SplitPlace: AI Augmented Splitting and Placement of Large-Scale Neural Networks in Mobile Edge Environments

Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings

Abstract—In recent years, deep learning models have become ubiquitous in industry and academia alike. Deep neural networks can solve some of the most complex pattern-recognition problems today, but come with the price of massive compute and memory requirements. This makes the problem of deploying such large-scale neural networks challenging in resource-constrained mobile edge computing platforms, specifically in mission-critical domains like surveillance and healthcare. To solve this, a promising solution is to split resource-hungry neural networks into lightweight disjoint smaller components for pipelined distributed processing. At present, there are two main approaches to do this: semantic and layer-wise splitting. The former partitions a neural network into parallel disjoint models that produce a part of the result, whereas the latter partitions into sequential models that produce intermediate results. However, there is no intelligent algorithm that decides which splitting strategy to use and places such modular splits to edge nodes for optimal performance. To combat this, this work proposes a novel AI-driven online policy, SplitPlace, that uses Multi-Armed-Bandits to intelligently decide between layer and semantic splitting strategies based on the input task's service deadline demands. SplitPlace places such neural network split fragments on mobile edge devices using decision-aware reinforcement learning for efficient and scalable computing. Moreover, SplitPlace fine-tunes its placement engine to adapt to volatile environments. Our experiments on physical mobile-edge environments with real-world workloads show that SplitPlace can significantly improve the state-of-the-art in terms of average response time, deadline violation rate, inference accuracy, and total reward by up to 46, 69, 3 and 12 percent respectively.

Index Terms—Mobile edge computing, neural network splitting, container orchestration, artificial intelligence, QoS optimization

1 Introduction

Modern Deep Neural Networks (DNN) are becoming the backbone of many industrial tasks and activities [2]. As the computational capabilities of devices have improved, new deep learning models have been proposed to provide improved performance [3], [4]. Moreover, many recent DNN models have been incorporated with mobile edge computing to give low latency services with improved accuracies compared to shallow networks, particularly in complex tasks like image segmentation, high frame-rate gaming and traffic surveillance [5]. The performance of such neural models reflects directly on the reliability of application domains like self-driving cars, healthcare and manufacturing [2], [6]. However, to provide high accuracy, such neural models are becoming increasingly demanding in terms of data and compute power, resulting in many challenging problems. To accommodate these increasing demands, such massive models are often hosted as web services deployed on the public cloud [7], [8].

- *Shreshth Tuli and Giuliano Casale are with the Department of Computing, Imperial College London, SW7 2BX London, U.K. E-mail: {s.tuli20, g.casale}@imperial.ac.uk.*
- *Nicholas R. Jennings is with the Department of Computing, Imperial College London, SW7 2BX London, U.K., and also with Loughborough University, LE11 3TU Loughborough, U.K. E-mail: n.r.jennings@lboro.ac.uk.*

Manuscript received 22 Oct. 2021; revised 6 May 2022; accepted 20 May 2022. Date of publication 24 May 2022; date of current version 4 Aug. 2023. (Corresponding author: Shreshth Tuli.) Digital Object Identifier no. 10.1109/TMC.2022.3177569

Challenges. Recently, application demands have shifted from either high-accuracy or low-latency to both of these together, termed as HALL (high-accuracy and low-latency) service delivery [2]. Given the prevalence and demand of DNN inference, serving them on a public cloud with tight bounds of latency, throughput and cost is becoming increasingly challenging [9]. In this regard, recent paradigms like mobile edge computing seem promising. Such approaches allow a robust and low-latency deployment of Internet of Things (IoT) applications close to the edge of the network. Specifically, to solve the problem of providing HALL services, recent work proposes to integrate large-scale deep learning models with modern frameworks like edge computing [9], [10], [11]. However, even the most recent approaches either provide a low Service Level Agreement (SLA) violation mode or a high-accuracy mode [9], [10] and struggle to provide the benefits of both modes at the same time.

Another challenge of using edge computing is that mobile edge devices face severe limitations in terms of computational and memory resources as they rely on low power energy sources like batteries, solar or other energy scavenging methods [12], [13]. This is not only because of the requirement of low cost, but also the need for mobility in such nodes [5]. In such systems, it is possible to handle the processing limitations of massive DNN models by effective preemption and prolonged job execution. However, memory bottlenecks are much harder to solve [14]. In a distributed edge environment, storage spaces are typically mapped to network-attached-storage (NAS) media. Thus, prior work that runs inference on a pre-trained DNN without memory-

aware optimizations leads to high network bandwidth overheads due to frequent overflow of memory and the use of virtual memory (swap space) on NAS media, making high fidelity inference using DNNs hard [10], [15]. To deploy an upgraded AI model, tech giants like Amazon, Netflix and Google usually consider completely revamping their infrastructures and upgrading their devices, raising many sustainability concerns [2]. This has made the integration of massive neural network models with such devices a challenging and expensive ordeal.

Solution. A promising solution for this problem is the development of strategies that can accommodate large-scale DNNs within legacy infrastructures. However, many prior efforts in this regard [16], [17], [18] have not yet tackled the challenge of providing a holistic strategy for not only distributed learning, but also inference in such memory-constrained environments. Recently, research ideas have been proposed like Cloud-AI, Edge-AI and Federated learning that aim to solve the problem of running enormous deep learning models on constrained edge devices by splitting them into modular fragments [17], [18]. However, in Cloud-AI where AI systems are deployed on cloud machines, the high communication latency leads to high average response times, making it unsuitable for latency-critical applications like healthcare, gaming and augmented reality [6], [19], [20]. Instead, Edge-AI provides low-latency service delivery, thanks to edge devices being in the same Local Area Network (LAN), where the input data from multiple edge nodes are combined to a single fixed broker node for processing. Edge-AI based methods aim at scheduling deep neural networks for providing predictable inference [21], [22]. However, due to the centralized collection of data, these solutions typically suffer from high bandwidth overheads and poor service quality [17]. Federated learning depends on data distribution over multiple nodes where the model training and inference are performed in a decentralized fashion. However, this paradigm assumes that neural models with data batches can be accommodated in the system memory. This is seldom the case for common edge devices like Arduinos or Raspberry Pis [23].

Other recent works offer lower precision models that can fit within the limited memory of such devices by using methods like Model Compression or Model Pruning [9], [24], [25]. However, compressed and low-precision models lose inference accuracy, making them unsuitable for accuracy-sensitive applications like security and intrusion detection [26]. Recently, split neural network models have been proposed. They show that using semantic or layer-wise splitting, a large deep neural network can be fragmented into multiple smaller networks for dividing network parameters onto multiple nodes [16], [27], [28], [29]. The former partitions a neural network into parallel disjoint models that produce a part of the result. The latter partitions a neural network into sequential models that generate intermediate results. We illustrate the accuracy and response time trade-offs through sample test cases in Section 2. Our experiments show that using layer and semantic splitting gives higher inference accuracies than previously proposed model compression techniques (see Section 6). However, no appropriate scheduling policies exist that can intelligently place such modular neural fragments on a distributed infrastructure to optimize both accuracy and SLA together. The placement of such split models is non-trivial considering the diverse and complex dynamism of task distribution, model usage frequencies and geographical placement of mobile edge devices [30].

Research Contributions. This work proposes a novel neural splitting and placement policy, *SplitPlace*, for enhanced distributed neural network inference at the edge. SplitPlace leverages a mobile edge computing platform to achieve low latency services. It allows modular neural models to be integrated for best result accuracies that could only be provided by cloud deployments. SplitPlace is the *first* splitting policy that dynamically decides between semantic and layer-wise splits to optimize both inference accuracy and the SLA violation rate. This decision is taken for each incoming task and remains unmodified until the execution of all split fragments of that task are complete. The idea behind the proposed splitting policy is to decide for each incoming task whether to use the semantic or layer-wise splitting strategy based on its SLA demands. Due to their quick adaptability, SplitPlace uses Multi-Armed-Bandits to model the decision strategy for each application type by checking if the SLA deadline is higher or lower than an estimate of the response time for a layer split decision [31]. Further, SplitPlace optimizes the placement decision of the modular neural network fragments using a split decision aware surrogate model. Compared to a preliminary extended abstract of this work [1], this paper provides a substantially expanded exposition of the working of MABs in SplitPlace. We also present techniques to dynamically adapt to non-stationary workloads and mobile environments. We present a gradient-based optimization approach for task placement decision conditioned on split decisions. Experiments on real-world application workloads on a physical edge testbed show that the SplitPlace approach outperforms the baseline approaches by reducing the SLA violation rate and improving the average inference accuracy.

Outline. The rest of the paper presents a brief background with motivation and related work in Section 2. Section 3 presents the system model assumptions and formulates the problem. Sections 4 and 5 give the model details of the proposed SplitPlace approach. We then validate and show the efficacy of the placement policy in Section 6. Finally, Section 7 concludes the work and proposes future directions. Additional experimental results are given in the Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TMC.2022.3177569, in the supplementary text.

2 Background and Related Work

As discussed in Section 1, there is a need for frameworks that can exploit the low latency of edge nodes and also high inference performance of DNNs to provide HALL services. However, complete neural models with the input batch can seldom be accommodated in the random-access-memory (RAM) of edge devices. Thus, ideas like model compression or splitting are required to make inference over large-scale neural networks plausible in such environments. Frameworks that aim at achieving this must maintain a careful

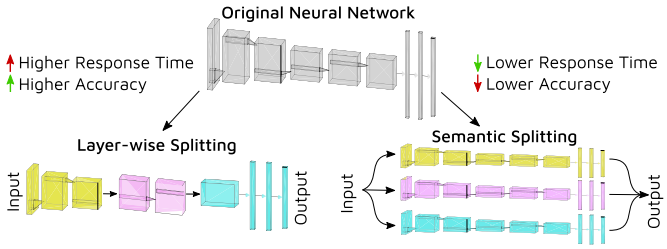


Fig. 1. Overview of layer and semantic splitting strategies.

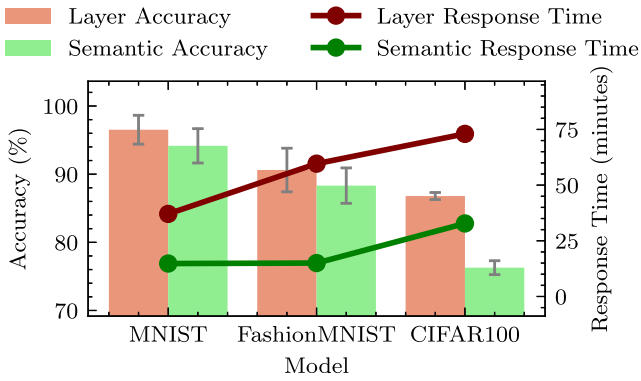


Fig. 2. Comparison of layer and semantic splits.

balance between accuracy requirements and response times for different user tasks. For such a framework, real-time analysis of the incoming tasks is required for quick decision making of task placement. This requires robust algorithms to seamlessly integrate different paradigms and meet the user's service level agreements.

Semantic and Layer Splitting. In this work, we leverage the only two available splitting schemes for neural networks: layer and semantic splitting [16], [32]. An overview of these two strategies is shown in Fig. 1. Semantic splitting divides the network weights into a hierarchy of multiple groups that use a different set of features (different colored models in Fig. 1). Here, the neural network is split based on the data semantics, producing a tree structured model that has no connection among branches of the tree, allowing parallelization of input analysis [16]. Due to limited information sharing among the neural network fragments, the semantic splitting scheme gives lower accuracy in general. Semantic splitting requires a separate training procedure where publicly available pre-trained models cannot be used. This is because a pre-trained standard neural network can be split layer wise without affecting output semantics. For semantic splitting we would need to first split the neural network based on data semantics and re-train the model. However, semantic splitting provides parallel task processing and hence lower inference times, more suitable for mission-critical tasks like healthcare and surveillance. Layer-wise splitting divides the network into groups of layers for sequential processing of the task input, shown as different colored models in Fig. 1.

Layer splitting is easier to deploy as pre-trained models can be just divided into multiple layer groups and distributed to different mobile edge nodes. However, layer splits require a semi-processed input to be forwarded to the subsequent edge node with the final processed output to be sent to the user, thus increasing the overall execution time. Moreover, layer-wise splitting gives higher accuracy compared to semantic splitting. Comparison of accuracies and average response times for the two strategies is shown in Fig. 2. The figure shows results for 10 edge worker nodes using popular image classification datasets: MNIST, FashionMNIST and CIFAR100 [33], [34], [35] averaged over ResNet50-V2, MobileNetV2 and InceptionV3 neural models [9]. As is apparent from the figure, layer splits provide higher accuracy and response time, whereas semantic splits provide lower values for both. SplitPlace leverages this contrast in traits to trade-off between inference accuracy and response time based on SLA requirements of the input tasks. Despite the considerable drop in inference accuracy when using semantic splitting scheme, it is still used in the proposed SplitPlace approach as it is better than model compression or early-exit strategies for quick inference. This is acceptable in many industrial applications [25], [36] where latency and service level agreements are more important performance metrics than high-fidelity result delivery. In this work, we consider a system with both SLA violation rates and inference accuracy as optimization objectives. This makes the combination of layer and semantic splitting a promising choice for such use cases.

2.1 Related Work

We now analyze the prior work in more detail. We divide our literature review into three major sections based on the strategy used to allow DNN inference on resource-constrained mobile-edge devices: model compression, layer splitting and semantic splitting. Moreover, we compare prior work based on whether they are suitable for edge-only setups (i.e., without leveraging cloud nodes), consider heterogeneous and mobile nodes and work in settings with adaptive Quality of Service (QoS). See Table 1 for an overview.

Model Compression. Efficient compression of DNN models has been a long studied problem in the literature [43]. Several works have been proposed that aim at the structural pruning of neural network parameters without significantly impacting the model's performance. These use approaches like tensor decomposition, network sparsification and data quantization [43]. Such pruning and model compression approaches have also been used by the systems research community to allow inference of massive neural models on devices with limited resources [44]. Recently, architectures like BottleNet and Bottlenet++ have been proposed [37], [38] to enable DNN inference on mobile cloud environments and reduce data transmission times. BottleNet++ compresses the intermediate layer outputs before sending them to the cloud layer. It uses a model re-training approach to prevent the inference being adversely impacted by the lossy compression of data. Further, BottleNet++ classifies workloads in terms of compute, memory and bandwidth bound categories and applies an appropriate model compression strategy. Other works propose to efficiently prune network channels in convolution neural models using reinforcement learning [39], [45]. Other efforts aim to prune the weights of the neural models to minimize their memory footprint [46]. Such methods aim at improving the accuracy per model size as a metric in contrast to the result delivery time as in BottleNet++. However, model compression does not leverage multiple

TABLE 1
Comparison of Related Works With Different Parameters (✓Means That the Corresponding Feature is Present)

Work	Edge Only	Mobility	Heterogeneous Environment	Adaptive QoS	Layer Split	Semantic Split	Model Compression	Optimization Parameters		
								Accuracy	SLA	Reward
[37], [38]							✓		✓	
[39]	✓		✓				✓	✓		
[27], [36], [40], [41]		✓			✓				✓	
[9]	✓		✓	✓	✓			✓	✓	✓
[11], [42]	✓			✓	✓					
[16], [25], [28], [36]	✓		✓	✓		✓		✓	✓	
[32]	✓		✓	✓	✓		✓	✓	✓	
SplitPlace	✓	✓	✓	✓	✓	✓		✓	✓	✓

compute nodes and has poor inference accuracy in general compared to semantic split execution (discussed in Section 6). Thus, SplitPlace does not use the model compression technique.

Layer Splitting. Many other efforts aim at improving the inference time or accuracy by efficient splitting of the DNN models. Some methods aim to split the networks layer-wise or vertically, *viz*, that the different fragments correspond to separate layer groups and hence impose the constraint of sequential execution. Most work in this category aims at segregating these network splits into distinct devices based on their computational performance [27], [36], [40], [41], [42]. In heterogeneous edge-cloud environments, it is fairly straightforward to split the network into two or three fragments each being deployed in a mobile device, edge node or a cloud server. Based on the SLA, such methods provide early-exits if the turnaround time is expected to be more than the SLA deadline. This requires a part of the inference being run at each layer of the network architecture instead of traditionally executing it on the cloud server. Other recent methods aim at exploiting the resource heterogeneity in the same network layer by splitting and placing DNNs based on user demands and edge worker capabilities [9]. Such methods can not only split DNNs, but also choose from different architectural choices to reach the maximum accuracy while agreeing to the latency constraints. Other works aim at accelerating the model run-times by appropriate scheduling of a variety of DNN models on edge-clusters [11]. The state-of-the-art method, *Gillis* uses a hybrid model, wherein it employs either model-compression or layer-splitting based on the application SLA demands [32]. The decision is taken using a reinforcement-learning model which continuously adapts in dynamic scenarios. As the model paritioning is also performed dynamically, the Gillis model cannot work with semantic splitting strategy that requires a new model to be trained for each partitioning scheme. It is a serverless based model serving system that automatically partitions a large model across multiple serverless functions for faster inference and reduced memory footprint per function. The Gillis method employs two model partitioning algorithms that respectively achieve latency optimal serving and cost-optimal serving with service-level agreement compliance. However, this method cannot jointly optimize both latency and SLA. Moreover, it does not consider the mobility of devices or users and hence is ineffective in efficiently managing large DNNs in mobile edge computing environments.

Semantic Splitting. The early efforts of semantic splitting only split the neural network at the input layer to allow model parallelization and size reduction [29]. Some methods divide the data batch itself across multiple nodes addressing computational contention problems but not memory limitations of fitting neural networks in the RAM [47]. Other methods use progressive slicing mechanisms to partition neural models into multiple components to fit in heterogeneous devices [25]. Such methods produce the complete output from each split or fragment of the DNN, adversely impacting the scalability of such methods to high-dimensional output spaces such as image segmentation applications [16], [28]. Moreover, these methods require significant cross-communication among network splits, significantly increasing the communication overheads. Recently, more intelligent approaches have been developed which hierarchically split neural networks such that each fragment produces a part of the output using an intelligently chosen sub-part of the input [16]. Such schemes use the "semantic" information of the data to create the corresponding links between input and output sub-parts being given to each DNN fragment, hence the name *semantic splitting*. Such splitting schemes require minimal to no interaction among network fragments eliminating the communication overheads and increased latency due to stragglers. As semantic splitting can provide results quickly, albeit with reduced accuracy, SplitPlace uses it for tasks with tight deadlines (Section 4).

3 System Model and Problem Formulation

In this work, we assume a scenario with a fixed number of multiple heterogeneous edge nodes in a broker-worker fashion, which is a typical case in mobile-edge environments [27], [38], [48], [49]. Here, the broker node takes all resource management related decisions, such as neural network splitting and task placement. The processing of such tasks is carried out by the worker nodes. Examples of broker nodes include personal laptops, small-scale servers and low-end workstations [48]. Example of common worker nodes in edge environments include Raspberry Pis, Arduino and similar System-on-Chip (SoC) computers [2]. All tasks are received from an IoT layer that includes sensors and actuators to collect data from the users and send it to the edge broker via the gateway devices. Akin to typical edge configurations [50], the edge broker then decides which splitting strategy to use and schedules these fragments to various edge nodes based on deployment constraints like sequential execution in a

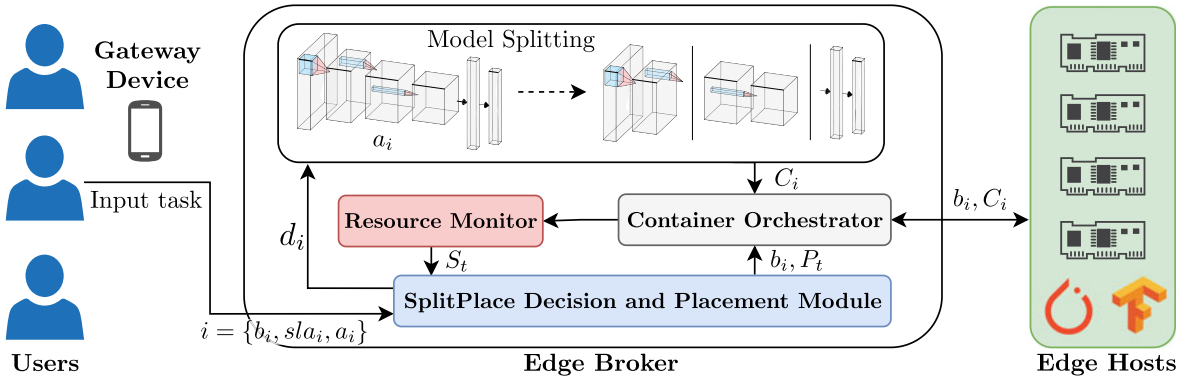


Fig. 3. SplitPlace system model.

TABLE 2
Table of Main Notation

Notation	Description
I_t	t-th scheduling interval
$\mathcal{T}_t$	Active tasks in I_t
$\mathcal{N}_t$	New tasks received at the start of I_t
$\mathcal{H}$	Set of workers in the edge layer
$i = \{b_i, sla_i, a_i\}$	Task as a collection input batch, SLA deadline and application type
$d^i \in \{L, S\}$	Splitting decision for input task i
C^i	Container realization of task i based on decision d^i
$C_t = \cup_{i \in \mathcal{T}_t} C^i$	Set of all active containers in the interval I_t
S_t	State of the system at the start of I_t
$P_t : C_t \times \mathcal{H}$	Placement decision of C_t to $\mathcal{H}$ as an adjacency matrix
O_t	Objective score for interval I_t

layer-decision. The data to be processed comes from the IoT sensors/actuators, which with the decision of which split fragment to use is forwarded by the broker to each worker node. Some worker nodes are assumed to be mobile, whereas others are considered to be fixed in terms of their geographical location. In our formulation, we consider mobility only in terms of the variations in terms of the network channels and do not consider the worker nodes or users crossing different networks. We assume that the CPU, RAM, Bandwidth and Disk capacities of all nodes are known in advance, and similarly the broker can sample the resource consumption for each task in the environment at any time (see *Resource Monitor* in Fig. 3). As we describe later, the broker periodically measure utilizations of CPU, RAM, Bandwidth and Disk for each task in the system. The broker is trusted with this information such that it can make informed resource management decisions to optimize QoS. Moreover, we consider that tasks include a batch of inputs that need to be processed by a DNN model. Further, for each task, a service level deadline is defined at the time the task is sent to the edge environment. We give an overview of the SplitPlace system model in Fig. 3. We decompose the problem into deciding an optimal splitting strategy and a fragment placement for each application (motivation in Appendix A.6, available in the online supplemental material, and more details in Section 1).

Workload Model. We consider a bounded discrete time control problem where we divide the timeline into equal duration intervals, with the tth interval denoted as I_t. Here, $t \in \{0, \ldots, \Gamma\}$, where $\Gamma + 1$ is the number of intervals in an execution. We assume a fixed number of worker machines in the edge layer and denote them as $\mathcal{H}$. We also consider that new tasks created at the interval I_t are denoted as $\mathcal{N}_t$, with all active tasks being denoted as $\mathcal{T}_t$ (and $\mathcal{N}_t \subseteq \mathcal{T}_t$). Each task $i \in \mathcal{T}_t$ consists of a batch input b_i, SLA deadline sla_i and a DNN application a_i. The set of all possible DNN applications is denoted by $\mathcal{A}$. For each new task $i \in \mathcal{N}_t$, the edge broker takes a decision d^i, such that $d^i \in \{L, S\}$, with L denoting layer-wise splitting and S denoting semantic split strategy. The collection of all split decisions for active tasks in interval I_t is denoted as $\mathcal{D}_t = \{d^i\}_{i \in \mathcal{N}_t}$. Based on the decision d^i for task i, this task is realized as an execution workflow in the form of containers C^i. Similar to a VM, a container is a package of virtualized software that contains all of the necessary elements to run in any environment. The set of all containers active in the interval I_t is denoted as $C_t = \cup_{i \in \mathcal{T}_t} C^i$. The set of all utilization metrics of CPU, RAM, Network Bandwidth and Disk for all containers and workers at the start of the interval I_t defines the state of the system, denoted as S_t. A summary of the symbols is given in Table 2.

3.1 Split Nets Placement

We partition the problem into two sub-problems of deciding the optimal splitting strategy for input tasks and that of placement of active containers in edge workers (see Fig. 4). Considering the previously described system model, at the start of each interval I_t, the SplitPlace model takes the split decision d^i for all $i \in \mathcal{N}_t$. Moreover, it also takes a placement decision for all active containers C_t, denoted as an adjacency

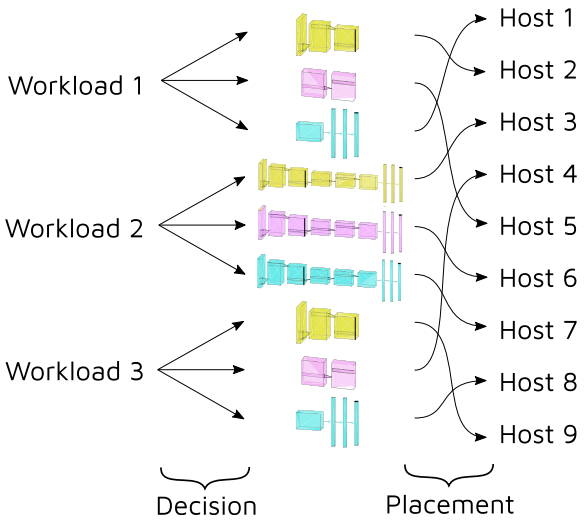


Fig. 4. SplitPlace decision and placement problems.

matrix $P_t : C_t \times \mathcal{H}$. This is realized as a container allocation for new tasks and migration for active tasks in the system.

The main idea behind the layer-wise split design is first to divide neural networks into multiple independent splits, classify these splits in preliminary, intermediate and final neural network layers and distribute them across different nodes based on the node capabilities and network hierarchy. This exploits the fact that communication across edge nodes in the LAN with few hop distances is very fast and has low latency and jitter [51]. Moreover, techniques like knowledge distillation can be further utilized to enhance the accuracy of the results obtained by passing the input through these different classifiers. However, knowledge distillation needs to be applied at the training stage, before generating the neural network splits. As there are many inputs in our assumed large-scale deployment, the execution can be performed in a pipelined fashion to further improve throughput over and above the low response time of the nodes at the edge of the network. For the semantic split, we divide the network weights into a set or a hierarchy of multiple groups that use disjoint sets of features. This is done by making assignment decisions of network parameters to edge devices at deployment time. This produces a tree-structured network that involves no connection between branched sub-trees of semantically disparate class groups. Each sub-group is then allocated to an edge node. The input is either broadcasted from the broker or forwarded in a ring-topology to all nodes with the network split corresponding to the input task. We use standard layer [32] and semantic splitting [16] methods as discussed in Section 2.

We now outline the working of the proposed distributed deep learning architecture for edge computing environments. Fig. 3 shows a schematic view of its working. As shown, there is a shared repository of neural network parameters which is distributed by the broker to multiple edge nodes. The layer and semantic splits are realized as Docker container images that are shared by the broker to the worker nodes at the start of each execution trace. The placement of tasks is realized as spinning up a Docker container using the corresponding image on the worker. As the process of sharing container images is a one-time event, transferring all semantic and layer split fragments does not impose a high overhead on network bandwidth at runtime. This sharing of containers for each splitting strategy and dataset type are transferred to the worker nodes is performed at the start of the run. At run-time, only the decision of which split fragment to be used is communicated to the worker nodes, which executes a container from the corresponding image. The placement of task on each worker is based on the resource availability, computation required to be performed in each section and the capabilities of the nodes (obtained by the *Resource Monitor*). For intensive computations with large storage requirements (Gated Recurrent Units or LSTMs) or splits with high dimension size of input/output (typically the final layers), the splits are sent to high-resource edge workers. The management of allocation and migration of neural network splits is done by the *Container Orchestrator*. Other attention based sub-layer extensions can be deployed in either edge or cloud node based on application requirements, node constraints and user demands. Based on the described model assumptions, we now formulate the problem of taking splitting and placement decisions to optimize the QoS parameters. Implementation specific details on how the results of layer-splits are forwarded and outputs of semantic splits combined across edge nodes are given in Section 5.

3.2 Problem Formulation

The aim of the model is to optimize an objective score O_t (to be maximized), which quantifies the QoS parameters of the interval I_t, such as accuracy, SLA violation rate, energy consumption and average response time. This is typically in the form of a convex combination of energy consumption, response time, SLO violation rates, etc. [50], [52]. The constraints in this formulation include the following. First, the container decomposition for a new task $i \in \mathcal{N}_t$ should be based on d^i. Second, containers corresponding to the layer-split decisions $\{C^i | d^i = L\}$ should be scheduled as per the linear chain of precedence constraints. This means that a container later in the neural inference pipeline should be scheduled only after the complete execution of the previous containers in the pipeline. This is because the output of an initial layer in an inference pipeline of a neural network is required before we can schedule a latter layer in the pipeline. Third, the placement matrix $P_t : C_t \times \mathcal{H}$ should adhere to the allocation constraints, i.e., it should not allocate/migrate a container to a worker where the worker does not have sufficient resources available to accommodate the container. Thus, the problem can be formulated as

$$\begin{aligned} \underset{P_t, \mathcal{D}_t}{\text{maximize}} \quad & \sum_{t}^{T} O_t \\ \text{subject to} \quad & \forall\, t, \forall\, i \in \mathcal{N}_t, C^i \text{ containers created} \\ & \text{based on splitting decision } d^i, \\ & \forall\, t, P_t \text{ is feasible}, \\ & \forall\, d^i = L, C^i \text{ follow precedence chain.} \end{aligned} \quad (1)$$

4 SplitPlace Policy

We now describe the SplitPlace decision and placement policy. For the first sub-problem of deciding optimal splitting

strategy, we employ a Multi-Armed Bandit model to dynamically enforce the decision using external reward signals.[1] Our solution for the second sub-problem of split placement uses a reinforcement-learning based approach that specifically utilizes a surrogate model to optimize the placement decision (agnostic to the specific implementation).[2] This two-stage approach is suboptimal since the response time of the splitting decision depends on the placement decision. In case of large variation in terms of the computational resources, it is worth exploring joint optimization of both decisions. However, in our large-scale edge settings, this segregation helps us to make the problem tractable as we describe next.

The motivation behind this segregation is two-fold. First, having a single reinforcement-learning (RL) model that takes both splitting and placement decisions makes the state-space explode exponentially, causing memory bottlenecks in resource-constrained edge devices [55]. Having a simple RL model does not allow it to scale well with several devices in modern IoT settings (see Section 6 with 50 edge devices the and Gillis RL baseline). One of the solutions that we explore in this work is to simplify this complex problem by decomposing it into split decision making and task placement. Second, the response time of an application depends primarily on the splitting choice, layer or semantic, making it a crucial factor for SLA deadline based decision making. To minimize the SLA violation rates we only use the response time based context for our Multi-Armed bandit model. Other parameters like CPU or RAM utilization have high variability in a volatile setting and are not ideal choices for deciding which splitting strategy to opt. Instead, the inference accuracy is another key factor in taking this decision. Thus, SLA violation and inference accuracy are apt objectives for the first sub-problem. Further, the energy consumption and average response time largely depend on the task placement, making them an ideal objective for optimization in the task placement sub-problem.

4.1 Multi-Armed Bandit Decision Module

Multi-Armed Bandit, in short MAB, is a policy formulation where a state-less agent is expected to take one of many decisions with each decision leading to a different reward. The objective of such an agent is to maximize the expected long-term reward [31]. However, in our case, the most important factor to consider when making a decision of whether to use layer or semantic splits for a task is its SLA deadline.

4.1.1 Estimating Response Time of Layer-Splits

The idea behind the proposed SplitPlace approach is to maintain MABs for two different contexts: 1) when SLA is greater than the estimate of the response time for a layer decision, 2) when SLA is less than this estimate. The motivation behind these two contexts is that in case of the SLA deadline being lower than the execution time of layer split, a "layer" decision would be more likely to violate the SLA as result delivery would be after the deadline. However, the exact time it takes to completely execute all containers corresponding to the layer split decision is apriori unknown. Thus, for every application type, we maintain estimates of the response time, i.e, the total time it takes to execute all containers corresponding to this decision.

Let us denote the tasks leaving the system at the end of I_t as E_t. Now, for each task $i \in E_t$, we denote response time and inference performance using r_i and p_i. We denote the layer response time estimate for application $a \in \mathcal{A}$ as R^a. To quickly adapt to non-stationary scenarios, for instance due to the mobility of edge nodes in the system, we update our estimates using new data-points as exponential moving averages using the multiplier $\phi \in [0, 1]$ for the most recent response time observation. Moving averages presents a low computational cost and consequently low latency compared to more sophisticated smoothing functions

$$R^a \leftarrow \phi \cdot r_i + (1 - \phi) \cdot R^a, \ \forall i \in E_t \wedge d^i = L, \ \forall a \in A. \tag{2}$$

Compared to simple moving average, the above equation gives higher weights to the latest response times, allowing the model to quickly respond to recent changes in environment and workload characteristics.

4.1.2 Context Based MAB Model

Now, for any input task $i \in \mathcal{N}_t$, we divide it into two cases: $sla_i \geq R^{a_i}$ and $sla_i < R^{a_i}$. Considering that the response time of a semantic-split decision would likely be lower than the layer-split decision, in the first case both decisions would most likely not lead to an SLA violation (*high SLA* setting). However, in the second case, a layer-split decision would likely lead to an SLA violation but not the semantic-split decision (*low SLA* setting). To tackle the problem for these different contexts, we maintain two independent MAB models denoted as MAB^h and MAB^l. The former represents a MAB model for the high-SLA setting and the latter for the low-SLA setting.

For each context and decision $d \in \{L, S\}$, we define reward metrics as

$$O^{h,d} = \frac{\sum_{i \in E_t} (\mathbb{1}(r_i \leq sla_i) + p_i) \cdot \mathbb{1}(sla_i \geq R^{a_i} \wedge d^i = d)}{2 \cdot \sum_{i \in E_t} \mathbb{1}(sla_i \geq R^{a_i} \wedge d^i = d)}, \tag{3}$$

$$O^{l,d} = \frac{\sum_{i \in E_t} (\mathbb{1}(r_i \leq sla_i) + p_i) \cdot \mathbb{1}(sla_i < R^{a_i} \wedge d^i = d)}{2 \cdot \sum_{i \in E_t} \mathbb{1}(sla_i < R^{a_i} \wedge d^i = d)}. \tag{4}$$

The first term of the numerator, i.e., $\mathbb{1}(r_i \leq sla_i)$ quantifies SLA violation reward (one if not violated and zero otherwise). The second term, i.e., p_i corresponds to the inference accuracy of the task. These two objectives have been motivated at the start of Section 4. Thus, each MAB model gets the reward function for its decisions allowing independent training of the two. The weights of the two metrics, i.e., accuracy and SLA violation can be set by the user to modify

1. Compared to other methods like A/B testing and Hill Climbing search [53], Multi-Armed Bandits allow quick convergence in scenarios when different cases need to be modelled separately, which is the case in our setup. Thus, we use Mult-Armed Bandits for deciding the optimal splitting strategy for an input task.

2. In contrast to Monte Carlo or Evolutionary methods, Reinforcement learning allows placement to be goal-directed, i.e., aims at optimizing QoS using it as a signal, and allows the model to adapt to changing environments [54]. Hence, we use a RL model, specifically using a surrogate model due for its scalability, to decide the optimal task placement of network splits.

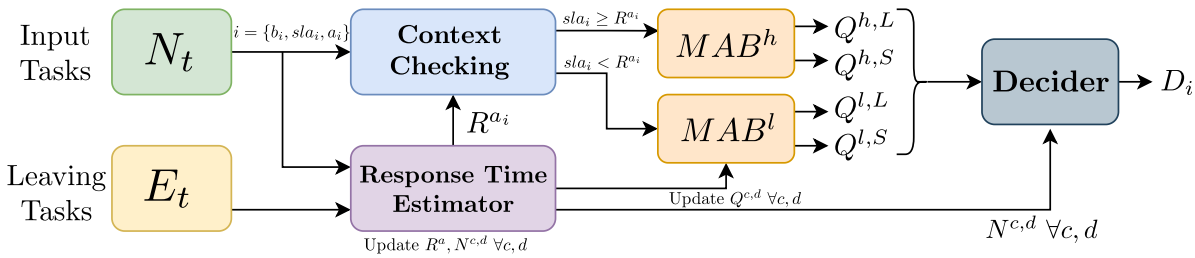


Fig. 5. MAB decision workflow.

the relative importance between the metrics as per application requirements. In our experiments, the weight parameters of both metrics are set to be equal based on grid-search, maximizing the average reward.

Now, for each decision context $c \in \{h, l\}$ and $d \in \{L, S\}$, we maintain a decision count $N^{c,d}$ and a reward estimate $Q^{c,d}$ which is updated using the reward functions $O^{h,d}$ or $O^{l,d}$ as follows:

$$Q^{c,d} \leftarrow Q^{c,d} + \gamma(O^{c,d} - Q^{c,d}),\ \forall d \in \{L, S\},\ \forall c \in \{h, l\}, \tag{5}$$

where γ is the decay parameter. Thus, each reward-estimate is updated by the corresponding reward metric.

Algorithm 1. SplitPlace Decision and Placement Module

Require:
Pre-trained MAB models MAB^h, MAB^l
Discounting factor $\gamma \in (0, 1)$
procedure SplitPlace(scheduling interval I_t)
  Get new tasks $\mathcal{N}_t$
  Get leaving tasks E_t
  Calculate $O^{c,d}$ $\forall c \in \{h, l\}, d \in \{L, S\}$ using (3) and (4)
  Update gain estimates $Q^{c,d}$ $\forall c, d$ using (5) ▷Q update
  Update decision counts $N^{c,d}$ $\forall c, d$ ▷Count update
  for $i \in \mathcal{N}_t$ **do**
    $i = \{b_i, sla_i, a_i\}$
    $d^i \leftarrow \begin{cases} \arg\max_{d \in \{L,S\}} Q^{h,d} + c\sqrt{\frac{\log t}{N^{h,d}}}, & sla_i \geq R^{a_i} \\ \arg\max_{d \in \{L,S\}} Q^{l,d} + c\sqrt{\frac{\log t}{N^{l,d}}}, & sla_i < R^{a_i} \end{cases}$
    ▷UCB based decision
  $\mathcal{D}_t \leftarrow \{d^i\}_{i \in \mathcal{N}_t}$
  Get S_t from resource-monitor ▷Current State
  $P_t \leftarrow \text{DASO}([S_t, P_{t-1}, \mathcal{D}_t])$ ▷Placement Decision
  $O^{MAB} \leftarrow \frac{1}{4}\sum_{c \in \{h,l\}} \sum_{d \in \{L,D\}} O^{c,d}$
  Fine-tune DASO using O^P calculated using (10)
  return P_t

For both these MAB models, we use a parameter-free feedback-based ϵ-greedy learning approach that is known to be versatile in adapting to diverse workload characteristics [56]. Unlike other strategies, this is known to scale asymptotically as the long-term Q estimates become exact under mild conditions [57, Section 2.2]. To train the model, we take the contextual decision

$$d^i = \begin{cases} \begin{cases} \text{random decision}, & \text{with prob. } \epsilon \\ \arg\max_{d \in \{L,S\}} Q^{h,d}, & \text{otherwise} \end{cases}, & sla_i \geq R^{a_i} \\ \begin{cases} \text{random decision}, & \text{with prob. } \epsilon \\ \arg\max_{d \in \{L,S\}} Q^{l,d}, & \text{otherwise} \end{cases}, & sla_i < R^{a_i} \end{cases}. \tag{6}$$

Here, the probability ϵ decays using the reward feedback, starting from 1. We maintain a reward threshold ρ that is initialized as a small positive constant $k < 1$, and use average reward $O^{MAB} = \frac{1}{4}\sum_{c \in \{h,l\}} \sum_{d \in \{L,D\}} O^{c,d}$ to update ϵ and ρ using the rules

$$\epsilon \leftarrow \begin{cases} decay(\epsilon), & O^{MAB} > \rho \\ \epsilon, & \text{otherwise} \end{cases}, \tag{7}$$

$$\rho \leftarrow \begin{cases} increment(\rho), & O^{MAB} > \rho \\ \rho, & \text{otherwise} \end{cases}. \tag{8}$$

Here $decay(\epsilon) = (1 - k) \cdot \epsilon$ and $increment(\rho) = (1 + k) \cdot \rho$. Note that $O^{MAB} > \rho$ refers to the current value of ρ prior to the update. The k value controls the rate of convergence of the model. The ϵ value controls the exploration of the model at training time allowing the model to visit more states and obtain precise estimates of layer-split response times.

However, at test time we already have precise estimates of the response times; thus exploration is only required to adapt in volatile scenarios. For this, ϵ-greedy is not a suitable approach as decreasing ϵ with time would prevent exploration as time progresses. Instead, we use an Upper-Confidence-Bound (UCB) exploration strategy that is more suitable as it takes decision counts also into account [58], [59]. Thus, at test time, we take a deterministic decision using the rule

$$d^i = \begin{cases} \arg\max_{d \in \{L,S\}} Q^{h,d} + c\sqrt{\frac{\log t}{N^{h,d}}}, & sla_i \geq R^{a_i} \\ \text{argmax}_{d \in \{L,S\}} Q^{l,d} + c\sqrt{\frac{\log t}{N^{l,d}}}, & sla_i < R^{a_i} \end{cases}, \tag{9}$$

where t is the scheduling interval count and c is the exploration factor. An overview of the complete split-decision making workflow is shown in Fig. 5. We now discuss the RL based placement module. It is worth noting that both components are independent of each other and can be improved separately in future; however, our experiments show that the MAB decision module accounts for most of the performance gains (see Section 6).

4.2 Reinforcement Learning Based Placement Module

Once we have the splitting decision for each input task $i \in \mathcal{N}_t$, we now can create containers using pre-trained layer and semantic split neural networks corresponding to the application a_i. This can be done offline on a resource rich system, where the split models can be trained using existing datasets. Once we have trained models, we can generate container images corresponding to each split fragment and distribute to all worker nodes [60]. Then we need to place

containers, translating to the worker node initializing a container for the corresponding image, of all active tasks C_t to workers $\mathcal{H}$. To do this, we use a learning model which predicts the placement matrix P_t using the state of the system S_t, decisions $\mathcal{D}_t = d^i \forall i \in \mathcal{T}_t$ and a reward signal O^P. To define O^P, we define the following metrics [50]:

1) *Average Energy Consumption* (AEC) is defined for any interval I_t as the mean energy consumption of all edge workers in the system.
2) *Average Response Time* (ART) is defined for any interval I_t as mean response time (in scheduling intervals) of all leaving tasks E_t.

The choice of these two objectives for the placement subproblem has been motivated at the start of Section 4. Using these metrics, for any interval I_t, O^P is defined as

$$O^P = O^{MAB} - \alpha \cdot AEC_t - \beta \cdot ART_t. \tag{10}$$

Here, α and β (such that $\alpha + \beta = 1$) are hyper-parameters that can be set by users as per the application requirements. Higher α aims to optimize energy consumption at the cost of higher response times, whereas low α aims to reduce average response time. Thus, a RL model f, parameterized by θ takes a decision P_i, where the model uses the reward estimate as the output of the function $f([S_t, P_t, \mathcal{D}_t]; \theta)$, where the parameters θ are updated based on the reward signal O^P. We call this learning approach "decision-aware" as part of the input is the split-decision taken by the MAB model.

Clearly, the proposed formulation is agnostic to the underlying implementation of the learning approach. Thus, any policy like Q-learning or Actor-Critic Learning could be used in the SplitPlace model [57]. However, recently developed techniques like *GOBI* [50] use gradient-based optimization of the reward to quickly converge to a local-maximum of the objective function. GOBI uses a neural-network based surrogate model to estimate the reward from a given input state, which is then used to update the state by calculating the gradients of the reward estimates with respect to the input. Moreover, advances like momentum, annealing and restarts allow such models to quickly reach a global optima [50].

DASO Placement Module. In the proposed framework, we use decision-aware surrogate based optimization method (termed as DASO) to place containers in a distributed mobile edge environment. This is motivated from prior neural network based surrogate optimization methods [50]. Here, we consider a Fully-Connected-Network (FCN) model $f(x; \theta)$ that takes an x as a tuple of input state S_t, split-decision $\mathcal{D}_t$ and placement decision P_t, and outputs an estimate of the QoS objective score O_t. This is because FCNs are agnostic to the structure of the input and hence a suitable choice for modeling dependencies between QoS metrics and model inputs like resource utilization and placement decision [49], [50]. Exploration of other styles of neural models, such as graph neural networks that can take the network topology graph as an input are part of future work. Now, using existing execution trace dataset, $\Lambda = \{[S_t, P_t, \mathcal{D}_t], O_t\}_b$, the FCN model is trained to optimize its network parameters θ such that the Mean-Square-Error (MSE) loss

$$\mathcal{L}(f(x;\theta), y) = \frac{1}{b}\sum_{t=0}^{b}(y - f(x;\theta))^2, \text{ where } (x, y) \in \Lambda. \tag{11}$$

is minimized as in [50]. To do this, we use AdamW optimizer [61] and update θ up till convergence. This allows the surrogate model f to predict an QoS objective score for a given system state S_t, split-decisions $\mathcal{D}_t$ and task placement P_t. Once the surrogate model is trained, starting from the placement decision from the previous interval $P_t = P_{t-1}$, we leverage it to optimize the placement decision using the following rule:

$$P_t \leftarrow P_t - \eta \cdot \nabla_{P_t} f([S_t, P_t, \mathcal{D}_t]; \theta), \tag{12}$$

for a given state and decision pair $S_t, \mathcal{D}_t$. Here, η is the learning rate of the model. The above equation is iterated till convergence, i.e., the $L2$ norm between the placement matrices of two consecutive iterations is lower than a threshold value. Thus, at the start of each interval I_t, using the output of the MAB decision module, the DASO model gives us a placement decision P_t.

4.3 SplitPlace Algorithm

An overview of the SplitPlace approach is given in Algorithm 1. Using pre-trained MAB models, i.e., Q-estimates $Q^{c,d}$ and decision counts $N^{c,d}$, the model decides the optimal splitting decision using the UCB metric (line 9). To adapt the model in non-stationary scenarios, we dynamically update the Q-estimates and decision counts (lines 5 and 6). Using the current state and the split-decisions of all active tasks, we use the DASO approach to take a placement decision for the active containers (line 12). Again, we fine-tune the DASO's surrogate model using the reward metric to adapt to changes in the environment, for instance the changes in the latency of mobile edge nodes and their consequent effect on the reward metrics (line 14). However, the placement decision must conform to the allocation constraints as described in Section 3.2. To relax the constraint of having only feasible placement decisions, in SplitPlace we allocate or migrate only those containers for which it is possible. Those containers that could not be allocated in a scheduling interval are placed to nodes corresponding to the highest output of the neural network f. If no worker placement is feasible the task is added to a wait queue, which are considered again for allocation in the next interval.

5 Implementation

To implement and evaluate the SplitPlace policy, we need a framework that we can use to deploy containerized neural network split fragments on an edge computing environment. One such framework is COSCO [50]. It enables the development and deployment of integrated edge-cloud environments with structured communication and platform independent execution of applications. It connects various IoT sensors, which can be healthcare sensors with gateway devices, to send data and tasks to edge computing nodes, including edge or cloud workers. The resource management and task initiation is undertaken on edge nodes in the broker layer. The framework uses HTTP RESTful APIs for communication and seamlessly integrates a `Flask` based web-

TABLE 3
Edge Worker Characteristics of Azure Edge Environment

Name	Qty	Core count	MIPS	RAM	RAM Bandwidth	Ping time	Network Bandwidth	Disk Bandwidth	Cost Model
Worker Nodes									
B2ms	20	2	4029	4295 MB	372 MB/s	2 ms	1000 MB/s	13.4 MB/s	0.0944 $/hr
E2asv4	10	2	4019	4172 MB	412 MB/s	2 ms	1000 MB/s	10.3 MB/s	0.148 $/hr
B4ms	10	4	8102	7962 MB	360 MB/s	3 ms	2500 MB/s	10.6 MB/s	0.189 $/hr
E4asv4	10	4	7962	7962 MB	476 MB/s	3 ms	2500 MB/s	11.64 MB/s	0.296 $/hr
Broker Node									
L8sv2	1	8	16182	17012 MB	945 MB/s	1 ms	4000 MB/s	17.6 MB/s	0.724 $/hr

environment to deploy and manage containers in a distributed setup [62].

We use only the edge-layer deployment in the framework and use the Docker container engine to containerize and execute the split-neural networks in various edge workers [60]. We uses the Checkpoint/Restore In Userspace (CRIU) [63] tool for container migration. Further, the DASO approach is implemented using the Autograd package in the PyTorch module [64].

To implement SplitPlace in the COSCO framework, we extend the `Framework` class to allow constraints for sequential execution of layer-splits. The function `getPlacementPossible()` was modified to also check for containers of layer-split partitioning scheme to be scheduled sequentially. Moreover, we implemented data transferring pipeline for broadcasting inputs in semantic-split decision and forwarding the outputs in layer-split decision. Finally, the inference outputs were synchronized and brought to the broker to calculate the performance accuracy and measure the workflow response time. For synchronization of outputs and execution of network splits, we use the HTTP Notification API.

6 Performance Evaluation

To test the efficacy of the SplitPlace approach and compare it against the baseline methods, we perform experiments on a heterogeneous edge computing testbed. To do this we emulate a setting with mobile edge devices mounted on self-driving cars, that execute various image-recognition tasks.

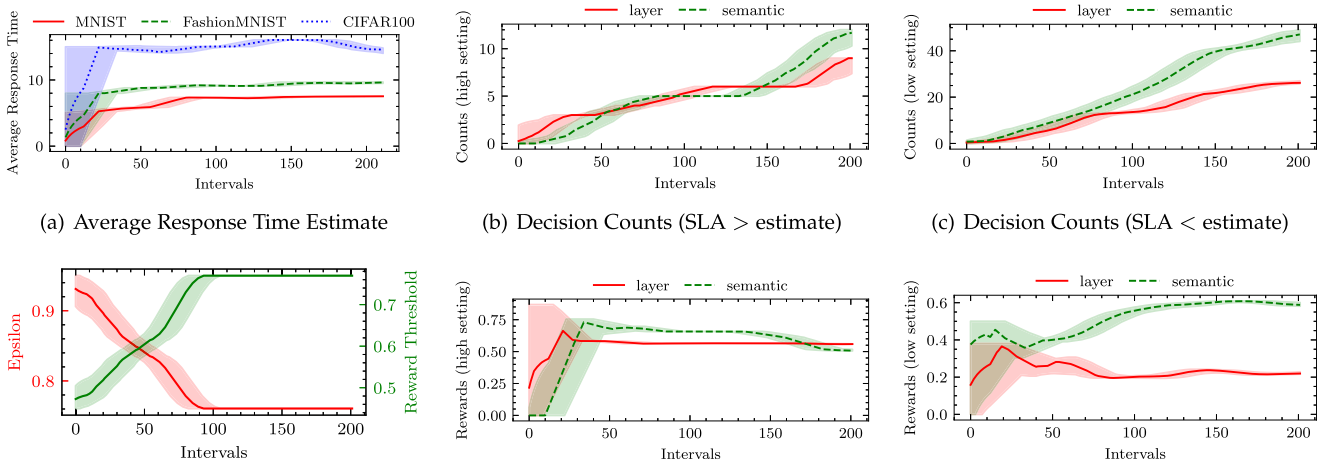
6.1 Experiment Setup

As in prior work [49], [50], [65], we use $\alpha = \beta = 0.5$ in (10) for our experiments (we consider other value pairs in Appendix A.2, available in the online supplemental material). Also, we use the exploration factor $c = 0.5$ for the UCB exploration and the exponential moving average parameter $\phi = 0.9$, chosen using grid-search using the cumulative reward as the metric to maximize. We create a testbed of 50 resource-constrained VMs located in the same geographical location of London, United Kingdom using Microsoft Azure. The worker resources are shown in Table 3. All machines use Intel i3 2.4 GHz processor cores with processing capacity of no more than a Raspberry Pi 4B device. To keep storage costs consistent, we keep Azure P15 Managed disk with 125 MB/s disk throughput and 256 GB size.[3] The worker nodes have 4-8 GB of RAM, whereas the broker has 16 GB RAM. To factor in the mobility of the edge nodes, we use the `NetLimiter` tool to tweak the communication latency with the broker node using the mobility model described in [66]. Specifically, we use the latency and bandwidth parameters of workers from the traces generated using the Simulation of Urban Mobility (SUMO) tool [67] that emulates mobile vehicles in a city like environment. SUMO gives us the parameters like `ping` time and network bandwidth to simulate in our testbed using `NetLimiter`. The moving averages and periodic fine-tuning allow our approach to be robust towards any kind of dynamism in the edge environment, including the one arising from mobility of worker nodes.

Our Azure environment is such that all devices are in the same LAN with 10 MBps network interface cards to avoid network bottlenecks while transferring inputs, outputs and intermediate results across neural network splits. Even if the nodes are in the same LAN with high bandwidth connections, the SUMO model would emulate the affects of mobility as is common in prior work on mobile edge computing [65], [68]. Further, we use the `cPickle`[4] Python module to save the intermediate results using `bzip2` compression and `rsync`[5] file-transfer utility to minimize the communication latency. For containers corresponding to a layer-split workload that are deployed in different nodes, the intermediate results are forwarded using the `scp` utility to the next container in the neural network pipeline. Similarly, for semantic splitting, the `cPickle` outputs are collected using `rsync` and concatenated using the `torch.cat` function.

We use the Microsoft Azure pricing calculator to obtain the cost of execution per hour (in US Dollars).[6] The power consumption models are taken from the Standard Performance Evaluation Corporation (SPEC) benchmarks repository.[7] The Million-Instruction-per-Second (MIPS) of all VMs are computed using the `perf-stat`[8] tool on the SPEC

3. Azure Managed Disks `https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types#premium-ssd`.
4. `cPickle` module `https://docs.python.org/2/library/pickle.html#module-cPickle`.
5. `rsync` tool `https://linux.die.net/man/1/rsync`.
6. Microsoft Azure pricing calculator for South UK `https://azure.microsoft.com/en-gb/pricing/calculator/`.
7. SPEC benchmark repository `https://www.spec.org/cloud_iaas2018/results/`.
8. `perf-stat` tool `https://man7.org/linux/man-pages/man1/perf-stat.1.html`.



(a) Average Response Time Estimate (b) Decision Counts (SLA > estimate) (c) Decision Counts (SLA < estimate)

(d) Training Parameters with time (e) Average Rewards (SLA > estimate) (f) Average Rewards (SLA < estimate)

Fig. 6. MAB training curves.

benchmarks. We run all experiments for 100 scheduling intervals, i.e., $\Gamma = 100$, with each interval being 300 seconds long, giving a total experiment time of 8 hours 20 minutes. We average over five runs and use diverse workload types to ensure statistical significance in our experiments. We consider variations of the experimental setup in Appendix A.3, available in the online supplemental material.

6.2 Workloads

Motivated from prior work [32], we use three families of popular DNNs as the benchmarking models: ResNet50-V2 [69], MobileNetV2 [70] and InceptionV3 [71]. Each family has many variants of the model, each having a different number of layers in the neural model. For instance, the ResNet model has 34 and 50 layers. We use three image-classification data sets: MNIST, FashionMNIST and CIFAR100 [33], [34], [35]. MNIST is a hand-written digit recognition dataset with 28×28 gray-scale images to 10-dimensional output. FashionMNIST has 28×28 RGB images with 10 dimensional output. CIFAR100 has 32×32 RGB images with 100-dimensional output. Thus the application set $\mathcal{A}$ becomes {MNIST, FashionMNIST, CIFAR100}. These models have been taken directly from the AIoTBench workloads [72]. This is a popular suite of AI benchmark applications for IoT and Edge computing solutions. The three specific datasets used in our experiments are motivated from the vertical use case of self-driving cars, which requires DNN-based applications to continuously recognize images with low latency requirements. Herein, an image recognition software is deployed that reads speed signs (digit recognition, MNIST), recognizes humans (through apparel and pose [73], FashionMNIST), identifies other objects like cars and barriers (object detection, CIFAR100). We use the implementation of neural network splitting from prior work [16], [32].

We use the inference deadline from the work [32] as our SLA. To create the input tasks, we use batch sizes sampled uniformly from $16,000 - 64,000$. At the beginning of each scheduling interval, we create $Poisson(\lambda)$ tasks with $\lambda = 6$ tasks for our setup, sampled uniformly from one of the three applications [50]. We consider other λ values and single workload type (from MNIST, FashionMNIST and CIFAR100) in Appendices A.1 and A.4, available in the online supplemental material. The split fragments for MNIST, FashionMNIST and CIFAR100 lead to container images of sizes 8-14 MB, 34-56 MB and 47-76 MB, respectively. To calculate the inference accuracy to feed in the MAB models and perform UCB exploration, we also share the ground-truth labels of all datasets with all worker nodes at the time of sharing the neural models as Docker container images. We also compare edge and cloud setups in Appendix A.5, available in the online supplemental material, to establish the need for edge devices for latency critical workloads.

6.3 MAB Training

To train our MAB models, we execute the workloads on the test setup for 200 intervals and use feedback-based ϵ-greedy exploration to update the layer-split decision response time estimates, Q-estimates and decision counts. Fig. 6 shows the training curves for the two models.

Fig. 6a shows how the response time estimates for the layer-split decision are learned starting from zero using moving averages. Fig. 6d shows how the reward-threshold ρ and decay parameter ϵ change with time. We use the decay and increment multipliers as 0.9 and 1.1 ($k = 0.1$ in (7)) for ϵ and ρ respectively, as done in [56]. Figs. 6b and 6c show the decision counts for high and low SLA settings for both decisions. Figs. 6e and 6f show the Q-estimates for high and low SLA settings. The dichotomy between the two settings is reflected here. When the sla_i of the input task i is less than the estimate R^{a_i} (low setting) there is a clear distinction between the rewards of the two decisions as layer-split is likely to lead to SLA violation and hence lower rewards. However, when sla_i is greater than the estimate R^{a_i} (high setting), both decisions give relatively high rewards with layer-split decision slightly surpassing the semantic-split due to higher average accuracy as discussed in Section 2.

The feedback-based ϵ-greedy training allows us to obtain close estimates of the average response times of the layer-split executions for each application type and average rewards for both decisions in high and low SLA settings. Thus, in our experiments, we initialize the expected reward (Q) and layer-split response time (R) estimates by the values we get from this training approach. At test time, we dynamically update these estimates using (2) and (5).

6.4 Performance Metrics

We use the following evaluation metrics in our experiments as motivated from prior works [2], [49], [50]. We also use AEC and ART as discussed in Section 4.

1) *Average Accuracy* is defined for an execution trace as the average accuracy of all tasks run in an experiment, i.e.,

$$Accuracy = \frac{\sum_t \sum_{i \in E_t} p_i}{\sum_t |E_t|}. \tag{13}$$

2) *Fraction of SLA violation* is defined for an execution trace as the fraction of all tasks run in an experiment for which the response time is higher than the SLA deadline, i.e.,

$$SLA\ Violations = \frac{\sum_t \sum_{t \in E_t} \mathbb{1}(sla_i \geq r_i)}{\sum_t |E_t|}. \tag{14}$$

3) *Average Reward* is defined for an execution trace as follows:

$$Reward = \frac{\sum_t \sum_{t \in E_t} \mathbb{1}(sla_i \geq r_i) + p_i}{2 \cdot \sum_t |E_t|}. \tag{15}$$

4) *Execution Cost* is defined for an execution trace as the total cost incurred during the experiment, i.e.,

$$Cost = \sum_{h \in \mathcal{H}} \int_x C_h(x) dx. \tag{16}$$

where $C_h(x)$ is the cost function for worker h with time.

5) *Average Wait Time* is the average time a task had to wait in the wait queue till it could be allocated to a worker for execution.
6) *Average Execution Time* is the response time minus the wait time, averaged for all tasks run in an experiment.
7) *Fairness* is defined as the Jain's fairness index for execution on tasks over the edge workers [50].

6.5 Baselines and Ablated Models

We compare the performance of the SplitPlace approach against the state-of-the-art baselines *Gillis* and BottleNet++ Model Compression (denoted as *MC* in our graphs) [32], [37], [38]. Gillis refers to the reinforcement learning method proposed in [32] that leverages both layer-splitting and compression models to achieve optimal response time and inference accuracy. Note that contrary to the original Gillis' work, our implementation does not leverage serverless functions. MC is a model-compression approach motivated from BottleNet++ that we implement using the `PyTorch Prune` library.[9] Further details in Section 2. We do not include results for other methods discussed in Section 2 as MC and Gillis give better results empirically for all comparison metrics. We also compare SplitPlace with ablated models, where we replace one or both of the MAB or DASO components with simpler versions as described below.

- *Semantic+GOBI (S+G)*: Semantic-split decision only with vanilla GOBI placement module.
- *Layer+GOBI (L+G)*: Layer-split decision only with vanilla GOBI placement module.
- *Random+DASO (R+D)*: Random split decision with DASO placement module.
- *MAB+GOBI (M+G)*: MAB based split decider with vanilla GOBI placement module.

The final SplitPlace approach is represented as MAB +DASO or M+D in shorthand notation in the graphs. These ablated baselines help us determine the relative improvements in performance by the two components of MAB and DASO separately.

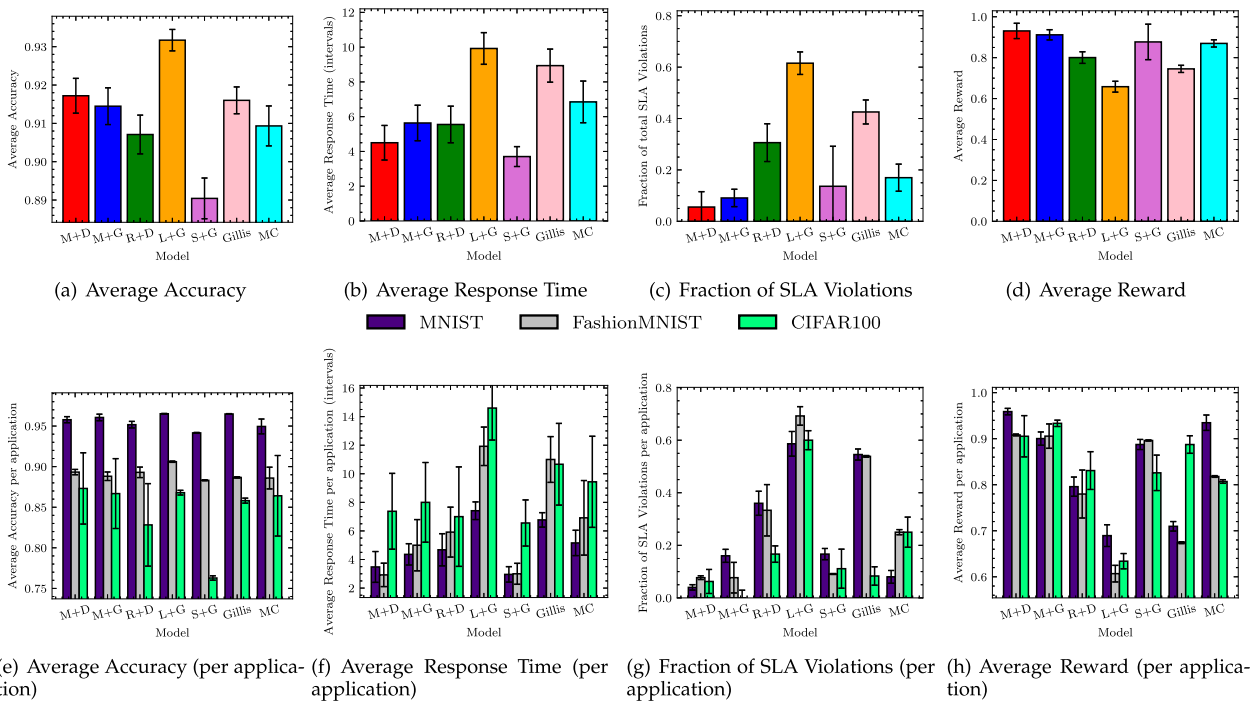
6.6 Results and Ablation Analysis

We now provide comparative results showing the performance of the proposed SplitPlace approach against the baseline models and argue the importance of the MAB and decision-aware placement using ablation analysis. We train the *GOBI* and *DASO* models using the execution trace dataset used to train the MAB models. The learning rate (η) was set to 10^{-3} from [50].

Fig. 7 shows the average reward and related performance metrics, i.e., accuracy, response time and SLA violation rate. As expected, the L+G policy gives the highest accuracy of 93.17% as all decisions are layer-wise only with a higher inference performance than semantic-split execution. The S+G policy gives the least accuracy of 89.04%. However, due to layer-splits only the L+G policy also has the highest average response time, subsequently giving the highest SLA violation rate. On the other hand, S+G policy has the least average response time. However, due to the intelligent decision making in SplitPlace, it is able to get the highest total reward of 0.9418. Similar trends are also seen when comparing across models for each application. The accuracy is the highest for the MNIST dataset and lowest for CIFAR100. Average response time is highest for the CIFAR100 and lowest for MNIST in general. Among the baselines, the Gillis approach has the lowest SLA violation rate of 22% and SplitPlace improves upon this by giving 14% lower SLA violations (only 8%). Gillis has higher accuracy between the baselines of 91.9%, with SplitPlace giving an average improvement of 0.82%. Overall, the total reward of SplitPlace is higher than the baselines by at least 10.1%, giving the reward of 94.18%.

Fig. 8 shows the performance of all models for other evaluation metrics like energy, execution time and fairness. Compared to the baselines, SplitPlace can reduce energy consumption by up to $4.41\% - 5.03\%$ giving an average energy consumption of 1.0867 MW-hr. However, the SplitPlace approach has higher scheduling time and lower fairness index (Table 4). The Gillis baseline has the highest fairness index of 0.89, however this index for SplitPlace is 0.73. SplitPlace has a higher overhead of 11.8% compared to the Gillis baseline in terms of scheduling time. Fig. 8i compares the average execution cost (in USD) for all models. As SplitPlace is able to run the maximum number of containers in the 100 intervals, it has the least cost of 3.07 USD/container. The main advantage of SplitPlace is the intelligent

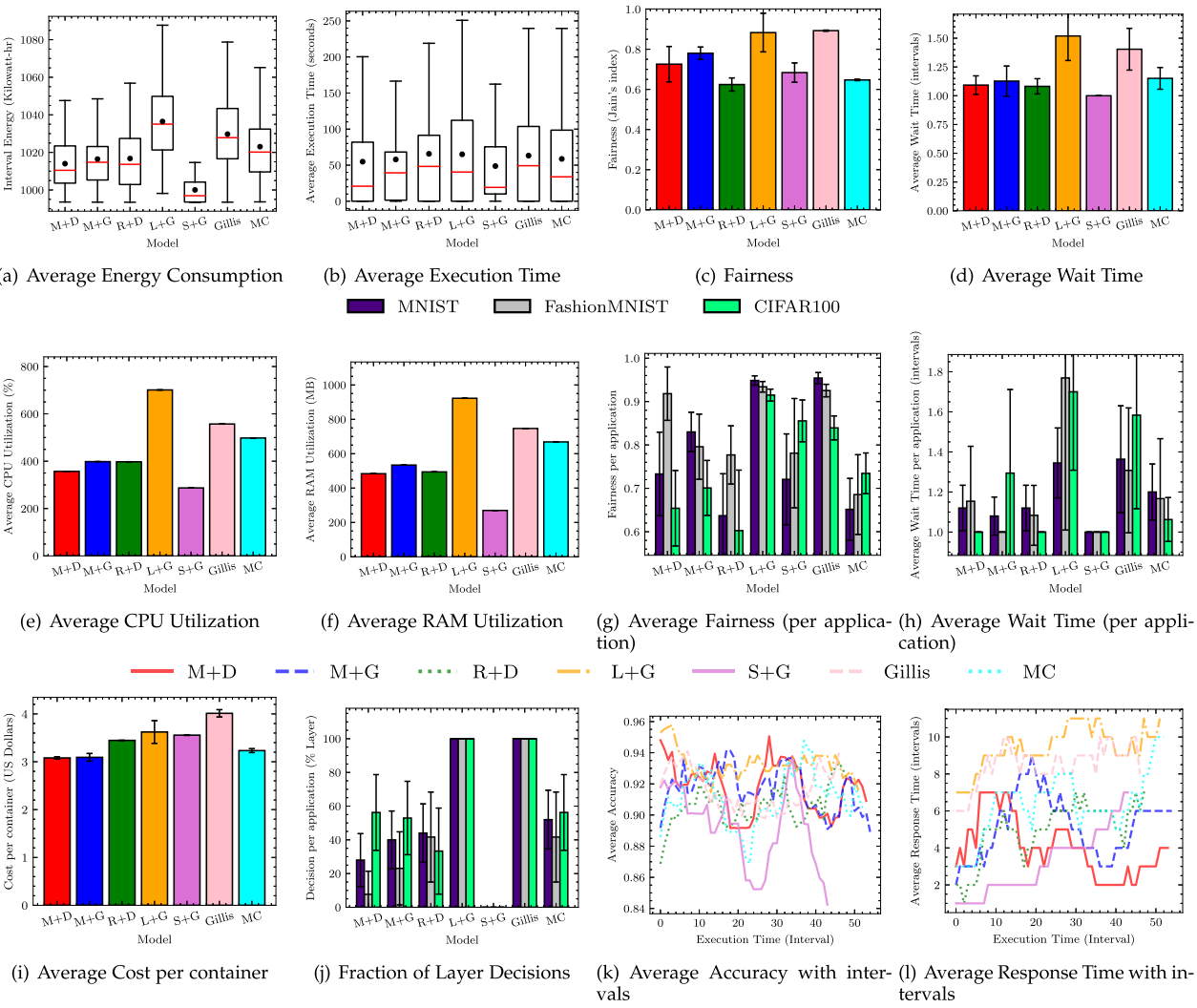
9. PyTorch Prune. `https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.ln_structured.html`. Accessed 10 October 2021.



(a) Average Accuracy (b) Average Response Time (c) Fraction of SLA Violations (d) Average Reward

(e) Average Accuracy (per application) (f) Average Response Time (per application) (g) Fraction of SLA Violations (per application) (h) Average Reward (per application)

Fig. 7. Comparison of SplitPlace against baselines and ablated models on physical setup with 50 edge workers.



(a) Average Energy Consumption (b) Average Execution Time (c) Fairness (d) Average Wait Time

(e) Average CPU Utilization (f) Average RAM Utilization (g) Average Fairness (per application) (h) Average Wait Time (per application)

(i) Average Cost per container (j) Fraction of Layer Decisions (k) Average Accuracy with intervals (l) Average Response Time with intervals

Fig. 8. Additional results comparing SplitPlace with baselines and ablated models.

TABLE 4
Comparison of SplitPlace With Baseline and Ablated Models

Model	Energy	Scheduling Time	Fairness	Wait Time	Response Time	SLA Violations	Accuracy	Average Reward
Baselines								
Model Compression	1.1368	8.84±0.02	0.65±0.01	1.15±0.09	6.85±1.20	0.26±0.02	89.93	83.98
Gillis	1.1442	**8.22±0.01**	**0.89±0.03**	1.40±0.18	8.39±0.95	0.22±0.03	91.90	84.17
Ablation								
Semantic+GOBI	1.1112	8.68±0.03	0.68±0.04	1.08±0.00	**3.70±0.57**	0.14±0.04	89.04	83.91
Layer+GOBI	1.1517	8.72±0.01	0.88±0.03	1.52±0.21	9.92±0.91	0.62±0.07	**93.17**	64.87
Random+DASO	1.1297	8.86±0.01	0.62±0.05	**1.00±0.07**	5.55±1.05	0.29±0.09	90.71	81.62
MAB+GOBI	1.1290	9.12±0.02	0.78±0.08	1.13±0.13	5.64±1.02	0.10±0.03	91.45	90.18
SplitPlace Model								
MAB+DASO	**1.0867**	9.32±0.02	0.73±0.01	1.09±0.08	4.50±1.00	**0.08±0.02**	92.72	**94.18**

The best achieved value for each metric is shown in bold. Units: Energy (MW-hr), Scheduling Time (seconds), Fairness (Jain's Index), Wait Time (Intervals), Response Time (Intervals).

splitting decisions facilitate overcoming the memory bottlenecks in edge environments, giving up to 32% lower RAM utilization compared to Gillis and Model Compression.

In terms of the initial communication time of the Docker container images, the SplitPlace method takes 30 seconds at the start of an execution. Gillis and MC have such communication times of 20 and 18 seconds, respectively. This demonstrates that SplitPlace has a low one-time overhead (up to 12 seconds) compared to the baselines when compared to the gains in response time (up to 46%) that linearly scales as the number of workloads increase.

A summary of comparisons with values of main performance metrics for all models is given in Table 4. The best values achieved for each metric are highlighted in bold.

7 Conclusion

In this work, we present SplitPlace, a novel framework for efficiently managing demanding neural network based applications. SplitPlace exploits the trade-off between layer and semantic split models where the former gives higher accuracy, but the latter gives much lower response times. This allows SplitPlace to not only manage tasks to maintain high inference accuracy on average, but also reduce SLA violation rate. The proposed model uses a Multi-Armed-Bandits based policy to decide which split strategy to use according to the SLA deadline of the incoming task. Moreover, it uses a decision-aware learning model to take appropriate placement decisions for those neural fragments on mobile edge workers. Further, both MAB and learning models are dynamically tuned to adapt to volatile scenarios. All these contributions allow SplitPlace to out-perform the baseline models in terms of average response time, SLA violation rate, inference accuracy and total reward by up to 46.3%, 69.2%, 3.1% and 12.1% respectively in a heterogeneous edge environment with real-world workloads.

We propose the following future directions for this work. An extension of the current work may be developed that dynamically updates the splitting configuration to adapt to more heterogeneous and non-stationary edge environments [74]. Moreover, the current model assumes that all neural models are divisible into independent layers. This may be hard for deep learning models like attention based neural networks or transformer models [75]. Finally, the model only considers splits and their placement as containers, more fine-grained methods involving Neural Architecture Search and cost efficient deployment methods may be explored like serverless frameworks [76]. Other considerations such as privacy concerns and non-stationary number of active edge nodes with extreme levels of heterogeneity such that the placement decision has a significant impact on response time is also part of future work.

Software Availability

The code is available at `https://github.com/imperial-qore/SplitPlace`. The Docker images used in the experiments are available at `https://hub.docker.com/u/shreshthtuli`.

Acknowledgments

Shreshth Tuli was grateful to the Imperial College London for funding his Ph.D. through the President's Ph.D. Scholarship scheme. We thank Feng Yan for helpful discussions. A preliminary version of this work was presented at the Student Research Competition in ACM SIGMETRICS Conference 2021 [1].

References

[1] S. Tuli, "SplitPlace: Intelligent placement of split neural nets in mobile edge environments," *SIGMETRICS Perform. Eval. Rev.*, vol. 49, pp. 63–65, 2021.
[2] S. S. Gill *et al.*, "Transformative effects of IoT, blockchain and artificial intelligence on cloud computing: Evolution, vision, trends and open challenges," *Internet Things*, vol. 8, pp. 100–118, 2019.
[3] H. Zhu *et al.*, "Benchmarking and analyzing deep neural network training," in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 88–100.
[4] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.
[5] A. Khanna, A. Sah, and T. Choudhury, "Intelligent mobile edge computing: A deep learning based approach," in *Proc. Int. Conf. Adv. Comput. Data Sci.*, 2020, pp. 107–116.
[6] F. A. Kraemer, A. E. Braten, N. Tamkittikhun, and D. Palma, "Fog computing in healthcare–A review and discussion," *IEEE Access*, vol. 5, pp. 9206–9222, 2017.

[7] L. Zhang and L. Zhang, "Deep learning-based classification and reconstruction of residential scenes from large-scale point clouds," *IEEE Trans. Geosci. Remote Sens.*, vol. 56, no. 4, pp. 1887–1897, Apr. 2018.

[8] M. Roopaei, P. Rad, and M. Jamshidi, "Deep learning control for complex and large scale cloud systems," *Intell. Autom. Soft Comput.*, vol. 23, no. 3, pp. 389–391, 2017.

[9] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, M. T. Kandemir, and C. R. Das, "Implications of public cloud resource heterogeneity for inference serving," in *Proc. 6th Int. Workshop Serverless Comput.*, 2020, pp. 7–12.

[10] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: Synergistic progressive inference of neural networks over device and cloud," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–15.

[11] Q. Liang, P. Shenoy, and D. Irwin, "AI on the edge: Characterizing AI-based IoT applications using specialized edge architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2020, pp. 145–156.

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

[13] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.

[14] J. Shao and J. Zhang, "Communication-computation trade-off in resource-constrained edge inference," *IEEE Commun. Mag.*, vol. 58, no. 12, pp. 20–26, Jan. 2021.

[15] J. Liu, M. L. Curry, C. Maltzahn, and P. Kufeldt, "Scale-out edge storage systems with embedded storage nodes to get better availability and cost-efficiency at the same time," in *Proc. 3rd USENIX Workshop Hot Topics Edge Comput.*, 2020, pp. 1–7.

[16] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "SplitNet: Learning to semantically split deep networks for parameter reduction and model parallelization," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1866–1874.

[17] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief, "Communication-efficient edge AI: Algorithms and systems," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 4, pp. 2167–2191, Oct.–Dec. 2020.

[18] W. Y. B. Lim *et al.*, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 3, pp. 2031–2063, Jul.–Sep. 2020.

[19] Y. Siriwardhana, P. Porambage, M. Liyanage, and M. Ylianttila, "A survey on mobile augmented reality with 5G mobile edge computing: Architectures, applications, and technical aspects," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 2, pp. 1160–1192, Apr.–Jun. 2021.

[20] S. Tuli *et al.*, "HealthFog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated IoT and fog computing environments," *Future Gener. Comput. Syst.*, vol. 104, pp. 187–200, 2020.

[21] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 595–610.

[22] A. Gujarati *et al.*, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation*, 2020, pp. 443–462.

[23] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.

[24] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Trans. Circuits Syst., II, Exp. Briefs*, vol. 67, no. 5, pp. 871–875, May 2020.

[25] J. Huang, C. Samplawski, D. Ganesan, B. Marlin, and H. Kwon, "CLIO: Enabling automatic compilation of deep learning pipelines across IoT and cloud," in *Proc. 26th Annu. Int. Conf. Mobile Comput. Netw.*, 2020, pp. 1–12.

[26] Q. Le, L. Miralles-Pechuán, S. Kulkarni, J. Su, and O. Boydell, "An overview of deep learning in industry," in *Data Anal. AI*. Boca Raton, FL, USA: CRC, 2020, pp. 65–98.

[27] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proc. Workshop Hot Topics Video Anal. Intell. Edges*, 2019, pp. 21–26.

[28] Y. A. Ushakov, P. N. Polezhaev, A. E. Shukhman, M. V. Ushakova, and M. V. Nadezhda, "Split neural networks for mobile devices," in *Proc. 26th Telecommun. Forum*, 2018, pp. 420–425.

[29] V. S. Gordon and J. Crouson, "Self-splitting modular neural network-domain partitioning at boundaries of trained regions," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2008, pp. 1085–1091.

[30] E. Ahmed and M. H. Rehmani, "Mobile edge computing: Opportunities, solutions, and challenges," *Future Gener. Comput. Syst.*, vol. 70, pp. 59–63, 2017.

[31] D. Bouneffouf, I. Rish, and C. Aggarwal, "Survey on applications of multi-armed and contextual bandits," in *Proc. IEEE Congr. Evol. Comput.*, 2020, pp. 1–8.

[32] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 138–148.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[34] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," 2017, *arXiv: 1708.07747*.

[35] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," M.S. thesis, Univ. Toronto, Toronto, ON, Canada, 2009.

[36] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee, "Migrating from monolithic to serverless: A FinTech case study," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2020, pp. 20–25.

[37] A. E. Eshratifar, A. Esmaili, and M. Pedram, "BottleNet: A deep learning architecture for intelligent mobile cloud computing services," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des.*, 2019, pp. 1–6.

[38] J. Shao and J. Zhang, "BottleNet++: An end-to-end approach for feature compression in device-edge co-inference systems," in *Proc. IEEE Int. Conf. Commun. Workshops*, 2020, pp. 1–6.

[39] S. Gao, F. Huang, J. Pei, and H. Huang, "Discrete model compression with resource constraint for deep neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1896–1905.

[40] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 328–339.

[41] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[42] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "DeepSlicing: Collaborative and adaptive CNN inference with low latency," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2175–2187, Sep. 2021.

[43] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[44] P. S. Chandakkar, Y. Li, P. L. K. Ding, and B. Li, "Strategies for retraining a pruned neural network in an edge computing paradigm," in *Proc. IEEE Int. Conf. Edge Comput.*, 2017, pp. 244–247.

[45] L. Wang *et al.*, "Context-aware deep model compression for edge cloud computing," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 787–797.

[46] F. Yu, L. Cui, P. Wang, C. Han, R. Huang, and X. Huang, "EasiEdge: A novel global deep neural networks pruning method for efficient edge computing," *IEEE Internet Things J.*, vol. 8, no. 3, pp. 1259–1271, Feb. 2021.

[47] A. Kaplunovich and Y. Yesha, "Automatic tuning of hyperparameters for neural networks in serverless cloud," in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 2751–2756.

[48] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "FogBus: A blockchain-based lightweight framework for edge and fog computing," *J. Syst. Softw.*, vol. 154, pp. 22–36, 2019.

[49] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan, "Learn-as-you-go with Megh: Efficient live migration of virtual machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1786–1801, Aug. 2019.

[50] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 101–116, Jan. 2022.

[51] S.-H. Park, O. Simeone, and S. Shamai, "Joint cloud and edge processing for latency minimization in fog radio access networks," in *Proc. IEEE 17th Int. Workshop Signal Process. Adv. Wireless Commun.*, 2016, pp. 1–5.

[52] R. A. C. da Silva and N. L. S. da Fonseca, "Resource allocation mechanism for a fog-cloud infrastructure," in *Proc. IEEE Int. Conf. Commun.*, 2018, pp. 1–6.
[53] R. Kohavi and R. Longbotham, "Online controlled experiments and A/B testing," *Encyclopedia Mach. Learn. Data Mining*, vol. 7, no. 8, pp. 922–929, 2017.
[54] M. Wiering and M. Van Otterlo, "Reinforcement learning," *Adapt. Learn. Optim.*, vol. 12, no. 3, 2012, Art. no. 729.
[55] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.
[56] A. Maroti, "RBED: Reward based epsilon decay," 2019, *arXiv: 1910.13701*.
[57] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[58] Y. Zhang, P. Cai, C. Pan, and S. Zhang, "Multi-agent deep reinforcement learning-based cooperative spectrum sensing with upper confidence bound exploration," *IEEE Access*, vol. 7, pp. 118 898–118 906, 2019.
[59] S. Gupta, G. Joshi, and O. Yağan, "Correlated multi-armed bandits with a latent random source," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2020, pp. 3572–3576.
[60] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proc. IEEE Int. Conf. Edge Comput.*, 2018, pp. 1–8.
[61] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–19.
[62] M. Grinberg, *Flask Web Development: Developing Web Applications With Python*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018.
[63] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory CRIU for docker containers," in *Proc. Int. Symp. Memory Syst.*, 2019, pp. 53–65.
[64] A. Paszke *et al.*, "Automatic differentiation in pytorch," in *Proc. Int. Conf. Neural Inf. Process. Syst. 2017 Workshop Autodiff*, 2017.
[65] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using A3C learning and residual recurrent neural networks," *IEEE Trans. Mobile Comput.*, vol. 21, no. 3, pp. 940–954, Mar. 2022.
[66] K. Gilly, S. Alcaraz, N. Aknin, S. Filiposka, and A. Mishev, "Modelling edge computing in urban mobility simulation scenarios," in *Proc. IFIP Netw. Conf.*, 2020, pp. 539–543.
[67] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO-simulation of urban mobility," *Int. J. Adv. Syst. Meas.*, vol. 5, no. 3/4, 2012, pp. 128–138.
[68] C. Wang, Y. Xu, J. Zhang, and B. Ran, "Integrated traffic control for freeway recurrent bottleneck based on deep reinforcement learning," *IEEE Trans. Intell. Transp. Syst.*, early access, pp. 1–14, Jan. 20, 2022, doi: 10.1109/TITS.2022.3141730.
[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[70] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.
[71] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. AAAI Conf. Artif. Intell.*, 2017, pp. 4278–4284.
[72] C. Luo *et al.*, "AIoT bench: Towards comprehensive benchmarking mobile and embedded device intelligence," in *Proc. Int. Symp. Benchmarking Measuring Optim.*, 2018, pp. 31–35.
[73] P. R. Tupe, P. M. Vibhute, and M. A. Sayyad, "An architecture combining convolutional neural network (CNN) with batch normalization for apparel image classification," in *Proc. IEEE Int. Symp. Sustain. Energy, Signal Process. Cyber Secur.*, 2020, pp. 1–6.
[74] K. A. Bonawitz *et al.*, "Towards federated learning at scale: System design," in *Proc. Mach. Learn. Syst.*, 2019, pp. 374–388.
[75] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[76] G. Casale *et al.*, "RADON: Rational decomposition and orchestration for serverless computing," *SICS Softw.-Intensive Cyber-Phys. Syst.*, vol. 35, no. 1, pp. 77–87, 2020.

Shreshth Tuli received the undergraduate degree from the Department of Computer Science and Engineering, Indian Institute of Technology - Delhi, India. He is a president's PhD scholar with the Department of Computing, Imperial College London, U.K. He has worked as a visiting research fellow with the CLOUDS Laboratory, School of Computing and Information Systems, University of Melbourne, Australia. He is a national level Kishore Vaigyanik Protsahan Yojana (KVPY) scholarship holder from the Government of India for excellence in science and innovation. His research interests include fog computing and deep learning.

Giuliano Casale joined the Department of Computing, Imperial College London, in 2010, where he is currently a reader. Previously, he worked as a research scientist and consultant in the capacity planning industry. He teaches and does research in performance engineering and cloud computing, topics on which he has published more than 100 refereed papers. He has served on the technical program committee of more than 80 conferences and workshops and as co-chair for several conferences in the area of performance and reliability engineering, such as the *ACM SIGMETRICS/Performance* and IEEE/IFIP DSN. His research work has received multiple awards, recently the Best Paper Award at ACM SIGMETRICS. He serves on the editorial boards of the *IEEE Transactions on Network and Service Management* and *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* and as current chair of ACM SIGMETRICS.

Nicholas R. Jennings is the vice-chancellor and president of Loughborough University. He is an internationally-recognised authority in the areas of AI, autonomous systems, cyber-security, and agent-based computing. He is a member of the UK government's AI Council, the governing body of the Engineering and Physical Sciences Research Council, and chair of the Royal Academy of Engineering's Policy Committee. Before Loughborough, he was the vice-provost for research and enterprise and professor of artificial intelligence with Imperial College London, the UK's first regius professor of computer science (a post bestowed by the monarch to recognise exceptionally high quality research) and the UK Government's first chief scientific advisor for national security.