Learning-Augmented Algorithms

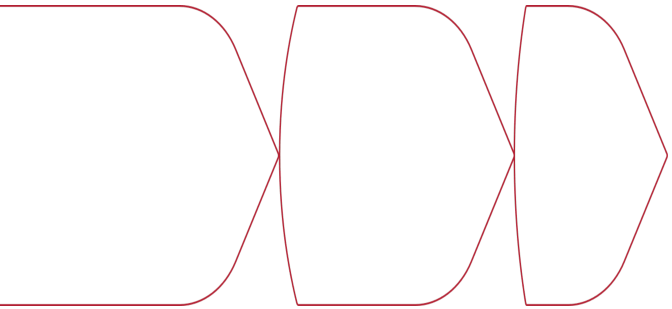
Bridging the Gap Between Theory and System

Hailiang ZHAO @ ZJU-CS

http://hliangzhao.me

November 10, 2024

2024 Future Computing Forum



Online Problems are Ubiquitous

Online decision-making problems are ubiquitous in *resource management* of different computing hierarchies.

1. **Cluster Level.**
   - job acceleration through offloading
   - load balancing across different nodes
   - QoS management of online services with co-located batch jobs
   - VM allocation (balls-into-bins)
2. **Node Level.**
   - virtual memory management (online paging)
   - threading scheduling (preemptive or non-preemptive, priority, single- or multi-queue)
   - placement with cache, memory, and storage

Online Decision-Making

Online resource management (allocation, reclamation, etc.) needs to be conducted based on

1. *online information (number and type of arriving entities) and*
2. *the current system status (resource utilization, load distribution, etc.).*

Gap between Theory and System

The algorithms for systems *with some theoretical performance guarantees* should be **simple**, **clear**, and **easy to implement**. Most importantly, their performance guarantees should not be based on unrealistic assumptions.

A Classic Example

Suppose you are about to go skiing for the first time in your life. Naturally, you ask yourself whether to rent skis or to buy them. Renting skis costs, say, \$1, whereas buying skis costs, say \$b. Your goal is minimize your total cost on all future ski trips. Unfortunately, you don't know how many such trips there will be. *You must make the decision online.*

If you know the future, the best algorithm is

$$\text{Decide to } \begin{cases} \text{rent for each trip} & x < b, \\ \text{buy} & x \geq b, \end{cases} \tag{1}$$

where x is the number of such trips. We name it OPT.

Competitive Ratio

Competitive ratio is the **worst-case** ratio across all inputs between the costs of any online algorithm ALG and an OPT:

$$\mathrm{CR}_{\mathrm{ALG}} = \max_{\forall I} \frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)}, \tag{2}$$

where I is an input instance (one among infinite possible futures).

We say ALG is $\mathrm{CR}_{\mathrm{ALG}}$-competitive if its competitive ratio is $\mathrm{CR}_{\mathrm{ALG}}$.

The Break-Even Algorithm

Rent for $b-1$ trips and buy skis at the beginning of the b-th if you are still up for skiing.

Theorem

The Break-Even algorithm is 2-competitive.

Proof.

The worst-case instance I is: One stops ski after the b-th trip. Then,

$$\frac{\mathrm{BEA}(I)}{\mathrm{OPT}(I)} = \frac{(b-1)+b}{b} \leq 2. \tag{3}$$

□

Algorithms with Predictions

The Big Question

Can we use machine-learned predictions to improve the quality of online decision-making?

Of course, you can always apply the learned models to the problems directly. But, what is the cost?

- **No interpretability** (we don't like this)
- **No robustness guarantees** (the performance can be very poor in some case)

We need to take the **imperfection** of predictions into decision making!

Theoretical Performance Analysis

Consistency & Robustness

Take the competitive ratio of ALG as a function of prediction error η: $\mathrm{CR}_{\mathrm{ALG}}(\eta)$. We say ALG is

- δ**-consistent** if $\mathrm{CR}_{\mathrm{ALG}}(0) = \delta$, and
- ϑ**-robust** if $\mathrm{CR}_{\mathrm{ALG}}(\eta) < \vartheta$ for all η.

We name algorithms with imperfect predictions as *Learning-Augmented Algorithms.*

Algorithms with Predictions for Ski-Rental

We denote by x the actual trip times.

Input: Predicted trip times y, a hpyer-parameter $\lambda \in [0, 1]$
if $y \geq b$ **then**
    /* Buy early for a better consistency */
    Buy at the beginning of the $\lceil \lambda b \rceil$-th trip
else
    /* Buy late for a better robustness */
    Buy at the beginning of the $\lceil b/\lambda \rceil$-th trip
end

λ models our confidence on the predictor.

- When set $\lambda \to 0$, the algorithm reduces to OPT (if $y \to x$)
- When set $\lambda \to 1$, the algorithm reduces to BEA (does not depend on prediction)

Algorithms with Predictions for Ski-Rental

We denote by x the actual trip times.

Input: Predicted trip times y, a hyper-parameter $\lambda \in [0, 1]$
if $y \geq b$ **then**
    /* Buy early for a better consistency */
    Buy at the beginning of the $\lceil \lambda b \rceil$-th trip
else
    /* Buy late for a better robustness */
    Buy at the beginning of the $\lceil b/\lambda \rceil$-th trip
end

Theorem

The algorithm is $(1 + \lambda)$-consistent and $(1 + \frac{1}{\lambda})$-robust.

Job Acceleration through Offloading

Consider a set of jobs $\{(t, p)\}$ arrive to some computing entity (CPU core, node, etc.) online.

- t: job arrival time
- p: job execution cost, e.g., latency

One may use accelerators (heterogenous computing devices) to speedup job execution.

Figure 1: CPUs and accelerators.

Job Acceleration through Offloading

Consider one type of accelerator that can be registered with a cost of C (e.g., offloading latency), and its usage duration is T. The accelerator can reduce the job execution cost to β times the original.

$$p \Longrightarrow \begin{cases} C + \beta p & \text{register an accelerator and use it} \\ \beta p & \text{already have a registered accelerator} \end{cases}$$

We need to decide to use accelerator or not for each newly arrived job.

The Best Online Algorithm: SUM

OPT can be obtained with dynamic programming. SUM is the best online algorithm, which is $(2-\beta)$-competitive.

The SUM Algorithm

At each arrival of (t, p), if the total cost of job executions without accelerators during $(t-T, t]$ reaches $\frac{C}{1-\beta}$, register the accelerator for the job:

$$\sum_{(t_i,p_i):t_i\in(t-T,t] \text{ and job } i \text{ is executed w.o. acc.}} p_i \geq \frac{C}{1-\beta}. \tag{4}$$

The PFSUM Algorithm

At each arrival of (t, p), if both the total job executions cost during $(t - T, t]$ and the total predicted job execution cost during $[t, t + T)$ reaches $\frac{C}{1-\beta}$, register an accelerator for the job:

$$\sum_{(t_i,p_i):t_i\in(t-T,t]} p_i(\text{or } \beta p_i) \geq \frac{C}{1-\beta} \tag{5}$$

and

$$\sum_{(t_i,p_i):t_i\in[t,t+T)} \hat{p}_i \geq \frac{C}{1-\beta}. \tag{6}$$

PFSUM

Theorem

Let $\gamma = \frac{C}{1-\beta}$. Denote by η the maximum prediction error, we have

$$\mathrm{CR}_{\mathrm{PFSUM}}(\eta) = \begin{cases} \frac{(2\gamma+(2-\beta)\eta)}{(1+\beta)\gamma+\beta\eta} & 0 \leq \eta \leq \gamma, \\ \frac{(3-\beta)\gamma+\eta}{(1+\beta)\gamma+\beta\eta} & \eta > \gamma. \end{cases} \tag{7}$$

PFSUM is $\frac{2}{1+\beta}$-consistent and $\frac{1}{\beta}$-robust.

Conclusion

PFSUM always guarantees a competitive ratio of $\frac{1}{\beta}$, no matter which prediction model is adopted and how poor its performance is, for any possible job arrival sequence.

Model Serving Systems

Model Serving Systems are designed to serve inference requests from DNN models, often using accelerators like GPUs to meet tight per-request latency SLOs, e.g., 10-500ms.

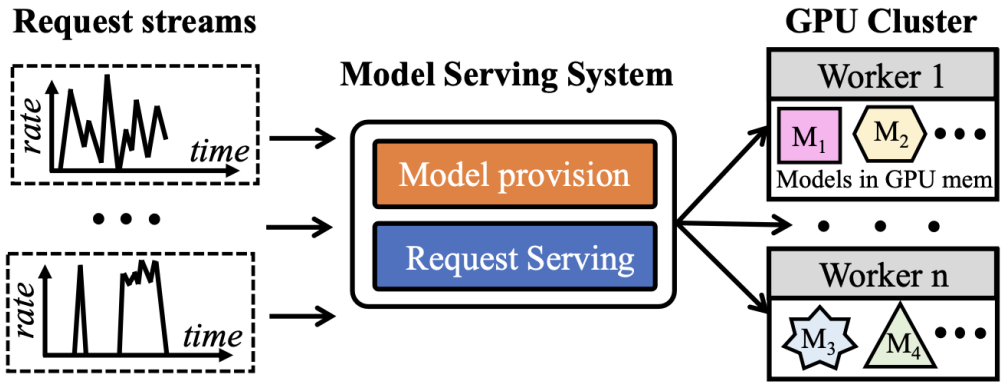


Figure 2: Model serving systems.

Requests with the same SLO and target model are typically grouped into *request streams*.

Model Serving Systems

We must make two types of scheduling decisions to meet system goals:

- Make model provisioning decisions to **scale** to a massive number of request streams using large pools of GPUs, while **ensuring high utilization** for the GPU pool for cost-efficiency. ▷ determine which model should be loaded on which and how many GPUs (replicated on multiple GPUs)
- Make request serving decisions to **maximize system goodput**, i.e., the number of requests that meet their SLO deadlines per unit time. ▷ determine batch (of requests) size, batch priority, and target GPU

Our Solution

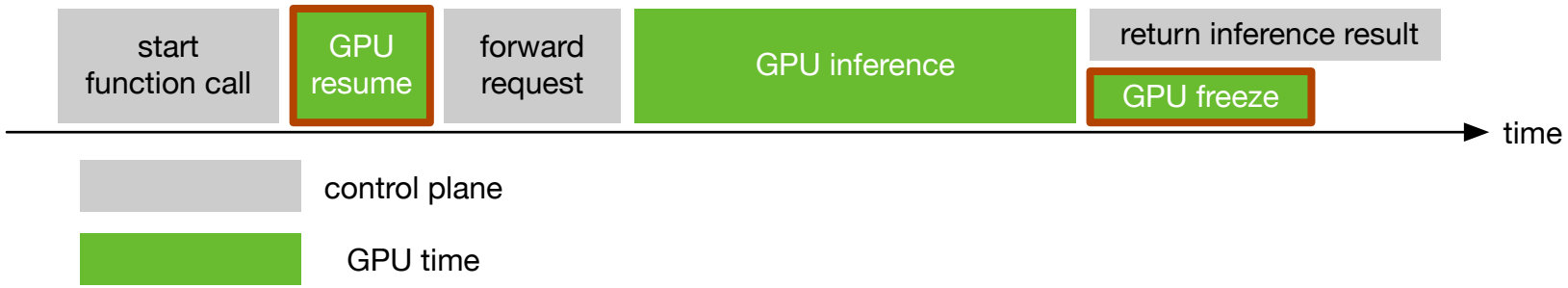
We decouple model serving into a *periodic planning* phase and an *online serving* phase.

1. **Periodic Planning** Periodically classifies inference request streams, DNN models, and GPUs into several serving groups (**Mapping**)
2. **Online Serving** Employ an online algorithm to serve requests across streams within each serving group independently (**Model Swapping is inevitable!**)

Online Model Swapping

There is a GPU and several trained models (including both traditional small models and LLMs).

Inference requests arrive online. Each request batch invokes a specific model and needs to be processed on the GPU.



Autoscaling in Online Serving

Autoscaling dynamically adjusts model provisioning based on workload. ▷ Especially in the serverless model serving, autoscaling is essential.

- Insufficient model provisioning leads to cold starts.
- Excessive model provisioning leads to wasted resources.

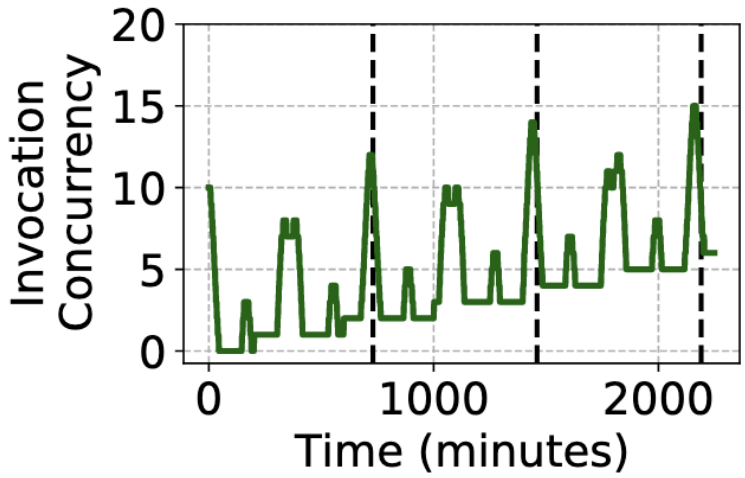


Figure 3: Model request workload

Autoscaling in Online Serving

Problem definition. Consider a time series $1, 2, \ldots, T$, suppose the required resource in the t-th time slot is w_t, and the resource we prepared just before the t-th time slot is x_t.
β for *cold start*. α for *keeping alive*.

Cost Function.

$$\text{Cost}_t(x_t) = \begin{cases} \beta(w_t - x_t) + \alpha w_t & , x_t < w_t \\ \alpha x_t & , x_t \geq w_t. \end{cases}$$

1. *The offline optimal cost:* $\sum_{t=1}^{t=T} \text{CostOPT}_t(w_t)$
2. *(Reactive) Follow the past's cost (x_t is set to w_{t-1}):*
   $\sum_{t=1}^{t=T} \text{CostREA}_t(w_{t-1})$, the competitive ratio is $2 + \frac{\beta}{\alpha}$.

Autoscaling in Online Serving

3. (FTP) Follow the prediction's cost: $\sum_{t=1}^{t=T} \text{CostFTP}_t(\hat{w}_t)$, where $\hat{w}_t$ is the predicted value, $\eta_t = |w_t - \hat{w}_t|$ is prediction error. The competitive ratio is $1 + \max(1, \frac{\beta}{\alpha}) \cdot \max(\frac{\eta_t}{w_t})$.

4. Dynamically follow the prediction (more robust):

for $t = 1, 2, \ldots, T$ **do**
  if $\sum_{i=1}^{i=t-1} \mathit{CostREA}_i \geq \epsilon \cdot \sum_{i=1}^{i=t-1} \mathit{CostFTP}_i$ **then**
    $\lambda_t \leftarrow 1$ /* follow the prediction */
  else
    /* partially follow the prediction */
    $\lambda_t \leftarrow f(\lambda_{t-1}, \epsilon, \text{CostREA}_{1:t-1}, \text{CostFTP}_{1:t-1})$
  end
  $x_t \leftarrow \hat{w}_t(\Pr = \lambda_t), x_t \leftarrow w_{t-1}(\Pr = 1 - \lambda_t)$
end

Conclusion

First, establish a model with an appropriate abstraction.

Do not use predictions directly if they are imperfect. Design an algorithm framework that enables consistency and robustness.

- **Consistency**: expected to be 1 when prediction error is small
- **Robustness**: expected to be *the competitive ratio of online optimal algorithm* when the prediction error is arbitrary large

Thanks for listening!

Hailiang ZHAO @ ZJU-CS
http://hliangzhao.me