EcoEdgeInfer: Dynamically Optimizing Latency and Sustainability for Inference on Edge Devices

Sri Pramodh Rachuri
PACE Lab
Dept. of Computer Science
Stony Brook University
srachuri@cs.stonybrook.edu

Nazeer Shaik
PACE Lab
Dept. of Computer Science
Stony Brook University
nshaik@cs.stonybrook.edu

Mehul Choksi
PACE Lab
Dept. of Computer Science
Stony Brook University
mchoksi@cs.stonybrook.edu

Anshul Gandhi
PACE Lab
Dept. of Computer Science
Stony Brook University
anshul@cs.stonybrook.edu

***Abstract*—The use of Deep Neural Networks (DNNs) has skyrocketed in recent years. While its applications have brought many benefits and use cases, they also have a significant environmental impact due to the high energy consumption of DNN execution. It has already been acknowledged in the literature that training DNNs is computationally expensive and requires large amounts of energy. However, the energy consumption of DNN inference is still an area that has not received much attention, yet. With the increasing adoption of online tools, the usage of inference has significantly grown and will likely continue to grow. Unlike training, inference is user-facing, requires low latency, and is used more frequently. As such, edge devices are being considered for DNN inference due to their low latency and privacy benefits. In this context, inference on edge is a timely area that requires closer attention to regulate its energy consumption.**

We present EcoEdgeInfer, a system that balances performance and sustainability for DNN inference on edge devices. Our core component of EcoEdgeInfer is an adaptive optimization algorithm, EcoGD, that strategically and quickly sweeps through the hardware and software configuration space to find the jointly optimal configuration that can minimize energy consumption and latency. EcoGD is agile by design, and adapts the configuration parameters in response to time-varying and unpredictable inference workload. We evaluate EcoEdgeInfer on different DNN models using real-world traces and show that EcoGD consistently outperforms existing baselines, lowering energy consumption by 31% and reducing tail latency by 14%, on average.

***Index Terms*—inference, energy, latency, workload changes**

I. Introduction

The adoption of Deep Neural Networks (DNNs) has skyrocketed in recent months and years with applications in image recognition, speech recognition, and natural language processing [1]. While these applications have brought many benefits to society in general, the underlying compute requirements of executing the DNNs have had a severe environmental impact, leading to *significantly high energy consumption and carbon emissions* [2]. This negative impact of training large and expensive DNNs has been highlighted in recent works, and has already resulted in research efforts to regulate the monetary and environmental cost of DNN training, for example, by leveraging hardware techniques [3], [4].

However, the sustainability costs of ***DNN inference*** have not received as much attention, yet. This is likely because DNN inference has much lower computational needs (usually completes in seconds or minutes, compared to hours or days for training). Unlike training, inference is usually a one-shot operation, where the input data is fed to the DNN and the output is generated without any further adjustments to the model or need for complex backpropagation operations; this makes inference a lightweight and short-lived task. Unfortunately, with the increasing popularity of online tools like ChatGPT [5] and GitHub Copilot [6], the usage of inference has significantly grown, and will likely continue to grow. Consequently, even though it has lower computational demand, inference is executed much more frequently than training [7], and can thus have a substantial environmental impact. *Addressing the high energy costs of the quickly scaling inference applications is thus a crucial and timely problem.*

DNN inference, unlike training, is ***user-facing***, and thus requires low (mean and tail) latency. The inference latency experienced by an end-user can be broadly categorized into network latency (to and from the cloud service) and computation latency. By *deploying DNN inference on edge devices*, the network latency can be significantly reduced as the input and output data does not have to be transferred to/from the cloud. This is especially useful for real-time and privacy-conscious applications like autonomous vehicles, augmented reality, and smart cities. Depending on the application, edge devices may be deployed in remote locations with limited access to power sources, and some might even be solar- or battery-powered [8]. As such, minimizing the energy consumption of DNN inference is important in the edge computing setting.

Addressing the ***energy costs of DNN inference on the edge*** is a challenging problem for several reasons:

(1) The energy consumption of inference on edge depends on both ***hardware and software configurations***. While there are tuning knobs that can influence energy consumption, such as GPU operating frequency and batch size, their impact on energy may be inter-related, and thus they cannot be optimized independently.

(2) There is an ***inherent tension between reducing energy consumption and improving inference latency***, resulting in a non-trivial optimization to balance the energy-latency tradeoff. For example, setting the GPU frequency to the maximum allowable value can reduce inference latency but at the expense of high energy.

(3) Although the effect of batch size on energy has been

studied, it is ***not always clear how batch size affects inference latency***, especially under dynamic workload.

(4) Inference applications are user-driven, and thus, the ***request load can change dynamically and unpredictably***, requiring an agile solution that can adapt to changing conditions quickly and with low overhead to prevent high inference tail latencies.

There have been several recent works that focused on reducing the energy consumption of DNN training. However, these solutions are not directly applicable to DNN inference as the requirements (e.g., memory, compute capacity, and tail latency) and usage patterns (e.g., request rate and burstiness) for training and inference are different. Moreover, most of the prior works on DNN inference have focused on server and cloud environments [9], and have not considered the unique challenges of running inference on edge devices, such as limited power/energy and low-overhead requirements. Finally, while there have been works that analyze the impact of hardware and software parameters on energy consumption of DNN execution [10], [11], they are offline studies that neither consider dynamic workload conditions nor focus on designing an adaptive solution for practical deployments. As such, there is a *gap in research on adaptive solutions for energy-efficient DNN inference on the edge*. We discuss related works in detail in Section III.

Our contributions: In this work, we first present *EcoEdgeInfer*, a *framework for DNN inference serving on edge devices that regulates their energy consumption and latency at runtime*. EcoEdgeInfer works by dynamically and transparently tuning hardware and software parameters based on recommendations from optimization algorithms. We have *implemented EcoEdgeInfer on NVIDIA Jetson devices* using Python, ensuring seamless integration with existing DNN inference libraries like PyTorch. The code for EcoEdgeInfer is publicly available as a GitHub repository to promote further research and facilitate reproducibility.[1]

Our key contribution is an *optimization algorithm, EcoGD, specifically designed for sustainable inference on edge deployments*. EcoGD is inspired by Gradient Descent, and enhances its functionality via real-time adaptation to changing workloads, estimating the cost of unexplored configurations and gradients, and maintaining low overhead. To prevent fluctuations in performance and maintain low tail inference latency, EcoGD limits its exploration space; to react to evolving workload conditions quickly, EcoGD maintains a limited memory of explored configurations.

We *experimentally evaluate EcoGD against existing solutions with different DNN inference models*. Our results show that EcoGD converges to superior configurations under fixed and bursty synthetic load patterns, reducing energy consumption by 22% and lowering mean inference latency by 13%, on average, compared to existing solutions. Further, EcoGD's achieved energy and latency values are within 2% and 10%, respectively, of those achieved by an (impractical) offline optimal solution. Importantly, EcoGD effectively adapts to changing load conditions, consistently outperforming other solutions by lowering energy consumption by as much as 66% (with 31% average reduction) and reducing tail latency by as much as 91% (with 14% average reduction) under real-world workload traces. To the best of our knowledge, *ours is the first work that jointly optimizes energy consumption and latency of DNN inference on edge devices at runtime while adapting to changing workload conditions*.

TABLE I
TECHNICAL SPECIFICATIONS OF NVIDIA XAVIER NX

Specification	Value
CPU	6-core Nvidia Carmel
CPU Freq. range	115 MHz – 1.9 GHz; 25 steps of 77 MHz
GPU	NVIDIA Volta
GPU Cores	384 CUDA Cores + 48 Tensor Cores
GPU Freq. range	114 MHz – 1.1 GHz; 15 steps of 90 MHz
Memory	8 GB LPDDR4x
Throughput	21 TOPs
Default Power Modes	10W, 15W, 20W
Jetpack version	5.1.3 [L4T v35.5.0]
Framework	PyTorch 2.1.0
Operating Sys. & Libraries	Ubuntu 20.04.6; CUDA 11.4 + cuDNN 8.6

II. BACKGROUND

In this section, we provide the necessary background on the problem of energy-efficient DNN inference on edge devices. We first discuss the types of edge devices we consider in this paper and the control knobs they offer to regulate energy consumption. We then discuss workload parameters that can also be adjusted to regulate energy consumption. Finally, we discuss the significance of the request arrival pattern on our problem and how it can impact energy consumption.

A. Smart Edge Devices for Deep Learning

Edge computing is the practice of processing data near the source of data generation rather than relying on centralized data centers. While a range of devices, from IoT sensors to on-premise workstations, can be considered edge devices, we focus on edge devices with limited computational resources, such as single-board computers, called *smart edge devices* [12]. These devices are typically equipped with low-power processors, small amounts of memory, and are energy-limited.

With the launch of NVIDIA's Jetson series [13], running Deep Learning (DL) models on edge devices has become feasible due to the integrated GPUs with CUDA support. For example, NVIDIA Xavier NX has 8 ARM cores, 384 CUDA cores, 48 Tensor cores, and 8GB of LPDDR4x memory; it consumes at most 20W of power. (More details about the Xavier NX device, which we also use in our evaluation, are provided in Table I for reference.) However, the limited computational resources on these devices also make it challenging to run large models in an efficient manner.

B. Hardware parameters on edge devices for managing energy

To regulate energy consumption, NVIDIA provides three power modes (10, 15, 20 Watts) as default presets to change

[1] https://github.com/PACELab/EcoEdgeInfer

hardware parameters of the Xavier NX device. However, we can also manually change hardware parameters for greater and finer control. While running DL inference, we noticed a significant impact on energy consumption by varying the GPU frequency, a moderate impact by varying the CPU frequency, and almost no impact when varying the number of CPU cores. This is because the GPU is the primary resource used by DL inference. Further, Python being single-threaded, the CPU is not used for parallelism; hence, the number of cores does not significantly impact energy or performance. So, we only focus on GPU and CPU frequencies in our experiments.

C. *DNN Inference Parameters*

In addition to hardware parameters, there are also software or workload parameters that can be adjusted to regulate energy consumption. When a user submits an inference request, the DNN model and their weights are fixed by the user. However, the inference system or framework can still adjust knobs like the batch size and how the model's task graph is loaded (statically or dynamically). Static loading using Pytorch's JIT Trace does not have a significant impact on energy consumption in the long run when compared to dynamic loading because it only shifts the model initialization and loading [14]. So, we ignore the model loading mechanism and focus on the batch size as a potential tunable parameter.

Da *et al.* [15] showed that increasing the batch size always leads to increased throughput (not necessarily latency) due to higher parallelism and utilization of the GPU. They also showed that increasing the batch size can lead to energy savings, but these savings taper off after a certain point. The maximum batch size a model can handle is limited by the device's memory. Given the limited memory on edge devices, the batch size is kept low to prevent out-of-memory errors.

D. *Request Arrival Pattern*

Another factor that impacts DNN inference performance, especially on the edge, is the arrival pattern of requests. This is also a factor that has ***not been thoroughly investigated by prior works*** that focus on regulating DNN inference energy. Note that the arrival pattern of requests does not just refer to the arrival rate, but can also include the distribution of requests and factors such as burstiness of arrivals.

Consider an inference serving system. When the rate of inference requests coming in is low, the system can adjust its parameters (such as CPU and GPU frequency) to strike a balance between inference latency and energy consumption. However, when the request rate is high, the system may have to tune its parameters to maximize performance to handle the high load. In fact, for resource-limited edge devices, it is possible that a long queue starts building up quickly at high request rate as the serving capability of the device may be limited, thereby constraining the system to exclusively pick the highest performance configurations. As such, the optimal system configuration and the potential for tuning the configuration parameters are affected by the request rate.

The request arrival pattern also indirectly affects the system performance through the batch size. While batch size can help increase the throughput and energy efficiency [15], it also leads to increased tail latency due to the time taken to accumulate the batch before processing [16]. For example, if we have a constant stream of requests with an inter-arrival time (IAT) of 10ms, it will take $\sim$100ms to accumulate a batch size of 10. But, if the IAT is 100ms, it will take about 1s to get to the same batch size. So, a lower batch size may be preferable for a high IAT workload to avoid large accumulation delays and long tail latencies. However, if the arrival pattern is bursty, the above conclusions can change; a burst of 10 (or more) uninterrupted requests can immediately lead to an accumulation of the required batch size, with additional requests now waiting to be served. Worse, the request arrival pattern can change unpredictably, especially for user-facing workloads [17], necessitating an agile solution that adapts to arrival pattern changes at runtime.

Finally, the content of requests can also impact inference latency and energy. For example, images with different blurring and brightness may have different processing times and energy consumption for image recognition. For this paper, we do not consider such content-based workload changes, and limit our focus to changes in arrival rate and request distribution.

III. Related Work

Optimizing the energy or carbon usage of Deep Learning workloads has been an important topic of research in the past few years. However, most of the work has focused on model training on server-grade GPUs and machines. There are very few works that focus on inference workloads on edge devices; we discuss these at the end of this section.

A. *Energy Efficient DNN Training*

Google's CICS [18] focuses on minimizing the carbon footprint of data centers by temporally delaying flexible workloads to greener times. Zeus [3] optimizes the energy efficiency of DNN training by finding optimal job and GPU-level configurations. PowerFlow [4] is a GPU cluster scheduler that reduces the average job completion time under an energy budget. Cloud providers have also started offering Machine-Learning-as-a-Service (MLaaS) platforms that run on heterogeneous GPU clusters and automatically optimize the hardware, software, cluster configuration, and hyperparameters for training DNNs to reduce the energy consumption and improve the performance of the training jobs [19].

Works like PowerFlow [4] and Zeus [3] change the frequency of server-grade GPUs using power-limit APIs and thereby reduce energy consumption. Since we are focusing on a single device optimization, Zeus is the closest work to ours. Zeus uses Multi-Armed Bandit (MAB) to find the optimal batch size and performs exhaustive search to find the optimal GPU power limit for server-grade GPUs. Zeus can afford to run an exhaustive search on GPU power limit settings since it does not consider a change in workload (i.e., only considers a static training setting). However, in our inference case, due to the varying workload patterns, an exhaustive search is not feasible, as we show in Section VI. The MAB approach is feasible, and we compare against MAB (as a baseline) in our evaluation in Section VI.

Prashanthi et al. [10] is one of the first works focusing on changing CPU and GPU frequencies for DNN workloads on edge devices. The authors study the impact of power modes (presets of frequency limits by NVIDIA) on CPU and GPU utilization, training time, and energy usage. However, the work focuses on DNN training. Instead, our work here focuses on DNN inference, which is better suited for resource-constrained edge devices than (computationally heavy) DNN training.

The observations made for training workloads do not readily apply to inference workloads. For example, Zeus [3] does not consider their system to have a queue of jobs that need to be scheduled because training workloads are usually long running and are not sent by users in a queue. Inference workloads, on the other hand, take a lot less time to complete but are more frequent and arrive dynamically. Hence, we must consider the real-time queueing effects on latency when optimizing for user-facing inference workloads.

B. *Energy Efficient DNN Inference on Edge*

JEDI [20] focuses on edge devices and considers inference workloads. The authors create a TensorRT-based pipelining framework to increase resource utilization and performance. They consider parameters like multi-threading, multi-device (CPU, GPU, DLA) processing, and buffer assignment. While JEDI results in lower energy consumption, this is not the primary objective and is a by-product of their optimization. As such, they did not attempt to tune energy-saving knobs, like CPU and GPU frequencies, which we believe are powerful and lightweight and lightweight tools to optimize energy consumption.

Dutt *et al.* [11] focus on the inference side of DNN workloads and profile the effects of processor frequency tuning on energy consumption. However, they do not consider workload parameters and do not consider latency or performance as an objective. Further, only a static offline workload is considered for evaluation. By contrast, we specifically focus on the tradeoff between energy consumption and inference latency, and consider dynamic changes in inference workload.

Based on our literature review above, we believe there are very ***few prior works on jointly optimizing the energy consumption and latency of DNN inference workloads on edge devices by tuning both hardware and software parameters***. Further, we are unaware of existing research that considers the ***impact of request arrival pattern on energy consumption and inference latency for edge devices***.

IV. System Design

In this section, we describe the system design of our solution, EcoEdgeInfer. We primarily break down the features of EcoEdgeInfer into: *(1) Core Inference System*, *(2) Metrics Collection*, *(3) Optimizer*, and *(4) Applying the optimal configuration*. Figure 1 illustrates the system design of EcoEdgeInfer and the interactions between these components.

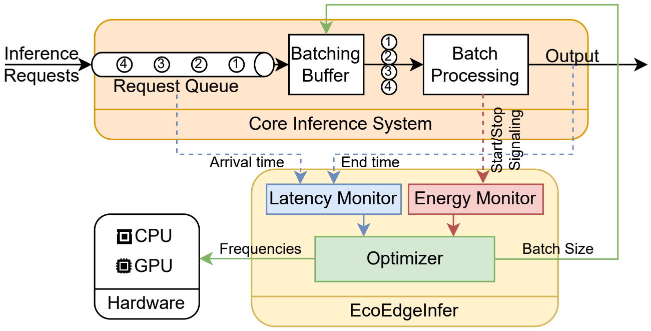

Fig. 1. Illustration of EcoEdgeInfer's system design.

A. *Core Inference System*

The core inference system is the component of EcoEdgeInfer that actually performs the DNN inference and batching, and is the only component that interacts with the developer's code. It also allows us to collect the metrics that are used by the optimizer.

Every time the inference method is called, the core inference system receives the input data and adds it to the request queue. On a separate thread, it waits for the request queue to have enough requests to form a batch (in the batching buffer) and then performs the inference on the batch. The batch size is determined by the optimizer and can be changed dynamically. The request queue, the batching buffer, and the tunable batch size together allow the system to process requests with varying arrival rates by staging the incoming requests; see Figure 1. The core inference system sends signals to the monitoring system to start collecting metrics while processing each batch.

B. *Metrics Collection*

The monitoring system is responsible for collecting (at least) the latency and energy consumption metrics based on the signals received from the core inference system. The latency monitor uses the arrival timestamp of each request in the request queue and the completion timestamp of the request to calculate the request's latency. The energy monitor starts and stops logging the power sensors based on the signals received from the core inference system. We log the power sensors every 10 ms over the i2c interface and the sysfs interface provided by NVIDIA's API [21]. The latency and energy metrics collected for a single batch may be too small and can lead to noisy values. So, the monitoring system stores the metrics in a temporary cache till enough samples are collected. In our experiments, we found that monitoring for $\sim$400 requests is enough to remove the noise and get a good estimate of the metrics. This duration (of 400 requests) is referred to as an *Optimizer Step* in the rest of the paper.

C. *Optimizer and optimization policies*

The optimizer is the core component of EcoEdgeInfer that determines the optimal parameter values to be used. It uses the metrics collected in the previous optimizer step to calculate a cost function. In our paper, we consider the cost function to be the mean of homogenized latency and homogenized energy consumption; we aim to minimize this cost function. In general, any combination of these two metrics could be employed as needed. To homogenize the latency and energy metrics, we divide the observed latency and energy values in each experiment by those obtained when all parameters are set to their maximum value. Similar approaches have been used in prior work to balance multiple metrics in the objective function (latency and energy, in our case) [3], [22].

The calculated cost is then used by the optimizer to predict the optimal CPU Frequency, GPU Frequency, and Batch Size that can be used for the next optimizer step. In addition to our algorithm, EcoGD (described in the next subsection), we have implemented five other optimization policies inspired by prior works as baselines, including Grid Search, Multi-Armed Bandit, and Bayesian Optimization. These baselines are detailed in Section V-C.

D. Predicting the optimal configuration via EcoGD

We design a specific optimization algorithm, EcoGD, for the problem of optimizing the parameters of inference on edge. Our EcoGD algorithm is inspired by the gradient descent algorithm. However, unlike standard gradient descent problems, in the inference on edge setting, we do not have access to the gradient of the cost function nor do we readily have all the data points needed to calculate it. As such, the traditional gradient descent algorithms cannot be used as-is; similar observations have been made by prior works addressing different problems as well, such as the problem of tuning the parameters of transactional memory [23]. EcoGD can be considered a variant of the broad family of local-search heuristic algorithms, which includes algorithms like hill climbing and simulated annealing, with Tabu search being the closest algorithm to EcoGD [24].

Algorithm 1 shows the pseudo code for EcoGD along with the input, output, and parameters of the optimizer (lines 1–4). At every optimizer step, EcoGD either explores the neighboring configurations (lines 7–9) or, if the neighborhood has been sufficiently explored, jumps to a new configuration (lines 11–20).

Every configuration in the 3-dimensional space of CPU Frequency, GPU Frequency, and Batch Size has 26 neighbors: 6 directly adjacent (i.e., change in only one dimension), 4 diagonally adjacent in the CPU-GPU plane (i.e., change in both frequency dimensions), and 16 other diagonally adjacent configurations (i.e., change in batch size and at least one frequency dimension). However, EcoGD only evaluates the cost of the 6 directly adjacent neighbors of the current configuration for exploration (line 6). This reduces the number of configuration evaluations from 26 to just 6, thereby significantly reducing the exploration time. The 4 configurations that are diagonally adjacent in the CPU-GPU plane are instead estimated (line 10) using the following equation:

$$Cost(i+x, j+y, k) = \frac{Cost(i, j+y, k) + Cost(i+x, j, k)}{2} \qquad (1)$$

where $Cost(i, j, k)$ is the cost of the configuration with CPU Frequency i, GPU Frequency j, and Batch Size k. Note that $x, y \in \{-1, 1\}$ indicates the diagonal neighbors of the current configuration in the CPU-GPU plane.

Algorithm 1 Pseudo code for EcoGD

```
 1: Input: History matrix $H$ containing the cost of all explored configurations.
 2: Output: Next configuration to run $C = (CPU, GPU, Batchsize)$
 3: Parameters: Memory size $Mem_{size}$, Maximum loops $Max\_Loops$, Starting Config $C_{start}$
 4: Initialize: $loop\_counter \leftarrow 0$, $last\_center \leftarrow C_{start}$

    Function $EcoGD(H)$:
 5: $H = copy(H, Mem_{size})$
                              ▷ Trim history to last $Mem_{size}$ configurations
 6: $unexplored\_nbhr$
        $\leftarrow \{nbhr \in 6\text{-}nbhrhd(last\_center)$ s.t. $nbhr \notin H\}$
                              ▷ Finding unexplored neighbors in 6-neighborhood
                              ▷ neighbour, neighborhood abbreviated as nbhr, nbhrhd
 7: if $Len(unexplored\_nbhrs) > 0$ then
 8:     $C \leftarrow unexplored\_nbhrs[0]$
 9:     return $C$            ▷ Return the first unexplored nbhr if any
10: $nbhrhd\_history$
        $\leftarrow explored\_nbhrs\_history(last\_center, H)$
        $+ unexplored\_nbhrs\_history(last\_center, H)$
                              ▷ History of unexplored neighbors are estimated
11: $best\_config \leftarrow \text{argmin}(nbhrhd\_history)$
12: if $best\_config \neq last\_center$ then
13:     $loop\_counter \leftarrow 0$
14: else
15:     $loop\_counter \leftarrow loop\_counter + 1$
16:     if $loop\_counter > Max\_Loops$ then
17:         $best\_config \leftarrow random\_nbhr(last\_center)$
                              ▷ Jump to a random neighbor if stuck
18:         $loop\_counter \leftarrow 0$
19: $last\_center \leftarrow best\_config$
20: return $best\_config$
```

The remaining 16 diagonally adjacent configurations are not considered for estimation because we could not find a reliable and accurate approach to estimate the cost of such configurations. With this approach of selective exploration and estimation, EcoGD is able to converge to the optimal configuration quickly while following a path with reasonable costs. This is a key feature of EcoGD that distinguishes it from Tabu search and other local-search heuristic algorithms.

If the neighborhood is already explored, EcoGD jumps to the neighbor that has the lowest cost and the process is repeated (line 11). We do not let EcoGD jump farther than the neighboring configurations to avoid the optimizer from moving to a vastly different configuration. In practice, we found that jumping farther than the neighboring configurations does not lead to a significant improvement in the cost function and can cause repeated overshooting over the optimal configuration. Note that a learning rate like mechanism cannot be used and tuned because the configuration space is discrete and only the immediate neighbors are possible due to quantization.

To avoid re-exploring the same configurations repeatedly, EcoGD maintains a list of recently explored configurations and the corresponding costs. We limit the number of configurations remembered to a fixed number to allow adaptation to changing workload conditions (lines 12–18). And to avoid getting stuck in a local minima, EcoGD jumps to a random neighbor when the same configuration is predicted more than a fixed number of times and all the neighbors have been explored already (line 17).

EcoGD maintains only a few configurations and costs in its history to minimize physical memory overheads and also to quickly adapt to changing workloads by discarding stale configurations (line 5). The details of the hyperparameters of EcoGD are discussed in Section V-E.

E. Applying the optimal configuration

After the optimizer predicts the optimal configuration, it sets the optimal CPU frequency using the cpufreq driver and the optimal GPU frequency using NVIDIA's API [21]. In parallel, the optimizer sends the optimal batch size value to the core inference system, to be used for the subsequent inferences.

V. Experimental Setup and Methodology

In this section, we describe the experimental setup, workload traces, baseline algorithms, and the methodology we employ for our experimental evaluation (presented in Section VI).

A. Experimental setup

We perform all evaluations on an NVIDIA Jetson Xavier NX device; see section II-A for the technical specifications of the device. We use PyTorch v2.1.0, CUDA v11.4, and cuDNN v8.6 for our experiments.

We primarily use Resnet50 from the torchvision library [25] to extend our evaluation to a different model; for this model, we use an image of size 224×224 as the input request. We also use a transformer-based model, BERT-Tiny [26], [27], to extend our evaluation; for this model, we use a random text of size ∼1KB as the input request.

B. Request arrival patterns and traces

A key contribution of our evaluation is considering the impact of the request arrival pattern. As such, we perform our evaluations under three different inference request arrival patterns. First, we consider fixed inter-arrival times (time between successive requests); these conditions represent a controlled workload setting. We choose inter-arrival times of 50ms and 90ms, representing high and low load, respectively; these values were also chosen to ensure a stable system, as the minimum inference service time is about 40ms for Resnet50. Second, we consider bursty arrivals, representing a more extreme workload setting. Specifically, requests arrive in bursts of 10 requests every 500ms or every 900ms, thus still maintaining an average inter-arrival time of 50ms and 90ms, respectively, but with a more bursty arrival pattern.

Finally, we also evaluate using snippets from three real-world time-varying load traces. Publicly accessible inference request traces are not widely available. As such, we consider the following traces in our evaluation.

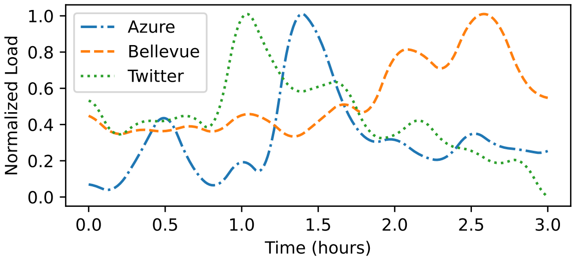

Fig. 2. Timeseries of normalized load for the traces used in our evaluation.

1) **Bellevue trace:** Number plate reading is a popular inference task at edge devices and smart cameras, and is used in applications such as traffic monitoring and parking management. The number of inference requests arriving at such edge devices is proportional to the number of vehicles passing through the camera at any given time [28]. To generate realistic traffic for this inference task, we use the Bellevue Traffic Video Dataset [29], which contains video data of vehicular traffic in Bellevue, Washington. We extract the number of vehicles per second from the dataset and use this as a proxy for the incoming rate of inference requests.
2) **Twitter trace:** Social media content moderation is another popular inference application to perform large-scale tasks such as sentiment analysis and fake news detection. In this application, each request for inference corresponds to a tweet that needs to be analyzed. To simulate the load for such an application, we use Twitter's sampled stream of tweets [30], [31]. We select a 3-hour period surrounding midnight on December 31, 2020, and January 1, 2021, from the East Coast of the United States to create a localized traffic trace. The number of tweets per second serves as a proxy for the inference request rate.
3) **Azure trace:** Owing to the increasing popularity of private edge data centers, well-known cloud paradigms such as serverless inference [32], [33] are expected to be employed at the edge in the near future [34]. We use the Azure Functions Traces [35] to represent the load on such an edge system. The rate of serverless invocations is used as a proxy for the arriving request rate at the edge device.

In all traces, we select 3 hours of data that exhibit diverse traffic patterns. Figure 2 shows how the load varies over time in the three traces; the load is normalized to the maximum for each trace for ease of comparison. We see that the Azure trace has a small surge at the start and then a more significant surge to the maximum load, and then a drop back to moderate load. The Bellevue trace has a more stable load with a gradual increase towards the end and then a sudden drop. In the Twitter trace, we see a sharp surge to maximum load and then a gradual drop to a very low load. The trace data is scaled to our device capacity and is used to create a timeseries of request rates arriving at our edge device. In terms of average load (or request rate) post-scaling, the Bellevue trace has the highest load, followed by the Twitter and the Azure traces.

C. Comparison baselines

In our experimental evaluation, we compare EcoGD with five other baseline algorithms.

1) **Grid Search** is a simple optimization policy that exhaustively tries all possible combinations of CPU Frequency, GPU Frequency, and Batch Size, and then selects the one with the lowest cost. Grid Search is easy to implement and should result in the optimal configuration but it requires significant running time to sweep over all possible combinations of parameters. For example, on the NVIDIA Xavier NX, the search space is $25 \times 15 \times 16 = 6,000$ combinations. While Grid Search may be impractical for real inference systems (requiring a full Grid Search to be re-performed every time the workload conditions change), we use this policy as a baseline in our evaluation.

2) **Linear Search** starts with a default configuration sweep through one dimension at a time (from among batch size, CPU frequency, and GPU frequency) and selects the configuration in that dimension that gives the lowest cost. It then moves to the remaining dimensions iteratively to optimize their configuration, while fixing the optimal configuration for the prior dimensions. Once the search is complete, a configuration which is locally optimal in each dimension is reported. The policy is much faster than Grid Search as it only searches through $25+15+16 = 56$ configurations. However, like Grid Search, Linear must be rerun every time the workload changes.

3) **DVFS** is the default policy (enabled by default) used in servers and edge devices to dynamically set the CPU and GPU frequency depending on the incoming load. The Xavier NX device ships with many governor policies for automatic DVFS control for CPU and GPU. Among them, schedutil and nvhost_podgov are the default governors for CPU and GPU, respectively. We also experimented with other governors, but there were no remarkable differences in performance. As such, we report DVFS results under their default CPU and GPU governors. DVFS, in theory, automatically adapts to changing workloads and is thus an adaptive policy. For example, schedutil constantly monitors the CPU utilization and adjusts the CPU frequency accordingly [36]. Note that DVFS does not manage the batch size, and so we use a reasonably large (for edge devices) batch size of 16 and a moderate batch size of 8 in our experiments.

4) **Bayesian Optimization** is an optimization technique that has been often used in systems literature [37]–[39] to optimize black-box functions. It uses a probabilistic model to predict the cost function and then uses an acquisition function to decide the next configuration to try. We use the GaussianProcessRegressor and RadialBasisFunction kernel from the scikit-learn library to implement Bayesian Optimization [40].

5) **Multi-Armed Bandit (MAB)** is another popular technique used in systems literature to find optimal configurations for systems [3], [41]. We chose the most popular variant of MAB, the epsilon-greedy algorithm, for our evaluation. At every step, the MAB algorithm probabilistically selects either exploration or exploitation. When exploration is selected, the algorithm tries new random configurations, and when exploitation is selected, the algorithm selects the configuration that has given the lowest cost so far. When the same configuration is selected multiple times, we choose to update the cost history using exponential moving average as it is the most common technique used in the literature when conditions are changing [42].

We informally refer to Bayesian Optimization, MAB, and EcoGD as *learning-based algorithms* since they learn from the cost of the configurations they have tried so far and use this information to decide on the next configuration to try.

D. Evaluation methodology

For our evaluation, we consider the energy consumption, inference latency, and cost as the primary metrics (see Section IV-C for our cost definition); we consider mean and tail values for these metrics in different evaluation settings. When running Grid Search and Linear Search, we run the experiments till the end of the search space. DVFS, Bayesian Optimization, MAB, and EcoGD are run for 150 optimizer steps in the case of fixed and bursty request arrival patterns. For the traces, we run the experiments for 3 hours as that is the length of the traces. We also repeat each experiment 3 times. The error bars in the bar plots of Section VI represent the standard deviation of the 3 runs.

All experiments are given 5 optimizer steps as a warm-up period to get rid of bootstrapping effects such as PyTorch's lazy initialization. All algorithms and result interpretations ignore these 5 steps. We also limit all algorithms' optimizer search space to prevent configurations that are known to be bad and can cause queue build-up (and may crash the experiments); these bad configurations comprised about 1% of the total possible search space.

E. Hyperparameters settings of different algorithms

Different algorithms have different hyperparameters that need to be tuned and set to achieve their best performance.

- **Grid Search** has no hyperparameters.
- **Linear Sweeps** has hyperparameters to define the order of parameters (batch size, GPU frequency, CPU frequency) to sweep through. We tried all 6 permutations of the order and found that the differences in the results, especially the cost achieved, were negligible. As such, we present the results for the permutation that performed the best (within the narrow margin of results): batch size sweep, then CPU frequency sweep, and finally GPU frequency sweep.
- **DVFS** only optimizes the CPU and GPU frequencies. So, we manually set the batch size to a fixed value. We initially consider batch sizes of 8 and 16, and show both sets of results in our evaluation. We find that batch size 16 is superior, so we fix the batch size to 16 for DVFS under trace-driven experiments.
- **Multi-Armed Bandit** (MAB) has hyperparameters to define the exploration/exploitation probability. While higher

exploitation probability results in lower energy consumption as the algorithm sticks to the best configuration, it may take much longer to find the best configuration. We set the exploitation probability to 0.9, exponential decay to 0.9, and the number of configurations when cold starting to 10, as they gave the best results in our experiments.

- **Bayesian Optimization**, similar to MAB, needs some initial configurations to start the optimization when cold starting. We set this value to 10 to make it comparable with MAB.
- **EcoGD** has hyperparameters to set the memory size of configuration cost history and the number of maximum loopbacks allowed to jump out of local minima. Having a higher memory size results in better performance when running under fixed inter-arrival patterns, but it harms the performance when the arrival pattern is dynamic. Setting a lower value for maximum loopbacks may result in faster convergence but can also lead to random spikes in the cost achieved. We set the memory size to 10 and maximum loopbacks to 10, as they gave the best results in our experiments. Unlike MAB and Bayesian, EcoGD does not need any initial configurations to start the optimization when cold starting.

VI. Evaluation Results

In this section, we present our evaluation results. We first consider Resnet50 under fixed and bursty load, in Sections VI-A and VI-B, respectively, to analyze the performance and characteristics of different algorithms. This performance analysis is summarized in Section VI-C for reference. We then present our main results in Section VI-D using Resnet50 and BERT-Tiny under three time-varying, real-world request traces; we analyze both mean and tail results under these trace-driven experiments.

A. Comparison of algorithms under fixed load

1) Best configuration achieved: We start our evaluation by comparing the performance of the *best* configuration achieved by each algorithm. For Grid Search and Linear, the best configuration is the one found after sweeping their respective parameter search spaces. In the case of learning-based algorithms (i.e., MAB, Bayesian, and EcoGD), the best configuration is the one that is selected by the algorithm after it converges; the algorithms are said to have converged when the same low-cost configuration is selected multiple times in a row. For DVFS, the frequency selection is done automatically by the governors based on various external factors that are not in our control, and DVFS does not have a distinct convergence phase. For a fair comparison, we select as 'best' configuration for DVFS the one that DVFS picks after it has taken the same number of steps as that taken by EcoGD to converge.

Figure 3 shows the energy consumed, latency achieved, and cost incurred by each algorithm once it converges to the best configuration; we show results separately for a fixed inter-arrival time (IAT) of 50ms and 90ms. All values for all algorithms are normalized by that under Grid Search for ease of comparison. Note that the cost is obtained based on homogenized energy and homogenized latency values (see Section IV-C); as such, the cost values are not simply a mean of the energy and latency values shown in the figure.

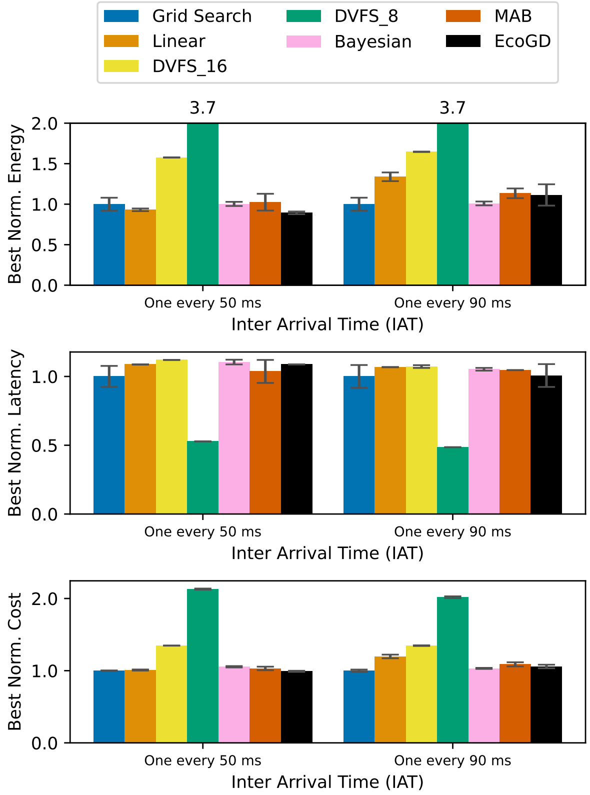

Fig. 3. Energy, latency, and cost incurred by all algorithms after convergence under fixed request inter-arrival times.

We see that Grid Search consumes the lowest cost in both the cases (tied with EcoGD for 50ms IAT), as it sweeps through all possible configurations to find the best configuration. This is to be expected as Grid Search is essentially an exhaustive search solution. Note that Grid Search does not necessarily consume the lowest energy in all cases; for example, it consumes more energy than Linear in the case of 50ms IAT. However, it does achieve the lowest *cost*, as that is the objective function being optimized; see Section IV-C for the cost definition. In the case of 50ms, it does so by obtaining a low latency to make up for the slightly higher energy consumption.

While Linear Search achieves roughly the same cost as Grid Search in the case of 50ms, it does so by sweeping through only a fraction of the configurations. But in the case of 90ms, Linear incurs more energy, latency, and cost than Grid Search. This is because Linear *independently* optimizes the three dimensions (batch size, CPU frequency, GPU frequency) and does not consider the interactions between them. As such, ***Linear Search may converge to a sub-optimal configuration***.

DVFS with batch size 16 has higher energy consumption, latency, and cost compared to Grid Search and Linear because it does not have access to workload information (e.g., batch size) and cannot optimize to it. DVFS with batch size 8 incurs the lowest latency among all algorithms and approximately 2× lower when compared with Grid Search because it has a lower batch accumulation delay. However, it incurs as much as 3.7× higher energy consumption. Consequently, it has the highest cost among all algorithms. This result highlights the ***need for joint optimization*** rather than only focusing on a single metric (latency or energy).

On further inspection, we found that this high energy consumption is because of the smaller batch size (8) for this variant of DVFS. All other algorithms converged to a high batch size (closer to 16) for their optimal configurations. In general, energy efficiency improves with batch size (due to amortization), and so a smaller batch size increases energy consumption. Prior works analyzing the impact of batch size in server-class machines for training and inference have made similar observations [3], [15]. This highlights the ***importance of optimizing workload parameters along with hardware parameters for jointly optimizing energy and latency***.

The learning-based algorithms (Bayesian, MAB, and EcoGD) typically achieve a cost only slightly higher than that achieved by (the exhaustive) Grid Search. They also achieve lower cost than sub-optimal configurations found by Linear in the case of 90ms IAT, and a much lower cost than that achieved by DVFS for both IATs. In general, we can conclude that the ***learning-based algorithms all achieve a fairly low cost under fixed IATs***. In fact, EcoGD achieves the lowest cost (tied with Grid Search) in the case of 50ms IAT.

2) Total cost till convergence, including overhead: The above best configuration results do not account for the energy, time, and cost incurred to arrive at the best configuration (via exploration, exploitation, and search space sweeps). To get a comprehensive view of the results, we now consider the cumulative energy consumption, inference latency, and cost incurred during the time it took each algorithm to converge to the best cost configuration. Figure 4 shows these metrics for each algorithm normalized by that under Linear Search; we chose Linear Search in this case for normalization because the performance of Grid Search is quite poor and would make it difficult to compare algorithms if normalized by Grid Search.

We see that ***Grid Search incurs prohibitively high overhead***, evidenced by the high values for all three metrics; this is to be expected as it needs to sweep through the 6,000 configurations (modulo the ones we discard due to predictably poor performance) to find the best one. Linear Search has much lower consumption when compared to Grid Search, as it only sweeps through 56 configurations, which represents more than a 100× reduction in search space size. In fact, Linear results in the lowest cost under 50ms IAT (but not under 90ms IAT).

Bayesian has high energy consumption, latency, and cost overhead; this is because Bayesian does not seem to converge well and keeps exploring newer configurations, resulting in frequent jumps in cost. This is because Bayesian Optimization does not have a mechanism to exploit the best configurations it has found so far and instead keeps exploring the search space. While Bayesian Optimization is useful in offline optimization (unlike our focus), it may not be very effective in online scenarios [43], [44]. While MAB has lower energy, latency, and cost compared to Bayesian, it still lags behind EcoGD in cost (and energy).

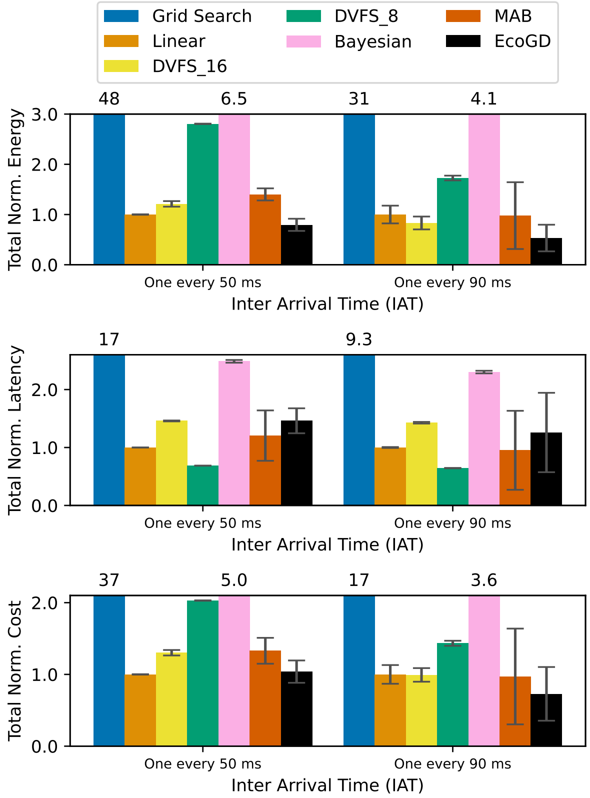

Fig. 4. Total energy, latency, and cost with overhead incurred by all algorithms till convergence under fixed request inter-arrival times.

DVFS with batch size 16 has higher energy consumption, latency, and cost than EcoGD. This is because DVFS does not have access to workload information and cannot optimize for it. Similar to the observations for best cost, we see that DVFS with batch size 8 has a lower latency because of the lower batch accumulation time but has much higher energy consumption and cost.

EcoGD has the lowest energy consumption among all algorithms. Further, and more importantly, ***EcoGD achieves the lowest cost under 90ms IAT and second-lowest cost (only 4% higher than Linear) under 50ms IAT***. This superior performance of EcoGD is because it is able to explore the search space through an efficient route using (observed and estimated) gradients while converging to the best configuration.

We also notice that the standard deviation is higher for MAB and EcoGD in case of 90ms IAT than other algorithms. This is because they explored the search space through different, diverse routes, although they individually converged to the same neighborhood of configurations eventually. For example, in case of 90ms IAT, EcoGD converged to the neighborhood of configurations [1.8, 0.96, 14], [1.91, 0.6, 15] and [1.19, 0.41, 15] in its three runs. Here, the first, second, and third elements of the configuration triplet refer to the CPU frequency (in

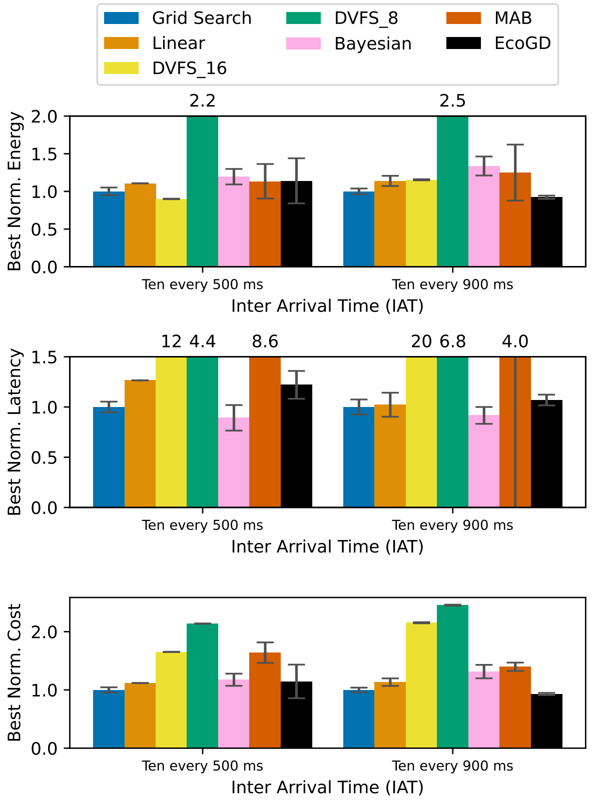

Fig. 5. Energy, latency, and cost incurred by all algorithms after convergence under bursty request inter-arrival times.

GHz), GPU frequency (in GHz), and batch size, respectively. We see that all three runs converged to high batch sizes; two of the runs converged to low GPU frequencies; two other runs converged to high CPU frequency. Despite the differences in the configurations, we find that their converged costs are within 4% of each other. This shows that ***multiple, local optima with similar cost values can exist***; since we optimize for cost, the actual converged configurations may be slightly different as long as they have similar cost values. In the case of MAB, the converged configurations were [1.57, 0.6, 13], [1.88, 0.51, 15], [1.91, 0.8, 15], and all resulted in converged costs that were within 5% of each other.

B. Comparison of algorithms under bursty load

We now move on to the case where the inter-arrival time of inference requests is bursty.

1) Best configuration achieved: Figure 5 shows the energy consumed, latency achieved, and cost incurred by each algorithm once it converges to the best configuration; we show results under bursty inter-arrival times of 500ms and 900ms between a batch of 10 successive requests. Grid Search incurs almost the lowest cost, energy, and latency in both cases of IATs as it sweeps through all configurations to find the best configuration. Linear Search takes more energy, latency, and cost than Grid Search in both IATs as it settles for sub-optimal configurations.

Both versions of DVFS have higher costs than Grid Search and Linear. We again find that DVFS with batch size 8 has the highest energy and cost among all algorithms (because of its lower batch size, as explained in Section VI-A1). We also see that DVFS with batch size 16 has significantly higher latency; this is because, with 10 requests arriving every 500ms and 900ms, it has to wait for enough batches to accumulate 16 requests before it can begin processing them.

MAB has significantly higher latency among the learning-based algorithms for both IATs. We find that ***MAB results in highly variable behavior under bursty load, often converging to sub-optimal configurations***. It also exhibits high standard deviation in latency under 900ms IAT due to the variable exploration routes it takes. Bayesian avoids the very high latency that MAB incurs, but still has higher cost than EcoGD.

Among the learning-based algorithms, EcoGD has the lowest cost and energy consumption for both bursty IATs. In fact, it has the lowest cost among all algorithms for 900ms IAT. It also has the second-lowest latency for both IATs among learning-based algorithms.

Figure 6 shows a heatmap of the converged cost achieved by each algorithm (see Figure 5) under the bursty 900ms IAT over the configuration parameters being optimized; we omit DVFS algorithms as their cost is quite high. The figure caption explains the axes and notations used. We see that different algorithms converge to different configuration neighborhoods. For example, Grid Search converges to lower GPU frequencies whereas EcoGD converges to higher GPU frequencies. Nonetheless, they both have low converged costs. This suggests that ***there are multiple local optima that could result in low cost***. Bayesian and MAB converge to smaller batch sizes, compared to EcoGD, which results in higher energy consumption as energy efficiency typically improves with batch size [3], [15].

2) Total cost till convergence, including overhead: Figure 7 shows the total energy consumed, latency achieved, and cost incurred by each algorithm until it converges to the best configuration; we show results under bursty inter-arrival times of 500ms and 900ms between a batch of 10 successive requests. All metrics for each algorithm are normalized by that under Linear Search.

Grid Search again performs quite poorly due to its very large search space. Linear continues to perform better than Grid Search. However, compared to fixed load (Figure 4), Linear does not perform as well in terms of total cost; for example, DVFS (with batch size 16) and EcoGD have lower cost than Linear for both IATs. *This suggests that a static algorithm like Linear is not well suited for bursty arrivals.*

Among the DVFS configurations, DVFS with batch size 8 has higher energy than DVFS with batch size 16 but much lower latency than DVFS with batch size 16 and all other algorithms. This is because a single burst of 10 requests can easily satisfy the batch size 8 requirement, resulting in very low batch accumulation time. But, as discussed in Section VI-A1, this comes at the cost of high energy consumption. Note that the cost values shown are based on homogenized versions of energy and latency (see Section IV-C for cost definition), and

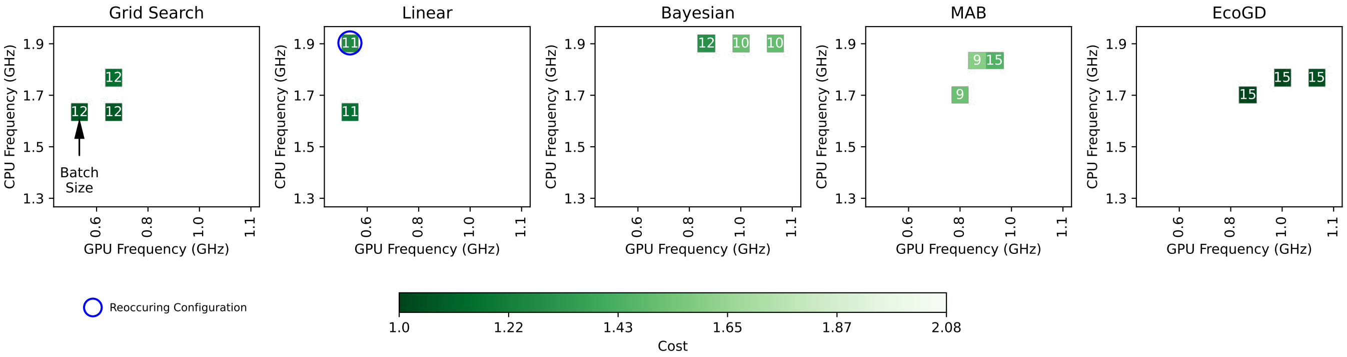

Fig. 6. Heatmap of converged cost, shown as squares, achieved by all algorithms under bursty request inter-arrival time of 900ms between even batch of 10 requests. The axes indicate the CPU and GPU frequency for the configurations; the batch size is denoted as the numbers inside the squares. The color shading indicates converged cost (see colorscale), with darker squares indicating lower cost. Circles indicate re-occuring converged configurations across runs.

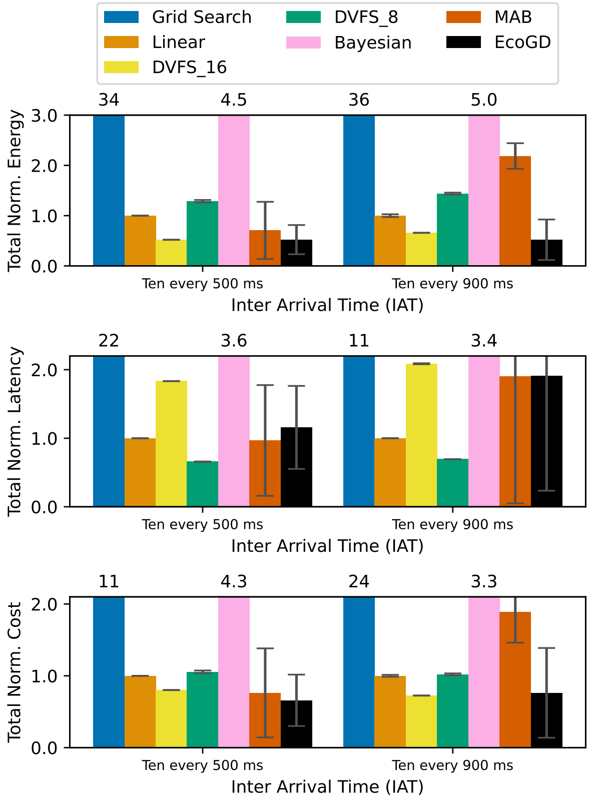

Fig. 7. Total energy, latency, and cost with overhead incurred by all algorithms till convergence under bursty request inter-arrival times.

so a very low (pre-homogenized) latency does not immediately translate to low cost.

Among the learning-based algorithms, we again see Bayesian performing poorly, similar to the case of fixed IATs, due to its inability to converge to good configurations. MAB again has a lower cost than Bayesian, but higher than EcoGD.

EcoGD continues to achieve very low cost, even under bursty load. Specifically, EcoGD has the lowest cost under 500ms IAT and the second-lowest cost (only 5% higher than DVFS with batch size 16) under 900ms IAT. This is primarily due to its low energy consumption coupled with its reasonably low latency. In fact, EcoGD achieves the lowest energy consumption for both IATs (tied with DVFS with batch size 16 for 500ms IAT). This shows that EcoGD performs well for static and bursty conditions.

C. Key takeaways from synthetic load patterns

The performance evaluation of algorithms from the above subsections can be informally summarized as follows. Grid Search, not surprisingly, converges to low cost configurations, but is not a viable option in practice due to its high overhead. Linear can be sub-optimal and is not well-suited for bursty workloads. DVFS with batch size 8 has low latency but high energy consumption and cost, whereas DVFS with batch size 16 has lower energy consumption but higher latency and high cost. Among the learning-based algorithms, Bayesian has low post-convergence cost but high overhead. MAB has higher energy overhead, which impacts its total cost; it also converges to configurations with very high latency in bursty IAT conditions, hurting its post-convergence cost. EcoGD achieves low post-convergence cost with low overhead in both fixed and bursty load conditions, typically achieving the lowest total cost among all algorithms.

D. Performance evaluation under real-world traces

We now present our key performance evaluation results by comparing the dynamic algorithms on real-world request traces described in Section V-B. One objective here is to evaluate how the algorithms perform under unpredictable and potentially abrupt changes in request arrival rate. Another objective is to study the impact on mean and tail metrics. We do not consider the static Grid Search and Linear Search algorithms in this subsection. Grid Search has very high overhead, as established in Figures 4 and 7; this overhead will be incurred much more frequently in trace-driven experiments, making Grid Search impractical. While Linear has lower overhead than Grid Search, its overhead and post-convergence performance are still sub-optimal, especially under bursty workload; this will get worse under dynamic traces.

1) Comparison for mean metrics: Figure 8 shows the energy consumption, latency, and cost achieved by different dynamic algorithms, normalized to those achieved under DVFS, for the three different traces we consider using

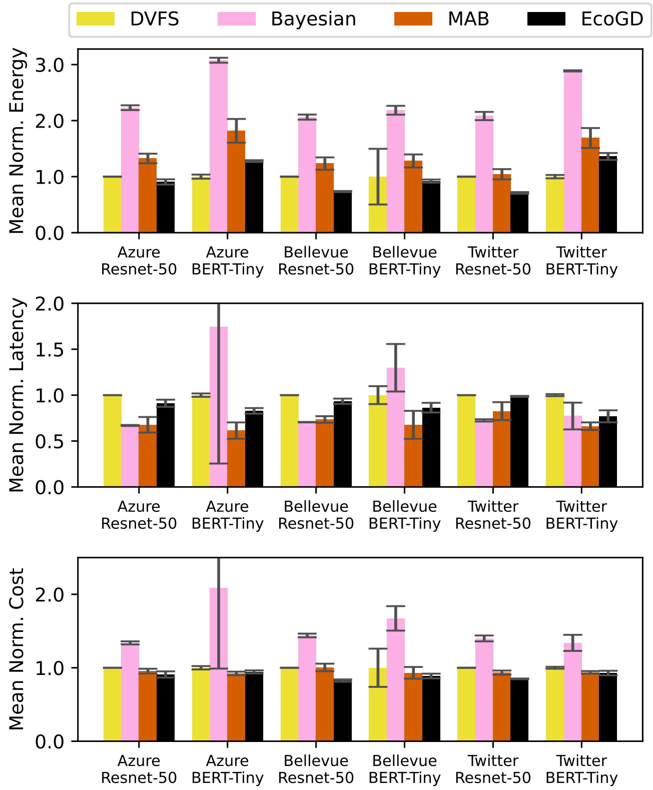

Fig. 8. Mean energy, latency, and cost incurred by all algorithms under real-world request traces using Resnet50 and BERT-Tiny.

Resnet50 and BERT-Tiny. Since DVFS with batch size 16 outperformed DVFS with batch size 8 for the fixed and bursty load conditions, we only consider DVFS with batch size 16.

We start by again noticing the poor performance of Bayesian, which incurs very high energy consumption, resulting in its high cost. We find that Bayesian does not converge to good configurations, which is in agreement with our findings from prior subsections. Coupled with the fact that the request rate changes frequently in the traces, ***Bayesian results in the highest cost*** in all six scenarios we consider.

DVFS maintains low energy consumption via its dynamic modulation of CPU and GPU frequencies. However, under the dynamic request traces, its fixed batch size contributes to high accumulation delay during periods of low request rate, resulting in high inference latency and (consequently) cost. MAB, on the other hand, is able to dynamically change the batch size and maintain lower latency compared to DVFS. However, it is unable to achieve the low energy consumption of DVFS, resulting in a somewhat similar (albeit slightly lower) eventual cost as DVFS.

EcoGD consistently achieves low cost across all scenarios by balancing energy consumption and latency. We see that EcoGD achieves the lowest energy among all algorithms for all Resnet50 scenarios. While it does not achieve the lowest energy for BERT-Tiny scenarios, it makes up for it with lower latency. On close inspection, we find that EcoGD has the lowest cost (which is our optimization metric) for all scenarios, except for Azure traffic with BERT-Tiny, where it has the second-lowest cost (2% higher than MAB). Averaged across all six scenarios, EcoGD reduces cost compared to DVFS, Bayesian, and MAB by 11%, 41%, and 6%, respectively. We find that the reduction is slightly higher for Resnet-50 scenarios (14%, 38%, and 10%, respectively).

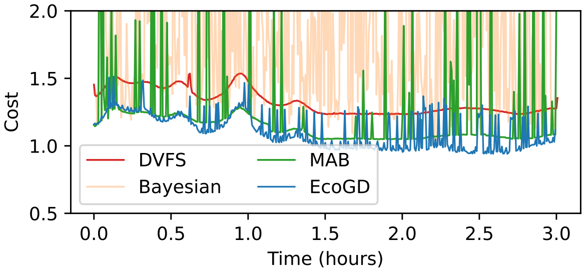

Fig. 9. Cost timeseries of DVFS, MAB and EcoGD for Bellevue trace with Resnet50 model.

2) Analyzing the adaptive behavior of algorithms: To better understand the adaptive behavior of the different algorithms, we plot their cost as a function of time for the Bellevue trace with Resnet50 model in Figure 9. We see that DVFS exhibits consistently high cost, largely because of its inability to adapt the batch size. EcoGD almost always achieves the lowest cost but does exhibit spikes in cost, occasionally resulting in momentarily higher costs than DVFS. These spikes are a result of our algorithm's configuration explorations as it attempts to look for better search spaces. MAB also exhibits such spikes for a similar reason, though the cost spikes quite significantly under MAB, often doubling its cost momentarily. This is because MAB has a mandatory exploration step, where it jumps to a random configuration from the entire search space, potentially resulting in a momentarily high cost. Bayesian (lightly colored to maintain visibility of other lines) can be seen to exhibit very variable cost behavior over time, with almost continual spikes. This is because Bayesian Optimization's expected improvement acquisition function is not designed to be adaptable to changing workloads [45]. As the load pattern changes under request traces, previously known configurations may become irrelevant, which Bayesian Optimization does not take into account.

The above observations provide insights into the workings of the algorithms and also highlight the consistently superior performance of EcoGD throughout the trace duration. We observe similar behavior for the other traces as well.

We also evaluated the performance of the algorithms under a synthetic trace where the inter-arrival time changed abruptly and immediately from 50ms to 90ms. We again found similar results with EcoGD performing the best (and obtaining 10% lower cost than the next best algorithm, MAB), suggesting that EcoGD is able to adapt to rapid changes in the workload. However, we note that the real-world traces we considered in Section VI-D1 did not exhibit such abrupt changes.

3) Comparison for tail metrics: In real-world scenarios, it is often important to optimize for tail metrics, such as tail latency, in addition to mean metrics [17], [46], [47]. As such,

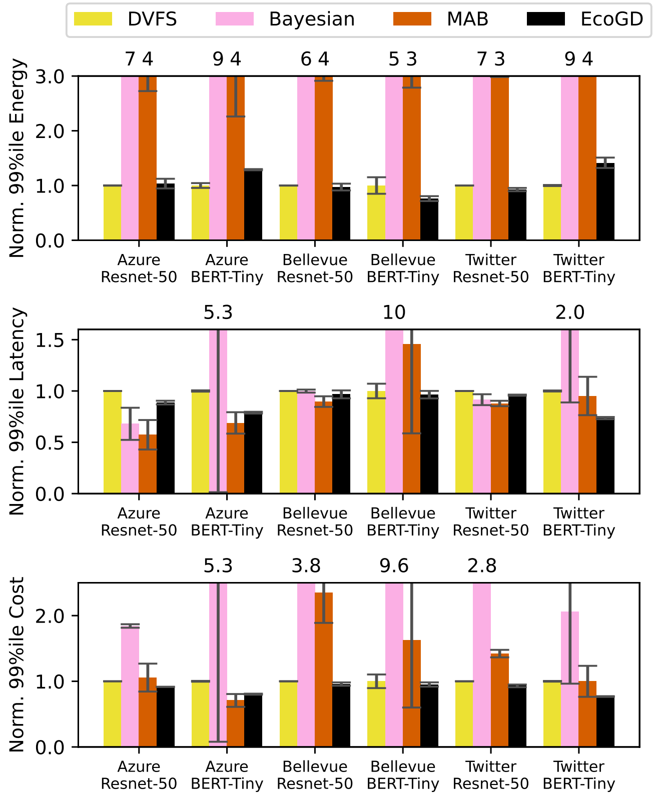

Fig. 10. 99th percentile of energy, latency, and cost incurred by all algorithms under real-world request traces using Resnet50 and BERT-Tiny.

we now analyze, in Figure 10, the 99th percentile energy consumption, latency, and cost achieved by different dynamic algorithms under all six trace-driven scenarios.

We see that the tail energy consumption under Bayesian and MAB is very high compared to DVFS and EcoGD; this was not the case for the mean energy consumption in Figure 8. We also find that the tail latency is high for Bayesian, especially for the BERT-Tiny model. MAB also exhibits high tail latency in some scenarios.

In terms of cost, Bayesian, as expected, has a high tail cost in all scenarios. MAB also results in a high tail cost, especially under the Bellevue request trace. This is because of MAB's probabilistic random exploration, which can result in high cost configurations being chosen over time. By contrast, ***EcoGD achieves the lowest 99th percentile cost for all scenarios, except for Azure traffic with BERT-Tiny, where it has the second-lowest cost (13% higher than MAB)***. Unlike MAB, EcoGD does not have a mandatory exploration phase and instead can decide to either continue exploiting the best known configuration or explore a new one based on its knowledge of the neighborhood. This allows EcoGD to avoid high cost configurations. Averaged across all six scenarios, EcoGD reduces tail cost compared to DVFS, Bayesian, and MAB by 11%, 72%, and 27%, respectively. We thus conclude that EcoGD achieves low mean and tail cost, making it readily applicable in real-world scenarios.

We also find that different algorithms perform slightly differently depending on the DNN model being employed. For example, MAB has lower latency and higher energy consumption under BERT-Tiny compared to Resnet50 in Figure 8, whereas Bayesian experiences high tail latency under the BERT-Tiny model in Figure 10. This is because BERT-Tiny has higher computational and processing requirements than Resnet50, resulting in different opportunities for optimization. However, the difference in performance for a given algorithm across traces is not that remarkable. This is likely because all real-world traces do exhibit similarly significant levels of load variations (see Figure 2). We do see slightly high tail latencies and costs under the Bellevue trace and slightly low tail latencies and costs under the Azure trace for MAB in Figure 10. This is likely because, among the traces we employed, Bellevue had the highest average load and Azure had the lowest average load.

VII. Conclusion

This paper focuses on dynamic and energy-efficient DNN inference on edge. We present EcoEdgeInfer, a framework that optimizes the tradeoff between inference latency and energy consumption by tuning hardware and software parameters dynamically and in response to changes in workload. Our core contribution is EcoGD, our adaptive parameter tuning optimization algorithm, inspired by Gradient Descent, that quickly converges to near-optimal configurations with low overhead while avoiding fluctuations in achieved cost. Evaluation results under three different request arrival traces and two different DNN workloads show that EcoGD consistently outperforms existing baselines, lowering the mean cost objective by as much as 55% (with 19% average reduction) and the tail cost objective by as much as 90% (with 36% average reduction).

Acknowledgment

This work was supported by NSF grants CCF-2324859, CNS-2214980, CNS-2106434, and CNS-1750109.

References

[1] R. Pugliese, S. Regondi, and R. Marini, "Machine learning-based approach: global trends, research directions, and regulatory standpoints," *Data Science and Management*, vol. 4, pp. 19–29, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666764921000485

[2] J. Vincent, "How much electricity does AI consume? — theverge.com," https://www.theverge.com/24066646/ai-electricity-energy-watts-generative-consumption, [Accessed 05-07-2024].

[3] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and optimizing GPU energy consumption of DNN training," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 119–139. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/you

[4] D. Gu, X. Xie, G. Huang, X. Jin, and X. Liu, "Energy-efficient gpu clusters scheduling for deep learning," 2023.

[5] OpenAI, "ChatGPT," https://openai.com/chatgpt/, [Accessed 05-07-2024].

[6] GitHub, "GitHub Copilot," https://copilot.github.com/, [Accessed 05-07-2024].

[7] S. Luccioni, Y. Jernite, and E. Strubell, “Power hungry processing: Watts driving the cost of ai deployment?” in *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 85–99. [Online]. Available: https://doi.org/10.1145/3630106.3658542

[8] R. Haight, W. Haensch, and D. Friedman, “Solar-powering the internet of things,” *Science*, vol. 353, no. 6295, pp. 124–125, 2016. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aag0476

[9] Z. Xu, L. Zhao, W. Liang, O. F. Rana, P. Zhou, Q. Xia, W. Xu, and G. Wu, “Energy-aware inference offloading for dnn-driven applications in mobile edge clouds,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 799–814, 2021.

[10] P. S.K, S. A. Kesanapalli, and Y. Simmhan, “Characterizing the performance of accelerated jetson edge devices for training deep learning models,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 3, dec 2022. [Online]. Available: https://doi.org/10.1145/3570604

[11] A. Dutt, S. P. Rachuri, A. Lobo, N. Shaik, A. Gandhi, and Z. Liu, “Evaluating the energy impact of device parameters for dnn inference on edge,” in *Proceedings of the 14th International Green and Sustainable Computing Conference*, ser. IGSC ’23. New York, NY, USA: Association for Computing Machinery, 2024, p. 52–55. [Online]. Available: https://doi.org/10.1145/3634769.3634809

[12] L. Foundation, “Sharpening the edge: Overview of the lf edge taxonomy and framework,” Jul 2020. [Online]. Available: https://www.lfedge.org/wp-content/uploads/2020/07/LFedge_Whitepaper.pdf

[13] “NVIDIA Embedded Systems for Next-Gen Autonomous Machines — nvidia.com,” https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/, [Accessed 05-07-2024].

[14] “Lernapparat - Machine Learning — lernapparat.de,” https://lernapparat.de/jit-optimization-intro/, [Accessed 05-07-2024].

[15] D. Li, X. Chen, M. Becchi, and Z. Zong, “Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus,” in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, 2016, pp. 477–484.

[16] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, “Batch: Machine learning inference serving on serverless platforms with adaptive batching,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s Highly Available Key-value Store,” in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07, Stevenson, Washington, USA, 2007, pp. 205–220.

[18] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne, “Carbon-aware computing for datacenters,” *IEEE Transactions on Power Systems*, vol. 38, no. 2, pp. 1270–1280, 2023.

[19] 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)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/weng

[20] E. Jeong, J. Kim, and S. Ha, “Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards,” *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 5, oct 2022. [Online]. Available: https://doi.org/10.1145/3508391

[21] Nvidia, “Software-based power consumption modeling,” https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3275/index.html, [Accessed 05-07-2024].

[22] E. Samikwa, A. D. Maio, and T. Braun, “Disnet: Distributed micro-split deep learning in heterogeneous dynamic iot,” *IEEE Internet of Things Journal*, vol. 11, no. 4, pp. 6199–6216, 2024.

[23] N. Diegues and P. Romano, “Self-Tuning intel transactional synchronization extensions,” in *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 209–219. [Online]. Available: https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues

[24] F. Glover, “Tabu search - part i,” in *ORSA Journal on Computing*, vol. 1, no. 3, 1989, pp. 190–206. [Online]. Available: https://doi.org/10.1287/ijoc.1.3.190

[25] “TorchVision — pytorch.org,” https://pytorch.org/vision/stable/index.html, [Accessed 05-07-2024].

[26] P. Bhargava, A. Drozd, and A. Rogers, “Generalization in nli: Ways (not) to go beyond simple heuristics,” 2021.

[27] I. Turc, M. Chang, K. Lee, and K. Toutanova, “Well-read students learn better: The impact of student initialization on knowledge distillation,” *CoRR*, vol. abs/1908.08962, 2019. [Online]. Available: http://arxiv.org/abs/1908.08962

[28] S. P. Rachuri, F. Bronzino, and S. Jain, “Decentralized modular architecture for live video analytics at the edge,” in *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, ser. HotEdgeVideo ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 13–18. [Online]. Available: https://doi.org/10.1145/3477083.3480153

[29] “GitHub - City-of-Bellevue/TrafficVideoDataset — github.com,” https://github.com/City-of-Bellevue/TrafficVideoDataset, [Accessed 05-07-2024].

[30] “Twitter Streaming Api — developer.twitter.com,” https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction, [Accessed 05-07-2024].

[31] “Internet Archive: Digital Library of Free &amp; Borrowable Books, Movies, Music &amp; Wayback Machine — archive.org,” https://archive.org/details/twitterstream, [Accessed 05-07-2024].

[32] “Deploy models with Amazon SageMaker Serverless Inference - Amazon SageMaker — docs.aws.amazon.com,” https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, [Accessed 05-07-2024].

[33] C. Karakus, R. Huilgol, F. Wu, A. Subramanian, C. Daniel, D. Çavdar, T. Xu, H. Chen, A. Rahnama, and L. Quintela, “Amazon sagemaker model parallelism: A general and flexible framework for large model training,” *CoRR*, vol. abs/2111.05972, 2021. [Online]. Available: https://arxiv.org/abs/2111.05972

[34] M. Zhang, C. Krintz, and R. Wolski, “Edge-adaptable serverless acceleration for machine learning internet of things applications,” *Software: Practice and Experience*, vol. 51, 12 2020.

[35] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, “Faster and cheaper serverless computing on harvested resources,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 724–739. [Online]. Available: https://doi.org/10.1145/3477132.3483580

[36] “Schedutil; The Linux Kernel documentation — docs.kernel.org,” https://docs.kernel.org/scheduler/schedutil.html, [Accessed 05-07-2024].

[37] S. Alabed and E. Yoneki, “High-dimensional bayesian optimization with multi-task learning for rocksdb,” ser. EuroMLSys ’21, 2021.

[38] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, “Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics,” in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’17, 2017.

[39] V. Dalibard, M. Schaarschmidt, and E. Yoneki, “BOAT: Building Auto-Tuners with Structured Bayesian Optimization,” in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW ’17, Perth, Australia, 2017, p. 479–488.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[41] G. Somashekar, K. Tandon, A. Kini, C.-C. Chang, P. Husak, R. Bhagwan, M. Das, A. Gandhi, and N. Natarajan, “OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy,” in *NSDI 2024*. Santa Clara, CA, USA: USENIX, 2024.

[42] S. Misra, S. P. Rachuri, P. K. Deb, and A. Mukherjee, “Multiarmed-bandit-based decentralized computation offloading in fog-enabled iot,” *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 10 010–10 017, 2021.

[43] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[44] B. Letham and E. Bakshy, “Bayesian optimization for policy search via online-offline experimentation,” 2019. [Online]. Available: https://arxiv.org/abs/1904.01049

[45] F. M. Nyikosa, M. A. Osborne, and S. J. Roberts, “Bayesian optimization for dynamic problems,” 2018. [Online]. Available: https://arxiv.org/abs/1803.03432

[46] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power management of online data-intensive services,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA ’11, San Jose, CA, USA, 2011, pp. 319–330.

[47] J. Dean and L. A. Barroso, “The Tail at Scale,” *Communications of ACM*, vol. 56, no. 2, pp. 74–80, 2013.