Many Models at the Edge: Scaling Deep Inference via Model-Level Caching

Samuel S. Ogden , Guin R. Gilman , Robert J. Walls , and Tian Guo

Computer Science Department, Worcester Polytechnic Institute
{*ssogden,grgilman,rjwalls,tian*}*@wpi.edu*

Abstract

Deep learning (DL) models are rapidly expanding in popularity in large part due to rapid innovations in model accuracy, as well as companies' enthusiasm in integrating deep learning into the existing application logic. This trend will inevitably lead to a deployment scenario, akin to the content delivery network for web objects, where many deep learning models—each with different popularity—run on a shared edge with limited resources. In this paper, we set out to answer the key question of *how to manage many deep learning models at the edge effectively*. Via an empirical study based on profiling more than twenty deep learning models and extrapolating from an open-source Microsoft Azure workload trace, we pinpoint a promising avenue of leveraging cheaper CPUs, rather than commonly promoted accelerators, for edge-based deep inference serving.

Based on our empirical insights, we formulate the DL model management problem as a classical caching problem, which we refer to as *model-level caching*. As an initial step towards realizing model-level caching, we propose a simple cache eviction policy, called *CremeBrulee*, by adapting BeladyMIN to explicitly consider DL model-specific factors when calculating each in-cache object's utility. Using a small-scale testbed, we demonstrate that *CremeBrulee* can achieve a 50% reduction in memory while keeping load latency below 92% of execution latency and less than 36% of the penalty of using a random approach to model eviction. Further, when scaling to more models and requests in a simulation, we demonstrate that *CremeBrulee* can keep the model load delay lower than other eviction policies that only consider workload characteristics by up to 16.6%.

Relevant research artifacts are available at https://github.com/cake-lab/CremeBrulee

I. Introduction

Deep learning models are exploding in popularity due to their widespread use both in industry [2] and in consumer-facing applications [11]. As the number of unique models and their popularity increases, the development of techniques that effectively host them on shared resources becomes critical. This trend will inevitably lead to a deployment situation akin to content delivery networks (CDNs), where hosting models near end-users on shared resources is essential to achieve the best performance [6]. Although numerous techniques exist for managing static, and more recently dynamic, content in CDNs, the complexity of deep learning models, and the requirements of using them make model serving complex.

Deep learning models are large in size, over 4GB in some cases [42], with complex execution graphs that need to be constructed upon model load. As such, naive memory management may encounter difficulties handling these models, experiencing unexpected latency variations, and not fully exploiting the characteristics of models. The scale of the workload can further compound the memory management complexity. As deep learning models proliferate, they are being used in myriad applications that were traditionally served by central servers or, more recently, run in serverless platforms. Extrapolating from a serverless trace [35], we expect deep learning models will see not only a huge number of requests but also a wide range of popularity, with some models being requested many orders of magnitude more often than others. Therefore, the core problem facing the serving of deep learning models is how to handle a large number of requests for a wide variety of models effectively in a shared infrastructure.

To this end, we propose *model-level caching* where deep learning models are objects in the cache and are kept, or removed, from the cache based on model characteristics and workloads. We next propose an initial step towards a solution through an eviction policy, *CremeBrulee*, that enables a basic form of model-level caching. Additionally, we introduce a methodology for testing model-level caching systems by creating a representative workload and a simulator. Our methodology demonstrates the effectiveness of model-level caching and provides a baseline for future systems.

We used an AWS-based testbed and a simulation to demonstrate our methodology and the effectiveness of model-level caching. We show that model-level caching can enable a trade-off between model load latency and memory usage. Specifically, by using *CremeBrulee* to manage models in our testbed, we can decrease the required memory by 2× while only incurring a 1.92x increase in response time, due to model load latency. Our simulation demonstrates that by explicitly considering the penalty of loading models into memory, we can improve over other eviction policies by reducing model load delay by up to 16.62%, and over random by up to 56.47%.

Our investigation, including the workload methodology and the DL-aware eviction policy *CremeBrulee*, provides a baseline for future model-level caching research. To summarize, we

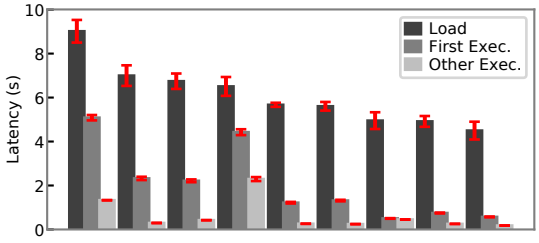


Fig. 1: Measurement of model latency. We show the average of model loading, first execution, and subsequent executions measure across 50 load-unload cycles with 10 executions per cycle for a subset of our characterized models including our largest and smallest models.

make the following contributions.

- We demonstrate that GPU-based inference is often not cost-effective for deep learning workloads due to deep learning models' popularity and resource requirements.
- We recast the deep learning model management problem as an extension of a classical caching problem.
- We propose a model-aware policy, called *CremeBrulee*, that is inspired by the BeladyMIN algorithm.
- We develop an empirical methodology for deriving the workload to evaluate the model-level caching policies and a baseline evaluation for future work on improving memory-computation trade-off via model-level caching.

The remainder of this paper is structured as follows. In Section II we will discuss deep learning models and their challenges in more detail. Next, in Section III we will characterize the resource demand of models and implications of workloads. In Section IV we show the strong parallels between it and traditional system caching problems. Drawing on this parallel, in Section V we introduce an eviction policy, *CremeBrulee*, for deep learning models as a first step towards model-level caching and evaluate it in Section VI by introducing a methodology for generating a realistic workload.

II. Motivation and Background

In this section, we introduce topics that are essential to understanding the state of deep learning serving. First, we discuss the models themselves, including function and cost. Next, we discuss accelerators, how they improve performance, and why they are appealing but not always needed. Finally, we need to discuss existing serving systems.

Deep Learning Models. The explosion of deep learning models, both in academia [22, 38, 42] and in industry [4, 11], has led to a need to efficiently service end-user requests. Models are now being used both to serve core components of both user end-devices [11] and business-oriented analytics models [2, 4]. These models are popular because they provide high accuracy for a wide range of problems. This accuracy is the result of leveraging extensive training resources, generally utilizing accelerators such as GPUs [1, 12, 17] or ASICs [25, 34], and large datasets [29, 33]. This high accuracy is related to having 100s of millions, or even billions, of weights [8, 22, 38, 42], leading to large model sizes.

Beyond simply being large, many of these models have several setup steps before usage due to having a complex execution structure [22, 38, 42]. In general, loading models consist of reading the model parameters from the disk and setting up the execution graph. We can see in Figure 1 that the first execution time of models is often much larger than the rest of the executions, likely due to graph initialization operations that must be run whenever the model is loaded. Therefore, systems must be aware that the penalty is not simply the size of a model.

Accelerators. Deep learning models consist of a large number of complex calculations, such as matrix multiplication. Accelerators, such as GPUs and ASICs [25, 34] can parallelize these operations efficiently, greatly decreasing the average latency due to increase throughput. This increased throughput is particularly useful during training when there is ready access to large amounts of training data. Additionally, parallel inference can greatly benefit the popular models that need to serve a large number of requests.

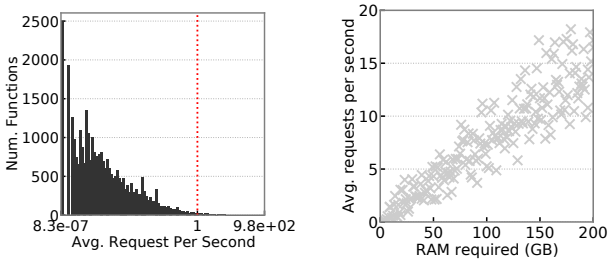
A core drawback of these accelerators is their high cost and limited memory access. On a per-request basis, accelerators generally offer no significant speedup but cost nearly 10× more, as we show in Section III. Further, they generally have limited on-device memory with potentially slow access times [17]. Essentially, accelerators are optimized for computation throughput rather than serving multiple models at once.

Model Serving. DL model inference serving systems have become an integral part of deep learning, benefiting from an active field of development. These include developer-oriented systems [3, 30], research-oriented systems [14, 18], corporate-oriented solutions [2, 4], and even what can best be described as boutique serving systems [23]. These systems aim to make it simple and easy to deploy models that developers and end-users can access, generally via a REST API or gRPC call. Often these systems are optimized for throughput targeting the hosting of a small number of extremely popular models [14, 18, 30].

However, with the increase in the number of models, and a move towards edge-based inference [26–28], these assumptions are no longer valid. The increasing memory pressure of hosting many models can quickly lead to hosting issues in edge- and fog-based systems where systems typically have limited available memory [1]. As we will see in the next section, this increased diversity of models and scarcity of resources drives the optimization of memory over computation.

III. Deep Learning Model Resource Usage

In this section, we show that CPU-based inference is better suited to host a large number of models than GPU-based inference. In particular, we show it has better cost-performance and strikes a better resource balance. Our findings are in contrast to the prevailing belief that these workloads require GPU-based servers. By considering the *relative cost* of resources in tandem with the characteristics of deep learning



(a) Avg. Requests per second. **(b) RAM Usage.**

Fig. 2: Azure Trace Overview. We analyze the Azure FaaS trace [35] to examine the distribution of the requests per second, and the implications of this distribution on resource usage. In (2a) we bin the number of requests per second for functions with logarithmic bins, showing that the majority of requests have less than *1 request per second*. In (2b) we show the potential queries per second when selecting functions from various quantiles, with the amount of RAM used being equal to the number of quantiles. We show that the amount of memory required increases much more rapidly than the number of requests per second.

serving *workload distributions*, we see a divergence from this traditional assumption. It is more cost-effective to host deep learning models on CPUs than on GPUs, and doing so will have minimal impact on model execution latency for most models. Further, memory is the most valuable resource for hosting deep learning models and CPUs have more cost-effective memory access.

Our key findings are based on a close study of the Azure FaaS workload [35] and empirical measurements of the TensorFlow Serving framework. These findings are as follows. First, for workloads that consist of multiple models, most inferences are memory-bound rather than throughput-bound. Therefore it is advantageous to have abundant memory. Second, in terms of execution latency, CPUs are largely comparable to GPUs, driving the conclusion that for individual inferences there is little downside to using CPUs. Finally, the monetary cost of CPUs is much lower than for GPUs in terms of per-unit memory. Therefore, leveraging CPUs is generally more cost-effective than GPUs and allows for access to a broader range of models at a lower cost.

A. Measurement Methodology

Our analysis consists of two parts. First, we examine a deep learning workload and draw conclusions based on the frequency of events. Next, we characterize many deep learning models and compare hardware used for serving them.

To assess the impact of workload characteristics, we parse an Azure FaaS trace [35]. We use this workload instead of a deep learning inference-specific workload as, at this time, no such workloads exist. We believe using a FaaS workload as a surrogate for inference is acceptable for two main reasons. First, many FaaS workloads have significant components comprised of requests for deep learning models [16, 23, 35]. Second, as deep learning is added into a range of applications it will likely augment or replace more traditional FaaS function invocations, leading to similar invocation patterns [7, 23]. Therefore, conclusions that can be drawn from considering the timing and distribution of events in a FaaS workload are very likely to apply to deep learning workloads.

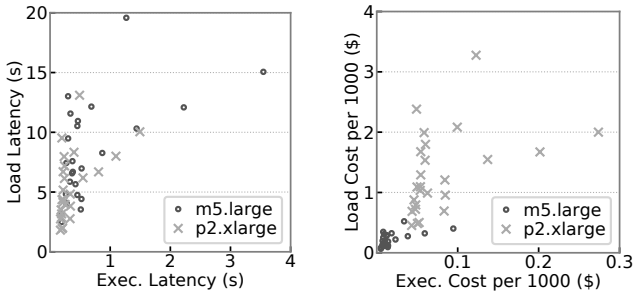
To characterize the real-world computational and memory requirements of serving deep learning models, we benchmark 26 CNN models available within Keras [13] using TensorFlow Serving [30]. For latency measurements, we ran directly on the hardware to allow access to available accelerators and reduce resource contention for both an AWS `m5.large` CPU instance and an AWS `p2.2xlarge` GPU instance with costs of $0.096/hr and $0.90/hr, respectively. To load and unload models, we leverage the TensorFlow Serving's feature to request model unload/load via a configuration file. We ran models within a docker container for memory measurements by measuring the delta in memory usage across each load, inference, and unload action. This delta-based approach allows us to consider the impact of each step, isolating the usage as much as possible.

B. Computational and Memory Usage Disparity

Through interpolating Azure's FaaS workload [35] for deep learning serving, we conclude that models can experience a wide distribution of request rates and require more memory than computation. Although some models have nearly 1000 requests per second, more than 99% of models are used less than once per second, as we see in Figure 2a. Further, over 97% of models are used less than once every 10 seconds. Based on what we will observe in the next section, these models are idle more often than they are in use, indicating that the vast majority of models could be unloaded to reduce memory usage. Overall, we see high memory consumption compared to computational usage, as shown in Figure 2b.

Commonly in inference serving, parallelism is used to increase throughput [8, 14, 30, 32], but we see that few models have sufficient throughput demand to take advantage of this parallelism. In Figure 2a we show the popularity of function calls in our trace by plotting the average usage per second broken into 100 buckets on a log scale. It can be seen that this distribution has an extremely long-tail distribution with the majority of functions are used less than once per second, and only 0.63% of functions being called more than once per second, and only 2.35% are called once every 10 seconds. Therefore, the majority of inferences would consist of a single request, removing the benefit of parallelism.

Such workload characteristics have ramifications for resource usage as well, as we show in Figure 2b, which shows potential RAM usage based on some simple assumptions. We selected functions from bins grouped by quantile of popularity, with the number of bins ranging from 1 to 200. We then summed and calculated the number of requests per second for each selection of functions while assuming each function required 1GB of memory. We repeated this process 1000 times and averaged the requests per second. We see that even as the memory usage increases to 200GB, the typical number of requests stays below 20 requests per second. Therefore, to get the equivalent number of requests as benchmarks target [31], it would be necessary



(a) Time per inference **(b) Cost per 1000 inferences**

Fig. 3: Latency and cost for 26 models on *m5.large* (CPU) and *p2.xlarge* (GPU) instances. We observe that while the CPU-based instance can have longer execution and load latencies they are generally on the same order of magnitude and it has a greatly reduced cost when compared to its GPU-based counterpart. While a GPU is better for high throughput operations, a CPU is much more cost-effective for less popular models, without sacrificing latency.

to host 200 individual models, making it unlikely that any will be computationally saturated.

C. Load and Execution Latency Penalty

We see in Figure 3a that overall execution latencies on both CPUs and GPUs are of approximately the same order of magnitude. This result indicates that both CPU and GPU can work equally well when serving inference workloads that largely exhibit low throughput requirements. While CPUs see a small number of outlier latencies, particularly for model loads, the majority of models have largely similar performance to GPU-based measurements. Overall, so long as unnecessary model loads are avoided, the latency differences are generally small.

D. Load and Execution Monetary Cost

Leveraging CPUs instead of GPUs can decrease the cost of hosting deep learning models, as we show in Figure 3b, even while maintaining similar execution and load latency. This is due to the much lower costs of CPUs, which is roughly 10× less in the case of our `m5.large` and `p2.xlarge` instance. Loading a model on a CPU-based instance costs approximately the same as executing a model on a GPU-based instance, despite being a much longer operation. Therefore, when using CPU-based systems, we can execute for a much lower cost than GPU-based systems, and the model loading also incurs relatively lower costs.

The cost difference between CPU and CPU instances extends to memory as well. GPU memory is 6.6× more expensive per GB/hr than CPU memory, and even the system memory of GPU-based systems is 1.2× more expensive. The cost disparity demonstrates that CPU-based systems are much more cost-effective for memory operations and thus implies that CPU-based systems are much more effective when hosting a large range of models. Note, the cost of memory in a FaaS system can be lower, but extra costs quickly grow, especially with a large workload [15], thereby making it inappropriate for our use case.

E. Key Takeaways

Our empirical characterization demonstrates the untapped potential of leveraging CPU as a cost-effective way for serving a large number of deep learning models with various popularity. We summarize our key takeaways.

Takeaway 1: It is often unnecessary to keep most models in the memory, given their usage patterns. The slowest CPU-based model execution we saw was under 4s, while more than 97% of functions were requested less than every 10s. The implication is that for 97% of models, over half of their in-memory time is spent idle.

Takeaway 2: When hosting more models, the vast majority will likely be unpopular, leading to a much higher memory demand than computational demand. A few models, as seen in Figure 2a are extremely popular and could benefit from dedicated accelerators, but most functions are much less commonly invoked, and thus consuming memory, but not computational, resources.

Takeaway 3: The monetary cost of keeping models in memory is non-negligible and is more when using accelerators than when using CPU-based instances. The cost of memory on a GPU is roughly 6.6× more expensive than system memory in a CPU-based system. Therefore, keeping uncommonly used models in GPU memory is expensive, and CPUs are a much more cost-effective option.

Putting all of this together, we see that CPU-based inference is cost-effective, does not introduce undue latency increases, and allows for removing unused models at a low cost.

IV. The Model-Level Caching Problem

We next formulate this deep learning model management problem as a caching problem, which we refer to as *model-level caching*, and focus on designing and evaluating deep learning-aware eviction policies. In general, for model-level caching, we consider each deep learning model as a cacheable object. In this paper, we specifically focus on a subset of this problem, determining which in-memory model(s) to evict to serve incoming model inference requests. Similar to traditional OS-level page caches, our goal when designing model eviction policies is to minimize the *cache miss penalty*—the penalty incurred when an inference request asks to run on a model that is not currently in the memory. Unlike page caches with homogeneous pages, it is non-trivial to quantify the cache miss penalty for model-level caching as deep learning models are *heterogeneous* both in terms of in-memory size and model load and preparation time.

At a given time, we assume the need to manage a set of n deep learning models $\mathbf{M} = \{m_0, \ldots, m_n\}$ for serving end-user inference requests. We denote the size of each model by $|m|$, and the size of a set of models as the sum of the size its constituent models, e.g. $\Sigma_{m \in \mathbf{m}} |m| = |\mathbf{M}|$[1]. Further, we assume access to a cache of size $\mathcal{G}$ and that only a proper

[1]Note, more fine-grained caching, such as per-layer approaches, could enable it to be the case that $\Sigma_{m \in \mathbf{m}} |m| > |\mathbf{M}|$ but this is considered future work.

TABLE I: Tracing Eviction Decisions We show the eviction decisions made by *BeladyMIN* and *CremeBrulee-oracle* given the same set of input requests. Each model has a penalty equal to one more than its number (e.g. model 0 has cost 1). We see that although they have the same number of cache misses *CremeBrulee-oracle* has a lower penalty.

	BeladyMIN				*CremeBrulee-oracle*			
Model	Hit/ Miss?	Evict	Resulting Cache	Cost	Hit/ Miss?	Evict	Resulting Cache	Cost
0	Miss		0	1	Miss		0	1
1	Miss		0,1	2	Miss		0,1	2
2	Miss	0	1,2	3	Miss	0	1,2	3
1	Hit		1,2	0	Hit		1,2	0
0	Miss	1	0,2	1	Miss	1	0,2	1
2	Hit		0,2	0	Hit		0,2	0
1	Miss	2	0,1	2	Miss	0	1,2	2
0	Hit		0,1	0	Miss	1	0,2	1
2	Miss	1	0,2	3	Hit		0,2	0
0	Hit		0,2	0	Hit		0,2	0
2	Hit		0,2	0	Hit		0,2	0
1	Miss	2	0,1	2	Miss	0	1,2	2
0	Hit		0,1	0	Miss	1	0,2	1
2	Miss	0	1,2	3	Hit		0,2	0
Total	**8 Misses**	-	-	**17**	**8 Misses**	-	-	**13**

subset of models $\mathbf{C} \subsetneq \mathbf{M}$ can be in-memory at once. For each incoming inference request specifying a model m_i, our goal is to pick an eviction set $\mathbf{E} \subset \mathbf{C}$ such that $\mathcal{G} - |\mathbf{C} - \mathbf{E}| \geq |m_i|$, i.e., to evict enough models to make room for the recently requested m_i. Further note, in a page cache, since pages are all the same size, the eviction set $\mathbf{E}$ will consist of one page; but in model caching, it may also consist of either no models or multiple models.

We break the model eviction problem into two subproblems: *(i)* how to quantify the *utility* of cached models? Unlike page caches where pages utility solely depends on the future access pattern [5], we will need to consider additional factors for model-level caching as described in Section V-A. *(ii)* how to form the eviction set $\mathbf{E}$ based on model utilities? Unlike page caches where pages are of the same size and the eviction is always one-to-one (one page in and one page out), we will need to consider a bin-packing-equivalent problem as described in Section V-B. For simplicity, we do not consider pre-fetching models as the success often hinges on the inherent workload locality and accurate workload prediction.

V. Model-level Cache Eviction Policies

In this section, we introduce a deep-learning model aware eviction policy called *CremeBrulee*, which follows a similar intuition as *BeladyMIN* [5]—a page replacement algorithm. Like paging, our model-level caching requires cachable objects (i.e., models) to be present in the memory before execution. *CremeBrulee* accounts for both the cache access pattern and the heterogeneous cache miss penalty in determining the model utility—amortizing the cache miss penalty across the time the model is out of the cache. In essence, *CremeBrulee* considers the utility of a model as the opportunity cost as if it were not removed by choosing to evict in-memory models with a lower amortized penalty (i.e., lower utility) to make room to service incoming inference requests. Intuitively, the sooner a model is requested again, the less time over which we can spread the cache miss penalty and the higher the utility. We refer to *CremeBrulee* as a *penalty-aware* eviction policy since it considers the penalty of having to reload a model.

A. Model Utility Calculation

Our *CremeBrulee* eviction policy calculates model utility as:

$$utility(m) = \frac{penalty(m)}{|m|} \cdot \frac{1}{B(m)} = \frac{penalty(m)}{|m| \cdot B(m)} \quad (1)$$

where $penalty(m)$ denotes the cache miss penalty and $B(m)$ (often referred to as *Belady boundary*) is the number of requests until the model m is next requested. We chose a cache miss penalty, $penalty(m)$ based on model *load time* based on empirical observations. Equation 1 thus balances the need to simultaneously reduce the number of model reloads, through the use of $B(m)$, and to keep high penalty but small models in the cache, via $penalty(m)$. This equation allows us to compare utility to determine eviction priority.

The key insight in designing *CremeBrulee* is that when a model is removed from the cache it will need to be reloaded at a future point. By considering the effect of amortizing this cache miss penalty across the time the model is out of cache, and thus the time during which it would afford cache space, we have a uniform metric to compare different models. Further, we can see that this has two important and useful properties. First, for a given model m, the value $utility(m)$ is *monotonically increasing* until the next usage. As such, if at the current time step t_0 the utility of model m_1 is $utility(t_0, m_1)$, then for any future time step t_n less than the next usage it will be the case that $utility(t_0, m_1) < utility(t_n, m_1)$. Further, for any other model, m_2, if $utility(t_0, m_1) < utility(t_0, m_2)$ then $utility(t_n, m_1) < utility(t_n, m_2)$ for all t_n until one of the models is next used. Therefore, we know that the model with the lowest utility is the best available option to remove. Second, in the case of homogeneous models, penalties and sizes $utility(m)$ can be reduced to the original *BeladyMIN*.

Table I presents the step-by-step caching states for a workload consisting of 13 requests using *BeladyMIN* and *CremeBrulee-oracle*. Recall that both *CremeBrulee-oracle* and *BeladyMIN* assume the knowledge of future inference workload. We see that each algorithm has a total of 8 misses, 3 of which are compulsory misses due to the cold start. However, *CremeBrulee-oracle* has a lower overall penalty, thanks to its preference to remove models with low cache miss penalty.

In lieu of a priori knowledge of upcoming requests for calculation of the Belady boundary $B(m)$, we leverage the inference request history of each model to predict the next request arrival. Specifically, for each model m, we maintain its average request rate λ_m and estimate $B(m) \approx B'(m) = \frac{1}{\lambda_m}$. Such approximation is reasonable for inference workloads that follow a Poisson distribution. We use *CremeBrulee-oracle* for establishing an upper bound performance in Section VI.

B. Model Eviction Set

To build the eviction set $\mathbf{E}$ we want to select the set of models that has the minimum utility. In general, this should be done by exhaustively testing all potential caches that would free

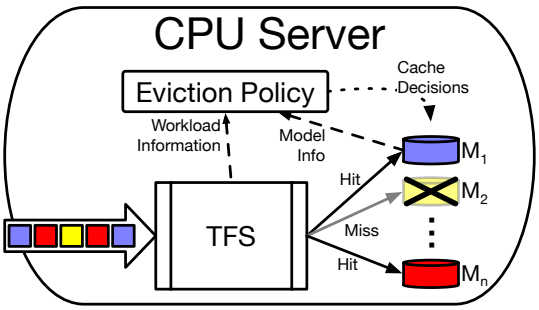


Fig. 4: Overview of evaluation testbed. Our testbed consists of a python script that invokes a Tensorflow Serving Docker image and controls which models it has loaded. As requests enter they are put in an internal queue. When a request reaches the head of the queue if the requested model is loaded (e.g. M_1 and M_n), then it is forwarded for execution. If the model is not loaded (e.g. M_2), then space will be freed to load the requested model. This load incurs some delay in the critical path of the model execution, the *model load delay*, which we report as a key metric. This eviction and loading is handled by the *eviction policy*, of which *CremeBrulee* is an example.

enough space to admit the new model. However, in practice, this is prohibitively expensive due to the number of models in the cache. Therefore we iteratively build our eviction set by selecting the model with the lowest utility from the cache and adding it to the potential eviction set. Once we have constructed an eviction set in this manner, we sort the models by decreasing size to remove the largest models first, in case this reduces the number of models removed.

VI. Evaluation

In this section, we first present our methodology in deriving the Azure FaaS workload trace [35] to evaluate the model-level caching policies. We use this methodology in a testbed (Section VI-B) and a simulation (Section VI-C) for evaluating *CremeBrulee*. We demonstrate that our *CremeBrulee* cache eviction policy achieves a 16.6% reduction over a non-penalty-aware eviction policy, and can smoothly reduce memory by up to half in our testbed.

A. Methodology

We consider our methodology in two parts. First, we consider the workload trace generation to aid other researchers in leveraging it as a deep learning trace. Next, we consider the specific experiments which utilize the workload.

1) Workload trace generation: In this section, we propose a methodology for generating a deep learning model inference trace from an existing Azure FaaS trace. As we have discussed previously in Section III-A, no publicly available deep learning-oriented traces exist, so we use a FaaS trace to simulate one. The Azure FaaS trace consists of per-minute function invocation counts and function statistics. We used a three-step process to generate a request-level workload: *(i)* generate a list of request events to represent when functions were invoked, *(ii)* connect these requests to known deep learning model statistics, *(iii)* uniformly downsample them to appropriate levels. The code for producing deep learning inference serving appropriate traces, as well as benchmarking models, can be found in our GitHub repository [10].

Event Generation. To generate the request-level workload, we counted the number of invocations of each function and equally distributed them in a random order across the minute. To best match the characteristics that make FaaS workloads to deep learning workloads, we only used functions triggered by HTTP requests and thus likely to be invoked by users. This filtering preserved the original population demographics, so models that were originally popular remained popular. In addition, we removed functions that were above the 90^{th} quantile of popularity, approximately once per minute or more, and one-hit-wonders. We chose to do this because extremely popular models should be optimized for throughput on dedicated hardware accelerators. At the same time, it is hard to quantify the caching benefit for one-hit-wonders.

Model Association. To associate models with specific events in the generated trace, we need to pair function invocations with model characteristics. The original characteristics of the function calls (e.g., the memory and latency provided with the Azure trace) are insufficient both because these would not be accurate to deep learning models and because, in many cases, they were incomplete. Therefore, for each function, we want to select the characteristics of one of our 26 deep learning models, profiled in Section III. Note that new models can be characterized but care should be taken to use the same class of machines.

For each function invocation in the event trace, we pick a model in one of four ways. First, a *uniform* approach in which we randomly selected a model. Second, a *round-robin* approach where models were chosen in a round-robin manner in order of decreasing load latency. Third, a *quantile* approach which selected models based on the popularity quantile of a function. This lead to more popular functions being associated with models that had high load latency. This allowed us to introduce a strong correlation between popularity and reload penalty. Fourth, *quantile-r* reverses this correlation.

Finally, we either renamed the invocations to use real names or distinct names. For testbed evaluation, real names are appropriate as they use the names of the models, and thus in our testbed use only the 26 real models. For simulations, distinct names are appropriate, leading to up to 30440 distinct models to simulate caching.

2) Experiment Setup: Our experiments consist of two main parts: a testbed and a simulation. These allow us to demonstrate the possibility of model-level caching and how well our proposed eviction policies scale, respectively.

Our testbed consists of a python wrapper around a TensorFlow Serving Docker image. An overview of our testbed is shown in Figure 4. The python wrapper communicates model changes to Tensorflow Serving through a model configuration file. When a request for a model in the cache arrives, it is forwarded for execution. When a request is received for a model that is not in the cache, our testbed determines the necessary cache modifications and alerts TFS to the changes.

TABLE II: Workload Statistics We test all four model correlations in our testbed and three of them in our simulation (the fourth, *round-robin* is similar enough to *uniform* to exclude). The key difference between our testbed and simulation workloads is that the testbed utilizes our 26 real models while our simulation uses 2408 unique models based on our real models.

Workload	Correlation (Pop $\sim$ Cost)	# Events (Testbed)	Memory (Testbed)	# Events (Simulation)	Memory (Simulation)
uniform	≈ 0	1000	21.7GB	–	–
round-robin	≈ 0	1000	21.7GB	–	–
quantiles	≈ 1	1000	21.7GB	–	–
quantiles-r	≈ -1	1000	21.7GB	16765	2.862GB

We ran our testbed on an AWS `m5.2xlarge` instance. This instance was chosen as model characteristics should be similar across all AWS `m5` type instances, and this is the smallest that can load all models into memory at once. We can emulate a baseline TFS installation by loading all models at once since no cache misses will occur.

Our testbed processed requests in a single-threaded manner to ensure that we could clearly delineate between measurements and reduce interference. This lead to increased overhead, which we demarcate in our results, that would not be present in a real-world implementation. Before testing, we loaded all models into memory in a random order, allowing our eviction policy to remove models as needed.

For our simulation, we built a custom trace-driven event-based simulator in python that is modeled on our testbed. We use the same module for both simulator and testbed to make eviction decisions, ensuring consistency across our tests. This simulation can be used as a basis for testing caching techniques, as well as extended to accommodate other workloads and models [10].

Workload Summary. For our testbed, we used 1000 requests over the course of an hour. This represents uniform downsampling of our requests to accommodate the machine processing power. These 1000 requests are distributed among our 26 real models with 4 different correlations to allow us to test a range of workload variations. Since we used real models, all variations would require 21.7GB to load concurrently. For our simulation, we used an hour of requests from our workload downsampled by 10%, consisting of 16765 requests for 2408 models. Since we are no longer bound by our single-threaded execution this allows us to consider the impact of many more requests over the course of an hour. Similar to our testbed, we used different workload correlations that would require approximately 2.9TB of memory to load concurrently. We summarize this information in Table II.

Key Metrics. We report two main metrics: cache size and model load delay. These two metrics represent the main trade-offs in our system, where smaller cache sizes mean more models need to be reloaded. *Cache size* is the size of the cache we are using in a given test. It allows us to see how much of a reduction in memory footprint we can make for a given increase in model load delay. *Model load delay* is the amount of time dedicated to loading models on the critical path for an inference. It is calculated as the load time for a model multiplied by the model-specific miss rate. For eviction policy evaluation, this is the most relevant metric since this is the delay that would be directly passed on to end-users. This is analogous to the *average memory access time*, although instead of measuring the impact of different cache levels, we measure the impact of diverse load and setup times.

Alternative Eviction Policies. To demonstrate the impact of our penalty- and size-aware eviction policies *CremeBrulee* and *CremeBrulee-oracle* to three other policies. First, we compare *CremeBrulee* to *popularity*, where the most popular model is selected since it is the impact of removing the penalty and size weighting. Next, we compare *CremeBrulee-oracle* to *BeladyMIN*, testing the inclusion of penalty when given future knowledge. Finally, we compare each set of policies to *Random* to demonstrate the improvements of informed selection.

B. Testbed

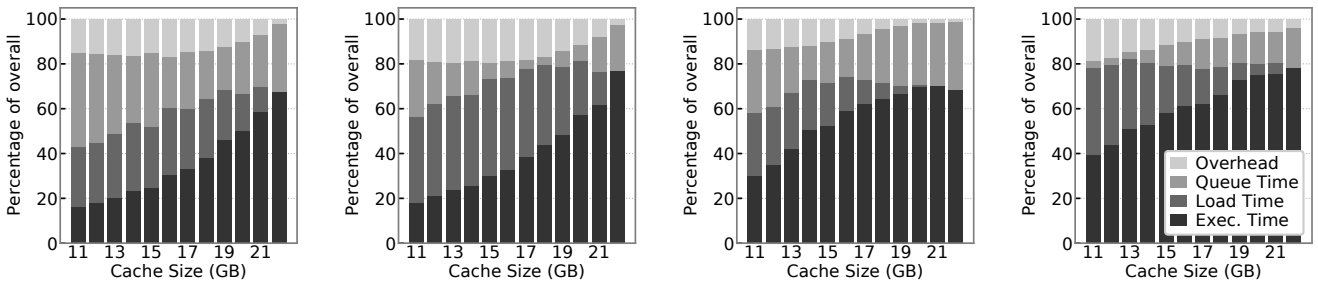
We first demonstrate the effectiveness of model-level caching in a testbed and the effects of different workload characteristics.

1) Impact of Model Caching: To examine the impact of model-level caching we use our *CremeBrulee-oracle* eviction policy compared to keeping all models loaded into memory. Keeping all models in memory requires 22GB of memory and is equivalent to baseline TensorFlow Serving. We saw that our testbed added overhead to request routing, decision making, and queueing delay, all of which we report but do not consider. The first two of these could be reduced by using a more optimized implementation. The latter is due to using single-threaded execution to best expose the impact of model caching, and thus would be expected to be much less in a multi-threaded implementation.

For each of the four workloads discussed in Section VI-A1 we tested with cache sizes ranging from 11GB to 22GB, the results of which are shown in Figures 5 and 6. We see that in many of our workloads at many cache sizes, the largest component of the overall latency is execution time. In fact, we observe that in workloads with correlation, such as we see in Figures 5c and 5d, it comprises the largest component at all cache sizes, at all times comprising 6.0% more of the overall execution time than the model load time. This strongly indicates that when introducing model-level caching in correlated workloads, the impact of model-loading is less than the impact of execution.

In Figures 5a and 5b we see that the model load time does overtake the model execution time. This increase relative to correlated workloads shows that when the popularity and penalty of models are uncorrelated, it is much more difficult to accurately perform caching. The worst case is in a *round-robin* workload where the model loading time contributes 13.5% more to the overall latency than the execution time.

Considering overall latency in Figure 6 we see that different workloads have significantly different results. Most dramatically, *uniform* and *quantiles-r* are the high- and low- points, respectively. The core of this difference is the correlation between model popularity and penalty. The *quantiles-r* workload



(a) ***Uniform*** **(b)** ***RoundRobin*** **(c)** ***Quantiles*** **(d)** ***Quantiles-r***

Fig. 5: Average component parts of overall latency at different testbed cache sizes. We show the impact of 1 hour of requests with various correlations between functions and models by the per-request average of components of the overall latency. At large cache sizes, the majority of the overall latency is dedicated to execution latency in all cases. As the cache size decreases from 22GB we see that load time rises from 0 to become a significant factor in non-correlated workloads. Similarly, we see that queue time increases as requests need to wait for the loading of models that arrive ahead of them, a byproduct of a single-threaded design.

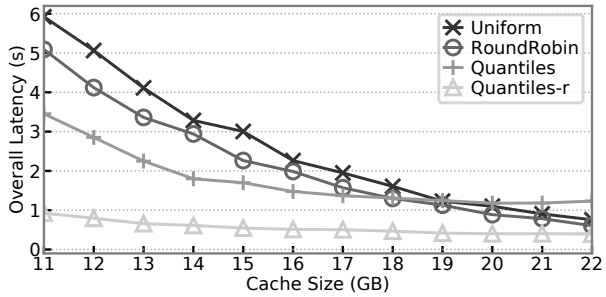
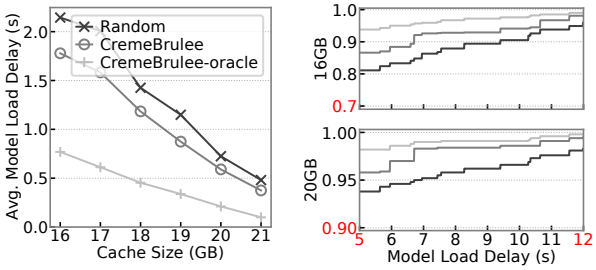


Fig. 6: Comparison of overall latency of different workloads in testbed. When varying the size of the cache in the testbed from 16GB to 22GB we show the variation in how different workloads and their overall latency.



(a) Avg. Model Load Delay **(b) Model Load Delays CDF**

Fig. 7: Avg. Model Load Delay of cache eviction policies in testbed. When varying the size of the cache in the testbed from 16GB to 21GB we show the variation in how different eviction policies affect the queue delay. Measured model performance information was given to all algorithms, but only *CremeBrulee-oracle* knew the future workload.

consists of models that are high-popularity and low-penalty, leading both to low execution latency and low penalty on misses for the most popular models. Meanwhile, the *uniform* workload has an average execution latency, as shown at cache size of 22GB, and is more difficult to determine the best models to keep in the cache. However, even with this difficulty, it can reduce the cache size significantly with minimal increase.

We can therefore conclude that model-level caching is viable in real-world applications. In the best case where model penalties are correlated with popularity, either positively or negatively, it can keep the model load time below the execution time of models while reducing the cache size in half. Even in the worst case, where model penalties are randomly associated with their popularity, it can still improve resource management. We expect the former case to be more likely, as these high-cost models are typically the more accurate, and thus can conclude that model caching would be quite effective in the real world.

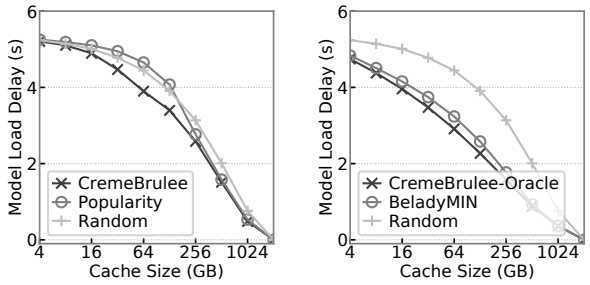
2) Comparison of caching algorithms: We next turn our attention to the importance of caching policies in the real-world testbed. Previously we used our *CremeBrulee-oracle* policy to examine how well we could perform model-level caching. To demonstrate the difference that the correct caching algorithm can make, we next compare it to *CremeBrulee* and *random* eviction policies. To do so, we used our testbed and a *uniform* model correlation to measure the impact of these different eviction policies. We show the results in Figure 7.

At all points, *CremeBrulee-oracle* outperforms the two alternative algorithms, as we see in Figure 7. This makes sense as it has foreknowledge of the upcoming requests and thus can more accurately select which models to evict. Even without knowledge of upcoming requests *CremeBrulee* performs well, outperforming *random* handily. This is due to being aware of the penalty of reloading models, as we can see in Figure 7b where high-cost models are more likely to be kept in memory.

C. Simulation

A key component of a deep learning inference workload is the number of models that we expect to host. While in the previous test, we demonstrated that we could effectively cache real models, real-world deployments would be much larger than our testbed, with many more models and much greater resources. Therefore, we use a simulation to compare the impact of penalty-aware eviction policies while scaling to many thousands of models.

For this simulation, we used a *quantiles-r* workload, which has the most popular models but also has the highest penalty.



(a) Practical Algorithm **(b) Oracle Algorithm**

Fig. 8: Comparison of cache eviction policies in simulation on *quantiles-r*. We show the impact of different eviction policies in simulation with cache sizes ranging from 4GB to 2048GB by powers of 2. We see that penalty-aware eviction policies, namely *CremeBrulee*, outperform policies that only consider time until next usage.

This has two effects. First, it forces eviction policies that are penalty-aware to choose between popular and costly models when evicting models. Second, for policies that do not consider reload penalty, it removes any correlation that they may have relied on for other workloads.

We see in Figure 8a that at all cache sizes *CremeBrulee* outperforms *popularity*. This is especially apparent at lower cache sizes when *CremeBrulee* is markedly lower than the alternative algorithms. At 128GB *CremeBrulee* is 16.62% better than *popularity*, with an average improvement of 7.6%. This good performance is due to *CremeBrulee* being aware of the trade-offs between penalty and popularity and prioritizing models that demonstrate both. We observe that *popularity* actually performs worse than random at many cache sizes. This is due to its difficulty in selecting models when it is only aware of one component of the overall load time. Instead, by considering penalties of reloading, *CremeBrulee* achieves better performance.

In Figure 8b we see similar effects when we have access to future knowledge where we see that *CremeBrulee-oracle* improves over *BeladyMIN* at all points. For example, at a cache size of 128GB, we see that *CremeBrulee-oracle* improves over its non-penalty-aware alternative by 12.15% and an overall average improvement of 6.19%. Further, we see that both oracle and non-oracle versions perform much better than *random*. *CremeBrulee* improves performance over random by up to 36.34%, while *CremeBrulee-oracle* improves performance by up to 56.47%.

Therefore, we see that making eviction policies penalty-and size-aware can dramatically improve performance over policies that don't consider this. This can lead to a large decrease in model-load time.

VII. Discussion

The ability to efficiently serve and manage deep learning models requires rethinking and adapting existing resource management techniques. In this paper, we characterize the need and evaluate the benefit of employing model-level caching—an essential aspect in trading-off resource utilization and inference performance. Other orthogonal avenues such as redesigning deep learning models, heterogeneous hardware optimization are left for future work. Below, we discuss potential directions one could pursue to further improve model-level caching.

When designing our eviction policy *CremeBrulee*, we focus on exponentially distributed workloads—a pattern that is widely observed for web requests [9]. However, real-world deep learning inference workloads may display different access patterns. Access to real-world inference workloads may allow for accurate characterization and modeling to provide insights for designing model-level caching policies. With access to real-world workloads, one potential approach is to using ML-based predictors such as LSTM models [40] and other deep learning models [37]. In conjunction, one could also investigate pre-fetching policies to mitigate the high cache miss penalty [24]. On a different aspect, one can also consider beyond CPU-based caching, especially with the increasingly heterogeneous hardware deployment at data centers. However, today, concurrently running models on the same GPU device still faces multiple challenges, including hard-to-predict performance and high context switch overhead [17, 32]. Designing suitable deep learning specific mechanisms can further promote the caching policy design space.

VIII. Related Works

Characterization of Deep Learning Models. By analyzing the performance of deep learning models, the execution latency can be reduced. Some works [12, 17] approach this by proposing alternative execution patterns that improve performance. Others, such as Neurosurgeon [26] use characterizations of models to make decisions about how to partially execute models on-device, reducing response latency. Still other approaches use the advances to serve models with higher efficiency [3, 14, 18, 30]. In contrast, our work compares the use of these accelerators to general-purpose hardware in an emerging scenario where many deep learning models share the same underlying hardware.

CPU-Based Inference. Prior work has examined the potential of using CPUs for inference as an alternative to GPU-based serving. Industry papers [21, 36] report that they often use CPUs in practice due to their reduced cost compared to accelerators, which are more advantageous for training. Another use of CPUs is found in DeepCPU [41], where the authors leverage them to serve RNNs, which are difficult to serve with accelerators. Our work concretizes that CPU-based inference is lower cost compared to using GPUS, especially when serving a wide range of models where memory access is critical.

Caching Policies. Caching policies are widely used in systems, such as page caches [5], content delivery networks [6], and FaaS container caches [16, 37], with the key goal of improving performance. For example, FaaSCache investigates the effectiveness of a classic greedy-dual caching policy in container management [16]. Meanwhile, AdaptSize [6] considers a size-based probabilistic approach on cache admittance in a CDN. Because our approach requires loading models before execution, every model must be admitted and is thus based on

BeladyMIN [5] which evicts system pages from the cache, as opposed to content caching systems, which can serve objects without them entering the cache. Unlike page caching where managed in-memory objects are of the same size, in our work, we consider the heterogeneity of the deep learning models when devising the utility function—used as a basis for evicting models. For mobile-based inference, both results caching [19, 39] and memory deduplication [20] have been proposed to speed up inference execution. However, these approaches rely on similar inputs and data pipelines, which becomes increasingly unlikely as the number of unique end-users increases.

IX. Conclusion

In this paper, we looked at how to effectively manage deep learning models—each with different popularity—at the edge. We argued that with the unprecedented popularity of DL models, developers/companies who wish to embed DL models to their existing applications can benefit from having access to deployment solutions, similar to what is being provided in today's content delivery network for web objects. In other words, we envision a deep-learning serving infrastructure where many models can share the underlying hardware resource, in a cost-effective and performant manner. Designing and implementing such infrastructures require innovations in many aspects, including DL-specific hardware support, DL serving frameworks, and DL-aware resource management policies.

We took the first step toward this vision and tackled the challenges in multiple aspects. We first demonstrated that when faced with a large number of models to host, serving models with CPUs, rather than GPUs, is both more cost-efficient and better utilization of resources. To address the lack of real-world traces problem for evaluating the model-level caching policies, we developed an empirical methodology for deriving the workload from an open-source FaaS trace. We proposed a simple cache eviction policy called *CremeBrulee*, and to understand its effectiveness, we further used the above-mentioned trace to compare its performance to several baselines. In a testbed, we saw a reduction in memory usage by half with a modest increase in model load delay; in simulation, we saw a reduction in model load delay by 36.34% over non-penalty aware eviction policies.

Acknowledgement

We thank all anonymous reviewers for their insightful feedback which helped improve this paper. This work is supported in part by National Science Foundation's support under grants CNS-#1755659 and CNS-#1815619.

References

[1] "Nvidia embedded systems for next-gen autonomous machines." [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

[2] "Machine learning for business: using amazon sagemaker and jupyter," 2020. [Online]. Available: https://aws.amazon.com/sagemaker/

[3] "Nvidia triton inference server," 2021. [Online]. Available: https://developer.nvidia.com/nvidia-triton-inference-server

[4] S. Balan and J. Otto, *Business Intelligence in Healthcare with IBM Watson Analytics*. CreateSpace Independent Publishing Platform, 2017.

[5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, 1966.

[6] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *NSDI 2017*.

[7] A. Bhattacharjee *et al.*, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *IC2E 2019*. IEEE.

[8] S. Bianco *et al.*, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, 2018.

[9] L. Breslau *et al.*, "Web caching and zipf-like distributions: evidence and implications," in *INFOCOM 1999*.

[10] CakeLab. https://github.com/cake-lab/CremeBrulee.

[11] T. Capes *et al.*, "Siri on-device deep learning-guided unit selection text-to-speech system." in *INTERSPEECH*, 2017.

[12] S. Chetlur *et al.*, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[13] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[14] D. Crankshaw *et al.*, "Clipper: A low-latency online prediction serving system," in *NSDI 2017*.

[15] A. Eivy and J. Weinman, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, 2017.

[16] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," *ASPLOS 2021*.

[17] G. Gilman *et al.*, "Demystifying the Placement Policies of the GPU Thread Block Scheduler for Concurrent Kernels," *Performance 2020*.

[18] A. Gujarati *et al.*, "Serving dnns like clockwork: Performance predictability from the bottom up," *OSDI 2020*.

[19] P. Guo *et al.*, "FoggyCache: Cross-device approximate computation reuse," in *MobiCom 2018*.

[20] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *ASPLOS 2018*.

[21] K. Hazelwood *et al.*, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *HPCA 2018*.

[22] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR 2016*.

[23] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *IC2E 2018*.

[24] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *SIGARCH Comput. Archit. News*, 1997.

[25] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA 2017*.

[26] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *SIGARCH Comput. Archit. News*, 2017.

[27] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, 2018.

[28] S. S. Ogden and T. Guo, "MDINFERENCE: Balancing Inference Accuracy and Latency for Mobile Applications," in *IC2E 2020*.

[29] S. S. Ogden, X. Kong, and T. Guo, "Pieslicer: Dynamically improving response time for cloud-based cnn inference," in *ICPE 2021*.

[30] C. Olston *et al.*, "Tensorflow-serving: Flexible, high-performance ml serving," in *Workshop on ML Systems at NIPS 2017*, 2017.

[31] V. J. Reddi *et al.*, "Mlperf inference benchmark," *ISCA 2020*.

[32] F. Romero *et al.*, "Infaas: Automated model-less inference serving," *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[33] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *IJCV 2015*.

[34] A. W. Services. Aws neuron - amazon web service. [Online]. Available: https://aws.amazon.com/machine-learning/neuron/

[35] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," *USENIX ATC 2020*.

[36] J. Soifer *et al.*, "Deep learning inference service at microsoft," in *OpML 2019*.

[37] Z. Song *et al.*, "Learning relaxed belady for content distribution network caching," in *NSDI 2020*.

[38] C. Szegedy *et al.*, "Rethinking the inception architecture for computer vision," in *CVPR 2016*.

[39] M. Xu *et al.*, "DeepCache: Principled cache for mobile deep vision," in *MobiCom 2018*, New York, NY, USA, 2018.

[40] C. Zhang *et al.*, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *USENIX ATC 2019*.

[41] M. Zhang *et al.*, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *USENIX ATCG 2018*.

[42] B. Zoph *et al.*, "Learning transferable architectures for scalable image recognition," in *CVPR 2018*.