RIBBON: Cost-Effective and QoS-Aware Deep Learning Model Inference using a Diverse Pool of Cloud Computing Instances

Baolin Li  
Northeastern University

Rohan Basu Roy  
Northeastern University

Tirthak Patel  
Northeastern University

Vijay Gadepally  
MIT Lincoln Laboratory

Karen Gettings  
MIT Lincoln Laboratory

Devesh Tiwari  
Northeastern University

ABSTRACT

Deep learning model inference is a key service in many businesses and scientific discovery processes. This paper introduces RIBBON, a novel deep learning inference serving system that meets two competing objectives: quality-of-service (QoS) target and cost-effectiveness. The key idea behind RIBBON is to intelligently employ a diverse set of cloud computing instances (heterogeneous instances) to meet the QoS target and maximize cost savings. RIBBON devises a Bayesian Optimization-driven strategy that helps users build the optimal set of heterogeneous instances for their model inference service needs on cloud computing platforms — and, RIBBON demonstrates its superiority over existing approaches of inference serving systems using homogeneous instance pools. RIBBON saves up to 16% of the inference service cost for different learning models including emerging deep learning recommender system models and drug-discovery enabling models.

CCS CONCEPTS

• Computer systems organization → Cloud computing; Heterogeneous (hybrid) systems; • General and reference → Performance; • Computing methodologies → Machine learning.

KEYWORDS

Machine learning, Inference serving, Quality-of-service, Bayesian optimization

1 INTRODUCTION

1.1 Background and Problem Formulation

Deep learning (DL) has been widely used in various areas of applications in recent years. The widely adopted models in areas including scientific computing [1–3], computer vision [4–6] and personalized recommendation [7–10] areas show the significance of studying these models. While model training focuses on the model accuracy and does not have strict time requirements, model inference is often conducted in real-time environments under quality-of-service (QoS) constraints. In this work, we focus on the model inference, where the models are set up to serve a continuous stream of queries submitted by the model users.

A few recent works have started to investigate and quantify the architectural performance bottleneck of various models when serving inference on different types of hardware architectures available in their data centers and platforms [9, 11–14]. However, these works do not address the cost-effectiveness of executing inference queries on different types of hardware. As DL models are becoming more widely adopted and used, inference service providers are more likely to utilize cloud computing resources to serve their customers. Thus, cost-effectiveness is becoming a primary optimization goal along with meeting the QoS targets. Cloud computing platforms offer a wide range of processor architectures and hence, how to properly make use of the various available resources for DL models to meet the QoS targets and minimizing the cost becomes a key challenge. This work addresses this challenge and advances the state-of-the-art by demonstrating how to serve inferences effectively under two competing objectives: QoS target and cost-effectiveness.

1.2 Contributions

This work makes the following contributions.

First, we provide evidence to demonstrate the performance and cost trade-off in serving deep learning inferences on various AWS cloud computing instances, and present a novel approach to exploit this trade-off. Our experimental characterization reveals that the performance ranking of the different cloud computing instances can be often different than their cost-effectiveness ranking, even when considering only a single model. This characteristic makes it difficult to find the optimal computing instance for serving model inferences when co-optimizing for cost and QoS together. To exploit this trade-off between cost and QoS, this work approaches the problem with a new perspective: instead of exhaustively looking for a cheaper instance to serve the model with similar or better performance than the current one, it adopts...
the higher-performance and lower-cost instances to create a diverse pool to serve incoming queries cooperatively.

Second, this work introduces design and implementation of a novel framework, Ribbon \(^1\), that minimizes the cost of serving a stream of inference queries while meeting the QoS target. The key idea behind this framework is to build a pool of diverse computing instances (e.g., a combination of different CPU and GPU instance types) to serve model inferences instead of provisioning multiple instances of the same type (a traditional homogeneous pool). Intuitively, Ribbon’s approach of building a diverse pool is rooted in the insights obtained from our experimental study: the optimal instance type changes depending on the model architecture, QoS target, and query batch sizes. Therefore, building a pool of instance types that are of high quality in different dimensions helps us achieve better outcomes when optimizing for multiple competing objectives (QoS target and cost). However, naively building a diverse pool of multiple computing instance types may lead to worse performance and cost. Ribbon formulates this as a search-space optimization problem and designs a Bayesian Optimization (BO)-based technique to find the optimal configuration from the diverse pool of multiple computing instance types.

Finally, Ribbon’s evaluation is based on inference streams emulating production-environment behavior [7, 15–18] and different types of DL models, such as CANDLE (predicts tumor cell line response to drug pairs) and deep learning recommender system models [7, 9, 10]. Our evaluation confirms Ribbon’s effectiveness over multiple competing strategies for multiple representative deep learning models on a range of AWS cloud computing instances. In particular, we show that Ribbon can increase the cost-effectiveness by up to 16% while meeting the QoS target. Ribbon consistently outperforms competing techniques under a wide range of scenarios including different QoS targets, query characteristics, and load fluctuations.

Open-source artifact. Ribbon framework is available at: https://doi.org/10.5281/zenodo.5262865.

1.3 Related Work

In this section, we compare Ribbon against prior works and discuss how the ideas in Ribbon advance current capabilities.

Recent works on memory-heavy models (e.g., recommendation models) have provided insights on the new challenges such workload brings on both training and inference [10, 12, 19]. DeepRecSys [7] builds a systematic approach to split and offload inference queries for higher production-scale throughput. However, none of the prior works have designed an online serving system in a multi-platform cloud environment setting, and exploited the diversity of compute instances for co-optimizing QoS target and cost-effectiveness (the core contributions of Ribbon).

Cloud providers offer a variety of configurations for cloud instances [20–23]. Traditionally, application developers provision instances based on expertise and tuning [24–29]. However, with increasing application types and cloud servers, this approach is inefficient due to its production overhead [30–33]. This led to the development of application-specific tools for optimal instance selection with the aim to maximize performance [34–40]. These approaches often rely on detailed application profiling prior to execution [20, 41–45]. In some cases, they rely on historical training data from similar applications running on similar hardware for performance optimization [46–53]. However, historical performance data is not available for the problem Ribbon is solving and such data will lose significance with an update in the application or the hardware [12, 19, 54–56]. These approaches neither consider the vast cloud-based compute and accelerator options, nor the query stream of different batch sizes to models.

Recent serverless inference frameworks such as MArk and BATCH [40, 57] cannot be directly applied to recommendation models with large embedding tables and other general large-scale models since they would not fit in limited serverless memory space, and the serverless underlying hardware is not exposed to the user, making it difficult to exploit hardware diversity. Existing automatic cloud instance suggestion frameworks [58, 59] do not explore the idea of a diverse instance pool like Ribbon. Several scheduling frameworks spawn queries among cloud instances that scale up depending on the progress of the first few tasks [60–65]. However, unlike Ribbon, they either do not deal with heterogeneous instances or assume that the queries will not change with time [66, 67]. Ribbon introduces a novel use case of Bayesian Optimization for building efficient inference service systems – expanding the surface area of what we had demonstrated previously for other use cases of Bayesian Optimization in the data center/cloud computing resource-management area (i.e., shared resource partitioning for microservices [68], shared resource management for fairness and performance [69], and performance auto-tuning [70]). None of these previous works exploit the diversity in hardware instances for serving cost-effective model inference queries.

2 BACKGROUND

In this section, we provide a brief background on deep learning models and the cloud computing instances used in this study.

### Studied inference models

To demonstrate the effectiveness and wide application range of Ribbon, the models in Table 1 cover

| Model | Description |
|-------|-------------|
| CANDLE [71] | A large-scale fully-connected DNN model in Cancer Distributed Learning Environment (CANDLE) project [72, 73]. Predicts tumor cell line response to drug pairs. |
| ResNet [74] | Developed by Microsoft, a CNN model with residual operations. Widely applied in computer vision areas such as image classification, object detection. |
| VGG [75] | A CNN model available on DLHUB [76]. Widely applied in image recognition areas. |
| MT-WND [77] | Multi-Task Wide and Deep, a recommendation model. Uses multiple DNN predictors in parallel to predict multiple metrics such as click-throughput rates (CTRs), ratings. Used for YouTube video recommendation. |
| DIEN [78] | Deep Interest Evolution Network developed by Alibaba for recommendation. Integrates gated recurrent units (GRUs) to capture time series information. Used in e-commerce recommendation. |
a wide range of deep learning applications. A description for each model is also available in the table. These models can be grouped into two major categories: the general deep neural network (DNN), convolutional neural network (CNN) models as one, and the recommendation-specific DNN and embedding table hybrid models as another.

The reason to include the CANDLE, ResNet, and VGG models is that they are very representative of the deep learning workloads in scientific research. CANDLE is a well-known cancer research project that applies deep learning techniques to biomedical studies, the model architecture is shown in Fig. 1. It combines the output feature of different DNNs to feed into a residual network for tumor’s drug response prediction. The CANDLE model size is larger than other models as it combines multiple DNNs. The residual network is also the key feature in ResNet models used in our study. ResNet and VGG are widely studied models in computer science, specifically in computer vision applications.

On the other side, MT-WND and DIEN are two representative workloads from industry personalized recommendation applications. Such applications are now consuming a large fraction of compute cycles in data centers according to the recent reports [8, 10]. Therefore, studying and evaluating RIBBON’s effectiveness on such models is as important as the general DNN/CNNs. Current state-of-the-art models take a hybrid approach, which is fundamentally different in their structure and characteristics from traditional deep learning models [9]. Each input sample (e.g., a platform user and a product pair) consists of categorical features for the user’s past interaction with products, as well as continuous features for user-specific information such as age.

As shown in Fig. 2, categorical features are processed by embedding tables and continuous features are processed by DNNs, their outputs get combined and fed to a final DNN for click-through rate (CTR, probability that user clicks on the product) prediction. The size of embedding tables is typically large due to the high number of product categories (in the order of tens of GBs of memory).

**Table 2:** Studied AWS instances. A brief description with example application fields is provided for each category.

| Category       | Instance | Size | Description                                                                 |
|----------------|----------|------|-----------------------------------------------------------------------------|
| General purpose| t3, m5,  | xlarge | t3, m5, and m5n families offer a balance of compute, memory, and network resources (powered by different processor types). |
| Compute optimized| c5, c5a  | 2xlarge | c5 and c5a family instances are suitable for compute-intensive (c) workloads, but at higher price. c5a uses AMD EPYC and c5 uses Intel Cascade Lake processor. |
| Memory optimized| r5, r5n  | large | r5 and r5n family instances are suited for memory-intensive workloads (r is a co-name for memory-optimized). |
| Accelerator (GPU) | g4dn     | xlarge | g4dn family instances are GPU-based cost-effective instances – targeted for machine-learning and graphic-intensive workloads ( ‘g’ refers to graphics/GPU). |

Inference Service in the Cloud. To efficiently serve the model inference queries under the QoS and budget constraints, we evaluate a wide variety of Amazon’s AWS cloud computing EC2 instances. We chose AWS EC2 as a primary experimental playground because it provides a rich number of node instance types at different price points (e.g., compute-optimized, memory-optimized, accelerators, etc.). The example EC2 instances used throughout the paper are shown in Table 2.

One common characteristic of online inference service is that each query contains multiple requests, the number of requests in a query is called batch size. The batch size varies across queries to the model, and the reason is different for general deep learning models and the recommendation models. In the former case, since adaptive batching is widely applied in deep learning inference [40, 57, 79], the number of requests that are batched into a query varies over time. In the latter case, since a query includes multiple products that need to be ranked and recommended to the user at the same time, it also varies between different queries [7, 9, 10].

**Figure of Merit.** An important metric for serving DL models in the cloud is the mean service latency and tail latency (more methodological details provided in Sec. 5). We define the instance performance as its achievable throughput (queries per second, or QPS), which is the reciprocal of the mean service latency. Specifically, let the mean service latency be $L_m$, the QPS can be written as $\frac{1}{L_m}$. A higher performance indicates that the latency of the queries served on this instance is lower, thus more queries can be processed per unit time. When serving an online query stream, there is a Quality-of-Service (QoS) target as online service is latency-critical. The target is typically specified in terms of tail latencies. Another metric associated with serving is the price per instance, which is defined by the AWS cloud service provider.

### 3 OPPORTUNITY AND CHALLENGES

In this section, first we describe the trade-off between performance and cost in cloud inference environment. Then, based on these trade-offs, we show the possibility of improving a system that is already performing well by introducing a diverse instance pool. Finally, we show the new challenges that come with a diverse pool and how a RIBBON-like solution can be applied for cost-effective and QoS-aware inference service on a diverse pool of instances.
3.1 Performance and Cost Trade-off

Fig. 3 (a) shows an example of how the batch size varies the relative performance of an instance type. For the MT-WND model with batch size 32, most instances shown have similarly high performance. But when the batch size becomes 128, g4dn becomes the optimal instance, significantly outperforming other instance types. This is also an expected result as GPU’s streaming multiprocessors are more suitable for inputs with larger batch sizes.

Interestingly, the results in Fig. 3 (b) tell a different side of the story about the cost-effectiveness. The cost-effectiveness is simply defined as the number of queries that an instance can process per dollar spent, as shown in the following Eq. 1.

\[
\text{Cost-Eff.} = \frac{\text{Perf. (query/sec)}}{\text{Price ($/hour)}} = \frac{3600 \cdot \text{Perf.}}{\text{Price}} \text{(query/$)}
\]

Even though the GPU instance g4dn has very strong performance for relatively larger batch sizes, its cost-effectiveness is the lowest among all instances shown. On the other side, the r5 and r5a instances are low on the performance of higher batch sizes, but their cost-effectiveness are consistently the highest. This is because, with the same amount of memory allocated, the memory optimized instances incur less cost than the general purpose instances and much less than compute optimized and accelerator instances. Notice that the cost-effectiveness only relates to the number of queries that an instance can serve given a budget; it does not consider whether the service satisfies the latency requirement. If there is no latency requirement, it would be optimal to serve the models with r5 or r5a instances. However, in a QoS-aware scenario, the cloud user would have to serve with g4dn just to stay within QoS requirements despite the extra cost.

3.2 Potential of Building Diverse Instance Pool

For a deployed cloud service, we assume it is already running at minimal cost on a specific instance type - reducing the number of total instances for lower cost will violate QoS. However, we ask ourselves a question: is it possible to further reduce the cost while not violating QoS? One common approach is to look for a more cost-effective instance type that also has the same or better performance so it does not violate QoS. But this is not always feasible as such instance type may not even exist. Instead, RIBBON takes an unorthodox approach - it makes use of instance types that would violate the QoS with lower performance, but also with lower cost. Based on the discussions in Sec. 3.1, we can see that the cost-effective instance could still be useful for lower batch size queries even though it greatly violates the QoS for larger batches. If we form a diverse instance pool composed of both the higher performance cost-ineffective instances and the lower performance cost-effective instances, there is an opportunity that we can lower the total serving cost while staying within the QoS.

We illustrate such improvement opportunity using a straightforward example. Fig. 4 quantifies the price and latency of serving a query stream of MT-WND model. The query inter-arrival times, batch size distributions and QoS target used in this example have been chosen to reflect production workloads [7, 15, 16] (Sec. 5). The red vertical line indicates the QoS target, meaning at least 99% of the queries have to be within the $99^{th}$ percentile tail latency target (20ms). Among all instances, the g4dn instance is the optimal type to satisfy QoS with minimum cost, no other instance can replace g4dn for a lower cost. Meeting the QoS requires at least 5 g4dn instances in total (orange triangle), as only 4 such instances would significantly violate QoS (blue circle), thus 5 g4dn is the current optimum.

To further improve the serving cost, we introduce the t3 instance type. As shown in the plot (purple star), this instance is much cheaper with limited performance: 12 t3 instance cannot satisfy the QoS, but the cost is still cheaper than 5 g4dn instances. If we use a diverse pool composed of both instance types, the serving cost can be further reduced while meeting the QoS target. The plot shows multiple heterogeneous or diverse pool configurations: (3+4), (2+4), and (4+4). We notice that the (4+4) configuration meets the QoS target, but is more expensive than the optimal homogeneous configuration (5+0). In fact, by reducing one instance of g4dn type to (3+4), our resulting configuration becomes less expensive than the (5+0) homogeneous pool configuration while meeting the QoS target. At this point, we have reached a lower serving cost by introducing the diverse pool serving. Further decreasing the g4dn instance number in the diverse pool to (2+4) will result in violation again, thus (3+4) becomes the new optimal configuration. The intuition behind diverse pool serving is that diversity increases our ability to balance the trade-offs (cost vs. performance) across query streams, instead of being one-dimensional. For example, cheaper instances can opportunistically help us reduce cost without violating the QoS when expensive instances are overloaded.

Figure 3: Relative performance (a) and cost-effectiveness (b) of instances serving MT-WND model with batch sizes 32 and 128. X-axis represents the normalized performance and cost-effectiveness as a score between 0 and 1. The normalization point is the best performance and most cost-effective instance, respectively. “1” indicates the best performance and the most cost-effectiveness, respectively.

Figure 4: MT-WND QoS satisfaction rate and service price for each instance configuration. The configuration (X + Y) means there are X g4dn instances and Y t3 instances used. Configurations on the right of the red vertical line satisfy the QoS requirement.
Figure 5: (a) Configurations with similar cost but significantly different QoS rate. (b) Configurations with significantly different cost but similar QoS rate.

### 3.3 Challenges in Building a Diverse Pool

While the diverse pool can further reduce serving costs beyond heterogeneous optimum, new challenges also appear with this new technique. The general guideline on picking instance types to add to the diverse pool is to use more cost-effective instances that can satisfy a relaxed QoS target. In the example in Fig. 4, we first relax the QoS target of 20ms for about 30% to 26ms, and t3 is an instance type that can satisfy the relaxed QoS while being more cost-effective than the g4dn as shown in Fig. 3(a). We find this degree of relaxation works well for all models during our experiments, instances selected with too much relaxation will not appear in the optimal diverse pool configuration as such instances would inevitably violate the QoS for its queries. As shown later in Sec. 5, this one-time profiling effort to form a diverse pool can also be skipped as the effective pool tend to be common for models of the same category. The real challenge is, once we have created a diverse pool, how to choose the number of each instance type such that the diverse configuration is optimal, i.e., how to quickly find the (3+4) configuration.

Finding the optimal diverse pool configuration is a non-trivial problem. Firstly, more dimensions are added to the search space as the number of each instance type represents a dimension. Due to higher dimensions, the search space becomes exponentially larger. Secondly, the interaction between the variable (a diverse pool configuration) and the corresponding QoS satisfaction rate cannot be described mathematically, resulting in “costly” evaluation for each configuration as the configuration has to be deployed to serve the queries to get its QoS satisfaction rate. This mathematically indescribable interaction is due to the unknown latency for each query and the randomness of query arrival and batch size. We assume no prior/offline knowledge about the latency corresponding to queries with a specific batch size is available, therefore we cannot simply map queries of certain batch sizes to certain instance types. In this work, we apply a straight-forward first-come-first-serve policy for all queries (more details in Sec. 5). Thirdly, when searching for a better configuration, there is no clear sign to follow, and the result can be counter-intuitive. As shown in Fig. 5 (a), configurations with similar cost may have significantly different QoS satisfaction rate, and in Fig. 5 (b), configurations with significantly different cost (config. 4 has almost twice the price of config. 3) may still have similar QoS satisfaction rate.

To address this challenge, RIBBON formulates this as an optimal configuration-finding optimization problem and designs a BO-based solution to find the optimal configuration with a minimum number of configuration evaluations.

### 4 RIBBON: THE SOLUTION

RIBBON determines the optimal heterogeneous/diverse pool configuration via search space exploration process. At the start of the exploration process, it takes the instance types and QoS target as the inputs. The goal is to determine how many instances of each instance type should be chosen (the optimal pool configuration) such that it meets the QoS targets with minimum cost. The most challenging part is quickly determining the number of instances for each type that leads to the optimal pool configuration – the number of evaluations for different such configurations should be small. RIBBON uses Bayesian Optimization (BO) at its core to mitigate this challenge. BO is chosen because it is a lightweight online learning model that does not require expensive training, apriori knowledge, or data. Our evaluation demonstrates that RIBBON’s BO is effective when compared against other competing search space exploration methods (Sec. 5).

**Working principles of RIBBON’s BO Engine.** BO is a black-box optimization method for minimizing an unknown objective function ($f(x)$) and is typically used when the sampling of the configurations is costly [80, 81] – RIBBON’s context shares this similarity. Next, we briefly describe how RIBBON uses a BO-based decision making engine at its core (Fig. 6).

Initially, RIBBON’s BO engine has no knowledge of the nature and distribution of the unknown objective function ($f(x)$). BO maintains a prior belief of $f(x)$ through its surrogate model. This surrogate model serves as a proxy for the true objective function. The success of BO depends on how the surrogate model represents the true objective as the optimization proceeds. RIBBON selects Gaussian Process (GP) as its surrogate model since it is adaptive toward representing any random process and is not input- or objective-specific. This enables RIBBON to handle changing streams of requests over various cloud instances, which is critical in RIBBON’s context. Other surrogate model choices such as Tree Parzen estimators and polynomial estimators are not suitable for RIBBON because they assume conditions that are not true for RIBBON (e.g., polynomial objective function, samples can be divided into classes).

As RIBBON starts sampling configurations, it gathers the true value of $f(x)$ at those configurations (posterior). RIBBON uses this knowledge to update the value of the surrogate model at sampled configurations. The surrogate model uses its covariance kernel to extrapolate the value of $f(x)$ for configurations that are not sampled yet. This extrapolated value is a function of two factors: (1) the covariance of sampled configurations, and (2) their distance from the not-yet-sampled configurations. RIBBON chooses the Matern 5/2 covariance kernel [82, 83] for ensuring smoothness, and it also
provides the capability that similar configurations will result in similar objective values, a desirable property in the context of Ribbon. Other possible choices of covariance kernels include Dot Product and Rational Quadratic assume the configurations to follow a monotonic or a particular kind of polynomial distribution of objective values – therefore, unsuitable for Ribbon.

The surrogate model of Ribbon gets updated after each sample is evaluated. The surrogate model associates a confidence value with configurations that are not sampled. It is inversely proportional to the variance of \( f(x) \) for the nearby sampled configurations. Ribbon needs to intelligently sample to move toward the optimal configuration. So, how does Ribbon decide what to sample next?

BO has an associated acquisition function \( a(x) \) which helps Ribbon determine what to sample next. The acquisition function takes into account the surrogate model and the confidence of the surrogate model for configurations that are not sampled yet (higher confidence means more likelihood of the surrogate model to represent the true objective function) to decide the next best configuration to sample. Ribbon uses Expected Improvement (EI) as its acquisition function. For each unexplored configuration, EI uses its GP mean and variance as input and calculates the expected improvement over the best explored configuration. By maximizing this expected improvement over all unexplored configurations, BO aims to sample a new configuration that provides the highest improvement over the current best.

This acquisition function is used by Ribbon to steer its way to the optimal configuration by actually sampling only a few configurations. Ribbon performs its optimization based on the exploration vs. exploitation of configuration space (exploration: exploring configurations with high uncertainty to find the value of the optimal configuration, and exploitation: exploring the vicinity of configurations where the objective function is already sampled). The acquisition function needs to maintain a balance between the two for the BO to effectively navigate through the configurations without losing the opportunity to sample better-performing ones. Since the optimization performance of Ribbon depends on the objective function \( f(x) \), it is important to carefully design the objective function so that it represents the metric that Ribbon is optimizing.

**How does Ribbon design an effective objective function?** Recall that Ribbon jointly optimizes for two competing objectives: meeting the QoS target and minimizing the cost. Since these objectives are competing, it is critical to construct an objective that carefully captures Ribbon’s goal: minimizing the cost while meeting the QoS target. Unfortunately, this objective cannot be expressed as a typical optimization function that can be minimized or maximized, unlike traditional use cases of BO [84, 85]. To address this challenge, Ribbon constructs a new objective function.

Ribbon’s objective function is guided by the following principle: when a configuration satisfies the query QoS, the objective function should guide the evaluation toward lower cost; when the diverse configuration does not satisfy the query QoS, the objective function should guide the evaluation towards a higher QoS satisfaction percentage. More formally, suppose the input variable over \( n \) instance types is \( X = \{ x_1, x_2, \ldots, x_n \} \) where \( x_i \) indicates number of instances of type \( i \). Eq. 2 shows the objective function designed over the two optimization aspects of Ribbon:

\[
\begin{align*}
  f(x) &= \begin{cases} \\
    \frac{1}{2} \frac{R_{qos}(x)}{T_{qos}} & \text{if violates QoS,} \\
    \frac{1}{2} + \frac{1}{2} \left(1 - \frac{\sum_{i=1}^{n} Pr(X_i)}{\sum_{i=1}^{n} Pr(X_i)}\right) & \text{otherwise.} 
  \end{cases}
\end{align*}
\]

\( R_{qos}(x) \) is the QoS satisfaction rate achieved by the chosen configuration, constant \( T_{qos} \) is the QoS target (i.e., 99% of queries meet a latency target). Constants \( p_i \) and \( m_i \) are the unit time price and the upper bound of the number of instances for instance type \( i \), respectively. This objective function returns a normalized output between 0 and 1, its design ensures that any configuration that satisfies the QoS is superior than a QoS violation configuration regardless of the serving price, as \( 0 \leq R_{qos}(x) < T_{qos} \). Maximizing the objective function guarantees the minimization of the serving cost while meeting QoS. Note that while this objective function can be evaluated at specific instance configurations when a model is run on that configuration, its general shape over the entire configuration space is unknown (\( R_{qos}(x) \) cannot be known without evaluating configuration \( x \)). Therefore, Ribbon’s BO attempts to obtain the true shape of the objective function and find its maximum with as few configuration evaluations as possible.

The parameter, \( m_i \), determines the upper bound on the search range. \( m_i \) corresponds to the maximum number of instances of a given type such that adding any more number of instances of the same type does not improve the QoS satisfaction rate. For example, when serving with \( u \) instances of a certain type, the QoS satisfaction rate is 95%, and this rate stays at 95% even when serving with \( u + 1 \) instances, then \( m_i \) for this type is chosen as \( u \).

We also experimented with other objective functions (e.g., a traditional single-metric function that returns the total cost but only for configurations that satisfy QoS and minimize the returned cost), but such design did not work well. The major lesson learned from this exploration was that the objective function needs to consider both the criterion (cost and QoS) together. Secondly, the objective function needs to maintain smoothness. For a non-smooth single-metric objective function, a large portion of the search space will be flat, which cannot provide guidance for configurations with a better QoS satisfaction rate. This is why Ribbon designed Eq. 2 to increase smoothness over both QoS-satisfaction and QoS-violating regions. Furthermore, we learned that the boundary between the QoS-violating region and QoS-satisfaction region can make it difficult to optimize for the acquisition function over a sudden change, and hence a steep jump needs to be avoided.

We also considered batch size distribution in the objective function, but we noted that changing the batch size only changes the search space, the underlying BO process will still find the optimal configuration. Also, we noted that exploiting batch size distribution information risks making the optimization benefits sensitive to such parameters, which may not be always desirable.

**Ribbon maintains a smooth distribution of configurations.** Recall that BO considers the outcomes of neighboring sampled configurations to determine the next configuration to sample. Specifically, BO’s underlying GP sets the surrogate model value of the
not-yet-sampled configurations based on the variance of already-sampled neighboring configurations. Consequently, the surrogate model indirectly guides the acquisition function toward the optimal configuration. Therefore, it is important that the true objective values of neighboring configurations follow a smooth distribution (i.e., less variance in the objection function values). High variance makes the acquisition function repeatedly sample configurations in the vicinity of that neighborhood, losing the opportunity to explore potentially better configurations in other neighborhoods.

Ribbon employs two specific measures to solve this problem. First, we note that QoS-violating configurations are sub-optimal regardless of how close they are in meeting the QoS, but Ribbon deliberately considers their performance in the objective function (Eq. 2). This makes the objective function smoother in neighborhoods with QoS-violating configurations. In a naive design, where all QoS-violating configurations simply return 0, the objective function will have a large sudden change in value between QoS-violating and QoS-satisfying configurations – Ribbon avoids this. (2) In individual dimensions (particular cloud instance type), Ribbon arranges the configurations in increasing order (increasing number of instances) before the BO begins to search. This guarantees a smooth distribution of configurations along each of the dimensions.

**Ribbon intelligently handles categorical variables.** By default, BO is designed to handle variables that have continuous values. However, in the case of Ribbon, BO samples configurations that only have categorical values as the number of cloud instances of each type are integers [86]. To tackle this issue, Ribbon applies a rounding technique to the underlying continuous GP kernel \( k(x_i, x_j) \) of BO to form a rounded off GP kernel \( k'(x_i, x_j) \):

\[
k'(x_i, x_j) = k(R(x_i), R(x_j))
\]  
(3)

Here, \( R(x) \) is the rounding function that rounds the variable \( x \) to its nearest integer. This ensures that the shape of the GP better matches the shape of the true objective function. Fig. 7(a) is the default BO where the true objective has a step-like behavior due to the integer categorical values of the configurations. However, the GP (determined by the GP mean) has continuous behavior. In this case the GP does not represent the true objective well. Fig. 7(b) shows how Ribbon uses the rounding mechanism on the GP, making the GP mean better represent the true objective function. The acquisition function remains consistent within the integer rounding range. In Fig. 7(a), when the next sampled configuration falls within the integer range of previously sampled configurations, the acquisition function cannot provide any useful information. In contrast, with Ribbon (Fig. 7(b)), the rounding mechanism guarantees the next sampled configuration to fall into the same integer range of a previously sampled configuration. Therefore, it moves faster toward finding the next sample that optimizes the objective.

The above two techniques (objective function design and categorical parameter handling) make the optimization process feasible and convergent. For example, if we use a non-smooth objective function, BO’s acquisition function optimizer fails in 35% of the cases and returns invalid values. Similarly, if Ribbon drops its mechanism to handle categorical parameters carefully, it would perform repetitive sampling of configurations that fall into the same integer range of previous sampled configurations, resulting in even more total samples than exhaustive search in more than 30% of the cases. Using Ribbon’s design, we converged quickly for all our cases.

**Ribbon performs active pruning to speed up the search.** Ribbon uses its construction of objective function to prune the search space beyond what a default BO can do, further reducing the number of samples needed to reach the optimal configuration. Recall that as per Ribbon’s objective function, a QoS-meeting configuration will be judged to be sub-optimal than the previously evaluated QoS-meeting configuration if the price is higher. A higher price configuration can also violate the QoS, hence, returning a value smaller than \( \frac{1}{2} \) which is sub-optimal. Ribbon actively maintains a prune set \( P \) of such configurations that are unlikely to outperform the current optimal configuration. When a configuration \( x_{c} = [c_1, c_2, ... , c_n] \) is evaluated to violate the QoS by more than a threshold \( \theta \) (e.g. 1%), any configuration \( x'_{c} = [c'_1, c'_2, ... , c'_n] \), where \( \forall \) integer \( 0 \leq i \leq n, c'_i \leq c_i \) cannot meet the QoS. Such configurations are added to the set \( P \). The set is updated after each sampling and is applied to the acquisition function as a constraint such that all the configurations in this set are avoided in the subsequent sampling. Whenever the acquisition function has the highest value for a configuration lying inside the set \( P \), Ribbon’s BO avoids sampling it and samples the next best configuration that has the highest acquisition function value among the ones not in \( P \). As the optimization process progresses, Ribbon’s BO identifies more configurations that belong to \( P \). This decreases the number of configurations to search among, and hence, the speed towards the optimal solution increases.

**Ribbon promptly responds to load changes.** Online service can experience fluctuations in the load. When such changes occur, Ribbon performs scaling at the instance level. Ribbon detects the load changes via performance monitoring – when the load goes up, more queries get queued in the query queue, and the QoS satisfaction rate will drop significantly due to the wait time. By monitoring the query queue size and the current QoS rate, one can determine whether the load has changed.

While previous works [40, 57] have resort to serverless functions as a means of scaling, we find that firstly, serverless functions do not guarantee the latency target for a query as the user has no...
information about the underlying hardware the serverless function runs on. Secondly, some workloads such as the recommendation models come with large embedding tables requiring tens of GBs of memory, general deep learning workloads can also have a large model size and large data size, which easily exceed the memory limit of serverless functions. Therefore, Ribbon performs scaling at the instance level. If the serverless function can guarantee latency targets and the workload can fit into its memory, Ribbon is still compatible with serverless as users can use Ribbon for base instance scaling and use serverless functions to deal with strong bursty workloads.

Ribbon maintains a complete record of the explored configurations in the instance space during its convergence to the optimal diverse pool. This information becomes useful when the load scales up as Ribbon will rely on the exploration record to avoid unnecessary sampling of low-QoS configurations and explore around the QoS satisfaction regions. After the scaling, the previous optimal configuration no longer satisfies the QoS target. The BO needs to restart the whole search process and find the new optimal configuration for the new load. A simple approach is to forget about the previous exploration results and restart BO from scratch. Ribbon tries to make use of the previous exploration results to help with the new BO process instead.

In the beginning, the only explored point is the previous optimal configuration on the new load. Firstly, from the previous exploration record, Ribbon identifies the configurations that return the same or lower QoS satisfaction rate than the previous optimal. Let such configurations form a set $S$. The intuition for collecting this set is that if the previous optimal cannot satisfy the QoS of the new load, any configuration that works as good as or worse cannot satisfy the new load QoS either. For each configuration in set $S$, Ribbon performs the same operation introduced in pruning, and add all other configurations associated with this configuration to the pruning set $P$.

In addition, Ribbon estimates the objective function values of the configurations in $S$ to feed to the new load BO as training data. Normally, the objective function value of an evaluation is only known after it has been evaluated by the BO process. However, since we know the configurations in $S$ cannot meet QoS for the new load just as the previous optimal, we can estimate how much it violates the QoS. From our experiment, we find that the estimation does not need to be highly accurate, as its only purpose is to convey a message to BO, which tells the BO to not evaluate configurations near the set $S$ configurations. We use a simple linear function to estimate the QoS satisfaction rate. For example, let the previous optimal configuration be $A$ and another configuration in set $S$ as $B$. If $A$ has 99.9% satisfaction rate on previous load and 33.3% rate on new load, and $B$ has 90% satisfaction rate on the previous load, we estimate that $B$ has 30% satisfaction rate on new load. With both pruning and estimation applied based on exploration record on the previous load, the new BO process receives a head start. The related scaling results are shown in Sec. 5.

## 5 METHODOLOGY AND EVALUATION

In this section, we analyze our evaluation results to understand different aspects of Ribbon, accompanied by further relevant methodological details corresponding to different evaluation focus.

### 5.1 Evaluated workloads

Ribbon is evaluated using the five models introduced in Sec. 2: CANDLE, ResNet50, VGG19, MT-WND, and DIEN. These models cover a wide range of DL networks: DNN, CNN, embedding tables, RNN. An important experimental methodology detail is the query arrival and batch size distribution. Our evaluation is driven by an actual trace [7] and follows the same batch size and arrival distribution as other recent works in the area which have shown these characteristics are representative of the model inferences in production workloads [15–18]. Specifically, our batch size distribution follows a heavy-tail log-normal distribution, similar to previous work [7] which confirms that heavy-tail log-normal is more representative behavior than traditional log-normal distribution. We acknowledge that the batch size distribution for CANDLE can be potentially different from other models. Therefore, for further robustness of Ribbon, we evaluated Ribbon using Gaussian distribution-based batch size because Gaussian distribution can also capture some production workload behavior [15]. Later in our evaluation and analysis, our results show that Ribbon benefits are not sensitive to the choice of these parameters.

The query inter-arrival time follows a Poisson distribution, same as other works in this area [10, 15–18, 87]. Tail latency is a typical QoS target used in cloud service applications, in this paper we use the 99th percentile tail latency as default. The query tail latency targets are selected within application Service Level Agreement (SLA) range, and ensured that the highest-performance instance (g4dn in our case) can satisfy the targets (otherwise no configuration can meet QoS). For the recommendation models, Google MT-WND and Alibaba DIEN both require an SLA target of tens of milliseconds [88, 89]. Therefore, we set the MT-WND and DIEN QoS targets to be 20ms and 30ms, respectively – similar to targets chosen by previous works for these models [7]. For other models, we set the QoS target to be within 1000ms, consistent with other works [90, 91]. According to the latency for the largest query on the g4dn instance, we set the QoS targets for CANDLE, ResNet50, and VGG19 to 40ms, 400ms, and 800ms, respectively. These targets cover a wide range to demonstrate the effectiveness of Ribbon over a variety of latency targets. We confirmed that Ribbon’s benefits are not sensitive to the chosen target QoS latency and achieve similar results when the QoS targets are varied.

The query processing follows a simple first-come-first-serve (FCFS) manner, with the first arrived query going to the first available instance following the heterogeneous type order in Table 3. We note that multiple queries are served concurrently by the available pool of instances.

### Table 3: Cloud Computing Instance types used for different models (refer to Table 2 for AWS cloud instance code names).

| Model   | Homogeneous Pool | Diverse Pool  |
|---------|------------------|---------------|
| CANDLE  | c5a              | c5a, m5, t3   |
| ResNet50| c5a              | c5a, m5, t3   |
| VGG19   | c5a              | c5a, m5, t3   |
| MT-WND  | g4dn             | g4dn, c5, r5n |
| DIEN    | g4dn             | g4dn, c5, r5n |
cost to meet the QoS is lower for all models (the results show improvement over the best homogeneous configuration instance). The cost savings range from 9% (VGG19) to 16% (ResNet50).

We found that such improvements do not tend to require specifically selected instance pools as they are model-specific. In fact, similar improvements are observed for other diverse instance pools (results not shown for brevity). Further, we note that even a single-digit percent cost saving can be significant when accumulated over time. A cost improvement of a few percent would mean millions of dollars of savings in expenditure. Our results (Fig. 11) also show that RIBBON’s cost savings are not sensitive to the batch size distribution. It still provides significant cost savings even when the batch size follows a Gaussian distribution. This is primarily because of the flexibility a diverse instance pool provides. Varying the batch distribution would make particular diverse pool configurations ineffective anymore (i.e., higher cost, QoS violation), but would also make some other configurations more effective (i.e., lower cost within QoS). The RIBBON optimization framework does not make assumptions about batch size distribution. It is designed to explore the heterogeneous configuration space carefully in order to achieve the best savings fast, regardless of where the actual optimal configuration lies.

Next, we look at how RIBBON helps find the optimal configurations most effectively when compared to other competing techniques for finding RIBBON-like configuration.

5.3 Fast Convergence to the Optimal Diverse Pool Configuration with RIBBON

Competing strategies. To evaluate the efficiency of RIBBON in finding the optimal configuration, we compare it against multiple competing online schemes that search for optimal heterogeneous pool configurations.

Random (RANDOM). This is a relatively simple strategy that evaluates different random configurations in the search space. To make it more intelligent, we do not evaluate a randomly picked configuration if a previous configuration with a higher number of instances for each type does not meet the QoS target, or a previous configuration with a lower number of instances for each type meets the QoS at a lower cost.

Hill Climbing (Hill-Climb). This scheme is designed on the hill-climbing optimization principles, widely-used for fast search space exploration [7, 93–95]. We have customized and optimized for our use case by intelligently increasing and decreasing the number of instances based on the observed QoS and cost in a multi-dimensional search space.

Response Surface Methodology (RSM). The Response Surface Methodology (RSM) [96] is an advanced technique used for exploring a certain unknown surface with a fixed number of samples. We employ an optimized 3-level 3-factor central composite face-centered design to explore the search space, since the central composite design is a widely used and one of the most promising RSM designs [97–99]. The RSM sampled configurations will be evaluated, and the scheme starts exploring around the most promising point.
RIBBON converges to cost-saving heterogeneous configurations with the least number of samples compared to all other competing strategies. In Fig. 10, the results show the number of samples required to reach a certain cost-saving configuration for each model. In order to reach a higher cost saving compared to the homogeneous configuration, more configuration samples are needed to be evaluated. To find the maximum possible cost-saving, RIBBON requires evaluating less than 40 configurations out of roughly 1000s of possible configurations in total, and for the MT-WND and DIEN models, only about 20 evaluations are needed. Compared to RIBBON, the other competing strategies require an order of magnitude more number of samples for the CANDLE model, and more than 2 to 3 times more samples for ResNet50, VGG19, MT-WND, and DIEN models.

The performance of Hill-Climb, RANDOM and RSM varies across models as RSM is better than Hill-Climb on ResNet50 and DIEN models while being close to each other on CANDLE, VGG19, and MT-WND models, meaning the central composite design sampled configurations are closer to the optimal configuration for ResNet50 and DIEN models; RANDOM consistently requires a lot more samples to reach optimum across most models, but RSM and Hill-Climb requires the same number of samples as RANDOM to reach the optimum for CANDLE, because in this case, the search space has a large number of local optima. Overall, RIBBON always consistently outperforms all other strategies to reach either an intermediate cost-saving configuration or the optimal configuration.

To dive further into why RIBBON performs better than other intelligent techniques like Hill-Climb and RSM, Fig. 12 shows a simplified two-dimensional example of the configurations explored by these techniques on the MT-WND model. The two instance types in the diverse pool are g4dn and t3. RIBBON actively searches for configurations across the whole space based on its best estimation from the surrogate model. It sampled the dark red region but was able to avoid extra samples nearby, and it was able to find the optimal configuration in eight evaluations.

On the other hand, Hill-Climb initially climbed to a local optimum at point (4,3), sampled every possible point with a cheaper cost, then realized there is no way to go as both (3,2) and (4,1) violate QoS. It then randomly sampled a new point (dark orange) to restart, and successfully climbed to the global optimum in 13 evaluations in total. Lastly, RANDOM first sampled the five fixed points marked in white, and then, it started searching near the best one (5,0). However, (5,0) is already a local optimum, so it switched to (5,12) and found the global optimum from there (18 evaluations in total). This shows that the RSM sampling does not fit the search space well as the sample (2,6) is actually the closest to the optimal configuration in distance, but was not picked as a starting point due to its QoS violation. In both Hill-Climb and RSM, many configurations around a local optimum are evaluated before moving on, but in RIBBON, being stuck at a local optimum is not observed due to the exploration aspect of its BO engine. RIBBON’s active pruning helps it to continually identify and avoid the set of sub-optimal configurations, thus making its search process faster.
5.4 Low Exploration Cost and High Benefits with Relaxed QoS Constraints

**RIBBON**’s total cost of exploration evaluation is also the lowest among all strategies. It also explores QoS-violating configurations the least number of times. While **RIBBON** minimizes the number of configurations sampled during exploration, a desirable goal is to also minimize the total cost of exploration – although we note that running the optimal diverse pool configuration for some time after it has been achieved by any method would outweigh the cost during exploration. Nevertheless, the cost can be further lowered by exploring fewer and cheaper configurations. Fig. 13 shows the total cost to find the optimal configuration as a percentage of the cost for evaluating all available configurations exhaustively (exhaustive-search cost). For **RIBBON**, this exploration cost is less than 3% for all models, while for other strategies it would cost 10 to 20% of the exhaustive-search cost to find the same optimal configuration as **RIBBON**.

Another desirable characteristic during the exploration process is to minimize the sampling number of QoS violating configurations. During the search process, it is unavoidable to evaluate configurations that violate QoS. Even though it is a temporary process, inference service providers would still find it sub-optimal to sample configurations that violate the QoS. In Fig. 14, during the search process before finding an optimal configuration, the total number of evaluated QoS-violating configurations are compared among the competing techniques. Because **RIBBON** requires much fewer samples, it gets an advantage on limiting the number of QoS-violating samples. In fact, for models CANDLE, VGG19, MT-WND, and DIEN, **RIBBON** samples the least number of QoS-violating configurations during the exploration phase. For example, for model CANDLE, **RIBBON** samples 20 QoS-violating configurations, while the other three techniques sample up to 100 QoS-violating configurations. Interestingly for model ResNet50, RSM is very close to **RIBBON**. In this case, the optimal configuration is close to the central composite design sampled configurations. So, RSM is able to quickly converge to it from one of its initial samples.

These results also support that spawning **RIBBON** exploration is always recommended due to its low exploration cost (less than 3% of exhaustive search, to be recouped within a few hundred queries when steady-state is reached) and lower number of QoS violations even during the exploration. Running **RIBBON** provides the opportunity to avail both transient and steady-state savings.

Our results so far have focused on strict 99th percentile tail latency targets as default for all models. To better understand the effectiveness of **RIBBON**, we relax the QoS target since some service providers can use relaxed QoS targets for certain types of customers. We note making the QoS-target more strict (e.g., 99.9th percentile tail latency) was not possible for some models since even the most performant AWS instances are not able to reach this target.

The benefit of using a diverse pool is more significant when the QoS is relaxed. Fig. 15 shows that if the QoS is relaxed slightly to the 98th percentile tail latency (p98), the cost savings compared to homogeneous configurations become more significant. This is because a more relaxed QoS gives more freedom to use the lower performance but also lower cost instances – this opportunity is exploited by **RIBBON** which builds a diverse pool, but remains unexploited by homogeneous pool instances. For example, the p98 optimal configuration for the CANDLE model consists of 2 c5a, 1 m5 and 6 t3 instances. This configuration is 17% less expensive than the optimal configuration to achieve the 99th percentile tail latency (p99) QoS (6 c5a and 2 t3 instances). Thus, **RIBBON** is capable of achieving further gains for workloads with lower QoS requirements. Nevertheless, we chose stricter QoS requirements as default to present conservative results.

### 5.5 **RIBBON**’s Quick and QoS-friendly Adaptation to Load Fluctuations

When the load level changes, **RIBBON** can quickly adjust to configurations that can serve the new load better, and start searching for the optimal configuration for the new load. In Fig. 16, **RIBBON**’s adjustment to load scale is demonstrated as time series. For each model, the x-axis represents the time as a percentage of total time to find the optimal configuration prior to the load scaling. For example, suppose it takes the CANDLE model 10 minutes to reach an optimal configuration from scratch, then after some time, the load becomes 1.5 times heavier, and a 50% time means **RIBBON** spends 5 minutes to reach the new optimal configuration on the new load. The load change occurs at time 0.
The orange curve shows the percentage of queries that violate QoS for each configuration RIBBON explores. It should stay within 1% for the service 99th percentile tail latency target to be satisfied. The blue curve shows the cost ($) of each configuration normalized to the optimal configuration cost before the load changes. RIBBON’s goal is to find a point such that the orange curve is below 1% while the blue curve is the lowest. We have several observations from this figure.

First, RIBBON can quickly respond to the load change by adjusting to a more expensive and better performance configuration so that the QoS violations are reduced significantly. The queries that violate QoS experience a surge on the load change, reaching more than 95% violation (not shown in Fig. 16 as lower violation regions are more interesting). This is because the current configuration’s allowable throughput is saturated, most of the queries have to stay in the query waiting, the accumulated queries make the queue longer and longer, resulting in violation for almost all queries. Once RIBBON detects such behavior, it resets the search space target values and based on prior knowledge, provides estimation to previously evaluated configurations on the new load, prunes away configurations that would definitely violate QoS. This process greatly helps RIBBON avoid the QoS violation region, explore the QoS satisfaction region and maneuver the diverse pool towards a lower-cost configuration. In the end, RIBBON is able to find an optimal configuration that is also around 1.5 times more expensive than the previous cost. Notice that the system does not know how much the load has changed when we apply the load change, so RIBBON does not know the new optimal cost is around 1.5 times more expensive prior to exploration.

Second, we note that once RIBBON has established an optimal configuration at a given load, the time it takes to converge to a new load optimum tends to be shorter than the previous convergence time. In Fig. 16, the time to reach the optimal cost star is less than 60% of the time needed to reach the previous optimum. This is also due to the benefit of estimation and pruning, as some configurations are considered as “already explored” before the exploration for new load actually starts, avoiding unnecessary sampling of these configurations that would greatly violate the QoS and point RIBBON to more promising regions. It is also worth noticing that after the optimal configuration is found, RIBBON tends to explore some configurations that would greatly violate the QoS again (red spikes after the star in Fig. 16). This is because RIBBON has finished exploiting the more promising configurations, and starts to sample the less promising but unexplored configurations. This is also a good sign that RIBBON should stop searching and settle on the best configuration until this point. In practice, it is possible to reduce the violation spikes during exploration by monitoring the query queue. If the queue stacks beyond a limit, it means the configuration may violate QoS. The evaluation of this configuration can be terminated early to avoid more queries from violating the QoS.

6 CONCLUSION

To the best of our knowledge, RIBBON is the first work to employ the idea of using a heterogeneous pool of cloud computing instances to significantly improve the cost-effectiveness and performance-efficiency of deep learning model inferences. RIBBON finds the most-optimal instance configuration to jointly optimize two competing goals: meeting the QoS and minimizing cost. To solve this problem, RIBBON uses a BO-based exploration-exploitation approach to find the optimal instance configuration. Our evaluation confirms that RIBBON can yield up to 16% cost savings without extensive offline knowledge and can adapt to load changes quickly.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive feedback. We are grateful to first-line responders and essential workers who have worked tirelessly to keep our community safe and functioning during the COVID-19 pandemic, and hence, enabled us to devote time to performing this research work. We acknowledge support from NSF awards 1920020, 2124897, and 1910601, MGHPCA, and Northeastern University. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Yujin Oh, Sangjoon Park, and Jong Chul Ye. Deep learning covid-19 features on cxr using limited training data sets. IEEE Transactions on Medical Imaging, 39(8): 2688–2700, 2020.
[2] Kristof T Schütt, Huziel E Sauceda, P-J Kindermans, Alexandre Tkatchenko, and K-R Müller. Schnet–a deep learning architecture for molecules and materials. The Journal of Chemical Physics, 148(24):241722, 2018.
[3] Bethany Lusch, J Nathan Katz, and Steven L. Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. Nature communications, 9(1):1–10, 2018.
[4] Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep learning vs. traditional computer vision. In Science and Information Conference, pages 128–144. Springer, 2019.
Luo Mai, Guo Li, Marcel Wagenlander, Konstantinos Fertakis, Andre-October Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In 14th [USENIX] Symposium on Operating Systems Design and Implementation (OSDI), 2020.

Sajib Kundu, Raja Ragaswami, Ajay Gulati, Ming Zhao, and Kaushik Dutta. Modulating virtualized applications using machine learning techniques. In Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments, pages 3–14, 2012.

Marcel Wagenlander, Luo Mai, Guo Li, and Peter Pietzuch. Spotmix: Designing distributed machine learning for transient cloud resources. In 12th [USENIX] Workshop on Hot Topics in Cloud Computing (HotCloud'20), 2020.

Changyeon Jo, Youngsu Cho, and Bernhard Egger. A machine learning approach to live migration modeling. In Proceedings of the 2017 Symposium on Cloud Computing, pages 351–364, 2017.

Jeff Zhang, Sameh Elhikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In 12th [USENIX] Workshop on Hot Topics in Cloud Computing (HotCloud'20), 2020.

Ron C. Chiang, Jinho Hwang, H Howie Huang, and Timothy Wood. Matrix: Achieving predictable virtual machine performance in the clouds. In 11th International Conference on Autonomic Computing (ICIAC'14), pages 45–56, 2014.

Álvaro López García, Jesús María De Lucas, Marica Antonacci, Wolfgang Zuev, Martin Davis, Marcus Harz, Lara Loretto Iglesias, German Moltó, Marcin Plociennik, Viet Tran, et al. A cloud-based framework for machine learning workloads and applications. IEEE access, 8:18681–18692, 2020.

Xiao Wang, Yuanjuan Zhang, Shengnan Yu, Xiweil Liu, Yong Yuan, and Fei-Yue Wang. E-learning recommendation framework based on deep learning. In 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pages 455–460, IEEE, 2017.

Moshe Ungar, Alexander Tuzhilin, and Amit Livne. Context-aware recommendations based on deep learning frameworks. ACM Transactions on Management Information Systems (TMIS), 11(2):1–15, 2020.

Hui Fang, Danning Zhang, Yiheng Shu, and Guibing Guo. Deep learning for sequential recommendation: Algorithms, influential factors, and evaluations. ACM Transactions on Information Systems (TOIS), 39(1):1–42, 2020.

Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In 2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pages 972–986. IEEE Computer Society, 2020.

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

Yang Li, Zhenhua Han, Quanlu Zhang, and Haisheng Tan. A machine learning based approach to live migration. In 2009 International Symposium on High Performance Computer Architecture, pages 937–954, 2009.

Christian Demetroulis and Christos Kozaris. Paragong: Qos-aware scheduling for heterogeneous datacenters. ACM SIGPLAN Notices, 48(4):77–88, 2013.

Christian Demetroulis and Christos Kozaris. Quasar: Resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices, 49(4):127–144, 2014.

Gengfeng Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. Towards (GPU) utilization prediction for cloud deep learning. In 12th [USENIX] Workshop on Hot Topics in Cloud Computing (HotCloud'20), 2020.

Jiexiu Li, Keyuan Li, Yaodong Tu, and Xiaofeng Xia. Efficient serving in large scale deep learning. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 109–120, 2017.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hristi Aradhya, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Isir, et al. Sockeye: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the AAAI conference on artificial intelligence, 33(01):5941–5948, 2019.
[92] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In Proceedings of the 26th international conference on world wide web, pages 173–182, 2017.

[93] Shuang Chen, Christina Delimitrou, and José F Martinez. Parties: Qos-aware resource partitioning for multiple interactive services. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 107–120, 2019.

[94] Ali Riza Yildiz. An effective hybrid immune-hill climbing optimization approach for solving design and manufacturing optimization problems in industry. Journal of Materials Processing Technology, 209(6):2773–2780, 2009.

[95] Mohammad Shehab, Ahamad Tajudin Khader, Mohammed Azmi Al-Betar, and Laith Mohammad Abualigah. Hybridizing cuckoo search algorithm with hill climbing for numerical optimization problems. In 2017 8th International conference on information technology (ICIT), pages 36–43. IEEE, 2017.

[96] Nuran Bradley. The response surface methodology. PhD thesis, Indiana University South Bend, 2007.

[97] M Ahmadi, F Vahabzadeh, B Bonakdarpour, E Mofarrah, and M Mehranian. Application of the central composite design and response surface methodology to the advanced treatment of olive oil processing wastewater using fenton’s peroxidation. Journal of Hazardous Materials, 123(1-3):187–195, 2005.

[98] M Bashiri and A Farshbaf Gerammaryeh. Tuning the parameters of an artificial neural network using central composite design and genetic algorithm. Scientia Iranica, 18(6):1600–1608, 2011.

[99] Thiagarajan Rajmohan and Kayaroganam Palanikumar. Application of the central composite design in optimization of machining parameters in drilling hybrid metal matrix composites. Measurement, 46(4):1470–1481, 2013.
Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED
The paper experiments report the convergence rate to the optimal hardware configurations when running different deep learning and recommendation models. We compare our Bayesian optimization technique against several competing techniques and conclude that our technique converges to the optimal configuration the fastest with the least cost. We have used an open-source Bayesian optimization library in python (fmin BayesianOptimization), and modified the source code of this library. To set up the Bayesian optimization engine, first follow the README file to install the dependencies on an x86-64 Linux machine (Ubuntu 18.04 and CentOS7 are verified to work). The original repository needs to be cloned, then the files from Ribbon repository need to be copied to the library repository. Building the library from source and adding it to python library path will finish up the setup. The command line instructions are available in the README file.

After setting up the optimizer library, the experiments are ready to run. To compare Bayesian optimization and competing techniques, follow the README file to extract the characterization logs. These logs will help evaluate whether a certain configuration can meet the target QoS without the need to physically allocate the hardware. Run the scripts in order, and visualize the results. The command line inputs are provided in the “Start Ribbon” section of the README file. After all scripts are executed, two visualized plots will appear, showing the scheme comparisons of number of samples explored and total exploration cost incurred.

The characterization includes several publicly available benchmarks: the CANDLE model, ResNet50, VGG19, and two recommendation benchmarks Multi-Task Wide and Deep (MT-WND) and Deep Interest Evolution Network (DIEN).

- CANDLE: ECP-CANDLE Combo
- ResNet50, VGG19: Keras default models
- MT-WND, DIEN: DeepRecSys

We have characterized the above models on the AWS EC2 platform. The instances used for each model are listed in the table below:

| Category               | Instance | Size   |
|------------------------|----------|--------|
| General purpose        | t3, m5, m5n | xlarge |
| Compute optimized      | c5, c5a  | xlarge |
| Memory optimized       | r5, r5n  | large  |
| Accelerator (GPU)      | g4dn     | xlarge |

The size for MT-WND and DIEN models are the same as Table 1. For CANDLE, ResNet50, VGG19 models, the instance sizes are doubled (i.e. t3.2xlarge, r5n.xlarge). We provide instructions to verify the characterization result, even though it does not directly relate to Ribbon’s major results. To characterize a model, allocate an instance on AWS, set up the software environment following the instruction in the “models” directory, and run the python scripts. To simplify the process of setting up the environment, we provide anaconda export files so that the environment can be directly replicated by anaconda with this file. Upon finishing the scripts, the collected data is in json format and can be compared against the logs in the “characterization” directory.

Author-Created or Modified Artifacts:
Persistent ID:
https://doi.org/10.5281/zenodo.5262865,
https://github.com/boringlee24/Ribbon
Artifact name: Ribbon: Cost-Effective and QoS-Aware Deep Learning Model Inference using a Diverse Pool of Cloud Computing Instances

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER
Relevant hardware details: AWS EC2 c5a, m5, t3, g4dn, c5, r5n instances.
Operating systems and versions: Ubuntu Server 20.04 LTS
Compilers and versions: Python 3.7
Applications and versions: CANDLE Combo, VGG19, ResNet50, MT-WND, DIEN
Libraries and versions: Pandas 1.0, Matplotlib 3.2, Numpy 1.18, Scipy 1.4, Scikit-learn 0.24, Jobjlib 0.14, Tensorflow 1.15, Pytorch 1.8, CUDA 11.2
Key algorithms: Bayesian Optimization, Gaussian Process
Input datasets and versions: CANDLE dataset, CIFAR10
URL to output from scripts that gathers execution environment information:
https://github.com/boringlee24/Ribbon/blob/main/env.log