Tetris: Memory-efficient Serverless Inference through Tensor Sharing

Jie Li, Laiping Zhao, and Yanan Yang, *College of Intelligence & Computing (CIC), Tianjin University, and Tianjin Key Lab of Advanced Networking (TANKLAB);* Kunlin Zhan, *58.com;* Keqiu Li, *College of Intelligence & Computing (CIC), Tianjin University, and Tianjin Key Lab of Advanced Networking (TANKLAB)*

https://www.usenix.org/conference/atc22/presentation/li-jie

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

TETRIS: Memory-efficient Serverless Inference through Tensor Sharing

Jie Li
College of Intelligence & Computing (CIC), Tianjin University
Tianjin Key Lab of Advanced Networking (TANKLAB)

Laiping Zhao *
CIC, Tianjin University
TANKLAB

Yanan Yang
CIC, Tianjin University, TANKLAB

Kunlin Zhan
58.com

Keqiu Li
CIC, Tianjin University, TANKLAB

Abstract

Executing complex, memory-intensive deep learning inference services poses a major challenge for serverless computing frameworks, which would densely deploy and maintain inference models at high throughput. We observe the excessive memory consumption problem in serverless inference systems, due to the large-sized models and high data redundancy.

We present TETRIS, a serverless platform catered to inference services with an order of magnitude lower memory footprint. TETRIS's design carefully considers the extensive memory sharing of runtime and tensors. It supports minimizing the runtime redundancy through a combined optimization of batching and concurrent execution and eliminates tensor redundancy across instances from either the same or different functions using a lightweight and safe tensor mapping mechanism. Our comprehensive evaluation demonstrates that TETRIS saves up to 93% memory footprint for inference services, and increases the function density by 30× without impairing the latency.

1 Introduction

Serverless computing has seen its popularity explode in recent years, due to the ease of use, cost efficiency, resource management-free, and autoscaling advantages. Serverless inference, i.e., deploying deep learning (DL) inference services atop a serverless platform, is continuously adopted in the backend of the internet of things (IoT), mobile and web applications [1,6,11,65,74]. Some typical explorations include the Amazon Alexa [5], Facebook Messenger bot [49], and Netflix media transformation [38].

Inference services are commonly memory-intensive, consuming a substantial amount of memory throughout the computation. If performing inference in a serverless environment, we find that the current serverless platforms (e.g., AWS Lambda [2]) do not support inference computation well. First, *they cannot accommodate large inference models*. For example, AWS Lambda limits the memory footprint of functions to ≤10GB [2], whereas the recent MT-NLG language model [36] even needs 2TB memory to load its 530 billion parameters. Second, *these platforms cause a significant waste of memory resources*. The "*one-to-one mapping policy*" of request to function instance by commercial platforms [2] introduces significant memory redundancy. The memory footprint of language runtime, libraries, and even tensors is repeated across function instances. On the other hand, serverless functions are usually *short-lived*. While the CPU processing of inference requests takes only a short time (e.g., 75% execute for less than 10 seconds in Microsoft Azure [60]), the memory is kept occupied without working for long due to the early reservation or keep-alive caching after completion (e.g., 15-60 minutes in AWS Lambda) [20,58,60]. Hence, how to improve the memory efficiency of serverless inference has become a crucial problem.

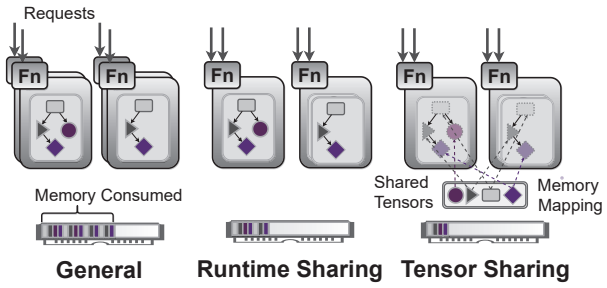


Figure 1: Schematic overview of memory consumption in serverless inference systems.

To reduce the memory footprint, prior works have proposed the *runtime sharing* method [16] (Figure 1), i.e., multiple requests share the same function instance runtime with increased instance concurrency. In serverless inference, *runtime sharing* can also be enabled through *batching*, which aggregates multiple inputs into a single batch submission for more efficient execution [1,71,74]. We study the memory footprint of serverless inference and observe that the *tensor redundancy* problem (i.e., tensors in the computational graphs of infer-

*Corresponding author: laiping@tju.edu.cn.

ence models are highly duplicated across function instances) still severely degrades memory efficiency. Such duplication is commonly caused by either replicated function instances or identical parts generated by pervasive pre-trained models or transfer learning [15, 52, 61, 62, 73]. We summarize 768 machine learning (ML) models utilized by *58.com*, the China's largest local life service website, finding that tensor redundancy exists in 67.5% of natural language processing models, 26.7% of image classification models, 30.1% of recommendation models.

The *tensor redundancy* problem can be mitigated through operating system kernel-level page merging methods (e.g., [3, 33, 47]), which scan for duplicate content in the memory and merge the content into a single physical page. However, they incur nonnegligible scanning overhead (e.g., 5 minutes scanning time) and thereby work well only for deduplicating fairly static memory pages, contradicting with *short-lived* nature of serverless functions. Since the scanning process occurs after applications are loaded, they can neither accelerate the function startup nor solve the *startup failure* problem caused by the out-of-memory (OOM) error. Moreover, their implementations require modifications to the operating system kernel, which is heavy and may introduce risks of side-channel attacks [42].

In this paper, we explore improving the memory efficiency of serverless inference system through both *runtime-level sharing* and *tensor-level sharing*. As illustrated in Figure 1, *the colocated function instances share tensor-memory pages with identical content*. Tensor sharing introduces several challenges that need to be addressed. (1) *Non-harming performance*: The sharing method should not impair the inference latency. (2) *Safe sharing*: The sharing process should not introduce data leakage risks. (3) *First-time sharing*: The sharing method should start working as long as the inference functions are activated, to accelerate the function startup and reduce the OOM errors. (4) *Low overhead*: The sharing method should be lightweight for easy integration with serverless frameworks, and transparent to tenants.

To overcome these challenges, we build TETRIS, a domain-specific serverless platform that caters to DL inference as backend-as-a-service (BaaS) offerings with high memory efficiency. For example, *after using* TETRIS*, the memory footprint of the LaBSE model (a multilingual BERT embedding model [18]) reduces from 1.97GB to merely 141.7MB, and the instance density also increases from 64 to 911 per 128GB-server without any modifications to the model.* TETRIS provides a complete solution for the memory bottleneck problem in serverless inference through automating runtime sharing, tensor sharing, memory reclaiming, and instance scheduling. It enables tensor sharing through a user-space page deduplication method, which has no pollution to the underlying systems and is thus incredibly lightweight and easy to implement. It is also highly efficient and supports to guarantee the Service Level Objectives (SLOs). DL developers would significantly benefit from TETRIS, as it can save tremendous memory as well as monetary costs, or the freed memory can be reused for other purposes like caching.

Our contributions can be summarized as follows:

- We observe the *tensor redundancy* problem in serverless inference systems and propose the corresponding *tensor sharing* idea for memory efficiency.
- We design a lightweight, user-space tensor mapping-based sharing method, eliminating the tensor redundancy problem in serverless inference systems.
- We implement a prototype system of TETRIS, which is built with the open-source OpenFaaS [17] and TensorFlow Serving [51], supporting memory sharing, memory reclaiming, and instance scheduling.
- We extensively evaluate TETRIS using a comprehensive set of benchmarks and production workloads. The experimental results reveal that TETRIS can save up to 93% of memory and increase the function density by 30×, compared to the state-of-the-art approaches.

The rest of the paper is structured as follows. We study the data redundancy problem in serverless inference (§2), and use the findings to guide the design (§3) and implementation (§4) of TETRIS. We then evaluate the performance of TETRIS (§5), discuss related work (§6), and conclude (§7).

2 Background and Motivation

2.1 Inference on Serverless

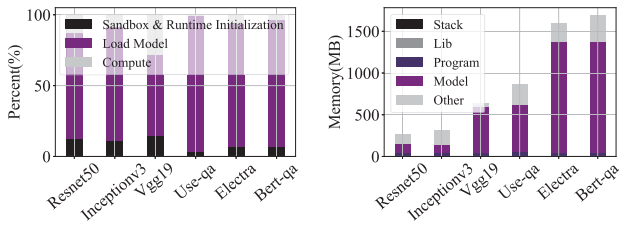
The use of machine learning is divided into two phases: training and inference. While model parameters are continuously updated throughout the training phase, they remain fixed during the inference phase. In DL frameworks (e.g., TensorFlow), models are organized as computational graphs, and data in the graph are stored as *tensors*. The nodes represent operators or variables and edges denote the direction in which the tensor flows. In particular, the variable nodes of parameters and intermediate tensors produced on edges consume the most memory. The parameterized tensors may also be identical across different models, due to the pervasive usage of pretraining and transfer learning. As an example, we investigate the VGGish-based audio classification models that assist the telephone customer service in the local life service website, which are used for detecting various noises, non-human voices, dialects, etc. We find that these models share the bottom embedding layers, accounting for over 90% of the model parameters.

In serverless inference, each deep learning model is typically deployed into a separate function instance (e.g., a Docker container [45]). The function instances automatically scale in or out according to the fluctuation of requests. In the case of scaling out, an identical instance is created and its runtime, library, and model parameters are loaded into the memory repeatedly.

2.2 Motivations

To improve memory efficiency, we study the memory footprint of typical serverless inference models. The benchmarks are selected from TensorFlow Hub [22], with an average of 6.8k downloads. All of them are implemented and deployed on an OpenFaaS [17] testbed (Table 3).

***Observation #1**: Memory-intensive startup: The loading of massive model parameters dominates the inference function instance startup.*



(a) Request processing time (b) Memory breakdown

Figure 2: Request processing time and memory breakdown for various inference models.

The DL inference functions require numerous model parameters to load at startup, typically saved in a serialized model file. In particular, before the function can conduct the inference, the model loading thread continuously reads the parameter tensors from the disk and then deserializes and populates them into the corresponding node in the computational graph. With the memory page cache enabled to accelerate the startup, we measure the time of the three phases during request processing (Figure 2(a)): *sandbox and runtime initialization*, *model loading*, and *inference computation*. The *function startup* time is significantly longer than the *inference computation* time. Especially in the Bert-QA model case (with 1,300MB parameters), only 3.73% of the time is spent on computation. Even for the small model of Resnet50 (with 102MB parameters), the inference computation time still only accounts for 13.37%. If merely considering the startup, the majority of time is further spent on loading model parameters. For example, the ratios exceed 89% for both Electra and Bert-QA.

***Observation #2**: Memory-intensive computing: The inference computation requires a substantial amount of memory, with the model parameters consuming the most.*

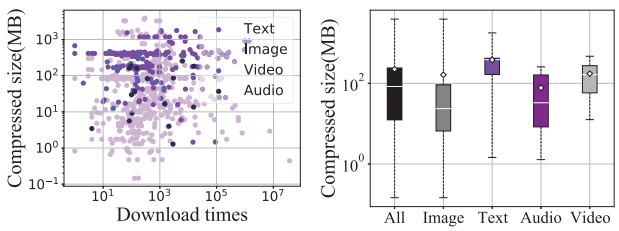


Figure 3: Analysis of the compressed model size and download times of deep learning models from TensorFlow Hub.

The memory footprint of an inference process can be divided into the following sections: *program code*, *libraries*, *the model* (for loading parameters in the form of tensors), *the function call stack*, *intermediate tensors* (i.e., generated at runtime) and *the network buffers* (i.e., allocated for receiving function requests). The *model* itself consumes the majority of memory.

We measure the memory consumption of different types of inference models at runtime and present the results in Figure 2(b). The model parameters occupy a major portion of all memory consumption (e.g., for VGG19, storing the parameter tensors accounts for more than 93% of the total memory consumption). We also compile a list of 625 machine learning models from TensorFlow Hub [22] and analyze their compressed model file sizes (Figure 3). In addition, 42% of the compressed model file sizes surpass 100MB, such as the popular text-embedding model *universal sentence encoder* (downloaded 1.4M times), which has a compressed model size of 0.89GB and consumes 1.72GB memory at runtime. Typically, the text processing models employing large embedding layers are substantially larger than the image, audio, and video processing models.

***Observation #3**: Runtime redundancy: The inference runtime is replicated in memory due to the multilaunched instances. While both concurrent execution and batching enable runtime sharing, how to use them in combination is challenging.*

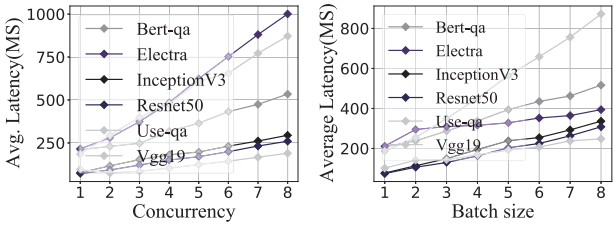
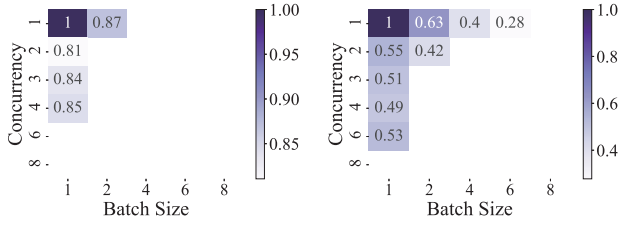


Figure 4: Average inference latency increases over batch size and concurrency.

Multilaunched instances lead to duplicated runtime memory consumption. The memory footprint can be reduced by packing multiple requests into an instance for sharing the runtime resources (e.g., ML model, framework, and network buffers.). In serverless inference, runtime sharing can be implemented in two ways: *batching* and *concurrent execution*. Given the same benchmarks and resource configurations, we evaluate the inference latency under both implementations. Figure 4 illustrates that both implementations increase the latency significantly: *batching* increases the latency due to the increased computing load, whereas *concurrent execution* increases latency due to the increased resource contention among threads, i.e., as concurrent requests compete for the computing threads in TensorFlow, the interleaved computation of operators increases the average latency.

We further measure memory consumption under various combinations of *batching* and *concurrency* (Figure 5). In par-

ticular, we generate a constant 200 requests per second (RPS) toward DenseNet169 [30] and Lstm [27] and specify their latency SLOs at 200ms and 30ms, respectively. After omitting combinations that do not guarantee the SLOs, we find that solely increasing either the concurrency or batch size leads to sub-optimal memory efficiency. For DenseNet169, increasing only the batch size quickly causes SLO violations due to the long waiting time in the batch queue, whereas increasing the concurrency does not. For the Lstm model, a larger batch size ($batchsize = 6$) achieves the least memory consumption under the SLO guarantee. As various combinations of batch size and concurrency may lead to completely different memory efficiency, selecting the best among them is essential.



(a) DenseNet169 (b) Lstm

Figure 5: Normalized memory consumption of function instances under various combinations of batch size and concurrency.

Observation #4: *Tensor redundancy: The tensors of the constants, model parameters, are extensively replicated across function instances.*

Besides the runtime redundancy, parameterized tensors are also replicated across instances from the same function, i.e., *tensor redundancy*. We summarize 768 DL models at *58.com*, finding that tensor redundancy also commonly exists across distinct functions. There are presently 27 business lines in *58.com*, including jobs, housing, vehicles, cellphones, home services, resumes, etc. These businesses have common demands on DL inferences (e.g., image, text, video, and audio processing) but with different datasets (e.g., pictures of houses, cars, people, and smartphones). Primarily, there are two scenarios that cause tensor redundancy across distinct functions [15, 40, 52, 61, 62, 73]:

(1) *Multi-versioned functions.* In production scenarios, a DL model is frequently reused directly in various business contexts. As online web services have strict latency requirements (e.g., $<$ 100 milliseconds), models with pipeline dependencies are commonly deployed together within a function to avoid the excessive network communications. In such cases, although these pipelines vary across businesses, tensors within the shared models stay identical. For example, the Optical Character Recognition (OCR) pipeline and the image moderation pipeline share the same text line identification and recognition models (e.g., Resnet), whereas the image moderation pipeline incorporates an extra model for keyword detection. Moreover, DL models are also commonly deployed with specific pre- and post-processing modules for processing data in different formats, thus generating multi-versioned model pipelines.

(2) *Pretraining & transfer learning.* It refers to training a model with massive datasets for one task, where the learned parameters could be reused in other related tasks. In such way, new models could significantly benefit from prior knowledge to accomplish new tasks rather than from scratch. At *58.com*, pre-trained models or transfer learning are widely used for reducing development costs. For example, in the house leasing business, webchats are recorded and different DL models are called to identify the status of the landlord (e.g., whether the house was leased) and tenant (e.g., a genuine renter or housing agent) respectively. In the telephone customer services, there are also distinct models to assess the recruitment, rental and housekeeping intentions of users, respectively. These models with similar tasks are all built from a pretrained Bert model. Instead of fine-tuning all parameters of a model, it is common to directly reuse partial parameters and only fine-tune a small set of model layers for two reasons: (i) It is able to significantly decrease the training overheads of similar downstream tasks, enabling rapid development; (ii) Fine-tuning the whole model may result in overfitting if the target dataset is small and the number of parameters is huge [23, 28, 34, 44, 48]. It even improves the accuracy in case of insufficient training samples (e.g., sporadic noises). For example in the advertising business, a Resnet50 model with only the top layers retrained achieves an accuracy of $>$ 98% in classifying QR codes.

It is highly empirical to determine which layers should be reused in practice. For tasks like sentiment classification, reusing the initial 12-16 layers of Bert even outperforms fine-tuning all layers on the SST-2 dataset [39]. The VGGish-based audio classification model for detecting noises with retained bottom embedding layers also achieves an accuracy of $>$ 93%. Table 1 summarizes the representative deep learning models used for image, text, and audio processing in our website, as well as their applications, potential redundant parts and redundancy ratios.

Table 1: Tensor redundancy at *58.com*.

Application Domains	Image	Text	Audio
Representative Models	Resnet50 EfficientNet MobileNet	Bert Roberta Albert GPT	VGGish
Applications	Advertising Housing Secondhand Trading	Reading Extraction Text Summary Text Classification	Audio Classification
Redundant Part	Bottom Layers	Embeddings Middle Layers	Embeddings
Redundancy Ratio	>70%	>31%	>90%

Observation #5: *Cache redundancy: Both the runtime and tensor redundancy are exacerbated by the widespread usage of in-memory caches in both the serverless platform and DL library.*

To alleviate the cold-start overhead, serverless platforms [20,60] often keep function alive and warm after the execution

is finished (a.k.a. caching). Caching exacerbates the runtime and tensor redundancy problem.

Moreover, caching also exists in DL frameworks. For example, TensorFlow makes use of high-performance computing libraries, such as MKL-DNN [32] to accelerate the execution of computational graphs. The internal tensor representation of TensorFlow is not identical to MKL-DNN when performing operations such as convolutions; thus, the convolutional kernel data in TensorFlow format must be converted into MKL-DNN format. These converted parameters are usually cached in memory for subsequent usage [4] (e.g., 87.7% of parameters are cached for Resnet50). Hence, these caches also increase tensor redundancy.

2.3 Implications

Due to the memory-intensive nature of startup and computing in DL inference (**Observation 1** and **Observation 2**), the main memory can be easily and massively harvested in serverless inference systems. Reducing the memory footprint of serverless inference needs to be addressed immediately. We study the in-memory data redundancy in serverless inference systems, and observe that the redundancy problem primarily derives from three aspects: *runtime redundancy* (**Observation 3**), *tensor redundancy* (**Observation 4**) and *cache redundancy* (**Observation 5**).

Table 2: Comparision of existing systems.

		KSM [33]	Photons [16]	INFless [71]	TETRIS
Func.	Runtime sharing	✓	concurrency	batching	✓
	Tensor sharing	✓	✗	✗	✓
	Cache sharing	✓	✗	✗	✓
	No-harming perf.	✓	limited	limited	✓
	First-time sharing	✗	✓	✓	✓
N.F.	Running level	kernel	container	container	container
	Imple. difficulty	high	low	low	low

Prior works [3, 16, 33, 47, 71] have proposed reducing the in-memory data redundancy through *page merging* or *runtime sharing* (Table 2). The *page merging* methods, such as KSM [3, 33, 47], support memory-saving deduplication through searching and merging equal physical pages of memory. However, it is a Linux kernel-level feature; therefore, its implementation is relatively difficult. It also does not support *first-time sharing* during the startup of function instances. *Runtime sharing* can be enabled through either *concurrent execution* (e.g., Photons [16]) or *batching* (e.g., INFless [71]). However, they only partially solve the *runtime redundancy* problem and their optimal combination is still worth further exploration. Moreover, the *tensor redundancy* and *cache redundancy* still exist and severely degrade the memory efficiency.

3 System Design

In this section, we present the design of TETRIS.

3.1 System Overview

The insight of TETRIS is that memory efficiency can be improved through the combined optimization of *tensor sharing* and *runtime sharing*. Figure 6 illustrates the overall architecture of TETRIS, which enables runtime sharing through its *scaling and scheduling* engine and supports tensor sharing through the *agent* in each container and the *tensor store* in each server. When a developer submits a trained DL model to the platform, TETRIS extracts the tensor information (including tensor sizes and hash values) and profiles the inference latency under various batch sizes and concurrency settings. The tensor information is used for directing *tensor sharing*, and the profiles are used by the *scaling and scheduling* module to determine the best *runtime sharing* configuration.

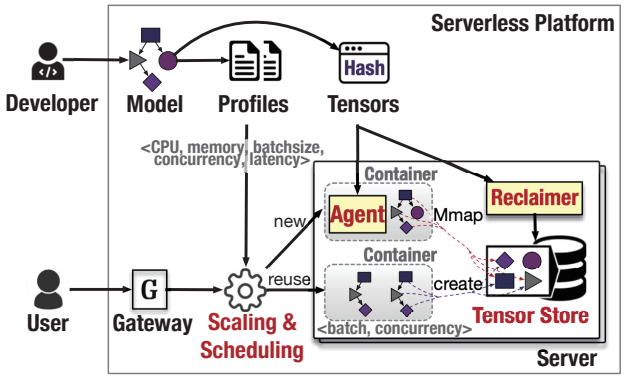


Figure 6: An overview of TETRIS.

When the user submits requests to the *gateway*, the *scaling and scheduling* engine first decides whether to scale out new instances or reuse existing ones according to the workloads. First, in the case of scaling out instances, the engine derives the batch size and concurrency configurations, minimizing the memory footprint under the SLO guarantee. Then, the new instances are launched on servers with maximum *tensor similarity*. TETRIS supports *first-time sharing* of tensors through a special *agent* in each sandbox. The *agent* parses the computational graph of the model and reads the *hash* value of the tensors. Then, the agent checks whether the *tensor store* has already stored it. If so, the *agent* only maps the memory address of the tensor to its local process using the syscall of *Mmap*, and the reference number of the tensor is increased by 1. Otherwise, the agent loads the tensor from the model file into memory directly and creates a new item in the *tensor store*. Each *agent* is only active during the startup of a new instance. In contrast to DL training where tensors may be updated every iteration, tensors in inference are fixed during its service time. Hence, we set them to be only readable to ensure safe memory access across functions. Second, in the case of reusing existing instances, requests are forwarded to instances following the load balancing principle.

When the workload decreases, the *scaling and scheduling* engine also selectively releases the least-loaded instances without violating the SLO. After release, the reference number of its tensors decreases by 1. In each server, there is a *memory reclaimer* periodically validates the reference number of each

tensor and reclaims the memory pages with reference $= 0$ so that the released functions no longer occupy memory. A *keep-alive* policy can be adopted to avoid reclaiming oscillation.

3.2 Scaling and Scheduling

The *scaling and scheduling* engine relies on the model profile and requests per second (RPS) to make the scaling decision.

Profiling: We develop an automatic profiler that measures the inference latency of each model under various configurations. Specifically, we define the profile of each model as a 5-tuple $\langle c,m,b,p,l\rangle$, where $c \in C$ denotes the number of allocated CPUs, $m \in M$ denotes the memory configuration for the inference, $b \in B$ indicates the maximum batch size for that instance to process requests, $p \in P$ represents the number of concurrent inference threads, and l represents the inference latency under previous configurations. Since current serverless platforms typically use CPUs for function computation and DL inference on CPUs typically uses small batches (e.g., $B = \{1,2,4\}$) and low concurrency (e.g., $P = \{1,2,3,4\}$) to achieve low latency. Thus, we profile only these combinations to narrow the profiling space. Since allocating excessive memory does not lower inference latency, we assign each configuration the bare minimum. Finally, we obtain $n = |C||M||B||P|$ 5-tuples characterizing the model, which are stored in the database.

In the profiling phase, TETRIS also computes and stores the hashes of tensors using the cyclic redundancy check (CRC) code, which is utilized in checking the memory status during *tensor sharing*.

Runtime-shared scaling: The scaling engine monitors the real-time RPS and judges whether the existing instances are sufficient to serve these requests. If not, it dispatches parts of requests to existing instances and launches new instances to process the residual ones. Given the model profile and residual RPS (denoted by R), the scaling engine explores the optimal configurations of new instances, to minimize memory usage while guaranteeing their latency SLO. We define an integer variable $x_i, \forall i \in [1,..,n]$, which indicates the configuration i is adopted for x_i new instances. Hence, the optimal configuration can be found by solving the following integer programming problem:

$$minimize: \quad \sum_{i=1}^{n} m_i x_i \tag{1}$$

$$l_i \le t_{slo}, \ \ \forall i \wedge x_i \ge 1 \wedge b_i = 1 \tag{2}$$

$$l_i \le t_{slo}/2, \ \ \forall i \wedge x_i \ge 1 \wedge b_i > 1 \tag{3}$$

$$\textstyle\sum_{i=1}^{n} x_i b_i p_i / l_i \ge R, \ \ \forall i \tag{4}$$

$$x_i \in \mathbb{N} \tag{5}$$

Objective (1) defines the memory occupied by the instances. Constraints (2) and (3) ensure that the latency SLO is satisfied: For $b_i = 1$ (i.e., no batch queue exists for an inference thread), the inference latency l_i should not exceed the SLO; For $b_i > 1$ (i.e., requests must wait in a queue to saturate the batch), we have $t^i_{wait} + l_i \le t_{slo}$, where t^i_{wait} denotes the waiting time in the batch queue. Suppose the RPS distributed to instance i is r_i, then we have $t^i_{wait} = b_i p_i / r_i$. Since $r_i \le b_i p_i / l_i$ must be established (otherwise, if the request arrival rate exceeds the batch processing rate, requests will be dropped), we obtain $l_i \le t_{slo}/2$. Constraint (4) ensures that the residual RPS can be fully processed by the new instances. Constraint (5) refers to the domain constraint.

Algorithm 1: DTS Algorithm

Input:
Requests R; profile $O = \{< c,m,b,p,l >\}$; t_{slo};
Output:
S: the set of selected instance configurations;

```
1  S = ∅;
2  sort O in descending order of (b_i p_i)/(l_i m_i), ∀i ∈ [1..n];
3  while R > 0 do
4      for each configuration o_i ∈ O do
5          if b_i = 1 ∧ t_exec(c_i, b_i, p_i) > t_slo then
6              continue;
7          if b_i > 1 ∧ t_exec(c_i, b_i, p_i) > t_slo/2 then
8              continue;
9          R ← R − (b_i p_i)/l_i;
10         S ← S ∪ {o_i};
11         break;
```

This problem can be reduced to the NP-Complete knapsack problem [54]. Hence, we design a heuristic algorithm, called Decreased Throughput Selection (DTS) to determine the instance configuration efficiently. Algorithm 1 presents the details. In line 2, we sort the configurations in O in descending order by normalized throughput. Then, we greedily select the configuration with higher normalized throughput as long as the latency SLO is satisfied (lines 3-11). The DTS algorithm can also be easily extended to balance CPU usage by normalizing the throughput with $m_i + \alpha c_i$, where α denotes the equilibrium factor.

Scheduling: Once the scaling engine has determined the configurations of new instances, the scheduler deploys them to the appropriate servers. To reduce cluster-level memory consumption through *tensor sharing*, TETRIS always tries to dispatch them on servers with maximum *tensor similarity* (denoted by θ). Specifically, for instance i, TETRIS first filters out servers that do not meet resource requirements (e.g., CPU and memory). Then, it derives the similarity value between a model i and a server j using $\theta_{ij} = Mem(T_i \cap T^j_{store})/Mem(T_i)$, where T_i represents the set of tensors in instance i, T^j_{store} represents the set of tensors already stored in server j, and $Mem(T_i)$ represents the aggregated memory of tensors in T_i.

3.3 Tensor Sharing

After an instance is scheduled on a server, TETRIS first launches a sandbox (e.g., container) on the target server to accommodate it. Then, an *agent* is activated in the sandbox to load the model into the main memory. For a tensor that has

previously been loaded into the *tensor store*, the *agent* maps its memory address to the instance. Otherwise, it creates a new item for it in the *tensor store*.

```
1  Status LoadTensor(Tensor& tensor,TensorReader& reader) {
2      // Get tensor hash value.
3      std::string tensor_hash = GetHash(reader,tensor);
4      // Get or create tensor lock in
5      // Shared Tensor Store atomically.
6      TensorLock lock = CreateOrGetTensorLock(tensor_hash);
7      // Obtain ownership of a tensor lock.
8      lock.Lock();
9      // Check if the tensor in Shared
10     // Tensor Store already exists.
11     if(!TensorExists(tensor_hash)) {
12         // Allocate the tensor memory in
13         // Shared Tensor Store and load
14         // the model parameters.
15         CreateTensor(reader,tensor,tensor_hash);
16     } else {
17         // Mapping already existing tensor
18         // memory from the Shared Tensor Store.
19         MmapTensor(tensor,tensor_hash);
20     }
21     // Release the lock.
22     lock.Unlock();
23     return Status::OK();
24 }
```

Listing 1: Simplified code snippet for loading tensors.

Agent: The agent is a lightweight, user-space process for sharing tensors across function instances. Whenever loading a model into memory, the *agent* reads the computational graph metadata stored in the model file and parses the tensors that need to be loaded into memory. As illustrated in Listing 1, the *agent* reads the hash value of a tensor using the interface *GetHash()* and checks whether the *tensor store* has already had the tensor. If the *TensorExists()* returns *FALSE*, the *agent* allocates memory for the tensor and puts it in the tensor store. Otherwise, it calls *MmapTensor()* to map the memory address of the existing tensor to the model. The *agent* maintains a loading queue for tensors. To ensure that the *tensor store* correctly behaves when it is accessed by multiple concurrent *agents*, we employ locks: an *agent* must acquire a lock before writing to the store (line 8) and release it after writing completion (line 22).

The *agent* is only active during the startup of an instance. It does not require modifications to the underlying operating system or virtualization layer and does not degrade inference performance.

Tensor store: The *tensor store* holds all tensor memory (parameters, constants, etc.) that need to be shared across all function instances on a server. Every function instance on the server can access tensors in the *tensor store*. Each tensor is uniquely identified by a hash value, calculated from the tensor content and dimensions. We use hash values because they are independent of the models and underlying frameworks. There is also a corresponding lock for each tensor to ensure their safe operation of constructing, mapping, or reclaiming. The *tensor store* is initially empty and does not hold any tensors or locks. During the running of the system, the *agent* continuously adds tensors into it. Each tensor is associated with a *reference number* initialized with 1 after its creation. Whenever the *agent* adds a new mapping to an existing tensor, the *reference number* is increased by 1. Similarly, whenever an instance is released after completion, the *reference number* is decreased by 1. The tensor memory is reclaimed after the *reference number* is set to 0.

While the *tensor store* is shared by all function instances by default, TETRIS also supports building a *dedicated tensor store* for a subset of functions at the local server (e.g., functions belonging to the same tenant). The tensors in a *dedicated tensor store* cannot be accessed by functions without permission.

3.4 Memory Reclaiming

When a function instance is released, the tensor that it maps from the *tensor store* is not instantly deleted, as it may still be referenced by other instances. To appropriately reclaim the memory of shared tensors, TETRIS runs a *memory reclaimer* on each server, which periodically detects and reclaims the memory of tensors with *reference number* $= 0$.

The *reclaimer* also supports *tensor caching* policies, which keep tensors in the *tensor store* even after their *reference numbers* become 0. Currently, we have added two caching policies in the *reclaimer*: (1) *keep-alive window:* it is a timeout threshold to determine how long a tensor is kept alive; (2) *Least Recent Used (LRU):* it only keeps the recently or often-used tensors in the *tensor store* after the *tensor store* is full. The cached tensors can accelerate the startup of function instances. Although TETRIS only supports these two policies at this time, other caching policies [20, 60] can be easily integrated into it. When a tensor's memory is reclaimed, the *reclaimer* is also required to obtain its lock. To sum it up, Figure 7 depicts the lifecycle of tensors in TETRIS.

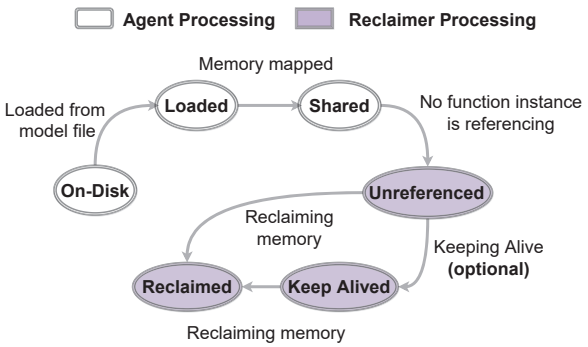


Figure 7: Overview of the tensor lifecycle in TETRIS.

4 System Implementation and Discussion

TETRIS is implemented in the open-source serverless platform OpenFaaS [17] with the DL inference framework TensorFlow Serving [22]. In particular, the *runtime sharing* ability is added to OpenFaaS by modifying its modules including *request dispatching*, *autoscaling*, *scheduling* and *instance creating*; The *tensor sharing* ability is implemented by modifying the TensorFlow Serving framework. We further implement

the *reclaimer* as a separate daemon. The entire system runs on Kubernetes. Overall, TETRIS's implementation introduces negligible pollution to the existing software stack except for the newly added module *reclaimer*.

Tensor store: Since maintaining a cluster-level global tensor store is costly due to frequent tensor access during inference and high network latency, TETRIS maintains a shared tensor store on each server for performance guarantee while minimizing cluster memory consumption through instance scheduling. The tensor store on each server is implemented as a shared memory region, which can be accessed by all *agents* and the *reclaimer* at the local server. Although shared memory can be enabled by Docker through setting the *–ipc=host* option at the container creation time, allowing all containers to share the host *ipc* namespace, this introduces significant risks of malicious activities or misoperations. Hence, we instead implement the shared memory by mounting a memory-based *tmpfs* [64], in which the tensor store is just a directory. Then, it could be mounted to each container during its creation time using command like *docker -v*. Tensors are stored as files under the mounted *tmpfs* directory and their hash values are set as the filenames. In this way, we can build multiple dedicated tensor stores flexibly, just by creating different mounted directories.

Agent: The *agent* is integrated into the existing loading process of the TensorFlow Serving system. In particular, we modify the *RestoreOp* interface and provide a new tensor memory allocator to manage the shared memory mappings using the *open* and *Mmap* syscalls. After a tensor is created and initialized, its memory pages are tagged as read-only to prevent modifications. File locks are leveraged to synchronize *agents*, avoiding manipulating tensors simultaneously. In the case of concurrently loading models, we pre-randomize the list of loaded tensors to reduce lock conflicts. Besides, we replace the *malloc* interface in the TensorFlow Serving framework with *tcmalloc* [59] to reduce the memory waste.

Scaling & scheduling: The scaling and scheduling engine is integrated into the *faas-netes* module of OpenFaaS. In particular, we modify its *autoscaling* module to implement the DTS algorithm. While OpenFaaS originally uses the default scheduler of Kubernetes, we also modify it using the *tensor similarity*-based scheduling algorithm. To enable both *runtime sharing* and *tensor sharing*, we directly create Kubernetes pods for instances and mount the *tmpfs* directory in the instance creating process.

Reclaimer: The *Reclaimer* is implemented as a Kubernetes DaemonSet. Although Linux provides *fuser* or *lsof* for identifying whether or not a tensor has been referenced by processes, these tools are highly inefficient. Instead, the *Reclaimer* retrieves the set of running functions on a server through Kubernetes' API, then infers the set of referencing tensors (denoted by T_{warm}) by querying the profile database. Then, the tensor set to be reclaimed (denoted by T_{cold}) can be derived using $T_{cold} = T_{all} \setminus T_{warm}$, where T_{all} represents the set of tensors residing in the Tensor Store. Such an implementation also ensures fault tolerance, i.e., unreferenced tensors can be detected even when *agents* crash.

GPU Inference: Although the current serverless platforms typically use CPUs for function computation, TETRIS is still effective for GPU inference since frameworks like TensorFlow usually keep a copy of tensors in CPU memory. TETRIS is also suitable for large models which cannot fit into GPU memory. As GPU memory could also be shared through *cudaIpc* [50] APIs, TETRIS can be extended to the GPU inference scenario by developing a specialized manager for maintaining GPU tensor mappings.

5 Evaluation

5.1 Methodology

Table 3: Experimental testbed configurations.

Machine Type	Type 1	Type 2
CPU Device	Intel(R) Xeon(R) CPU E7-4820 v4	Intel Xeon Silver-4215
Number of Sockets	4	2
Processor BaseFreq	2.0 GHz	2.50 GHz
CPU Threads	80 (40 physical cores)	32 (16 physical cores)
Memory Capacity	256 GB	128 GB
Shared LLC Size	25 MB	11 MB
Operating System	Ubuntu 16.04	Ubuntu 16.04
Kubernetes Version	v1.19.2	v1.20.0

Testbed: We evaluate TETRIS using an 8-server testbed consisting of two types of machines: 2 machines are equipped with 80 cores and 256GB of memory while 6 others are equipped with 32 cores and 128GB memory. The machines are interconnected via a 100 Gbps, full-bisection bandwidth Ethernet. Table 3 lists the hardware and software details.

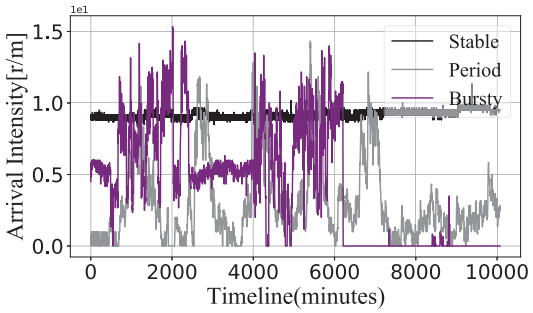
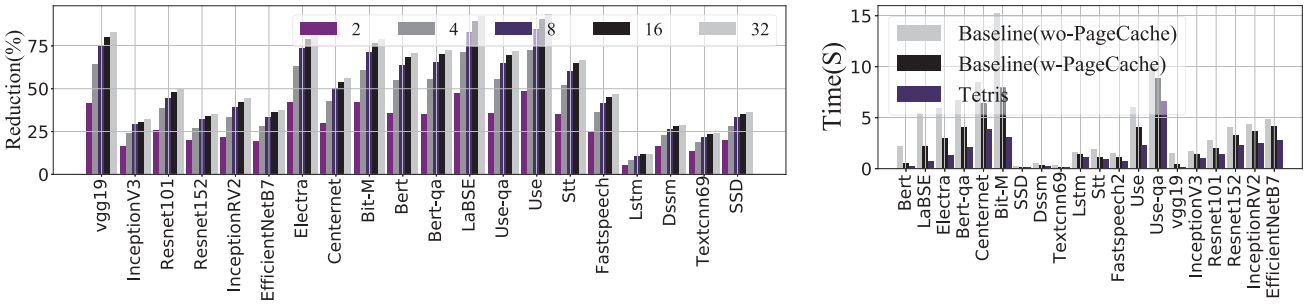


Figure 8: Production workload trace examples.

Workloads: We collect a comprehensive set of benchmark DL models with high heterogeneity from both TensorFlow Hub [22] and the real-world local life service website (Table 4). The benchmark suite is comprised of 21 inference models, ranging in model sizes from 11MB to 3.5GB, varying in download times from 310 to 1.1M, and covering application domains including text, image, audio, etc. Based on these models, we further construct four real-life applications: A *second-hand vehicle trading (SVT) application* adopts the SSD model for object detection and two Resnet152 models for classifying cars and motorcycles. An *audio question & answering (QA-Audio) application* employs the Stt, Use-qa and Fastspeech2 models for speech-to-text translation, QA



(a) Memory reduction under different # of instances

(b) Intra-model acceleration

Figure 9: (a) Memory reduction rate under tensor sharing over the number of instances from the same function. (b) Accelerating the startup using tensors from existing instance of the same function.

Table 4: Inference benchmark suite.

DL Model	Size	Description	Download times
Bit-M [37]	3.5GB	Feature vector extraction	1.4k
LaBSE [19]	1.8GB	Sentence Embedding	24.9K
Bert-qa [14]	1.3GB	Question Answering	501
Electra [10]	1.3GB	Discriminator	6.1K
Use [7]	980MB	Sentence Encoder	1.4M
Centernet [75]	731MB	Object Detection	12.8K
Use-qa [72]	568MB	Question Answering	16.3K
Use-large [7]	563MB	Sentence Encoder	1.1M
Vgg19 [63]	549MB	Image Classification	commercial
Bert [14]	392MB	Text Processing	197.5K
EfficientNetB7 [68]	255MB	Image Processing	2.2K
Resnet152 [26]	231MB	Image Processing	1.6K
InceptionResnetV2 [66]	214MB	Image Processing	6K
Stt [69]	176MB	Speech-To-Text	398
Resnet101 [26]	171MB	Image Processing	1.6K
Fastspeech2 [57]	119MB	Text-To-Speech	310
InceptionV3 [67]	92MB	Image Processing	11.6K
SSD [43]	29MB	Object Detection	commercial
Dssm [29]	25MB	Text Processing	commercial
Lstm [27]	23MB	Text Processing	commercial
Textcnn69 [9]	11MB	Text Processing	commercial

and text-to-speech translation, respectively; A *semantic similarity computation (SS) application* uses the *Use* model for semantic similarity computation; A *text question & answering (QA-Text) application* uses Textcnn69, Lstm, and Dssm for understanding user questions and finding matched answers.

These services are triggered by dynamic invocations simulated using the production trace from the Azure Function [46], where the invocations per hour illustrate diurnal and weekly patterns. Figure 8 displays all the three typical types of production traces used: stable, periodic, and bursty.

Competing approaches: We compare TETRIS with *INFless* [71], *Photons* [16], and the runtime sharing-only version of TETRIS: *Tetris-RO*. *INFless* is a state-of-the-art serverless inference system that natively supports batching and fine-grained CPU-GPU allocation for low-latency, high-throughput inference on serverless. *Photons* supports runtime sharing through concurrent execution in each instance. Since *Photons* is not serverless inference oriented and provide no SLO guarantee, we redevelop it atop OpenFaaS and extend it with function profiles while greedily selecting instance configurations with the maximum concurrency under SLO constraints. *Tetris-RO* is a variant of TETRIS that disables the tensor sharing part.

5.2 Tensor Sharing Evaluation

We first evaluate the tensor sharing effectiveness by solely activating the TETRIS's *agent*, and compare the memory footprint to that in the native OpenFaaS system. In particular, the memory footprint under tensor sharing can be derived by $M_{ts} = M_{tensor} + I \times M_{others}$, where M_{tensor} denotes the memory for parameterized tensors, M_{others} indicates the memory for runtime, libraries and others that cannot be shared across instances, and I represents the number of colocated instances in a server. Likewise, the memory footprint under the native OpenFaaS system can be derived by $M_{baseline} = I \times (M_{tensor} + M_{others})$.

Memory footprint: Sharing tensors across instances of an inference function saves memory by up to 93%. Figure 9(a) depicts the memory reduction rate by various models under various number of instances (from 2 to 32). Clearly, the memory reduction rate benefits more from increasing the number of colocated instances. The reduction rate depends on the percentage of memory that can be shared. Basically, the more parameters a model contains (i.e., M_{tensor}) and the less other memory (i.e., M_{others}) a model requires lead to a higher memory reduction rate. For example, the memory reduction rate of VGG19 (i.e., with $M_{tensor} = 549MB$ and $M_{others} = 95.4MB$) reaches to 82% when colocating 32 instances. For models with fewer parameters and relatively larger temporary memory footprints, Lstm for example, the memory reduction rate is limited to 4.4% in the 2-instance co-locating scenario. When deploying 32 function instances, the memory reduction rates of large models like LaBSE and Use even exceed 91% and 93%, respectively. For the model of Bit-M, TETRIS reduces its memory consumption by 108.5GB, because TETRIS only keeps a single copy of the model parameters in memory. Even with only two instances, the memory footprint can still be reduced by an average of 28%.

Memory footprint: Sharing tensors across functions can further reduce memory by up to 36.3%. We collect model variants which are generated by transfer learning from the InceptionV3, EfficientNetB7, Resnet101, Resnet152, Vgg19,

and InceptionResnetV2 models for applications in housing rental, recruitment, second-hand products, travel, and catering businesses. They are commonly only retrained the top dense layers. Figure 11(a) reveals that, as the number of model variants increases, the system memory can be further reduced for all models by up to 36.3%, and at least by 18.8%. Among these models, the memory reduction rate for Vgg19 is relatively limited since merely 77MB parameters are shared among its variants, although it still achieves 13.8% under the co-location of 32 variants.

Higher function density: Tensor sharing reduces the memory consumption of functions, improving the function density by up to 30×. Figure 10 presents the increasing rate of function density under various memory configurations. For the LaBSE model with 1.8GB parameters that consume 1.9GB memory, the function density is improved by 20×. In the case of the Use model, it even achieves an improvement of 30×. For large models, such as Bit-M, a server with 64GB of memory is only capable of maintaining 14 instances. After sharing tensors, the density increases to 74, i.e., more than 60 instances can be accommodated in the same machine. Tensor sharing still significantly improves function deployment density for models with small memory footprints. For instance, 1161 and 1720 additional instances can be created on a 256GB-memory server for Dssm and Textcnn69, respectively.

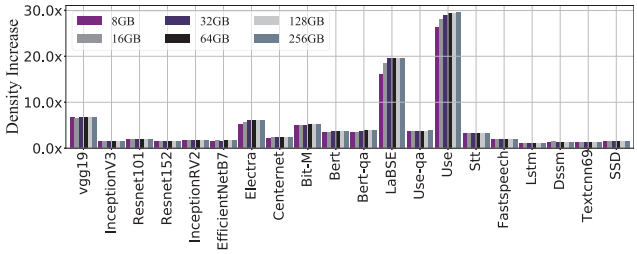
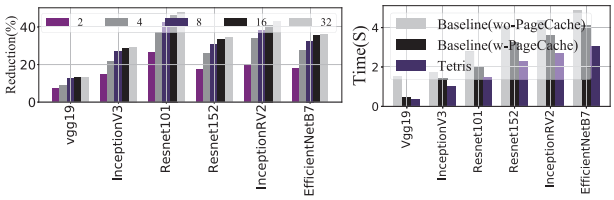


Figure 10: Function density improvement under various machine memory capacities.

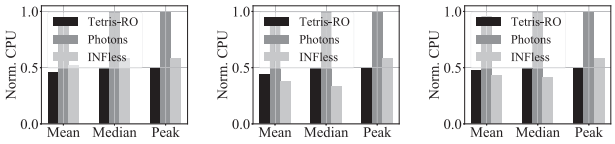
Accelerating startup: The first-time sharing of tensors reduces the startup overhead of inference functions, accelerating the startup speed by 91.56%. The loading of tensors dominates the startup process. Directly mapping an existing tensor address to a new instance is much faster than loading and decoding it from the file on disk; thus, the first-time tensor sharing feature of TETRIS can significantly accelerate the startup process. Figure 9(b) demonstrates that tensor sharing from the same function's previous instance can significantly speed up the startup process. For example, the model loading time of Bit-M exceeds 15 seconds in a native system whose page cache is disabled, which remains to be 7.9s with page cache acceleration, while tensor sharing can reduce its startup time to 2.8 seconds. Even for models with fewer tensors, such as SSD and Textcnn69, the speedup from tensor sharing still achieves 17.3% and 32.8%, respectively.

Besides the acceleration from sharing tensors of the same function, cross-function tensor sharing could accelerate startup even when the model has never been loaded. Although it may be slower than that from the same function's instance due to less redundancy across functions, the speedup still achieves an average of 46.9% (Figure 11(b)). For Vgg19, the startup process is still accelerated by 12.3% when compared with the page cache.



(a) Memory reduction across functions (b) Inter-model acceleration

Figure 11: (a) Memory reduction rate under tensor sharing over the number of model variants. (b) Accelerating the startup using tensors from the existing instance of a different function.



(a) Stable (b) Period (c) Burst

Figure 12: CPU consumption of the SS application under varying workloads.

5.3 Overall Evaluation

We further evaluate the overall efficiency of TETRIS by deploying four applications: *SVT*, *QA-Audio*, *SS* and *QA-Text*, and requests towards them are generated using three types of production traces in Figure 8.

Memory footprint: TETRIS outperforms both *INFless* and *Photons* significantly in memory consumption. Figure 13 presents the normalized mean, median and peak memory consumption by TETRIS, *INFless* and *Photons*. We find that *INFless* consumes most of the memory in all experiments for two reasons: (1) *INFless* can only reduce memory consumption when the model supports batching. However, for the applications *QA-Audio* and *SS*, there are models (i.e., *Fastspeech2* and *Use*) that do not support batching. (2) Batching introduces additional queuing time. While the inference computation on CPU is slow and the SLO is tight, we are not able to configure a much larger batch size even for batch-enabled models. (3) *INFless* prefers to use the fragmented resources spanning over multiple servers for better accommodating the residual load and improving the resource utilization, which exacerbates memory consumption. For the four applications SVT, QA-Audio, SS and QA-Text under a *stable* request load, TETRIS can reduce the mean memory footprint by more than 86%, 64%, 69%, 65%, respectively, and reduce the peak memory consumption by more than 87%, 68%, 71% and 65%, respectively. In particular, TETRIS achieves the best memory

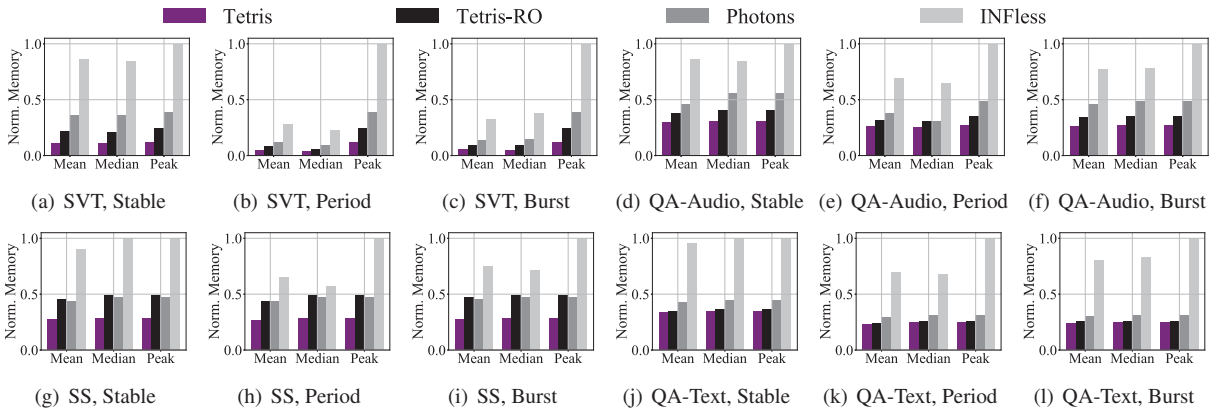


Figure 13: Normalized memory consumption by four applications under stable, period and bursty workloads.

efficiency in the SVT case because its Resnet152 models benefit more from tensor sharing as they are built from the same set of pretrained parameters. Resnet152 is also computationally intensive and limits the chance for packing requests to share runtimes. *Photons* consumes much less memory than *INFless* since it executes requests concurrently within the same instance without batch queuing time, however tensor redundancy across function instances still exists. Because of tensor sharing, TETRIS can further decrease the mean memory consumption by more than 68%, 33%, 37%, and 21%, and the peak memory consumption by more than 67%,43%,39%, and 22% for the applications, respectively.

Efficient runtime sharing: The combined optimization of batching and concurrent execution in TETRIS outperforms either *INFless*'s batching or *Photons*'s concurrent execution. As illustrated in Figure 13, *Tetris-RO* can reduce more memory footprint than *INFless* and *Photons* in cases of SVT, QA-Audio, and QA-Text: *Tetris-RO* can select configurations flexibly by exploring the various combinations of batch size, concurrency and CPUs. Taking SVT as an example, *Photons* only chooses to execute at most 2 concurrent requests within each instance due to its fixed mapping between concurrency and CPUs, while *Tetris-RO* can greedily activate 3 concurrent threads. For the QA-Audio application, although *Photons* greedily packs 4 requests into the same instance, *Tetris-RO* could further decrease runtime redundancy by concurrently executing 2 requests with a batch size of 6. For the QA-Text application consisting only of small models, the average memory consumption is still reduced by 21% from the runtime sharing.

Although *Tetris-RO* consumes more memory than *Photons* in the SS application case, *Tetris-RO* requires much fewer CPU resources. Figure 12 illustrates that *Tetris-RO* and *INFless* allocate CPU resources averagely by 42% and 39% less than *Photons*. This outcome is because each time *Photons* increases concurrency, it also requires a fixed amount of additional CPUs, resulting in excessive CPU allocations.

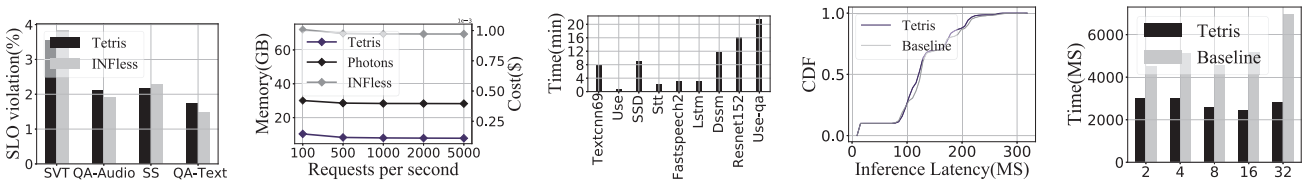
SLO guarantee: TETRIS can guarantee the latency SLO of inference workloads. Figure 14(a) demonstrates that both TETRIS and *INFless* achieve a low SLO violation rate ($< 4\%$) for all applications. For SVT and SS, TETRIS outperforms *INFless* since more CPUs are allocated for each instance in TETRIS to support concurrent processing. For QA-Audio and QA-Text, the SLO violation rate of TETRIS is slightly higher than that of *INFless* since it uses a larger batch size in runtime sharing, introducing additional batch queuing time. Overall, TETRIS achieves significant improvement in memory efficiency at the cost of a negligible increase in the SLO violation rate.

Cost savings: The reduction of memory consumption by TETRIS saves considerable monetary cost for inference service providers. To further demonstrate the benefits of TETRIS, we conduct a large-scale simulation with a combination of the mentioned applications, and gradually increase the RPS from 100 to 5000. As illustrated in Figure 14(b), TETRIS reduces memory of 61.5GB and 20.2GB per 100 requests, compared to that of *INFless* and *Photons*, respectively. If following the pricing model of $0.0504 per hour according to the r6g.medium service at AWS EC2, such memory reduction can be transformed into $0.000861 and $0.000283 cost savings per 100 requests. If considering the local life website, which serves 1.9 billion requests per day, TETRIS could reduce the monetary cost by $16,359 per day, about $5.97 million per year.

5.4 Overhead

The implementation of TETRIS does not require modifications to the ML model, the virtualization sandbox, or the underlying operating system. Developers are simply required to submit the model (instead of the function code) as the model itself contains sufficient information for deployment.

Profiling overhead: TETRIS relies on the offline profiling phase to estimate latency under various CPU, batch size, and concurrency configurations. Since inference on CPU typically incurs high latency, the exploration space of combinations of batch size and concurrency is actually limited. The automatic profiler can test each inference model under various configurations and generate profiles within several minutes. In a real system, as inference services are invoked repeatedly, the profiling only generates a one-time cost and the profiles can



(a) SLO violation rate (b) Cost savings (c) Profiling time (d) Inference time (e) Concurrent loading

Figure 14: (a) SLO violation rate. (b) Memory consumption and financial cost per 100 requests under varying workloads. (c) Average profiling time for models used in the evaluation. (d) Inference latency distribution. (e) Average loading latency over the number of concurrent loading function instances with models from Table 4.

be reused for later invocations. As depicted in Figure 14(c), profiling an inference model takes an average of 12 minutes.

Inference latency: The memory address mapping method in TETRIS does not introduce latency overhead. We measure the latency distribution of the models in Table 4. Figure 14(d) demonstrates that, compared to the latency yielded by deploying on the TensorFlow Serving framework directly, there is no performance degradation observed after using TETRIS.

Lock contention: When multiple instances are created simultaneously, the contention on the tensor lock may cause startup overhead. We measure the average model loading latency without page cache and compare it with the baseline of the TensorFlow Serving framework. Figure 14(e) indicates that TETRIS still outperforms the baseline by employing randomization in the tensor loading process, thereby reducing much of the lock contentions. Once a tensor is loaded into memory successfully, TETRIS need only to map the memory address for newly launched instances, whereas the baseline still requires expensive file reading and decoding.

6 Related Work

Memory deduplication: Prior works [3, 8, 21, 47, 70] have proposed eliminating redundant data loaded in memory by scanning, comparing, and merging duplicated pages. However, such page-level scanning methods could incur colossal overhead and lag for large, high-density serverless inference functions. As they are effective at the kernel level, TETRIS is compatible with these methods and can be used in conjunction to reduce memory consumption further.

Model compression: Large inference models can also be substantially compressed through fine-grained model design, pruning and quantization techniques (e.g., [24, 25, 31, 56]). However, unlike TETRIS, these methods involve model modifications. The compressing process may also reduce inference accuracy, resulting in side effects for businesses [53]. TETRIS and model compression can coexist.

Inference runtime optimizations: The memory footprint of inference can also be reduced by optimizing memory allocation during inference runtime [13, 35, 40, 41, 55]. However, these optimizations are either framework-specific or require model conversion, whereas TETRIS is orthogonal to these optimizations and can be employed together to further reduce memory consumption. Moreover, they may require developers to redesign or retrain the model, increasing the development burden, whereas TETRIS requires no model modifications.

Serverless inference: The existing serverless inference systems [1, 6, 71, 74] generally focus on improving inference throughput without violating the SLOs. Instead of improving memory efficiency, they primarily optimize the allocation of computational resources (e.g., CPU and GPU). TETRIS explores solving the memory efficiency problem in serverless inference and can be integrated into such systems. Trims [12] accelerates the data shipping between CPU and GPU through sharing existing models in GPU memory, whereas TETRIS explores sharing tensors among function instances.

7 Conclusion

Modern applications (such as IoT data processing, advertising recommendations, autonomous driving, and e-commerce) continuously rely on inference services. It is expected that serverless inference can reduce the maintenance cost for service providers, however, memory resources can be easily and massively harvested in existing serverless inference systems. Our proposal, TETRIS, can significantly reduce the memory footprint of inference services through runtime sharing and tensor sharing. TETRIS can be easily integrated into serverless platforms as a user-space plug-in. With TETRIS, cloud providers can deploy an order of magnitude more instances in each server without violating the SLOs, thus significantly decreasing the deployment and maintenance costs. The prototype of TETRIS is available at https://github.com/JelixLi/Tetris.

In the future, we would like to further optimize the GPU memory efficiency of serverless inference systems.

Acknowledgments

We thank our shepherd and other anonymous ATC reviewers for their extremely insightful comments and suggestions that have significantly improved the quality of this paper. We also thank Dr. Tao Li for his significant contribution to this work. This work is supported by the National Natural Science Foundation of China under grant 61872265, 62141218, U1836214; the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant 19ZXZNGX00010 and the open project of Zhejiang Lab (2021DA0AM01/003). It is also supported by Meituan.

References

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[2] Amazon. Aws lambda. https://aws.amazon.com/lambda/.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.

[4] Mahmoud Abuzaina Ashraf Bhuiyan. Improving tensorflow* inference performance on intel® xeon® processors. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/improving-tensorflow-inference-performance-on-intel-xeon-processors.html.

[5] AWS. Serverless application lens: Alexa skills. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html. Referenced 2022.

[6] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.

[7] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[8] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: Classification-based memory deduplication through page access characteristics. *ACM SIGPLAN Notices*, 49(7):65–76, 2014.

[9] Yahui Chen. Convolutional neural network for sentence classification. Master's thesis, University of Waterloo, 2015.

[10] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[11] Clive Cox, Dan Sun, Ellis Tarn, Animesh Singh, Rakesh Kelkar, and David Goodwin. Serverless inferencing on kubernetes. *arXiv preprint arXiv:2007.07366*, 2020.

[12] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-mei Hwu. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 372–382. IEEE, 2019.

[13] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proceedings of The 31st International Conference on Machine Learning*, pages 647–655, 2014.

[16] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[17] Alex Ellis. Serverless functions, made simple. https://www.openfaas.com/.

[18] Fangxiaoyu Feng, Yinfei Yang, Daniel Cer, Naveen Arivazhagan, and Wei Wang. Language-agnostic bert sentence embedding, 2020.

[19] Fangxiaoyu Feng, Yinfei Yang, Daniel Cer, Naveen Arivazhagan, and Wei Wang. Language-agnostic bert sentence embedding. *arXiv preprint arXiv:2007.01852*, 2020.

[20] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[21] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 44–59, 2017.

[22] Google. Tensorflow hub. https://tensorflow.google.cn/hub/.

[23] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4805–4814, 2019.

[24] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[25] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[29] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338, 2013.

[30] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[32] Intel. Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). https://oneapi-src.github.io/oneDNN/v0/index.html.

[33] Hugh Dickins Izik Eidus. Kernel samepage merging. https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[34] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437*, 2019.

[35] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems*, 2:1–13, 2020.

[36] Paresh Kharya and Ali Alvi. Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model. https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/. Referenced 2022.

[37] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*, pages 491–507. Springer, 2020.

[38] AWS Lambda. Netflix & aws lambda case study. https://aws.amazon.com/cn/solutions/case-studies/netflix-and-aws-lambda/.

[39] Jaejun Lee, Raphael Tang, and Jimmy Lin. What would elsa do? freezing layers during transformer fine-tuning. *arXiv preprint arXiv:1911.03090*, 2019.

[40] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.

[41] Shuangfeng Li. Tensorflow lite: On-device machine learning framework. *Journal of Computer Research and Development*, 57(9):1839, 2020.

[42] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 183–192, 2018.

[43] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[44] Pedro Marcelino. Transfer learning from pre-trained models. *Towards Data Science*, 10:23, 2018.

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

[46] Microsoft. Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[47] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. {XLH}: More effective memory deduplication scanners through cross-layer hints. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 279–290, 2013.

[48] Sangwoo Mo, Minsu Cho, and Jinwoo Shin. Freeze the discriminator: a simple baseline for fine-tuning gans. *arXiv preprint arXiv:2002.10964*, 2020.

[49] Philipp Muens. Serverless facebook messenger bot. https://github.com/pmuens/serverless-facebook-messenger-bot. Referenced 2022.

[50] Nvidia. Cuda runtime application programming interface. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html.

[51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[52] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[53] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[54] David Pisinger and Paolo Toth. Knapsack problems. In *Handbook of combinatorial optimization*, pages 299–428. Springer, 1998.

[55] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*, 2017.

[56] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[57] Yi Ren, Chenxu Hu, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. Fastspeech 2: Fast and high-quality end-to-end text to speech. *arXiv preprint arXiv:2006.04558*, 2020.

[58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[59] Paul Menage Sanjay Ghemawat. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[60] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 205–218, 2020.

[61] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[62] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. *arXiv preprint arXiv:1406.2199*, 2014.

[63] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the autumn 1990 EUUG Conference*, pages 241–248, 1990.

[65] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[66] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[67] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[68] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[69] Alexander Veysov. Towards an imagenet moment for speech-to-text. *The Gradient*, 2020.

[70] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. {UKSM}: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 325–340, 2018.

[71] Yanan Yang, Laiping Zhao, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, Feb 28- March 4, 2022*, pages 1–14. ACM, 2022.

[72] Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, et al. Multilingual universal sentence encoder for semantic retrieval. *arXiv preprint arXiv:1907.04307*, 2019.

[73] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*, 2014.

[74] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1049–1062, 2019.

[75] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.