DeepBAT: Performance and Cost Optimization of Serverless Inference Using Transformers

Bowen Sun
Department of Computer Science
William and Mary
Williamsburg, USA
bsun02@wm.edu

Riccardo Pinciroli
Zimmer Biomet
Milan, Italy
riccardo.pinciroli@gssi.it

Giuliano Casale
Department of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

Evgenia Smirni
Department of Computer Science
William and Mary
Williamsburg, USA
esmirni@cs.wm.edu

***Abstract*—Serverless computing is an autoscaling pay-as-you-go paradigm that can efficiently support machine learning inference especially under bursty workload conditions. Within the serverless paradigm, batching ML inference requests before serving them is widely adopted. Thanks to its parallelism properties, batching can highly improve inference performance while reducing the monetary cost of serverless. Identifying the correct serverless parameterization to simultaneously meet conflicting targets (i.e., keep monetary cost at a minimum while meeting pre-defined service level objectives, SLO) may be cast as a resource allocation problem. In this paper, we illustrate that a deep surrogate model can quickly discover optimized serverless configurations by learning the relationship among the workload patterns and achieve performance measures. We develop DeepBAT, an SLO-aware framework that leverages the Transformer encoder and multi-head attention mechanism to optimize the performance of serverless inference subject to bursty and previously unobserved workloads. We illustrate the effectiveness of DeepBAT on a set of case studies and show that for the problem of inference serving on AWS Lambda, DeepBAT can speed up the solution time of state-of-the-art analytic solutions by over 55 times while generalizing remarkably well on unseen workloads.**

***Index Terms*—Bursty Workloads, Transformer, Multi-head Attention, Batching Technique, SLO-Aware, Serverless Computing**

I. Introduction

Serverless computing is a new cloud paradigm that is widely used in big data analysis [1]–[3], Internet of Things [4], [5], and machine learning (ML) tasks including model training [6]–[8] and ML inference serving [9], [10]. A strength of serverless computing is that it simplifies application management thanks to autoscaling [11], [12] coupled with a pay-as-you-go pricing model, which further makes it a cost-effective solution, especially for bursty workloads. However, efficient use of serverless platforms requires proper parameterization. For example, AWS Lambda requires the user to explicitly configure the memory size [13], which has a direct impact on application performance and associated operational costs.

Serving batched inference requests in a serverless platform is a popular strategy to reduce operational expenditure [14]–[16]. Batching takes advantage of the inherent parallelism of ML serving and reduces the monetary cost of serverless by bundling several inference requests to serve them together as a batch. However, for batching to be effective, especially if inference requests are bursty, three parameters need to be identified: the aforementioned memory size, the size of the batch, and the timeout (i.e., when to stop accumulating requests to bundle together and begin serving to avoid unacceptable delays).

Main Challenges. State-of-the-art frameworks that adopt batching [10], [17], [18] aim to identify the optimal system configuration that allows reaching the best trade-off between a user-defined SLO and the serverless monetary cost. BATCH [10], [18] is the first solution in this space and offers an analytical methodology to find the optimal memory, batch size, and batch timeout on AWS Lambda. BATCH is shown to be effective but its effectiveness is limited by how well the immediate past workload is representative of the future (incoming) workload. Sudden workload changes require the parameter optimization to be dynamically adjusted, which bears computational overhead. Therefore, it is unavoidable that a portion of the incoming workload is served with suboptimal parameters until the parameter adjustment is completed.

Motivation. In this work, we explore the ability of deep surrogate models based on the Transformer architecture to return the configuration parameters of inference serving for dynamic and bursty workloads. In addition, we show that the deep surrogate model generalizes seamlessly on *unseen* workloads and offers a competitive alternative to the state-of-the-art. The motivation of this work is three-fold.

1) Deep Surrogates vs. Analytical Models. The optimization of serverless machine learning inference proposed in past work [10], [17]–[20] involves complex trade-offs among parameters such as memory allocation, batch size, and timeout settings. Traditional analytical models [21], [22] often fail to capture the dynamic and non-linear relationships among their inputs and outputs, due to their limitations on assumptions [23]. In contrast, deep learning models offer significant adaptability and scalability advantages by learning complex patterns and relationships thanks to their ability to handle high-dimensional data and capture their inner dependencies [24].

2) Transformer Encoder and Self-attention Mechanism vs. Other Machine Learning Models. While traditional deep learning models like Long Short-Term Memory (LSTM) and Recurrent Neural Networks (RNN) can be used for sequence-based tasks, they often suffer from limitations such as van-

ishing gradients and difficulty in capturing long-range dependencies [25]. The Transformer architecture, leveraging self-attention mechanisms, addresses these issues. Transformers can capture dependencies across long sequences without the limitations of RNNs [26], making them ideal for modeling interarrival times in serverless workloads.

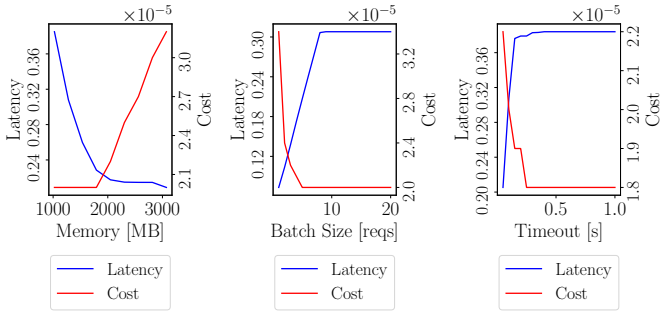
3) Patterns Learned and Generalization Ability. Serverless environments commonly admit highly volatile and unpredictable workloads [27]. The self-attention mechanism in Transformers supports the understanding of the impact of different arrival patterns and configuration settings on inference performance [28]. Additionally, the ability of the Deep Surrogate Model to integrate features such as memory, batch size, and timeout through embeddings allows it to learn complex interactions between these parameters and the workload arrival pattern. This characteristic can allow the model to easily generalize to unseen workload scenarios, especially after its enhancement by a fine-tuning process [29].

Our Contribution. In this paper, we propose DeepBAT that harnesses the power of the Transformer encoder and multi-head attention mechanism of a Deep Surrogate Model for performance and cost optimization of serverless inference. To the best of our knowledge, DeepBAT is the first deep learning-based framework designed to improve ML inference service on serverless, by providing the optimal serverless configurations given a user-provided SLO while only observing a short window of workload data. Comparing DeepBAT to the state-of-the-art BATCH [10], [18], we show that DeepBAT reaches the optimal configurations 55.93 times faster. We show that even for challenging workload conditions and specifically bursty requests, the deep surrogate model gives remarkably accurate predictions and generalizes to *unseen* workloads.

II. Background

We focus on serverless inference, a widely studied real-world application [19], [30], [31], that involves serving inference requests using Machine Learning/Deep Learning (ML/DL) models deployed in a serverless environment. Serverless computing is a cloud paradigm that leverages loosely coupled components (i.e., functions) to provide enhanced performance and simplified application management [32]. Despite the many advantages of serverless, there are still drawbacks that stymie its adoption [33], [34]. For example, developers need to provide certain parameters to maximize request execution efficiency, e.g., AWS Lambda requires users to specify the memory size M for each deployed function that defines CPU, memory, and network performance. Small memory sizes reduce the monetary cost of serverless at the expense of longer latency. As shown in Fig. 1a, underestimating the application memory requirements (i.e., allocating less memory than needed) leads to longer latencies. Conversely, overestimating the memory size can (unnecessarily) increase the monetary cost of deployed functions.

Batching for ML inference utilizes parallelism to improve performance, thus reducing cost by bundling and serving requests together instead of serving them individually. Batching



(a) Memory (b) Batch Size (c) Timeout

Fig. 1: Impact of different values of memory size, batch size, and timeout on latency and cost using trace data [27].

requires two parameters: batch size B and timeout T [17], [35], where batch size is the maximum number of requests to form a batch, and timeout determines the maximum time allowed to wait for requests to form the batch [10]. After a batch is collected or a timeout is reached, inference requests are processed. As shown in Fig. 1b and Fig. 1c, the cost is obviously reduced by choosing higher values of B and T, whereas such configurations also result in larger latency thus risk violating the user SLO. From a practical perspective, the batching parameters (that is, B and T), together with the memory size M should be decided based on the target SLO and subject to a target monetary cost. The identification of the optimal parameters B, T, and M for ML inference is a challenging problem, especially under bursty inference demands [10], [17], [18].

III. Methodology

DeepBAT is based on a deep surrogate model to provide the M, B, and T to optimize the performance and cost of ML inference within a serverless setting. We illustrate the components, control flow, and request flow of DeepBAT in Fig. 2. DeepBAT is an *On-Top-of-Platform* approach (i.e., a new layer on top of the available commercial platform) that leverages a *Workload Parser* to collect workload interarrival times, an *Optimizer* to find the optimal *system configuration* (i.e., memory size, batch size, and timeout), and a *Buffer* to accrue incoming requests. DeepBAT uses a trained model based on Transformer and multi-head attention mechanism to compute the system performance and identify the best system configuration. The deep surrogate model can rapidly process and analyze data compared to analytical models, allowing quick performance predictions. Speed is important in dynamic real-time environments where timely decision-making is crucial.

A. DeepBAT Overview

As depicted in Fig. 2, DeepBAT is placed between users and the serverless platform. DeepBAT can work either as discrete-time control (i.e., trigger prediction at a certain time point) or after an accumulation of inference requests. For simplicity and consistency, we assume that the deep surrogate model inside

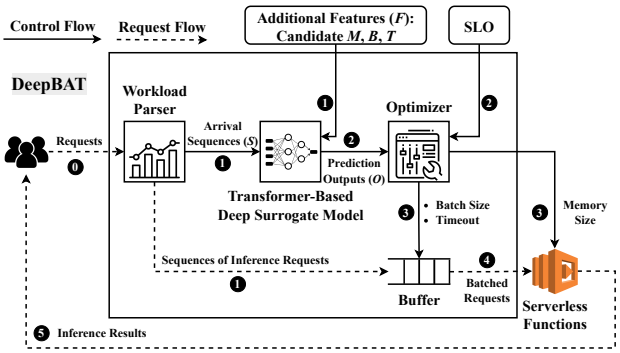


Fig. 2: Design of DeepBAT. Dashed arrows show the request flow, solid ones depict the control flow. All arrows are labeled with a number representing the order of execution.

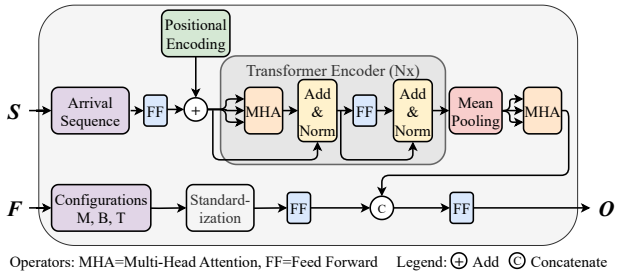


Fig. 3: The architecture of the DeepBAT surrogate model. The model accepts arrival sequence and candidate batching configurations as input, and outputs the predictions of cost and latency percentiles.

DeepBAT returns the optimized configuration once it collects a fixed length of sequence requests as model input. Techniques for padding [26], [36] or sliding windows can be used when there are fewer arrivals.

Inference requests ❶ are initially received and analyzed by the *Workload Parser*. The Parser forwards the sequence of requests ❶ to the *Buffer* component. This component acts as a queue by holding requests until they can be served. The Parser also collects the timestamps of arrival requests ❶ and forwards them to the *Deep Surrogate Model* that uses them as input, together with the candidate system configurations (M, B, and T) ❶. The Deep Surrogate Model ❷ predicts the latency distribution of different system configurations and their associated monetary cost. The output prediction of the Deep Surrogate Model ❷ together with the user-defined SLO ❷ are used as input by the *Optimizer* to identify the optimal system configuration. In addition, the Optimizer seeks to minimize the monetary cost of serving each request by determining the optimal values of the batch size and timeout ❸ that are forwarded to the *Buffer* component and the memory ❸ allocated to the serverless function. Depending on the batch size and timeout values, the Buffer can forward a single request, or a batch of multiple requests ❹, to the serverless function for processing. After the serverless function processes requests, the inference results are returned to the users ❺, completing the request-response cycle.

B. Buffer

The Buffer component manages the collection and forwarding of requests to the serverless function for processing. By strategically batching requests based on the parameters batch size (B) and timeout (T), it aims to minimize the frequency of invoking the serverless function and subsequently reduce its monetary cost.

C. Workload Parser

Unlike state-of-the-art solutions [10], [18] that require a significant amount of incoming requests to fit into a Markovian Arrival Process (MAP) [37], [38], DeepBAT directly uses the original arrival process to parse the inter-arrival time, bypassing the fitting process and eliminates any fitting error [37].

This approach offers two key benefits: it reduces computational overhead by avoiding the expensive fitting process, and it enables more frequent updates to workload statistics, enhancing the ability of DeepBAT to handle dynamic and evolving workloads in real-time.

D. Transformer-based Deep Surrogate Model

Fig. 3 shows the architecture of the deep surrogate model. We assume that at certain time t, the model takes as input arrival sequence $\mathbf{S}$ within a window length (l) of interarrival times $\{x_1, x_2, ..., x_l\}$ and additional features $\mathbf{F}$ (composed of three scalars: memory M, batch size B, timeout T). After processing these inputs, the model outputs a vector of cost and latency percentiles distribution $\mathbf{O}$. The original input arrival sequence is first embedded via

$$\mathbf{E}_{seq} = \text{FeedForward}(\mathbf{S}) \tag{1}$$

The output $\mathbf{E}_{seq}$ maps each element in the sequence from its original space to a higher-dimensional space, which can capture more complex relationships and features that may not be apparent in the raw data. We further have positional encoding for $\mathbf{E}_{seq}$ to add positional information to the sequence embeddings since the Transformer is not aware of sequence order. The output is denoted as $\mathbf{E}_{pos}$. Different from the recurrent neural network (for example, GRU, LSTM), TransformerEncoder could accept as input the complete sequence and infer patterns across time in a parallel fashion with its attention operations. Hence, we could get the encoded output $\mathbf{E}_{Trans}$ as

$$\mathbf{E}_{Trans} = \text{TransformerEncoder}(\mathbf{E}_{pos}) \tag{2}$$

The Transformer encoder can be stacked as N layers to enhance the representation ability during encoding [25], [26].

To prepare for concatenation with additional features $\mathbf{F}$, mean pooling is used for $\mathbf{E}_{Trans}$ to match the dimensions to get $\mathbf{E}_p$. For any three input Q, K, V, we use a multi-head attention [25] with n-head to get Q_i, K_i, V_i for $i \in 1, 2, ..., n$, then use the scaled-dot product attention operation as

$$\begin{aligned} \text{MultiHeadAtt}(Q, K, V) &= \text{Concat}\,(H_1, \ldots, H_n) \\ H_i &= \text{Attention}\,(Q_i, K_i, V_i) \end{aligned} \tag{3}$$

Hence, the attention output of the sequence is calculated as

$$\mathbf{E}_1 = \text{Mask}\left(\text{MultiHeadAtt}\left(\mathbf{E}_p, \mathbf{E}_p, \mathbf{E}_p\right)\right) \tag{4}$$

We argue that with this extra multi-head attention operation for $\mathbf{E}_p$, the model could refine the learned representation and enhance the feature interactions after doing the dimension-reducing via the pooling layer. Moreover, it could help improve the interpretability of the model by analyzing how the pooled sequence representation interacts with additional features $\mathbf{F}$.

For the additional features $\mathbf{F}$, we first implement standardization to scale the values to stabilize the training and improve the performance of the model by lowering the effect of outliers. Then we pass the standardized features through a feed forward layer to get output $\mathbf{E}_2$ as

$$\mathbf{E}_2 = \text{FeedForward}(\text{Standardize}(\mathbf{F})) \tag{5}$$

Finally, to make the prediction based on both sequence information and the additional features, we concatenate $\mathbf{E}_1$ and $\mathbf{E}_2$ and pass them to another feed forward layer to get the final output $\mathbf{O}$, which contains the predicted cost C and the different percentiles P. The output layer may be described as

$$\begin{aligned} \mathbf{E}_O &= \text{Concat}\left(\mathbf{E}_1, \mathbf{E}_2\right) \\ \mathbf{O} &= \text{FeedForward}\left(\mathbf{E}_O\right) \end{aligned} \tag{6}$$

Offline Model Training. We randomly sample a batch of arrival sequence S_t with length l from the processed historical data. Then we further combine S_t with a randomly picked feature set F_t including M, B, and T, chosen from the sub-collection of the whole space. In this way, the model could learn various patterns from the dataset and also generalize well to unseen cases.

During training, we perform grid search to optimize the model parameters and ensure stability and high accuracy. We set the layer numbers of Transformer encoders to 2 and its embedding dimension to 16. The dimension of the hidden state for feed forward network is set to 32 and we use ReLU as activation functions. For loss definition, we use a combination loss of Huber Loss HL and Mean Absolute Percentage Error (MAPE) ML as the training loss function. For any true value y and its corresponding predicted value $\hat{y}$, the Huber loss is defined as

$$HL_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y}))^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta \cdot \left(|y - \hat{y}| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases} \tag{7}$$

and is chosen considering the fact that it is less sensitive to outliers than the squared error loss. We set δ to 1 based on the small magnitude of target inputs. Under these same conditions, the MAPE and the weighted combined loss L used for training are computed as

$$ML(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{y}_i - y_i|}{y_i} \cdot 100\%, \tag{8}$$

$$L(y, \hat{y}) = \alpha \cdot ML(y, \hat{y}) + (1 - \alpha) \cdot HL_\delta(y, \hat{y}), \tag{9}$$

where α is a factor to control the weights of both errors. In this case, we set $\alpha = 0.05$. The loss function is intentionally defined to penalize more for those configurations that violate the SLO, both for latency and cost prediction. We train the model for 100 epochs and load the training data with a batch size of 8. We use the Adam [39] optimizer for parameter tuning and the initial learning rate is set to 0.001. The model is implemented in PyTorch [40]. The decreasing trends of training and validation loss are proved to be stable under used parameters, i.e., a relatively small loss is reached after around 50 epochs. More details are shown in the sensitivity analysis part in section V.

Online Model Inference. The trained DeepBAT model predicts the cost and latency percentiles for various configurations of M, B, and T given an incoming arrival sequence S. This set of predictions is then passed to the Optimizer, which selects the optimal configuration based on a 2-step optimization that balances the SLO constraint and cost, see III-E. The complexity of DeepBAT is $O(n^2)$ where n is the length of the input sequence S. We tune the value of n based on multiple tests to ensure that this is not a bottleneck during deployment. For more details on this, see Section V.

Model Fine-Tuning. To further improve the prediction ability of the model, we also consider the fine-tuning process using coupled simulation [29] if there is a noticeable performance drop observed due to differences in data distributions of the distribution used as trained data and the distribution of the incoming arrival process, (namely out-of-distribution, *short as* OOD). Fine-tuning is commonly used in deep learning [26], [41] to enhance performance, and it can leverage existing knowledge within the model, making it computationally efficient and effective in enhancing model performance in new environments. For our case, we fine-tune the pre-trained model on a small portion of the new dataset for additional epochs, allowing it to better learn its parameters and better capture the specific characteristics of the new OOD data. As a fast reaction to the totally unseen arrival process or dealing with strict SLO constraints, we also set a penalty factor γ to further constrain the target SLO during optimization, making the model more robust. γ can be measured by the MAPE between the predicted latency percentile metric $\hat{P}$ and the simulated ground truth P as $\gamma = (|\hat{P} - P|)/P$.

E. Optimizer

For the sake of comparison with the state-of-the-art, the optimization problem considered here (i.e., maximize the efficiency of serverless inference) uses the same formulation as in [10] and is presented in Eq. (10).

$$\begin{aligned} \text{minimize} \quad & \text{Cost}_{req}(M, B, T) && \text{(10a)} \\ \text{subject to} \quad & F_R^{-1}(i/100) \leq \text{SLO} && \text{(10b)} \\ & B \geq 1 && \text{(10c)} \\ & T \geq 0 \text{ msec} && \text{(10d)} \\ & 128 \text{ MB} \leq M \leq 10240 \text{ MB} && \text{(10e)} \end{aligned}$$

Eq. (10a) states the problem objective, i.e., minimizing the monetary cost of serving individual requests that depend on memory size (M), batch size (B), and timeout (T). Eqs. (10b

to e) are the constraints of the optimization problem: Eq. (10b) requires that the ith-percentile latency is smaller than or equal to the SLO; Eq. (10c) asks that there is at least a request in the batch to be forwarded to the serverless function; Eq. (10d) defines a minimum timeout to be used to collect requests in the buffer; and Eq. (10e) defines memory sizes that can be used (as for [13]).

The Optimizer component of DeepBAT solves this optimization problem in Eq. (10) by performing an exhaustive search within the solution space constructed on the latency and monetary cost predictions provided by the deep surrogate model. Specifically, all possible system configurations (i.e., memory size, batch size, and timeout) are associated with their latency distribution and monetary cost.

IV. Evaluation

In this section, we evaluate the performance of DeepBAT and compare it with the state-of-the-art analytical model [10]. The evaluation uses both real-world and synthetic workloads to assess the accuracy of predictions, cost efficiency, and ability to meet service-level objectives (SLOs). In addition, we record the prediction time required by the deep surrogate model and the analytical model to illustrate the efficiency of DeepBAT.

A. Experimental Setup

DeepBAT Deployment. We deployed a prototype of DeepBAT atop an AWS EC2 instance, i.e., a t2.xlarge virtual machine. DeepBAT's deployment recorded 0.55ms of CPU time and 2MB of memory, as measured by the PyTorch Profiler [40]) to maintain efficient performance. The flexibility of DeepBAT enables it to be deployed efficiently in various serverless environments, although our evaluation focuses on AWS Lambda [10]. The deep surrogate model is trained on profiled data from AWS Lambda.

Ground Truth and Baseline. The ground truth, which represents the optimal system configuration with its latency and monetary cost, is obtained by simulation as in [10], [18]. Note that simulations for the specific problem are validated by extensive experimentation on Amazon Lambda, see [10], [18]. The ground truth is obtained using a search across all possible configurations of memory size, batch size, and timeout.

Inference Requests. We extract inference requests from the TED-LIUM corpus [44], a widely used real-world Natural Language Processing (NLP) dataset. TED-LIUM corpus consists of 2351 audio talks and a total of 452 hours of audio recordings. We profile the execution of inference requests extracted from this dataset to observe their service time against different system configurations. Inference requests have deterministic service times, as established in the literature via experimentation on actual systems [10], [45], [46]. Users can profile the service time of new inference requests with a minimum observation of samples (i.e., manageable and efficient profiling). After profiling the inference requests, we can power the ground truth simulations, as well as predictions from BATCH and DeepBAT.

Workloads. To assess the robustness and effectiveness of DeepBAT, we conduct experiments using both real-world traces [27], [42], [43] and synthetic traces [37]. Fig. 4 shows the arrival rate of the traces.

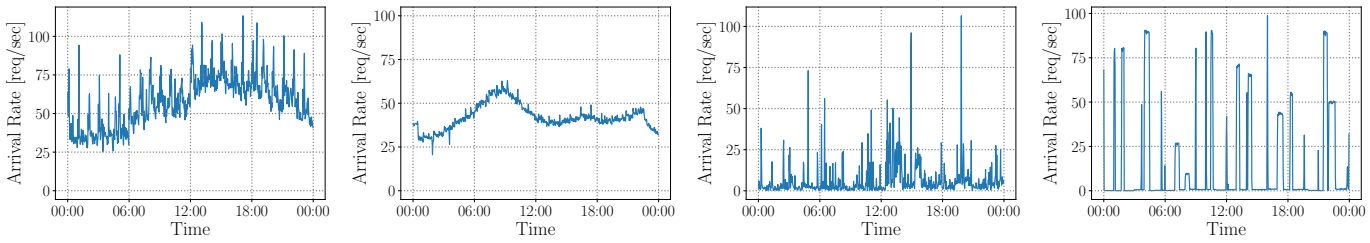
1) *Real-world traces.* The Azure function traces (referred to simply as *Azure trace*) are the representative traces of Azure Functions invocations, collected over two weeks in 2019 and made public via [27]. The Twitter trace is a publicly available subset of requests collected from the general Twitter stream [42]. The Alibaba cluster trace [43], describing the AI/ML workloads in the MLaaS (Machine-Learning-as-a-Service), is made public by Alibaba PAI (Platform for Artificial Intelligence) on GPU clusters. All three traces capture arrival patterns of real workloads, range from moderately to very bursty, and ensure that our evaluation is grounded in realistic and representative workload patterns.

2) *Synthetic trace.* In addition to the real-world traces, we carefully design a synthetic trace to explore the ability of DeepBAT to handle varying degrees of burstiness. Burstiness is a common property of real-world workloads [47], [48], benefiting from the auto-scaling property of the serverless paradigm [11], [49]. Burstiness in the arrival and/or service process causes high variation in performance metrics [50]–[52]. We use Markovian Arrival Processes (MAPs) to capture workload burstiness [37], [38] and generate 24 unique workload streams, one for each 24-hour period. These streams exhibit significant variation, capturing on-off traffic behaviors common in serverless environments.

Fig. 4 illustrates the arrival rate of the four traces used in our evaluation. While both the Azure and Twitter traces exhibit notable variability, the Alibaba and synthetic traces present significantly higher levels of burstiness, making them more challenging.

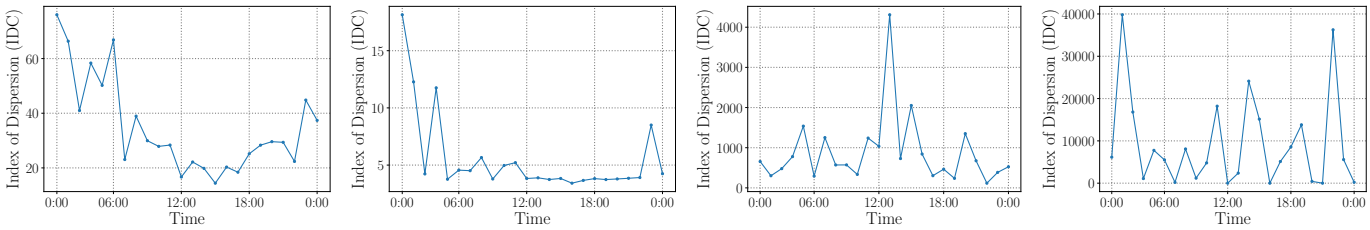
To further quantify and compare the burstiness across the three traces, we calculate the index of dispersion (IDC) of four three traces. The IDC is a well-established metric used to describe process variability, capturing both autocorrelation and the relationship between the mean and variance [53] and is given by $IDC = (\sigma^2/\mu^2)\,(1 + 2\sum_{k=1}^{\infty} \rho_k)$, where μ is the mean, σ^2 is its variance, and ρ_k is its lag-k autocorrelation. By calculating the IDC, we capture the entire autocorrelation function within a single value. For empirical traces, the autocorrelation generally vanishes after high lag values, allowing us to obtain finite estimates of IDC.

Fig. 5 illustrates the IDC of the four traces. The Twitter trace has an IDC of around 4 for most periods (Fig. 5(b)), indicating mild burstiness (autocorrelation). An IDC value of 1 signifies no autocorrelation. The Azure trace has a higher and more variable IDC over time, as shown in Fig. 5(a), compared to Twitter. In contrast, the Alibaba cluster trace and the synthetic trace, see Fig. 5(c) and Fig. 5(d) correspondingly, have much higher IDC values, with significant variability across different hours. The MLaaS requests of the Alibaba trace and the arrival rate of synthetic workload fluctuate sharply between low and high values. These two workloads provide insights into the responsiveness and scalability of DeepBAT under challenging



(a) Azure functions Trace, day 2 [27] (b) Twitter trace, 5/25/2017 [42] (c) Alibaba trace, 3/8/2020 [43] (d) Synthetic trace: 24 MAPs [37]

Fig. 4: Arrival rate of the workloads used to test the prediction accuracy of DeepBAT.



(a) Azure functions Trace, day 2 [27] (b) Twitter trace, 5/25/2017 [42] (c) Alibaba trace, 3/8/2020 [43] (d) Synthetic trace: 24 MAPs [37]

Fig. 5: Index of Dispersion for the four traces: Azure, Twitter, Alibaba, and synthetic (MAP-based).

and bursty workload conditions.

Metric for Evaluation. To evaluate the performance of DeepBAT in terms of compliance with SLOs, we introduce a custom metric, the SLO Violation Count Ratio (VCR). This metric assesses how often the measured latency $\hat{l}_t$ based on predicted configurations exceeds the SLO threshold over a specified time period t. The VCR is defined as:

$$VCR(t) = \frac{\sum_{i \in S_t} 1(\hat{l}_i > SLO)}{|S_t|} \cdot 100\%, \qquad (11)$$

where $1(\hat{l}_i > SLO)$ is the indicator function that equals 1 when the predicted latency for the i-th request sequences exceeds the SLO value, and 0 otherwise. $|S_t|$ is the total number of sequences within t. The VCR provides a clear indication of how often SLO violations occur, with lower values indicating better compliance. This metric allows for a straightforward comparison between DeepBAT and baseline models under different workload conditions.

B. Performance Evaluation with Azure and Twitter Traces

Here, we compare the prediction accuracy of DeepBAT with BATCH and ground truth. We focus on the 95th percentile latency and overall cost. Both DeepBAT and BATCH identify optimal configuration parameters (memory size, batch size, and timeout). The effectiveness of the chosen parameters is then evaluated through simulation.

Our implementation of BATCH follows closely the steps introduced in [10]. Every hour, BATCH profiles the workload and fits its arrival process into a MAP. Fitting is a time-consuming process and requires the collection of sufficient data for a successful fit, this process can take from a few minutes to an hour depending on the workload intensity. Once the MAP is constructed, it is used as input to the analytical model to determine the optimal system configuration [10].

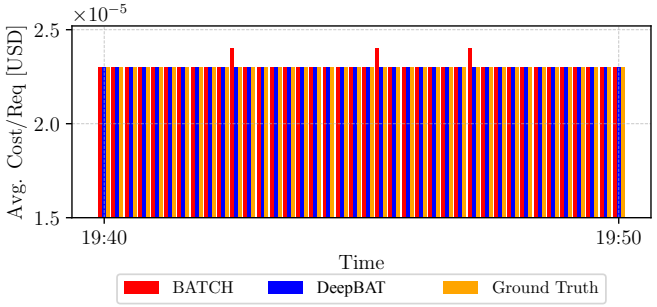


Fig. 6: Comparisons configuration costs returned by BATCH and DeepBAT for the time period 19:40 to 19:50 of Azure.

DeepBAT uses a pre-trained deep surrogate model built on historical data. We train the model using the first 12-hour Azure data, using random sampling (at a 0.05% rate) of the arrival process, along with a deterministic service process. Then, we test the model with the remaining time period of the Azure trace (i.e., the last 12 hours), and directly test it on the Twitter trace. For this test, training is done only once, and costs about 26 seconds per epoch using the 0.05% fraction of data for training. The rest of the data is used for testing.

During testing, DeepBAT processes the original arrival sequences without any fitting or retraining. *Because the Azure and Twitter traces are statistically similar, we directly use the model trainied on AZure, without retraining or fine-tuning, to*

test its performance on the Twitter trace.

Results. DeepBAT and BATCH both perform closely to the ground truth for the Azure and Twitter traces, consistently meeting the target SLO of 0.1 seconds (95th percentile latency) without any violations. BATCH occasionally provides configurations that cost more than the ones returned by DeepBAT. Fig. 6 illustrates the cost comparison for a representative 10-minute interval of the Azure trace as a snapshot (the Twitter trace is not illustrated here because it follows the same pattern due to its similarity in workload characteristics).

The cost advantage of DeepBAT stems from its ability to quickly adjust configurations in response to changing workloads, whereas BATCH is slower. This slower adaptation leads to occasional cost, even though both models avoid SLO violations (VCR=0) due to the moderate burstiness of the Azure and Twitter traces.

We conclude that the DeepBAT works on par with the state-of-the-art BATCH model, maintaining low costs while meeting SLO requirements for unseen, moderately bursty workloads.

Observation #1. DeepBAT works well with real-world traces and generalizes very well to the dataset that is unseen but is statistically similar to the training distribution. It always returns the optimized configurations of batching with the lowest cost while not violating the SLO.

C. Adaptability to Dynamic Workloads: The Alibaba Trace

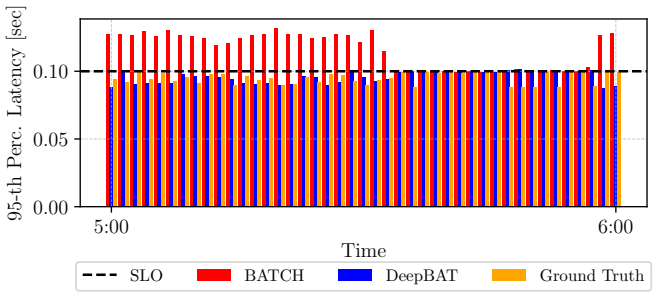
The Alibaba cluster trace presents a highly dynamic workload, with requests exhibiting a wide range of arrival rates, as illustrated in Fig. 4 (c). This is further corroborated by the high IDC in Fig. 5 (c), indicating significant burstiness.

Model Setup. BATCH continues to optimize system configurations hourly [10], but due to the bursty nature of the Alibaba trace, it requires even more than an hour to collect sufficient data for fitting. The SLO is still set to 0.1 seconds.

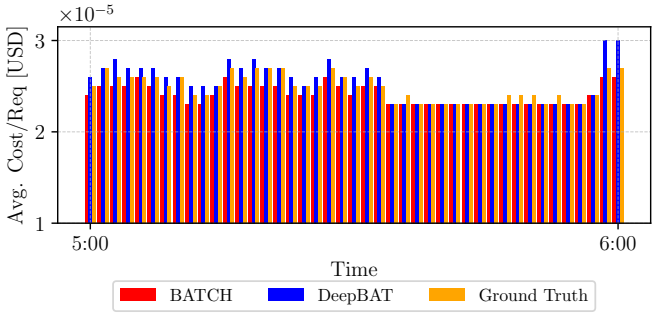
When applying the DeepBAT model trained on the Azure trace directly to the Alibaba trace, i.e., witout fine-tuning, the latency prediction error (MAPE) increases to 5.73%, as expected due to the significant differences between the two datasets. We attribute this to the out-of-distribution (OOD) dataset. To mitigate this, we fine-tuned DeepBAT using data from the first hour of the Alibaba trace, enabling it to better capture the unique characteristics of this workload, as introduced in Section III-D.

Results. Fig. 7 illustrates the latency and cost calculated for hour 5 to 6 based on the configurations returned by DeepBAT and BATCH. The figure shows that the configurations provided by BATCH frequently violate the SLO.

Analysis. The significant performance gap between DeepBAT and BATCH is due to the rapidly changing nature of the Alibaba workload. BATCH's analytical model relies heavily on historical data, which limits its ability to adapt to dynamic changes. For example, BATCH fails to predict the high peak of the 4th hour of the Alibaba trace due to the flat pattern during the previous hour, see Fig. 4. This pattern also repeats in other intervals, e.g., the 6th and 20th hours.



(a) SLO and response time



(b) Cost

Fig. 7: Latency and cost achieved by BATCH and DeepBAT (Transformer) for hour 5-6 using the Alibaba trace.

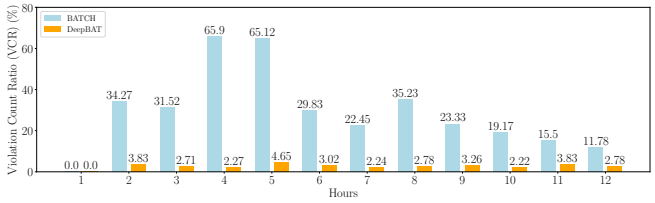


Fig. 8: VCR for 12 hours for the Alibaba trace.

Conversely, when the previous hour shows a peak, BATCH tends to overestimate especially if the workload tends to be less intense in the upcoming hour.

In contrast, DeepBAT demonstrates good performance by leveraging its learning ability through the Transformer model, which enables it to adapt to changing patterns more effectively. As a result, DeepBAT incurs much fewer SLO violations compared to BATCH. Fig. 8 illustrates the VCR (Violation Count Ratio) metric calculated per hour, for 12 hours, for Alibaba for BATCH and DeepBAT. The VCR metric clearly illustrates the superiority of DeepBAT under quickly changing workload conditions. For example, BATCH frequently results in SLO violation during the 4th and 5th hours, primarily due to its limited capability in generalizing from historical data to newly observed workload traces. Specifically, it shows VCR of 65.9% and 65.12%, respectively, during these intervals. In contrast, the fine-tuned DeepBAT achieves much lower VCR values of 2.27% and 4.65% for the same periods. *To clearly illustrate the impact of the fine-tuning step, we further*

evaluated the performance of the pretrained DeepBAT without additional fine-tuning on the 4th and 5th hours, observing VCRs of 14.18% and 17.06%, respectively. This shows the performance improvements of DeepBAT that benefits from the fine-tuning process.

We conclude that the analytical model in BATCH performs well when the workload patterns are similar to historical data. Otherwise, SLO violations occur even though the cost returned by the analytical model seems to be lower (as shown in Fig. 7b) due to the underestimation of the configurations.

Observation #2. When applied to out-of-distribution (OOD) dataset (compared with the distribution of the training set), DeepBAT benefits from fine-tuning to achieve competitive performance. The fine-tuned DeepBAT performs efficiently even under highly bursty and volatile workloads.

D. Evaluation with Extreme Burstiness: The Synthetic (MAP-generated) Trace

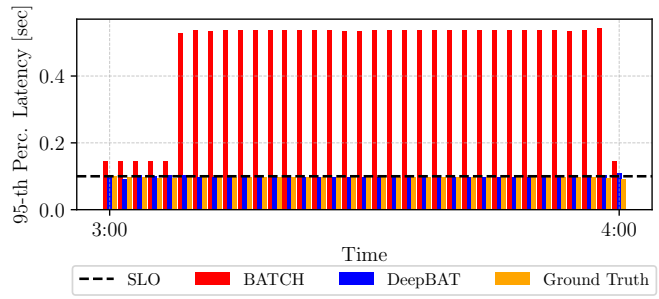
The synthetic workload, as depicted in Fig. 4 (d) and Fig. 5 (d), presents a more challenging scenario due to significant fluctuations in both arrival rate and burstiness. This allows us to evaluate how well DeepBAT adapts to rapidly changing conditions and its ability to learn data patterns.

Model Setup. Similar to the Alibaba trace setup, BATCH is configured to use the one-hour window of data, allowing it to collect enough points, with SLO set to 0.1 seconds. We fine-tune the deep surrogate model, originally trained on the Azure trace, with the first hour of the synthetic trace to capture the unique characteristics of this OOD dataset.

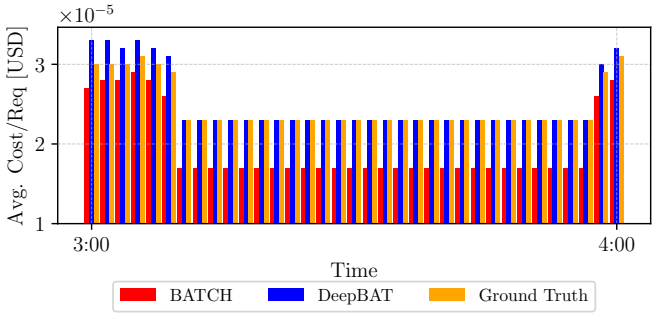
Results. Fig. 9 illustrates the percentile latency and cost calculated for hour 3 to 4 based on the configurations returned by DeepBAT and BATCH. The figure shows that the configurations provided by BATCH violate the SLO, due to the sudden change in the arrival intensity of inference requests, i.e., the previous hour is not a good predictor of the upcoming one. The results are qualitatively similar to those of the Alibaba trace: DeepBAT results in fewer SLO violations than BATCH but unavoidably to higher cost.

Similarly to the Alibaba trace, DeepBAT is able to quickly adapt to this very challenging workload and achieve much fewer SLO violations compared to BATCH. Fig. 10 illustrates the VCR (Violation Count Ratio) metric calculated per hour, for 12 hours, for the Synthetic trace for BATCH and DeepBAT. The VCR metric clearly illustrates the superiority of DeepBAT even under this dramatically changing workload.

Analysis of Returned Parameters. Fig. 11 displays the configuration parameters (i.e., memory size, batch size, and timeout) returned by DeepBAT, BATCH, and the ground truth for the synthetic trace. The graph shows that DeepBAT could adaptively adjust to changes in the workload, often selecting configurations that closely match the ground truth. This adaptivity reduces SLO violations, albeit at the cost of slightly higher serverless charges due to increased function calls. To better understand this behavior, we analyze the loss function of the deep surrogate model training in DeepBAT.



(a) SLO and response time



(b) Cost

Fig. 9: Latency and cost of BATCH and DeepBAT for hour 3-4 using the MAP-generated synthetic trace.

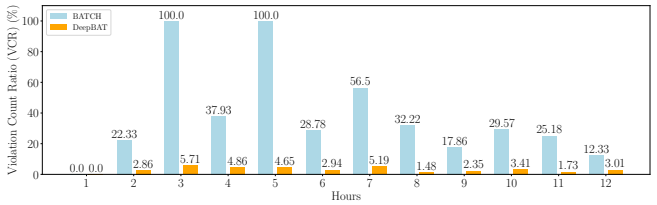
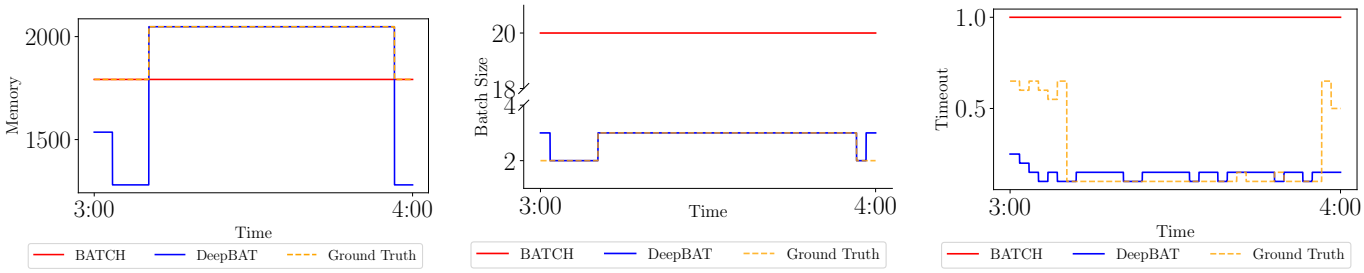


Fig. 10: VCR for 12 hours for the MAP-generated trace.

The loss function is intentionally defined to penalize more for those configurations that violate the SLO, both for latency and cost prediction. This approach ensures that DeepBAT remains within acceptable SLO limits, even if it leads to a marginal increase in cost.

SLO Variations and Model Robustness. To investigate the sensitivity of models to different workload conditions, we varied the SLOs for a direct comparison between BATCH and DeepBAT. Fig. 12 shows the latency based on the configurations returned by both models and compares them with the ground truth, with SLO being set to 0.15 seconds. BATCH still fails to meet the SLO while DeepBAT always returns configurations that yield latency smaller than 0.15 seconds. Experiments with SLO set to 0.05, 0.2, and 0.25 seconds (not shown here for lack of space) confirm that the effectiveness of the analytical model used in BATCH heavily relies on the pattern similarity between the data used to derive the analytical model and the test data. On the other hand, DeepBAT easily generalizes due to its self-learning ability, and benefits from



(a) Memory for Hour 3 to 4 (b) Batch Size for Hour 3 to 4 (c) Timeout for Hour 3 to 4

Fig. 11: Configurations of memory, batch size, and timeout of BATCH and DeepBAT for hour 3-4 of the synthetic trace.

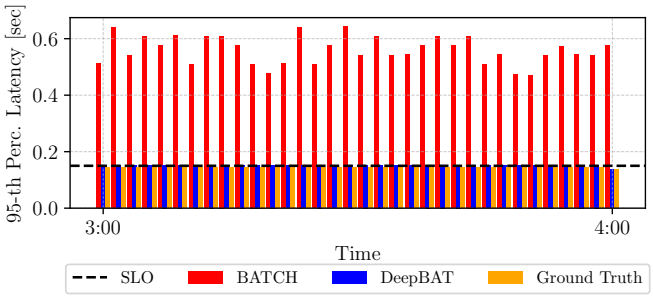


Fig. 12: Comparisons of latency returned by BATCH and DeepBAT for the hour 2-3 using the MAP-generated synthetic trace. The SLO is set to 0.15. The analytical model of BATCH is fitted with the data of the previous hour.

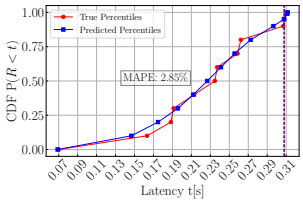
the design of its deep surrogate model to directly learn the patterns from the original data.

Observation #3. For the MAP-generated workload, the analytical model may heavily result in SLO violations since it relies on the pattern similarity between the incoming workload and the past observed one. DeepBAT avoids this issue through its robust generalization capability, which allows it to adapt quickly and efficiently to dynamic workloads.

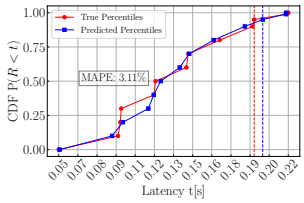
E. Latency Distribution Prediction

Fig. 13 depicts the Cumulative distribution functions (CDFs) of the predicted and observed latencies for the four traces. Here, we still use the model that is trained based on the first 12-hour historical data of Azure as the basic model. As shown in Fig. 13a, when tested on the Azure dataset itself where the statistics are all identical to the training data, the model has an average MAPE of 2.85%. The 95th percentile is highlighted using the vertical line, the predicted value is close to the ground truth. We directly use this model (trained with the Azure trace) to predict the latency distribution of the Twitter trace. Here the model is not re-trained or even fine-tuned. As plotted in Fig. 13b, the model still works well, with only 3.11% MAPE overall for all percentiles.

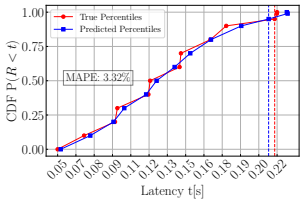
We directly use the model trained with Azure but with the more challenging Alibaba trace, this is an OOD case. Here,



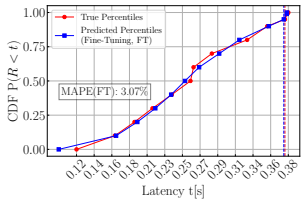
(a) Azure trace, 20-21th hour, Memory=2816MB, Max Batch Size=12, Timeout=100ms



(b) Twitter trace, 10-11th hour, Memory=2048MB, Max Batch Size=10, Timeout=50ms



(c) Alibaba trace, 4-5th hour, Memory=2048MB, Max Batch Size=10, Timeout=50ms

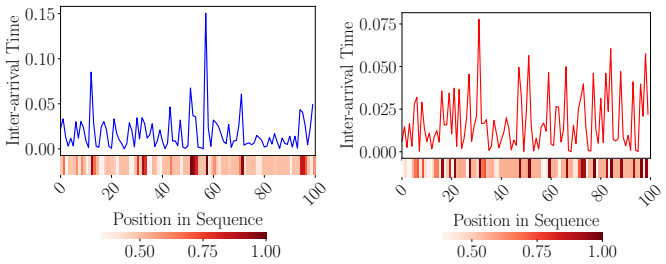


(d) Synthetic trace, 10-11th hour, Memory=2304MB, Max Batch Size=16, Timeout=100ms

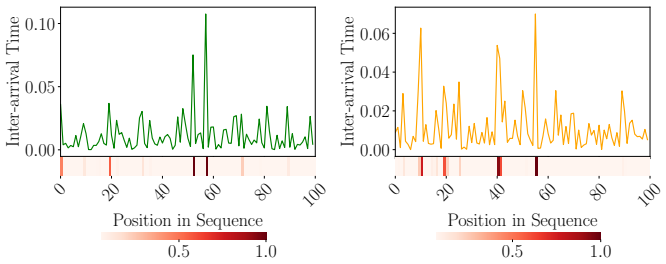
Fig. 13: Request latency distribution with arrivals driven from different traces. Vertical lines show the 95th percentile latency of the ground truth and that predicted by DeepBAT.

we use fine-tuning to improve the quality of the prediction and achieve a 3.32% MAPE, see Fig. 13c. Fig. 13d illustrates 3.07% MAPE with fine-tuning for the most challenging synthetic trace. These two traces clearly illustrate the ability of the deep surrogate model to learn patterns of the training data and successfully make predictions on unseen OOD data.

Attention Score Exploration. Fig. 14 visualizes the instances of aggregated normalized attention scores on these four datasets obtained by the model *only* trained on the Azure Dataset (without fine-tuning). Through the analysis of batches of results (more than 300 sequences for the Azure and Twitter traces and about 80 sequences for the Alibaba and synthetic traces due to the fewer arrival points), we conclude that the model automatically relates the predicted results to the specific parts of sequences that have longer inter-arrival period. This can be inferred from the correspondence of deeper color, which indicates a higher attention rate, and the peaks of



(a) Azure (b) Twitter



(c) Alibaba (d) MAP-generated

Fig. 14: Visualization of Attention scores.

the plotted sequences. The generalization ability of the deep surrogate model is also illustrated for the other three traces (see Fig. 14b, Fig. 14c, and Fig. 14d).

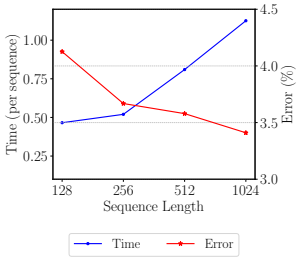
Observation #4. DeepBAT can accurately predict the latency distribution and can directly learn the characteristics of data patterns from the training data. It generalizes well to the unseen workloads with OOD data and prediction performance is further improved via fine-tuning.

F. Model Prediction Time: DeepBAT vs BATCH

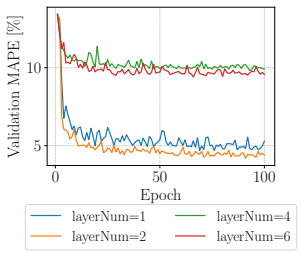
We use a Python built-in module *datetime* to record the prediction time of BATCH and DeepBAT. Experiments show that BATCH takes 40.83 seconds to return the optimal configuration, whereas DeepBAT only takes 0.73 seconds (broken down as milliseconds for identifying the configuration and the remaining time for the configuration cost optimization). This translates to a speedup of 55.93. Beyond the required time to make a prediction, BATCH still needs sufficient time to collect *enough* arrival points and do the fitting into a MAP, this process may be slow (if the arrival rate of requests is low) and error-prone (if the fitting into a MAP is not successful) [37]. Such features also make it infeasible to directly compare the overhead of DeepBAT and BATCH for per request because the later needs to perform its fitting process [54] (that is computationally expensive) every time it is triggered. DeepBAT eliminates the fitting process of BATCH and outputs the predicted value in a much shorter time compared to the analytical model that is the core of BATCH.

V. Sensitivity Analysis

Sensitivity Analysis on Sequence Length. Fig. 15a illustrates the change of computational time and prediction error



(a) Impacts of different sequence lengths.



(b) Impacts of different layer numbers of Transformer Encoders.

Fig. 15: Sensitivity Analysis.

with the sequence length. The sequence lengths are selected among {128, 256, 512, 1024}. A trade-off could be observed as sequence length increases. The predicted time per sequence increases sharply, representing higher computational costs, while the error rate decreases, indicating improved prediction accuracy. This demonstrates that longer sequences enable the model to capture more detailed patterns at the expense of increased computational time, which is as expected. For the experiments of DeepBAT shown in this paper, we choose a sequence length of 256 as a balanced point considering both time cost and accuracy.

Ablation Study on Transformer Encoder Layers Fig. 15b shows the effect of varying the number of Transformer encoder layers on validation MAPE. Four configurations are compared: 1, 2, 4, and 6 layers. The results reveal that compared with the case of a single layer, the model is smoothly trained in a stable way with 2 encoder layers, resulting in a small overall MAPE and it does not benefit from more layers (i.e., 4 and 6). Hence, we set the Transformer encoder layers of DeepBAT's deep surrogate model to 2.

VI. Related Work

Serverless is a popular paradigm for deep learning training [8], [55], [56] and inference serving [10], [18], [57]–[59]. Gao et al. [15] propose cellular batching to improve the throughput of RNN inference. Silfa et al. [16] propose E-BATCH for RNN inference. Clipper [14] uses batching to improve the performance of ML inference requests using exhaustive profiling, which is not practical if the workload is dynamic. Zhang et al. [17] propose MArk, a general-purpose ML inference serving system. While MArk can adjust its parameters to changing workloads, this adjustment is not timely for the case of bursty workloads. Similar to the work presented in this paper, Ali et al. propose BATCH [10] and a multiclass variation called MBS [18] that optimize the latency and cost based on SLO through batching. BATCH and MBS are based on an analytical framework that requires workload profiling and fitting into MAPs and numerical solutions of several matrix exponentials to reach the optimization parameters, a process that requires deep domain knowledge. In this paper, we offer an alternative methodology: how to use the deep learning-based model instead of classic performance models to achieve the same targets as BATCH and MBS. The more

recent work, INFless, proposed by Yang et al. [19] also aims at meeting the low latency and high throughput demands of ML services with an analytical framework.

The Transformer model [25] is widely used in the areas of NLP [60], time-series forecasting [61], and image processing [41]. The multi-head attention mechanism helps the model process the input sequence in a parallel way, and enhances the representative ability of the model by effectively capturing long-range dependencies. Such characteristics make it an effective module for tasks including text generating [62] and time-series forecasting [63]. In this paper, we leverage the power of Transformer encoder and the multi-head attention mechanism, to show that such deep learning-based models can be used as performance prediction models in systems that admit bursty workloads. We also utilize the deep surrogate model to predict latency distributions in addition to the SLO as the performance measure of interest.

VII. Conclusion

We introduce DeepBAT, an SLO-aware framework that leverages a deep surrogate model to optimize the performance of ML inference on serverless platforms. DeepBAT works well with bursty and previously unobserved workloads. The deep surrogate model that resides in the core of DeepBAT learns the data pattern directly from the arrival sequences and gives accurate predictions for the cost and latency of inference serving. Experiments show that DeepBAT can provide optimization parameters for inference serving in a serverless platform more accurately and 55x faster than analytical models.

Acknowledgment

This work is supported by the National Science Foundation (NSF) grants (#2402942) and the Commonwealth Cyber Initiative (CCI) grant (#HC-3Q24-047).

References

[1] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless big data processing using matrix multiplication as example," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 358–365.

[2] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[3] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 115–130.

[4] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa, "Serverless computing for internet of things: A systematic literature review," *Future Generation Computer Systems*, vol. 128, pp. 299–316, 2022.

[5] I. Wang, E. Liri, and K. Ramakrishnan, "Supporting iot applications with serverless edge clouds," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–4.

[6] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λdnn: Achieving predictable distributed dnn training with serverless architectures," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.

[7] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE, 2018, pp. 334–341.

[8] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 857–871.

[9] Z. Tu, M. Li, and J. Lin, "Pay-per-request deployment of neural network models using serverless architectures," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 2018, pp. 6–10.

[10] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM, 2020, p. 69.

[11] A. Mampage, S. Karunasekera, and R. Buyya, "A holistic view on resource management in serverless computing environments: Taxonomy and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[12] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 804–811.

[13] Amazon Web Services, "AWS Lambda: Run code without thinking about servers or clusters," https://web.archive.org/web/20230530084210/https://aws.amazon.com/lambda/, 2023.

[14] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[15] P. Gao, L. Yu, Y. Wu, and J. Li, "Low latency rnn inference with cellular batching," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[16] F. Silfa, J. M. Arnau, and A. González, "E-batch: Energy-efficient and high-throughput rnn batching," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–23, 2022.

[17] C. Zhang, M. Yu, W. Wang, and F. Yan, "{MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1049–1062.

[18] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing Inference Serving on Serverless Platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[19] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2022, pp. 768–781.

[20] W. Shi, S. Zhou, Z. Niu, M. Jiang, and L. Geng, "Multiuser co-inference with batch processing capable edge server," *IEEE Transactions on Wireless Communications*, vol. 22, no. 1, pp. 286–300, 2022.

[21] A. Riska and E. Smirni, "M/g/1-type markov processes: A tutorial," in *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, ser. Lecture Notes in Computer Science, M. Calzarossa and S. Tucci, Eds., vol. 2459. Springer, 2002, pp. 36–63. [Online]. Available: https://doi.org/10.1007/3-540-45798-4_3

[22] A. Riska and E. Smirni, "Mamsolver: A matrix analytic methods tool," in *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, ser. Lecture Notes in Computer Science, T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, Eds., vol. 2324. Springer, 2002, pp. 205–211. [Online]. Available: https://doi.org/10.1007/3-540-46029-2_14

[23] K. Kojs, "A survey of serverless machine learning model inference," *arXiv preprint arXiv:2311.13587*, 2023.

[24] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons, "Tributary: spot-dancing for elastic services with latency {SLOs}," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 1–14.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX annual technical conference (USENIX ATC 20)*, 2020, pp. 205–218.

[28] Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," *arXiv preprint arXiv:1703.03130*, 2017.

[29] S. Tuli, G. Casale, L. Cherkasova, and N. R. Jennings, "Deepft: Fault-tolerant edge computing using a self-supervised deep surrogate model," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[30] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency," in *Proceedings of the International Conference on Parallel Processing (ICPP)*. ACM, 2021, pp. 14:1–14:12.

[31] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient Serverless Inference through Tensor Sharing," in *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 2022.

[32] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the Planet of Serverless Computing: A Systematic Review," *ACM Transactions on Software Engineering Methodology*, 2023, [Early Access].

[33] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[34] M. Kolny, "Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%," https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90, 2023.

[35] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.

[36] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[37] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, "How to parameterize models with bursty workloads," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, pp. 38–44, 2008.

[38] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th International Conference on Autonomic Computing, ICAC 2009, June 15-19, 2009, Barcelona, Spain*, S. A. Dobson, J. Strassner, M. Parashar, and O. Shehory, Eds. ACM, 2009, pp. 149–158. [Online]. Available: https://doi.org/10.1145/1555228.1555267

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

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[41] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[42] ArchiveTeam, "JSON Download of Twitter Stream 2017-05," https://archive.org/details/archiveteam-twitter-stream-2017-05, 2017.

[43] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters," in *19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22)*, 2022.

[44] F. Hernandez, V. Nguyen, S. Ghannay, N. A. Tomashenko, and Y. Estève, "TED-LIUM 3: Twice as Much Data and Corpus Repartition for Experiments on Speaker Adaptation," in *Proceedings of the International Conference on Speech and Computer (SPECOM)*, ser. Lecture Notes in Computer Science, vol. 11096. Springer, 2018, pp. 198–208.

[45] F. Yan, Y. He, O. Ruwase, and E. Smirni, "SERF: efficient scheduling for fast deep neural network serving via judicious parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2016, pp. 300–311.

[46] F. Yan, Y. He, O. Ruwase, and E. Smirni, "Efficient deep neural network serving: Fast and furious," *IEEE Trans. Netw. Serv. Manag.*, vol. 15, no. 1, pp. 112–126, 2018. [Online]. Available: https://doi.org/10.1109/TNSM.2018.2808352

[47] J. Xue, R. Birke, L. Y. Chen, and E. Smirni, "Managing data center tickets: Prediction and active sizing," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 335–346.

[48] W. Miao, G. Min, X. Zhang, Z. Zhao, and J. Hu, "Performance modelling and quantitative analysis of vehicular edge computing with bursty task arrivals," *IEEE Transactions on Mobile Computing*, 2021.

[49] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: state-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[50] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Proceedings*, ser. Lecture Notes in Computer Science, V. Issarny and R. E. Schantz, Eds., vol. 5346. Springer, 2008, pp. 265–286. [Online]. Available: https://doi.org/10.1007/978-3-540-89856-6_14

[51] G. Casale, N. Mi, and E. Smirni, "Bound analysis of closed queueing networks with workload burstiness," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 13–24, 2008.

[52] G. Casale, N. Mi, and E. Smirni, "Model-driven system capacity planning under workload burstiness," *IEEE Trans. Computers*, vol. 59, no. 1, pp. 66–80, 2010. [Online]. Available: https://doi.org/10.1109/TC.2009.135

[53] R. Gusella, "Characterizing the Variability of Arrival Processes with Indexes of Dispersion," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 2, pp. 203–211, 1991.

[54] G. Casale, E. Z. Zhang, and E. Smirni, "Kpc-toolbox: Best recipes for automatic trace fitting using markovian arrival processes," *Perform. Evaluation*, vol. 67, no. 9, pp. 873–896, 2010. [Online]. Available: https://doi.org/10.1016/j.peva.2009.12.003

[55] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, 2018, pp. 2–8.

[56] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[57] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International conference on cloud engineering (IC2E)*. IEEE, 2018, pp. 257–262.

[58] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[59] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.

[60] L. Dong, S. Xu, and B. Xu, "Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition," in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2018, pp. 5884–5888.

[61] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," *International Journal of Forecasting*, vol. 37, no. 4, pp. 1748–1764, 2021.

[62] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[63] Z. Wu, S. Pan, G. Long, J. Jiang, X. Chang, and C. Zhang, "Connecting the dots: Multivariate time series forecasting with graph neural networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 753–763.