Context-Aware Deep Model Compression for Edge Cloud Computing

Lingdong Wang*, Liyao Xiang*, Jiayu Xu*, Jiaju Chen*, Xing Zhao*, Dixi Yao*, Xinbing Wang*, Baochun Li†
* John Hopcroft Center for Computer Science, Shanghai Jiao Tong University, Shanghai, China
† Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada

***Abstract*—While deep neural networks (DNNs) have led to a paradigm shift, its exorbitant computational requirement has always been a roadblock in its deployment to the edge, such as wearable devices and smartphones. Hence a hybrid edge-cloud computational framework is proposed to transfer part of the computation to the cloud, by naively partitioning the DNN operations under the constant network condition assumption. However, real-world network state varies greatly depending on the context, and DNN partitioning only has limited strategy space. In this paper, we explore the structural flexibility of DNN to fit the edge model to varying network contexts and different deployment platforms. Specifically, we designed a reinforcement learning-based decision engine to search for model transformation strategies in response to a combined objective of model accuracy and computation latency. The engine generates a context-aware model tree so that the DNN can decide the model branch to switch to at runtime. By the emulation and field experimental results, our approach enjoys a $30\% - 50\%$ latency reduction while retaining the model accuracy.**

***Index Terms*—Edge Cloud Computing, Neural Architecture Search, Reinforcement Learning**

I. Introduction

With the recent surge in the popularity of deep learning, edge computing platforms have begun to embrace the trend of using deep learning models. Smartphones and wearable devices are now communicating with users through 'smart' interfaces such as faceID, voice commands, typing suggestions, shopping recommendations, and many others, all driven by deep learning algorithms and DNN models. However, there is still a wide gap between the DNN capability that the edge devices can fully utilize and the state-of-the-art DNN power, largely due to a lack of computational resource and stringent latency requirement on devices.

Even equipped with the most advanced chips, today's edge devices are still at least 10 times slower than a GPU-powered server. On the other hand, with the rapid development of 5G, fast connectivity has become pervasive. The hybrid edge-cloud computing infrastructure is introduced to exploit the best of two worlds: by offloading intensive computation to the remote cloud server and retrieving the results, the edge is gaining significant speedup. Previous works [1]–[7] have closely studied partitioning and offloading policies. However, those policies are not satisfactory: since the structure of the DNN is fixed, one usually does not have much feasible choices, but only partition the DNN rigidly. Further, previous works always implicitly assume a constant context at the inference runtime. These two reasons inhibit the edge DNN from fully utilizing the edge-cloud infrastructure.

Most previous works only consider partitioning within a fixed DNN while ignoring its structural flexibility. In fact, a DNN can be transformed into another with mild degradation on the performance. Considering the resource limitation of the edge device, we can not only partition the DNN, but also compress the part residing on the edge to accelerate computation further. Previous works on neural architecture search [8], [9] have shown the viability of the approach. By trading off the model accuracy for a smaller model size, we reduce the computation time, the storage space and the energy consumption on edge devices. This inspires us to explore the sub-model structure of a DNN model for seeking the optimal placement strategy, which potentially leads to a better trade-off between computation accuracy and latency, since the search space has been enlarged.

In practice, the assumption of constant network condition at runtime rarely holds, because inference on complicated DNN models is typically computation-intensive and time-consuming, particularly those continuously receive and process inputs. Moreover, the edge devices are often in a constantly changing environment, such as switching from 4G to WiFi, from a still state to a moving one, etc. If taking the varying context into account, many of the previous approaches would perform poorly or even fail. If a decision engine only makes the placement decision once for all before inference, it is quite possible that the network condition would change during the inference, which means the original placement decision is not optimal from a temporal point of view. For example, if the network connectivity is poor at the beginning, the decision engine has determined to compress the model to run on the device; it will later regret its decision when the network condition gets better, and a fully-fledged model could have been running. The varying environment leads to challenges as well as opportunities for optimizing the DNN structure and placement, if we think about searching in a dynamic, rather than static DNN architectural space.

In a nutshell, we propose to search for the optimal DNN

*Liyao Xiang (xiangliyao08@sjtu.edu.cn) is the corresponding author with the John Hopcroft Center, Shanghai Jiao Tong University, China.

*This work was partially supported by National Natural Science Foundation of China (61902245), and the Science and Technology Innovation Program of Shanghai (19YF1424500).

deployment strategy in a richer context (with the awareness of varying network states) and larger space (in terms of the model granularity). By choosing a series of compression and partitioning strategies, the DNN is transformed into different states sequentially, the process of which can be modeled by a Markov Decision Process (MDP). Besides, the strategy search space is huge as we need to select compression techniques for each layer and decide to partition per DNN block. Thus, we employ a reinforcement learning-based optimizer to solve the MDP problem. Highlights of our contributions are:

- By expressing the DNN placement and structures as unified hyperparameters, we design a reinforcement learning-based engine to search for the optimal strategy to transform base DNN models.
- We propose a flexible model structure so that it can make the decision of compression and partition on the fly depending on the real-time context.
- We implement the decision engine and test it in a variety of real-world scenarios. The results show that our method reduces $30\% - 50\%$ inference latency while keeping the accuracy loss at about 1%.

II. Related Work

While DNN and deep learning algorithms have been widely applied, it still faces significant computational challenges when migrated to the edge. A wide range of work has been proposed to address the issue.

A. Running DNN in Mobile Systems

The focus of this category lies in speeding up the execution of DNN on the edge device, either by taking advantage of the hardware or designing a suitable DNN structure to run on the device. To fully utilize the processors on the device, Lane *et al.* [10] were among the first to design a low-power DNN inference engine on the device, taking advantage of both CPU and DSP chips to collaborate on mobile sensing and analysis tasks. Later, in [11] they decomposed the DNN architectures into blocks of various types, with each block efficiently executed by heterogeneous local device processors such as GPUs and CPUs.

There are works trying to explore the model's internal structure to speed up execution. For example, Huynh *et al.* [12] and Xu *et al.* [13] specifically exploited the inputs' temporal locality for reusing the results of the previous frame for calculating the current frame. The results of reusable image regions were cached to facilitate execution. Other works [1], [8], [14], [15] explore the accuracy-latency trade-offs for each deep learning model to fit the model's resource demand to a system's available runtime resources. In particular, Fang *et al.* [15] employed a novel multi-capacity model comprised of a set of models with different resource-accuracy trade-off for dynamically selecting the optimal one at runtime. Our data structure — model tree — shares some similarities with the multi-capacity model but with different purposes: we store the optimal strategy of adjusting the DNNs in accordance with varying network conditions, with the support of the cloud. Liu *et al.* [8] explored the trade-off between model accuracy and system performance with a reinforcement learning-based approach to find a set of compression techniques to transform the DNN into one that fits the customized system requirements. Differing from their work, we take the reinforcement learning-based approach to compute the optimal solution across the edge and the cloud.

B. Edge-Cloud Deep Learning

Although the cloud would provide abundant computational power to support DNN inference, as pointed out in [4], a straightforward partitioning of DNNs over the computing hierarchy may incur prohibitively high communication costs. Hence a number of works [1], [2], [4]–[6] proposed various ways to minimize such cost. In [4], Teerapittayanon *et al.* proposed to divide the DNN into different modules to deploy on the edge-cloud to improve the accuracy and fault tolerance while keeping the communication cost low. Han *et al.* [1] systematically traded off DNN classification accuracy for resource use in the multi-programmed, streaming setting with an optimization-based heuristic scheduling algorithm. Lin *et al.* [6] extended the problem to contain multiple edges, fogs, as well as cloud devices, and solved it with a genetic algorithm. Besides heuristic algorithms, some works proposed deterministic algorithms to find the optimal partition policy. Kang *et al.* [2] managed to find the optimal partition for chain-like DNNs. Hu *et al.* [5] turned the problem into a min-cut problem and found the optimal partition for DNNs represented by Directed Acyclic Graphs.

The aforementioned works typically optimized the configuration of a DNN under a constant network state and ignored architecture transformation of the deployment target. In contrast, we search for both compression and partition strategies without assuming that the network condition stays the same during DNN execution. To address the issue, we propose to compose the DNN on the fly from a model tree trained offline. The approach is able to find a more suitable DNN configuration scheme than the fixed plans.

C. DNN Compression

Our work adopts a variety of DNN compression techniques for DNN transformation. Many compression techniques have been proposed as in [11], [16]–[22]. In [16], Han *et al.* proposed to prune unimportant model weights to compress the neural network. Along with quantization, their method has reduced the neural network size by 35 times with almost no accuracy degradation. Unlike the non-structured pruning in [16], structured pruning kept the layer-wise structures intact but shrank the size of the neural network by removing the entire layer or scaling down the kernel size or the filter number [17]. Targeted at multiple mobile platforms, Liu *et al.* [8] trimmed down the network complexity by several compression techniques to fit resource constraints.

III. Preliminaries

In this section, we give a brief overview of the techniques adopted in this paper.

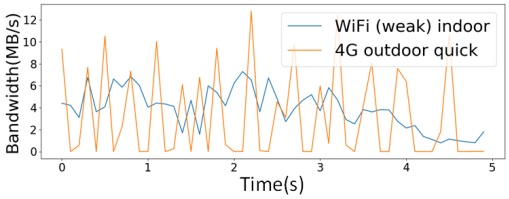

Fig. 1. Real-world network context.

TABLE I
INFERENCE LATENCIES ON XIAOMI MI 6X WITH INPUT SIZE: $1 \times 224 \times 224 \times 3$.

Model	Latency(ms)
VGG19	5734.89
ResNet50	1103.20
ResNet101	2238.79
ResNet152	3729.10

A. Neural Architecture Search

Until now, the architecture of most neural networks are subject to manual selection, but there is an ongoing trend to automatically search for optimal neural network structures [23], of which the accuracy could match the hand-designed one. However, the automatic search faces significant challenges when the state space is huge, and the choice of search space largely determines the difficulty of our problem. It is often critical to design the action space so that the optimization method can effectively search in the state space. To address the issue, we propose a two-stage strategy for reducing the search space.

Many search strategies can be used to explore the search space, such as greedy search, random search, Bayesian optimization, evolutionary methods, reinforcement learning, and gradient-based methods. Under the assumption of fluctuating network conditions, the decision engine needs to decide the compression and partition strategy according to the changing network context. The search space is that of the two strategies combined and can be huge. Hence we adopt a reinforcement learning-based approach and carefully design its action space, state space, and reward function to effectively search for an optimal policy.

B. Reinforcement Learning

Reinforcement learning is mostly concerned with how software agents take actions in an environment to maximize the cumulative reward. At each state, the agent takes action and communicates with the environment, which returns new observations and a reward for the current state and action. The agent adapts its policy, which returns a new action, and the action would lead to a new state of the agent. The objective of the agent is to learn an optimal policy to maximize the reward accumulated in the long run.

Since the optimal policy is far too complicated to find, many methods are proposed to approximate one. For example, there are value-based methods, policy-based methods, and combined methods like Actor-Critic. In our work, we adopt the Monte-Carlo policy gradient, a simple version of the policy-based method, to search for the optimal DNN structure and placement strategy across the edge and the cloud.

IV. MOTIVATION AND OVERVIEW

We first examined the static network assumption that most previous works make. We inspect the real-time bandwidth in different circumstances and Fig. 1 gives two samples. One sample depicts the bandwidth under 4G when the device moves quickly outdoor and the other shows a weak WiFi signal indoor. Both of them are measured on smartphone Xiaomi MI 6X. We observe the bandwidth changes drastically even within a small time window like 1s. However, on the same device, the inference time of classical deep learning models is larger than this scale as reported in Table 1.

The following example motivates our design. Imagine we have a DNN inference application that takes advantage of the edge-cloud infrastructure by sending intermediate-layer features to the cloud. Without inspecting the network condition, the application may make a bad decision and offload features when the network connectivity is poor. However, even if we inspect the network condition to decide when to offload, we may regret the decision at a later time if the network condition fluctuates during the inference. For example, the network condition may be poor at the beginning, and the decision engine determines to run a compressed DNN model on the device to obtain a result of lower accuracy; however, as the network condition instantly gets better, the running application misses the opportunity to offload fully-fledged features to the cloud to complete the task.

Our solution lies in exploring the *sub-model* structure of DNNs to seek the optimal decision in response to the varying network condition. The architecture and deployment of the DNN are determined on the fly at the time of inference. In the above example, we could have chosen to propagate the neural network on the device until the time point that the network connectivity recovers, and then offload the rest computation to the cloud. If the network connectivity does not get better, we can choose to run a compressed model on the device. By taking context into consideration, we make more precise offloading decisions for DNN inference.

Overview. To enable faster and more accurate DNN inference on the edge, we design a DNN structure and placement decision engine as shown in Fig. 2. The decision engine is trained offline to meet customized accuracy and latency requirements under various contexts. More precisely, the decision engine learns the strategy to cope with the varying context by transforming each DNN block and configuring its placement. The two strategies are *compression* and *partition*. The transformation result is stored as a data structure called *model trees* such that each branch (from the root to the leaves) represents a complete DNN model.

At the online phase, if the user runs the DNN inference API, the decision engine fires to decide which branch of the

model tree as the inference model. Before running each block, the engine decides whether the block should run on the device or on the cloud. If no offloading takes place, before running the following block in sequence, the decision engine makes a decision about the following block based on the current network condition. The procedure repeats until all computation is transferred to the cloud or inference is done on edge. To sum up, the decision engine selects which branch to switch to at each fork of the model tree, and each selection it makes composes a complete DNN model.

Design space. Overall, we aim to design a decision engine to learn the strategy of *transforming* the DNN model and *placing* blocks to different platforms to reach the sweet spot in the trade-off between accuracy and latency. Moreover, we make searching highly efficient by using a predefined model tree structure. A branch of the tree represents the most suitable DNN model within the given context.

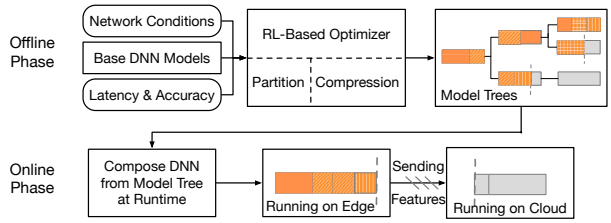

Fig. 2. Reinforcement learning based decision engine for searching the optimal DNN structure and placement.

V. A Reinforcement Learning based Decision Engine

In this section, we introduce our decision engine which transforms the DNN model and decides where to place the model blocks. For simplicity of discussion, we assume the network connectivity remains constant throughout the inference period for now. We first model the DNN transformation and placement as a Markov Decision Process which gradually modifies the DNN layer by layer. Then we introduce the accuracy and latency objective. The detailed method will be given in the end.

A. Markov Decision Process

We formally define the model searching problem as a Markov Decision Process (MDP). We aim to search for the optimal DNN configuration strategy π across the edge and the cloud in accordance with the current network context. We learn such a strategy through a MDP model $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$ where its major components are defined as follows.

State: We consider the DNN model with its configurations (in terms of partition and compression) as the state $s \in \mathcal{S}$ which can be switched from one to another should an action be taken. We explicitly express a DNN layer as a sequence of its hyper-parameters. For example, the i-th layer can be denoted as

$$x_i = (l, k, s, p, n) \tag{1}$$

where l, k, s, p, n respectively represent layer type, kernel size, stride, padding, and the number of output channels. This formulation can be easily extended to include other hyper-parameters, for example, the starting and terminal layer of a skip connection in ResNet. Since a DNN layer is described by a string, we can use a sequence of strings to denote the state of an entire DNN model.

Action: An action $a \in \mathcal{A}$ transfers one state to another. There are two kinds of actions involved in this paper to configure a DNN model. One is *partition* — dividing the DNN into parts running respectively on the edge and the cloud. The other is *compression*, transforming one DNN model to a more compact one.

Policy: π decides which action to choose under a given state. Formally, $\pi : \mathcal{S} \to \mathcal{A}$. Here we adopt a stochastic policy, where $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$.

Transition Probability: The transition probability $\mathcal{P}$ records the probability of transition from one state to another after taking an action. In this project, all the probabilities are deterministic since every action definitely changes the state.

Discount Factor: γ represents how a reward's importance decays given a series of rewards. We set $\gamma = 1$ to make each reward contribute equally to the final return.

Reward: Reward $r : \mathcal{S} \mapsto \mathbb{R}$ stands for the gain of a state. However, we do not assign rewards to intermediate states. The reward is only calculated for the final DNN state when both partition and compression are done.

B. Reward Function

We take a combination of the accuracy and latency measurement as the reward for each state. We explicitly define these two metrics as well as the reward function.

Without loss of generality, we consider a classification task as an example, but similar strategies can be derived for other tasks. Let the dataset consist of m examples: $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ where $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ denotes the set of input images and $\mathbf{y} = \{y_1, \ldots, y_m\}$ is the corresponding true label. The parameters of the DNN models at the edge and the cloud are respectively θ_e and θ_c. Hence the **accuracy** is defined as

$$\begin{aligned} A &= 1 - \sum_i 1[\hat{y}_i \neq y_i]/m \\ &= 1 - \sum_i 1[f_c(f_e(\mathbf{x}_i; \theta_e); \theta_c) \neq y_i]/m. \end{aligned} \tag{2}$$

Note that as long as the DNN is composed by θ_e and θ_c, accuracy has nothing to do with where we partition the model parameters between the edge and the cloud.

The **latency** of running the inference algorithm consists of three parts: the running time on the edge T_e, the transfer latency T_t, and the running time on the cloud T_c. It can be expressed as:

$$T = T_e + T_t + T_c. \tag{3}$$

We assume the size of the final result is so small that the latency of transferring it back to the edge can be ignored. We

indeed can measure latencies by real-world experiments but found it extremely inefficient and inaccurate. On the contrary, a proper estimation model can quickly give an approximate latency. Thus we adopt the estimation method and show how to estimate the latency for each part.

Computational latency T_e, T_c. The running time depends on the computational platform, as well as the total number of multiply-accumulate operations (MACCs) [24]. Most MACCs spent for DNN inference lie in two types of layers — convolutional (Conv) layers and fully-connected (FC) layers. Other layers like batch normalization layers, pooling layers, and drop-out layers cost little time according to our measurement and can be ignored.

Hence we approximate the total number of MACCs as the sum of MACCs in the Conv and FC layers, and the MACCs for each type of layer can be calculated as follows:

$$\#\text{MACC}_{conv} = K \times K \times C_{in} \times C_{out} \times H_{out} \times W_{out}, \quad (4)$$

$$\#\text{MACC}_{FC} = C_{in} \times C_{out}, \quad (5)$$

where K represents the kernel size, H_{out}, W_{out} are the height and width of output feature, and C_{in}, C_{out} denote the number of input and output channels of the layer.

Our experimental results on a number of devices reveal the linearity between the number of MACCs and computational latency. For FC layers, the coefficients between the MACCs and the computational latency are the same for the same device, whereas the coefficients differ by kernel sizes for Conv layers. The linearity between computational latency and the number of MACCs is salient on CPU-based platforms like smartphones, but obscure on GPU-based platforms because of their parallel executions. However, a rough estimation of the computational latency is enough because our work is not specific to any particular estimation model. For a real-world evaluation of such a latency model, please refer to Sec. VII.

Transfer latency T_t. Frequently-used file transfer protocols send packages in the pipeline, and thus the transfer latency can be divided into two parts — propagation delay for the first package and transmission delay for the rest of the file. When the file size is not too large, these two delays can be considered approximately linear to the file size.

We represent the latency for the first package's propagation as a linear function of file size given bandwidth, and transmission delay as file size over bandwidth. Formally, we adopt the following model for estimating the transfer latency:

$$T_t = f(S|W) + \frac{S}{W}, \quad (6)$$

where S means the file size measured by bytes, W means bandwidth and $f(\cdot)$ is a linear function of S given W. We conduct a series of experiments to fit function $f(\cdot)$ to estimate transfer latency based on S and W. Please refer to Sec. VII for an evaluation of the model.

Given the above definitions about the accuracy and the latency, our reward function is defined as

$$R = N_1(A) + N_2(T), \quad (7)$$

where $N(\cdot)$ are the normalization functions, A is the accuracy and T is the latency. Since the units for accuracy and latency are different, we normalize them in the reward

$$N_1(x) = \frac{x - \min_x}{\max_x - \min_x}, N_2(x) = \frac{\max_x - x}{\max_x - \min_x},$$

to balance the contribution of the two metrics to the total reward. A weight factor can also be applied should we assign different weights to the two metrics.

Algorithm 1 Model Compression and Partition

Input: a base DNN model B, a network bandwidth W
Output: a partitioned and compressed DNN model C

1: **repeat**
2: Initialize partition strategy π_p and compression strategy π_c randomly;
3: Input B, W to the partition search controller, obtain action $a_p \sim \pi_p$, apply it to B and get B_{edge}, B_{cloud};
4: Input B_{edge}, W to the compression search controller, obtain action $a_c \sim \pi_c$, apply it to B_{edge} and get B'_{edge};
5: Concatenate B'_{edge} and B_{cloud} to compose a model C;
6: $R \leftarrow$ the reward of C;
7: Update π_p with a_p, R and π_c with a_c, R;
8: **until** both controllers converge;
9: **return** C with the highest reward

C. Optimal Branch Search

We show in this subsection how to search for a transformed DNN model by two reinforcement learning-based controllers. The algorithm is shown in Alg. 1.

Our algorithm takes in a pair of inputs — a pre-trained DNN model as the deployment target and a pre-defined network context. Here we use a constant network bandwidth to represent the network context. The algorithm feeds the inputs to the partition search controller and gets a partition strategy. By the partition strategy, the DNN model is divided into two parts: the first half remains on edge device and the second half will be uploaded to the cloud. Next, the algorithm feeds the DNN model on the edge to the compression search controller and gets a combination of compression techniques. These compression techniques are applied to the edge DNN model layer by layer. By concatenating the compressed edge half and the unmodified cloud half, we get a candidate DNN model. As the last step, we compute the reward of this candidate by Eqn. (7), and correspondingly update both controllers. The procedure is repeated until both controllers converge, and the candidate with the highest reward is the result. The detailed structure and optimization techniques of controllers used in Alg. 1 will be further described in the next section.

As one can tell, compared to a rigid partition policy, our search engine searches partition as well as compression strategies on a base DNN model. The two strategies interact with each other and enlarge the search space, which leads to a better trade-off. But the method has not considered contexts and is still suboptimal. We will introduce a method in the

next section considering the network context and searching on a new structure called 'model tree'. Compared to model tree, the method in this section works like searching on a particular branch of the tree. So we name it as 'optimal branch.'

VI. Context-Aware Model Tree

We have shown how to search for the optimal DNN configuration strategy under a constant network context. However, searching the optimal strategy under a *varying* network context is more challenging. As we have shown in Sec. IV, it is very likely that the network context fluctuates during inference, and one may regret the decision made at the beginning. To deal with the varying context, we novelly design a model tree by taking advantage of the flexibility of DNN models.

A. Model Tree

Fig. 3 gives an example of a model tree. Formally, each node of the tree stands for a DNN block containing one or a few layers, either directly extracted or transformed from a base DNN block. If we consider K types of network conditions for a total of N blocks, in the worst case, we have a complete tree with N layers and K forks for each node in the tree. The child node at the k-th fork of node i represents that the corresponding block is chosen to append to node i. A DNN model is composed by sequentially adding blocks in accordance with the network condition until reaching the final layer of the tree or reaching a node that transfers computation to the cloud. Hence each branch of the tree, from the root to the leaf, constitutes a valid DNN model.

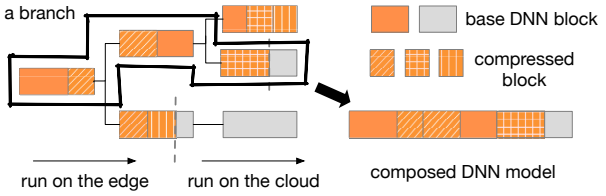

Fig. 3. The composition of a DNN model.

The model tree is designed to take advantage of the flexibility in DNN models — it is possible for several DNN models to share parts of model parameters but also have their distinctive parts, and those models will end up with different architectures and performance. In particular, the sharing mechanism has great flexibility in deciding how much and what to share, if new models are transformed from one base DNN structure. A model tree can be obtained by composing different blocks, where each block is either trained branch by branch or simultaneously. The blocks can be generated by compressing the base DNN models with different compression techniques or growing each from scratch. Once trained, each block becomes 'pluggable' such that a block can substitute another at the inference time depending on its structure and performance. In the online (inference) phase, our decision engine examines the network condition and composes the new DNN block by block. In the worst case, the decision engine iterates through a total of K^N nodes to compose a DNN to run on the edge device. Alg. 2 gives the detail on how to compose a DNN model from the model tree.

Model tree provides us a new dimension of the search space — it depicts the possibility of how a model could vary in time. As we have discussed in the previous section, the optimal DNN configuration found at the static context can be viewed as a local optimum, *i.e.*, the optimal decision made on a branch of the model tree. With varying network contexts, we would like to search for a global optimum by considering the time perspective: each block would have different structures and thus different inference duration; partition and compression decision should not only be made for static network contexts but also for a series of possible states.

Algorithm 2 Composing a DNN Model from a Model Tree

Input: a N-layer K-fork model tree E
Output: a composed DNN model M

```
Initialize an empty DNN model M;
Block B ← root node of E;
Concatenate block B to the model M
repeat
    Measure current network bandwidth, and match it to the k-th branch of B;
    B ← the k-th child block of B;
    Concatenate block B to the model M
until B's child = ∅;
return M;
```

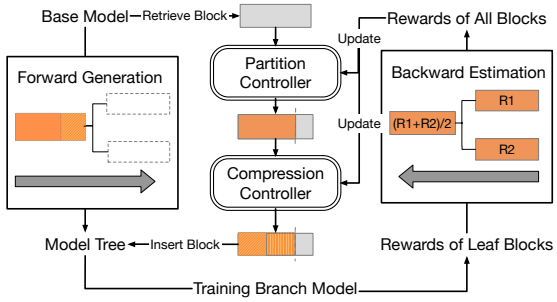

Fig. 4. Reinforcement learning based decision engine for searching the optimal DNN structure and placement.

B. Optimal Tree Search

Following the same reinforcement learning-based framework as in Sec. V, we propose a search framework for model trees. Different from Alg. 1, the input to the algorithm is a DNN block rather than the entire model. However, it is difficult to determine the reward for a single block. In particular, the accuracy of a block is unknown until we compose those blocks into a model. Hence it is challenging to decide the partition and compression strategies for a DNN block.

Fig. 4 illustrates the procedure of searching the optimal strategy on a model tree. Since we take model accuracy into account, rewards can only be calculated after a complete model

is built, or after completing updating a model tree branch. In contrast, the states and actions are updated respectively for each block. Therefore, the reward for each block's actions cannot be obtained immediately, but only after the model tree branch is complete. To address the problem, we propose Alg. 3, a two-stage process — the forward generation stage and the backward estimation stage — to train the controllers.

In the forward generation stage, we generate a model tree from a base DNN model considering various network bandwidths, measure the reward for each branch, and assign it to the leaf block of the branch. In detail, considering N blocks and K kinds of bandwidths, we have a N-depth K-fork empty complete tree as the model tree initially. We traverse every node of the model tree in Breadth First Search (BFS) order. Each node of the model tree is a block transformed from the corresponding block in the base model by the compression controller, with a deployment configuration decided by the partition controller. Note that if we search the partition strategy and compression strategy at the same time, the search space would be huge and one cannot efficiently find the optimal solution. Thus we reduce the search space by searching the partition strategy first and then the compression strategy. Once a partition occurs on one block, its following blocks are directly inherited from the base DNN model, since we do not need to compress model on the cloud. At the end of this stage, we concatenate a model for each branch of the model tree, measure its reward to assign to the leaf block of each branch.

In the backward estimation stage, we compute the reward for actions on each block from leaf to root by averaging rewards of its child blocks. Each branch has a reward. However, there is no clear definition for the reward of the shared blocks. In our work, we assign a parent block's reward as the average of its child blocks' rewards, and the reward for each block is computed in a backward fashion from leaves to the root. After the reward for each block is computed, we update the partition and compression controllers with action-reward pairs in the episode and start another episode.

In summary, we run a latent reward-assigning mechanism to train our controllers. Actions are taken in the forward generation stage to produce a model tree, whereas rewards are assigned in the backward estimation stage.

Algorithm 3 Model Tree Search

Input: a base DNN
Output: a model tree E

Initialize partition strategy π_p and compression strategy π_c randomly;
Slice the base DNN into blocks;
Initialize an empty complete tree E with cloud flag 0;
repeat
  for j-th-layer k-th-fork node i of E in BFS order **do**
    Skip this iteration if node i has cloud flag $= 1$;
    $B_j \leftarrow$ the j-th block of the base DNN;
    $W \leftarrow$ the k-th type of the bandwidth;
    Input B_j, W to the partition search controller, obtain action $a_p^i \sim \pi_p$, and apply it to B_j;
    Input B_j, W to the compression search controller, obtain action $a_c^i \sim \pi_c$, and apply it to B_j;
    Insert B_j to E at the position of node i ;
    $R_i \leftarrow 0$;
    if node i is a leaf node of E **then**
      Get B_j's ancestor blocks $B_{1:j-1}$ from the 1-st to $(j-1)$-th layer of E;
      Concatenate $B_{1:j-1}$, B_j to compose a DNN;
      $R_i \leftarrow$ reward of the new DNN;
    end if
    if B_j has a partition **then**
      Get B_j's following blocks $B_{j+1:N}$ from the $(j+1)$-th to N-th blocks of the base DNN;
      Insert $B_{j+1:N}$ to E as the child of B_j;
      Mark the cloud flag of $B_{j+1:N}$ as 1;
      Get B_j's ancestor blocks $B_{1:j-1}$ from E;
      Concatenate $B_{1:j-1}$, B_j and $B_{j+1:N}$ as DNN;
      $R_i \leftarrow$ reward of the new DNN;
    end if
  end for
  for node i of E in reversed BFS order **do**
    if node i has parent z whose cloud flag $= 0$ **then**
      $R_z \leftarrow R_z + \frac{1}{K} R_i$;
    end if
  end for
  for each node i of E **do**
    Update π_p with a_p^i, R_i and π_c with a_c^i, R_i;
  end for
until both controllers converge;
return E with the highest reward among all branches

C. LSTM-based Controller

Our reinforcement learning-based controllers adopt Monte-Carlo policy gradient method. With this method, a model is required to approximate the policy. In our work, we utilize a recurrent neural network (RNN), particularly, a bidirectional LSTM, as the model for controllers. This is because RNN's superior capability to search for hyper-parameters and its wide application in neural architecture search.

Structures of the partition and compression controllers are given by Fig. 6. The basic unit is a DNN layer. The controller takes a sequence of layers' hyper-parameters as input. And a DNN layer x_i is fed into a forward LSTM as well as a backward LSTM to compute the corresponding hidden states H_i. The partition controller outputs one action for a block of layers while the compression controller outputs one action for each layer. The logits before the softmax represent the preference of the partition layer or the set of compression techniques that controllers choose over others. The partition controller outputs a_p, the layer to divide the DNN. The compression actions a_c^i for the i-th layer represent a set of compression techniques that the controller decides. Given the produced actions, we are able to transform the structure of a DNN as well as its placement, and move it to the next state.

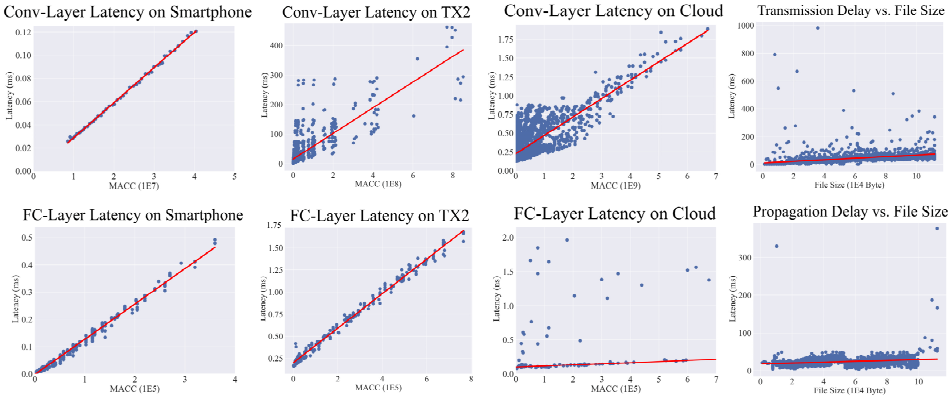

Fig. 5. Estimation model for the computational latency and the transfer latency.

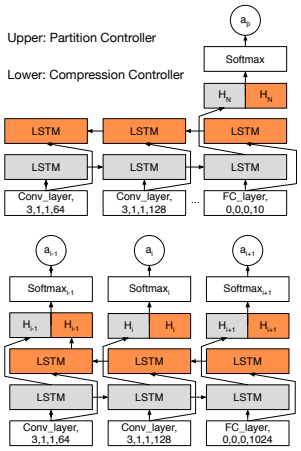

Fig. 6. The architectures of partition and compression search controllers.

D. Optimization

We use the policy gradient method to update π_p and π_c in each episode. The optimization procedure is the same for both policies, thus we use π to represent a policy parameterized by θ. According to the policy gradient theorem, for any differentiable policy $\pi_\theta(s,a)$ and any policy objective function, the policy gradient of the objective function is,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{8}$$

In Monte Carlo (MC) policy gradient, we replace the long-term state-action value $Q^{\pi_\theta}(s,a)$ with its unbiased estimation — total return G. Therefore, for each step in an episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$, we have

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(s,a) G_t. \tag{9}$$

We also apply a common trick, called baseline, to reduce the estimation's variance without affecting its expectation. Policy gradient with baseline is shown as follow.

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(s,a)(G_t - b) \tag{10}$$

Here we choose an exponential moving average of the previous rewards as the baseline function b.

To facilitate convergence, we also adopt the technique of knowledge distillation, *i.e.,* we train each composed DNN with the output logits of the corresponding base DNN instead of ground-truth labels. In this way, the knowledge of the base DNN can be exploited to speed up convergence and enhance accuracy.

VII. Evaluation

In this section, we show that our reinforcement learning based decision engine is able to find the optimal DNN structures and placement across the edge and the cloud in a variety of experimental settings. The results of simulation and field tests are displayed by charts and figures.

Setup. We choose image classification on CIFAR10 as our targeted task. VGG11 and AlexNet are chosen as the base DNNs, whose baseline accuracy is 92.01% and 84.04%. For the edge devices, we adopt two platforms: NVIDIA Jetson TX2 (TX2), which is a GPU-based mobile computational platform, and Xiaomi MI 6X (smartphone) with Android 8.1.0. We test the algorithms in 11 real-life scenes for the smartphone VGG11, 3 for the TX2 VGG11, and 4 for the smartphone AlexNet. As to the cloud, we use two Intel Xeon Processor E5-2630 with GTX 1080 Ti. For model compression, we adopt the most common compression techniques summarized in Table 1.

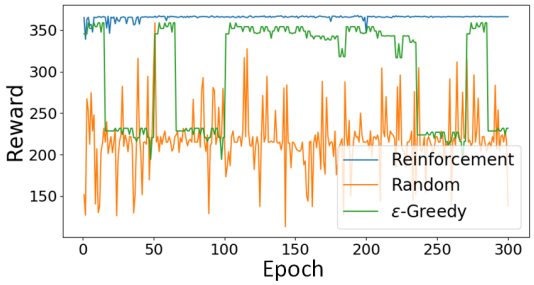

Fig. 7. Comparison of different search methods.

We set the total number of blocks $N = 3$ and the number of bandwidth types $K = 2$. Specifically, we choose the upper quartile and the lower quartile of the bandwidth to represent the 'good' and 'poor' network conditions. For normalization, we set 50% to be the minimal accuracy, 100% the maximal accuracy, 0ms the minimal latency and 500 ms the maximum. The total reward is designed to be 400, where latency and accuracy respectively take up 300 and 100.

Baselines. Our baseline is the method in *dynamic DNN surgery* [5], which finds out the optimal partition for a fixed DNN model under a constant network state by searching the min-cut on a DAG. The method takes the network bandwidth as an input to obtain an optimal cut on the DNN. We also compare the reinforcement learning-based decision engine with other methods, such as random search and ϵ-greedy

TABLE II
COMPRESSION TECHNIQUES

Name	Replaced Structure	New Structure	Applied Layer Types
F_1 (SVD)	$m \times n$ weight matrix	$m \times k$ and $k \times n (k \ll m)$ weight matrices	FC layer
F_2 (KSVD)	same above	same above with sparse matrices	FC layer
F_3 (Global Average Pooling)	FC layers	a global average pooling layer	FC layer
C_1 (MobileNet)	Conv layer	3×3 depth-wise Conv layer and 1×1 point-wise Conv layer	some Conv layer
C_2 (MobileNetV2)	Conv layer	same above with additional point-wise Conv layer and residual links	some Conv layer
C_3 (SqueezeNet)	Conv layer	a Fire layer	some Conv layer
W_1 (Filter Pruning)	Conv layer	insignificant filters pruned Conv layer	Conv layer

search, since an exhaustive search is unaffordable due to the exponentially growing search space.

Latency Model. We conduct a series of experiments to verify that our latency model truthfully reflects the real-world latency. The estimation model is verified on the smartphone, TX2 and the server, and the results of which are shown in Fig. 5. Most of the measured data points fit the model well, except that the latency of Conv-layers on TX2 and the cloud do not strictly follow due to the parallel execution of GPU.

A. Implementation Details

We encounter several problems in the practical implementation of the model tree search. One problem is that the implemented reinforcement learning algorithm always converges to a local optimal solution, for example, partitioning at the first few layers despite varying conditions. Besides that, training from scratch also makes it hard for the reinforcement learning algorithm to find a good solution. We show our countermeasures as follows.

Exploration with fair chances: We find the first problem is mostly because, with randomly initialized parameters, the output of the partition controller approximately follows a uniform distribution. But the uniformly distributed output would result in extremely biased probabilities when exploring different blocks. For example, if the size of a block is L, the output of the partition search controller is a $L+1$-length one-hot vector, with the $L+1$-th one denoting no partitioning takes place. Hence the probability for a block to go without partitioning is $\frac{1}{L+1}$, which is also the chance that the partition controller chooses to explore the next block. Therefore, a block at the n-th layer in the tree has $(\frac{1}{L+1})^{n-1}$ chances of being explored. Hence, the blocks closer to the root are constantly visited but the ones closer to the leaves are rarely explored, which makes our RL algorithm highly biased towards a local optimum in the first few layers.

Our countermeasure is to force the partition controller to assign a n-th layer block with none-partitioning action with $\alpha \cdot \frac{N-n}{N}$ probability, where α is a decaying factor and reduces to zero after the first several episodes. In this way, we allow the controller to explore each block with fair chances, which facilitate the convergence closer to the global optimum.

Optimal branch boosting: To resolve the second issue, we utilize the optimal branch model searched under a static network context. The optimal branch model represents the local optimum for a branch of the model tree. Hence we use the solution as an initial state to boost searching on a model tree. In particular, before composing the model tree, we search for an optimal branch under each type of bandwidth condition, and replace corresponding branches of the model tree with these pre-trained branches. Our algorithm will converge faster in this way.

Training time: We utilize several techniques to reduce the training time to 0.5 2 hours with one GPU. First, we use a latency estimation model instead of measuring on real devices. Second, we implement a memory pool storing the hash code of searched models to avoid redundant computations. Third, we optimize our reinforcement learning controllers with Monte-Carlo policy gradient method, which is simple and low-cost.

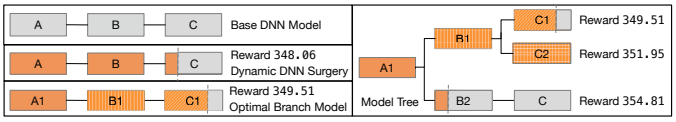

Fig. 8. An illustration of the searching processes by different strategies.

TABLE III
OFFLINE TRAINING REWARD

Device	Environment	Surgery	Branch	Tree
VGG11				
Phone	4G (weak) indoor	353.57	354.29	**355.93**
Phone	4G indoor static	358.90	362.06	**365.64**
Phone	4G indoor slow	354.45	355.94	**357.08**
Phone	4G outdoor quick	360.43	365.99	**368.68**
Phone	WiFi (weak) indoor	359.75	363.94	**365.07**
Phone	WiFi (weak) outdoor	359.25	363.47	**366.53**
Phone	WiFi outdoor slow	357.88	361.77	**363.69**
TX2	4G (weak) indoor	335.94	340.54	**346.33**
TX2	4G indoor static	337.89	343.83	**353.13**
TX2	WiFi (weak) indoor	343.30	347.31	**353.64**
Average		352.14	355.92	**359.57**
AlexNet				
Phone	4G indoor static	348.64	358.54	**359.77**
Phone	WiFi (weak) indoor	341.08	356.59	**359.96**
Phone	WiFi (weak) outdoor	354.34	358.02	**359.61**
Phone	WiFi outdoor slow	344.13	357.42	**358.89**
Average		347.05	357.64	**359.56**

B. Results

1) Offline Training: Since we care about both latency and accuracy, we use the expected reward as a general metric to compare different methods. We train our models and baselines

TABLE IV
EMULATION RESULTS

Result			Reward			Latency (ms)			Accuracy (%)		
Model	Device	Environment	Surgery	Branch	Tree	Surgery	Branch	Tree	Surgery	Branch	Tree
VGG11	Phone	4G (weak) indoor	334.92	**346.48**	344.21	81.83	61.12	64.96	92.01	91.58	91.59
VGG11	Phone	4G indoor static	335.65	340.35	**352.27**	80.62	69.72	50.21	92.01	91.09	91.2
VGG11	Phone	4G indoor slow	326.19	345.63	**345.76**	96.39	60.55	60.42	92.01	90.98	91.01
VGG11	Phone	4G outdoor quick	349.39	354.99	**361.36**	57.71	57.71	31.86	92.01	89.52	90.24
VGG11	Phone	WiFi (weak) indoor	351.85	357.26	**358.71**	53.62	40.45	38.27	92.01	90.76	90.84
VGG11	Phone	WiFi (weak) outdoor	334.66	353.83	**354.03**	82.27	38.67	38.90	92.01	88.52	88.69
VGG11	Phone	WiFi outdoor slow	351.33	356.26	**356.57**	54.48	44.45	43.96	92.01	91.47	91.47
VGG11	TX2	4G (weak) indoor	326.85	328.82	**329.66**	95.28	87.25	85.93	92.01	90.58	90.61
VGG11	TX2	4G indoor static	323.31	330.27	**332.58**	101.18	88.46	84.77	92.01	91.67	91.72
VGG11	TX2	WiFi (weak) indoor	336.36	**344.18**	343.54	79.43	60.78	61.84	92.01	90.32	90.32
Average on VGG11			337.05	345.81	**347.87**	78.28	60.91	56.11	92.01	90.65	90.77
AlexNet	Phone	4G indoor static	342.68	341.73	**343.43**	42.47	44.29	41.42	84.08	84.15	84.14
AlexNet	Phone	WiFi (weak) indoor	348.46	356.87	**357.19**	32.83	19.43	18.88	84.08	84.26	84.26
AlexNet	Phone	WiFi (weak) outdoor	346.68	346.58	**347.15**	35.80	34.97	34.10	84.08	83.78	83.8
AlexNet	Phone	WiFi outdoor slow	339.50	354.49	**354.84**	47.77	19.58	19.10	84.08	83.12	83.15
Average on AlexNet			344.33	349.92	**350.65**	39.72	29.57	28.37	84.08	83.83	83.84

TABLE V
FIELD TEST RESULTS

Result			Reward			Latency (ms)			Accuracy (%)		
Model	Device	Environment	Surgery	Branch	Tree	Surgery	Branch	Tree	Surgery	Branch	Tree
VGG11	Phone	4G (weak) indoor	297.96	319.65	**324.87**	143.44	104.85	98.58	92.01	91.28	92.01
VGG11	Phone	4G indoor static	339.63	344.40	**345.27**	73.99	66.03	64.58	92.01	92.01	92.01
VGG11	Phone	4G indoor slow	296.77	304.92	**319.89**	145.41	131.83	106.89	92.01	92.01	92.01
VGG11	Phone	4G outdoor quick	327.02	335.68	**337.78**	95.00	65.46	77.07	92.01	87.48	92.01
VGG11	Phone	WiFi (weak) indoor	308.19	**325.87**	322.46	126.38	90.71	96.41	92.01	90.15	90.15
VGG11	Phone	WiFi (weak) outdoor	293.21	328.73	**333.16**	151.36	74.82	84.77	92.01	86.81	92.01
VGG11	Phone	WiFi outdoor slow	305.65	312.24	**317.93**	130.62	116.91	107.41	92.01	91.19	91.19
VGG11	TX2	4G (weak) indoor	272.46	323.66	**328.96**	185.93	100.60	91.77	92.01	92.01	92.01
VGG11	TX2	4G indoor static	**323.73**	322.45	323.43	100.49	102.61	100.98	92.01	92.01	92.01
VGG11	TX2	WiFi (weak) indoor	249.94	343.17	**347.81**	223.47	54.42	46.68	92.01	87.91	87.91
Average on VGG11			301.46	326.08	**330.16**	137.61	90.82	87.51	92.01	90.29	91.33
AlexNet	Phone	4G indoor static	351.15	353.12	**353.73**	28.35	25.06	25.91	84.08	84.08	84.64
AlexNet	Phone	WiFi (weak) indoor	257.74	325.12	**329.70**	184.04	73.17	64.10	84.08	84.519	84.08
AlexNet	Phone	WiFi (weak) outdoor	254.43	265.29	**294.71**	189.55	171.46	114.22	84.08	84.08	81.62
AlexNet	Phone	WiFi outdoor slow	277.76	**337.07**	327.07	150.67	46.85	63.52	84.08	82.59	82.59
Average on AlexNet			285.27	320.15	**326.30**	138.15	79.14	66.94	84.08	83.82	83.23

within different real-world network contexts. Fig. 7 shows the model tree search process of our reinforcement learning-based method, random search, and ϵ-greedy search. The context is a 4G indoor environment where the device remains static. The maximum reward our reinforcement learning method finds is 367.70, which is higher than 358.77 found by random search and 358.90 found by ϵ-greedy search.

The complete training results are shown in Table 3. We trained the decision engine under abundant contexts from 4G to WiFi, with weak or normal signals, and we even include three mobility patterns: static, slow, and quick. We also provide results of the optimal branch search, which are obviously inferior to optimal tree search, but superior to dynamic DNN surgery. Above all, our method outperforms the baseline.

Here we present a concrete example to show how our method can improve the performance through adaptation to network condition fluctuation. Fig. 8 compares the results of different methods under the environment of '4G indoor static.' While dynamic DNN surgery provides an optimal partition with a reward of 348.06, the optimal branch search explores the edge-cloud deployment as well as the DNN architecture transformation and thus obtains a higher reward. In our model tree, with the branch boosting technique, branch $A1-B1-C1$ of the model tree reaches the same reward as that of the optimal branch search, guaranteeing the model tree performs at least as well as the optimal branch search. Beyond that, other branches $A1-B1-C2$ and $A1-B2-C$ take full advantage of network condition's resurgence and achieve better rewards. Overall, the model tree gains the highest reward.

2) Emulation: We run emulation tests with real-world network condition traces and estimated latencies. We show the reward, latency and accuracy of each method in Table 4, which illustrates the trade-off between the accuracy loss and latency reduction. Specifically, in tests with VGG11, the model trades 1.35% accuracy loss for 28.32% latency reduction compared with dynamic DNN surgery, and gains 7.88% latency

reduction as well as slight accuracy increase compared with the optimal branch search. As for AlexNet, the model tree reduces 34.33% time consumption with almost no accuracy loss compared with the dynamic surgery, and also 4.06% faster than the optimal branch search. Again, the model tree achieves the highest reward in almost all cases.

3) Field Test: We conduct field tests to examine the real-world performance of our algorithms, and the result is shown in Table 5. The gap between field test results and emulation results comes from the inaccuracy of our latency model and a coarse estimation of network conditions. Despite the gap, our algorithm still shows its superiority in almost all scenarios.

For VGG11, the model tree reduces 36.40% latency with 0.74% accuracy loss compared with dynamic surgery. Its inference speed improves by 3.65% and its accuracy increases by 1.16% compared with the optimal branch search. For AlexNet, the model tree reduces 51.55% and 15.42% inference latency respectively in comparison while keeping both accuracy losses around 1%. However, in cases when the network is good and stable, our algorithm does not have that much advantage over baselines as it is designed for dynamic network conditions after all. For example, in the 4G indoor static case, our method performs equally well with the baseline for VGG11 on TX2.

VIII. Conclusion

To close the gap between the limited computational resource on edge devices and the stringent latency requirement, we present a reinforcement learning-based decision engine to search for a proper DNN configuration with the awareness of runtime context. By taking advantage of the hybrid edge-cloud deployment and flexible DNN architectures, our RL-based decision engine generates a model tree in the offline stage. In the online stage, we dynamically grow a DNN model from the model tree in response to the fluctuating contexts. Evaluation results in real-world contexts reveal that our method provides a favorable trade-off between latency and accuracy — a $30\% - 50\%$ latency reduction at the cost of 1% accuracy loss — which is significantly beyond the baselines.

References

[1] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing under Resource Constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2016, pp. 123–136.

[2] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, p. 615âĂŞ629.

[3] W. Liu, J. Cao, L. Yang, L. Xu, X. Qiu, and J. Li, "AppBooster: Boosting the Performance of Interactive Mobile Applications with Computation Offloading and Parameter Tuning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1593–1606, 2017.

[4] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed Deep Neural Networks over the Cloud, the Edge and End Devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[5] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2019.

[6] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li, "Cost-Driven Offloading for DNN-based Applications over Cloud, Edge and End Devices," *IEEE Transactions on Industrial Informatics*, 2019.

[7] A. Yousefpour, S. Devic, B. Q. Nguyen, A. Kreidieh, A. Liao, A. M. Bayen, and J. P. Jue, "Guardians of the Deep Fog: Failure-Resilient DNN Inference from Edge to Cloud," in *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, 2019, p. 25âĂŞ31.

[8] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2018, pp. 389–400.

[9] T. Huang, R.-X. Zhang, C. Zhou, and L. Sun, "Qarc: Video Quality Aware Rate Control for Real-Time Video Streaming based on Deep Reinforcement Learning," in *2018 ACM Multimedia Conference on Multimedia Conference (MM)*. ACM, 2018, pp. 1208–1216.

[10] N. D. Lane and P. Georgiev, "Can Deep Learning Revolutionize Mobile Sensing?" in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 2015, pp. 117–122.

[11] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*. IEEE Press, 2016, p. 23.

[12] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2017, pp. 82–95.

[13] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "DeepCache: Principled Cache for Mobile Deep Vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2018, pp. 129–144.

[14] N. D. Lane, S. Bhattacharya, A. Mathur, C. Forlivesi, and F. Kawsar, "DXTK: Enabling Resource-Efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit," in *MobiCASE*, 2016, pp. 98–107.

[15] B. Fang, X. Zeng, and M. Zhang, "NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2018, pp. 115–127.

[16] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *International Conference on Learning Representations (ICLR)*, 2016.

[17] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani, "N2N learning: Network to Network Compression via Policy Gradient Reinforcement Learning," *International Conference on Learning Representations (ICLR)*, 2018.

[18] S. Bhattacharya and N. D. Lane, "Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 2016, pp. 176–189.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018.

[21] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[22] M. Lin, Q. Chen, and S. Yan, "Network In Network," *arXiv preprint arXiv:1312.4400*, 2013.

[23] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *International Conference on Learning Representations (ICLR)*, 2017.

[24] S. I. Venieris and C. S. Bouganis, "Latency-driven Design for FPGA-based Convolutional Neural Networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.