor. r_t denotes the immediate reward obtained when the agent executes inference after choosing the appropriate batch size, exit point, and edge node. We define the reward function r_t using the accuracy τ_k and SLO violation rate s_k, based on Eq. (3):

$$r_t(\xi; \eta) = \frac{1}{1 + e^{-\xi/\eta}} \sum_{n=1}^{N} rps_k(b, n, m, q) \cdot \tau_k \cdot \psi_{b,n,m,q} \tag{16}$$

Maximum Entropy Reinforcement Learning-Based Scheduling Algorithm. Our proposed scheduling algorithm is based on the discrete soft Actor-Critic (SAC) [2] framework. SAC is maximum entropy DRL algorithm, which aims to maximize both the reward and the entropy of the visited states, enabling the agent in DRL to learn more near-optimal actions and accelerating the learning process. Meanwhile, it also allows the agent to explore a larger search space and enhances the robustness of the system.

The policy π is the function that determines the next action chosen by the agent based on current state. The optimal strategy π^* can be formulated as:

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{T} E_{(s_t, a_t) \sim \rho_\pi} [r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot \mid s_t))] \tag{17}$$

where ρ_π is the trace distribution generated by the policy π. α is the temperature parameter that balances the relative importance of entropy and reward. $\mathcal{H}(\pi(\cdot \mid s_t)) = -\log \pi(\cdot \mid s_t)$ denotes the entropy of policy π with state s_t.

We utilize soft policy iteration [2] to achieve the optimal policy, which consists of soft policy evaluation and soft policy improvement. These two steps are alternated during the training process.

Soft policy evaluation involves the calculation of the policy value, that is, soft state value function (V-function) and soft action-state value function (Q-function). The discrete V-function with entropy is defined as:

$$V\left(s_t\right) := \pi\left(s_t\right)^T\left[Q\left(s_t\right)-\alpha \log \left(\pi\left(s_t\right)\right)\right] \tag{18}$$

We then obtain the soft Q-function using the soft Bellman equation:

$$\mathcal{T}^\pi Q\left(\mathbf{s}_t, \mathbf{a}_t\right) \triangleq r\left(\mathbf{s}_t, \mathbf{a}_t\right)+\gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p}\left[V\left(\mathbf{s}_{t+1}\right)\right] \tag{19}$$

where $\mathcal{T}_\pi$ is the modified Bellman backup operator. Based on the soft Bellman equation in Eq. (19), soft policy evaluation can converge to the soft Q-function of the optimal policy π^* under limited state and action spaces.

We update the policy using the following soft policy improvement:

$$\pi_{\text{new}} = \arg \min_{\pi' \in \Pi} \mathrm{D}_{\mathrm{KL}}\left(\pi'\left(\cdot \mid \mathbf{s}_t\right) \| \frac{\exp \left(\frac{1}{\alpha} Q^{\pi_{\text{old}}}\left(\mathbf{s}_t, \cdot\right)\right)}{Z^{\pi_{\text{old}}}\left(\mathbf{s}_t\right)}\right) \tag{20}$$

where D_{KL} denotes the KL divergence, and $Z^{\pi_{old}}(s_t)$ is the partition function.

In our approach, we utilize two Q-networks, and for each training step, we select the network with the lower Q-value, in order to alleviate the overestimation of Q-values. The loss function of any Q-network $\mathcal{L}_Q(\theta)$ is calculated by minimizing the soft Bellman residual in Eq. (19):

$$\begin{aligned}\mathcal{L}_Q(\theta) = E_{(s_t,a_t)\sim\mathcal{D},a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}[\frac{1}{2}(Q_\theta(s_t,a_t)-(r(s_t,a_t)\\ +\gamma(\min_{j=1,2} Q_{\theta_j}(s_{t+1},a_{t+1})-\alpha\log_\pi(a_{t+1}\mid s_{t+1}))))^2]\end{aligned} \tag{21}$$

where $\mathcal{D}$ is the replay buffer that used to store the collected historical traces. To facilitate more stable training, we utilize two target Q-networks $\bar{Q}_{\bar{\theta}_j}(j=1,2)$ that correspond to the two Q-networks used to calculate Q-values.

The loss function of policy $\mathcal{L}_\pi(\varphi)$ is calculated by minimizing the KL divergence in Eq. (20):

$$\mathcal{L}_\pi(\varphi) = E_{s_t\sim D}\left[\pi_t\left(s_t\right)^T\left[\alpha\log\left(\pi_\varphi\left(s_t\right)\right)-Q_\theta\left(s_t\right)\right]\right] \tag{22}$$

Note that it is critical to choose the appropriate temperature parameter α. For instance, in states where the optimal action is highly uncertain, the importance of entropy should be increased. As a reference [2], the loss function of the temperature parameter $\mathcal{L}_\alpha$ is formulated as:

$$\mathcal{L}_\alpha = \pi_t(s_t)^T[-\alpha(\log(\pi_t(s_t))+\bar{H})] \tag{23}$$

where $\bar{\mathcal{H}}$ is a constant vector. More precisely, when the entropy of the policy is lower than $\bar{\mathcal{H}}$, the loss function $\mathcal{L}_(\alpha)$ will increase the value of α, thereby enhancing the importance of entropy during training.

Algorithm 1 provides an overview of our proposed learning-based scheduling search algorithm. We take as input the request information collected by the profiler and the latency predicted by the latency predictor. Before training, we first initialize all network weights and the replay buffer (*line 1*∼*3*). For each training episode, we take the current request as the initial state of the environment (*line 5*). At each environment step, we select an action a_t (*line 7*) based on the current policy $\pi_\varphi(a_t \mid s_t)$, and execute the action while receiving a reward (*line 8*). Next, the scheduler feeds back the decisions made by the DRL to the corresponding edge nodes for progressive inference (*line 9*). When inference is complete, DRL updates the environment state (*line 10*) while storing the current trajectory in the replay buffer (*line 11*). For each gradient step, we calculate the soft state value and Q value by random sampling (*line 14*∼*15*), and update all network weights and the temperature parameter (*line 16*∼*19*). As this process repeats, the learning-based algorithm eventually converges on the optimal policy that maximizes the overall throughput of the edge cluster.

Algorithm 1: Learning-based scheduling search algorithm.

```
Input  : set of requests $\mathbb{K} = \{1, 2, \ldots, K\}$, information per request $k$ with
         input data size $D_k$, bandwidth $\mathcal{B}_k$, predicted latency $\hat{L}_b^N$ and request
         rate $\tau_k$, target budget with accuracy $\xi_k$ and SLO $s_k$
Output: the optimal schedule $\{b_k, n_k, q_k\}$ for each request $k$
1  Initialize actor network $\pi(s \mid \varphi)$ with $\varphi$ and critic network $Q_{\theta_1}, Q_{\theta_2}$ with $\theta_1$ and
   $\theta_2$, respectively
2  Initialize target network $\bar{Q}_{\bar{\theta}_1}, \bar{Q}_{\bar{\theta}_2}$: $\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$
3  Initialize an empty replay buffer $\mathcal{D} \leftarrow \emptyset$
4  for each epoch $e = 1 \rightarrow E$ do
5      Generate current request $k(D_k, \mathcal{B}_k, tau_k, \hat{L}_b^N)$ with target budget $\{\tau_k, s_k\}$ as
       the initial state of the environment $s_1$
6      for each environment step $t = 1 \rightarrow T$ do
7          Sample action $a_t = s_t(b_k, n_k, q_k) \sim \pi_\varphi(a_t \mid s_t)$
8          Execute action $a_t$ and obtain instant reward $r_t\,(a_t \mid s_t)$ using Eq. (16)
9          Execute progressive inference with action $a_t(b_k, n_k, q_k)$
10         Update state $s_t \leftarrow s_{t+1}$
11         Store the transition $(s_t, a_t, r(s_t, a_t), s_{t+1})$ in the replay buffer $\mathcal{D}$
12     end
13     for each gradient step $g = 1 \rightarrow G$ do
14         Sample transition from the environment $s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)$
15         Calculate the soft state value $V\,(s_t)$ with policy $\pi$ using Eq. (18) and
           the soft Q-function $Q(s_t, a_t)$ using Eq. (19), respectively
16         Update critic network weights $\theta_i$ for $i \in \{1, 2\}$ using Eq. (21)
17         Update actor network weight $\varphi$ using Eq. (22)
18         Update temperature parameter $\alpha$ using Eq. (23)
19         Update target network weights $\bar{Q}_{\bar{\theta}_1} \leftarrow \lambda Q_{\theta_i} + (1-\lambda)\bar{Q}_{\bar{\theta}_1}$ for $i \in \{1, 2\}$
20     end
21 end
```

6 Prototype and Performance Evaluation

6.1 Implementation

Octopus Prototype. Octopus is implemented using PyTorch. We use an NVIDIA Xavier NX as the master node to receive inference requests from multiple clients. Additionally, we employ three heterogeneous edge devices as nodes to execute inference for specific multi-exit DNN models. The detailed configurations of each edge device are detailed in Table 1. For offline training of Algorithm 1, we use an edge server equipped with four NVIDIA GeForce GTX 3080 GPUs, using a mini-batch size of 128 for 2000 epochs. All networks are trained using the Adam optimizer with a learning rate of 10^{-3}. Each network comprises a two-layer ReLU neural network with 64 and 32 hidden units, respectively. The size of the replay buffer is fixed at 10^6. The trained learning-based scheduler is ultimately deployed online on the master node.

Table 1. The detailed configurations of edge devices.

Edge Device	CPU	GPU	Memory	Computility
NVIDIA Jetson Nano	ARM Cortex-A57	128-core Maxwell	4 GB	0.47TFLOPS
NVIDIA Jetson TX2	ARM Cortex-A57	256-core Pascal	8 GB	1.33TFLOPS
NVIDIA Xavier NX	Carmel ARMv8.2	384-core Volta	8 GB	21TOPS

DNN Zoo and Datasets. Three domain-specific DNN models are used to process image and speech data, as summarized in Table 2. We adopt the BranchyNet framework [19], which supports multi-exit DNN training with five early exit points per DNN model. The DNN Zoo, comprising these multi-exit DNN models, is deployed on each edge node.

Table 2. The specific information for inference requests.

Request Type	DNN Model	Dataset	SLO(ms)	Accuracy(%)
Object Detection	YOLOv4-Tiny	VOC-2012	50	64.31
Semantic Segmentation	EfficientViT-B1	Cityscapes	75	81.65
Natural Language Processing	BERT-Base	SQuAD v1.1	25	79.52

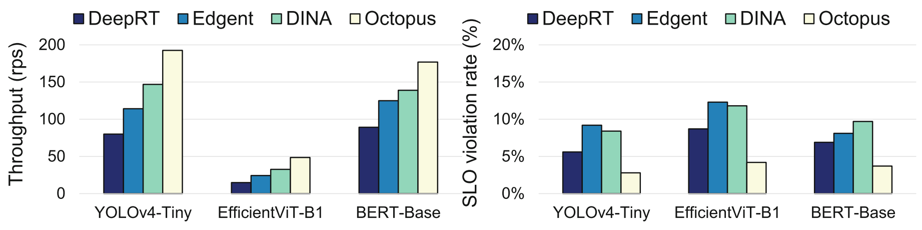
Baselines. We compare Octopus with three baselines: DeepRT [23] develops a soft real-time scheduler for single edge device that leverages earliest-deadline-first (EDF) [4] to schedule batch requests. DINA [15] utilizes matching theory to achieve distributed inference with adaptive DNN partitioning and offloading. Edgent [11] proposes a regression-based predictive model for multi-exit DNN inference through device-edge synergy. Since our proposed Octopus is the first framework for multi-tenant progressive inference serving on heterogeneous edge clusters, we scale three baselines to the edge cluster in Table 1 and compare the sum of their throughput on each edge node, for a fair comparison.

Workloads and Network. We synthesize workloads using the three datasets detailed in Table 2, and use three Jetson Nano edge devices as multi-clients to generate inference requests based on these synthetic workloads. Note that inference requests arrive randomly to simulate the real-world applications, and each client always submits inference requests for a specific DNN model. The default request rate is fixed at $30rps$, unless otherwise specified. Besides, we use $WiFi$ to connect clients and edge devices, with available bandwidth ranging from 2 Mbps to 24 Mbps to simulate fluctuations in dynamic network conditions.

6.2 End-to-End Performance

Overall Throughput Improvement. As shown in Fig. 4(a), the overall throughput of Octopus, as detailed in Table 2, consistently outperforms the baselines. More precisely, Octopus achieves 1.3×–3.3× improvement in overall throughput. Although DeepRT utilizes batching to improve throughput, it suffers from resource-constrained edge devices and high memory overhead associated with executing entire DNN models. As a result, its throughput is lower than those of the two multi-exit DNN-based methods, Edgent and DINA, which do not utilize batching. Octopus takes advantage of both batching and multi-exit inference to improve the overall throughput of the edge cluster while significantly reducing resource occupancy.

SLO Violation Rate. We also analyzed the SLO violation rate of Octopus at a request rate of $30rps$. As shown in Fig. 4(b), Octopus exhibits the lowest SLO violation rate, below 5%, thanks to its SLO-aware latency predictor. The admission control module in DeepRT aims to reduce the SLO violation rate by analyzing the schedulability of inference requests, but it ignores the temporal relationship between inference requests, resulting in a higher SLO violation rate than Octopus. Edgent and DINA do not focus on SLO awareness for inference requests, and thus have significantly higher SLO violation rates.

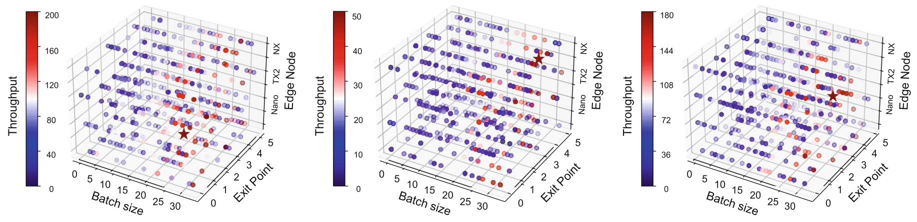


(a) Overall throughput (b) SLO violation rate

Fig. 4. The end-to-end performance of Octopus.

6.3 Visualization of Scheduler in Three-Dimensional Search Space

As shown in Fig. 5(a), within the complex three-dimensional search space, the scheduler chooses the Jetson Nano with the lowest computility, a moderate batch size, and a moderate exit point for YOLOv4-Tiny, which has the lowest computing density. For EfficientViT-B1 in Fig. 5(b), which has the highest computing density, the optimal configuration, is a larger batch size, the Xavier NX with the highest computility, and a later exit point. For BERT-Base, which has a computing density between that of YOLOv4-Tiny and EfficientViT-B1, as shown in Fig. 5(c), the scheduler chooses the Jetson TX2 with moderate computility, a larger batch size and a later exit point. Overall, Octopus can seamlessly adapt to heterogeneous edge nodes and different multi-exit DNN models for various inference requests, choosing the optimal batch size and exit point to maximize throughput while satisfying SLO budget and accuracy requirement.



(a) YOLOv4-Tiny (b) EfficientViT-B1 (c) BERT-Base

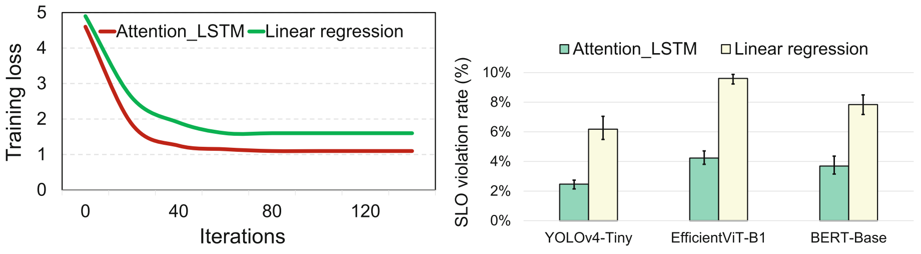
Fig. 5. The learning process of sheduler. ⋆ represents the optimal joint configuration.

6.4 Impact of Latency Predictor

To evaluate the effect of proposed SLO-aware latency predictor, we collected end-to-end latency and SLO for a total of 500 inference requests. We randomly selected 400 information of requests as training data and 100 data for validation. Figure 6(a) presents the training loss curve over 120 epochs. The results demonstrate that Octopus enables more accurate SLO-awareness based on contextual historical requests compared to the widely adopted linear regression-based predictive model used in prior work [1]. Furthermore, benefit from the combination of lightweight LSTM and attention, Octopus significantly reduces training loss and achieves similar convergence to linear regression. We also report the SLO violation rate in Fig. 6(b), which shows that the attention-based LSTM reduces the average SLO violation rate from 7.9% to 3.5%, compared with linear regression. It reveals that linear regression is inefficient for accurately predicting the SLO of unknown inference requests. In contrast, the attention-based LSTM focuses on the temporal relationship between inference requests, which utilizes neural networks to model the nonlinear relationship between inference latency and complex influencing factors, thereby effectively avoiding SLO violations.

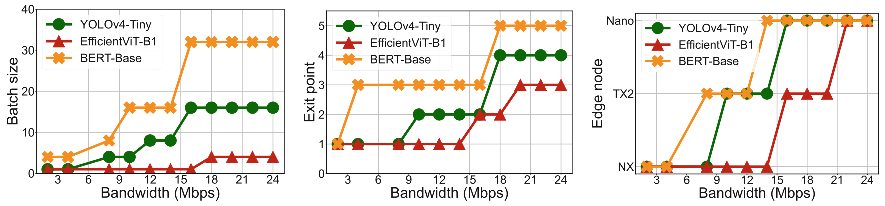
6.5 Impact of Network Bandwidth

In this section, we evaluate the impact of dynamic network conditions on the optimal configuration chosen by the scheduler in Octopus. The trend in Fig. 7(a) indicates that the batch size increases as network bandwidth improves, allowing more requests to be processed and resulting in higher throughput. Figure 7(b) shows that the scheduler chooses the earliest exit point for all requests when the available bandwidth is only 2 Mbps. Similarly, as the network bandwidth improves, the position of exit point is gradually moved back to improve accuracy while satisfying SLO. The results in Fig. 7(c) demonstrate that Octopus tends to schedule requests to edge nodes with high computility, such as Xavier NX, under poor network bandwidth conditions. In contrast, when bandwidth is not the bottleneck, Octopus schedules requests to edge nodes with low computility, such as Jetson Nano or TX2, to achieve load balancing of edge cluster.



(a) Training loss (b) SLO violation rate

Fig. 6. The performance of the proposed latency predictor.



(a) Batch size (b) Exit point (c) Edge node

Fig. 7. The impact of dynamic network on the optimal configuration.

6.6 Evaluation of Scalability

Different Request Rates. Since the performance of scheduler, especially the position of the exit point, is affected by the request rate, we evaluate inference accuracy by gradually increasing the request rate. Note that DeepRT, which does not employ multi-exit DNNs, exhibits the highest accuracy. Figure 8 shows that Octopus outperforms Edgent and DINA in terms of accuracy, with the performance improvement increases as the request rate increases. For instance, at 50rps request rate, the average accuracy improvement of Octopus is up to 10.6% and 7.6% compared to Edgent and DINA, respectively. Additionally, the accuracy loss of Octopus remains within 5%. The results indicate that the SLO-aware latency predictor and learning-based scheduler enable Octopus to handle high request rates while maintaining high accuracy.

Number of Edge Nodes. We evaluate the scalability of Octopus by scaling the number of edge nodes. As shown in Fig. 9(a), the average overall throughput improvement of Octopus with eight nodes is 3.1×, 2.1× and 1.2× that of two, four and six nodes, respectively. This indicates that the number of edge nodes is highly linear with the overall throughput of edge cluster. We also report the effect of the number of edge nodes on SLO violation rate and inference accuracy in Fig. 9(b). Intuitively, inference accuracy gradually improves as the number of edge nodes increases. For instance, the average accuracy of Octopus with eight nodes is 2.6%, 1.8% and 1% higher than that of two, four and six nodes, respectively. Additionally, the SLO violation rate at the default request rate (30rps) remains within 5%, demonstrating the flexible scalability of Octopus.

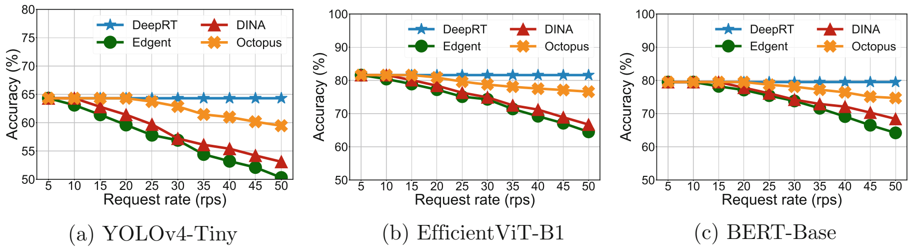


Fig. 8. The impact of different request rates on inference accuracy.

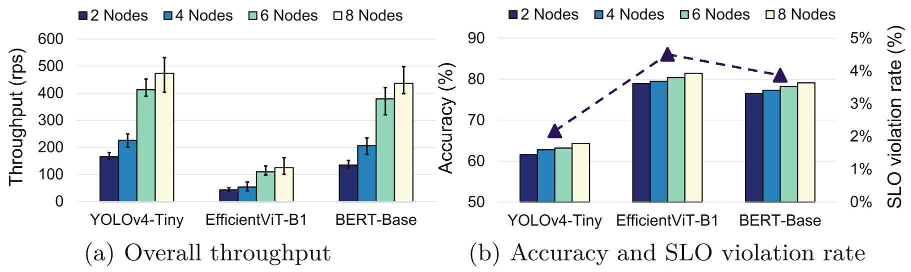


Fig. 9. The impact of number of edge nodes on throughput and inference accuracy.

7 Conclusion

In this paper, we propose Octopus, a multi-exit DNN-based progressive inference serving system for heterogeneous edge clusters. The learning-based scheduler in Octopus aims to maximize the overall throughput of edge clusters by automatically co-optimizing the joint configuration of batch size, exit point, and node dispatching for each inference request. Additionally, Octopus leverages an attention-based LSTM as a latency predictor to achieve SLO-aware. Our prototype implementation illustrates that Octopus has flexible scalability, and it can improve the overall inference serving throughput by up to 3.3$\times$ compared to the state-of-the-art schemes, while satisfying SLO and maintaining high inference accuracy. We emphasize that Octopus is primarily targets edge clusters, but is also applicable to individual edge devices. For the future work, Octopus can be combined with various inference optimization technologies (such as cloud-edge collaborative inference, compilation optimization, model compression, etc.) to further improve inference performance.

Acknowledgment. We thank our anonymous reviewers for their helpful comments and feedback. This work is partly supported by the National Key R&D Program of China under Grant No. 2021ZD0110905, and An Open Competition Project of Heilongjiang Province, China, on Research and Application of Key Technologies for Intelligent Farming Decision Platform, under Grant No. 2021ZXJ05A03.

References

1. Choi, S., Lee, S., Kim, Y., Park, J., Kwon, Y., Huh, J.: Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In: 2022 USENIX Annual Technical Conference (USENIX ATC 2022), pp. 199–216 (2022)
2. Christodoulou, P.: Soft actor-critic for discrete action settings. arXiv preprint arXiv:1910.07207 (2019)
3. Dong, F., et al.: Multi-exit DNN inference acceleration based on multi-dimensional optimization for edge intelligence. IEEE Trans. Mob. Comput. (2022)
4. Faggioli, D., Trimarchi, M., Checconi, F., Bertogna, M., Mancina, A.: An implementation of the earliest deadline first algorithm in linux. In: Proceedings of the 2009 ACM Symposium on Applied Computing, pp. 1984–1989 (2009)
5. Gujarati, A., et al.: Serving {DNNs} like clockwork: performance predictability from the bottom up. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020), pp. 443–462 (2020)
6. Hao, J., Subedi, P., Ramaswamy, L., Kim, I.K.: Reaching for the sky: maximizing deep learning inference throughput on edge devices with AI multi-tenancy. ACM Trans. Internet Technol. **23**(1), 1–33 (2023)
7. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
8. Jeon, S., Choi, Y., Cho, Y., Cha, H.: Harvnet: resource-optimized operation of multi-exit deep neural networks on energy harvesting devices. In: Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services, pp. 42–55 (2023)

9. Jeong, J.S., et al.: Band: coordinated multi-DNN inference on heterogeneous mobile processors. In: Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, pp. 235–247 (2022)
10. Laskaridis, S., Venieris, S.I., Almeida, M., Leontiadis, I., Lane, N.D.: Spinn: synergistic progressive inference of neural networks over device and cloud. In: Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, pp. 1–15 (2020)
11. 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**(1), 447–457 (2019)
12. Liang, Q., Hanafy, W.A., Bashir, N., Ali-Eldin, A., Irwin, D., Shenoy, P.: Dělen: enabling flexible and adaptive model-serving for multi-tenant edge AI. In: Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation, pp. 209–221 (2023)
13. Ling, N., Huang, X., Zhao, Z., Guan, N., Yan, Z., Xing, G.: Blastnet: exploiting duo-blocks for cross-processor real-time DNN inference. In: Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems, pp. 91–105 (2022)
14. Liu, Z., Lan, G., Stojkovic, J., Zhang, Y., Joe-Wong, C., Gorlatova, M.: Collabar: edge-assisted collaborative image recognition for mobile augmented reality. In: 2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), pp. 301–312. IEEE (2020)
15. Mohammed, T., Joe-Wong, C., Babbar, R., Di Francesco, M.: Distributed inference acceleration with adaptive DNN partitioning and offloading. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications, pp. 854–863. IEEE (2020)
16. Nigade, V., Bauszat, P., Bal, H., Wang, L.: Jellyfish: timely inference serving for dynamic edge networks. In: 2022 IEEE Real-Time Systems Symposium (RTSS), pp. 277–290. IEEE (2022)
17. Seo, W., Cha, S., Kim, Y., Huh, J., Park, J.: SLO-aware inference scheduler for heterogeneous processors in edge platforms. ACM Trans. Archit. Code Optim. **18**(4), 1–26 (2021)
18. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J. **3**(5), 637–646 (2016)
19. 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. IEEE (2016)
20. Teng, S., et al.: Motion planning for autonomous driving: the state of the art and future perspectives. IEEE Trans. Intell. Veh. (2023)
21. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
22. Wu, J., Wang, L., Pei, Q., Cui, X., Liu, F., Yang, T.: HiTDL: high-throughput deep learning inference at the hybrid mobile edge. IEEE Trans. Parallel Distrib. Syst. **33**(12), 4499–4514 (2022)
23. Yang, Z., Nahrstedt, K., Guo, H., Zhou, Q.: Deeprt: a soft real time scheduler for computer vision applications on the edge. In: 2021 IEEE/ACM Symposium on Edge Computing (SEC), pp. 271–284. IEEE (2021)
24. Zhang, W., et al.: ELF: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In: Proceedings of the 27th Annual International Conference on Mobile Computing and Networking, pp. 201–214 (2021)
25. 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**(8), 1738–1762 (2019)