Automating Cloud Deployment for Real-Time Online Foundation Model Inference

Yang Li, Zhenhua Li, *Senior Member, IEEE, ACM*, Zhenhua Han, Quanlu Zhang, and Xiaobo Ma, *Member, IEEE*

***Abstract*— Deep neural network (DNN) foundation models are currently exhibiting high prediction accuracy and strong adaptability to broad tasks with remarkably large model scales. They are increasingly becoming the backend support of DNN-driven real-time online services, e.g., Siri and Instagram. Such services require low-latency and cost-efficiency for quality-of-service and commercial competitiveness. When deployed in a cloud environment, these services call for an appropriate selection of cloud configurations (i.e., specific types of VM instances), as well as a considerate device placement plan that places the operations of the model to multiple GPUs via model parallelism for cost-efficiency. Currently, the deployment mainly relies on service providers' manual efforts, which is not only onerous but also far from satisfactory oftentimes due to the huge joint search space of cloud configurations and device placement plans (for a same service, a poor deployment can incur significantly more costs by tens of times). In this paper, we attempt to efficiently automate the cloud deployment for real-time foundation model inference with minimum costs under the constraint of acceptably low latency. This attempt is enabled by 1) jointly leveraging the Bayesian Optimization and Deep Reinforcement Learning to adaptively unearth the (nearly) optimal cloud configuration and device placement with limited search time, and 2) enhancing the cost-efficiency of the deployment based on the probing-informed block multiplexing mechanism and Tensor Algebra SuperOptimizer. We implement a prototype system based on TensorFlow, conduct extensive experiments on top of Microsoft Azure, and demonstrate the generality and scalability of our solution. Results show that for lightweight DNN models and foundation models, our solution essentially saves inference costs by up to 15% and 47% with 57% and 38% lower search overheads respectively, compared with non-trivial baselines.**

***Index Terms*— Automating, cloud configuration, deep learning inference, real-time services.**

Manuscript received 4 February 2023; revised 18 September 2023; accepted 24 September 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. Date of publication 13 October 2023; date of current version 18 April 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500703; in part by the National Natural Science Foundation of China under Grant 61972313, Grant 61902211, and Grant 62202266; in part by the Natural Science Basic Research Program of Shaanxi Province under Grant 2023-JC-JQ-50; in part by the China Postdoctoral Science Foundation under Grant 2022M721831; and in part by the Microsoft Research Asia under Grant 100336949. *(Corresponding author: Zhenhua Li.)*

Yang Li and Zhenhua Li are with the School of Software, Tsinghua University, Beijing 100190, China (e-mail: liyang14thu@gmail.com; lizhenhua1983@gmail.com).

Zhenhua Han and Quanlu Zhang are with Microsoft Research Asia, Beijing 100080, China (e-mail: hzhua201@gmail.com; quzha@microsoft.com).

Xiaobo Ma is with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: xma.cs@xjtu.edu.cn).

Digital Object Identifier 10.1109/TNET.2023.3321967

I. INTRODUCTION

DEEP learning is currently the *de facto* standard technique used in various areas, such as computer vision [1], [2], speech recognition [3], [4], [5], [6], and natural language processing [7], [8], [9], [10]. In recent years, deep neural network (DNN) foundation models [11] are increasingly becoming crucial back-end support of many real-time online services [12], [13], such as Siri and Instagram. Such models are pre-trained common basises, from which many task-specific models are built via lightweight adaptation instead of trained from scratch, while manifesting high prediction accuracy due to the sheer scale of operations and parameters. As the accuracy is ensured by the foundation model, the performance of a real-time online service mainly depends on the response time for handling user requests, which includes the network transmission time, task scheduling time, *inference time* (i.e., the execution time of the DNN inference), and so forth. In the response time, inference time usually occupies the dominant portion [14], especially for the large foundation model. Hence, we take inference time as the major constraint of quality-of-service (QoS) in DNN-driven real-time online services.

Due to the economies of scale and elasticity of cloud computing, many real-time online services choose to deploy their pre-trained foundation models in public clouds (e.g., Amazon Web Services, Microsoft Azure, and Google Cloud) and provide the corresponding inferences to users. A public cloud typically offers a variety of (e.g., 100+) *cloud configurations* (i.e., specific types of VM instances with different hardware and OSes [15], [16]) to its customers, which are specialized to support machine learning jobs. At the moment, (DNN-driven real-time online) service providers usually artificially select their cloud configuration. Among the numerous available cloud configurations, it is not easy for them to find the best one [15], and thus their selected VM instances are often either over-configured that lead to a waste of money or under-configured that slow down the inference.

For the cost-efficiency of service deployment, service providers also need to consider model parallelism for DNN inference. Specifically, they should explicitly place the operations of a DNN on multiple GPUs to accelerate its inference [17], [18]. Through model parallelism, configurations with multiple low-end GPUs may have similar inference performance with configurations with one high-end GPU, while having less cost (as shown in Table IV). Consequently, a considerate device placement plan, which can well trade

off the computation parallelism and the inter-device communication overheads, is also called for. In practice, such a plan is usually artificially designed by service providers at present. Once again, it is hard for them to make an optimal or near-optimal device placement plan, especially for the foundation model with a remarkably large computation graph [17] (which contains a set of operations with inter-operation dependencies).

Given the computation graph of a foundation model, finding the optimal cloud configuration and device placement is highly challenging, since it involves a huge search space – the joint space of all available cloud configurations and all possible device placement plans. Hence, we pose a critical question for today's DNN-driven real-time services: *how can we automatically determine the cloud configuration and device placement for the foundation model inference, so as to minimize the inference cost while satisfying the inference time constraint?* Here inference cost is the product of inference time (in the unit of second per request) and the price of the cloud configuration (in the unit of dollar per hour).

In the preliminary work [19], we proposed AutoDeep to answer the question how to automate the cloud deployment of the relatively lightweight DNN model inference under a QoS constraint. Given a DNN model and the inference time constraint (which should be acceptably low), AutoDeep attempts to compute the cloud deployment with the lowest inference cost. We formulate the attempt as a two-fold joint optimization of cloud configuration and device placement. In order to enable the attempt, AutoDeep leverages Bayesian Optimization (BO) for unearthing the (nearly) best cloud configuration within limited search time, and meanwhile utilizes Deep Reinforcement Learning (DRL) for making the (nearly) optimal device placement plan. In detail, AutoDeep employs BO to judge which cloud configuration should be sampled next to best reduce the inference cost; as for each sampled cloud configuration, AutoDeep iteratively trains a DRL model to make the optimal device placement plan. In a nutshell, AutoDeep strategically learns the characteristics of a DNN model and the available cloud configurations to figure out a cost-efficient cloud configuration and device placement plan under the inference time constraint.

In this paper, when attempting to reuse AutoDeep for the cloud deployment of the emerging foundation model-based real-time online services, we encounter two-fold additional challenges. First, AutoDeep's joint adoption of BO and DRL requires running the time-consuming foundation model inference trial in considerable iterations, making the cloud deployment rather inefficient. Second, to enable real-time inference of the large foundation model, the cloud configuration optimized by AutoDeep is high-end, leading to a high inference cost.

Given the above question and challenges, we upgrade AutoDeep in terms of the cost-efficiency of the foundation model inference deployment through two-fold innovations (Section IV). First, by dynamically tracing the runtime inference trial through basic-block code instrumentation [20] in TensorFlow internals, we uncover the root cause of the long trial run time: there exist heavy startup code blocks

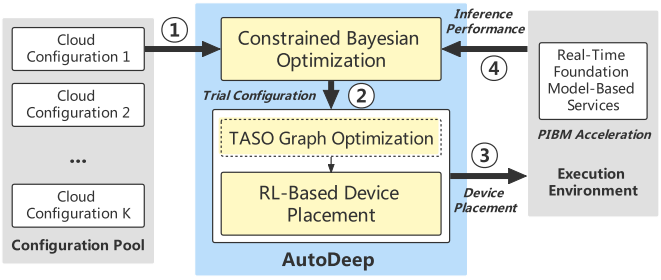


Fig. 1. Architectural overview of the AutoDeep framework.

(including hardware detection, computation graph building, and CUDA initialization), which are independent of the trial input but take more than 98% of the time spent by the trial. Therefore, we advance the original trial scheduling mechanism of AutoDeep by identifying the startup blocks with the probes in code instrumentation and pre-executing them in a single pass, so that the search overhead is significantly reduced since all trials can reuse the startup basis; the resulting mechanism is dubbed Probing-Informed Block Multiplexing or PIBM for short. We also analyze the adaptability of the PIBM mechanism to other mainstream deep learning frameworks in Section IV-D.

Moreover, we observe that in the computation graph of the foundation model, there exist inefficient subgraphs which severely slow down the model inference. Therefore, we attempt to reduce the inference cost by employing an adaptive graph substitution approach called Tensor Algebra SuperOptimizer (TASO) [21], which can find a semantically equivalent but remarkably lower-cost DNN computation graph. The high-level architecture of the up-to-date AutoDeep is depicted in Figure 1, and the dashed box means that TASO graph substitution only needs to be executed once in the first iteration of the joint optimization. Compared with other deep learning compilers, TASO has better flexibility, compatability, and optimization performance regarding the integration with AutoDeep, as illustrated in Section IV-C.

We implement AutoDeep based on TensorFlow [22] and build the prototype system on top of VM instances rented from Microsoft Azure. To comprehensively evaluate the effectiveness and adaptability of AutoDeep, we conduct extensive experiments using both commonly-used lightweight DNN models and well-known foundation models, including models for natural language processing (i.e., RNNLM [23], NMT [24], BERT [25], Transformer [26], and GPT-2 [27]) and models for online image classification (i.e., Inception-V3 [28], ResNeXt-50 [29], and VGG19 [30]). The experiment results show that AutoDeep improves the inference speed by up to 68% with 98% lower optimization overhead for lightweight DNN models, and achieves up to 23% faster inference with 90% lower optimization overhead for foundation models, compared with the non-trivial baselines such as Google's RL-based device placement [31]. Moreover, AutoDeep essentially reduces up to 15% of the inference cost and 57% of the search time for lightweight DNN models, and saves up to 47% of the inference cost and 38% of the search time for foundation models, compared with heuristic baselines such as

TABLE I
INFERENCE COST (10000 TIMES) OF DIFFERENT MODELS ACROSS DIFFERENT CLOUD CONFIGURATIONS

Model	Min Inference Cost	Max Inference Cost
RNNLM	\$0.11	\$0.18
Inception-V3	\$0.07	\$0.74
VGG19	\$0.20	\$2.13
BERT	\$0.23	\$3.27

greedy search. We also clarify the scalability of AutoDeep when the testbed expands in Section V-C.

Roadmap. The remainder of the paper is organized as follows. In Section III, we present the system model and formulate the problem. In Section IV, we present our algorithm AutoDeep. In Section V, we demonstrate the prototype setting and the experiments results. We survey the related works in Section VI and conclude this paper in Section VII.

II. MOTIVATION

In this section, we show the problem and challenges of automating the cloud deployment for deep learning inference of real-time online services. We also explain why existing solutions do not solve the problem.

A. Problem

An appropriate cloud configuration is crucial to the inference performance and the operation cost of online services. Different from training a DNN model, the inference of a DNN model usually supports the online services that run over months or even years. Table I shows the minimum and maximum cost of 10000 times inference for 4 popular DNN models across all cloud configurations in a cloud provider. We observe a poor cloud configuration can incur up to 14 times cost compared to the best one.

Online services have the trade-off between operation cost and performance. Simply using the cheapest or the most expensive cloud configuration can hardly achieve the optimal trade-off. Thus, it is important to find a cost-efficient configuration (and device placement) within a QoS constraint. We focus on model parallelism with multiple GPUs on one physical machine, since inference performance with multiple physical machines causes extra bandwidth cost and is highly affected by the reliability of the communication network [32].

B. Challenges

There are two challenges for picking the cost-efficient cloud configuration and the optimal device placement plan.

Huge search space: Finding the cost-efficient cloud configuration and the optimal device placement involves a huge search space. Firstly, cloud service providers usually have many VM instance types that require users to decide which ones to use. For example, both AWS and Microsoft Azure provide over 100 types of cloud configurations. Secondly, even with a fixed cloud configuration, there still exist a large amount of different device placement plans. A DNN model can have hundreds to thousands of operations. Each operation can be placed on a list of feasible devices (e.g., CPUs or GPUs). Therefore, the space of feasible device placement plans grows exponentially with the number of operations. The search space further expands with the joint of cloud configuration and the device placement.

Complex performance model: The VM instance types offered by cloud service providers have heterogeneous configurations on the number of CPU cores, the RAM sizes, the type of GPUs, the number of GPUs, etc. The cloud charges users with the amount of running time of the VMs, which is independent with the job running inside them. It relies on users to pick the suitable cloud configuration for their workloads.

Under the premise of meeting the QoS requirement (i.e., the commercial-grade inference speed), users typically consider low-end cloud configurations for cost-efficiency. In this case, model parallelism is essential given the following facts: 1) the inference of large foundation models may run out of memory with a single low-end GPU device (e.g., the inference of BERT with a long sequence input runs out of memory on one GTX 980Ti GPU) but can run on multiple such devices via model parallelism to satisfy the memory requirement; 2) cloud configurations with one high-end GPU device can cost times as much as those with multiple low-end GPU devices (e.g., a configuration with one A100 GPU costs over 3.3 times as much as that with 2 GTX 980Ti GPUs); 3) inference of lightweight models that can fit in commercial GPU memory may be further accelerated through well-organized graph partition via model parallelism on specific configurations [33].

However, the performance of a DNN with model parallelism is very complicated [34]. It is hard for users to predict the inference performance over different cloud configurations. Especially, different cloud configurations may need different device placement plans for the best performance. The typical practice is to heuristically place some code-level operators on a given device (e.g., a GPU) based on the domain expertise. But such decisions can be challenging for dynamic DNN with multiple branches, due to the unclarity and variation of the hardware performance [35]. Existing algorithmic solvers for graph partition, such as Scotch [36] and Metis [37], do not work for this problem, because they need accurate cost models, which is almost impossible for the complex DNN models.

Specific challenges for foundation model inference: Different from other lightweight task-specific DNN models, the foundation model is a common basis for many task-specific models to be built from. Compared with other models, foundation models exhibit higher prediction accuracies due to the sheer scale of operations and parameters in their computation graphs and thus their inferences are much more time- and resource-consuming. Besides, the increasing model scale and graph complexity of the foundation model make the aforementioned joint search space even larger, and hence slow down the convergence of the joint optimization and increase the overheads significantly.

C. Black-Box Optimization for Combinatorial Problem

The huge search space and the complex performance of DNN models motivate us to adopt black-box optimization techniques. Black-box optimization algorithms aim to optimize an objective function $f(x)$ with or without constraints through

a "black-box" interface: the algorithm can query the value of $f(x)$ at the point x without knowing any other information (e.g., gradient) and assuming any forms of $f(x)$ (e.g., being linear or convex). The goal is to find a value of $f(x)$ as good as possible within the limited time.

The black-box optimization is naturally suitable for solving the joint optimization of cloud configuration and device placement for DNN models. Due to the complexity of DNN's performance, the inference time of a given device placement plan under a fixed cloud configuration can be regarded as a black-box function. An input of the black-box function is the combination of a cloud configuration associated with a device placement plan. The goal is to find the minimum inference cost with a given QoS requirement.

Black-box optimization techniques, such as Bayesian Optimization (BO), have been proved to be effective when the search space is small [15]. However, they cannot be simply applied to solve the combinatorial device placement problem due to the extremely large and exponentially growing search space. Therefore, we seek for the Deep Reinforcement Learning (DRL), which has been proved to be effective for solving the large-scale combinatorial optimization problem [38] with a black-box objective function, which adopts deep neural network to exploit the problem structure.

III. System Model and Problem Formulation

In this section, we introduce the system models used in this work and formally define the problem of joint optimization of cloud configuration and device placement.

A. Cloud Configuration

We consider a cloud service provider that offers the GPU servers in the form of various cloud configurations. A cloud configuration is a combination of the computing resources, typically CPUs and GPUs. For example, Microsoft Azure provides NC24 configuration with 24 CPU cores and four NVIDIA K80 GPUs with the price of $3.60 per hour. Users can choose among the configurations to run their DNN inference jobs on the clouds.

Suppose there are K types of cloud configurations in total. The k-th cloud configuration is represented by a set of computing devices $D_k = \{d_{k,1}, d_{k,2}, \ldots, d_{k,|D_k|}\}$. Each device $d_{k,i}$ can be a CPU core or a GPU device. Given that the performance bottleneck of the model inference mostly lies on the computation capability of GPU devices, we assume the memory and disk space in all cloud configurations are sufficient since for the inference of a typical foundation model, the requirements for the memory and disk space are quite low (the memory usage is usually less than 2 GB and the disk space generally takes less than 5 GB according to our tests). Therefore, we do not consider memory and disk space in the rest of the paper.

We assume the CPUs in different configurations have the same computing capability.[1] The price of the k-th cloud configuration is m_k (in the unit of dollar per hour).

[1]Nowadays, the CPUs in the cloud datacenter are usually customized. The cloud service provider guarantees that the CPUs in different configurations have similar performance. Thus only the number of CPU cores matters in different configurations.

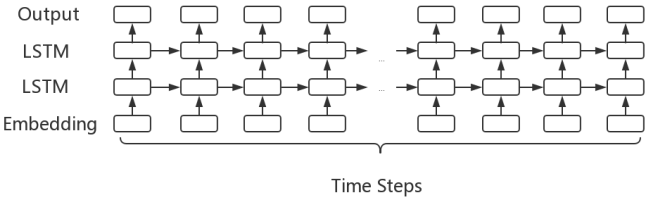


Fig. 2. The computation graph of RNNLM.

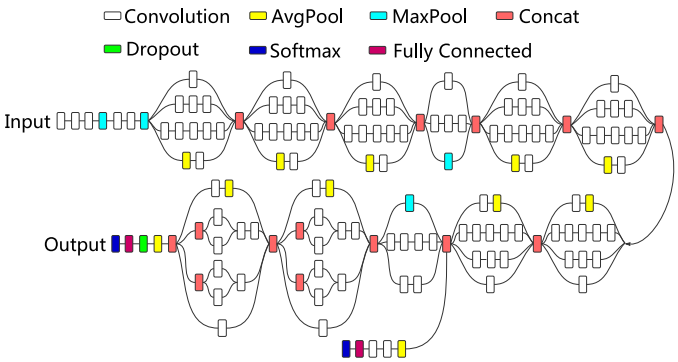


Fig. 3. The computation graph of Inception-V3.

B. Computation Graph and Device Placement

The prevalent machine learning frameworks usually abstract the computation of a DNN inference as a computation graph (e.g., TensorFlow [22]). Fig. 2 and Fig. 3 demonstrate the computation graphs of two popular DNN models, RNNLM [23] and Inception-V3 [28] respectively. The former is designed for natural language processing and the latter is designed for image classification. Denote the computation graph of a DNN inference job as $\mathcal{G}$. The computation graph $\mathcal{G}$ consists of n_o operations (denoted as $\mathcal{O} = \{o_1, o_2, \ldots o_{n_o}\}$). There is a set of directed edges in $\mathcal{G}$. Each edge connects two operations that represent the dependency relationship of them. If a directed edge connects o_i and o_j, then the operation o_j can only be started after the finish of the operation o_i.

To execute a DNN inference job, each operation in its computation graph should be placed on a computing device, e.g., a CPU core or a GPU. We define a device placement $\mathcal{P} = (p_1, p_2, \ldots p_{n_o})$ as a mapping from $\mathcal{O}$ to D_k, where p_i is the device the operation o_i placed on. Some operations have the requirement of the placed device, e.g., the input data reading operation should only be placed on a CPU. We denote the device requirement of the operation o_i in the k-th configuration as $\mathcal{F}_i(D_k)$, i.e., any feasible device placement should satisfy $p_i \in \mathcal{F}_i(D_k)$. Due to the heterogeneity of GPUs, different placements will result in different computation time of the graph $\mathcal{G}$. We denote the computation time of the graph $\mathcal{G}$ under the cloud configuration D_k using the device placement $\mathcal{P}$ as $\mathcal{T}(\mathcal{G}, \mathcal{P}, D_k)$. Since the graph execution and environment involve very complex trade-off between computation and communication in the hardware, it is hard to define the graph execution time in a close-form. Therefore, we assume the inference time is a black-box function but can be profiled accurately given the device placement and cloud configuration.

TABLE II
IMPORTANT NOTATIONS

K	the number of cloud configurations
D_k	the computing devices in the k-th configuration
$d_{k,i}$	the i-th computing device in D_k
m_k	the price of the k-th cloud configuration
$\mathcal{G}$	the computation graph of the DNN inference job
$\mathcal{O}$	the operations in the computation graph $\mathcal{G}$
o_i	the i-th operation in $\mathcal{O}$
$\mathcal{P}$	the device placement plan
$\mathcal{F}_i(D_k)$	the device requirement of the operation o_i in the k-th cloud configuration
$\mathcal{T}(\mathcal{G},\mathcal{P},D_k)$	the inference time of $\mathcal{G}$ under the device placement $\mathcal{P}$ in the k-th cloud configuration
$\overline{\mathcal{T}}$	the QoS constraint
$f_k(\mathcal{G})$	the time spent on the k-th configuration to find the device placement

C. Problem Formulation

In this paper, we study automating the cloud deployment, which is formulated as a joint optimization problem of cloud configuration selection and device placement. We consider the scenario that we are given a DNN computation graph $\mathcal{G}$ and a QoS constraint, which is the inference time requirement. Our goal is to find the cloud configuration and device placement with the lowest cost that satisfies the QoS constraint. We denote the inference time requirement as $\overline{\mathcal{T}}$. Since simply searching the cost-efficient cloud configuration in a brute-force manner is too expensive, the problem should be solved within a limited search time, which is denoted as M. Formally, we formulate the optimization problem as follows (important notations are summarized in Table II):

$$\textit{Minimize}: \sum_{k\in[K]} \hat{x}_k \cdot m_k \cdot \mathcal{T}(\mathcal{G},\mathcal{P},D_k), \tag{1}$$

$$\textit{subject to}: \begin{cases} x_k \in \{0,1\}, \forall k \in [K], & \text{(2a)}\\ \hat{x}_k \in \{0,1\}, \forall k \in [K], & \text{(2b)}\\ p_i \in \mathcal{F}_i(D_k), \forall i \in [n_o], & \text{(2c)}\\ \sum_{k\in[K]} \hat{x}_k = 1, & \text{(2d)}\\ \sum_{k\in[K]} \hat{x}_k \cdot \mathcal{T}(\mathcal{G},\mathcal{P},D_k) \le \overline{\mathcal{T}}, & \text{(2e)}\\ \sum_{k\in[K]} x_k \cdot f_k(\mathcal{G}) \le M. & \text{(2f)} \end{cases}$$

where x_k indicates whether the k-th configuration is tried during configuration searching, $f_k(\mathcal{G})$ is the time spent on finding the device placement of $\mathcal{G}$ using the k-th configuration, $\hat{x}_k$ indicates whether the k-th configuration is the final configuration in the solution. Constraint (2c) guarantees the feasibility of each candidate device placement. Constraint (2d) ensures there is only one cloud configuration that is used in the final solution. Constraint (2e) specifies the QoS constraint of the final cloud configuration and the device placement. Constraint (2f) limits the search time. The optimization objective in (1) is to minimize the inference cost of the final solution. In the following section, we design an efficient algorithm that will iteratively find the cost-efficient cloud configuration and the device placement, without assuming any knowledge of the execution environment and the statistical information of the DNN inference computation graph.

IV. AUTODEEP: UNEARTHING THE COST-EFFICIENT CLOUD CONFIGURATION AND THE DEVICE PLACEMENT

In this section, we present AutoDeep that can iteratively unearth the cost-efficient cloud deployment given a foundation model inference computation graph and its QoS constraint. Our objective is two-fold: optimizing the cloud configuration and the device placement while using the least search time. We start with a high-level overview of the proposed algorithm AutoDeep, and then describe the details on how we choose the cloud configuration and find the optimized device placement.

A. Overview of AutoDeep

AutoDeep iteratively finds the cost-efficient cloud configuration and the device placement. Fig. 1 illustrates the algorithm framework of AutoDeep. In each iteration, AutoDeep first decides the cloud configuration using a Bayesian Optimization (BO) based approach. Then, AutoDeep will substitute the computation graph of the foundation model with a semantically equivalent but lower-cost one through Tensor Algebra SuperOptimizer (TASO), and try to learn the environment and optimize the device placement with a DRL based method. After AutoDeep finds the device placement that satisfies the QoS constraint or asserts that the QoS constraint cannot be achieved under this cloud configuration, AutoDeep will extract the underlying characteristics of the inference job and the cloud configuration from existing observations and try a new cloud configuration in the next iteration. The iterations in AutoDeep are efficiently executed using the proposed Probing-Informed Block Multiplexing (PIBM) mechanism.

B. Finding the Cost-Efficient Cloud Configuration

For a given DNN with specified QoS constraint, we prepare a set of common GPU configurations and use Bayesian Optimization to get the cost-efficient configuration via multiple iterations. Bayesian Optimization is a sequential design strategy for global optimization of black-box functions that do not require derivatives. To make the paper self-contained, we briefly explain the basic concept of Bayesian Optimization. Please refer to [39] for more details.

Bayesian Optimization has two essential components: 1) a probabilistic model and 2) an acquisition function. In Bayesian Optimization, Gaussian Process is the most commonly used probabilistic model for building the model of the black-box function. The probabilistic model can be used to estimate the inference performance under different cloud configurations. The acquisition function is usually used to predict the expected information gain of each cloud configuration if it is selected for the trial. Bayesian Optimization iteratively

estimates the objective function according to the observed samples. Then it uses a pre-defined acquisition function to get the potential gain of the rest candidate samples and choose the highest one as the next sample. Since the conventional Bayesian Optimization only optimizes the objective function without considering any constraint, we use the constrained acquisition function [40] to overcome this drawback.

To extract more information from the cloud configuration, we replace D_k with the detailed configuration of the computing devices (including the number of CPU cores, the CPU clock speed, the number of CUDA cores, the GPU clock speed, the GPU memory bandwidth and the number of GPUs on the server). We aggregate these information into a vector $\mathbf{D}_k$, which redefines the black-box inference time function $\mathcal{T}(\mathcal{G}, \mathcal{P}, D_k)$ as $\mathcal{T}(\mathcal{G}, \mathcal{P}, \mathbf{D}_k)$ (we use $\mathcal{T}(\mathbf{D}_k)$ for ease of elaboration when there is no ambiguity).

We begin with the expected improvement (EI) acquisition function and show we extend it to the *constrained* EI acquisition function. Let $\hat{\mathbf{D}}_k$ be a candidate cloud configuration for next trial. Define $\tilde{\mathcal{T}}(\hat{\mathbf{D}}_k)$ as Gaussian process posterior estimation for $\mathcal{T}(\mathbf{D}_k)$. The improvement function is defined as

$$\tilde{I}(\hat{\mathbf{D}}_k) = \max\{0, m_k \mathcal{T}(\hat{\mathbf{D}}_k) - m_{k^*} \tilde{\mathcal{T}}(\mathbf{D}_{k^*})\}, \quad (3)$$

where k^* is the cloud configuration with the minimum inference cost, i.e. $k^* = \arg\min_k m_k \cdot \mathcal{T}(\mathbf{D}_k)$. Thus the expected improvement acquisition function becomes

$$E\tilde{I}(\hat{\mathbf{D}}_k) = \mathbb{E}[\tilde{I}(\hat{\mathbf{D}}_k)|\hat{\mathbf{D}}_k], \quad (4)$$

which can be easily computed with the closed form derived by Jones et al. [41].

To extend the acquisition function to cover the QoS requirement, we first define the constrained improvement acquisition function as follows:

$$\tilde{I}_C(\hat{\mathbf{D}}_k) = \tilde{\Delta}(\hat{\mathbf{D}}_k) \max\{0, m_k \mathcal{T}(\hat{\mathbf{D}}_k) - m_{k^*} \tilde{\mathcal{T}}(\mathbf{D}_{k^*})\}, \quad (5)$$

where $\tilde{\Delta}(\hat{\mathbf{D}}_k)$ is an indicator function whose value is 1 if the QoS constraint is satisfied (i.e., $\mathcal{T}(\hat{\mathbf{D}}_k) \leq \overline{\mathcal{T}}$), and 0 otherwise. In fact, the quantity $\tilde{\Delta}(\hat{\mathbf{D}}_k)$ is a Bernoulli random variable with the parameter:

$$\begin{aligned}\Gamma(\hat{\mathbf{D}}_k) &= Pr[\hat{\mathbf{D}}_k \leq \lambda] \\ &= \int_{-\infty}^{\lambda} \delta(\mathcal{T}(\hat{\mathbf{D}}_k)|\mathbf{D}_k, \hat{\mathcal{T}}(\hat{\mathbf{D}}_k)) d\, \mathcal{T}(\hat{\mathbf{D}}_k), \end{aligned} \quad (6)$$

where $\delta(\cdot)$ is the probability density function. Conveniently, due to the marginal Gaussianity of $\hat{\mathbf{D}}_k$, the quantity $\Gamma(\hat{\mathbf{D}}_k)$ is a univariate Gaussian cumulative distribution function [42].

Finally, we obtain the expected constrained improvement acquisition function as follows:

$$\begin{aligned} EI_C(\hat{\mathbf{D}}_k) &= \mathbb{E}\left[\tilde{I}_C(\hat{\mathbf{D}}_k)|\hat{\mathbf{D}}_k\right] \\ &= \mathbb{E}\left[\tilde{\Delta}(\hat{\mathbf{D}}_k)\tilde{I}(\hat{\mathbf{D}}_k)|\hat{\mathbf{D}}_k\right] \\ &= \mathbb{E}\left[\tilde{\Delta}(\hat{\mathbf{D}}_k)|\hat{\mathbf{D}}_k\right]\mathbb{E}\left[\tilde{I}(\hat{\mathbf{D}}_k)|\hat{\mathbf{D}}_k\right] \\ &= \Gamma(\hat{\mathbf{D}}_k) E\tilde{I}(\hat{\mathbf{D}}_k). \end{aligned} \quad (7)$$

In fact, the expected constrained improvement acquisition function in eqn. (7) is the expected improvement of $\hat{\mathbf{D}}_k$ over the probability that $\hat{\mathbf{D}}_k$ satisfies the QoS constraint.

Although the goal of cloud configuration searching we defined is to find the configuration with the lowest inference cost while satisfying the QoS constraint, our approach can be easily extended to other performance-related goals, such as finding the configuration with lowest inference time within an inference cost constraint [42]. Because we have $\tilde{\mathcal{T}}(\hat{\mathbf{D}}_k)$ to estimate the inference time of a DNN model under different cloud configurations, we can design the improvement function and the acquisition function to cover other performance-related objectives and constraints. Thus, our approach is very general for cloud configuration searching.

C. Finding the Device Placement

Graph substitution using TASO. Given that applying the preliminary AutoDeep [19] for the deployment of the foundation model results in a high inference cost, we explore graph opimization techniques to further discover the potential of inference acceleration on affordable cloud configurations. Motivated by our observation that there exist inefficient subgraphs in the computation graph of the foundation model, we adopt an adaptive graph substitution approach called Tensor Algebra SuperOptimizer (TASO) [21], which can find a semantically equivalent but remarkably lower-cost DNN computation graph. Compared with mainstream deep learning compilers (e.g., TVM, XLA, MLIR, etc.), TASO has better compatibility with AutoDeep. This is because these compilers are end-to-end frameworks that aim at generating executable modules for diverse computing resources, while TASO solely works on the computation graph. TASO's graph substitution optimizes the graph structure and AutoDeep's DRL approach works on the graph computation parallelism, they can promote each other to reach the optimal inference time of the foundation model. Besides, the evaluation of TASO [21] shows that it significantly outperforms these compilers on inference acceleration. We insert the TASO graph substitution between the BO-based configuration selection and DRL-based device placement searching in the first iteration of the joint optimization, so that TASO only needs to be executed once and the optimized graph can be reused by all subsequent iterations.

TASO automatically generates and verifys the graph substitutions along with the runtime performance optimization. Specifically, a graph substitution has 3 components: a) a target graph which is matched to subgraphs in the original computation graph, b) a rewrite graph which is a functionally equivalent new subgraph to replace the matched target, and c) a mapping relation between input/output data in the target and rewrite graphs. TASO automatically generates and optimizes the graph substitutions with 3 main modules: the graph substitution generator, the graph substitution verifier, and a joint optimizer of graph substitution and data layout.[2] Given a set of operator specifications, the graph substitution

[2]Tensor data in a computation graph can be stored in memory in diverse layouts. The layout selection highly impacts the runtime performance and depends on both the operator type and the hardware.

generator constructs all acyclic computation graphs which do not include duplicate computation based on a depth-first search algorithm [21]. To efficiently find the rewrite graphs, the generator employs a two-step hash function to compute the fingerprint for each graph as follows:

$$f(g) = hash_2(\{hash_1(t_i)|i \in outs(g)\}), \tag{8}$$

where t_i is one of the output tensors of graph g. $hash_1$ calculates the size, shape, and content of the output tensor data. $hash_2$ is symmetric and applied to an unordered set of hash values. Then the generator tests the equivalence of graphs with the same fingerprint on randomized test cases (floating point numbers ranging between -1 and 1). Two graphs are classified as equivalent if their outputs differ by no more than 10^{-5}, and this threshold can be adjusted to filter out the rewrite graphs that can incur floating point errors. In this way, all candidate graph substitutions can be generated (the mapping relation between the input/output tensors can be inferred from test cases). Furthermore, TASO prunes redundant substitutions, which are identical to more general valid ones, through renaming input tensors and identifying common subgraphs.

To ensure the correctness of the generated graph substitutions, TASO's graph substitution verifier utilizes a set of operator properties expressed in first-order logic to formally verify the substitutions. Specifically, the verifier models tensor operators using functions of both parameters and input tensors. For instance, $matmul(x, y)$ represents the matrix multiplication operator applied to tensors x and y, and the fact that matrix multiplication is linear can be captured by the operator property below ($ewadd$ represents element-wise tensor addition):

$$\begin{aligned}&\forall x, y, z. matmul(x, ewadd(y, z))\\ &\quad = ewadd(matmul(x, y), matmul(x, z)).\end{aligned} \tag{9}$$

With the operator properties, the verifier uses a first-order theorem prover (Z3 [43]) to check whether the operator properties ensure functional equivalence of the target and rewrite graphs in a generated substitution.

Finally, TASO extends the MetaFlow [44] cost-based backtracking search algorithm to find the optimized rewrite graph by applying verified substitutions to the original computation graph, while considering possible layout optimization opportunities. In particular, the cost model in MetaFlow is motivated by the fact that DNN operators perform dense linear algebra with no branches, and thus their performance on hardware is highly consistent and predictable when the data layouts and configuration parameters are set. Following the idea of MetaFlow, TASO collects the execution time of a DNN operator once for each configuration and data layout, and sums up the measured execution time of operators to estimate the graph performance. When searching the optimized graph, TASO maintains a priority queue of all candidate graphs in the increasing order of cost. The joint optimizer applies each verified substitution and possible layouts to search for the optimized functionally equivalent rewrite graph. A hyperparameter α is used to prune the search space: all graphs whose cost is α times worse than the best discovered graph are ignored to tradeoff between the search overhead and the rewrite graph performance. In our evaluation, the default setting ($\alpha = 1.05$) exhibits good performance.

DRL-based device placement optimization. After the computation graph of the target model is optimized with TASO, we explore the potential of graph computation parallelism to further speed up the inference. We design a model based on DRL to find the (nearly) optimal device placement for the target graph in a specified configuration. In our problem, we should encode the information of the target computation graph as our model's input. A natural idea is to input the information of all the operations in the graph as a sequence of data to the model. The output of the model can be constructed as a sequence of devices corresponding to the input operators. The sequence-to-sequence (Seq2Seq) model works well on the modeling of sequence data, so we design a Seq2Seq model as the agent in our DRL method. The agent places the next operator one-by-one on an available device. Each time an operator is placed, the system changed to a new state for the output of the DRL model until all operators are placed. Then we start to measure the inference time of this placement, which is the reward for training the Seq2Seq model.

Under the cloud configuration $\mathbf{D}_k$, we propose to train a policy $\pi(\mathcal{P}|\mathcal{G};\theta)$ to minimize the objective:

$$J(\theta) = \mathbb{E}_{T(\mathcal{G},\mathcal{P},D_k)\sim\pi(\mathcal{P}|\mathcal{G};\theta)}[(\mathcal{P})|\mathcal{G}]. \tag{10}$$

The policy is defined by an attentional Seq2Seq model which is introduced in detail below, and θ denotes the weight parameters in the Seq2Seq network. The parameters in the network are learned by Adam optimizer [45] based on the REINFORCE equation [46], a commonly used policy gradient method, which is given as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}[\mathcal{T}(\mathcal{G},\mathcal{P},D_k) \cdot \nabla_\theta logp(\mathcal{P}|\mathcal{G};\theta)].$$

We estimate the gradient by drawing K samples from $\mathcal{P}_i \sim \pi(\cdot|\mathcal{G};\theta)$. We reduce the variance of policy gradients by using a baseline term B:

$$\nabla_\theta J(\theta) \approx \frac{1}{K}\sum_{i=1}^{K}((\mathcal{P}_i) - B) \cdot \nabla_\theta logp(\mathcal{P}|\mathcal{G};\theta). \tag{11}$$

We set B as our baseline experiments' results. The reward function (P_i) is simply designed as the execution time of the DNN under the placement P_i in current configuration, which works well in the training process. We also set a random rate, which reduces with the increasing number of episodes, to encourage our model to explore more placements.

We use a sequence-to-sequence model with LSTM [23] and a content-based attention mechanism to predict the placements, as shown in Fig. 4. Traditional sequence models encode the input information into a fixed-length vector. In a DNN computation graph, there are usually thousands of operations and it is difficult for the model to compress all the necessary information into a fixed-length vector. In contrast, the attention mechanism encodes the input sentence into a sequence of vectors and chooses a subset of these vectors adaptively while decoding, thus the model can make better use of the input information of the encoder. Our model can be divided into two parts: encoder and decoder. The details are as follows.

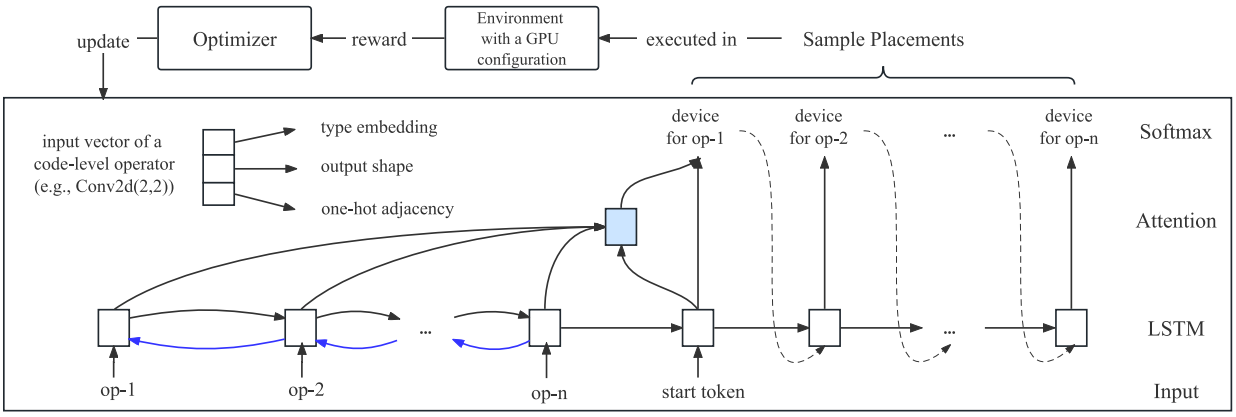


Fig. 4. Architecture of the device placement model. Each operator (op-i for short, $i \in [1, n]$) in the DNN inference job is placed based on the model sampling and executed on the customized TensorFlow in each iteration of DRL.

Our encoder is a bidirectional RNN and its input is the sequence of operations of the input graph, which is in topological order. We hope that our model can learn not only each operation's output but also input information, so we use bidirectional RNN as the encoder. We embed the operations by concatenating their information (including three attributes: type, output shape and adjacency information). The type of an operation describes its underlying computation. We use operations' types at the code level, such as Conv2D, and store a tunable embedding vector for each type. We also collect the size of each operation's list of output tensors and change them into a fixed-size zero-padded list called the output shape. The adjacency information of the input graph is constructed as an one-hot encoding vector that represents the operations that are direct inputs and outputs to each operation. Finally, the input vector of each operation is the concatenation of its type, output shape and adjacency vector.

The decoder is an attentional LSTM with a fixed number of time steps. The number of time steps is equal to that of input operations in the DNN inference model. The decoder outputs the GPU devices for the operation at the same encoder time step and each GPU device has its own embedding vector. The output of the decoder's one time step is fed as input to the next decoder time step because there are no correct labels in our problem and the model should predict the device according to the previous information.

D. Probing-Informed Block Multiplexing

In practice, we find that the computation-intensive nature of the foundation models brings additional challenges regarding the cost-efficiency of the deployment. Specifically, large amounts of inference trials need to be run with the joint adoption of BO and DRL, and in each trial the startup and inference of the foundation model (which is time-consuming) are performed, leading to high search overhead.

To uncover the reason why the foundation model inference trials in AutoDeep are time-consuming, we perform the basic-block code instrumentation to the internals of TensorFlow in order to dynamically trace the runtime inference trial. In particular, a basic block is defined as a code sequence with no branches in except to the entry and no branches out except at the exit. A typical feature of a basic block is that as soon as the first instruction in the block is executed, all the codes within the block are executed only once in sequence. Keeping this in mind, we scan over the key TensorFlow function calls in the trial process and trace back to the internal code of each function (mainly located in $tensorflow/tensorflow/core/common\_runtime$ and $tensorflow/tensorflow/core/graph$ in TensorFlow r1.4), and mark block boundaries which are instructions that may either begin or end a basic block. Between each pair of adjacent ending and starting boundaries, we instrument analysis code as probes to record key runtime information (including the functionality, input/output, and time overhead) of the ending block. In this way, the core functions of the trial are decomposed into basic code blocks, and the critical path of the trial process can be clearly observed. After functional and statistical analysis of the blocks, we notice that there exist heavy startup blocks (including hardware detection, computation graph building, and CUDA initialization), which serve as the preparation phase of the inference in TensorFlow. Such blocks are independent of the trial input (i.e., the device placement plan) but contribute more than 98% of the time spent by the trial. Fig. 5 further shows the comparison between the startup time and the inference time of eight popular DNN models. The inference time is usually faster than the startup time by one or two orders of magnitude.

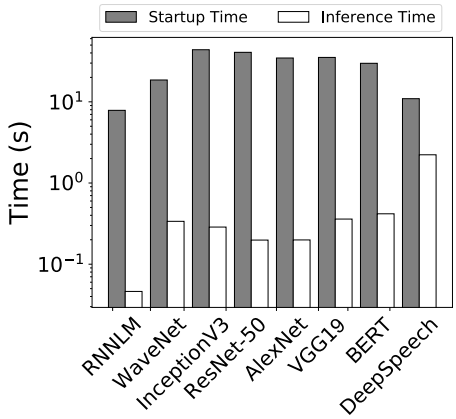


Fig. 5. Startup time and inference time of various DNN models.

TABLE III
COMPARISON OF THE INITIALIZATION TIME AND INFERENCE TIME FOR ONE-TIME BERT INFERENCE WITH A NVIDIA RTX 2080TI GPU USING DIFFERENT FRAMEWORKS. THE STARTUP RATIO IS THE PROPORTION OF THE INITIALIZATION TIME IN THE INFERENCE TRIAL EXECUTION TIME

Platform	Initialization Time (s)	Inference Time (s)	Startup Ratio
TensorFlow	9.44	0.16	98%
PyTorch	5.44	0.17	97%
MXNet	3.88	0.08	98%
PaddlePaddle	7.86	0.19	98%

Based on the above observation, we propose the probing-informed block multiplexing (PIBM) mechanism to accelerate the iterative search of device placement plans in AutoDeep. We first utilize the probes instrumented in the critical path of the trial process to precisely identify the startup blocks. To avoid repeating the heavy startup executions, we then reorganize the iterative trials into one program, where the startup blocks are executed first in a single pass so that other iteratively-executed blocks can reuse the startup basis. In this way, the overhead of the redundant startups after the change of the input device placement plan can be avoided.

To make clear the potential adaptability of the PIBM mechanism to other mainstream deep learning frameworks (including PyTorch, MXNet, PaddlePaddle, etc.), we study their internals and further conduct benchmark tests based on each of them, as shown in Table III. It is observed that the initializations of all the tested frameworks take more than 97% of the time spent by one inference of BERT. In fact, hardware detection and CUDA initialization are required in all frameworks based on NVIDIA GPUs [47], and NVIDIA GPUs are more powerful with better technical supports than other GPUs regarding DNN inference on these frameworks [48].

As for computation graph building, there are mainly two types: one is the static graph represented by TensorFlow, and the other is the dynamic graph represented by PyTorch. The static graph needs to be completely built first and then executed (leading to better runtime performance but poorer user-friendliness), while the dynamic graph is defined on the fly via the actual forward computation and thus it is easier to program and debug with relatively inferior runtime performance. This explains why TensorFlow suffers from longer initialization than PyTorch in our tests, and it can also be indicated that PIBM can be effective for dynamic-graph-based frameworks like PyTorch according to the startup ratio column in Table III. To conclude, the proposed PIBM mechanism should be mostly applicable to other mainstream deep learning frameworks.

E. Prototype Implementation

We implement AutoDeep on TensorFlow r1.4 [49] with 1500 lines of code. Given the fact that the graph substitution can be pre-executed before the joint searching of BO and DRL, we use an independent python script to optimize the computation graph of the target model with TASO graph substitution and convert the onnx-format graph-optimized model to a TensorFlow model. Since TensorFlow has no APIs for extracting the computation graph and changing the device placement, we customize TensorFlow to add these APIs so that AutoDeep can interact with TensorFlow to get and change the device placement plan. To accelerate the search of device placement, we introduce the aforementioned PIBM mechanism in TensorFlow to mitigate the overhead of rebooting a job after the device placement plan is changed. The core functions of TensorFlow's computation graph partition is in

$$tensorflow/tensorflow/core/graph/graph\_partition.cc.$$

The function *Partition()* is used to execute graph partition. We modify *Partition()* so that it can read our placement file, which is generated by our model and formed as $(operator\_name, device\_id)$, and apply the placement before inference. In order to obtain necessary information and change the device placement in TensorFlow, we introduce three APIs:

- *DumpComputationNode()*: to dump the operator information (e.g., type, name and assigned device) of each node;
- *DumpTensorShape()*: to dump the input or output tensor dimensions of directed edges (including control edges) of the computation graph;
- *SetDevicePlacement(new_plan)*: to replace the device placement plan in TensorFlow with a new plan.

The first two APIs extract the graph information, which is the input of the Seq2Seq model. The third API sets the device placement plan with the decisions given by the DRL model.

Without using the *SetDevicePlacement* API we introduced, by default, the device placement should be decided when the computation graph is built before the inference. When the device placement is changed by our DRL algorithm, the trial should be restarted to rebuild the computation graph with the new placement plan, which incurs the significant overhead of startup time, as illustrated in Sec IV-D.

In order to speed up our training process, we implement the PIBM mechanism in *SetDevicePlacement* API to change the device placement in an on-demand manner. After sampling the inference time of one placement, the program will hang to wait for a new device placement plan derived from the DRL model. In this way, the startup overhead after the change of device placement can be avoided.

V. PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness of AutoDeep using both commonly-used lightweight DNN models (i.e., RNNLM, InceptionV3, NMT, and ResNeXt-50) and well-known foundation models (i.e., BERT, VGG19, Transformer, and GPT-2). Through the experiments under real cloud environments, we illustrate in a fine-grained manner how AutoDeep finds the cloud configuration, and compare the quality of the device placements of AutoDeep and non-trivial baselines. The highlights are:

- Under the same cloud configuration, AutoDeep improves the inference speed for lightweight DNN models and foundation models by up to 68% and 23% with 98% and 90% lower optimization overheads, respectively.

TABLE IV
CONFIGURATION DETAILS

CPU	GPU	GPU Number	Price (USD/hour/GPU)
Core i7-5930K	GTX 980Ti	1-3	0.56
Core i7-6850K	GTX 1080	1-4	0.70
Xeon E5-2690 v4	P100	1-4	2.07
Xeon E5-2690 v3	K80	1-4	0.90

- Given a fixed search time limit, AutoDeep finds the cloud configuration satisfying the QoS constraint with up to 15% lower inference cost for lightweight DNN models and 47% for foundation models.
- AutoDeep saves up to 57% and 38% search time on light weight DNN models and foundation models, respectively.

A. Testbed Experiment

Setup: We deploy AutoDeep in Microsoft Azure cluster and our local testbed. There are 15 cloud configurations including NVIDIA K80, NVIDIA P100, NVIDIA GTX 1080 and NVIDIA 980Ti. The detailed configurations and their price are listed in Table IV.

Workloads: To test the performance of AutoDeep, we use both popular lightweight DNN models (i.e., InceptionV3, ResNeXt-50, RNNLM, and NMT) and foundation models (i.e., VGG19, Transformer, BERT, and GPT-2) in computer vision and natural language processing.

- Inception-V3 [28] is one of the most popular DNN models for image classification and visual feature extraction. Note that the model is connected by multiple blocks. Each block consists of multiple branches of convolution layers and pooling layers. Thus, within each block, the operations on different branches can be computed in parallel. However, the barrier at the end of each block limits the potential for exploiting higher parallelism.
- ResNeXt-50 [29] is a 50-layer homogeneous neural network that reduces the number of hyperparameters required by conventional ResNet. The ResNeXt block in its architecture has a cardinality of 32.
- Recurrent Neural Network Language Model (RNNLM) has multiple LSTM layers [23]. Since the architecture is grid-based, this model has great potential to be executed in parallel on multiple devices. The LSTM cells can be executed as soon as their dependent outputs become available.
- Neural Machine Translation (NMT) [24] is an 8-layer sequence-to-sequence model with the encoder-decoder architecture. The structure of its computation graph is also grid-based with huge potential for model parallelism.
- VGG19 [30] is a popular foundation model trained on ImageNet [50] for image classification. Since VGG19 has an inherently tightly-coupled structure, we explore its potential of model parallelism by decoupling its structure through the DeCNN model parallelism optimization approach [51], which has negligible effect on the model accuracy. In this way, the inference of VGG19 can be further accelerated through the device placement optimization in AutoDeep.
- The transformer [26] is a foundation model that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease. Its architecture follows an encoder-decoder structure and it relies on the parallel multi-head attention mechanism.
- Bidirectional Encoder Representations from Transformers (BERT [25]) is a typical foundation model for natural language processing with a large number of operations and a complex structure. The original English-language BERT has 2 versions of different sizes (BERT-Base and BERT-Large). We use BERT-Base with a maximum sequence length of 325 and a batch size of 24.
- Generative Pre-trained Transformer 2 (GPT-2) [27] is a language foundation model by OpenAI. Its architecture allows for greatly increased parallelization.

B. Heuristic Baselines

To compare the performance of AutoDeep, we further implement several heuristic baselines in our experiments. Since AutoDeep is the first algorithm that jointly optimizes the cloud configuration and the inference speed, we choose the baselines only achieving one of the two objectives.

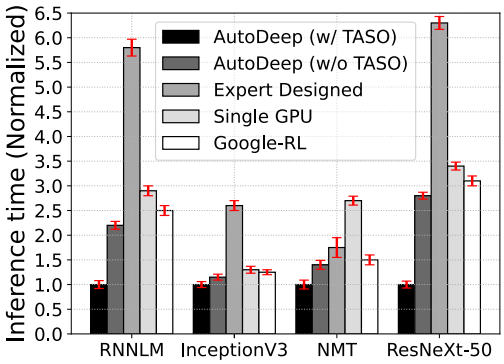
For the cloud configuration searching, we choose the following baselines:

- **Genetic Algorithm (GA) [52]:** GA is inspired by the process of natural selection. Its genetic representation is defined as the vector $\mathbf{D}_k$ in Sec. IV-B, and its fitness function is set as the expected improvement acquisition function of BO.
- **Differential Evolution (DE) [53]:** The representation and the fitness function in DE share the same settings with GA. Different from GA, the mutation in DE is based on the differential results of the representation vectors.
- **Lowest Cost First (LCF):** LCF follows the greedy strategy and tries the cloud configurations in the ascending order of their unit price. Since our goal is to find the configuration that satisfies the QoS constraint with the minimum inference cost, it stops searching until the QoS constraint is satisfied.
- **Uniform:** Uniform tries the cloud configurations with the uniform probability and stops until the search time exceeding a time limit.

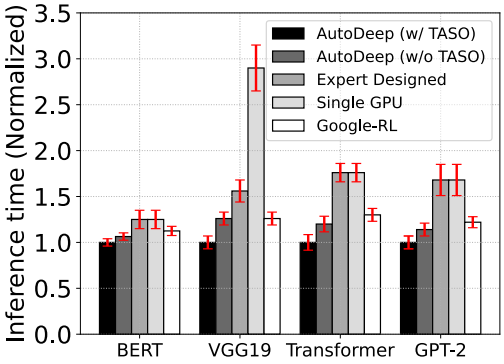
Note that knowing inference cost of a cloud configuration (i.e., inference time$\times$configuration unit price) requires revealing the inference performance, which is expensive and prior unknown. Thus LCF uses the unit price instead of inference cost when deciding the searching priority.

For inference acceleration using device placement, we choose the following baselines:

- **Expert Designed**: For lightweight DNN models, we mainly use the hand-crafted placements given by Mirhoseini et al. [17]. The Inception-V3 model and the ResNeXt-50 model are heuristically partitioned into the parts with almost the same number of layers. Each LSTM layer in the RNNLM/NMT model is put on a GPU device. For VGG19, we use the partitioning scheme



(a) Lightweight DNN models



(b) Foundation models

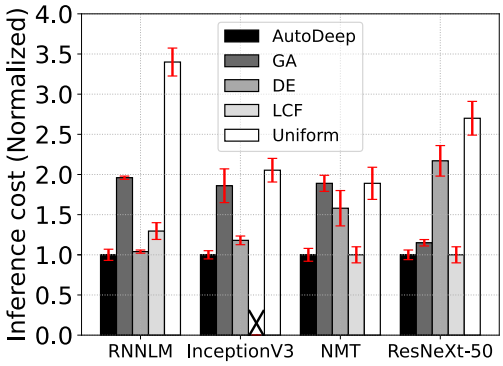
Fig. 6. The performance of device placement on four K80 GPUs.

given by Du et al. [51]. For language models, we use the BERT implementation given by Google and use the implementations of Transformer and GPT-2 given by HuggingFace [], which do not support model parallelism on multiple GPUs by default.

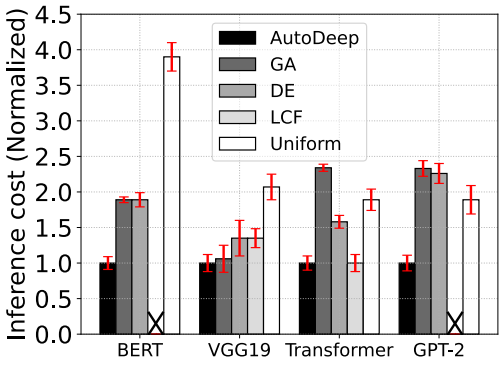
- **Google's RL-based Device Placement (Google-RL)**: The reinforcement learning-based approach proposed by Google that only considers the device placement under a fixed cloud configuration [31]. Different from our approach, Google-RL uses GraphSAGE as the graph encoder combined with a Transformer-based placer.
- **Single-GPU**: This placement executes the entire DNN model on a single GPU. We only place the operation to CPU when it has no GPU implementation.

TABLE V
COMPARISON OF OPTIMIZATION OVERHEADS

Model	Google-RL	AutoDeep (w/ TASO)	Speedup
RNNLM	9.5 hours	0.3 hours	31.7×
NMT	12.8 hours	0.4 hours	32×
InceptionV3	8.6 hours	0.2 hours	43.0×
ResNeXt-50	10.2 hours	0.3 hours	34×
BERT	8.3 hours	1.5 hours	5.5×
Transformer	9.3 hours	1.5 hours	6.2×
GPT-2	18.7 hours	1.9 hours	9.8×
VGG19	6.1 hours	1.0 hour	6.1×



(a) Lightweight DNN models



(b) Foundation models

Fig. 7. Inference cost under commercial-grade QoS constraints.

C. Experiment Results

Performance of Inference Acceleration. To evaluate the performance of the inference acceleration module in AutoDeep (the joint adoption of TASO graph substitution and the DRL-based device placement optimization), we first fix the cloud configuration to the server with four NVIDIA K80 GPUs. Since we additionally optimize the graph through TASO graph substitution compared with other baselines, we also take the optimization time overhead into consideration to fairly evaluate the cost-efficiency. Fig. 6 demonstrates the performance of AutoDeep (w/TASO), AutoDeep (w/o TASO), and the three baselines. The results are normalized to the inference time of the device placement derived by AutoDeep. Table V presents the optimization overheads of AutoDeep (w/TASO) and Google-RL. The data of AutoDeep (w/o TASO) is not listed since it is the same as the data of AutoDeep (w/TASO) when the data is shown in hours. This is because the graph optimization of TASO only takes a few seconds, while AutoDeep (w/o TASO) takes tens of minutes or even hours.

Thanks to the lightweight feature of TASO and the proposed PIBM mechanism, AutoDeep (w/ TASO) outperforms the baselines in both inference time of target models and optimization overheads. AutoDeep (w/ TASO) improves the inference time of lightweight DNN models and foundation models by up to 68% and 23%, and saves up to 98% and 90% optimization overheads compared with heuristic baselines, respectively. For lightweight DNN models, it may be surprising that the expert-designed device placement has worse performance than that in the single-GPU configuration. The

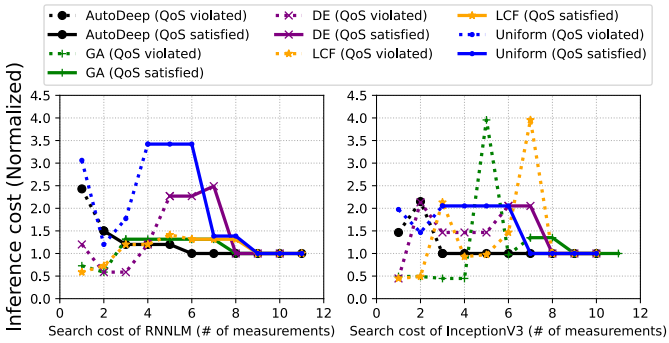
reason is that a human expert has no knowledge of the underlying GPU configuration when deciding the placement. Thus the human-designed device placement may not be suitable for the cloud configuration. Actually, the bad performance of human-designed device placement in our experiments is consistent with previous observations on DNN training jobs reported by Mirhoseini et al. [17]. Although Google-RL experienced an upgrade from the Seq2Seq model-based approach [17] to the encoder-placer structure [31], it does not find much better device placement plans than using a single GPU for lightweight DNN models but takes much longer optimization time than AutoDeep (w/ TASO).

For foundation models, although AutoDeep (w/o TASO) has similar performance with Google-RL, the integration of AutoDeep and TASO not only achieves up to 23% faster inference than Google-RL, but also reduces up to 90% optimization overheads compared with Google-RL. We believe this is because TASO brings additional potential on model parallelism when substituting the inefficient subgraphs, resulting in synergy effects on the inference acceleration. The expert-designed partitioning scheme of VGG19 outperforms the single-GPU baseline significantly, but it still has performance gaps compared to the DRL-based approaches.

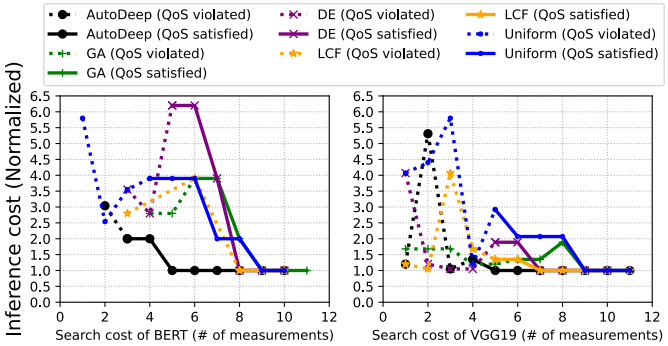
As the inference acceleration of AutoDeep (w/ TASO) outperforms all the baselines, we use the DRL-based algorithm combined with TASO graph substitution to speed up the model inference in each cloud configuration searching trial in the following evaluation.

Performance of Cloud Configuration Searching. Since the DRL-based algorithm combined with TASO graph substitution takes $> 99\%$ of the time spent by each cloud configuration searching trial in both AutoDeep and the baselines, we use the number of measurements as the search cost to evaluate the searching efficiency of AutoDeep. Specifically, we set the search time limit to 6 trials and compare the inference cost of the cloud configuration found by the three algorithms given commercial-grade QoS constraints. Fig. 7(a) and Fig. 7(b) depict the (normalized) inference cost of light weight DNN models and foundation models when satisfying the QoS constraint, respectively. AutoDeep achieves the best performance in all models. For lightweight DNN models and foundation models, AutoDeep reduces the inference cost by up to 15% and 47% compared to the best baselines, respectively. For BERT and GPT-2, LCF fails to find any feasible cloud configuration since they either run out of memory or take a long inference time that does not satisfy the QoS constraint in low-end cloud configurations. Besides, we further observe that with more relaxed QoS constraints, all five algorithms find cloud configurations with lower inference cost, and AutoDeep finds the most efficient one in all settings.

Dissecting the Search Efficiency. To understand why AutoDeep is more efficient on configuration searching, we dissect how the inference cost changes with more measurements. Given the local similarities between the graph structures of certain workloads, we select 4 typical workloads (2 lightweight DNN models and 2 foundation models) with significantly different graph structures for detailed analysis. Fig. 8(a) and Fig. 8(b) demonstrate the change of inference cost



(a) Lightweight DNN models



(b) Foundation models

Fig. 8. Inference cost with varying number of measurements.

(normalized to the best configuration) of lightweight DNN models (RNNLM and InceptionV3) and foundation models (BERT and VGG19), respectively. The QoS constraint is set to be close to $2\times$ of the inference time of the best configuration for each model. The dash lines show the inference cost before the algorithms find a QoS-satisfied cloud configuration.

For RNNLM, AutoDeep finds the feasible configuration that satisfies the QoS constraint at the 2nd trial, and the optimal configuration at the 6-th trial. Both GA and DE find the best configuration at the 8-th trial, while GA finds the feasible configuration at the 3-rd trial but stalls in the following 4 trials. Both Uniform and LCF find the best configuration at the 9-th trial, while Uniform finds the feasible configuration at the 4-th trial with a high inference cost. Similarly, for Inception-V3, both AutoDeep and Uniform find the feasible configuration at the 3rd trial but AutoDeep finds the optimal configuration. Since Inception-V3 performs much better on the P100 GPUs, which are more expensive than the other configurations, LCF performs not well as it finds the feasible (and the optimal) configuration at its 8-th trial. Both GA and DE find the optimal configuration after the 8-th trial, since such evolution-based heuristics require more warm-up measurements.

For BERT, we hide the trials in which the BERT model runs out of memory with the corresponding configurations. As shown in Fig. 8(b), AutoDeep reaches the feasible configuration that satisfies the QoS constraint at the 3rd trial, and the optimal configuration at the 5-th trial. DE finds the feasible configuration at the 5-th trial with a very high inference cost, but finds the optimal configuration within the following 3 trials. Since BERT requires relatively large GPU memory, high-end configurations are preferred considering the QoS

constraint. This leads to the bad performance of LCF as it finds the first feasible and also the best configuration at the 8-th trial. For VGG19, AutoDeep also exhibits the best performance as it finds the feasible configuration at the 4-th trial and the optimal configuration at the 5-th trial. DE, LCF, and Uniform all find the feasible configuration at the 5-th trial, and DE and LCF find the best one with 22% lower search cost compared with Uniform. For both models, GA and Uniform perform worst as they both find the optimal configurations at their 9-th trials. Compared with the best baselines, AutoDeep finds the optimal configurations with 38% and 29% lower search costs for BERT and VGG19, respectively.

Discussing the Scalability of AutoDeep. Due to the limitation of our testbed, there are 15 configurations included in the performance evaluation. However, if the configuration pool is much larger with more than 100 candidate VMs, we believe AutoDeep can also be well applied to find a near-optimal solution in a reasonable time. In fact, even for the inference of large foundation models, one up-to-date high-end GPU device (e.g., NVIDIA A100) is capable of satisfying the commercial-grade QoS requirement (but is too costly for users). Configurations with multiple high-end GPU devices are highly over-configured. When exploring into such configurations in the searching process AutoDeep turns back fast without wasting additional overheads, since in such cases BO prevents high inference cost and DRL converges fast. Besides, the PIBM mechanism significantly speeds up the searching of AutoDeep by up to 98%. Above all, we believe AutoDeep has the potential to derive a near-optimal solution in several hours.

VI. Related Work

Our work integrates cloud configuration searching with DNN acceleration. AutoDeep determines the cloud configuration using Bayesian Optimization and accelerate DNN using graph partition based on the technique of DRL. We review related literature in this section.

Cloud Configuration. Choosing the right cloud configuration for DNN inference is essential to the quality of service and commercial competitiveness [54]. Early work such as [55] develops a platform called CloudAdvisor to explore various cloud configurations which are recommended based on user preferences. CherryPick is a system designed in [15] to choose the best cloud configurations for big data analytics. These methods are for big data applications but they ignore the characteristics of DNN inference for the deployment of real-time DNN-driven services, in a sense that even in a fixed configuration there exist different device placements with different inference speeds. Our approach can find both the cost-efficient configuration and its appropriate device placement satisfying the QoS constraint.

Parameter Tuning with Bayesian Optimization. Bayesian Optimization is one of the promising techniques used for parameter tuning. It has been used in searching optimal DNN hyperparameters for higher accuracy [40], [56], and finding the best cloud configuration for big-data analytics [15], [57]. These works usually use BO to optimize the objective function without considering any constraint. AutoDeep is a parallel work which jointly optimizes the cloud configuration and the device placement. Moreover, AutoDeep not only minimizes the inference cost but also considers the QoS constraint when optimizing the DNN inference model.

DNN Acceleration. Performing inference on DNN models meets the requirement of low-latency in practice [17], [18], [58], [59], [60]. In 2022, Unger et al. [61] propose unified parallel computation graph representations to jointly optimize algebraic transformations and parallelization in distributed DNN training. They pursuit extreme DNN training efficiency without considering the economic cost of the device configurations, while AutoDeep brings the monetary cost into modeling to optimize the cost-efficiency of the cloud deployment for the real-time foundation model inference since cost-efficiency is crucial for such services. Zheng et al. [62] design a number of compilation passes to jointly optimize inter-operator and intra-operator parallelisms. Their modeling is static given fixed cluster configuration information, while AutoDeep is more practical with the joint modeling informed by inference performance in the dynamic execution environments of the cloud configurations. From 2017 to 2021, Mirhoseini et al. [17], Gao et al. [18], Zhou et al. [31], and Lan et al. [63] propose various DRL-based approaches to optimize the device placement for DNN training with similar modeling algebras. Our work applies this technique to accelerate DNN inference and combines it with BO to compute the cost-efficient cloud configuration in consideration of the runtime environment.

Conventional Graph Partition. Graph partition has been intensively studied in various domains, such as sensor networks [64]. Existing works [36], [65], [66], [67], [68] start from an initial partition and use several refinement methods to explore similar partitions to improve after iterations. Other works such as [37] and [69] perform spectral analysis on the matrix representation of the graph and also employ an iterative refinement approach to partition them. However, for DNN computation graphs, they do not work well because it is hard to construct cost models for the graphs under all kinds of cloud configurations.

VII. Conclusion

In this paper, we study the problem of automating the cloud deployment for real-time foundation model inference. We propose AutoDeep that can adaptively choose the cost-efficient cloud configuration and the device placement for foundation model inference jobs. We further enhance the cost-efficiency of AutoDeep's deployment with TASO graph substitution and the PIBM mechanism. We implement AutoDeep with TensorFlow and conduct extensive experiments on Microsoft Azure. The experiments with both popular lightweight DNN models and foundation models show that AutoDeep can significantly improve the inference speed, the search speed, and reduce the inference cost compared with non-trivial baselines, including Google's RL-based method for device placement and Differential Evolution for cloud configuration.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[3] G. Hinton et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[4] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proc. ICML*, 2014, pp. 1764–1772.

[5] A. Hannun et al., "Deep Speech: Scaling up end-to-end speech recognition," 2014, *arXiv:1412.5567*.

[6] W. Chan, N. Jaitly, Q. Le, and O. Vinyals, "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 4960–4964.

[7] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NIPS*, 2014, pp. 3104–3112.

[8] K. Cho et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," 2014, *arXiv:1406.1078*.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.

[10] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[11] R. Bommasani et al., "On the opportunities and risks of foundation models," 2021, *arXiv:2108.07258*.

[12] D. Wang, W. Cao, J. Li, and J. Ye, "DeepSD: Supply-demand prediction for online car-hailing services using deep neural networks," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 243–254.

[13] Q. Ye, Z. Zhang, and R. Law, "Sentiment classification of online reviews to travel destinations by supervised machine learning approaches," *Expert Syst. Appl.*, vol. 36, no. 3, pp. 6527–6535, Apr. 2009.

[14] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, Dec. 2017, pp. 109–120.

[15] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. USENIX NSDI*, vol. 2, 2017, pp. 469–482.

[16] H. Lin et al., "Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility," in *Proc. ACM MobiCom*. New York, NY, USA: ACM, 2023, pp. 1–17, Art. no. 45, doi: 10.1145/3570361.3613259.

[17] A. Mirhoseini et al., "Device placement optimization with reinforcement learning," 2017, *arXiv:1706.04972*.

[18] Y. Gao et al., "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. ICML*, 2018, pp. 1662–1670.

[19] Y. Li, Z. Han, Q. Zhang, Z. Li, and H. Tan, "Automating cloud deployment for deep learning inference of real-time online services," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Jul. 2020, pp. 1668–1677.

[20] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 86–96, Jul. 2002.

[21] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 47–62.

[22] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016, pp. 265–283.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[24] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. ICLR*, 2015, pp. 1–15.

[25] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.

[26] A. Vaswani et al., "Attention is all you need," in *Proc. NIPS*, vol. 30, 2017, pp. 1–11.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *Proc. OpenAI Blog*, vol. 1, 2019, pp. 1–24.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[29] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5987–5995.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[31] Y. Zhou et al., "GDP: Generalized device placement for dataflow graphs," 2019, *arXiv:1910.01578*.

[32] Y. Li et al., "A nationwide study on cellular reliability: Measurement, analysis, and enhancements," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 597–609.

[33] H. Zhou, M. Li, N. Wang, G. Min, and J. Wu, "Accelerating deep learning inference via model parallelism and partial computation offloading," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 2, pp. 475–488, Feb. 2023.

[34] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX OSDI*, 2018, pp. 595–610.

[35] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 184–200.

[36] F. Pellegrini, "Distillating knowledge about SCOTCH," in *Proc. Dagstuhl Seminar*, 2009, pp. 1–12.

[37] G. Karypis and V. Kumar, "METIS—Unstructured graph partitioning and sparse matrix ordering system," Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, White Paper Version 2.0, 1995.

[38] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016, *arXiv:1611.09940*.

[39] M. Pelikan, D. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian optimization algorithm," in *Proc. GECCO*, 1999, pp. 525–532.

[40] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. NIPS*, 2012, pp. 2951–2959.

[41] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Global Optim.*, vol. 13, no. 4, pp. 455–492, Dec. 1998.

[42] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, "Bayesian optimization with inequality constraints," in *Proc. ICML*, 2014, pp. 937–945.

[43] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*. Berlin, Germany: Springer, 2008, pp. 337–340.

[44] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN computation with relaxed graph substitutions," in *Proc. MLSys*, vol. 1, 2019, pp. 27–39.

[45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[46] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, May 1992.

[47] NVIDIA. (2023). *CUDA Zone*. [Online]. Available: https://developer.nvidia.com/cuda-zone

[48] M. J. White. (2023). *AMD is Losing the AI Battle, and It's Time It Started to Worry*. [Online]. Available: https://www.digitaltrends.com/computing/why-amd-is-losing-the-ai-battle/

[49] R. Mayer, C. Mayer, and L. Laich, "The TensorFlow partitioning and scheduling problem: It's the critical path!" in *Proc. 1st Workshop Distrib. Infrastruct. Deep Learn.*, Dec. 2017, pp. 1–6.

[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, Jun. 2009, pp. 248–255.

[51] J. Du et al., "Model parallelism optimization for distributed inference via decoupled CNN structure," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1665–1676, Jul. 2021.

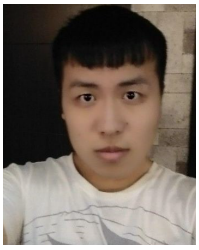
[52] S. Mirjalili, "Genetic algorithm," in *Evolutionary Algorithms and Neural Networks*. Cham, Switzerland: Springer, 2019, pp. 43–55.

[53] K. V. Price, "Differential evolution," in *Handbook of Optimization*. Berlin, Germany: Springer, 2013, pp. 187–214.

[54] Z. Li, Y. Zhang, and Y. Liu, "Towards a full-stack devops environment (platform-as-a-service) for cloud-hosted applications," *Tsinghua Sci. Technol.*, vol. 22, no. 1, pp. 1–9, Feb. 2017.

[55] G. Jung, T. Mukherjee, S. Kunde, H. Kim, N. Sharma, and F. Goetz, "CloudAdvisor: A recommendation-as-a-service platform for cloud configuration and pricing," in *Proc. IEEE 9th World Congr. Services*, Jun. 2013, pp. 456–463.

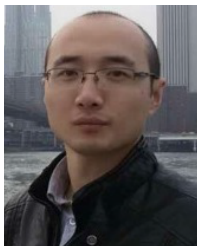
[56] J. Bergstra, D. Yamins, and D. Cox, "Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms," in *Proc. Python Sci. Conf.*, Jan. 2013, pp. 13–20.
[57] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Low-level augmented Bayesian optimization for finding the best cloud VM," 2017, *arXiv:1712.10081*.
[58] U. Picchini and J. L. Forman, "Accelerating inference for diffusions observed with measurement error and large sample sizes using approximate Bayesian computation," *J. Stat. Comput. Simul.*, vol. 86, no. 1, pp. 195–213, Jan. 2016.
[59] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency RNN inference with cellular batching," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–15.
[60] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," 2018, *arXiv:1806.01683*.
[61] C. Unger et al., "Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization," in *Proc. USENIX OSDI*, 2022, pp. 267–284.
[62] L. Zheng et al., "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022, *arXiv:2201.12023*.
[63] H. Lan, L. Chen, and B. Li, "Accelerated device placement optimization with contrastive learning," in *Proc. 50th Int. Conf. Parallel Process.*, Aug. 2021, pp. 1–10.
[64] L. Wang, Z. Yu, D. Yang, T. Ku, B. Guo, and H. Ma, "Collaborative mobile crowdsensing in opportunistic D2D networks: A graph-based approach," *ACM Trans. Sensor Netw.*, vol. 15, no. 3, pp. 1–30, Aug. 2019.
[65] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.
[66] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
[67] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Design Automat. Conf.*, Jun. 1982, pp. 175–181.
[68] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I—Graph partitioning," *Oper. Res.*, vol. 37, no. 6, pp. 865–892, Dec. 1989.
[69] L. Hagen and A. B. Kahng, "New spectral methods for ratio cut partitioning and clustering," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 11, no. 9, pp. 1074–1085, Sep. 1992.



Yang Li received the B.S. and M.Eng. degrees from the School of Software, Tsinghua University, Beijing, China, in 2018 and 2021, respectively, where he is currently pursuing the Ph.D. degree. He has published several papers in recognized conferences, such as SIGCOMM, MobiSys, and INFOCOM. His current research interests include cloud computing, big data analysis, network measurement, and machine learning. He won the Best Student Paper Award from SIGCOMM in 2021.

Zhenhua Li (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science and technology from Nanjing University in 2005 and 2008, respectively, and the Ph.D. degree in computer science and technology from Peking University in 2013. He is currently a Tenured Associate Professor with the School of Software, Tsinghua University. His current research interests include network measurement, mobile networking/emulation, and cloud computing/storage. He is a Senior Member of ACM.

Zhenhua Han received the B.Eng. degree in electronic and information engineering from the University of Electronic Science and Technology of China in 2014 and the Ph.D. degree in computer science from The University of Hong Kong in 2020. He is currently a Senior Researcher with Microsoft Research Asia. His current research interests include cloud computing, cluster scheduling, and machine learning systems.



Quanlu Zhang received the B.Sc. degree in computer science and technology from the Harbin Institute of Technology in 2012 and the Ph.D. degree in computer science and technology from Peking University in 2017. He is currently a Principal Researcher with Microsoft Research. His current research interests include AutoML systems, deep learning compilers, distributed execution of deep learning models, resource management, and the job scheduling of GPU clusters.

Xiaobo Ma (Member, IEEE) received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, Xi'an, China, in 2014. He was a Postdoctoral Research Fellow with The Hong Kong Polytechnic University, Hong Kong, in 2015. He is currently a Professor with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University. His current research interests include Internet measurement and cybersecurity.