Automating Cloud Deployment for Deep Learning Inference of Real-time Online Services

Yang Li[1,2*], Zhenhua Han[2,3*], Quanlu Zhang[2], Zhenhua Li[1✉], Haisheng Tan[4]
[1]School of Software, BNRist, and KLISS MoE, Tsinghua University, China
[2]System Research Group, Microsoft Research Asia, China
[3]Department of Computer Science, University of Hong Kong, China
[4]School of Computer Science and Technology, University of Science and Technology of China
{*liyang14thu, hzhua201, lizhenhua1983, karl.haisheng.tan*}*@gmail.com, Quanlu.Zhang@microsoft.com*

***Abstract*—Real-time online services using pre-trained deep neural network (DNN) models, e.g., Siri and Instagram, require low-latency and cost-efficiency for quality-of-service and commercial competitiveness. When deployed in a cloud environment, such 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 a DNN model to multiple computation devices like GPUs and CPUs. Currently, the deployment mainly relies on service providers' manual efforts, which is not only onerous but also far from satisfactory oftentimes (for a same service, a poor deployment can incur significantly more costs by tens of times). In this paper, we attempt to automate the cloud deployment for real-time online DNN inference with minimum costs under the constraint of acceptably low latency. This attempt is enabled by jointly leveraging the Bayesian Optimization and Deep Reinforcement Learning to adaptively unearth the (nearly) optimal cloud configuration and device placement with limited search time. We implement a prototype system of our solution based on TensorFlow and conduct extensive experiments on top of Microsoft Azure. The results show that our solution essentially outperforms the non-trivial baselines in terms of inference speed and cost-efficiency.**

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

I. Introduction

Deep learning is currently the *de facto* standard technique used in various areas, such as computer vision [1], [2], speech recognition [3]–[6], and natural language processing [7]–[10]. In recent years, deep neural network (DNN) models have become crucial back-end support of many real-time online services [11], [12], such as Siri and Instagram. As the accuracy is ensured by the DNN 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 [13], especially for a complicated DNN 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 DNN 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., over a hundred) *cloud configurations* (i.e., specific types of VM instances with different hardware and OSes) 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 cloud configuration [14], 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 speed.

Besides searching for an appropriate cloud configuration, service providers need to consider *computation parallelism* in the meantime. Specifically, they should explicitly place the operations of a neural network on multiple computation devices like GPUs and CPUs to accelerate the DNN inference [15], [16]. Consequently, a considerate device placement plan is also called for, and in practice it is also 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 when the DNN model has a large computation graph [15] (which contains a set of operations with inter-operation dependencies).

Given the computation graph of a DNN model, finding the optimal cloud configuration and device placement is highly challenging, because it involves a huge search space – the joint space of all available cloud configurations and all possible device placement plans. Therefore, we pose a critical question for today's DNN-driven real-time online services: *how can we automatically determine the cloud configuration and device placement for the inference of a DNN model, so as to minimize the inference cost while satisfying the inference time constraint?* Here inference cost is defined as 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 this paper, we answer the above question with our proposed solution, named AutoDeep. 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

* Co-primary authors. Zhenhua Li is the corresponding author.

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.

We implement AutoDeep based on TensorFlow [17] and build the prototype system on top of VM instances rented from Microsoft Azure. We evaluate the effectiveness of AutoDeep through extensive experiments using typical workloads for natural language processing (i.e., RNNLM [18]) and online image classification (i.e., Inception-V3 [19]). The experiment results show that AutoDeep improves the inference speed by 43% for RNNLM and 14% for Inception-V3, compared with the non-trivial baselines such as Google's RL-based device placement [15]. Moreover, AutoDeep essentially reduces 23% of the inference cost and 33% of the search time for RNNLM, and saves 51% of the inference cost and 57% of the search time for Inception-V3, compared with the heuristic baselines such as greedy search.

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 5 popular DNN models across all cloud configurations in a cloud provider. We observe a poor cloud configuration can incur up to 16 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.

TABLE I
Inference cost (10000 times) of different models across different cloud configurations

Model	Min Inference Cost	Max Inference Cost
RNNLM	\$0.17	\$1.15
Inception-V3	\$0.40	\$6.39
VGG16	\$0.58	\$7.26
ResNet-50	\$0.60	\$4.74
AlexNet	\$0.59	\$4.45

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 the cloud service providers have heterogeneous configurations on the number of CPU cores, the RAM size, 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 performance running inside the VMs. It relies on users to pick the suitable cloud configuration for their workloads.

However, the performance of a DNN model is very complicated [20]. 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 [21]. Existing algorithmic solvers for graph partition, such as Scotch [22] and Metis [23], do not work for this problem, because these algorithms need accurate cost models, which is almost impossible for the complex DNN models.

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 model'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 of the black-box optimization 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 [14]. However, they cannot be simply applied to solve the combinatorial device placement problem due to the extremely large and exponentially growing search space. Thus, we seek to the Deep Reinforcement Learning (DRL), which has been proved to be effective for solving the large-scale combinatorial optimization problem [24] 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 $D_k = \{d_{k,1}, d_{k,2}, ..., d_{k,|D_k|}\}$. Each device $d_{k,i}$ can be a CPU core or a GPU device. We assume the memory and disk space in all cloud configurations are sufficient enough, so 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.

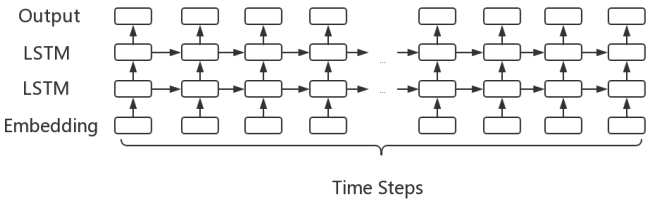

Fig. 1. The computation graph of RNNLM.

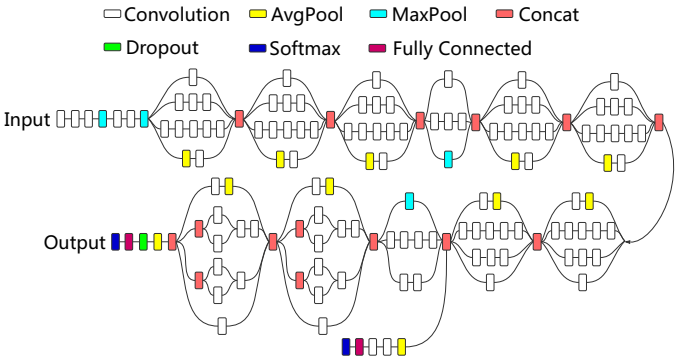

Fig. 2. The computation graph of Inception-V3.

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 [17]). Fig. 1 and Fig. 2 demonstrate the computation graphs of two popular DNN models, RNNLM [18] and Inception-V3 [19] 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 $\mathcal{O} = \{o_1, o_2, ...o_{n_o}\}$). There is a set of directed edges in $\mathcal{G}$. Each edge connects two operations that represent the dependency relationship of these two operations. 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 $\mathcal{P} = (p_1, p_2, ...p_{n_o})$ 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.

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):

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

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

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 the final 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 DNN 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.

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

A. Overview of AutoDeep

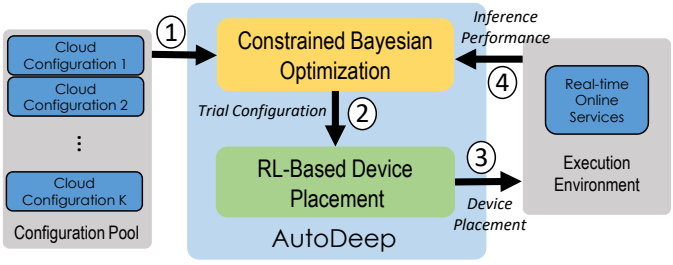

Fig. 3. Architectural overview of the AutoDeep framework.

AutoDeep iteratively finds the cost-efficient cloud configuration and the device placement. Fig. 3 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 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.

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 [25] 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 [26] 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. [27].

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.

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

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

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. 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

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 sequence-to-sequence 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:

$$J(\theta) = \mathbb{E}_{T(\mathcal{G},\mathcal{P},D_k)\sim\pi(\mathcal{P}|\mathcal{G};\theta)}[(\mathcal{P})|\mathcal{G}]. \quad (8)$$

The policy is defined by an attentional Seq2Seq model, which is introduced in detail below. The parameters in the network are learned by Adam optimizer [28] based on the

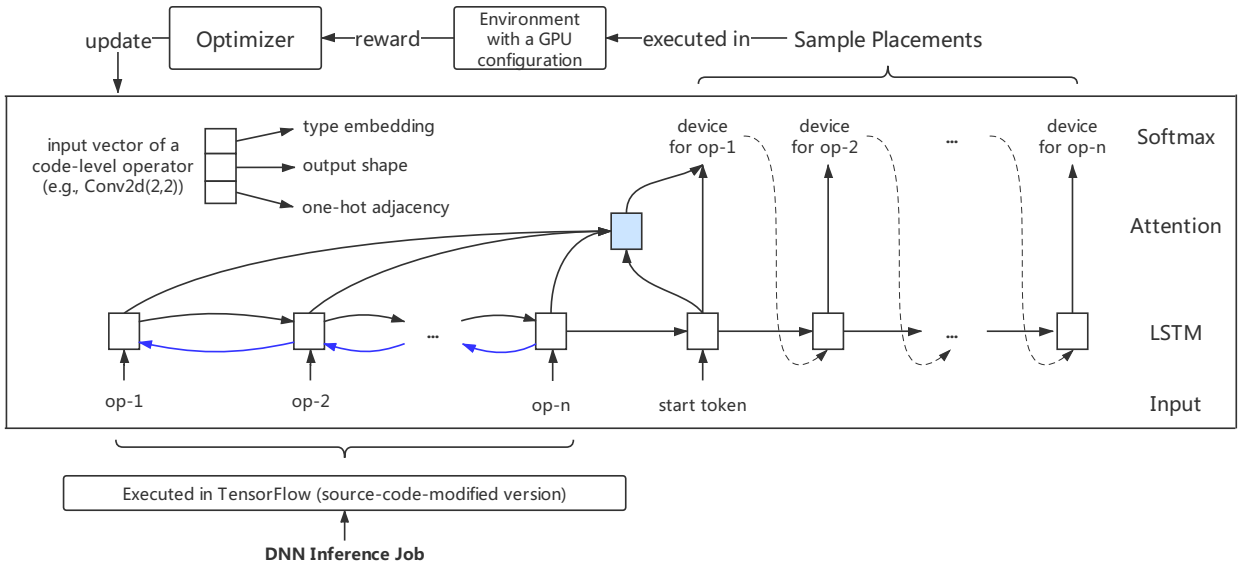

Fig. 4. Architecture of the device placement model.

REINFORCE equation [29], a commonly used policy gradient method, which is given as follows:

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

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

$$\nabla_\theta J(\theta) \approx \frac{1}{K}\sum_{i=1}^{K}((\mathcal{P}_i) - B) \centerdot \nabla_\theta logp(\mathcal{P}|\mathcal{G};\theta). \quad (9)$$

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 [18] 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.

Our encoder is a bidirectional RNN and the input of the encoder 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 choose operations' types at the code level, such as Conv2D and MaxPool, 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. We also construct the adjacency information of the input graph 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.

V. Performance Evaluation

In this section, we evaluate the effectiveness of AutoDeep using the popular DNN inference models. 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 device placement for RNNLM and Inception-V3 by 43% and 14%, respectively.
- Given a fixed search time limit, AutoDeep finds the cloud configuration satisfying the QoS constraint with 23% lower inference cost for RNNLM and 51% for Inception-V3.

TABLE III
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

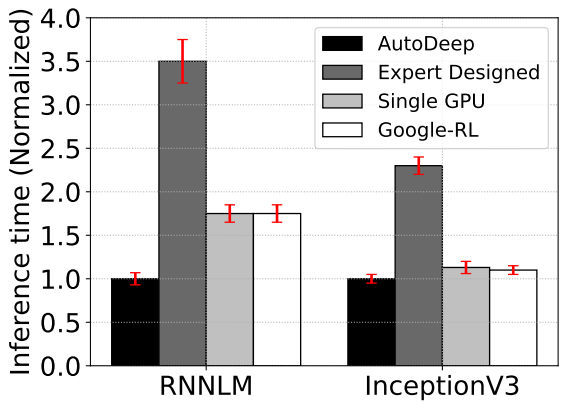

Fig. 5. The performance of device placement on four K80 GPUs.

- AutoDeep saves 33% and 57% search time on RNNLM and Inception-V3, 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 III.

Workloads: To test the performance of AutoDeep, we use two popular DNN models in computer vision and natural language processing. Fig. 1 and Fig. 2 illustrate the computation graphs of the two DNN models.

- Recurrent Neural Network Language Model (RNNLM) has multiple LSTM layers [18]. 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.
- Inception-V3 [19] 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.

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 device placement, we choose the baselines only achieving one of the two objectives.

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

- **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 the device placement, we choose the following baselines:

- **Expert Designed**: We use the hand-crafted placements given by Mirhoseini et al. [15]. For Inception-V3, the model is heuristically partitioned into the parts with almost the same number of layers. For RNNLM, we put each LSTM layer on a GPU device.
- **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 [15]. Different from our approach, Google-RL uses an unidirectional RNN in their encoder.
- **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.

C. Experiment Results

Performance of Device Placement. To evaluate the performance of the device placement, we first fix the cloud configuration to the server with four NVIDIA K80 GPUs. Fig. 5 demonstrates the performance of AutoDeep and the three baselines. The results are normalized to the inference time of the device placement derived by AutoDeep.

AutoDeep improves the inference time of the RNNLM and Inception-V3 model by 43% and 14% compared with heuristic baselines, respectively. 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

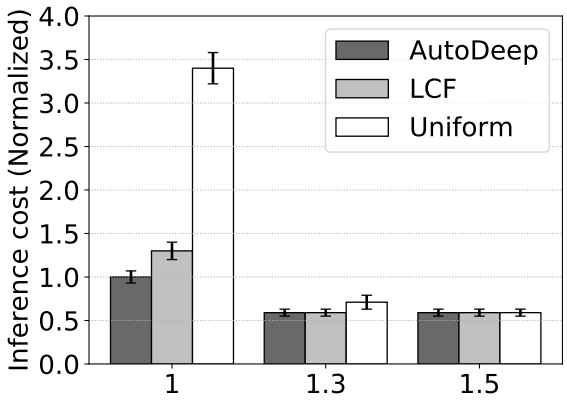

Fig. 6. Inference cost of RNNLM under varying QoS constraint.

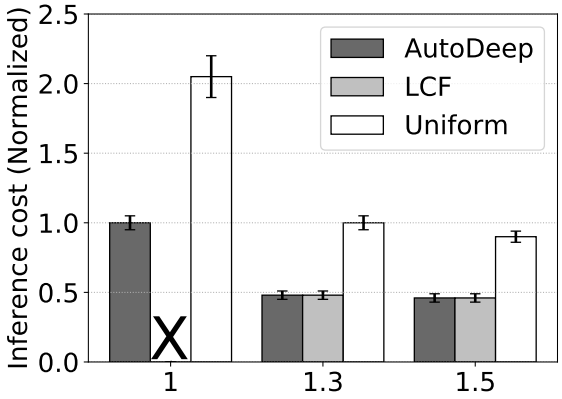

Fig. 8. Inference cost of Inception-V3 under varying QoS constraint.

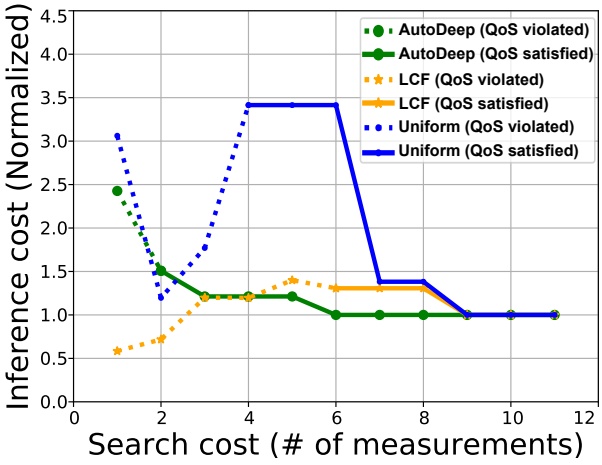

Fig. 7. Inference cost of RNNLM with varying number of measurements.

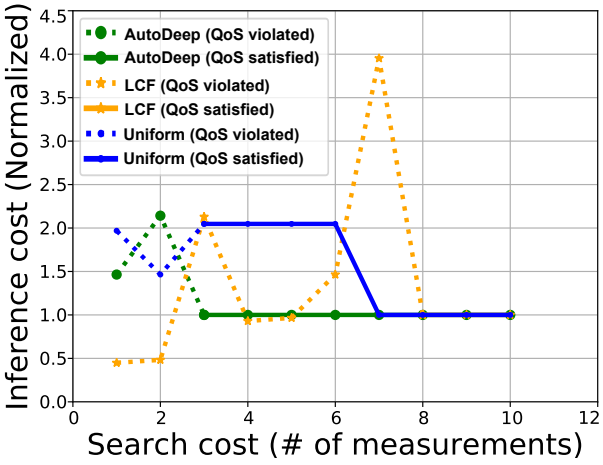

Fig. 9. Inference cost of Inception-V3 with varying number of measurements.

by Mirhoseini et al. [15]. Google-RL does not find much better device placement plan than using a single GPU. For RNNLM, Google-RL places all operations onto a single GPU. For Inception-V3, although Google-RL finds the placement on four GPUs, however, it has little improvement on the inference time than using one GPU.

As AutoDeep's device placement outperforms all the baselines, we use the DRL-based algorithm to find the device placement in the rest of the paper.

Performance of Cloud Configuration Searching. To evaluate the searching efficiency of AutoDeep, we set the search time limit to 6 trials and compare the inference cost of the cloud configuration found by the three algorithms. Fig. 6 and Fig. 8 depict the (normalized) inference cost of the two DNN models while increasing the QoS constraint from $1\times$ to $1.5\times$ of the original QoS constraint. AutoDeep achieves the best performance in both models. For RNNLM, AutoDeep reduces the inference cost by 23% compared to LCF. For Inception-V3, LCF does not find any feasible cloud configuration, and AutoDeep reduces 51% inference cost compared to Uniform. With more relaxed QoS constraints, all three 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. Fig. 7 and Fig. 9 demonstrate the change of inference cost (normalized to the best configuration) of RNNLM and Inception-V3, respectively. We set the QoS constraint to $1.5\times$ and $2\times$ of the inference time of the best configuration for RNNLM and Inception-V3, respectively. The dash lines show the inference cost before the algorithms find the cloud configuration satisfying the QoS constraint.

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 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 very high inference cost. Similarly, for Inception-V3, both AutoDeep and LCF 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 the worst as it finds the feasible (and the optimal) configuration at its 8-th trial.

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 [30]. Early work such as [31] develops a platform called CloudAdvisor to explore various cloud configurations which are recommended based on user preferences. CherryPick is a system designed in [14] 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 [26], [32], and finding the best cloud configuration for big-data analytics [14], [33]. 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 [15], [16], [34]–[36]. Existing works such as Picchini et al. [34] use an Approximate Bayesian Computation MCMC algorithm to accelerate Bayesian inference. Recent works such as Gao et al. [35] use cellular batching to accelerate RNN inference, but their approach can only be applied on cellular-like networks like RNN. As for the acceleration of CNN, Abdelouahab et al. [36] survey several methods to accelerate CNN inference on FPGAs, such as batch parallelism, inter-layer parallelism and so on. Mirhoseini et al. [15] and Gao et al. [16] propose to use the DRL to optimize the device placement for DNN training. Our work applies this technique to accelerate DNN inference and combines it with BO to compute the cost-efficient cloud configuration.

Conventional Graph Partition. Graph partition has been intensively studied in various domains, such as sensor networks [37]. Existing works [22], [38]–[41] start from an initial partition and use several refinement methods to explore similar partitions to improve after iterations. Other works such as [23], [42] 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, these approaches do not work well because it is hard to construct cost models for the graphs under all kinds of cloud configurations.

VII. Conclusion and Future Work

A. Conclusion

In this paper, we study the problem of automating the cloud deployment for online real-time DNN inference. We propose a novel algorithm AutoDeep that can adaptively choose the cost-efficient cloud configuration and the device placement for the DNN inference jobs. We implement AutoDeep with TensorFlow and conduct extensive experiments on Microsoft Azure. The experiments with two popular DNN inference models show that AutoDeep can significantly improve the inference speed (with better device placement), the search speed, and reduce the cost of inference compared with the non-trivial baselines, including Google's RL based method for device placement and Lowest Cost First for cloud configuration.

In this work, we only optimize the problem for online real-time DNN inference. Training a DNN takes long time that usually lasts for hours or even days. We believe it is a promising future direction to jointly optimize the cloud configuration and the device placement for DNN training jobs, where the trade-off between search time and training time is very critical.

B. Future Work

As this work has shown, the proposed AutoDeep, which combines BO and DRL, focuses on optimizing the cloud deployment of online DNN services, while BO and DRL are heavy-weight learning methods that require a large number of trials to generate accurate predictions, which may lead to a high searching cost.

To improve the search efficiency, the first promising direction is to improve their learning efficiency, e.g., developing a general network architecture so that re-training is not needed for new DNN inference models [43]. The other interesting direction is to optimize the system efficiency. We observe that each time the device placement is changed in the DRL sampling, the initialization of a DNN job (including obtaining the hardware information, building the computation graph, and CUDA initialization) takes a long time. Over 90% of searching time is wasted to initialize the computation graph. Allowing placing operations in a fine-grained manner (i.e., without restarting a job) could speedup the searching significantly.

Acknowledgment

This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61632013, 61632020 and 61772489, and the Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Proc. of NIPS*, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of IEEE CVPR*, 2016, pp. 770–778.

[3] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[4] A. Graves and N. Jaitly, "Towards End-to-end Speech Recognition with Recurrent Neural Networks," in *Proc. of ICML*, 2014, pp. 1764–1772.

[5] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep Speech: Scaling up End-to-end Speech Recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[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. of IEEE ICASSP*, 2016, pp. 4960–4964.

[7] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to Sequence Learning with Neural Networks," in *Proc. of NIPS*, 2014, pp. 3104–3112.

[8] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations Using RNN Encoder-decoder for Statistical Machine Translation," *arXiv preprint arXiv:1406.1078*, 2014.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *arXiv preprint arXiv:1409.0473*, 2014.

[10] Y. Wu, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[11] D. Wang, W. Cao, J. Li, and J. Ye, "DeepSD: Supply-demand Prediction for Online Car-hailing Services Using Deep Neural Networks," in *Proc. of IEEE ICDE*, 2017, pp. 243–254.

[12] Q. Ye, Z. Zhang, and R. Law, "Sentiment Classification of Online Reviews to Travel Destinations by Supervised Machine Learning Approaches," *Expert Systems with Applications*, vol. 36, no. 3, pp. 6527–6535, 2009.

[13] A. Gujarati, S. Elnikety, Y. He, K. McKinley, and B. Brandenburg, "Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services With Resource Efficiency," in *Proc. of ACM Middleware*, 2017, pp. 109–120.

[14] 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. of USENIX NSDI*, vol. 2, 2017, pp. 4–2.

[15] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," *arXiv preprint arXiv:1706.04972*, 2017.

[16] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing Device Placement for Training Deep Neural Networks," in *Proc. of ICML*, 2018, pp. 1662–1670.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A System for Large-scale Machine Learning," in *Proc. of USENIX OSDI*, vol. 16, 2016, pp. 265–283.

[18] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proc. of IEEE CVPR*, 2016, pp. 2818–2826.

[20] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective Cluster Scheduling for Deep Learning," in *Proc. of USENIX OSDI*, 2018, pp. 595–610.

[21] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks," in *Proc. of ACM SOSP*, 2017, pp. 184–200.

[22] F. Pellegrini, "Distillating Knowledge about Scotch," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[23] G. Karypis and V. Kumar, "METIS–Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," 1995.

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

[25] M. Pelikan, D. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian Optimization Algorithm," in *Proc. of GECCO*, 1999, pp. 525–532.

[26] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Proc. of NIPS*, 2012, pp. 2951–2959.

[27] D. Jones, M. Schonlau, and W. Welch, "Efficient Global Optimization of Expensive Black-box Functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[28] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[29] R. Williams, "Simple Statistical Gradient-following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[30] Z. Li, Y. Zhang, and Y. Liu, "Towards A Full-Stack DevOps Environment (Platform-As-A-Service) for Cloud-Hosted Applications," *Tsinghua Science and Technology*, vol. 22, no. 01, pp. 1–9, 2017.

[31] 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. of IEEE SERVICES*, 2013, pp. 456–463.

[32] J. Bergstra, D. Yamins, and D. Cox, "Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms," in *Proc. of Citeseer SciPy*, 2013, pp. 13–20.

[33] C.-J. Hsu, V. Nair, V. Freeh, and T. Menzies, "Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM," *arXiv preprint arXiv:1712.10081*, 2017.

[34] U. Picchini and J. Forman, "Accelerating Inference for Diffusions Observed with Measurement Error and Large Sample Sizes Using Approximate Bayesian Computation," *Journal of Statistical Computation and Simulation*, vol. 86, no. 1, pp. 195–213, 2016.

[35] P. Gao, L. Yu, Y. Wu, and J. Li, "Low Latency RNN Inference with Cellular Batching," in *Proc. of ACM EuroSys*. ACM, 2018, p. 31.

[36] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN Inference on FPGAs: A Survey," *arXiv preprint arXiv:1806.01683*, 2018.

[37] 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 Transactions on Sensor Networks (TOSN)*, vol. 15, no. 3, p. 30, 2019.

[38] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[39] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[40] C. Fiduccia and R. Mattheyses, "A Linear-time Heuristic for Improving Network Partitions," pp. 175–181, 1982.

[41] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, vol. 37, no. 6, pp. 865–892, 1989.

[42] L. Hagen and A. Kahng, "New Spectral Methods for Ratio Cut Partitioning and Clustering," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 11, no. 9, pp. 1074–1085, 1992.

[43] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: Efficient Progressive Device Placement Optimization," in *Proc. of NIPS Machine Learning for Systems Workshop*, 2018.