Serving Unseen Deep Learning Models with Near-Optimal Configurations: a Fast Adaptive Search Approach

Yuewen Wu*
Heng Wu*†
Institute of Software, Chinese Academy of Sciences
Beijing, China
wuyuewen@otcaix.iscas.ac.cn
wuheng@iscas.ac.cn

Diaohan Luo
Yuanjia Xu
Yi Hu
University of Chinese Academy of Sciences
Beijing, China
luodiaohan21@mails.ucas.ac.cn
xuyuanjia2017@otcaix.iscas.ac.cn
huyi19@otcaix.iscas.ac.cn

Wenbo Zhang‡§
Hua Zhong
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangwenbo@iscas.ac.cn
zhonghua@iscas.ac.cn

ABSTRACT

Public clouds provide a bewildering choice of configurations for Deep Learning (DL) models, and the choice of configuration will significantly impact the performance and budget. However, it is an obvious challenge to recommend a near-optimal configuration for a particular DL model from a wide range of candidates. The huge search overhead of finding such a configuration is the notorious cold start problem in state-of-the-art efforts, and this problem becomes more severe when they are faced with unseen DL models.

In this paper, we present Falcon, a novel configuration recommender system that can quickly adapt to unseen DL models. Through a large-scale evaluation, we find that there are some Key Operators (KOPs) that can be used to estimate the performance of DL models, and their resource sensitivity can be represented by four typical Key Operator Resource Curves (KOP-RCs). This work can effectively alleviate the cold start problem, because an unseen DL model can be characterized by its KOPs and corresponding KOP-RCs, and this characterization can be constructed as a tree structure in which near-optimal configurations can be searched quickly through a combination of Monte Carlo Tree Search and Bayesian optimization (MCTS-BO). Experiments show that Falcon can effectively reduce the search overhead for unseen DL models by up to 80% compared to state-of-the-art efforts.

CCS CONCEPTS

• **Social and professional topics** → **Pricing and resource allocation**.

KEYWORDS

cloud computing, deep learning, configuration recommender

ACM Reference Format:
Yuewen Wu, Heng Wu, Diaohan Luo, Yuanjia Xu, Yi Hu, Wenbo Zhang, and Hua Zhong. 2022. Serving Unseen Deep Learning Models with Near-Optimal Configurations: a Fast Adaptive Search Approach. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563485

1 INTRODUCTION

With the rapid advancement of Deep Learning (DL) technologies, more and more DL models are serving on public clouds to provide DL inference services (e.g., computer vision [39], natural language processing [50], speech recognition [41]). Deploying such services must deal with complex configurations, including *runtime* configurations (e.g., batch size) and *resource* configurations (e.g., GPU type, GPU memory), and the configuration of DL models may involve over 1,000 composition candidates in today's public clouds, such as Amazon EC2 [3]. Therefore, it is crucial to recommend near-optimal

*Both authors contributed equally to this research.
†Also affiliated with Chongqing School, University of Chinese Academy of Sciences.
‡Corresponding author.
§Also affiliated with (1) University of Chinese Academy of Sciences, Nanjing, and (2) Nanjing Institution of Software Technology.

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563485

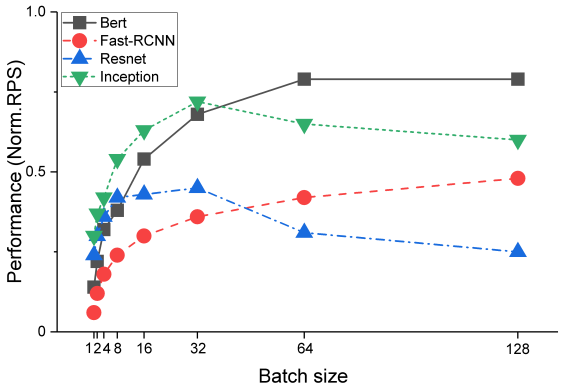
configurations in such a scenario. As shown in Figure 1, an optimal configuration can result in up to 8x performance improvement and over 60% budget reduction. In this context, unearthing near-optimal configurations for DL inference services from a wide range of candidates through *trials* [1] presents both an opportunity and a great challenge.

To address this challenge, existing configuration recommender (CR) systems, such as Morphling [43] and HeterBO [48], make important contributions by reusing historical data from previous DL models to improve the configuration search of other models, thus maximizing the *performance per budget*. However, they both require considerable search overhead (*trials*) to find near-optimal configurations. This is the notorious **cold start problem** of CR systems, which is further exacerbated when they are faced with **unseen DL models** [2]. As shown in Figure 2, Morphling takes 10 trials and 2.5 hours [3] to find a near-optimal configuration for a "seen" model [4], but requires 30 trials and 7.5 hours for an unseen one.

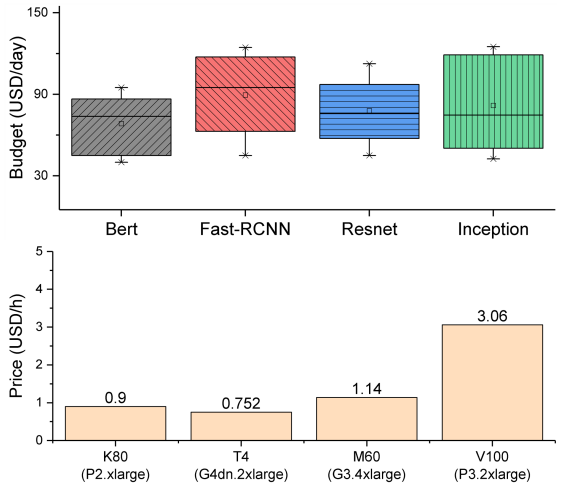
According to our observation, Morphling can only quickly adapt to "seen" DL models with high model-level similarity, but performs inefficiently for other DL models (see Section 2.1). We further analyzed those DL models with considerable similarity from the view of operators [7] and concluded that operators are the root cause of model-level similarity and they are better suited to describe the performance of DL models (see Section 2.3).

Guided by these findings, this paper presents Falcon, a novel CR system that can quickly adapt to unseen DL models. We believe our work makes the following advancements:

- We managed to alleviate the cold start problem from a new perspective at the operator level instead of the model level. There are two key insights to support this shift: (a) how to find the Key Operators (KOPs) that have a dominant impact on the performance of DL models, and (b) how to learn the resource sensitivity curves of KOPs, or the Key Operator Resource Curves (KOP-RCs), to navigate the search for different models in a large search space. To this end, we conducted a large-scale evaluation of 30 typical DL models on Amazon EC2 to learn KOPs and KOP-RCs.
- We design a novel fast adaptive search approach in Falcon, which combines Monte Carlos Tree Search and Bayesian Optimization (MCTS-BO), effectively employing the KOPs and KOP-RCs of DL models, thus we can quickly locate near-optimal configurations for unseen DL models.
- Experiments show that Falcon can effectively find near-optimal configurations, with up to 80% reduction in search overhead for unseen DL models compared to state-of-the-art efforts. The search trials can be significantly reduced from dozens to an average of six.



(a) Performance with varying batch sizes.



(b) Budget with varying GPU types.

Figure 1: Impact of serving DL models with different configurations: (a) with varying batch sizes, the performance is measured by *requests per second*, or RPS, and then normalized by the highest RPS to obtain norm.RPS, and (b) with varying GPU types.

2 MOTIVATION

In this section, we first discuss the limitation of existing CR systems. Next, we explain that serving unseen DL models is

[1]The trial is a stress test of the target DL inference model in a certain configuration to measure its performance.

[2]In this paper, new DL models and model variants developed by users are two main sources of unseen DL models, such as ResNet variants ResNet-101 and ResNet-152.

[3]Each trial takes around 15 minutes, including launching containers, stress testing, and results collection.

[4]The "seen" model in this paper implies that it has similar resource sensitivity and configuration preferences to historical DL models.

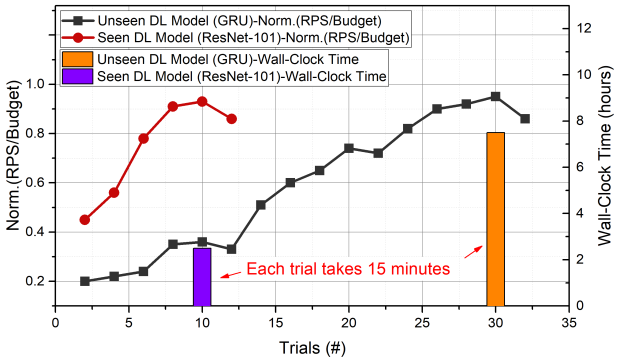


Figure 2: Search overhead for a "seen" DL model (ResNet-101) and an unseen DL model (GRU) in Morphling. The X-axis shows the number of trials. The left Y-axis shows the normalized *performance per budget*, or norm.(RPS/Budget). The right Y-axis shows the wall-clock time of the search.

a common requirement. After that, we show two important observations of DL models and operators. Finally, we present our expected improvements and discuss the challenges of achieving them.

2.1 Limitation of Existing CR Systems

Existing CR systems require considerable search overhead to find a near-optimal configuration, especially for unseen DL models, mainly because they are based on the model-level similarity and work as follows: (a) In the offline phase, existing CR systems [43, 45, 50] run *trials* with different configurations for diverse DL models to collect data, and then use statistical or machine learning techniques to learn model-level resource sensitivity (e.g., model-level resource sensitivity curves in Morphling). (b) In the online phase, when the target DL model is similar to existing DL models, the CR system can quickly adapt it by reusing the optimal configurations in existing data and searching for those nearby configurations based on probability theory (e.g., Bayesian optimization). Otherwise, in the worst case, the CR system must repeat the offline trial process for the target model to improve the search results.

Based on the above, we tested Morphling, a representative CR system. Figure 2 shows the *wall-clock time* required for Morphling to search for a near-optimal configuration, and the results show that it takes 7.5 hours for an unseen DL model (GRU) that is not similar to models learned offline. As a comparison, it takes only 2.5 hours for a "seen" DL model (ResNet-101). This implies Morphling has a severe **cold start problem** for unseen DL models, and this problem is common because our experiments (Figure 12) show that for Morphling, 40% of our tested DL models take 7.5 hours to get a near-optimal configuration.

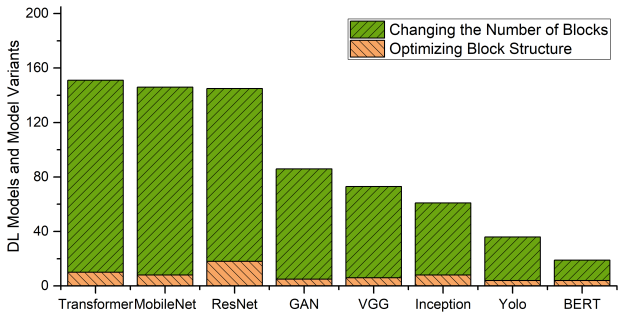


Figure 3: Statistics of DL models and model variants on TensorFlow hub.

2.2 Serving Unseen DL Models is a Common Requirement

Today, unseen models are usually derived from user developed models and model variants. The main purposes of developing model variants include: improving model accuracy [35, 36], improving performance [6, 30], reducing resource consumption [12, 13], etc.

We have analyzed 1,200+ DL models and model variants on TensorFlow hub [37], and those model variants can be divided into two categories: (a) Changing the number of blocks [5] [12]. These model variants weigh the accuracy and resource consumption of the model to suit different application scenarios. For example, *ResNet-152* is one of the variants of the model *ResNet*, which has more blocks, deeper networks, and also requires more resources. (b) Optimizing block structure [35]. The optimized block structure usually has better accuracy and lower resource consumption. For example, *Inception V4* combines the residual network structure of *ResNet* into blocks, which was not present in previous versions of *Inception* (v3/v2/v1).

Figure 3 shows part of the statistics of model variants on TensorFlow hub, which indicates a great variety of models. Based on the above analysis, the CR systems have to frequently serve unseen DL models and model variants developed by users with different resource sensitivity and configuration requirements.

2.3 Observations

Falcon presents a new perspective to alleviate the cold start problem by taking full advantage of operators. Unlike DL models which are highly variable, DL operators are relatively fixed [28]. In this context, we present a comprehensive analysis of typical DL models, various operators and diverse configurations, and come up with the following two important observations.

[5] The block is a few stacked layers in a DL model, such as the *Inception* block and *ResNet* block.

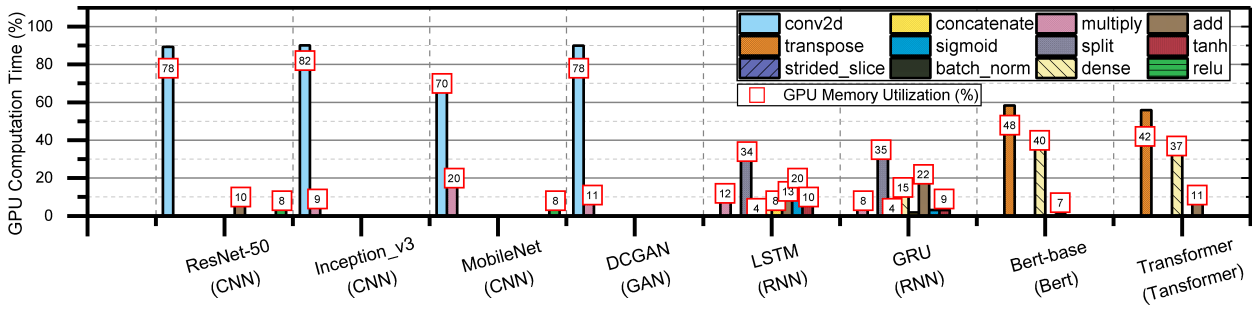
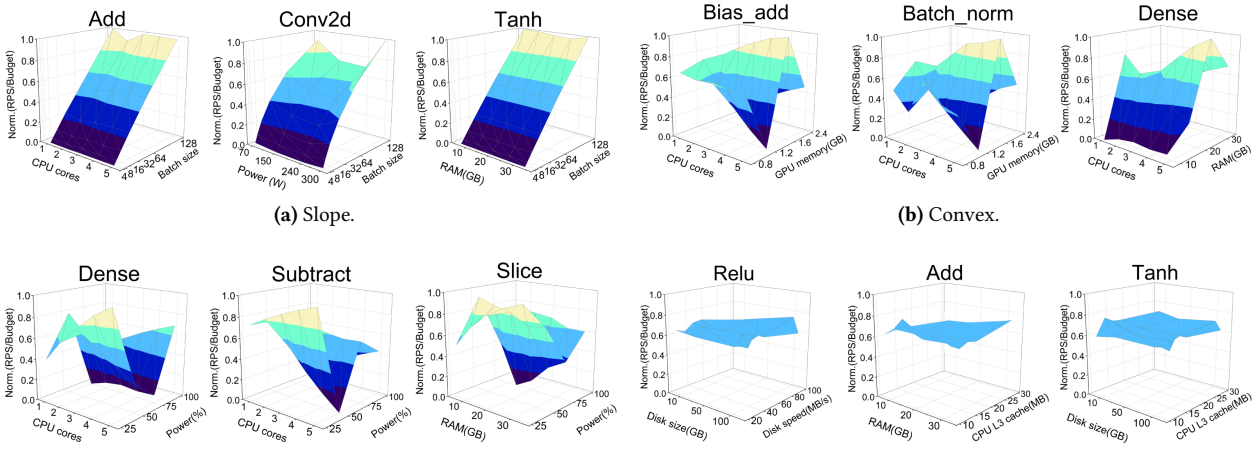


Figure 4: KOPs in different DL models. The X-axis shows DL models, and the Y-axis shows the GPU computation time of operators. The red boxes show the GPU memory utilization of operators.



(a) Slope. **(b)** Convex.

(c) Concave. **(d)** Plane.

Figure 5: KOP-RCs in the configuration-performance relationship. The X and Y axis show two different configurations. The Z-axis shows the *performance per budget*, which is evaluated by norm.(RPS/Budget).

Key Operators. We conduct comprehensive tests on all types of operators in commonly used DL models to investigate the performance relationships between DL models and operators. An important observation is that the Key Operators (KOPs) in the DL model take up most of the GPU computation time and the GPU memory [6].

Figure 4 shows the GPU computation time (the Y-axis) and GPU memory utilization (red boxes) of operators in different DL models. The results show that the *conv2d* operator is the only KOP of most CNN and GAN models, which occupies more than 80% GPU computation time and over 70% GPU memory utilization in average. Meanwhile, RNN, Bert and Transformer have multiple KOPs (Section 3.1 evaluates KOPs by thresholds).

Based on the above observations, we can conclude that KOPs can be used to estimate the performance of DL models. This means that if we can find near-optimal configurations for KOPs, we can at least narrow the search space based on these configurations. This provides us a neat shortcut to alleviating the cold start problem, i.e., identifying KOPs for an unseen DL model after its first run and navigating the search direction based on pre-prepared resource sensitivity of KOPs.

Key Operator Resource Curves. To further investigate the resource sensitivity of operators and how their resource sensitivity can help navigate the space of overall combined configuration search, we evaluate KOPs with different configurations and analyze their performance.

Figure 5 presents some of the results showing that the configuration-performance relationships of KOPs can be categorized into four types of resource sensitivity curves, or Key Operator Resource Curves (KOP-RCs): *slope*, *convex*, *concave* and *plane*. For instance, all plots in Figure 5(a) conform to the *slope* curve because the normalized *performance per budget*

[6] In terms of GPU resource usage, one can consider two high-level resources: (a) GPU computation time and (b) GPU memory.

(the Z-axis) is mainly determined by one kind of resource configuration, which means that the search needs to test that resource first, while needing to avoid searching in bad regions with low norm.(RPS/Budget). In addition, there are some resources that match the *convex* and *concave* curves shown in Figure 5(b) and Figure 5(c), and we need to configure them simultaneously to locate good search regions. In contrast, the *plane* curve (Figure 5(d)) represents some configurations that have negligible impact on performance and should be filtered out before searching.

Based on the above observations, we can conclude that KOP-RCs can not only greatly reduce the search space, but also help us to divide the search space into good and bad search regions. This can help us to efficiently navigate the search for unseen DL models in a large search space.

2.4 Improvements and Challenges

In a complex environment with 1,000+ configuration candidates and new models being developed frequently, we believe that a good CR system needs to have the ability to quickly adapt to unseen DL models. Unfortunately, existing CR systems suffer from the cold start problem due to the limitation of model-level similarity. To address this problem, we analyze a wide range of DL models, operators and configurations, and come up with two important observations: KOPs and KOP-RCs, which give us a hopeful solution to alleviate the cold start problem. However, we have to face at least two main challenges:

- Although DL models have been abstracted to KOPs and KOP-RCs, the similarities between them remain to be explored. Especially for those DL models that consist of multiple KOPs.
- Effective search based on KOPs and KOP-RCs is still very difficult because we need not only to divide the search space into good and bad regions, but also to avoid continuous search in bad regions.

3 DESIGN

In this section, we first show our large-scale evaluation on Amazon EC2 to learn KOPs and KOP-RCs. Next, we construct trees to represent KOPs and divide the search space into good and bad search regions according to KOP-RCs. After that, we show how we quickly adapt to unseen DL models, and discuss how MCTS-BO alleviates the cold start problem.

3.1 Learning KOPs and KOP-RCs from a Large-Scale Evaluation

Observations on KOPs and KOP-RCs (see Section 2.3) provide inspiration for alleviating the cold start problem for unseen DL models. To analyze KOPs for commonly used DL models and learn the KOP-RCs across a wide range of configuration candidates, we carry out a large-scale evaluation on Amazon EC2. Specifically, we evaluate 30 typical DL models from CNN, RNN, Bert, Transformer, and GAN from TensorFlow hub, including the following categories:

- **Computer vision.** It includes VGG, ResNet, YOLO, DenseNet, etc.
- **Natural language process.** It includes LSTM, GRU, Bert, etc.
- **Generative adversarial network.** It includes DCGAN, WG-AN, SGAN, etc.
- **Recommend system.** It includes NCF, DCN, DRN, etc.

We prepare the runtime environment for DL models by renting GPUs instances (e.g., P2.xlarge) on Amazon EC2. To simulate a tunable configuration environment in these GPU instances, we deploy containers [25] as the resource control panel and tune the following configuration knobs:

- **Runtime configuration knobs.** We mainly consider the *batch size* as the runtime configuration knob, because it can profoundly impact the performance of DL models [28].
- **Resource configuration knobs.** These configuration knobs can be tuned when we deploy DL models on public clouds. They include GPU type, GPU memory, CPU cores, CPU L3 cache, RAM, GPU power [7], disk speed, disk size, network speed, etc.

To profile the KOPs and KOP-RCs of a given DL model, we first analyze its operators using TVM [7]. Then, we run the DL model in a *sandbox* [8] environment and evaluate the GPU computation time and GPU memory of each operator through program instrumentation, and use these two metrics to identify KOPs. After that, we tune the configuration knobs to profile the KOP-RCs (as shown in Figure 5) and collect norm.(RPS/Budget) data. Concretely, we make the following efforts to better profile KOPs and KOP-RCs:

- **Identifying KOPs for a given DL model.** We use TVM to analyze the operator composition for a given DL model and set thresholds for the GPU computation time and GPU memory utilization of operators. In this way, we can identify the KOPs for a given model. To verify the effectiveness of the threshold settings, we design an experiment in Section 5.3.
- **Pruning redundant configurations in KOP-RCs.** We use Principal Component Analysis (PCA) [32] to prune configurations that have negligible impact on norm.(RPS/Budget), such as configurations in the *plane*

[7] In this paper, we tune the percentage of power supplied to the GPU. For example, Tesla V100 is rated at 250 watts, and the 50% power supply is 125 watts.

[8] The sandbox environment satisfies resource requirements of the tested DL model.

Table 1: Key configurations evaluated by PCA.

Configuration	Description
CPU cores	For DL inference, CPU is responsible for data processing and I/O. Recent studies [28, 43] show that adding more CPU cores to a serving DL model enables a higher degree of parallelism for data processing and I/O. For most inference services, the RPS improvement diminishes with the increase of CPU cores.
GPU memory	To run a DL inference service, there is a minimum requirement of GPU memory to fully load the serving model [50]. Further increasing GPU memory allows it to serve larger request batches with higher RPS. However, when we consider both RPS and budget, it is difficult to strike a balance between them.
GPU type	There are various types of GPU dedicated to DL inference services, and their ideal use cases are different [4]. For instance, ideal use cases for NVIDIA T4 include machine learning and graphics-intensive DL models with better RPS.
Batch size	For DL inference, system throughput (RPS) can be increased by setting an appropriate batch size. In particular, configuring a large batch size is not always beneficial [28, 43], as it may decrease statistical efficiency.
GPU power	Recent studies [16, 23] set power caps for DL services to keep peak power consumption below a given power budget, and they also illustrated that GPU power can impact the RPS of DL training and inference services.

resource curves in Figure 5(d). Specifically, we use the principal component feature matrix to do dimensionality reduction on the sample matrix, and then use the transpose of the feature matrix to reverse back to the original features. By comparing the reversed feature matrix with the sample matrix, we can evaluate the impact of the configurations on the performance. The configurations with larger impact have a high correlation with the performance, while the configurations with smaller impact have almost no correlation with the performance. After the above steps, we can prune redundant configurations in KOP-RCs. Table 1 summarizes key configurations that can significantly impact norm.(RPS/Budget) in our dataset, and we evaluate the parameter setting of PCA in Section 5.3.

Finally, we prepare the offline dataset for KOPs and KOP-RCs. This dataset includes the performance and resource sensitivity information for KOPs and KOP-RCs to navigate the search for other DL models in a large search space.

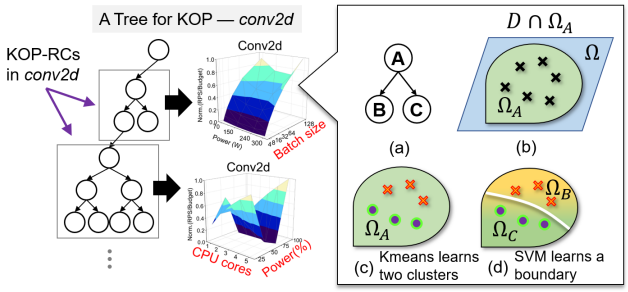


Figure 6: Using a tree to represent DL model *ResNet-101* and its KOPs and KOP-RCs. *ResNet-101* has one KOP—*conv2d*. KOP *conv2d* has multiple KOP-RCs, and this plot shows only two of them for space reasons. Each KOP-RC partitions tree regions based on its resource sensitivity.

3.2 Constructing Trees to Represent KOPs and KOP-RCs

Before the online search phase, we need to construct a data structure to represent the resource sensitivity in the offline dataset. This process needs to achieve two goals: (a) to represent the complex relationship between DL models, KOPs, and KOP-RCs, and (b) to partition a large search space into good and bad regions, enabling the online search phase to quickly guide the search in good regions for unseen models.

We choose the tree model because it has been widely used to solve the black-box search problem [34, 46, 47]. It can divide the search space into small search regions, which has the potential to achieve fast and accurate search in a very large search space.

First, we construct trees for a given DL model. Figure 6 shows how trees can be used to represent model *ResNet-101* and its KOPs and KOP-RCs. The details are as follows:

- **Construct a tree for a given KOP.** As shown in Figure 6, *ResNet-101* has a KOP—*conv2d*, so we construct a tree for *conv2d*. To control the height of the tree to reduce the overhead of online search, the tree contains only *key* KOP-RCs that include only the key configurations in Table 1. For this purpose, we calculate the average norm.(RPS/Budget) of KOP-RCs and select configuration knobs by descending order. The plot shows top two KOP-RCs in KOP *conv2d*.
- **Split tree regions in a tree according to KOP-RCs.** Figure 6 also shows how we split a large search space into small tree regions. On the left side of this plot, we show that the region of a *slope* KOP-RC will split into two sub-regions, and on the right side we show the details of the split. (a) A tree region for the *slope* KOP-RC, which contains a root node A and two subsequent nodes B and C. (b) Denote D as the offline dataset,

then $D \cap \Omega_A$ represents the data samples falling into node A. (c) We apply Kmeans to find a good and a bad clusters in $D \cap \Omega_A$. (d) We employ Support Vector Machines (SVM) to learn a decision boundary, thus the data of Ω_A split into sub-regions Ω_B and Ω_C, and data in Ω_B have better norm.(RPS/Budget). Since *convex* and *concave* KOP-RCs have to satisfy two decision boundaries simultaneously, they will split a two-level tree region as shown in the left side of Figure 6.

- **Cover all KOPs.** We repeat the above steps until all KOPs in *ResNet-101* have been learned.

Next, we use trees to represent the entire search space. Specifically, suppose Ω denotes the entire search space of the offline dataset $D = \{(a_i, v_i)\}$, where a_i corresponds to the ith data, and v_i corresponds to the norm.(RPS/Budget) of a_i. It can be divided into $n \in K$ trees, where K equals the number of KOPs in dataset D. Each tree Ω_n has M tree nodes, and Ω_{nm} denotes a tree region Ω_n of tree node $m \in M$. Finally, we can present each tree as

$$\Omega_n = \sum_{m \in M} \Omega_{nm}. \tag{1}$$

Each Ω_{nm} is corresponding to a KOP-RC, its data will be grouped into a good and a bad clusters by Kmeans, and the decision boundary of these clusters will be measured by SVM. In this process, we use average performance $V(\Omega_{nm})$ to evaluate how "good" or "bad" a tree region Ω_{nm} is. Specifically, the average performance of each region Ω_{nm} is estimated by

$$V(\Omega_{nm}) = \frac{1}{\mathcal{N}} \sum_{v_i \in D \cap \Omega_{nm}} v_i \tag{2}$$

where $\mathcal{N}$ denotes the number of data, and $D \cap \Omega_{nm}$ denotes all data in this tree region.

Finally, when a target DL model comes, why do we still need online search? There are two main reasons:

- **Complex DL model variants.** In our study, unseen DL models are mainly derived from model variants. However, the resource sensitivity of model variants is difficult to predict accurately, as their structure and parameters may be completely different [13, 35]. In this case, reusing the best configuration of the offline learned models may lead to poor performance.
- **Conflicts in KOPs.** The resource sensitivity may also be obscured by conflicts in multiple KOPs. For instance, KOP *conv2d* requires a larger batch size of 128, while KOP *dense* achieves optimal norm.(RPS/Budget) when batch size is 64. Therefore, when different KOPs are in a same DL model, it is difficult to evaluate the impact of them via accurate estimation.

After the above steps, as long as KOPs of an unseen model have been learned before, we are able to represent it with trees. These trees can depict good search regions for the target DL model, and our online search process can help to quickly locate near-optimal configurations.

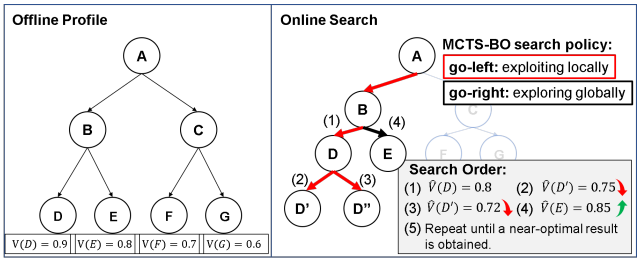


Figure 7: In the online phase, MCTS-BO controls the search strategy: *go-left* to exploit local tree regions, and *go-right* to explore other tree regions or other trees.

3.3 Fast Adaptive Searching via MCTS-BO

Problem statement. Existing CR systems suffer from the cold start problem, requiring dozens of *trials* to search for a near-optimal configuration for unseen DL models. Our goal is to minimize the number of *trials*.

How MCTS-BO works. To achieve the above goal, we implement an MCTS-BO algorithm (based on a basic algorithm [31]) that reuses trees from the offline phase to navigate the search for unseen DL models. Figure 7 provides an example to illustrate how MCTS-BO works: (1) It reuses optimal configuration in the offline phase. (2) However, this configuration performs a sub-optimal result on the target DL model, so MCTS-BO applies the *go-left* strategy to exploit local sub-regions. (3) It continues to exploit more sub-regions, but the results do not improve. (4) It uses the *go-right* strategy to explore global tree regions or other trees. (5) Repeat the above steps until a near-optimal result is obtained.

Specifically, MCTS-BO reuses the tree structure of Ω_n and updates the tree regions Ω_{nm} with new data from unseen DL models. Suppose $D' = \{a_j, v_j\}$ corresponds to the dataset which contains trial data from an unseen DL model, where a_j corresponds to the jth trial, and v_j corresponds to the performance of a_j. Then, we update existing tree regions Ω_{nm} by estimating

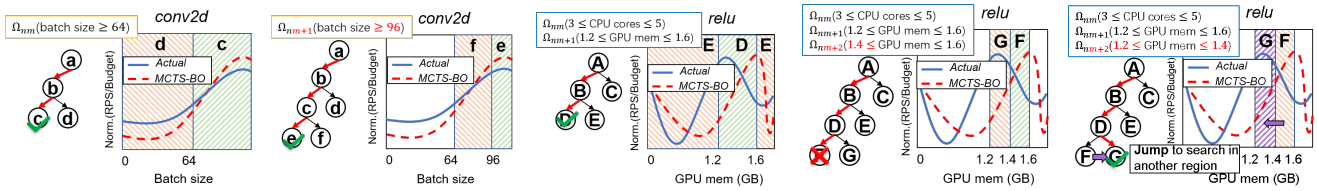
$$\hat{V}(\Omega_{nm}) = \frac{1}{\mathcal{N}} \sum_{v_j \in D' \cap \Omega_{nm}} v_j. \tag{3}$$

To evaluate whether the search in Ω_{nm} is efficient, we minimize the deviation of the trials

$$\underset{(a_j, v_j) \in D' \cap \Omega_{nm}}{\text{minimize}} \sum (\hat{V}(\Omega_{nm}) - v_j)^2 \tag{4}$$

where our goal is to minimize the gap between the predicted result $\hat{V}(\Omega_{nm})$ and the actual result v_j.

The above is the basic Monte Carlos Tree Search (MCTS) process. However, gaps in Equation 4 may cause us to search



(a) KOP *conv2d* the 1st trial. **(b)** KOP *conv2d* the 2nd trial. **(c)** KOP *relu* the 1st trial. **(d)** KOP *relu* the 2nd trial. **(e)** KOP *relu* the 3rd trial.

Figure 8: Searching for a near-optimal configuration for an unseen DL model via MCTS-BO. The model contains two KOPs *conv2d* and *relu* represented by two trees. The X-axis shows the configuration while the Y-axis shows the norm.(RPS/Budget). The blue solid lines represent the actual curve of the DL model, and the red dashed lines plot the predicted curve of MCTS-BO. The green and red regions represent Ω_{good} and Ω_{bad} search regions, respectively. The purple region represents a original bad search region turned into a good one because of deviation between the actual curve and the predicted curve.

in bad regions. In this case, we employ Bayesian Optimization (BO) while performing MCTS to strike a balance between exploitation (search in sub-regions) and exploration (search in another region or another tree). BO has been shown to assign more trials in good regions compared to other methods [42], such as Support Vector Machines (SVM) [34] and Linear Regression (LR) [5]. This helps us to quickly jump to another potentially good region when the search results become worse.

Figure 8 and Algorithm 1 show the search process for an unseen DL model via MCTS-BO. Specifically, it works as follows:

- **Initializing and analyzing unseen DL models.** As shown in Algorithm 1 lines 1~5, for an unseen DL model, we randomly pick up a configuration to run it, and identify its KOPs. If this model has more than one KOPs, we calculate the average GPU computation time and GPU memory utilization of these KOPs to decide their descending search order.
- **Searching for a near-optimal configuration.** As shown in Algorithm 1 lines 6~16 and Figure 8, we first search for the next trial (Figure 8(a) and Figure 8(c)). Next, because there are performance gaps between the actual curve (blue solid lines in Figure 8) and the MCTS-BO's predicted curve (red dashed lines in Figure 8), the algorithm needs to balance the exploration and exploitation when searching. In this case, MCTS-BO either generates new sub-regions with new data to further exploit a potentially good region (as shown in Figure 8(b) [9]), or jumps to search in another region or another tree to avoid continuing searching in a bad region (as shown in Figure 8(d) and Figure 8(e)). Finally, MCTS-BO ends up with a near-optimal configuration.

[9] After node e, there are subsequent nodes to cover all configuration knobs, so it can provide a runnable configuration, which we do not show fully for space reasons.

Algorithm 1 MCTS-BO

Require:

- The a_j denotes the jth *trial* of the target DL model.
- The v_j denotes actual performance of a_j.
- The v_{target} denotes the performance target.
- The Ω_n denotes the nth tree.
- The Ω_{nm} denotes the mth tree region in Ω_n.
- The $\hat{V}(\Omega_{nm})$ denotes predicted performance.
- The x denotes recommended configuration.

1: Initialize MCTS-BO with a random picked configuration and run a *trial* for the target model to collect data a_1.
2: Analyze a_1 to identify the KOPs of the target model, suppose it has K KOPs corresponding to K trees.
3: **if** $K \geq 2$ **then**
4: Calculate the average GPU computation time and GPU memory utilization of KOPs to get their descending search order $1, 2, \cdots, K$.
5: **end if**
6: **for** j trials in $1, 2, \cdots, J$ **do**
7: **for** n trees in $1, 2, \cdots, K$ **do**
8: Minimize $\sum(\hat{V}(\Omega_{nm}) - v_j)^2$ to search for x.
9: **if** $v_j \geq v_{j-1}$ **then**
10: **Exploitation:** generate new sub-regions in a potentially good region with new data a_j to achieve better performance.
11: **else**
12: **Exploration:** jump to another region or another tree to avoid continuing searching in a bad region.
13: **end if**
14: **Break when** $v_j \geq v_{target}$.
15: **end for**
16: **end for**

Ensure:

Recommended configuration x after j trials.

From Algorithm 1 we can find that in the worst case, Falcon has a time complexity of $O(NM)$, which is comparable to Morphling (its algorithm works in two stages with a time complexity of $O(NM)$). However, common DL models usually have only a few or even one KOPs (e.g., most CNN and GAN models have only one KOP). This means that in most cases, the algorithm only needs a few iterations on line 7, which is why Falcon is able to reduce the time complexity and alleviate the cold start problem.

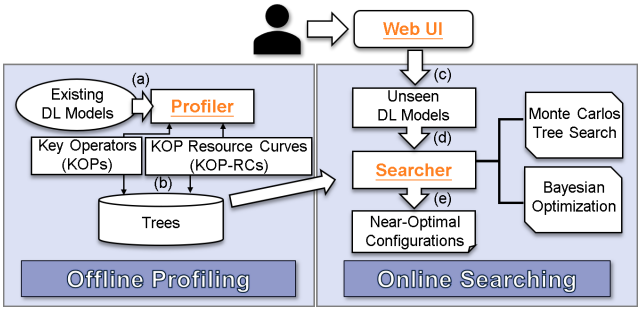


Figure 9: Falcon's architecture.

4 IMPLEMENTATION

In this section, we introduce Falcon's architecture and its main components [10]. We have implemented Falcon as a managed service in CentOS. The MCTS-BO algorithm is developed based on a basic algorithm called SPBOpt [31]. Our implementation consists of around 5k lines of Python.

Workflow. As shown in Figure 9, Falcon implements an offline-online CR system that works as follows: (a) Falcon uses TVM to load DL models developed by different DL frameworks (e.g., TensorFlow [1], PyTorch [27], PaddlePaddle [21]) and runs experiments in TVM in a container runtime environment. (b) Falcon constructs trees and stores them in Json files, making it easier to be queried by other components with Http requests. (c) Users submit DL models via Falcon's web UI, giving budget caps and performance requirements (e.g., peak RPS). (d) For a user and the DL model, Falcon iteratively returns the next recommended configuration. (e) The user iteratively starts trial (on public clouds) to evaluate the configuration, until the budget exhausts or the performance requirement meets, or Falcon announces that it finds a near-optimal configuration with maximized *performance per budget*. During this process, users can use recommended configurations to serve DL models on public clouds.

System components. The *Web UI* is developed by the VUE framework [40] and allows users to submit DL models and run trials to obtain a near-optimal configuration. The *profiler* runs experiments for each operator with varying configuration knobs and records the running time. The running time is then converted into performance metrics RPS (batch size/running time) and the performance per budget is measured by (RPS/Budget), which is then normalized by the min-max algorithm [32] to obtain norm.(RPS/Budget). The *searcher* communicates with other components by Http requests, and it works as querying trees, recommending configurations via MCTS-BO algorithm, running trials for users, and returning results to the *Web UI*.

[10]Falcon is now available at https://github.com/dos-lab/Falcon.

5 EVALUATION

In this section, we evaluate Falcon by the following experiment design:

- **Effectiveness.** We evaluate the effectiveness of Falcon, including alleviating the cold start problem, apple-to-apple comparisons, model-by-model comparisons, and searching in good regions.
- **Robustness.** We evaluate robustness of Falcon by applying different parameters to the methods used by Falcon, such as thresholds for identifying KOPs and the parameters for PCA.
- **Practical benefits for real-world applications.** To evaluate practical benefits of configuration recommendation, we use an enterprise-level DL benchmark [29] to simulate real-world applications.

Table 2: DL models in our experiments.

	No.	Name		No.	Name
Source set	1	ResNet-152	Target set	19	MobileNet V2
	2	DenseNet-121		20	VGG 16
	3	WGAN		21	ResNet-101
	4	DCGAN		22	ResNet152 V2
	5	SGAN		23	Inception V2
	6	MobileNet V3		24	Inception V4
	7	Inception-ResNet V2		25	DenseNet-201
	8	Inception V3		26	Bert-large
	9	VGG19		27	GRU
	10	Fast-RCNN		28	RoBERTa
	11	Bert-base		29	Transformer
	12	NCF		30	Tacotron2
	13	DCN			
	14	DRN			
	15	NasNet-large			
	16	LSTM			
	17	EfficientNet-widese-b4			
	18	YOLO V5			

5.1 Experiments Setup

DL models As shown in Table 2, we select 30 open source models of CNN, RNN, Bert, Transformer, and GAN from

TensorFlow hub, including typical categories such as computer vision, natural language process, recommend system. To simulate the environment where users submit unseen DL models and model variants, we divide these models into a *source* set and a *target* set, where the models in the *source* set are used for the offline phase and the models in the *target* set are used for the online phase.

Note that, as we investigated in Section 2.2, model variants ending with a number (e.g., ResNet-101) change the number of blocks, and those ending with a version (e.g., YOLO V5) optimize the block structure.

Configuration search space. To simulate the scenario of deploying DL models on the public clouds, we select instance types on Amazon EC2 covering GPU types such as M60, T4, K80, and V100. In addition, we use a container runtime environment to ensure that all configuration knobs can be tuned within a reasonable range. For instance, we set the upper bound of the batch size to 128 because the performance of DL inference in our selected GPU types no longer improves when batch size $\geq$ 128. The detailed configuration knobs are as follows:

- CPU cores: 1, 2, 3, 4, 5.
- GPU type: M60, T4, K80, V100.
- GPU memory (GB): 0.8, 1.2, 1.6, 2.4.
- Batch size: 4, 8, 16, 32, 64, 128.
- GPU power [11]: 50%, 75%, 100%.

With the combination of these configuration knobs, we obtain a search space with 1,440 configuration candidates, and all experiments are evaluated under this search space.

Baselines. We compare Falcon with the following state-of-the-art works.

- Morphling[43] is a CR system based on meta-learning and BO, it evaluates configuration-performance curves for existing DL models and adapts unseen DL models by iteratively updating meta-model with newly trial data using stochastic gradient descent (SGD), which is inefficient and time-consuming in a large search space. To evaluate Morphling, we train the meta-learning model using data from the *source* set, and validate it in the *target* set. We tune the parameters based on Morphling's recommendations. For instance, it suggests setting the weight knob δ to a small constant.
- Vesta[45] leverages a transfer learning-based CR system that transfer configuration-performance *knowledge* to cross-framework applications. It only works well in the model level. To evaluate Vesta, we use models from the *source* set to build its two-layer bipartite graph and validate it using models from the *target* set. Vesta also provides a comprehensive parameter suggestion, such as the K value for K-means.
- HeterBO[48] is a CR system that observes prior features of other DL training models and accelerates the training process of new models via BO. It is a model-level optimization work. To evaluate HeterBO, we train the Conventional BO model using data from the *source* set. For parameter setting, HeterBO supports a dynamic constraints setting and provides a heuristic rule to support various scenarios. We use data from the *source* set to simulate this process and use the *target* set for validation.
- Ernest[38] summarizes computation and communication patterns to build the predictive function for advanced analytics models. However, it requires a large number of samples to retrain the predictive function for unseen models. We run Ernest for each model in the *source* set in a *sandbox* environment to collect data and build the predictive function, and use models in the *target* set to dynamically tune its parameters for best practice.

Metrics. We use three metrics in our evaluations.

- Search accuracy: we evaluate the search accuracy by measuring the gaps between the recommended and the optimal configurations by $\frac{recommended}{optimal} \times 100\%$. Note that, we obtain the optimal configuration via exhaustive search.
- Search overhead: we use the number of *trials* to evaluate the search overhead, and we also record the *wall-clock time* for running the *trials*, which in our environment takes an average of 15 minutes per *trial*.
- Practical benefits: we set the objective of the CR system as to use the recommended configurations on clouds to maximize the performance per budget, i.e.,

$$\text{maximize Norm.(RPS/Budget)} \tag{5}$$

where the budget is calculated by *instance price* $\times$ *running time* in Amazon EC2, and we use min-max to normalize RPS/Budget for better evaluation.

5.2 Effectiveness of Falcon

Alleviating the cold start problem. Falcon alleviates the cold start problem by taking full advantage of operators, it learns KOPs and KOP-RCs to divide search space into good and bad search regions, and employs MCTS-BO to quickly adapt to unseen DL models with relatively lower search overhead (trials). Figure 10 shows the search accuracy of unseen DL models (from the *target* set in Table 2) with varying number of trials. In the six trials case, only Falcon can achieve over 90% search accuracy, while the baseline value is less than 60%. This implies that Falcon can recommend

[11] The four GPU types are rated at 150 watts for the K80, 70 watts for the T4, 240 watts for the M60, and 300 watts for the V100.

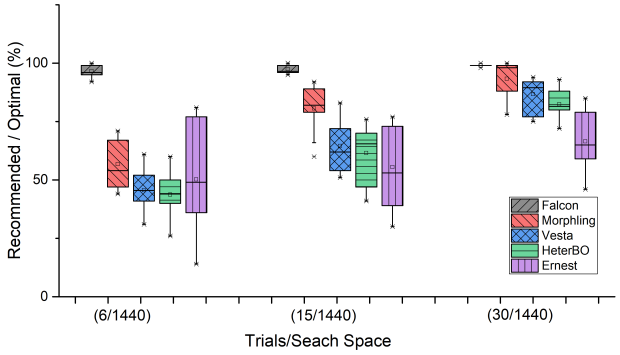
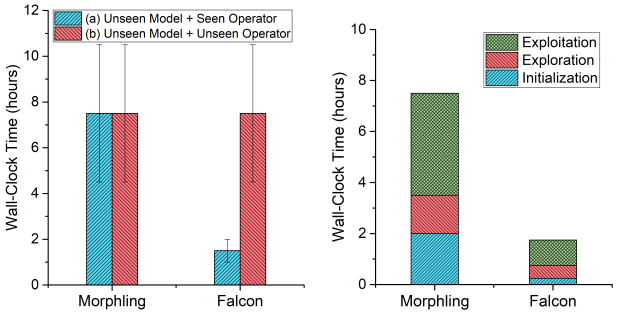


Figure 10: Comparison of different number of trials. The X-axis shows 6, 15, 30 trials, respectively. The Y-axis shows the search accuracy of the DL models in the *target* set.



(a) Wall-clock time in two cases. **(b)** Wall-clock time of the online search phase.

Figure 11: Apple-to-apple comparison with Morphling, and the tested model is *Bert-large*. (a) Comparing the wall-clock time of the search in two cases, and the bar shows the deviations. (b) Comparing end-to-end time cost of the online search phase.

near-optimal configurations after six *trials*, and the *wall-clock time* is 1.5 hours. As the number of trials increases, the search accuracy of baselines is more or less improved, and Morphling can achieve search accuracy that close to Falcon at 30 *trials*, and the *wall-clock time* is 7.5 hours. In this context, Falcon can find a near-optimal configuration with **80% less search overhead** for unseen DL models compared to state-of-the-art efforts.

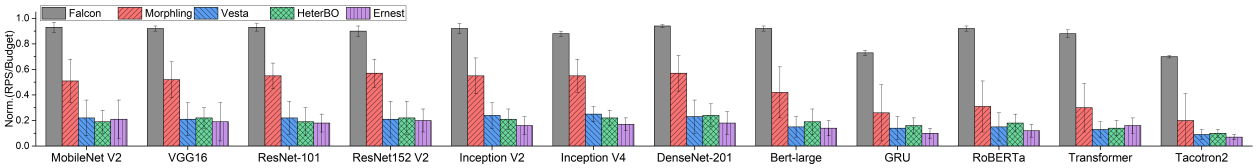
Apple-to-apple comparison with Morphling. To further investigate whether Falcon is still effective under different experiment settings. We conduct an apple-to-apple comparison with Morphling. As shown in Figure 11, we design two experiments, the first one compares two cases: (a) unseen model + seen operator, and (b) unseen model + unseen operator. The results are shown in Figure 11(a), where the Y-axis is the wall-clock time of the search. For each case, we run 10 times to take a conservative estimate of *P90* value. In case (a) the wall-clock time of Morphling is 7.5 hours, while the wall-clock time of Falcon is 1.5 hours. In case (b) both of them spend 7.5 hours on searching, because Falcon needs to profile unseen KOPs first (*Bert-large* has two KOPs—*dense* and *transpose*), just as Morphling needs to profile the unseen model.

The second experiment compares the end-to-end wall-clock time for the online search phase (initialization, exploration, exploitation) in the case of unseen model + seen operator. Figure 11(b) shows the comparison results. (a) Initialization: Morphling needs two hours (eight trials) to initialize the meta-model, while Falcon costs only 15 minutes (one trial) to analyze the KOPs. (b) Exploration: Morphling spends an average of 1.5 hours (six explorations) due to its poor adaptation to the resource curves of the unseen model, which requires frequent exploration to avoid sub-optimal results, while Falcon costs 15 to 30 minutes (one or two explorations) due to the KOP-RCs can precisely navigate near-optimal configurations. (c) Exploitation: Morphling's frequent exploration resulted in a long time exploitation (four hours and 16 trials in average), while Falcon only runs three or four exploitations in 45 minutes to one hour.

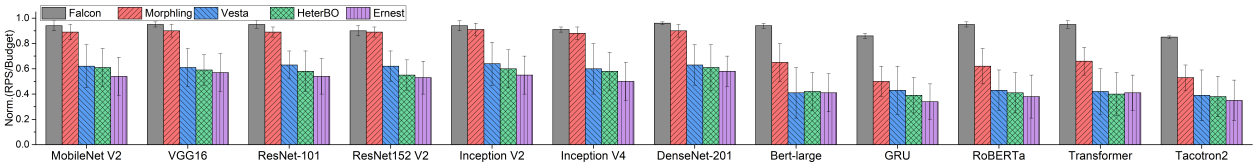
Model-by-model configuration optimization. The experiment in Figure 10 has compared the overall results of configuration optimization. In this experiment, we evaluate the model-by-model cases for DL models in the *target* set by evaluating the metrics in Equation 5.

As shown in Figure 12, in most of the cases, Falcon can find near-optimal configurations after 1.5 hours (six trials), except for GRU and Tacotron2 (see Figure 12(a)). After analyzing, we find that they have more KOPs than others (see Table 3), leading to scattered search results in a large search space. Even so, Falcon performs much better than baselines. From Figure 12(b) we can see that by running more trials (15 trials) after 3 hours and 45 minutes, Falcon can achieve near-optimal results for all *target* models. We also find that after 15 trials, Morphling only works well for CNN and GAN models, which is because most CNN and GAN models have only one KOP—*conv2d*, making them have similar model-level resource sensitivity and thus can get better results. However, other DL models (e.g., Bert-large, GRU, RoBERTa) still perform poorly in Morphling because the resource sensitivity curves of these models vary widely. When the wall-clock time has passed 7.5 hours (30 trials), as shown in Figure 12(c), Morphling's results are greatly improved, with norm.(RPS/Budget) improving to above 0.8 for all models, while results for other baselines improve more slowly.

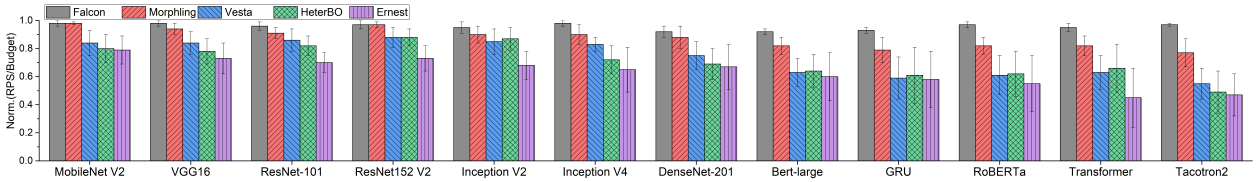
Searching in good regions. One of Falcon's greatest strengths is the significant improvements it makes in the first several trials. Falcon achieves this by employing MCTS-BO



(a) After 1.5 hours (six trials).



(b) After 3 hours and 45 minutes (15 trials).



(c) After 7.5 hours (30 trials).

Figure 12: Searching for a near-optimal configuration for unseen DL models. (a) Evaluating norm.(RPS/Budget) after 1.5 hours (six trials), only Falcon can find near-optimal configurations with over 0.9 norm.(RPS/Budget). (b) Evaluating norm.(RPS/Budget) after 3 hours and 45 minutes (15 trials), Morphling can achieve comparable norm.(RPS/Budget) to Falcon for the first seven DL models because their model-level similarity is higher than other models. (c) Evaluating norm.(RPS/Budget) after 7.5 hours (30 trials), Morphling can achieve 0.8 to 1.0 norm.(RPS/Budget) on all DL models, while other baselines performs 0.5 to 0.8 norm.(RPS/Budget).

Table 3: KOPs of different DL models with varying thresholds.

Models	Setting a 1% threshold	Setting a 5% threshold	Setting a 10% threshold
Inception V2, Inception V4	conv2d, add	conv2d	conv2d
MobileNet V2	conv2d, relu, add	conv2d, relu	conv2d
Bert-large, RoBERTa, Transformer	dense, transpose, add	dense, transpose	dense, transpose
VGG16, ResNet-101, ResNet152 V2, DenseNet-201	conv2d, relu, add	conv2d	conv2d
GRU	split, strided_slice, dense, concatenate, add, sigmoid, tanh, multiply	split, dense, concatenate, add, sigmoid, tanh, multiply	split, dense, multiply
Tacotron2	split, strided_slice, dense, concatenate, add, sigmoid, tanh, multiply, conv1d	split, strided_slice, dense, add, sigmoid, tanh, multiply	split, dense, multiply

to continuously search in good search regions. To evaluate the effectiveness of searching in good regions, we plot the search path of two configuration knobs (GPU memory and batch size) for Falcon and Morphling (we choose Morphling because it outperforms other baselines in our experiments). Figure 13 shows the search path of an unseen DL model RoBERTa, where both Falcon and Morphling start with a bad configuration since they randomly pick up a configuration in the first trial. After that, Falcon analyzes the KOPs and matches them with good search regions ($1.6 \leq$ GPU memory ≤ 2.4 and batch size ≥ 64) via MCTS-BO, while Morphling struggles in bad search regions due to poor model-level similarity.

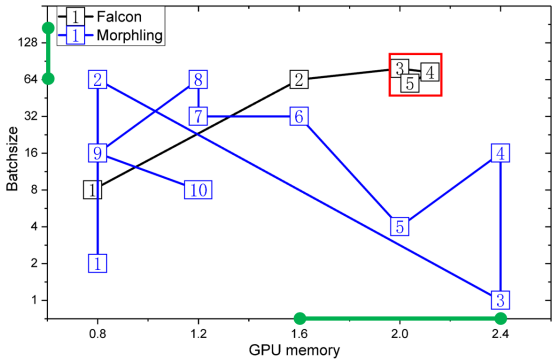


Figure 13: Comparison of the search path for an unseen DL model (RoBERTa). The number in the plot shows the xth trial. The green solid lines highlight the good search regions. The red box highlights near-optimal configurations.

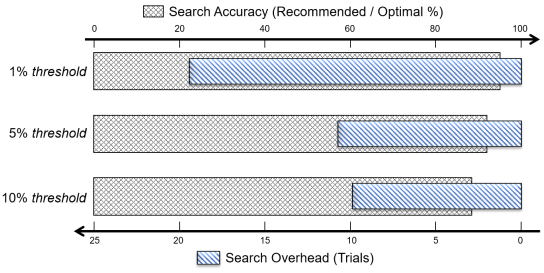


Figure 14: Tuning *threshold* for identifying KOPs. The setting of the *threshold* will impact the search accuracy and search overhead.

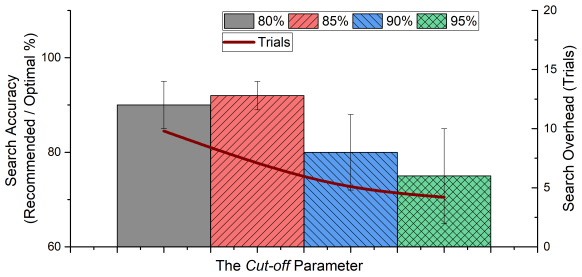


Figure 15: Tuning the *cut-off* parameter in PCA for pruning redundant configurations in KOP-RCs. The setting of *cut-off* will impact the search accuracy (the left Y-axis) and search overhead (the right Y-axis).

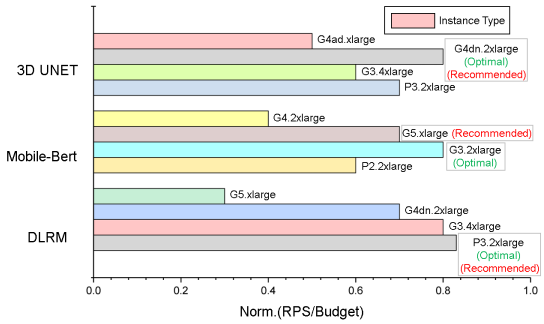


Figure 16: Practical benefits of applying recommended configurations for three benchmark applications. The optimal configurations were found by exhaustive search.

5.3 Robustness

Tuning thresholds for identifying KOPs. Falcon identifies KOPs in the offline phase by setting two thresholds: GPU computation time ≥ *threshold1* and GPU memory utilization ≥ *threshold2*. In this experiment, however, we decided to combine these two thresholds into one (the *threshold* that appears below) for two reasons: (a) our large-scale evaluation shows (see Section 3.1) that GPU computation time and GPU memory utilization are usually at the same level in KOPs, and (b) to avoid that one threshold has a greater impact than the other. Intuitively, setting a smaller *threshold* will keep more KOPs while setting larger one will keep less KOPs. This process is important for strike a balance between search accuracy (keep more) and search overhead (keep less). In this context, we evaluate the *threshold* of 1%, 5%, and 10%. Figure 14 shows that we set the *threshold* to 5% to achieve best practice in our environment and Table 3 shows the KOPs of different DL models (from the *target* set in Table 2) with different thresholds.

Tuning parameters for pruning redundant configurations in KOP-RCs. To further reduce the amount of data and leave only important data, Falcon employs PCA to prune redundant configurations. There is a parameter *cut-off* in PCA to control the configurations to be kept, and we tune *cut-off* from 80% to 95%. Note that, *cut-off* below 80% is not in consideration because we observe that it cannot filter out configurations in the *plane* KOP-RCs (see Figure 5(d)). As shown in Figure 15, Falcon obtains better search accuracy when the *cut-off* is 85%, and the corresponding search overhead is close to six *trials* in average.

5.4 Practical Benefits

To investigate whether Falcon's recommended configurations provide practical benefits for users, we choose three DL applications from an enterprise-level benchmark [29] and evaluate them on Amazon EC2. As shown in Figure 16, Falcon can recommend an optimal (or a near-optimal) instance type for better norm.(RPS/Budget).

6 DISCUSSION

Generalizability of KOPs and KOP-RCs. The generalizability of KOPs and KOP-RCs is crucial for Falcon to alleviate the cold start problem and to find a near-optimal configuration efficiently. Therefore, we do following efforts to guarantee the generalizability of KOPs and KOP-RCs: (a) Some studies [7, 11] have shown the variability of operators developed in different DL frameworks (e.g., TensorFlow, PyTorch). For this reason, we evaluate KOPs and KOP-RCs by Euclidean distance and the result shows that the average variability of KOPs and KOP-RCs in different platforms is less than 6% (KOPs=5%, KOP-RCs=4%). (b) We find that some operators may differ from existing KOP-RCs, making them difficult to evaluate. Fortunately, they are not included in KOPs of any of the models we tested. (c) We consider KOPs and KOP-RCs to be easily scalable. For a new operator, we offline profile it to learn its KOP-RCs (see Section 3.1). For a complicated new pattern that is difficult to represent in tree structure, we can try other complicated machine learning models (e.g., neural networks) if necessary.

Effectiveness of MCTS-BO. We employ MCTS-BO algorithm because it can take full advantage of reusing KOPs and KOP-RCs to search for a near-optimal configuration efficiently. We make the following efforts to guarantee the effectiveness of MCTS-BO in our scenario: (a) MCTS is a game theory-based tree search algorithm that is commonly used to solve search efficiency problems in a large search space (e.g., AlphaGo [18]). Recent studies [34, 44] show that *tree height* is an important hyper-parameter that affects the efficiency of MCTS. In this case, we measure the *tree height* empirically and set it to eight in our environment. (b) In our design, we use MCTS to navigate the search (see Figure 8b) for relatively small search regions in a large search space. When the search result becomes worse, we use BO algorithm [9] to identify potentially good search regions. In particular, we evaluated commonly used algorithms, including BO, SVM and LR, and BO achieved better results.

7 RELATED WORK

Many systems have been proposed to recommend optimal configurations and optimize resource allocation. However, none of them can solve our problem.

Configuration recommendation. These efforts recommend optimal configurations for cloud applications in two ways: (a) *Black-box searching* [2, 8, 14, 24, 43, 48, 49]. Morphling [43] trains a meta-model offline that captures the similarity of resource sensitivity curves for a wide range of DL inference services. It searches for the optimal configuration by combining BO and meta-learning techniques, and use meta-model to accelerate the search process. HeterBO [48] makes a balance between scale-up (more capable instances) and scale-out (more instances) for ML training. It achieves this by profiling machine learning specific prior from ML models and leverages a *cost-aware* BO algorithm to adapt to similar ML models. However, as we discussed in Section 2.1, these approaches are inefficient for unseen DL models. (b) *System modeling* [10, 15, 19, 33, 38, 45]. Vesta [45] abstracts knowledge from offline profile of cross-framework applications. It builds a two-layer bipartite graph to represent cross-framework knowledge and reuse them through transfer learning. Hence, it can adapt to applications developed by different frameworks. Ernest [38] summarizes computation and communication patterns for large-scale analytics. It trains a linear regression model to predict a near-optimal configuration. However, these approaches need to retrain their models which is time consuming.

Resource allocation optimization. Recent studies [11, 17, 20, 22, 26, 28, 51] analyze the resource sensitivity for DL models to optimize their resource allocation. Pollux[28] proposes a formulation of throughput rate for DL workloads and optimizes the actual throughput rate by co-adjusting the resource allocation and model configuration. nn-Meter[51] proposes a system for predicting the inference latency of edge devices by partitioning inference process into kernels and adaptively tuning their configurations to reduce the latency. Pesto[11] jointly optimizes DL model placement and scheduling at a fine-grained operator-level to minimize communication between GPUs while maximize model parallelism. However, none of them focus on unseen DL models.

8 CONCLUSION

This paper presents Falcon, a novel CR system that can quickly adapt to unseen DL models to maximize the performance per budget. The main insight is that Falcon presents a new perspective to alleviate the cold start problem by leveraging Key Operators (KOPs) and Key Operator Resource Curves (KOP-RCs). It first learns KOPs and KOP-RCs from a large-scale evaluation on Amazon EC2. Next, it represents KOPs and KOP-RCs in trees to divide search space into good and bad search regions. Finally, it employs MCTS-BO algorithm to search efficiently in good regions and avoid to continuing searching in bad regions. Experiments show that Falcon significantly reduces the search overhead for unseen DL models from dozens of trials to an average of six.

ACKNOWLEDGMENTS

We thank our shepherd Cheng Tan and the anonymous reviewers for their constructive feedback and suggestions to improve this work. This work was partially supported by Provincial Key Research and Development Program of Shandong, China (2021CXGC010101) and National Natural Science Foundation of China (No.61972386).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 469–482.

[3] Amazon EC2 2022. Amazon EC2. https://aws.amazon.com/

[4] Amazon EC2 Instance Explorer 2022. Amazon EC2. https://aws.amazon.com/cn/ec2/instance-explorer/

[5] Graeme Best, Oliver M Cliff, Timothy Patten, Ramgopal R Mettu, and Robert Fitch. 2019. Dec-MCTS: Decentralized planning for multi-robot active perception. *The International Journal of Robotics Research* 38, 2-3 (2019), 316–337.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. https://doi.org/10.48550/ARXIV.2004.10934

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[8] Romain Egele, Prasanna Balaprakash, Isabelle Guyon, Venkatram Vishwanath, Fangfang Xia, Rick Stevens, and Zhengying Liu. 2021. AgEBO-tabular: joint neural architecture and hyperparameter search with autotuned data-parallel training for tabular data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[9] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. 2019. Scalable global optimization via local bayesian optimization. *Advances in Neural Information Processing Systems* 32 (2019).

[10] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean:{VM} Allocation Service at Scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 845–861.

[11] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. 2021. Towards optimal placement and scheduling of DNN operations with Pesto. In *Proceedings of the 22nd International Middleware Conference*. 39–51.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 1314–1324. https://doi.org/10.1109/ICCV.2019.00140

[14] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent Freeh. 2018. Micky: A cheaper alternative for selecting cloud instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 409–416.

[15] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A Cost-Effective Deep Learning Inference System. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 624–638. https://doi.org/10.1145/3472883.3486993

[16] Dong-Ki Kang, Yun-Gi Ha, Limei Peng, and Chan-Hyun Youn. 2021. Cooperative Distributed GPU Power Capping for Deep Learning Clusters. *IEEE Transactions on Industrial Electronics* 69, 7 (2021), 7244–7254.

[17] Kyungtae Kim, Chung Hwan Kim, Junghwan John Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. Vessels: efficient and scalable deep learning prediction on trusted processors. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 462–476. https://doi.org/10.1145/3419111.3421282

[18] Fangxing Li and Yan Du. 2018. From AlphaGo to power system AI: What engineers can learn from solving the most complex board game. *IEEE Power and Energy Magazine* 16, 2 (2018), 76–84.

[19] Shijian Li, Robert J Walls, and Tian Guo. 2020. Characterizing and modeling distributed training with transient cloud gpu servers. *arXiv preprint arXiv:2004.03072* (2020).

[20] Yan Li, Bo An, Junming Ma, Donggang Cao, Yasha Wang, and Hong Mei. 2020. SpotTune: Leveraging Transient Resources for Cost-efficient Hyper-parameter Tuning in the Public Cloud. *arXiv preprint arXiv:2012.03576* (2020).

[21] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[22] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 937–954.

[23] Xinxin Mei, Xiaowen Chu, Hai Liu, Yiu-Wing Leung, and Zongpeng Li. 2017. Energy efficient real-time task scheduling on CPU-GPU hybrid clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[24] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 831–840.

[25] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[26] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[28] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

[29] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[30] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. https://doi.org/10.48550/ARXIV.1804.02767

[31] Mikita Sazanovich, Anastasiya Nikolskaya, Yury Belousov, and Aleksei Shpilman. 2021. Solving Black-Box Optimization Challenge via Learning Search Space Partition for Local Bayesian Optimization. In *NeurIPS 2020 Competition and Demonstration Track*. PMLR, 77–85.

[32] Scikit-learn 2022. Machine Learning in Python. https://scikit-learn.org/stable/

[33] Tong Shu, Yanfei Guo, Justin Wozniak, Xiaoning Ding, Ian Foster, and Tahsin Kurc. 2021. Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[34] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2021. Monte Carlo tree search: A review of recent modifications and applications. *arXiv preprint arXiv:2103.04931* (2021).

[35] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. 2016. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. https://doi.org/10.48550/ARXIV.1602.07261

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[37] TensorFlow Hub: a repository of trained machine learning models 2022. TensorFlow Hub. https://www.tensorflow.org/hub/

[38] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, USA, 363–378.

[39] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. 2018. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience* 2018 (2018).

[40] VUE: The progressive JavaScript framework 2022. VUE: The progressive JavaScript framework. https://vuejs.org/

[41] DeLiang Wang and Jitong Chen. 2018. Supervised speech separation based on deep learning: An overview. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 26, 10 (2018), 1702–1726.

[42] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. 2020. Learning search space partition for black-box optimization using monte carlo tree search. *Advances in Neural Information Processing Systems* 33 (2020), 19511–19522.

[43] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving. In *Proceedings of the ACM Symposium on Cloud Computing*. 639–653.

[44] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. 2019. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059* (2019).

[45] Yuewen Wu, Heng Wu, Yuanjia Xu, Yi Hu, Wenbo Zhang, Hua Zhong, and Tao Huang. 2021. Best VM Selection for Big Data Applications across Multiple Frameworks by Transfer Learning. In *50th International Conference on Parallel Processing*. 1–11.

[46] Yue-Wen Wu, Yuan-Jia Xu, Heng Wu, Lin-Gang Su, Wen-Bo Zhang, and Hua Zhong. 2021. Apollo: Rapidly Picking the Optimal Cloud Configurations for Big Data Analytics Using a Data-Driven Approach. *Journal of Computer Science and Technology* 36, 5 (2021), 1184–1199.

[47] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *SoCC*. ACM, 452–465.

[48] Jun Yi, Chengliang Zhang, Wei Wang, Cheng Li, and Feng Yan. 2020. Not All Explorations Are Equal: Harnessing Heterogeneous Profiling Cost for Efficient MLaaS Training. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 419–428. https://doi.org/10.1109/IPDPS47924.2020.00051

[49] Jun Yi, Chengliang Zhang, Wei Wang, Cheng Li, and Feng Yan. 2020. Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient mlaas training. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–428.

[50] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[51] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.