Challenges and Opportunities of DNN Model Execution Caching

Guin R. Gilman, Samuel S. Ogden, Robert J. Walls, Tian Guo
Department of Computer Science
Worcester Polytechnic Institute
Worcester, Massachusetts
{grgilman,ssogden,rjwalls,tian}@wpi.edu

Abstract

We explore the opportunities and challenges of *model execution caching*, a nascent research area that promises to improve the performance of cloud-based deep inference serving. Broadly, model execution caching relies on servers that are geographically close to the end-device to service inference requests, resembling a traditional content delivery network (CDN). However, unlike a CDN, such schemes cache *execution* rather than static objects. We identify the key challenges inherent to this problem domain and describe the similarities and differences with existing caching techniques. We further introduce several emergent concepts unique to this domain, such as *memory-adaptive models* and *multi-model* hosting, which allow us to make dynamic adjustments to the memory requirements of model execution.

Keywords deep learning, edge server, caching algorithms

ACM Reference Format:
Guin R. Gilman, Samuel S. Ogden, Robert J. Walls, Tian Guo. 2019. Challenges and Opportunities of DNN Model Execution Caching. In *Workshop on Distributed Infrastructures for Deep Learning (DIDL '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366622.3368147

1 Introduction

Cloud-based deep neural networks (DNNs) are increasingly leveraged to provide rich features—such as real-time language translation, image recognition, and personal assistants—making it possible for resource-constrained end-user devices to benefit from complex models [2, 9, 12, 14, 21]. However, the benefits of cloud-based deep learning come at the cost of novel challenges. For example, applications must transmit inference requests and receive responses over highly-variable cellular networks; to illustrate, we measured the time to send a small image file (330KB) over a LTE network and observed times ranging from 148ms to 1405ms. Such variation is amplified when sending larger inference requests, e.g., audio or video clips, leading to unpredictable end-to-end response times and making it challenging to meet quality-of-service requirements.

In this work, we explore *model execution caching* as a novel research area for addressing network variability and, more generally, improving the performance of cloud-based deep inference serving. Intuitively, the idea is to adopt the basic structure of a content delivery network (CDN)—wherein the CDN provides server clusters geographically close to end-users—but instead of servicing requests for static objects, such new systems must service requests for model inference. Serving inference requests requires executing the appropriate model. We adopt the term caching, as solutions in this space must decide, for example, which models should be loaded into the limited GPU memory (i.e., the cache) to maximize the probability of servicing an incoming request without swapping models in and out of memory (i.e., maximizing the chance of a *model hit*).

For ease of exposition, we frame our discussion in the context of a hypothetical system for model execution caching. Like a traditional CDN, this system must host the resources (i.e., models) from multiple customers. It responds to inference requests from end-user applications by executing the requested model with the given input. However, before a model can be executed it must be loaded into memory, a process that varies from hundreds of milliseconds to a number of seconds (see Table 1). To reduce the average response time, it is prudent to keep a subset of the models pre-loaded into memory. If an incoming request is for a model loaded into memory, we refer to that as a *model hit*, otherwise the request represents a *model miss*.

We assume that the number of models hosted by this system is far larger than the size of the cache (i.e., GPU memory). The need to host a large number of models is not just due to the number of customers, but also emerges from nascent but practical real-world scenarios beyond image classification—such as per-stock prediction and language

ACM ISBN 978-1-4503-7037-0/19/12…$15.00
https://doi.org/10.1145/3366622.3368147

translation—wherein a single application/customer might use thousands of models [1].

The reader might wonder why we advocate caching deep learning models instead of caching the inference results. We focus on model execution caching as it is better suited to workloads that are *unique and specific to a particular end-device user*, e.g., images taken by user or audio clips of the user's voice. Caching the inference results for user-specific requests would not be particularly useful as there is a low likelihood of receiving an identical request that could reuse these results. An alternative and complimentary line of research might be designing schemes to perform non-trivial preprocessing [6] of incoming requests in order to map to previously cached results [19]. However, we argue that model execution caching poses fewer demands on the model developer and thus represents a lower barrier to adoption.

2 Challenges of Model Caching

Caching deep learning models, at its core, is about moving computation closer to end users. Similar to traditional content delivery networks, systems that support model execution caching will likely utilize a multi-level caching structure—e.g., the model may be placed in GPU memory, in RAM, in local disk-based storage, or at a remote location. Further, one goal will be improving the ratio of *hits* to *misses* at each cache level. However, while traditional CDNs often use servers to serve fixed-size objects to end-users to minimize response time, execution of deep learning models often requires access to servers with GPUs and focuses on an additional metric called *inference accuracy*. These and other unique characteristics of serving deep learning models pose new challenges to effectively caching deep learning models, which we describe next in detail.

There are four key differences between traditional CDN caching and model execution caching. First, DNN models have dynamic memory requirements that can depend on runtime parameters such as request batch sizes. Second, DNN models require a different cache structure than traditional CDN caches. Third, model misses need to be handled differently, e.g., loading models into memory prior to being executed. Finally, DNN serving requires efficient management of GPU memory.

Challenge 1: Runtime Memory Requirements. Traditional CDNs cache fixed-size static objects. This property simplifies resource management as CDNs know memory requirements *a priori* and can allocate the same amount of memory for an object in all cache levels. However, the runtime memory requirements of a deep learning model are often drastically different from the space requirements on disk [4]. We can attribute these differences, in part, to differences between the in-memory versus on-disk representation and memory overheads introduced by the deep learning framework. Furthermore, the memory consumption depends on runtime configuration parameters, such as the number of concurrent requests that are serviced by the model (i.e., batch size) [4]. Request batching is widely used in deep learning model serving systems, as a way to increase throughput [5, 15]. However, as determining the actual batch size depends heavily on the inference request rates as well as the performance goals, it is difficult to predict the actual memory requirement of each model.

Challenge 2: DNN Cache Levels. Another key difference between traditional content caching and model execution caching is the number of objects that need to be stored. In particular, we envision that the set of possible deep learning models will be small enough such that each server cluster will be able to store all models [1]. This avoids the need to decide which models should be stored in which cluster location and the need to transfer models across different locations when the inference request workload shifts. However, loading models from disk to GPU memory—i.e., the difference between model miss and model hit time as shown in Table 1—might be order of magnitude slower than forwarding an inference request to another cluster. This suggests the need for caching algorithms to have knowledge of other caches, an approach actively avoided by many traditional caching schemes.

Challenge 3: Miss Behavior. In the case of DNN caching systems, when a cache miss occurs, the requested model needs to be admitted to the top level cache, i.e., GPU memory, completely before inference can occur. If this top-level cache is full, then such systems need to make a model eviction decision based on the factors detailed in Section 3.2. This is in contrast to CNNs where requested objects can be read from any cache level and do not necessarily get admitted to higher-level caches completely before the request begins to be served. The requested object can be sent in pieces before it has been totally loaded into the cache. Implicitly, this means that after each *miss*, traditional cache systems have the additional flexibility to not *admit* the entirety of the missed objects to higher level caches.

Challenge 4: Managing GPU Memory. Another important class of differences arises from the GPU hardware needed for model execution caching. In particular, model caching requires the ability to efficiently manage multiple distinct models stored in GPU memory. Only recently have researchers considered using memory paging techniques to simplify management and improve utilization [22]. Currently, GPU paging is still much slower than RAM memory paging owing to the lack of pipelined page-fault handling similar to what is used by modern CPUs.

Further, while GPUs are highly parallel, data must first be loaded from RAM into GPU memory. Thus, GPU memory performance is heavily dependent on the capacity of its connection to the CPU. This connection has traditionally been the bottleneck in GPU optimization, and the typical

	accuracy(%)	inference time model hit (ms)	inference time model miss(ms)	size(MB)	#params (M)
SqueezeNet	72.9	28.6 ± 1.1	173.4 ± 25.7	4.8	1.2
MobileNetV1 0.25	74.1	25.7 ± 1.2	272.8 ± 45.0	1.9	0.5
MobileNetV1 0.5	84.9	26.3 ± 1.2	302.8 ± 45.5	5.2	1.3
DenseNet	85.6	49.6 ± 3.2	1149.0 ± 108.0	43.9	-
MobileNetV1 0.75	88.1	28.0 ± 1.1	351.9 ± 47.4	10.5	2.6
MobileNetV1 1.0	90.6	28.2 ± 1.2	421.2 ± 47.1	17.1	4.2
NasNet Mobile	91.5	55.3 ± 4.1	2817.2 ± 123.7	21.9	5.3
InceptionResNetV2	94.0	76.3 ± 5.7	2844.3 ± 106.5	121.6	55.8
InceptionV3	93.8	55.8 ± 1.2	1950.7 ± 101.2	95.7	23.8
InceptionV4	95.1	82.8 ± 0.9	3162.2 ± 134.0	171.2	42.7
NasNet Large	96.1	112.6 ± 6.1	7054.5 ± 238.4	356.6	42.3

Table 1. Statistics of popular CNN Models. We measured the average inference time using a `p2.xlarge` GPU server with 12GB GPU memory on Amazon EC2.

approach has been for programmers to explicitly control this data transfer.

3 Caching Model Design Principles

Given that model execution caching is inherently a caching problem, we first evaluate the effectiveness of existing cache replacement algorithms [3, 8, 20] in managing deep learning models at edge locations. One promising way to adapt prior cache algorithms to the domain of DNN execution caching is by defining the utility and cost of a cache miss, as well as isolating the features of DNN models which the caching algorithm should consider in making admission and eviction decisions. We next define the concept of model misses and outline such potential design factors.

3.1 Model Misses

In a DNN cache, a model miss indicates that we will not be able to perform the inference request before the model is loaded into GPU memory. Unlike in the case of typical web servers, loading the model on-demand can take seconds. Moreover, the cost of a *model miss* is dominated by the time it takes to load models from storage to memory. There is also a further cost, as once loaded, the model must be run on the requested input, which costs additional memory and time. This is a cost not present when caching static objects such as in web serving. Because of this, it will be necessary to use DNN runtime memory consumption and the time to load DNN models into memory as two key aspects of the cost of model misses.

This also means that in attempting to reduce the cost of model misses, we must answer the question of how to handle the model misses to mitigate performance impact. The essential aspect of this problem is identifying the elements of the models which increase the cost of a model miss, and what the trade-offs are for attempting to account for those qualities. For instance, one could trade accuracy for time by using a model that is already present in the GPU memory instead of the requested one. This in turn translates to the problem of which in-memory model we should use. The choice of the model needs to consider the impact on inference delays and accuracy, as well as memory usage increase. In particular, different models might have batched inference requests waiting for different amounts of time, and the GPU memory increases non-linearly with the number of inference requests.

3.2 Possible Factors Influencing Caching Policy

Model Size. In AdaptSize [3], a novel caching algorithm for web servers, the authors showed that a probabilistic admission function dependent on the size of the object being admitted drastically improved the object hit ratio simply by virtue of there being more objects in the cache when a given request is received.

One issue particular to our case is that the variable *size* on which approaches such as the AdaptSize algorithm depend is not as straightforward in the case of DNN serving as it is when considering typical CDN caching problems. Because model complexity is a fairly reliable estimate of the memory overhead required to run the models, and batch size varies with memory usage [4], a size statistic incorporating both model complexity and input batch size may be a useful metric for cache admission.

Since the model has to be run on the given input, the memory requirements for the models are dynamic. Additionally, the extent of the relationship between batch size and memory requirements is largely model-dependent. Some offline model measurements may be necessary to understand the relationship between these factors before the caching algorithm can completely account for them.

Frequency of Use. A typical factor in caching algorithms is the frequency of use of the object in question; more frequently used items are less likely to be evicted because they are more likely to be requested in the future.

AdaptSize [3] presents an eviction policy that employs a Markov chain model to track an object's position in the least recently used list, so that the overhead required scaled linearly instead of exponentially with the number of objects in the cache. This allows for efficient serving of a large number of small cache objects. The calculation in our case would once again be further complicated by the dynamic component to the size of the models.

In addition, a global least-recently-used list could allow for cluster-wide awareness of admissions/eviction decisions to improve efficiently, but may be too costly to implement.

Model Accuracy. In the case where the user does not have to specify a model to run, there are aspects of the model that influence which should be selected for the request, and one of those is the accuracy of the output. Because accuracy is not directly proportional to model complexity or the number of operations executed, model accuracy must be considered separately from size. It may be the case that more accurate models are requested more often regardless of their overhead cost, since the inference would not be taking place on the mobile device and therefore the user does not have to factor the model overhead into their choice of model. This would lead to a policy of more accurate models being admitted more frequently. However, time complexity would still play a role when considering model accuracy (see below).

Model Speed. The time complexity of the DNNs will have a direct impact on rate of servicing requests. While it seems that with high-end hardware, most models can achieve low inference times, it is also true that maintaining a certain throughput also diminishes model accuracy, meaning that there exists a tradeoff between these two features. To resolve this, it may be possible to split requests with large batch sizes into subsets of smaller batches and run them concurrently.

4 Opportunities for Model Caching

The ability to cache deep learning models brings the benefits of improved inference response time. However, as we have discussed in previous sections, the inherent differences between deep learning models and traditionally cached objects make it very challenging to design efficient model caching algorithms. The challenges can be summed up as high model miss penalty, i.e., loading the model into the GPU memory can take up to a few seconds; and the dynamic model memory requirements caused by model type and batch sizes. The three broad open questions in the area of model execution caching can be categorized as: *(i)* how to reduce the model miss rate, *(ii)* how to handle a model miss, and *(iii)* when to make model cache decisions. Bearing these questions in mind, we next describe a few emerging opportunities.

Designing memory-adaptive models. One of the key insights in caching deep learning models is to make the models we use to service requests dynamic. To achieve this, we would need a new type of model, which we refer to as *memory-adaptive* models. These models, akin to scalable video codec [16], could provide us the flexibility to dynamically adjust the total model memory requirement based on available GPU memory size and its corresponding batch size. More concretely, to use these memory-adaptive models, we could load the base model—a fully functional model that is capable of producing inference results, i.e., having the output layer, albeit at lower quality—when the batch size is large. When the batch size decreases, we could enhance the model complexity and accuracy by adding in additional layers. This removes the need to evict and load the *entire* highest-quality model, thus reducing the potential penalty of a model miss. Currently, some existing works [17, 18] have demonstrated promising results towards automatically searching for models that could balance trade-offs between accuracy and other metrics of interest, such as memory consumption.

Post-loading of deep learning models. What if we could start executing models before the model is fully loaded? Here, fully loaded means that all the model weights, layers, and neurons are initialized and ready to serve the inference request. By only partially loading the model into the GPU memory before starting execution, we can reduce the model cache miss penalty in terms of time. The challenge lies in deciding which model layers need to be preloaded into the GPU memory without introducing "jitter", i.e., unwanted execution delay after inference execution started that is caused by having to wait for loading in additional model data. One needs to at least consider and understand the difference between model execution time and loading time. In addition, if all the models residing inside the GPU memory are partially loaded, we could also fit more models given a fixed size GPU memory. To achieve post-loading, we could potentially leverage techniques such as prepaging and page faults, from post-copy live migration [11] in which the destination VM can start executing after a small amount of data, e.g., processor states, has been transferred.

Co-designing scheduling and selection algorithms. In addition to redesigning deep learning models, we could achieve similar effects by co-designing the request scheduling, model selection, and model caching algorithms. Here, a request scheduling algorithm is in charge of assigning inference requests to the model servers and a model selection algorithm decides which specific model instance to assign the requests to. Note that in the case of approximating inference results, i.e., trading accuracy for fewer model cache misses, the model selection algorithm could choose the models that are already residing in the memory instead of the ones that are requested. More concretely, for requests that belong to the same type of inferences, e.g., image classification or object detection, the model selection algorithms can have the liberty to choose from a set of functionally-equivalent models—in essence, whichever models that are currently in the GPU memory. The challenge is to define and maintain functionally-equivalent models, e.g., which two models can be considered as functionally-equivalent given the performance requirements, or make sure at least one model from each functional class is present in the GPU memory at all time.

To design a cache-aware request scheduling algorithm, the key idea is to organize queued requests into buckets ordered by models, and assign a batch of requests to models once the bucket is full. Batching requests is currently used to improve training speed [13] and inference throughput [5], and can naturally happen in inference serving systems for popular models as well as when model execution speed is slower than request rate. The natural question is then how we should configure the batch size for each model. One way is to set the batch size based on the memory requirements of each model (for a given batch size). Another way is to set the timeout based on the maximum desired response time for a request [5]. In summary, both co-designs help with reducing the cache miss rate as well as mitigating the impact of model cache miss.

Proactively loading and evicting models. Currently, model caching decisions are made reactively, i.e., when there is not enough GPU memory to load the requested DNN models, in-memory models have to be evicted to load the request model. What if we could predict the model utility, e.g., memory requirements and model accuracy, and pre-evict and pre-load models? This is akin to preloading objects to memory, a commonly-used technique to speed up application execution time. However, the effectiveness of preloading is often limited by the ability to predict the short-term model request pattern. To account for this, we might be able to leverage observed workload spikes, e.g., a burst of inference requests for a particular DNN model [7]. While that model is not currently present in the cache, we could start serving requests using the functionally-equivalent model in memory and start preloading the requested model with the anticipation of the increasing demand.

Coordinating and pooling clusters' GPU memory. Conventionally, in the web caching scenario, when a cache miss happens, the request is forwarded to the next-level authoritative source to fetch the content. In the case of model cache misses of a GPU server, this could mean forwarding the inference request to a nearby location where the requested model is present in the GPU memory. However, it is challenging to have up-to-date information regarding model cache status without incurring costly control messages—the number of messages per second can be proportional to $O(nmr)$ where n is the total number of locations, m is the maximum neighbor location set, and r is the inference request rates. For example, whenever a model loads into or unloads from GPU memory, an accompanying control message will be sent to a group of nearby cluster locations. On the other hand, it might be possible to combine server resources from different cluster locations to create a virtual pool that coordinates the model caching decisions—communicating among closer cluster locations can still be orders of magnitude faster than loading the model on-demand. This also helps to avoid *one-hit-wonder* requests as we could now wait to load a model into local GPU memory on the subsequent second inference request.

5 Summary

In this paper, we outlined a nascent research area–model execution caching. We identified unique challenges and principles associated with designing such systems. We further recognized opportunities in this domain, such as designing memory-adaptive models, that help with building coherent and complementary solutions. As part of future work, we plan to design a new caching algorithm for managing deep learning models.

Acknowledgments

This work extends a preliminary discussion [10] and is supported in part by the National Science Foundation under Grant No. 1755659 and No. 1815619.

References

[1] 2019. Lazily load models and on insufficient resources for a load, look to unload idle models · Issue #1403. https://github.com/tensorflow/serving/issues/1403.

[2] 2019. List of Hosted Models. https://www.tensorflow.org/lite/guide/hosted_models.

[3] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *NSDI 17*.

[4] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark analysis of representative deep neural network architectures. *IEEE Access* (2018).

[5] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI 17*.

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

[7] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2017. Cachier: Edge-caching for recognition applications. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 276–286.

[8] Binny S Gill and Dharmendra S Modha. 2005. SARC: Sequential Prefetching in Adaptive Replacement Cache.. In *USENIX Annual Technical Conference 15*.

[9] Tian Guo. 2018. Cloud-based or on-device: An empirical study of mobile deep inference. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*.

[10] Tian Guo, Robert J. Walls, and Samuel S. Ogden. 2019. EdgeServe: Efficient Deep Learning Model Caching at the Edge. In *The Fourth ACM/IEEE Symposium on Edge Computing*.

[11] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.* (2009).

[12] V. Ishakian, V. Muthusamy, and A. Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*.

[13] Jakub Konečnỳ, Jie Liu, Peter Richtárik, and Martin Takáč. 2015. Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing* (2015).

[14] Samuel S. Ogden and Tian Guo. 2018. MODI: Mobile Deep Inference Made Efficient by Edge Computing. In *HotEdge 18*.

[15] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139

[16] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. 2007. Overview of the scalable video coding extension of the H. 264/AVC standard. *IEEE Transactions on circuits and systems for video technology* (2007).

[17] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[18] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*.

[19] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision *(MobiCom '18)*.

[20] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. 2017. Mithril: Mining Sporadic Associations for Cache Prefetching *(SoCC '17)*.

[21] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[22] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.