Murmuration: On-the-fly DNN Adaptation for SLO-Aware Distributed Inference in Dynamic Edge Environments

Jieyu Lin
University of Toronto
Toronto, Ontario, Canada
jieyu.lin@mail.utoronto.ca

Minghao Li
University of Toronto
Toronto, Ontario, Canada
mingh.li@mail.utoronto.ca

Sai Qian Zhang
New York University
New York, New York, USA
sai.zhang@nyu.edu

Alberto Leon-Garcia
University of Toronto
Toronto, Ontario, Canada
alberto.leongarcia@utoronto.ca

ABSTRACT

The proliferation of Virtual and Augmented Reality (VR/AR) and the Internet of Things (IoT) applications is driving the demand for efficient Deep Neural Network (DNN) inference at the edge. These applications often impose stringent Service Level Objectives (SLOs), such as latency or accuracy, that must be met under the constraints of limited resources and dynamic network conditions. In this study, we explore a novel approach to DNN inference across multiple edge devices, incorporating both model customization and partitioning dynamically, to better align with these constraints and SLOs. Unlike conventional methods that employ a single fixed DNN network, our system, termed *Murmuration*, combines one-shot Neural Architecture Search (NAS) and Reinforcement Learning (RL) to dynamically customize and partition DNN models. This approach adapts in real-time to the capabilities of the edge devices, network conditions, and varying SLO requirements. The design of Murmuration allows it to effectively navigate the large search space defined by DNN models, network delays, and bandwidth, offering a significant improvement in managing trade-offs between accuracy and latency.

We implemented and evaluated Murmuration using a variety of edge devices. The results show that our approach outperforms state-of-the-art methods in terms of inference accuracy by up to 5% or latency by up to 6.7×. With the flexibility of model customization, Murmuration can meet SLO under a wider range of network delays and bandwidths, improving SLO compliance rate by up to 52%.

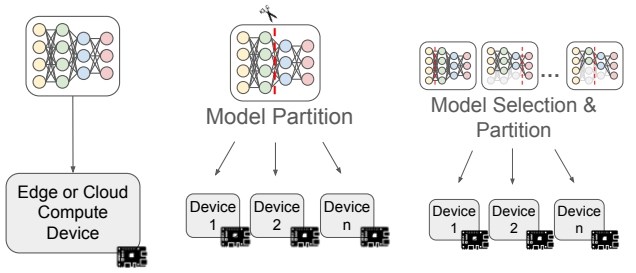
CCS CONCEPTS

• **Computing methodologies** → *Distributed algorithms.*

KEYWORDS

DNN, Distributed Inference, Model Partitioning, Reinforcement Learning, One-shot NAS, Service Level Objective (SLO)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPP '24, August 12–15, 2024, Gotland, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673154



(a) Single Model Single Device Inference (b) Single Model Multi Device Inference (c) Customizable Model Multi Device Inference (**Ours**)

Figure 1: Comparison of different types DNN distributed inference methods

ACM Reference Format:
Jieyu Lin, Minghao Li, Sai Qian Zhang, and Alberto Leon-Garcia. 2024. Murmuration: On-the-fly DNN Adaptation for SLO-Aware Distributed Inference in Dynamic Edge Environments. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673154

1 INTRODUCTION

Deep neural networks (DNNs) are revolutionizing many fields such as computer vision and natural language processing. These DNNs play an important role in many edge computing applications such as Augmented/Virtual Reality (AR/VR), Internet of Things (IoT), and autonomous driving. Many of these applications have real-time requirements, where they must meet strict Service Level Objects (SLOs) for latency or accuracy. This is essential for ensuring safety or enhancing the user experience.

Meeting SLOs of DNN inference in edge computing applications is challenging, due to the limited computing resources of edge devices and the dynamic network conditions. It is important to consider both the DNN model selection and deployment strategy in order to meet the accuracy or latency requirements.

One traditional way is to run DNN inference in a single edge device or a cloud server, shown in Figure 1(a). However, it can often be challenging to meet strict SLOs using these methods. When running the inference of a DNN model in an edge device, it can

incur high latency due to the limited computing resources. On the other hand, deploying DNN inference in a cloud server can also result in high latency, attributed to the network delay from communicating with a far-away cloud server.

Recent research has demonstrated that partitioning a DNN model across multiple devices can enhance inference performance by reducing latency. An illustration is shown in Figure 1(b). The papers by Teerapittayanon [13] and Kang [7] explore the concept of partitioning deep neural network models on a layer-by-layer basis. This involves executing the initial layers on a local device and the subsequent layers on a separate remote node with greater computational resources. The authors of [16] focus on spatial partitioning where a DNN layer is partitioned into multiple sublayers and distributed across multiple nodes for parallel execution. While these techniques have the potential to enhance inference latency, they all use a single fixed DNN for inference, thereby limiting adaptability during runtime to handle varying accuracy and latency requirements under complex environments. For instance, when varying bandwidth between 5-500 Mbps (shown in Figure 16(b)), a state-of-the-art fixed DNN method only has a 0-44% SLO compliance rate. In other words, using a static DNN fixes the accuracy and restricts the trade-off between latency and accuracy.

To this end, we propose an SLO-aware distributed inference framework termed *Murmuration*, which jointly adapts neural network architecture settings and model partition strategy according to the underlying infrastructure condition and SLOs. This is illustrated in Figure 1(c). The goal is to offer a more dynamic and adaptive solution to meet application SLOs under diverse network characteristics.

Designing this framework entails addressing two primary challenges. Firstly, there exists a multitude of potential DNN models and model configurations to select from, aiming to effectively cater to diverse SLOs, network conditions, and device types. For instance, for a 5-device (1 local and 4 remote) environment, with ten bandwidth settings, ten network delay settings, and ten inference latency constraints, we have 10^9 different settings (9 = 4 remote devices $\times$ 2 metrics + 1 inference latency constraint). This further makes preparing the DNN models and finding the appropriate models challenging. Secondly, to support mobility of devices at the edge and varying network connectivity, it is imperative to dynamically adjust the DNN model and its deployment in real-time during runtime to consistently adhere to the SLO.

To tackle the first challenge of finding different DNN models specialized for different environment settings and SLOs, we use one-shot Neural Architecture Search (NAS) techniques to train partitionable DNNs. In effect, we train a "supernet", which is a large network that encapsulates smaller subnetworks. Once trained, the supernet provides a large number of submodels that can be sampled according to specific targets. We use spatial partition, feature map quantization, model depth, and model width as the search criteria. This makes the learned model more partition-friendly in comparison to other existing approaches.

To support real-time model adaptation, Murmuration formulates the DNN model and partitioning strategy search as a goal-conditioned multi-task Reinforcement Learning (RL) problem, where the SLO is the goal and network conditions construct different tasks. Murmuration proposes a novel RL training algorithm called SUPREME (Share, bUcketed, PRunE, Epsilon-greedy, Mutation Exploration), to improve exploration and data sharing across multiple tasks. SUPREME maintains a bucketed replay buffer for different SLO constraints and network conditions, and shares the model selection and partitioning strategies between the buckets when possible. It tries to explore and find critical SLOs and network conditions points to reduce the search space, so it can attain better joint DNN model and partitioning decision performance.

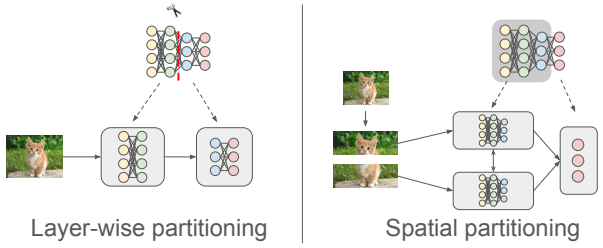


Figure 2: Background of layer-wise and spatial partitioning

We implemented Murmuration to support efficient execution on a wide range of edge devices. We conducted extensive experiments for image classification tasks with a variety of network conditions and SLOs. Under different device and network conditions, Murmuration is able to achieve up to 6.7× latency reduction or up to 5% higher accuracy compared to existing approaches. Murmuration excels in meeting stringent SLOs concerning accuracy and latency. Compared to existing approaches, it adapts more efficiently under a wider range of network conditions, achieving up to 52% SLO compliance rate improvement compared to the baselines.

The main contributions of the paper are:

- A novel approach for jointly tailoring neural network settings and partition strategy for distributed inference based on device network conditions and SLOs;
- A one-shot NAS supernet for distributed execution by using new search space and training techniques;
- An RL training technique called SUPREME, which improves exploration and sharing during training, and significantly outperforms existing RL algorithms;
- Design and implementation of Murmuration, a SLO-aware distributed inference system with smart model selection and partitioning;
- Extensive evaluation of Murmuration, demonstrating its performance and its ability to meet SLOs with high probability.

2 BACKGROUND & RELATED WORKS

2.1 Distributed Inference

The literature on model-based distributed inference at the edge can be broadly classified into: 1. Layer-wise partitioning; 2. Spatial partitioning. Figure 2 illustrates these two types of model partitioning.

Layer-wise partitioning executes early DNN layers in a local compute node and offloads later layers to a remote node. [4] is one of the earlier works that offloads the later stages of image classification computation to the cloud. Neurosurgeon [7] proposes a method to automatically partition DNN models between a mobile device and a cloud server based on network latency and energy

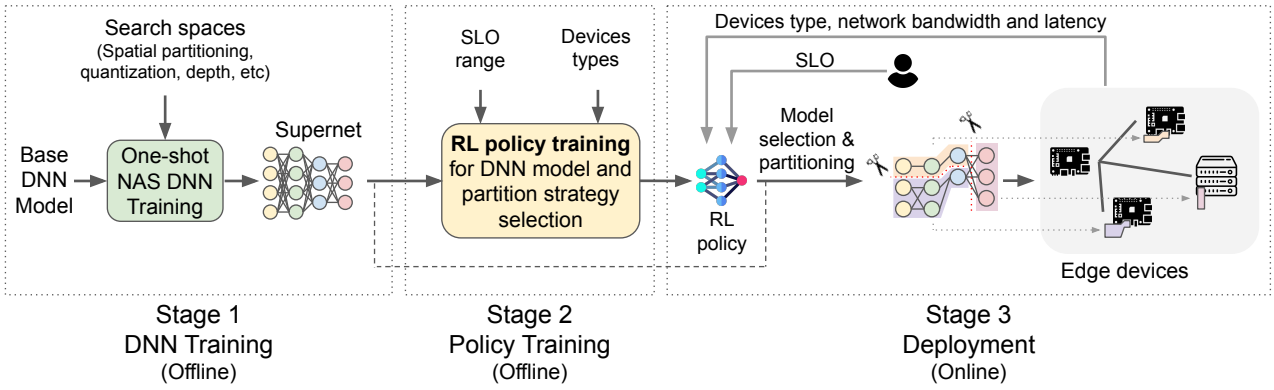


Figure 3: Murmuration High-Level Operation Procedure

consumption. [5] goes a step further to design a Dynamic Adaptive DNN Surgery Scheme that generates an optimal partition of a DNN based on dynamic network conditions. The authors formulated the problem as a min-cut problem and a optimal solution is proposed. However, the performance gain from distributed execution is limited as the model structure and the network bandwidth can restrict the inference latency. BranchyNet [13] utilizes exit points in initial network layers for quick local decision-making. If early layers are uncertain, computations are offloaded to the cloud for enhanced accuracy. While this approach improves inference latency, its accuracy varies based on input data, leading to uncertainty in inference outcomes.

Spatial Paritioning is another way model partitioning, where each layer's input feature is split into multiple parts, and multiple nodes are used to execute them in parallel. The goal of these partitioning methods is to reduce inference latency or single-node computation load for inference tasks. DeepThings [17] tries to offload inference tasks to a cluster of edge devices by partitioning the CNN inputs of each layer spatially into multiple tiles. ADCNN [16] proposes a method called Fully Decomposable Spatial Partition (FDSP) to efficiently partition the input spatially. It adds zero padding to the edge of each tile to reduce cross-partition dependency and to use progressive training to assure high inference accuracy. [9] studied spatial partitioning using Matching theory and proposed an adaptive partitioning method to automatically partition the input to minimize communication time and reduce storage requirements. CoEdge [14] designs a system for distributed inference using spatial partitioning, but it focuses more on reducing the energy consumption of the IoT computing devices. EdgeFlow [6] models DNNs as Directed Acyclic Graphs (DAGs) and proposes a progressive algorithm to optimize spatial partitioning.

The aforementioned works only consider a single network at a time which limits inference performance and their ability to provide flexible accuracy and latency tradeoff. In this paper, we co-design the model and the system in order to optimize the inference performance based on the application requirements and infrastructure conditions. Moreover, we aim to develop a more general distributed inference system that combines the advantages of the above two types of distributed inference.

2.2 DNN model adaptation/switching

Some previous works focus on DNN model adaptation based on resource constraints. One type of adaptation falls in the subfield of one-shot NAS. "Once-for-all" [1] proposes a progressive shrinking technique to train a "supernet" that contains a large number (10^{19}) of submodels with different kernel sizes, depths, and channel sizes. Once trained, it can search for the best model for the target device based on its resource constraint and latency constraint using an evolutionary search algorithm. Later, Autoformer [2] extends this approach to Visual Transformer. Another type of model adaptation is investigated in the DNN server literature. INFaaS[11] and [15] are frameworks that are used for AI-as-a-service running in the cloud and select a model based on the load and underlying resources. Although these works consider DNN model adaptation, they all operate in a single device, while our work focuses on the distributed execution of DNN across multiple devices with DNN model adaptation.

3 REQUIREMENTS & TARGET SCENARIOS

We aim to create a real-time, adaptable framework for distributed Deep Neural Network (DNN) inference, focusing on optimizing DNN model selection and partitioning strategies based on user-specific SLOs and network conditions. This framework focuses more on image classification tasks with Convolutional Neural Networks (CNNs), but the same technique can also be further extended to support other DNNs such as transformer-based models. The framework operates on multiple diverse devices located in close proximity with varying computational power and networking capabilities (in terms of network bandwidth and delay). Based on user-specified SLOs in terms of latency or accuracy, the framework selects DNN models and partitioning strategies in real-time by considering device-network attributes, ensuring the decision process does not impact inference execution.

4 METHOD

In this section, we present the detailed design for Murmuration. The operation of Murmuration has 3 stages: 1) DNN training; 2) Policy training; and 3) Deployment. Figure 3 shows a high-level overview of how the framework operates. During the first stage, we employ

NAS techniques to train a supernet DNN. This supernet contains multiple submodels, which can be selected according to SLO and network conditions. In the second stage, we employ our SUPREME algorithm to train a RL policy capable of handling model selection and partitioning. This RL policy is applied in the third stage, during runtime, to make decisions on model selection and partitioning, which guide the distributed execution of DNN inference.

We will first briefly describe partition-ready one-shot NAS model training. Then we focus on details of our SUPREME RL algorithm for joint DNN model and partition decision-making.

4.1 Stage 1: Partition-Ready One-Shot NAS Training

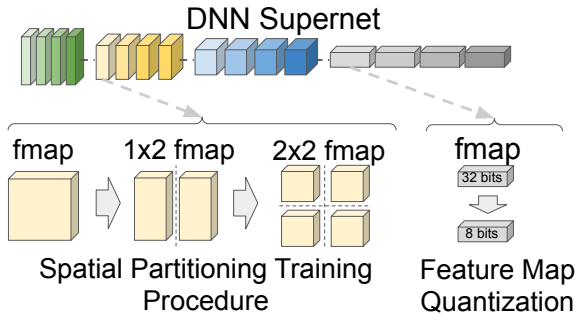


Figure 4: Supernet and Search Space (dynamic model depth and width not shown)

To tailor a DNN for different network conditions and SLOs, we need flexible DNN settings that can cover different situations. One-shot NAS techniques are a promising approach to training a large supernet that contains multiple submodels with different model architecture settings. However, most of the one-shot NAS techniques such as [1] are designed for a single device, which makes them not suitable for our problem. Distributed execution of a DNN requires the model to be partitionable layer-wise and spatial-wise. We also need to minimize the amount of data that is transferred between devices to reduce inference latency caused by communication.

With this in mind, we propose to include spatial partitioning [16] and input quantization, on top of kernel size and block depth (number of layers), as search spaces during one-shot NAS training. Figure 4 shows a high-level illustration of this idea. For CNN models, spatial partitioning enables the division of a CNN layer's input feature map into smaller patches (as shown in the yellow boxes in Figure 4), allowing for separate computations to be performed on each patch, which further enables the parallel computation for each layer across multiple devices. To further reduce the communication required between CNN operations on these patches, we leverage the FDSP technique from [16], which adds zero padding to each patch to eliminate the need for communication between neighbor patches. Using FDSP improves the model's performance in latency due to reduced network communication and parallel execution, but it can also have a small impact on the model's accuracy. This characteristic in fact provides us more flexibility in trading off between accuracy and latency. We introduce different options for spatial partition (e.g., 1×2, 2×2, etc) into the NAS training, which allows us to select different spatial partition strategies at run time. This gives us more variety of models to support different network conditions and SLOs. Note that this spatial partitioning strategy can also be applied to other DNN models such as Vision Transformers, where different image patches are sent to different devices for parallel attention computation in order to improve inference latency.

Furthermore, the input quantization search space involves the quantization bitwidth selection for the inputs of each DNN layer. This quantization option is useful when transmitting the intermediate DNN data between two devices (such as in layer-wise partitioning), as it can effectively reduce the communication volume during transmission. With these search spaces for one-shot NAS, we obtain a partition-ready DNN supernet that can be used by the following stages for policy training and deployment.

4.2 Stage 2: Reinforcement Learning Problem Formulation and Policy

4.2.1 Formulation. Similar to prior research in NAS [10] and DNN placement [8], we employ RL as our approach to select the submodel in supernet DNN produced in the first stage. Nonetheless, our challenge diverges from earlier research in two aspects: 1) We are making joint decisions about both the submodel configuration and the partition/placement strategy, whereas prior studies typically focus on only one of these aspects. 2) Our objective is to meet user-specified SLOs across a wide spectrum of network conditions, significantly complicating our problem compared to prior work.

We formulate the above problem as a goal-condition multi-task RL problem. The SLO provided by users is considered the goal for the RL problem. Different network conditions of the devices are considered different tasks. Intelligent RL policy needs to be developed to select a DNN submodel and partition strategy to meet the SLO under diverse network conditions.

More specifically, the model selection and partitioning problem is formulated as a sequential decision-making problem where the policy makes decisions for each DNN layer sequentially. Each layer of DNN model is separated into multiple actions, each either corresponding to a DNN model setting y_i of the layer k ($a^k_{y_i}$) or the device selection for partitions p_j of the layer (a^k_p), where y_i is the ith model setting and p_j is the jth partition in the layer. We define all the actions $\bar{a} = \{a^0_{y_0}, ..., a^0_{y_Y}, a^0_{p_0}, ..., a^0_{p_P}, ..., a^1_{y_0}, ..., a^m_{y_0}, ...\}$, where Y is the number of model settings for each layer, P is the maximum number of partition in each layer, and m is the total number layers. The goal of this RL problem is to optimize the following formula:

$$\max \mathbb{E}_{c\sim p(c)}\left[\mathbb{E}_{\bar{b}\sim p(\bar{b}),\bar{l}\sim p(\bar{l})}\left[\mathbb{E}_{\tau\sim\pi}(r(\bar{a}))\right]\right] \tag{1}$$

where r is the reward of the model, c is the SLO, $\bar{b}$ is a vector of the bandwidths of all the devices, $\bar{l}$ is the network delays of all devices, π is the RL policy, and τ is the trajectory. In this work, there are two ways to define the SLO type. With the goal-conditioned formulation, if inference latency is used as SLO, we have:

$$r = \begin{cases} \alpha A - \beta & \text{if } L \le L_{SLO} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

and if inference accuracy is used as SLO:

$$r = \begin{cases} \alpha L - \beta & \text{if } A \ge A_{SLO} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where A, A_{SLO} are inference accuracy and accuracy constraint respectively, L, L_{SLO} are the inference latency and latency constraint respectively, and α, β are hyperparameters.

4.2.2 RL Policy Design. Using the above formulation, we perform model setting and partitioning strategy selection through sequential decision-making. Figure 5 shows the model of the policy we used. The backbone of our policy model is the Long Short-Term Memory (LSTM) network, which facilitates the propagation of information across successive decisions. An LSTM is preferred over a transformer in this instance due to its lower computational power requirement. The state of the policy, denoted as s, consists of the DNN's model settings decided so far (m^{l-1}), the SLO constraint (c), and the network bandwidth $b_{d_1}, b_{d_2}, ...b_{d_n}$, network delay $l_{d_1}, l_{d_2}, ..., l_{d_n}$, and device type $u_{d_1}, ...u_{d_n}$ for devices $\{d_1, d_2, ..., d_n\}$. For each category of action (e.g., spatial partitioning, quantization, etc.), the model incorporates specialized fully connected layers. Through a sequence of decisions for both model setting and partition placement, we choose the strategy that meets the SLO under a specific network condition.

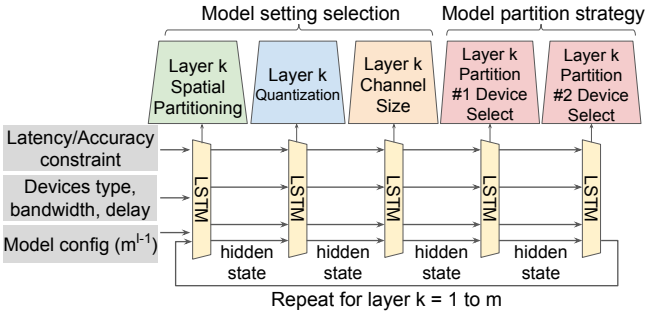


Figure 5: RL policy design

4.3 Stage 2: Challenges of Existing RL Methods

One may consider solving the above problem formulation using traditional RL algorithms such as Proximal Policy Optimization (PPO) or Deep Q Network (DQN). However, our experiments showed that traditional RL algorithms give suboptimal performance. We believe this is due to the high dimensionality of our problem and the limited exploration of traditional RL algorithms. With the goal-conditioned formulation, the model would obtain 0 rewards if the SLO constraint is not met. In this case, the RL model can often get no signal if the exploration does not end up finding a model and partition combination that satisfies the SLO. This problem is further exacerbated with the multi-task setting where the number of different devices' network condition configurations is extremely large, This makes the exploration even harder as the network needs to explore and find trajectories that can cover the different network conditions while satisfying the SLO. Our evaluation results in Figure 12 confirm this hypothesis.

Goal-Condition Supervised Learning (GCSL) is shown to have better or similar performance compared to the standard RL technique for goal-reaching tasks [3]. It learns goal-reaching behaviors without the need for external demonstrations or a value function. It collects trajectories, relabels them using hindsight to be optimal for the goals that were actually reached, and uses supervised imitation learning to train an improved goal-conditioned policy. However, GCSL also suffers from poor exploration. To Solve the above problem, we propose the SUPREME algorithm below.

4.4 Stage 2: RL Training with SUPREME

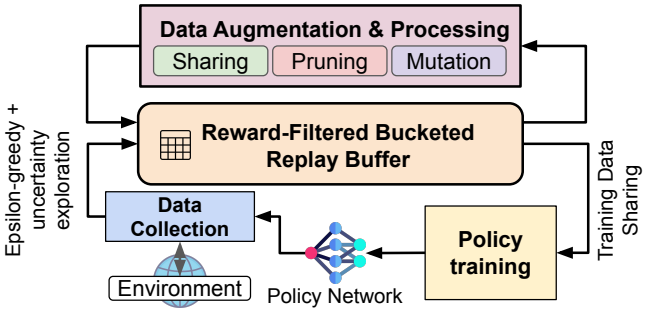


Figure 6: SUPREME RL Training Overview

In the SUPREME training flow, as illustrated in Figure 6, a central component is the reward-filtered bucketed replay buffer, holding replay data used for policy training. The training comprises two loops. The lower loop aligns with conventional RL policy training, involving using replay buffer data for policy training and applying the updated policy with exploration to accumulate more data into the replay buffer. We use GCSL for training the RL policy. The upper loop focuses on optimizing the replay buffer through processes such as sharing, pruning, and mutating data, aiming to augment the exploration capacity of the RL training. SUPREME differs from existing RL training techniques in three main areas: 1) Replay buffer structure; 2) Training data share across tasks; 3) Replay buffer augmentation through sharing, pruning, and mutation. Next, we provide details of the components in the figure.

4.4.1 SUPREME Algorithm. The SUPREME algorithm is designed to enhance exploration within our specific problem domain. It integrates bucketed data sharing, data pruning, epsilon-greedy exploration, and mutation mechanisms. The foundation of SUPREME's design is the following observation:

A strategy (model selection and partitioning), discovered for a specific SLO and network conditions constraint, effectively serves as a lower bound when these constraints are relaxed.

For instance, if a strategy is discovered for a specific SLO, bandwidth, and network delay, it remains applicable when there is a higher inference latency allowance, increased bandwidth, or reduced network delay. While this strategy might not represent the optimum solution, it serves as a valuable lower bound to facilitate our search process for high-reward strategies. Figure 7 shows an example of this observation in 2D. This idea is applicable to higher dimensional space as well. In 3D, for example, the grey area would be a box.

This observation allows us to **share the exploration results across multiple tasks**, which can significantly increase the data available for training under different constraints. From our observation, this can quickly boost the policy's performance under a large

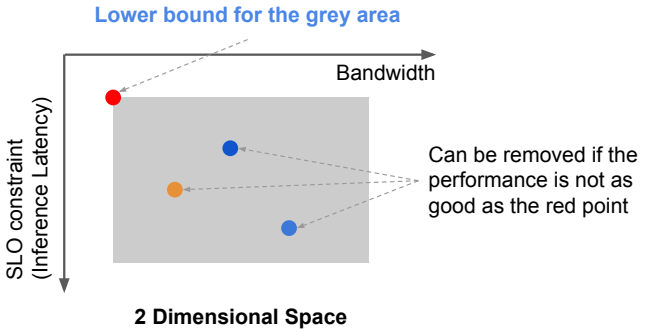


Figure 7: Illustration of a strategy as a lower bound for relaxed constraints (in 2D)

number of constraints, and in turn, improve the exploration in the next iteration.

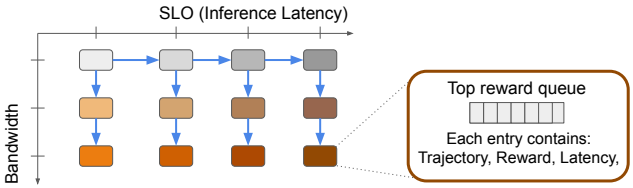


Figure 8: Bucketed Replay Buffer with reward filtering and bucket tree

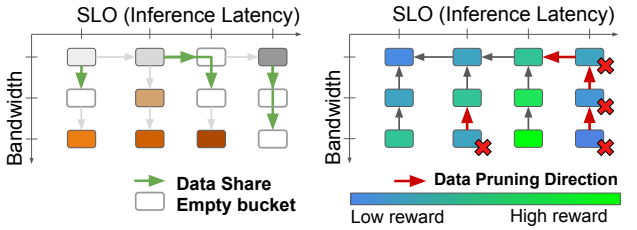
Bucketed Replay Buffer: In the SUPREME implementation, we utilize a bucketed replay buffer, enhanced with reward prioritization. This is illustrated in Figure 8. The process begins by discretizing the constraint space into distinct buckets. Each bucket comprises a queue to store replay trajectories, retaining only the top n reward data, where n symbolizes the queue size.

When the exploration mechanism yields new trajectory data, it undergoes a reward and state relabeling process. Subsequently, if the data is among the top n rewards, it is added to the appropriate bucket.

To enable efficient replay data sharing across various tasks, a tree structure is constructed among the buckets. It's worth noting that the tree structure varies based on the dimensionality of the constraint space. For instance, a 2D constraint space results in a binary tree as depicted in Figure 7, while a 3D space gives rise to a ternary tree, and so on.

Data Share Across Tasks: During training, we share replay data across multiple buckets by following the reverse direction of the tree. If a bucket is empty, we use the data in the nearest ancestor's bucket. Figure 9(a) illustrates this idea. This ensures that all the buckets contain data when sampled (as long as a non-empty ancestor bucket is available). The tree structure design also makes sure we can look up data quickly.

Data Pruning: Another key piece of information obtained from our aforementioned observation is that: if we find a strategy $\bar{a}_{new}$ with a lower reward (the blue or yellow dots in Figure 7) than the lower bounded strategy $\bar{a}_{lb}$ (red dot in Figure 7), then this $\bar{a}_{new}$ strategy can be pruned as it's not as good as the low bound strategy.



(a) Data Share across buckets (b) Data pruning

Figure 9: Data share and pruning in bucketed replay buffer

This observation essentially allows us to cover the continuous constraint space using a discrete number of strategies. Mathematically, SUPREME turns the RL optimization objectives from Equation 1 into the following:

$$\max \mathbb{E}_{c_i,\bar{b}_i,\bar{l}_i \sim p(\mathbf{z})} [\mathbb{E}_{\tau\sim\pi}(r(\bar{a}))] \tag{4}$$

where $c_i, \bar{b}_i, \bar{l}_i$ is a critical SLO and network condition, and $\mathbf{z}$ is a set of all critical configurations. This method allows us to focus on fewer data points while not compromising performance.

To implement this in our bucketed replay buffer, we use the tree structure of the buckets to help with the pruning task. Again we traverse the reverse direction of the tree and if the nearest ancestor bucket has higher rewards, then data in this bucket can be pruned. Figure 9(b) illustrates this idea.

Data Mutation: Lastly, we add data mutation to data in the replay buffer. This basically tries to randomly perturb some actions of the trajectory data in the replay buffer to get new trajectory data, which is then relabeled and added back to the replay buffer. We also added simple mutation heuristics such as improving execution locality and updating suboptimal buckets.

5 MURMURATION IMPLEMENTATION

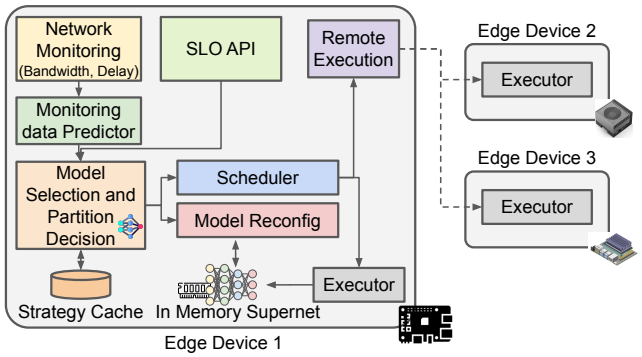


Figure 10: Murmuration Implementation Diagram

We implemented Murmuration to support the deployment of distributed inference. This section focuses on the deployment and runtime (i.e. stage 3 of the Figure 3) of the Murmuration framework. Figure 10 shows the high-level diagram of the system. We will provide details of each module below: The **Network Monitoring**

module monitors network delay and bandwidth using active and passive methods to measure connectivity between local and remote devices. The **SLO API** enables users to specify latency or accuracy SLOs as a scalar value.

The **Monitoring data Predictor** module predicts short-term monitoring data change, which allows us to precompute the model selection and partition strategy in advance. It utilizes a lightweight linear regression method for monitoring data prediction.

The **Model Selection and Partition Decision** module utilizes the RL model to determine the optimal DNN model selection and partitioning, based on both real or predicted monitoring data.

A **Strategy Cache** is utilized to store the known constraint (network condition and SLO) to strategy (model selection and partition) mapping. This reduces the need for redundant execution of the RL model. The model selection and partitioning decisions are sent to the **Model Reconfig** module for updating the local submodel. Then the **Scheduler** dispatch different parts of the model to be run locally or remotely. We use gRPC for device communication.

5.1 Fast Model Adaptation

The Murmuration system decides on model selection and partitioning before each inference request to adapt to changing network conditions. On top of the relatively fast RL algorithm in decision-making, a Strategy Cache is used to further reduce the need to frequently execute the RL algorithm. Moreover, The Monitoring Data Predictor forecasts network conditions, allowing for precomputation with RL algorithm and caching of strategies for fast adaptation.

Furthermore, we load the full supernet model in memory instead of loading only the submodel. This is desirable as we can perform model switching without memory copies or disk access, and thus significantly reduce the model switch time. We show in Section 6.4.5 that we can perform model switching in a few milliseconds. These techniques combine to enable fast model adaptation during runtime.

6 EVALUATION

To assess the effectiveness of the Murmuration framework, we first evaluate the performance of the SUPREME RL algorithm. Following that, we evaluate the overall performance of the Murmuration system when deployed across multiple edge devices. Evaluations are done on two scenarios:

- **Augmented Computing Scenario**: This configuration comprises one device with lower computational capacity, designated as the local device, paired with a higher computational power device equipped with a GPU. An example use case is AR/VR where a resource-constrained headset is often paired with a more powerful device. In our experiement, we use one Raspberry Pi 4 and one Desktop with AMD Ryzen 5500 CPU and Nvidia GTX1080 GPU.
- **Device Swarm Scenario**: This configuration consists of five edge devices with modest computational power. One acts as the local device, and the remaining four are considered as remote devices. Example use cases are cooperative robots and drones in search and rescue, remote sensing, and agriculture. We used 5 Raspberry Pi 4 in our experiment.

6.1 SUPREME Algorithm Evaluation

In the evaluation of the SUPREME algorithm, we focus on the two main metrics: reward and SLO compliance rate. We compare SUPREME to the following baselines:

- **PPO** [12]: Proximal Policy Optimization is a state-of-the-art on-policy RL algorithm.
- **GCSL** [3]: Goal-Condition Supervised Learning is an iterative supervised learning algorithm for goal-condition RL problems.

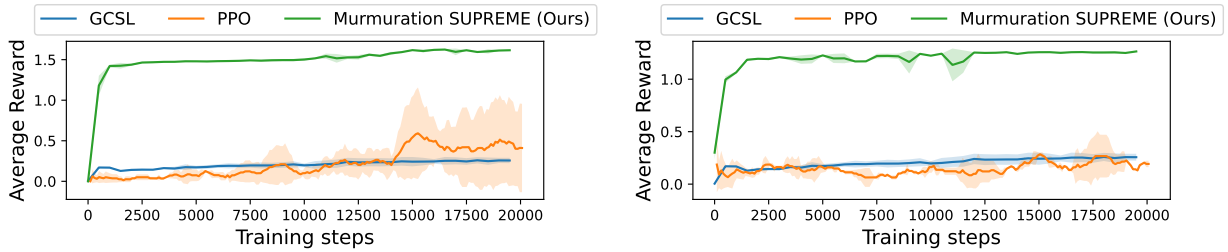
6.1.1 Training Experiment Setups. Experiments are done for the two aforementioned scenarios. We allow each device to have different network latency and bandwidths. We set a maximum and minimum value for all the metrics: network delays, bandwidths, and latency SLOs. For each of these metrics, we use 10 discrete points for training. Note that during inference time, continuous values can be used for these metrics.

For model selection in SUPREME, we train a supernet using a variance of MobileNetV3 as the base network. The supernet consists of 6 customizable settings for each layer: varying spacial partitioning (1x1 to 2x2 partitioning); input feature quantization (32bits to 8bits); image resolution (224 to 160); model block depth (4 to 2), model kernel size (7 to 3). This supernet is trained with the ImageNet data, and an accuracy predictor is used for accuracy prediction during RL policy training. For the Policy network, we use a 1-layer LSTM with 256 hidden units, and each action type (e.g. model setting, device selection) uses a different fully connected layer for generating the output.

For SUPREME training, we also incorporate curriculum learning where we gradually add in new constraint space. For instance, we start with varying SLOs and device 1 bandwidth, then we slowly add device 1 delay, device 2 bandwidth, and so on. For both GCSL and Murmuration, two trajectories (one for the max size submodel and one for the min size submodel) are used to bootstrap training. We run each training 3 times to obtain the average training reward and variance.

6.1.2 RL Policy Training Results. Figure 11 shows the training performance in terms of average reward achieved versus the number of training steps, when using inference latency as the SLO. The average reward is the mean of reward across all validation constraints (i.e. evenly distributed points in the SLO and network conditions space). We see that SUPREME significantly outperforms the baselines in terms of reward achieved. This illustrates SUPREME's enhanced efficiency in exploring model selection and partition strategies, leading to high inference accuracy.

Figure 12 shows the normalized SLO compliance rate throughout the training. Since some constraints are not achievable (e.g. 100ms inference latency constraint with a network delay > 100ms), we normalize the compliance rate by the highest achievable compliance rate of all methods (i.e. focusing on the satisfiable constraints). SUPREME is able to achieve a much higher compliance rate than the baselines. This again underscores the advantages of SUPREME's data sharing and exploration mechanisms. Moreover, SUPREME is able to learn the policy with a relatively low amount of data, attributed to the bucketed reward-filtered replay buffer data structure. Overall, SUPREME provides an effective way to train an RL



(a) Augmented Computing Scenario

(b) Device Swarm Scenario

Figure 11: Average Reward Throughout RL Policy Training with Different Number of Devices

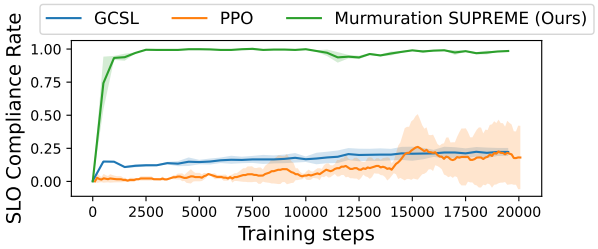


Figure 12: SLO Compliance Rate Throughout RL Policy Training

policy for selecting the DNN model and partitioning strategy under a large number of SLO, network delay, and bandwidth constraints.

6.2 Murmuration Distributed Inference Evaluation

We evaluated the overall performance of Murmuration when deployed on distributed devices at the edge. We compare the Murmuration's performance with other distributed inference frameworks.

6.2.1 Baselines. We compare Murmuration to the following distributed inference work:

- **Neurosurgeon** [7]: Layer-wise partition framework that splits DNN on two devices for inference acceleration.
- **ADCNN** [16]: A spatial partition framework that partitions feature maps spatially across multiple devices for parallel execution and inference speed up. It uses a finetuned CNN to reduce communication.

We combine these baseline distributed methods with different existing DNN models where possible to get a more comprehensive list of baselines.

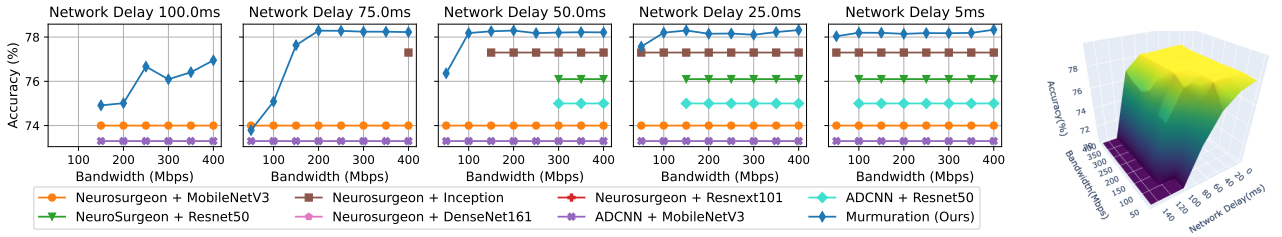
6.3 Experiment Setup

We evaluate the Murmuration system for the two scenarios introduced at the beginning of this section. These physical devices are connected to an Ethernet switch with 1GB wired connections. We control the network bandwidth and latency by using the tc traffic control tool. Each data point is collected by running 20 inferences with ImageNet images and taking the average inference latency.

6.4 Results

6.4.1 Inference Latency as SLO. We first evaluate Murmuration's performance when constraining on the inference latency. We vary the SLOs and network conditions, and compare inference accuracy. Figure 13(a) and Figure 14 depict the inference accuracy for the Augmented Computing Scenario and Device Swarm Scenario under various constraints, respectively. In Figure 13(a), we fix the latency constraint (at 140ms) and vary the bandwidth and network delay. For Figure 14, we keep the network delay fixed (at 20ms) and change the Latency SLO constraint and bandwidth of one out of five devices. For each subfigure, a dot is shown if the specific method is able to satisfy the latency SLO constraint. From these results. We can conclude that Murmuration is able to adapt the model setting and the placement in order to meet the SLO. Murmuration has the largest coverage of different network conditions while satisfying the SLO. Notice that some of the high accuracy baselines such as Neurosurgeon + DenseNet161 (77.1%) and Neurosurgeon + Resnext101 (79.3%) have DNN models that are resource demanding, and are not able to satisfy any SLO in Figure 13(a). Murmuration is also able to achieve the highest accuracy while satisfying the latency SLO constraint. More specifically, Murmuration achieves up to 5% higher accuracy compared to other baselines. This is attributed to Murmuration's model adaptability. A 3-D illustration of this is provided in Figure 13(a).

6.4.2 Inference Accuracy as SLO. Next, we look at Murmuration's performance when we use inference accuracy as the SLO. Here, we want to compare the inference latency between different methods when a certain inference accuracy is required. Figure 15 shows inference latency when constrained on different inference accuracies (each subfigure corresponds to a different bandwidth). Again, we can see Murmuration being able to adapt its model and partition strategy to achieve lower inference latency when the accuracy constraint tightens. Across different bandwidths (subfigures), the Murmuration's latency curve also changes its shape to be lower when more bandwidth is available. Furthermore, Murmuration is able to cover the most accuracy constraint range while maintaining a low inference latency. In fact, Murmuration is able to achieve up to 85% (6.7×) latency reduction at high accuracy constraints when compared with baselines that can also satisfy the SLO. Overall, Murmuration is able to customize its model selection and partition strategy to efficiently meet inference accuracy SLO.



(a) Results Grouped by Network Delay (b) 3D plot

Figure 13: Augmented Computing Scenario: Inference Accuracy for Different Network Conditions @ Latency SLO = 140ms

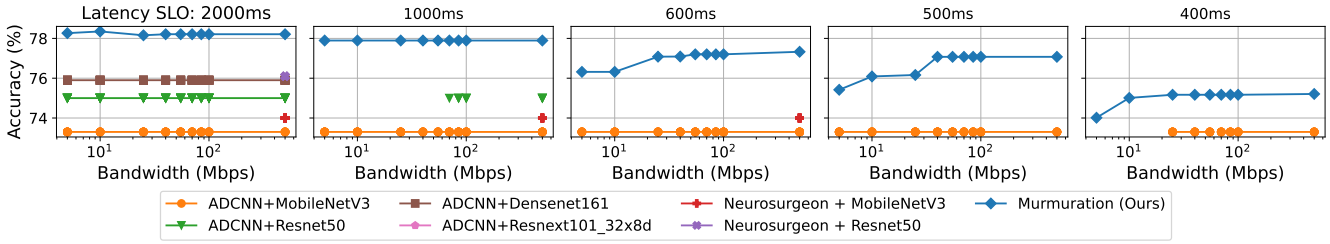


Figure 14: Device Swarm Scenario: Inference Accuracy for Different Latency SLO and Bandwidth @ Network delay=20ms

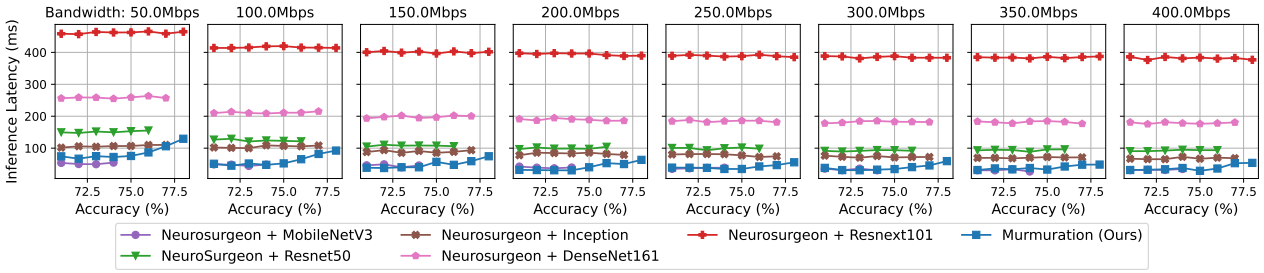
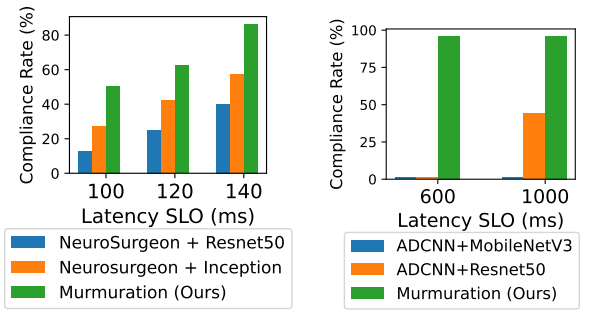


Figure 15: Inference Latency for Augmented Computing Scenario when using Accuracy SLO (Lower Inference Latency is Better)

6.4.3 SLO Compliance Coverage Comparison. One of the key benefits of Murmuration is the ability to support challenging network conditions. We demonstrate the SLO compliance rate of Murmuration and some of the best baselines under both latency and accuracy SLO. Given a fixed latency and accuracy SLO, we run each method across a wide range of network delays and bandwidth settings. Then compliance rate is defined as the ratio of the number of network settings that the method can achieve the SLO. Figure 16 shows the SLO compliance rate for the two experiment scenarios under different latency and accuracy constraints. We can see that Murmuration is able to significantly improve the compliance rate of by up to 52%, demonstrating Murmuration's ability to adapt and continue to satisfy SLO under diverse network settings.

6.4.4 Scalability of Murmuration. We analyze the scalability of Murmuration by evaluating its inference latency when working with different numbers of devices (Raspberry Pi 4 in this case) under a fixed network condition (1Gbps link with 2ms delay) with an accuracy SLO. The results are shown in Figure 17. With the increased number of devices, we can obtain 1.7x to 4.5x latency speed up compared to executing in a single device. The scalability bottleneck is mainly attributed to the communication overhead and execution time for the centrally executed fully connected layers.

6.4.5 Runtime Efficiency Evaluation. Another key aspect of Murmuration is the ability to make the model selection and partition adaptation during run time. We run Murmuration's RL policy in two kinds of devices to measure the time it needs to make an adaption decision. We compare it to Evolutionary Search which is a commonly used technique for finding submodel in supernet. Figure 18 shows the results. We can see that Murmuration's RL approach significantly reduces the time needed to make a decision. The decision-making is about 1 second and 30 milliseconds in Raspberry Pi and GPU, respectively. This performance should be



(a) Compliance rate for the augmented computing scenario with 75% accuracy SLO. 40 network settings (delay: 5-100ms; bw: 50-400 Mbps)

(b) Compliance rate for the device swarm scenario with 74% accuracy SLO. 9 network settings (delay: 20ms; bw: 5-500 Mbps

Figure 16: SLO Compliance Rate Comparison

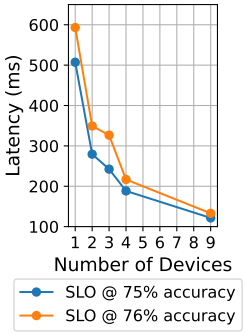


Figure 17: Inference latency with different number of devices

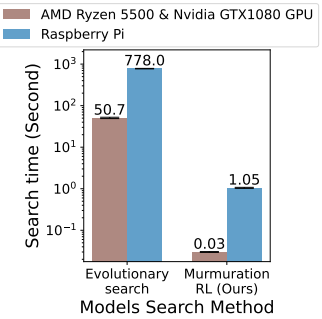


Figure 18: Average Reward Throughout RL Policy Training with Different Number of Devices

sufficient for most model adaptation tasks, especially if we use our precompute technique discussed in Section 5.

Next, we also evaluate the time for Murmuration's supernet to change its submodel. We compare it to the time of switching between different types of existing models. Figure 19 shows the results. We assume limited memory and switching different types of models will require reloading the weights. Murmuration's model can adapt its setting significantly faster than changing the model type as the supernet is preloaded in memory. This demonstrates the practicality of Murmuration, and it also justifies our design choice to load the complete supernet in memory.

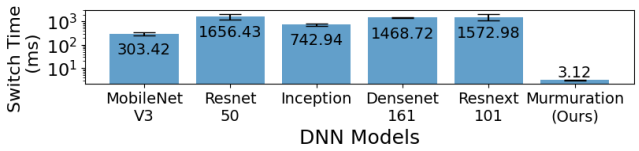


Figure 19: Model Switch Time Comparison (Raspberry Pi 4)

7 CONCLUSION

In conclusion, this work presents Murmuration, a framework for optimizing distributed inference at the edge. Murmuration, utilizing RL and one-shot NAS, dynamically customizes and partitions DNN models, addressing challenges posed by diverse network conditions, limited resources, and vast search spaces. It adapts to various network conditions and SLOs, demonstrating enhanced performance in inference accuracy and latency.

REFERENCES

[1] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791* (2019).

[2] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. 2021. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF international conference on computer vision*. 12270–12280.

[3] Dibya Ghosh, Abhishek Gupta, Ashwin Reddy, Justin Fu, Coline Devin, Benjamin Eysenbach, and Sergey Levine. 2019. Learning to reach goals via iterated supervised learning. *arXiv preprint arXiv:1912.06088* (2019).

[4] Johann Hauswald, Thomas Manville, Qi Zheng, Ronald Dreslinski, Chaitali Chakrabarti, and Trevor Mudge. 2014. A hybrid approach to offloading mobile image classification. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 8375–8379.

[5] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1423–1431.

[6] Chenghao Hu and Baochun Li. 2022. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 330–339.

[7] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[8] Hao Lan, Li Chen, and Baochun Li. 2021. Accelerated device placement optimization with contrastive learning. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–10.

[9] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. 2020. Distributed inference acceleration with adaptive DNN partitioning and offloading. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 854–863.

[10] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*. PMLR, 4095–4104.

[11] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[13] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, 2464–2469.

[14] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608.

[15] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. {Model-Switching}: Dealing with Fluctuating Workloads in {Machine-Learning-as-a-Service} Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[16] Sai Qian Zhang, Jieyu Lin, and Qi Zhang. 2020. Adaptive distributed convolutional neural network inference at the network edge with ADCNN. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[17] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.