EdgeML: An AutoML Framework for Real-Time Deep Learning on the Edge

Zhihe Zhao[†,§], Kai Wang[†], Neiwen Ling[†] and Guoliang Xing[†,*]
[†]The Chinese University of Hong Kong, Hong Kong SAR, China
[§]Duke University, USA

ABSTRACT

In recent years, deep learning algorithms are increasingly adopted by a wide range of data-intensive and time-critical Internet of Things (IoT) applications. As a result, several new approaches, including model partition/offloading and progressive neural architecture, have been proposed to address the challenge of deploying the computation-intensive deep neural network (DNN) models on resource-constrained edge devices. However, the performance of existing approaches is highly affected by runtime dynamics. For example, offloading workload from edge to cloud suffers from communication delays and the efficiency of progressive neural architecture supporting early-exit DNN executions relies on input characteristics. In this paper, we introduce EdgeML, an AutoML framework that provides flexible and fine-grained DNN model execution control by combining workload offloading mechanism and dynamic progressive neural architecture. To achieve desirable latency-accuracy-energy system performance on edge platforms, EdgeML adopts reinforcement learning to automatically update model execution policy in response to runtime dynamics in real-time. We implement EdgeML for several widely used DNN models on the latest edge devices. Comparing to existing approaches, our experiments show that EdgeML achieves up to 8× performance improvement under dynamic environments.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; • **Computing methodologies → Neural networks**.

KEYWORDS

Reinforcement Learning, Edge Computing, Deep Neural Network

1 INTRODUCTION

The recent advances of Internet of Things (IoT) technologies have enabled a wide range of data-intensive and time-critical applications such as autonomous driving [25], embedded computer vision [1], and virtual reality [34]. These applications often require sophisticated processing of large amount of sensor data within stringent time constraints, which motivates the emerging trend of applying Deep Neural Network (DNN) algorithms on edge and IoT devices [24].

In practice, state-of-the-art DNN models [39] incur significant compute overhead, which imposes barrier for deploying them on resource-limited edge and IoT platforms. Moreover, several aforementioned applications are mission-critical in nature, and hence must maintain a high level of model inference accuracy. Recently, a few approaches have been proposed to address these challenges. *Model compression* aims at accelerating the execution of a DNN model locally. For example, filter pruning [23] accelerates the model execution via reducing the convolution calculation, and quantization techniques adopted in [10, 36] use low-precision data representation to speed up the model inference. In *progressive neural architecture* [33], branches are inserted across the layers of the original model such that different input data can exit at different branches during DNN inference, leading to reduced compute workload. In addition to these techniques focused on the optimization of local DNN execution, several approaches are proposed to accelerate the inference through collaboration between edge and cloud. Specifically, part of DNN compute workload can be offloaded to the cloud [17] and such offloading decision can be dynamically adjusted [20, 22] .

However, the above approaches face several major challenges when being applied in real-world applications. First, their performance including timeliness and accuracy is inherently affected by runtime dynamics. For instance, the fluctuations on communication bandwidth between edge and cloud may make the workload offloading strategy sub-optimal, incurring additional inference delay. The changes of data characteristics, e.g., switching from bright scene to dark scene for an image object detector, will cause dynamic end-to-end delays in the progressive neural architecture since the inference path may exit at different branches. Second, existing approaches usually require careful consideration of model architecture as well as optimization of a large set of model parameters, e.g., the model partition point in workload offloading mechanism and compression level in model compression approach. In order to maintain a desirable trade-off between high-level model accuracy and bounded inference latency in the presence of runtime dynamics, such optimization decisions must be made in an *online* manner, which incurs significant compute overhead.

In this paper, we propose a new automated machine learning (AutoML) [9, 38] approach called EdgeML to address these challenges. AutoML is an emerging paradigm that aims to automate the pipeline of DNN design, which has shown promise in several problems such as automated model compression [14] and neural architecture search [41]. Current AutoML solutions are largely focused on automated model design and training on heterogeneous embedded platforms. In this paper, we apply the principle of AutoML to address the challenge of dynamic adaptation of model inference between the edge and cloud. Specifically, EdgeML integrates model partition and progressive neural architecture, which allows to explore the optimal model *execution policy* that consists of branch thresholds in execution control of progressive neural architecture and the model partition point in compute workload offloading mechanism. However, providing a fine-grained model execution control, such an approach leads to a huge design space. Taking DNN model VGG16 as an example, there can be up to 17 candidate model partition point choices, 3 branches and a threshold value at each branch ranging continuously between 0 and 1. An exhaustive search of optimal execution policy

*Corresponding author.

DOI: 10.1145/3450268.3453520

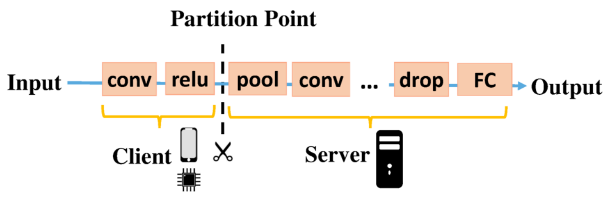

Figure 1: An illustration of DNN model partition and offloading.

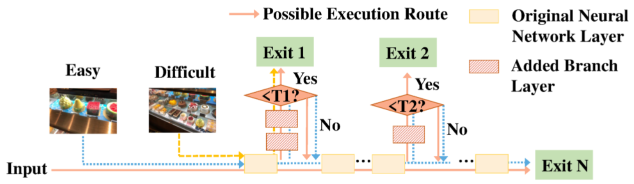

Figure 2: An illustration of the progressive neural architecture.

in this solution space will take hours on off-the-shelf edge devices like NVIDIA Jetson TX2 [31, 41], which makes it impractical for online acceleration of DNN models. To address this issue, EdgeML adopts a new Reinforcement Learning (RL) algorithm as an AutoML framework, which is not only lightweight but also capable of automatically optimizing DNN model execution policy in response to runtime dynamics such as allowing more local compute workload when communication bandwidth drops. Although reinforcement learning has also been adopted in existing AutoML approaches, it is used for automating the offline searching process of DNN model parameters, instead of optimizing the online model execution policy as in the EdgeML design. EdgeML leverages the historical execution performance on the edge to train the reinforcement learning neural network at runtime and hence produces new DNN model execution policies in response to the current environmental conditions. As a result, EdgeML can achieve a desirable performance of model accuracy, edge device energy and end-to-end latency under significant runtime dynamics.

We have implemented EdgeML on latest NVIDIA edge devices and evaluated it with two widely used DNN models VGG16 and Resnet50. Quantitative results show that EdgeML outperforms state-of-art collaborative DNN acceleration solutions by up to 8× on meeting user requirements (i.e., bounds of latency and power) with nearly no loss of accuracy (<0.2%). Moreover, EdgeML incurs little compute overhead, which takes less than 2% of total end-to-end delay of each execution cycle, while providing adaptive model execution control in response to environment dynamics.

The rest of the paper is organized as follows. Section 2 introduces the background of EdgeML, Section 3 describes the system overview of our proposed approach, Section 4 introduces the details of our AutoML design for EdgeML system. Section 5 evaluates the performance of EdgeML and Section 6 reviews the related work. Section 7 concludes the paper and discusses future work.

2 BACKGROUND

2.1 Model Partition and Offloading

To achieve real-time performance and meet energy constraints on edge devices, an efficient approach for accelerating DNN model is to offload the compute workload to the cloud. Fig. 1 illustrates the model partition and offloading for a typical DNN model. The neural network is partitioned into two parts, and executed on the edge and cloud respectively. As an advantage, workload offloading supports flexible execution of a DNN model via adjusting the model partition point and hence provides more opportunities to meet the required runtime system performance. To adapt to the edge-cloud communication channel condition, dynamic workload offloading mechanisms have been proposed [11, 40], in which the amount of transmitted data depends on how the DNN model is partitioned.

2.2 Progressive Neural Architecture

In progressive neural architecture (also known as branched model), the DNN inference does not necessarily go through all the layers. This approach enables the DNN inference to exit at a branch (i.e., early-exit), inserted to the original model [33], leading to a reduction in model execution time. Such a design exploits the fact that different input data has different "difficulty" level of processing (i.e., input complexity) and the inference can stop early for easy input while still being able to produce desired output.

Each inserted branch in the progressive neural architecture is a classifier consisting of a few layers, which only requires a small amount of additional memory space and compute time, compared to the original DNN model. When reaching a branch during feed-forward inference, the confidence of the classifier is measured by the entropy of the classification results at the branch [33]. Meanwhile, if this confidence does not exceed a predefined threshold value, the DNN inference terminates at the branch, known as *early-exit*. Therefore, the expected model inference time depends on the characteristics of input data and the thresholds. The threshold at each branch controls the *exit rate*, which is the percentage of executions exited from the branch. We refer to the original neural network as *main branch* and the inserted layers as *side branches*.

Fig. 2 illustrates the data flow in a branched model. The shortest execution route corresponds to the first side branch, and it will be firstly selected for execution once input data is fed into the model. The result confidence of this route will then be calculated and compared with the assigned threshold to decide if the execution can be early completed. This progressive neural architecture enables easy input to exit at an early branch (e.g., Exit 1 in Fig. 2), avoiding the compute workload of the remaining layers or branches. Moreover, this progressive neural architecture can adapt to dynamic input with variable input complexity via adjusting the thresholds at branches.

3 SYSTEM OVERVIEW

We propose EdgeML to accelerate real-time deep learning task on edge devices so as to fulfill user requirements on task latency, accuracy and edge energy consumption. EdgeML uses an AutoML framework to collaboratively combine the model partition and the progressive neural architecture to reduce the compute overhead on edge devices while achieving desirable latency-accuracy-energy trade-off under dynamic environmental conditions.

The main challenge of EdgeML's design is to adapt to runtime dynamics. Firstly, the communication channel quality highly affects

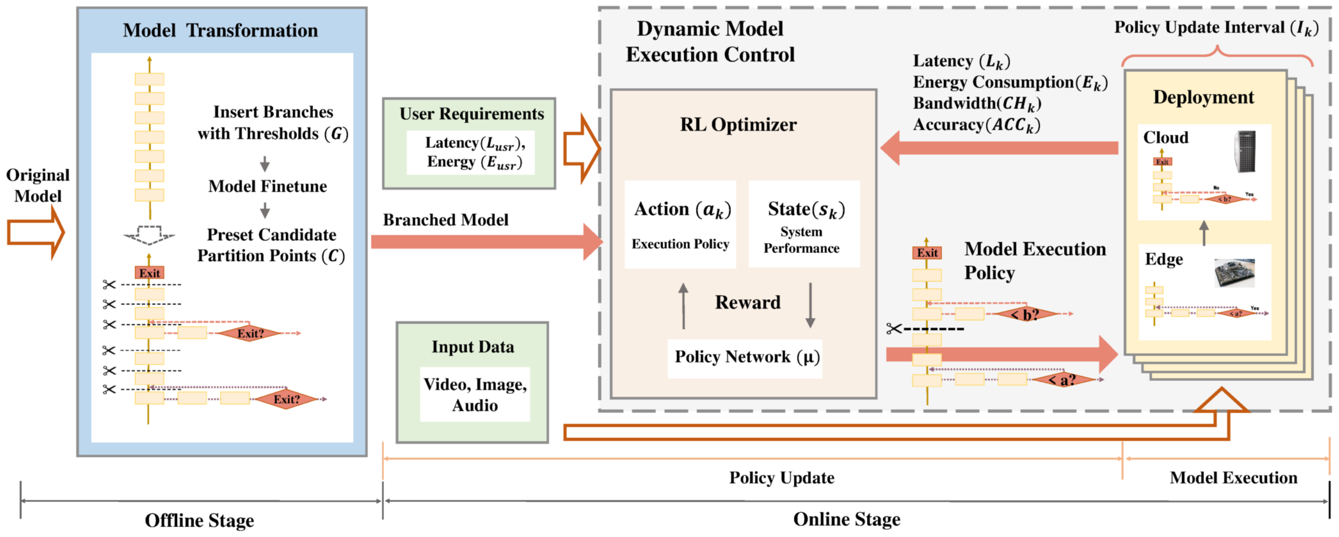

Figure 3: System overview of EdgeML.

the data transmission delay of model partition and offloading mechanism. Secondly, in the progressive neural architecture, the expected model inference time depends on the characteristics (or complexity) of the input data. For example, switching from bright scene to dark scene for image input might cause a longer inference time. Considering the communication dynamics and unpredictable characteristics of the data input, EdgeML aims to jointly optimize the model partition and offloading decisions and early-exit controls to meet the performance requirements of users. Specifically, our problem can be defined as follows.

$$\max\ Accuracy(I)\ \ s.t.$$
$$Latency(I) < L_{usr},\ Energy(I) < E_{usr} \tag{1}$$

where I is a specified time interval, and the accuracy $Accuracy(I)$, task latency $Latency(I)$ and energy consumption $Energy(I)$ are averaged over the processed tasks within time interval I. Specifically, the latency is defined as the end-to-end task response time and the energy indicates the consumed energy of edge device for computing the task. The user-defined upper bounds on the task latency and device energy consumption are denoted as L_{usr} and E_{usr}, respectively.

The execution of EdgeML system contains two stages, offline and online, as shown in Fig. 3. In the offline stage, the DNN model is transformed in the *progressive neural architecture* and trained locally. Firstly, a *branched model* is constructed by inserting several branches across the layers of the original DNN model. Each branch is further fine-tuned on the cloud to increase accuracy. Afterwards, a series of model partition options are formed by selecting the partition points at the granularity of the neural network layers to support a fine-grained layer-level workload offloading. The branched model and the model partition options are synchronized between the edge and the cloud. The *model execution policy*, which consists of the model partition points and the threshold for each branch will be generated at run-time to determine how the DNN model processes the input data.

In the online stage, EdgeML adopts an AutoML framework to update the *model execution policy* dynamically via the cooperation between the edge and the cloud. Due to the huge solution space for model execution policy and the dynamic environmental conditions, EdgeML adopts a reinforcement learning (RL) algorithm to search for the optimal execution policy at runtime. Each execution policy consists of partition point, threshold at each branch and policy update interval. The new execution policy is generated based on the measured performance of the previous execution policies from the edge, including the task latency, model execution accuracy, edge energy consumption and edge-cloud data transmission rate.

During model execution on the edge, the input data will be processed based on the configured execution policy. Specifically, the inference starts at the main branch until reaching a side branch. The assigned threshold value at the side branch determines whether the execution can be terminated, i.e., whether the output generated at the branch is accepted or not. If not, the execution process continues on the main branch until the next side branch. Moreover, when the execution process reaches the partition point at the main branch, the intermediate inference data from the partitioned layer will be transmitted to the cloud to offload the compute workload of the remaining layers.

Our AutoML framework can also be extended for DNN model execution control across different edge devices that are connected to the could. Specifically, when a new edge device is present instead of training the RL optimizer from scratch, EdgeML can transfer the trained RL optimizer from an existing edge device to the new device, which reduces compute overhead and delay. Directly transferring the trained RL optimizer may cause serious performance drop, because even the same execution policy yields different performance on different edge devices. EdgeML achieves efficient knowledge transfer of RL optimizer via transferring the MDP(Markov decision process) traces to the new device. Since RL learns the execution policies in response to environmental factors instead of platform-specific parameters, such a simple "knowledge transfer" strategy allows EdgeML to accommodate heterogeneous platforms. We evaluate the effectiveness of this design in Section 5.3.3.

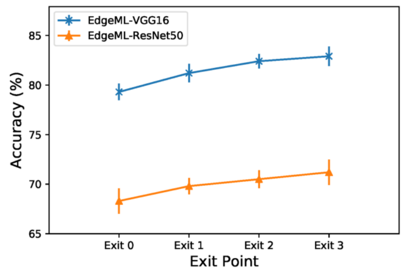

Figure 4: Accuracy of the branches in the branched model.

4 EDGEML DESIGN

In order to meet the performance requirements under variable environmental conditions, EdgeML supports a dynamic DNN model architecture by combining the model partition with progressive neural architecture and hence provides a wide variety of adjustments in DNN model execution. As discussed in Section 2, variation in communication channel condition between the edge and the cloud leads to unpredictable data transmission delay in workload offloading. Furthermore, variation in input characteristics leads to the change of the expected exit branch and compute overhead on the edge. In order to adapt to such dynamic environment conditions, we propose a RL-based AutoML design that automatically updates the execution strategy of the DNN model at runtime.

4.1 Offline Model Transformation

4.1.1 Branch Insertion. The side branches are inserted at equidistant points of compute workload (i.e. FLOPs) across the original neural network layers. We adopt the following rules to design the structure of the side branches, which extends the existing approach [33]. (1) Keep equal gradient of compute workload between the adjacent execution branches by selectively choosing the layer in main branch for branch insertion with the consideration of the additional compute workload incurred by the inserted branch; (2) Include more fully connected layers for earlier branches and fewer layers for later branches; (3) Each inserted branch only consists of fully-connected layers. The purpose for adopting branched model is to provide multiple execution options of the DNN model with different compute workloads and accuracy. As a result, execution branches with similar compute workload should be excluded, which leads to rule (1). For rule (2), partitioning earlier branches in a DNN model may significantly increase the transmission overhead because the intermediate inference results from the first a few layers normally have a large footprint. For rule (3), since fully-connected layers are widely used as a classifier, we use them as the inserted branch, which takes feature maps from the main branch as input and outputs the inference results.

To illustrate the variable model accuracy in each side branch, we conducted experiments with modified DNN model VGG16 [32] and ResNet50 [13], shown in Fig. 4. Based on the original neural network VGG16, we insert 3 branches into the model and there are totally 4 exits. The inserted branches contain 3, 2, 2 fully-connected layers, respectively. Note that the early-exit execution mechanism incurs additional compute overheads, because to exit at a latter branch, all the former branches must have been executed. In other words, the compute workload for each branch includes all executions of the former branches. Thus, we place the branches across the original layers to ensure that the actual compute workload when exiting at each branch is proportional to the accuracy of the branch. From Fig. 4 we can see that later branches with more compute workload yields higher accuracy with almost linear correlation.

4.1.2 Model Partition. To support fine-grained workload offloading, we preset a series of possible candidate model partition points at the equidistant compute workloads across the neural network layers. In our design, the candidate model partition points are selected only on the main branch. If the DNN model is partitioned at a side branch, both the intermediate model inference results of the main branch and the side branch should be offloaded to the cloud in case inference does not exit at side branch, resulting in higher communication overhead and complicated design. For each selected candidate model partition point, we synchronize the corresponding offloading policy between edge and cloud in the offline stage so that the DNN model execution at runtime can be flexibly switched between these offloading policies in real-time.

4.2 Dynamic Model Execution Control

The branched model constructed in the offline stage supports a wide variety of model execution policies at runtime, and hence provides flexible opportunities to meet user requirements on latency, accuracy and edge energy consumption. The purpose of the reinforcement learning algorithm (also refer to as *RL optimizer*) is to provide an automated and adaptive control on model execution policy over time. The RL optimizer uses the collected execution performance data of the historical execution policies from the edge to make near-future decisions, i.e., generating a new execution policy.

We focus on addressing two different types of dynamics that affect the runtime performance of EdgeML. Communication channel quality (i.e., available bandwidth) directly affects the data transmission delay, while the input characteristics (i.e., distribution of easy/difficult input) affects the expected exit branch of model execution. We model the change of environmental conditions, i.e., communication channel quality and input characteristics, as a Markov decision process with unknown transition probabilities. However, the input characteristics cannot be directly measured by the exit branch of each execution, which not only depends on the input data but also depends on the execution policy at runtime (i.e., configured threshold values at the branches). Instead, we use the average latency and energy measured during DNN execution rather than the exit branch to quantify the input characteristics. For example, more difficult tasks in the inputs will lead to the increase in compute overhead on the edge and hence the increase in latency and energy consumption, under the same execution policy and bandwidth. The measured latency, energy and data transmission rate forms the state of the MDP, which is then used to calculate a reward to quantify the performance of the current execution policy. Afterwards, this reward is used as feedback to train the reinforcement learning neural network on the cloud, as well as to produce new execution policy. The new execution policy is updated regularly in order to timely respond to the change of environment conditions and also the change of user requirements. In order to adapt to the rate of environmental

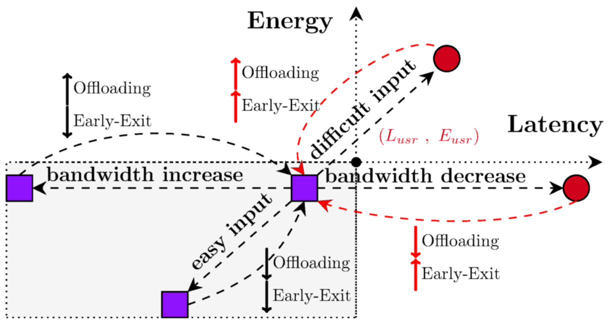

Figure 5: An illustration of how EdgeML adapts to environments. The change in input characteristics and bandwidth will cause the system's transition to a new quadrant with certain actions, such as increasing/decreasing offloaded workload and early-exit model executions.

changes, the update rate of execution policy (which is equivalent to *Action Interval* introduced later) is also learned by the RL optimizer.

Fig. 5 illustrates how RL optimizer responds to environmental changes. The coordinate indicates the average latency and energy consumption for the current execution policy. In particular, the coordinate in the bottom-left quadrant corresponds to a feasible execution policy that meets user requirements L_{usr}, E_{usr}. When bandwidth decreases, the task latency will exceed L_{usr} due to higher transmission delay. Therefore, the RL optimizer aims to find a new execution strategy that has a latter-layer partition point, i.e., more workload at the edge. Meanwhile, to compensate for the increase in compute demand, the new strategy also raises the thresholds at the branches to allow more early-exits during task executions. This will result in system's transition from bottom-right quadrant to bottom-left quadrant, and hence the new strategy can satisfy the latency and energy constraints. Similarly, when the distribution of the input shifts to be more difficult, more workload will be executed on the edge due to the low probability to exit at early branches. Then, the RL optimizer will switch to a new execution strategy that offloads more compute workload to the cloud and at the same time raises the thresholds at some branches to allow more early-exit executions, resulting in system's transition from upper-right quadrant to bottom-left quadrant.

In the following, we define the notation for describing the RL optimizer. Let $\mathcal{C}$ be the set of candidate partition points preset in the offline stage. An *execution policy* (EP) consists of the model partition point $C \in \mathcal{C}$ and the threshold value G_j of the j-th branch in the branched model. The threshold value $G_j \in [0, 1]$ in the branched model controls the *exit rate* (i.e., the percentage of model executions that terminate at this branch) of the j-th branch. The model partition point C determines the offloading policy, which affects both the transmission latency and compute workload on the edge. In summary, the RL optimizer aims to maximize the task accuracy while satisfying the user requirements of task latency L_{usr} and edge energy consumption E_{usr}, as in Eq. (1).

4.2.1 State Space.

The state of RL optimizer is used to estimate the environment conditions. As both the input characteristics and the communication channel quality fluctuate over time and are unpredictable, a single measurement is not sufficient to represent the current state of environments. Therefore, the reinforcement learning

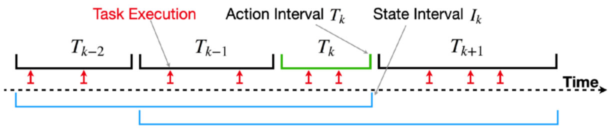

Figure 6: An illustration for reinforcement learning state.

state s_k is defined in the average level during a time interval I_k, which includes the average values (per task) of response time $\mathbf{L}(I_k)$, the energy consumption on edge $\mathbf{E}(I_k)$ and the data transmission rate $\mathbf{CH}(I_k)$ during time interval I_k.

$$s_k = \{\mathbf{L}(I_k), \mathbf{E}(I_k), \mathbf{CH}(I_k)\} \tag{2}$$

The state parameters are observable and measured during execution of all inputs during time interval I_k, which is referred to as *State Interval*. In state s_k, the combination of parameters $\mathbf{L}(I_k)$ and $\mathbf{E}(I_k)$ quantifies the average compute cost of all inputs in interval I_k on the edge. As shown in Fig. 6, the state interval I_k contains several action intervals (i.e., execution policies), where each action only takes effect for a particular number of task executions. Two consecutive state intervals I_k, I_{k+1} have intersections on time domain, implying that they are related in the sense that some action interval occurs in both state intervals. Such a design can smooth the influence of a single action to the whole state, which prevents the RL optimizer from changing the execution policy back and forth.

Note that the parameters $\mathbf{L}(I_k)$ and $\mathbf{E}(I_k)$ also include the task delay and energy consumption caused by the policy update operation during time interval I_k. This will prevent the RL optimizer from frequently adjusting the execution policy when the environment conditions are stable. In other words, apart from seeking for the best execution policy, the RL optimizer also seeks for the best period of time for each updated execution policy.

4.2.2 Action Space.

Each execution policy determines the parameters for the execution of the branched model and we regard each update of execution policy as an action. Each execution policy is assigned with a time period, indicating the time duration that the model is executed under this policy, which is also referred to as *action interval*. Therefore, an action contains a particular execution policy and the corresponding action interval. The k-th action a_k is defined as

$$a_k = \{\mathbf{G}_k, C_k, T_k\} \tag{3}$$

where $\mathbf{G_k} = \{G_j\}$ includes the threshold values for all branches and G_j denotes the threshold value of the j-th branch. The threshold values are normalized, that is $G_j \in [0, 1]$. The model partition point C_k is drawn from predefined set of candidate partition points $\mathcal{C}$ and it corresponds to a layer in the main branch which will be partitioned. Action interval T_k indicates that the action is valid for a time period of length T_k, after which (i.e, expiration) a new action should be triggered. Considering the dynamics of the input, T_k is measured by the number of tasks executed under the execution policy given by the action. In other words, $T_k \in \{1, 2, ..., N_{max}\}$, where N_{max} (e.g., $N_{max} = 5$ in our experiments) denotes the maximum number of executed tasks in one execution policy. To prevent from frequently updating the execution policy, action interval T_k is learned by the RL optimizer.

To avoid inefficient optimization caused by the discrete properties of action space, we consider continuous action space and round down the floating values of partition point C_k and time T_k into integers to generate actions.

4.2.3 *Reward Function.*

For a given action, a reward is calculated to estimate the performance of the execution policy given by the action. Accurate performance estimation requires more input samples that are performed under the current execution policy. However, in practice, to ensure rapid response to environment change, the action interval T_k is normally small. Therefore, it is reasonable to calculate the reward based on the tasks executed during the state interval I_k, instead of action interval T_k. To simplify the calculation, in our design, the state interval I_k is defined to be the union of the most q recent action intervals, i.e. $I_k = T_k \cup T_{k-1} \cup ... \cup T_{k-q+1}$. With this design, for any two consecutive states s_k, s_{k+1}, the two state intervals I_k, I_{k+1} will have intersection and hence connected.

The reward $R_k(s_k, a_k)$ is calculated based on the average task accuracy ACC_k, average task latency L_k and average energy E_k, measured in state interval I_k.

$$R_k(s_k, a_k) = \begin{cases} \text{BE}_k & L_k < L_{usr},\ E_k < E_{usr} \\ -\gamma & L_k > L_{usr},\ E_k > E_{usr} \\ -\frac{2}{\pi}\arctan(pnl_k) & else \end{cases} \tag{4}$$

Intuitively, we give positive reward when the performance of the action (i.e., the current execution policy) satisfies both latency and energy requirements, and negative reward otherwise. For the first condition, when latency and energy both meet user requirement, we aim to optimize the average task accuracy. Due to the fact that it is difficult to evaluate the actual task accuracy for a real-life input, we use the index of the exit branch of the input to quantify the accuracy of the task execution. As shown in our experiments in Section 4.1.1, the execution of an input that exits from an earlier branch has a lower accuracy. Formally, BE_k indicates the expected exit branch for a task under the current execution policy, which is calculated by averaging the indexes of exit branches over interval I_k of the task executions that satisfy the requirements of latency and energy (i.e., first condition in reward function). Task exit rates from an earlier branch can be controlled by its threshold, so that the RL optimizer can make trade-off between accuracy and performance constraints (i.e. latency and energy). In other words, the reward BE_k will guide the RL optimizer to decrease the threshold value so that task execution are more likely to exit from a longer branch, under the premise of satisfying the requirements of latency and energy. For the second condition where both requirements are not satisfied, we penalize this incorrect outcome with $-\gamma$, which we set $\gamma = 1$ in order to ensure that the degree of punishment is consistent with the reward.

For the last condition, when only one user requirement (i.e., latency or energy) is satisfied, we define the reward in this situation as $-\frac{2}{\pi}\arctan(pnl_k)$ where pnl_k is defined in Eq. (5) and it represents the difference between the current latency/energy and the user expected requirement. We use $\arctan(\cdot)$ function to map this difference to fit the expected varying trend, and then multiply it with $\frac{2}{\pi}$ for normalization. Note that there are no weight coefficients for latency and energy requirements here, since they share the same order of magnitude as shown in our experiments. When pnl_k is small, i.e., the requirements are nearly satisfied and the penalty is also small. On the contrary, if pnl_k is large, it is bounded by 1 due to the nature of $\arctan(\cdot)$ function and such a design allows the RL optimizer to explore Pareto Optimal strategies [1] since one user requirement is satisfied.

$$pnl_k = \begin{cases} L_k - L_{usr}\ , & \text{if } L_k > L_{usr},\ E_k < E_{usr} \\ E_k - E_{usr}\ , & \text{if } L_k < L_{usr},\ E_k > E_{usr} \end{cases} \tag{5}$$

4.2.4 *RL Agent.*

Due to the huge action space, we apply a variant of deep deterministic policy gradient (DDPG) algorithm [26] to generate action, which is based on the 'critic-actor' framework. The 'actor' model, combined with a random process (Ornstein-Uhlenbeck process) to allow action exploration, generates action a_k based on a given state s_k by a policy network μ as follows.

$$a_k \sim \mu(s_k) + OU(v, \sigma^2) \tag{6}$$

Similarly, the 'critic' model simulates function Q to estimate the quality of action a_k in form of Bellman function as below.

$$Q\left(s_k, a_k\right) = \mathbb{E}\left[R\left(s_k, a_k\right) + \gamma Q\left(s_{k+1}, \mu\left(s_{k+1}\right)\right)\right] \tag{7}$$

Action value function Q is the expectation over the distribution of the current reward $R(s_k, a_k)$ and the estimated future reward by executing the policy μ sequentially over the period episodes. With the calculated reward $R(s_k, a_k)$ and the observed state s_{k+1} caused by action a_k, the actor model aims to generate new action $a_{k+1} = \mu(s_{k+1})$ such that the new expected reward caused by a_{k+1} is maximized, where the new expected reward $Q(s_k, a_k)$ is calculated by the critical model Q according to Eq. (7). With the feedback from the reward function $R(s_k, a_k)$, both the actor model and the critical model are trained in a back propagation process from the approach in [26].

4.3 Model Execution

Upon receipt of each RL action from the cloud, the edge updates the corresponding execution policy of the branched model according to the action. During model execution, the side branches will be executed orderly once the input data is fed into the branched model on the edge, until execution exits at some branch or partition point is reached. For exit point at each branch, the softmax and entropy of the output will be calculated based on the method from [33] to measure the confidence level of the classifier of this exit point on the input. If the calculated confidence does not exceed the predefined threshold value (i.e., the result generated at the branch has high probability to be correct), the execution will be terminated and the output result from this branch will be returned as the model output. Otherwise, the execution resumes on main branch and continues to the next branch. When a partition point is reached and no output is returned, the edge device then transmits the intermediate inference results from the partitioned layer to cloud for computing the remaining layers of the model. The execution in cloud is similar as on the edge device because they share the same branched model and execution policy, while the execution time in cloud is much shorter than that on edge device.

[1] Here it means the strategy that satisfies one user requirement and meanwhile optimizes the other user requirement.

Cross-platform adaptation. When deploying EdgeML to a new edge device, training the RL optimizer from scratch requires collecting many training samples from the new edge device, which leads to poor transit performance of EdgeML during this period. To address this issue, we transfer the trained RL optimizer from an existing deployed edge device to the new device. Intuitively, directly transferring the trained RL optimizer onto the new device for automatically model execution control may cause serious performance drop. This is because the same generated execution policy from RL optimizer might yield totally different performance on different edge devices due to the diversity in computing power and energy consumption profiles.

To achieve more efficient knowledge transfer, we collect the MDP traces of *state, action, reward, subsequent new state* during the training process of the RL optimizer from the existing deployed edge device, as described below.

$$[s_k, a_k, R(s_k, a_k), s_{k+1}] \tag{8}$$

We then mix these (additional) MDP traces with the collected MDP traces from the new edge device together as training samples to train the RL optimizer for the new edge device. These additional MDP traces contain the knowledge of model execution control learned from the existing deployed edge device, which can be adopted on the new device and hence speed up the training process of EdgeML.

5 EVALUATION

We demonstrate the effectiveness of EdgeML by comparing with several baselines and ablated variants of EdgeML. Moreover, we demonstrate the adaptivity of EdgeML to environment changes, including bandwidth and input characteristics. Finally, we analyze the system overhead of EdgeML. The source code of EdgeML is available at: https://github.com/Kyrie-Zhao/EdgeML.git.

5.1 Experimental Setup

Our edge platforms include NVIDIA Jetson TX2 and NVIDIA Jetson Nano [31]. A cluster configured with a NVIDIA GeForce GTX1070 GPU and 8-core 2.80-GHz Intel i7-7700HQ CPU served as the cloud. To mimic the dynamic edge-cloud communication channel conditions, we use the network tool *wondershaper* [3] to adjust the network bandwidth between the cloud and the edge in real-time.

We have developed EdgeML on top of TensorFlow. For RL optimizer of EdgeML, we adopt a three-layer neural network architecture to construct both the actor network and the critic network. For each RL action, the generated execution policy is executed on the edge for one epoch, after which a new execution policy is required. The number of images processed in one epoch is also referred to as the *interval*. The RL state is computed at the end of each epoch based on the measured performance during the most recent 3 epochs, i.e., one state interval contains 3 epochs. To evaluate the generality of EdgeML, we choose two representative DNN models (i.e., VGG16 and ResNet50) as original neural network, and select two datasets that are widely used for mobile vision (i.e., CIFAR-10 [19] and CIFAR-100 [19]).

Evaluation Metrics. We use four metrics to evaluate the performance of EdgeML: accuracy, latency, GPU frequency and Satisfaction Rate (SR). *Accuracy* is the top-1 accuracy of the model execution and *latency* is measured as the end-to-end delay in processing an input (e.g., an image). We use *GPU frequency* to quantify the power of edge device, which is reasonable because there usually exists approximately linear correlations between power and frequency [7, 35]. We use *SR* to represent the percentage of executions that satisfy the latency and energy constraints imposed by users. Note that for all the following experiments, the user bounds on latency and GPU frequency are set as 0.1s and 1.2 GHz, respectively.

Baselines. In order to validate the effectiveness of the EdgeML, we compare the performance of EdgeML with three types of baselines: *1) Ablated EdgeML.* We consider three ablated variants of EdgeML, denoted as EdgeML-T1, EdgeML-TL and EdgeML-NonBranch respectively. *Edge-NonBranch* refers to EdgeML without an early-exit mechanism, which means that no branch is added to the original model. In *EdgeML-T1*, the threshold values at all side branches are rounded down to be the same as the first branch. Similarly, for the ablated variant *EdgeML-TL*, the threshold values at all the side branches are rounded up to be the same as the last side branch of EdgeML. EdgeML-T1 and EdgeML-TL represent a typical static early-exit approach where the decision of exiting different layers is controlled by a single threshold [20]. *2) Neurosurgeon.* To reflect the advantage of dynamically adjusting execution policy in EdgeML, we compare EdgeML with the approach that based on static workload offloading mechanism (i.e., Neurosurgeon [17]), which is one of the state-of-the-art DNN offloading frameworks. Referring to the design of Neurosurgeon, we iterate all the partition points at run-time and choose the best offloading policy that achieves a desirable satisfaction rate and overall accuracy for execution. *3) MOEA.* We design a baseline based on multi-objective evolutionary algorithm (MOEA) to evaluate the effectiveness of RL optimizer in EdgeML. MOEA baseline adopts the Non-dominated Sorting Genetic Algorithm III (NSGA-III) [5] to replace the RL optimizer in EdgeML. NSGA-III has been shown effective in finding efficient solutions for a wide range of complex nonlinear optimization problems and has been successfully applied to real-world engineering problems [30]. To apply NSGA-III in our problem, we transform the energy and latency constraints into objective function, i.e., minimizing latency and on-device power consumption while maximizing the accuracy.

5.2 Overall Performance Analysis

5.2.1 Model and Platform Applicability. In this section, we demonstrate the effectiveness of EdgeML for different DNN models and applications scenarios. We choose three applications, each of which (i.e., VGG16-CIFAR19-TX2) consists of model type, dataset and edge platform. We compare the overall performance of EdgeML with baselines introduced in Section 5.1. Each baseline is evaluated for 5000 epochs, the mean and variance of the experiment results are presented in Fig. 7.

We observe that compared with baselines Neurosurgeon and MOEA, EdgeML achieves a satisfaction rate 1.43×, 8× better on VGG16-CIFAR10-TX2, 4×, 2.62× on ResNet50-CIFAR100-TX2

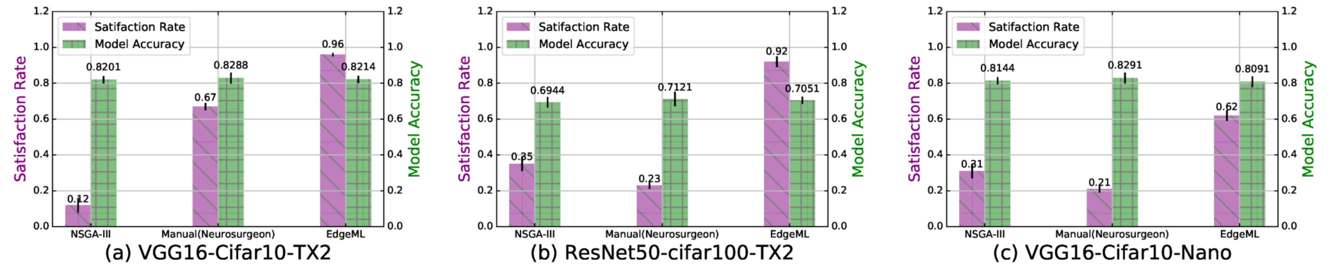

Figure 7: Overall performance of EdgeML and baselines.

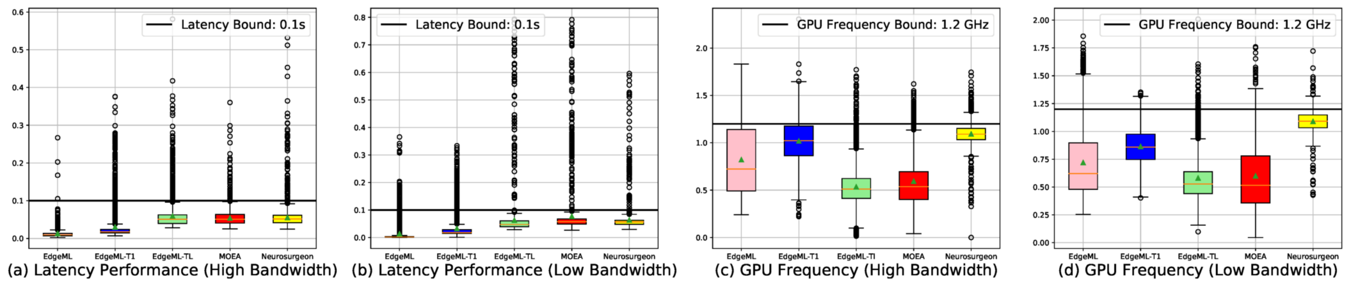

Figure 8: Comparison performance of latency and on-device power consumption

and 2.95×, 2× on VGG16-CIFAR10-Nano. However, we also observe that the accuracy achieved by EdgeML is slightly lower than the other two baselines. For example, EdgeML yields 0.7% lower accuracy than that of Neurosurgeon on ResNet50-CIFAR100-TX2. This is because EdgeML searches for the most effective strategy that trades accuracy for latency to meet user requirements. In general, EdgeML has better performance regardless of DNN model type, dataset and edge platform.

5.2.2 Bandwidth Adaptability. To demonstrate the effectiveness of EdgeML under various communication conditions, we compare the performance of EdgeML to four baselines, MOEA, Neurosurgeon, EdgeML-T1 and EdgeML-TL under the settings of high bandwidth and low bandwidth, respectively. We set the high bandwidth to about 15 Mbps and low bandwidth to about 6 Mbps, each alternating for 500 epochs, and we show the statistical results of five alternates.

We firstly demonstrate the effectiveness of the RL optimizer in EdgeML by comparing it with MOEA and the effectiveness of dynamic workload offloading adopted in EdgeML by comparing it with Neurosurgeon. Fig. 8 shows the comparison results of latency and GPU frequency performance across high and low bandwidth periods. We make two observations. First, EdgeML shows the most robust performance against different levels of bandwidth, since the results above the user bounds are significantly fewer than MOEA and Neurosurgeon. Second, It can be observed from Fig. 8 (c) (d) that MOEA shows some adaptability especially on GPU frequency, while sacrificing significantly on latency performance, This is because that the action space for MOEA to search is too large, and hence can easily converge to local optimal when the environment does not change.

During low bandwidth periods, both EdgeML-T1 and EdgeML-TL can adapt to the bandwidth variations, with fewer results above the user bounds. However, from Fig. 9 (c), the average accuracy achieved by these two baselines are lower than EdgeML. It is also

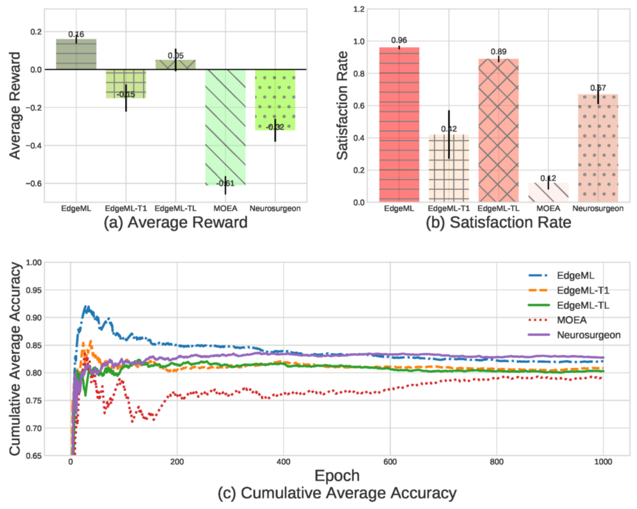

Figure 9: Comparison of average rewards, satisfaction rates among EdgeML, EdgeML-T1, EdgeML-TL, Neurosurgeon and MOEA models

observed that EdgeML-TL generally performs better than EdgeML-T1 and worse than EdgeML.In general, the results show that the adaptive setting of multi-threshold mechanism adopted by EdgeML outperforms the "one-threshold-for-all" mechanism.

Fig. 9 (a) and (b) show the comparison of average reward and average satisfaction rate among these baselines. We can see that EdgeML significantly outperforms other baselines. It is also observed that the average reward is basically positively correlated to the average satisfaction rate. Therefore, the achieved high reward is directly translated into the system performance improvement at runtime. Fig. 9 (c) shows the comparison of the average accuracy.

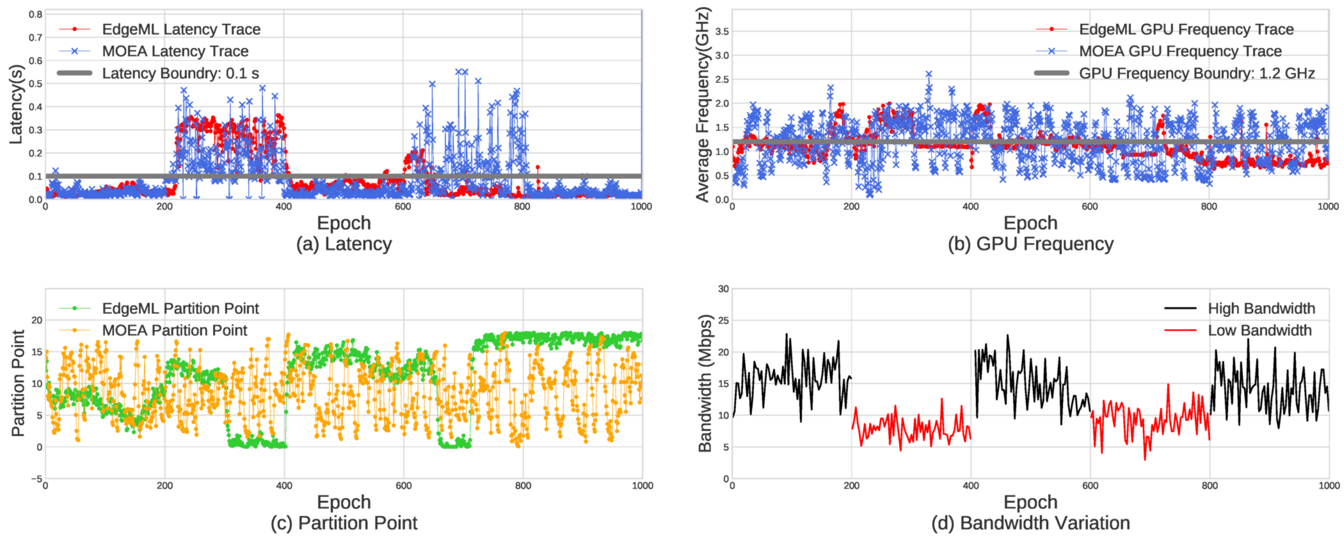

Figure 10: EdgeML system performance and generated actions versus time

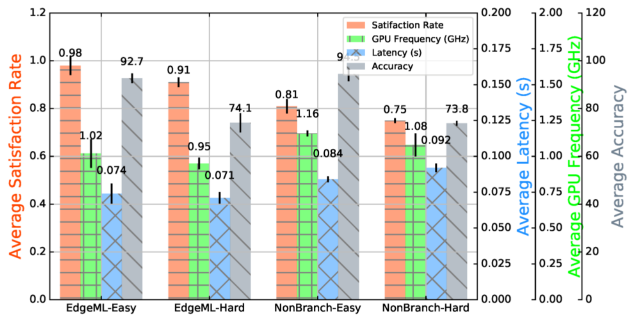

Figure 11: Performance of EdgeML and baseline EdgeML-Nonbranch over easy and hard sub-dataset (VGG16-CIFAR10-TX2)

We observe that compared to Neurosurgeon, which does not support trading accuracy for latency, EdgeML only sacrifices <0.2% accuracy. At the same time, EdgeML yields the highest accuracy compared to other baselines. Note that since we measure the accumulative average accuracy in this experiment, the traces basically maintain stabilized after 1000 epochs.

5.2.3 Impact of Input Variation. In this part, we demonstrate the adaptive performance of EdgeML in the presence of changes of input characteristics. We manually split the initial CIFAR-10 dataset into two sub-datasets: *easy* and *hard*. Specifically, we use VGG16 to quantify the difficulty level of each sample in the dataset. Among all correctly classified instances, according to the value of confidence, we categorize the top 60 percent of them as the *easy*. The remaining instances in the dataset, including those with wrong labels, are categorized as the *hard*. We validate the performance of EdgeML by comparing it with EdgeML-NonBranch on the easy and hard sub-datasets. Fig. 11 shows that, in terms of accuracy, EdgeML outperforms EdgeML-NonBranch by 1.2× on the easy dataset and 1.21× on the hard dataset. Meanwhile, a 1.07× gain is achieved on easy dataset over hard dataset based on EdgeML and a 1.08× gain is achieved based on EdgeML-NonBranch, which shows the adaptability of our framework.

5.3 Adaptive Performance of EdgeML

5.3.1 Impact of Network Bandwidth Variation. In this section, we demonstrate that EdgeML can adapt to dynamic bandwidth. To create dynamic bandwidth variations, we set a period of low bandwidth (7Mbps) during 200-400 and 600-800 epochs and normal bandwidth (15Mbps) through the rest epochs (1000 in total), which is shown in Fig. 10 (d).

Fig. 10 (a) and (b) show results of latency and GPU frequency, while Fig. 10 (c) and (d) show the corresponding action sets generated by the RL optimizer versus bandwidth variations. We note that during the first period of low bandwidth (200-400 epoch), the latency has a slight decrease, and the average latency is 0.29s for 200-310 epoch and 0.218s for 310-400 epoch. The GPU frequency increases considerably during 200-310 epochs and decreases drastically starting from around the 310th epoch. Observed from Fig. 10 (c), we can find that the partition points also exhibits fluctuations. The variance of partition points in this period is 30.94 (while the variances of the three thresholds are 0.145, 0.15, 0.161 respectively). These fluctuations in this period are the result of RL optimizer exploring new action for adapting to new bandwidth. The sharp decrease of partition point at around 300 leads to the decrease of GPU frequency.

During the second period of low bandwidth (600-800 epoch), the system is considerably more adaptive to this bandwidth variations. The average latency for the period 600-800 epoch (0.057s) is 5.1× better than in 200-400 epoch (0.251s) while the average GPU frequency for the period 600-800 epoch (1.02 GHz) is 1.33× better than in 200-400 epoch (1.36 GHz). This is because the RL optimizer will explore new action space with its greedy mechanism. Gradually, it will find the optimal action sets that do not cause significant variations on the policy, resulting in a stable performance. This

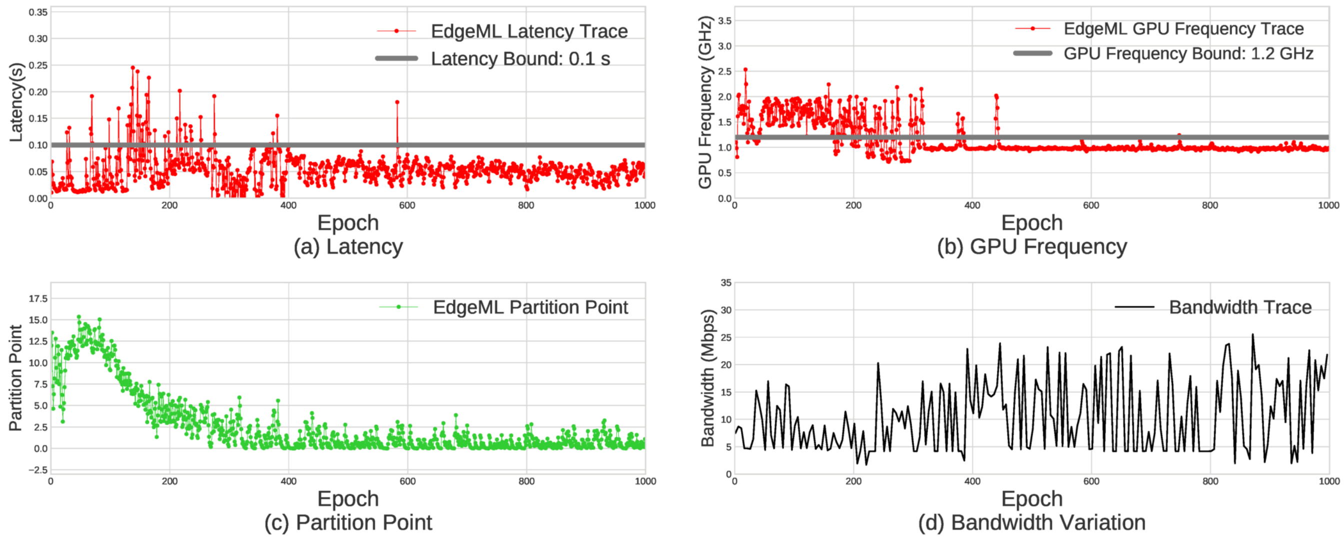

Figure 12: Case study: real-world mobile bandwidth

result shows that the RL optimizer effectively leverages the knowledge learned from the first period of bandwidth variation (200-400 epochs).

In Fig. 10 (a) and (b), there exists some indications that MOEA baseline also learns to adapt to the bandwidth fluctuation (e.g. 250-300 epoch in Fig. 10 (a)), while overall it can hardly find the optimal policy when the bandwidth fluctuates sharply.

5.3.2 Case Study: Real-World Mobile Bandwidth. To further assess the responsiveness of EdgeML in adapting to dynamic network conditions, we collected a real-world bandwidth trace from a mobile phone (iPhone X [2]). The trace contains time series of network bandwidth variability during different user activities, which cover a few different spatial locations such as basement and kitchen, where Fig. 12 (d) shows that the trace exhibits significant dynamics, reflecting the diverse signal coverage levels. EdgeML executes VGG16-CIFAR10 with NVIDIA TX2. Fig. 12 (a) (b) (c) show the latency, GPU frequency and partition point traces. At low bandwidth (<10 Mbps), especially during 0-250 epochs, the RL optimizer adopts a remote-execution policy, falling back the partition point from 15 to 2 in average. Nonetheless, the average latency and GPU frequency correspondingly fall beneath the user bounds. After 250 epochs, although there exists some cases that either latency or GPU frequency exceeds the bound (e.g. around epoch 425 for GPU frequency), the optimizer can find the optimal action sets by leveraging the learned prior knowledge, leading to the rapid system convergence.

5.3.3 Platform Adaptation. When deploying EdgeML to a new edge device, a naive way is to train the RL optimizer of EdgeML from scratch. To speed up the deployment of EdgeML, we discussed the approach of transferring RL optimizer from a deployed edge device to a new device in the Section 4.3. In this section, we will show that this knowledge transferring mechanism allows EdgeML to quickly adapt to changes in compute resources caused by device migration.

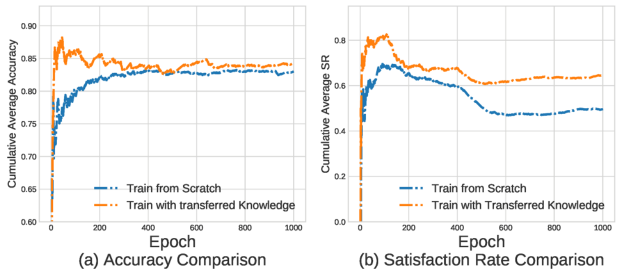

Figure 13: Cross-platform performance of EdgeML.

In the experiment, we first train the RL optimizer on NVIDIA Jetson TX2 for 1000 epochs, and then transfer the MDP traces to the target device (NVIDIA Jetson Nano). We adopt VGG16 as the test model and CIFAR10 as the evaluation dataset. We compare our transferred method with the baseline which trains the RL optimizer from scratch on the target device. The results of cumulative average accuracy and satisfaction rate of the transferred VGG16-CIFAR10 application on the target device is shown in Fig. 13. The first 200 epochs in Fig. 13 suggests that our approach can adapt to the target device and achieve better performance than baseline. From Fig. 13 (a) we also observe that there is a search process at the very beginning of the deployment of our method. This is because the computing power on Nano platform is substantially weaker than TX2 platform, and thus the execution policy generated by the original RL optimizer may not be efficient on the new platform (Nano) anymore. However, our method can quickly adapt to the new device compared with the baseline train-from-scratch (0-200 epoch). SR shown in Fig. 13 (b) is the satisfaction rate of the complete execution process from the beginning. It can be seen that the gap between the two curves tends to saturate after convergence. We also set different bandwidth to show the robustness of two methods. When the bandwidth decreases (400-600 epoch), our method yields a better robustness than the baseline train-from-scratch as shown in Fig. 13 (b).

5.4 Overhead Analysis

Under all settings of DNNs and datasets, the RL optimizer executes in average 20 ms/epoch. The execution time includes reading data from memory buffer in RL, computation of RL inference, storing MDP trace and returning new action policies found. When the system performance converges (e.g. at around 450 epoch in Fig. 12), the interval at that RL optimizer is invoked will increase (i.e., maximum 5 steps/epoch). To further minimize the overhead, we employ multi-thread programming to parallelize part of the RL calculation with the DNN inference on server, which reduces the execution time of RL to 7 ms/epoch. Overall, the optimzer's execution time is within 2 percent of the system's end-to-end latency. On the other hand, EdgeML takes more hard disk space than original DNN models due to the existence of branches. For example, EdgeML takes 1.2× and 1.365× the storage than those of original ResNet50 and VGG16, respectively.

6 RELATED WORK

DNN acceleration has attracted significant attention in recent years due to the growing interests of running deep learning tasks on resource-constrained platforms. Recent chips like Nervana Neural Network Processor (NNP) [18] and Tensor Processing Units (TPU) [16] can accelerate both training and inference process of DNN. At algorithm level, Winograd Algorithm [21, 37] and FFT based methods [6, 29] are proposed to accelerate convolution operations in DNN inference. These solutions are complementary to our work, which can be integrated in EdgeML design.

Extensive efforts are made on making deep learning tasks affordable on IoT and edge platforms. One of the most prevalent methods is to reduce the resource demand of DNN model via compression, which usually leads to a modest loss in accuracy. Knowledge Distillation [4, 15] trains a compact model with information from the original complex one, effectively reducing the compute overhead. However, the need of pre-training makes it less applicable to dynamic environments. A number of pruning methods [12, 28] overcome the issue of pre-training. Although they can reduce the model size through selectively removing neurons, they do not usually lead to lower compute overhead. Filter pruning [23] can effectively reduce the computation cost by pruning the whole filter set based on the importance. Although above techniques reduce the overhead of DNN execution, they are not designed to optimize the performance of DNN in the presence of dynamics of input data and resource budget. Extending the static model compression approach, several efforts [7, 8, 27] proposed dynamic neural networks that allow selective execution to improve DNN compute efficiency. D2NN [27] optimizes dynamic resource-accuracy trade-offs, while its complicated network structure incurs significant memory overhead, making it ill-suited for resource-constrained platforms. BranchyNet [33] proposed a dynamic neural network architecture with added blocks and branches. EdgeML leverages this approach to achieve dynamic selective execution of DNN via proper early-exit control. In [11], dynamic workload offloading polices are proposed for executing DNN models either on cloud or locally, which do not account for dynamic communication delay. Neurosurgeon [17] predicts the latency and power consumption of each layer, and automatically partitions DNN at the layer granularity for workload offloading. EdgeML adopts a similar idea as Neurosurgeon to dynamically partition DNN model based on its hierarchical architecture. Different from the heuristic partition approach in Neurosurgeon, EdgeML searches for the optimal joint partition and early-exit control decisions, using a principled reinforcement learning algorithm, which addresses unpredictable communication delay and dynamics in data input.

AutoML is an emerging paradigm that aims to automate the pipeline of DNN design. He et al. [14] applies AutoML approach to achieve fully automated model compression, by searching and predicting the layer redundancy. Zoph and Le [41] introduce AutoML approach for neural architecture search, maximizing the average accuracy of the generated neural architectures. Similar to several existing AutoML solutions, EdgeML also adopts reinforcement learning for automated decision making. However, different from current works on AutoML that focus on automated model design, EdgeML applies the principle of AutoML to dynamically optimize the model execution policy and in real-time.

Similar to our work, Edgent [22] integrates the workload offloading and progressive neural architecture. However, it is based on simplistic system models, e.g., only a single branch is allowed, which cannot fully exploit the benefit of progressive neural architecture. Most recently, SPINN [20] also adopted this combined approach and considered multiple early-exit branches to allow flexible model inference. Compared to SPINN, EdgeML presents two major novel designs. First, SPINN uses a single threshold for all inserted branches in the progressive neural architecture, while each branch is assigned with an independent threshold in EdgeML, which leads to substantial performance improvement as shown in our experiments (Fig. 9). Second, SPINN employs a multi-objective optimization approach for generating execution policy, which may not be efficient for coping with huge solution space incurred by the continuous threshold values and model partition points. In contrast, EdgeML adopts reinforcement learning-based AutoML framework to dynamically optimize the execution policy, which not only enables more flexible adaptation of model inference between the edge and cloud in dynamic environments, but also accommodates different platforms via knowledge transfer. We validated these advantages of EdgeML in our experiments, in comparison with a baseline solution MOEA that is designed based on a multi-objective optimization approach similar to SPINN.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose EdgeML, an AutoML framework that accelerates deep learning tasks on edge devices so as to fulfil user requirements on task latency, accuracy and edge device energy consumption. EdgeML combines the workload offloading mechanism and the progressive neural architecture to support flexible DNN model execution policies on the edge. The RL-based framework can allow EdgeML to automatically adapt to the environmental variations such as communication channel conditions between the cloud and edge devices, as well as dynamic input characteristics.

EdgeML represents the first framework that supports online AutoML framework for edge computing applications. It allows adaptive environment-aware network structures that can run with limited resources on edge and IoT devices. In our experiments, we evaluate two widely studied DNN models VGG-16 and ResNet-50 in

EdgeML. However, EdgeML can also be integrated with many other DNN models.

As illustrated in Section 5.4, the early-exit mechanism can bring high compute overhead, especially when there are more exit paths in the DNN models. A possible improvement is to parallelize the computation of branch execution, allowing the model inference to continue at the cross of side branch and main branch. However, this solution may incur more memory and storage space.

In the design of RL optimizer, we make the assumption that the DDPG algorithm will learn the inner transition knowledge under the dynamic environment (e.g., network bandwidth variations). Although DDPG is known to have strong mathematical guarantees regarding fast convergence to the optimal policy, its stability in a dynamic environment has not been well understood, which will be investigated in our future work.

ACKNOWLEDGEMENT

The work described in this article was partially supported by the Research Grants Council of Hong Kong under Grant No. 14203420, and by the Centre for Perceptual and Interactive Intelligence (CPII) Ltd under the Innovation and Technology Fund.

REFERENCES

[1] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[2] APPLE. Apple. https://www.apple.com/iphone/ Accessed 2020.

[3] Simon Séhier Bert Hubert, Jacco Geul. The wonder shaper 1.4.1. https://github.com/magnific0/wondershaper Accessed 2020.

[4] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems*, pages 742–751, 2017.

[5] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

[6] Charles Dubout and François Fleuret. Exact acceleration of linear object detectors. In *European Conference on Computer Vision*, pages 301–311. Springer, 2012.

[7] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[8] Biyi Fang, Xiao Zeng, and Mi 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*, pages 115–127. ACM, 2018.

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[10] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[11] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind 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*, pages 123–136. ACM, 2016.

[12] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[14] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[16] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[17] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGPLAN Notices*, 52(4):615–629, 2017.

[18] Carey Kloss. Intel nervana™ neural network processor: Architecture update. www.intel.ai/intel-nervana-neural-network-processor-architecture-update. June 12, 2017.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images.(2009), 2009.

[20] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.

[21] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[22] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.

[23] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[24] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101, 2018.

[25] Peiliang Li, Xiaozhi Chen, and Shaojie Shen. Stereo r-cnn based 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7644–7652, 2019.

[26] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[27] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[28] Zelda Mariet and Suvrit Sra. Diversity networks. *Proceedings of ICLR*, 2016.

[29] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[30] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[31] NVIDIA. Meet jetson, the platform for ai at the edge. https://developer.nvidia.com/embedded-computing Accessed April 11, 2020.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[33] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *Pattern Recognition (ICPR), 2016 23rd International Conference on*, pages 2464–2469. IEEE, 2016.

[34] Zhihong Tian, Wei Shi, Yuhang Wang, Chunsheng Zhu, Xiaojiang Du, Shen Su, Yanbin Sun, and Nadra Guizani. Real-time lateral movement detection based on evidence reasoning network for edge computing environment. *IEEE Transactions on Industrial Informatics*, 15(7):4285–4294, 2019.

[35] Qiang Wang and Xiaowen Chu. Gpgpu power estimation with core and memory frequency scaling. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):73–78, 2017.

[36] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[37] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[38] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Hu Yi-Qi, Li Yu-Feng, Tu Wei-Wei, Yang Qiang, and Yu Yang. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[39] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE access*, 6:6900–6919, 2017.

[40] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. Ecrt: An edge computing system for real-time image-based object tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 394–395. ACM, 2018.

[41] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.