Summary of **Serving DNNs like Clockwork: Performance Predictability from the Bottom Up**

Ruiyang Zhu (ryanzhu), Wenyuan Ma (wenyuanm), Shuowei Jin(jinsw)

March 2021

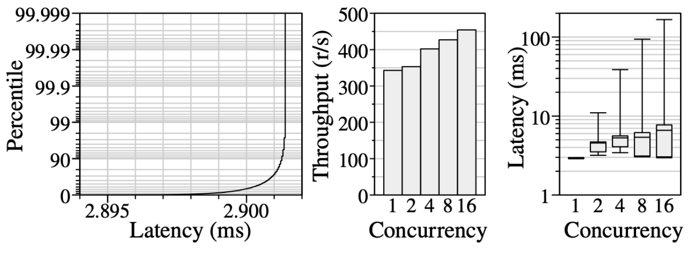
1 Problem and Motivation

The problem the paper wants to solve is that the existing DNN model cannot effectively curtail tail latency caused by unpredictable execution times. The existing systems always presume that the components have unpredictable latency performance, such variability will lead to the propagation of tail latency to the higher layers.

The motivation of the paper comes from the observation that DNN's inference time is predictable. To be more specific, the inference is a pre-defined sequence of tensor multiplications and activation and it is a deterministic execution without branches, so functions exhibit negligible latency variability. Having such predictable inference time, the paper proposes to centralize all resource consumption along with scheduling decisions, to only execute inference requests which the system is confident to meets the latency SLO.

2 Hypothesis

The first and most important hypothesis is that there is no conditional branch in DNN inference, which is a fully deterministic execution. Each DNN inference request is a pre-defined sequence of tensor multiplications and activation functions. The author also did experiments to validate such a hypothesis. They execute 11 million inference in isolation over the NVIDIA Tesla v100 GPU using random inputs and batch size 1. As Fig. 1 demonstrates, the 99.99^{th} percentile latency was within 0.03% of the median latency.



(a) CDF of 1-thread latency (b) Inference throughput and latency. Whiskers show min and max.

Figure 1: Inference results in isolation

The second hypothesis is the model serving users' requirement of a timely response. Most cloud and data center services have service-level objectives (SLOs) which codify the users' expectations of the service. The latency is one of the most important factors. If a service fails to meet the latency SLO, the service provider may risk a penalty. The biggest challenge for a service provider to meet latency SLOs lies in minimizing the tail latency. A large number of latency variabilities exist in the complex system architecture.

3 Solution Overview

The high-level idea of the solution is to restrict lower layers' choices and consolidate choices in the upper layers. Predictable DNN workers are crucial to achieving the consolidated choices. To address the predictable problem, the paper proposes three major design choices which focus on memory management, inference execution and interface with controller respectively.

3.1 Predictable DNN

3.1.1 Memory management

Memory can be unpredictable because there is performance variability between cache misses and hits. To overcome this variability, clockwork treats GPU memory as a cache letting commonly or recently used models avoid expensive loads. Workers explicitly expose LOAD and UNLOAD actions to the controller for copying models to and removing models from worker's GPU memory with deterministic latency.

3.1.2 Inference execution

The controller only sends an INFER action when a model is present in GPU memory or a LOAD action will momentarily complete. Then the action is divided into three steps. First, the input vector is transferred from host to GPU memory. Then, the EXEC performs the actual DNN GPU calculations. Finally, OUTPUT transfers the output vector from the GPU back to host memory.

3.1.3 Interface with the controller

Workers strictly follow the controller's schedule, in which the controller communicates the timestamps with every action, and designate a time interval for the worker to begin the action. Actions that can't start in the window are canceled and never executed.

3.2 Consolidating Choice

Clockwork's consolidated choices are designed from three perspectives. First, no worker operation should have implicit performance side-effects on any future operation. Second, the predictable component either delegates scheduling decisions that may influence the centralized controller or make deterministic schedules. Thirdly, when the predictable component is unable to execute a schedule as instructed, it is treated as an error to enable workers to get back on schedule.

4 Limitations and Possible Improvements

The clockwork only focuses on DNN inference. It didn't fully consider the data preprocessing and post-processing steps, which are also crucial components of a system. These steps can be the bottleneck in a real-world system scenario.

Also, as the author mentioned the network latency can also have a negligible impact on the system's performance. The network hand-offs and bandwidth variation can happen a lot in a real-world scenario. A possible solution to this is also to request some low-level network information, like RSRP, etc to access the user's request, and also do prediction on the network latency based on these information.

5 Summary of Class Discussion

Below aiare the summary of Q&A for the paper in the class/piazza:

Q:For experiments on Clockwork's performance, what are those Microsoft Azure Functions? Are they a trace of functions arriving at different rates?

A:They are inference requests that Microsoft is serving in this particular case.

Q:Is this workload open source? Is it possible to reuse it?

A:Yeah the authors have specified where they come from by giving a couple of citations.

Piazza Question: How does batching inference requests work? The paper mentions that 'each model has a request queue per batch size'. Does that mean we can change the batch size of a model? How to vary batch size?

A:The batch size is the number of inputs appended for parallel execution. Models don't explicitly have a batch size. Clockwork predicts the best batching strategy which allows for maximum execution without violating any SLO constraints.

Q:How would Clockwork work in a Clipper setting? It seems that Clockwork focuses on the performance of individual worker, would it be still the same in a very large scale?

A:CLockwork by design is contrasting to what Clipper does. The extent of distribution of Clockwork is limited. For a large scale, you probably need to have them in close-by datacenters, so the network time doesn't add too much overhead.

Q:This could be an interesting topic: how will you do a similar optimization like Clockwork in a large-scale distributed share.

Q:Is it possible to automate the configuration of hardware, e.g. I come up with a new accelerator tomorrow? should not be too difficult from upper layers

A:I am not sure but from my understanding it should not be too difficult because the choices are not left to hardware but are done in the upper layers of the system itself.

Q:My thought about merging the two types of system is that you want something like Clipper but allow the model container interface to be a bit fatter and make more fine grained scheduling decisions.

A:That would be a good start but there may be many sources of interference like over the network. Also, they are all running in separate loops and the system is decentralized, so it's a problem how they are going to interact.

A:One of the ideas behind clockwork is to avoid the container based approach where the GPUs or the hardware is dedicated to some models.

A:Even the Clipper is less efficient but it's very easy to manage. Is it possible to have this ease of manageability in a container?

Q:Will compiler optimizations help, i.e. you compile the neural nets with grammar and you don't need such large batch sizes?

A:I don't have a good answer for that but these are all interesting research questions.

Summary of **Clipper: A Low-Latency Online Prediction Serving System**

Ruiyang Zhu (ryanzhu), Wenyuan Ma (wenyuanm), Shuowei Jin(jinsw)

March 2021

6 Problem and Motivation

Machine learning application deployment often includes two stages: training and inference. While many system research works focus on problems that address model training, little attention is given to model serving and deployment. Nevertheless, given the rise of diverse machine learning frameworks, models and their hardware requirements, deploying those models is not a trivial task and often to be error prone. Furthermore, those models may need to be updated from time to time when new techniques arise and are being deployed, updating the deployment between different ML frameworks can be even more time consuming and require system expertise since the models are tightly coupled with the application. Current solutions often pre-record possible queries and store them in order to improve the latency performance for the application. This approach is inefficient because it wastes computational resources and spaces and is very hard to update. The proposed solution, Clipper, is motivated by those problems and aims to build a system that optimizes application latency and throughput as well as providing unified interfaces to different machine learning models.

7 Hypothesis

The paper brings up several main challenges and addresses them in the system Clipper.

1. How to handle the framework and model heterogeneity as they might require different ML frameworks and may need to be updated from time to time?
2. How to optimize the prediction latency and throughput for queries as inference often happens in real-time?
3. As model development is often an iterative process, how can the system help on selecting which model to use for online inferences?

The proposed system Clipper, theorizes that by decoupling those challenges into a two-layer system architecture which includes model abstraction layer and model selection layer. Model abstraction layer deals with the problem of heterogeneous models and provides abstraction through containers and the model selection layer makes model selection decisions based on selection policies to optimize the query performance.

8 Solution Overview

Figure 2 illustrates the system architecture of Clipper, which contains two layers. Inside the model abstraction layer, common techniques used in systems such as caching and batching are adopted to optimize performance.

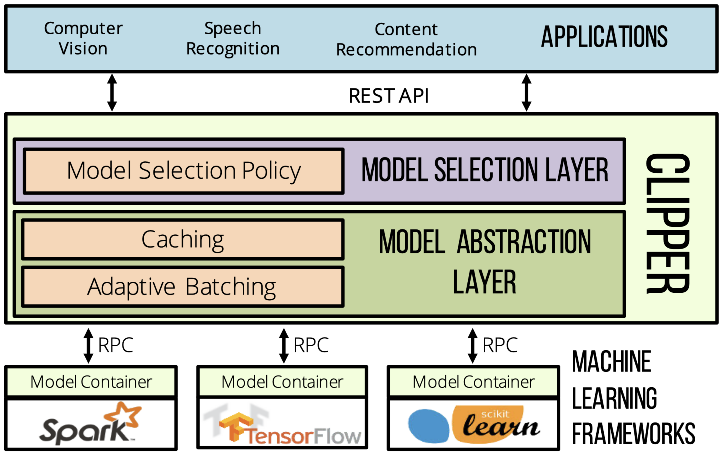


Figure 2: Clipper Architecture

8.1 Model Abstraction

To address the complexity of deploying different ML models, Clipper introduces a model abstraction layer that provides common interfaces that isolates applications from various ML frameworks. To be detailed, models deployed in Clipper are encapsulated in a docker container and communicating with Clipper through RPC. And inside the model abstraction layer, caching and batching are used to optimize the runtime performance of the ML application. Caching can be useful because most popular requests can be queried repeatedly and the result can be saved and given back quickly directly from the cache to the client. Clipper employs the traditional LRU eviction policy for cache eviction and stores the prediction results with index of model id and query x. To improve the throughput, Clipper transforms the concurrent stream of prediction queries into batches and does prediction on a batched basis. This technique better utilizes the parallelism provided by many ML models which often involve matrix multiplication and amortize the framework overheads in the system. To support the application latency deadline, Clippers tries to find the batch size that maximizes the throughput while satisfying the application latency deadline through an AIMD searching manner. For scaling the models in a cluster, it can be done easily with data parallelism so that model container is replicated in a cluster to benefit from horizontal scaling.

8.2 Model Selection

The model selection layer dynamically selects one or many of the deployed models' outputs for more accurate and robust predictions. Clipper provides a simple select, combine, and observe API abstracted from common techniques for model selection and composition. And it isolates the model selection and makes an initialization function to instantiate selection policies. Currently both single model selection policy and ensemble selection policy are supported. For single model selection, the solution is modeled as a multi-armed bandit problem (a well studied problem with many works). And for ensemble model selection policies, Clipper adopts a linear ensemble method that computes the weighted average of the based model predictions. To address stragglers in multi-model selection, Clipper adopts the theory that inaccurate but intime predictions are better than late but accurate predictions and makes decisions based on the subset to satisfy the latency constraints.

9 Limitations and Possible Improvements

At the high level, the system clipper still treats the ML models as black boxes and adds two layers on top of them. The assumption that the ML models are always below the serving system stack might not be always effective as there can be cases where incorporating the characteristics of certain models and specialized serving systems can definitely perform better. For possible improvements, it might be worthwhile to study combining the Clipper architecture with some lower level architectures that have more fine-grained control over the GPU resources to account for the limitations.

On another aspect, robustness of the model selection can be explored more thoroughly. Currently the paper focuses on assuming the queries are well formed. While the topic of adversarial machine learning (AML) is very popular, the proposed system has the potential to detect adversarial queries/examples by model assembling and potentially find vulnerabilities for different ML models at the same time.

10 Summary of Class Discussion

Below are the summary of Q&A for the paper in the class/piazza:

Q: I think they mention in the paper they actually aren't able to utilize GPUs for their models, did I misunderstand this?

A: The clipper system does support GPU for models, in a coarse-grained way. Each container might have full access to GPU resources, but fine-grained sharing of such resources is not possible for different models in Clipper.

Q: For ensemble model selection, how to figure out how many models (copies) to run since that require much more resources (number of copies deployed)?

A: The question is more like a design choice made by the developer, e.g. how much models they want to deploy to get good accuracy, and also there is single model selection policy.

Q: Can we create fine-grained sharing of GPU resources in the container?

A: As long as it conforms to RPC, it should be usable. This is more like a configuration problem. With the current github page of NVIDIA Container Toolkit, it seems it is not supported right now.

Q: Can we make Clipper more general (e.g. Any workloads that contains huge concurrent requests)?

A: Yes. The high level techniques used in Clipper are well-studied for load balancing. There're many works addressing those issues.

Q: Does anyone know why Clipper is no longer being maintained?

A: I think it was created by PhD students and they graduated, since they no longer go to school together they decided to move on.

Q: What is the value of creating a new system for just supporting machine learning workloads?

A: It's more like a prototype rather than an end product. Such ideas like model ensemble might be useful and those changes are needed to make to some existing product and name it as a new system, showing that benefit of such changes. In the end, ideas are more important than whether the implementation is starting everything from scratch.