Swayam
Distributed Autoscaling for Machine Learning as a Service

Arpan Gujarati, Björn B. Brandenburg
Sameh Elnikety, Yuxiong He
Kathryn S. McKinley
Machine Learning as a Service (MLaaS)

Microsoft Azure Machine Learning

Amazon Machine Learning

IBM Data Science & Machine Learning

Google Cloud AI
Machine Learning as a Service (MLaaS)

1. Training
   
   Dataset + Untrained model = Trained Model

2. Prediction
   
   Query + Trained Model = Answer

- Microsoft Azure Machine Learning
- Amazon Machine Learning
- IBM Data Science & Machine Learning
- Google Cloud AI
This work

2. Prediction

Models are already trained and available for prediction
Swayam

Distributed autoscaling

of the compute resources needed for prediction serving

inside the MLaaS infrastructure
Prediction serving (application perspective)

MLaaS Provider

Image classifier

image

"cat"

Application / End User
Lots of trained models!

MLaaS Provider

Finite compute resources
"Backends" for prediction
Prediction serving (provider perspective)

MLaaS Provider

- Lots of trained models!
- Finite compute resources "Backends" for prediction

Application / End User

(1) New prediction request for the pink model

(2) A frontend receives the request

Multiple request dispatchers "Frontends"
Lots of trained models!

MLaaS Provider

Finite compute resources "Backends" for prediction

Application / End User

(1) New prediction request for the pink model

(2) A frontend receives the request

(3) The request is dispatched to an idle backend

(4) The backend fetches the pink model

Multiple request dispatchers "Frontends"
Lots of trained models!

MLaaS Provider

Finite compute resources
"Backends" for prediction

Application / End User

(1) New prediction request for the pink model

(2) A frontend receives the request

(3) The request is dispatched to an idle backend

(4) The backend fetches the pink model

(5) The request outcome is predicted

(6) The response is sent back through the frontend

Multiple request dispatchers "Frontends"
Prediction serving (objectives)

MLaaS Provider

Lots of trained models!

Multiple request dispatchers "Frontends"

Finite compute resources "Backends" for prediction

Application / End User
Prediction serving (objectives)

MLaaS Provider

Resource efficiency

Lots of trained models!

Finite compute resources "Backends" for prediction

Multiple request dispatchers "Frontends"

Application / End User

Low latency, SLAs
Static partitioning of trained models
Static partitioning of trained models

MLaaS Provider

The trained models partitioned among the finite backends
Static partitioning of trained models

MLaaS Provider

The trained models partitioned among the finite backends

Application / End User

Multiple request dispatchers "Frontends"

No need to fetch and install the pink model
Static partitioning of trained models

MLaaS Provider

The trained models partitioned among the finite backends

Application / End User

Problem: Not all models are used at all times

No need to fetch and install the pink model

Multiple request dispatchers "Frontends"

Static partitioning of trained models
Static partitioning of trained models

MLaaS Provider

The trained models partitioned among the finite backends

Application / End User

Problem: Not all models are used at all times

Problem: Many more models than backends, high memory footprint per model

No need to fetch and install the pink model

Multiple request dispatchers "Frontends"

MLaaS Provider

Application / End User
MLaaS Provider

Resource efficiency

Multiple request dispatchers "Frontends"

The trained models partitioned among the finite backends

Problem: Not all models are used at all times

Problem: Many more models than backends, high memory footprint per model

No need to fetch and install the pink model

Application / End User

Low latency, SLAs

Static partitioning is infeasible
Classical approach: **autoscaling**

The number of active backends are automatically scaled up or down based on load.

Time

# Active backends for the pink model

Request load for the *pink model*

The number of active backends are automatically scaled up or down based on load.
Classical approach: **autoscaling**

The number of active backends are automatically scaled up or down based on load.

With **ideal autoscaling** ...

- Enough backends to guarantee **low latency**
- # Active backends over time is **minimized for resource efficiency**
Autoscaling for MLaaS is challenging [1/3]
Autoscaling for MLaaS is challenging [1/3]

Lots of trained models!

MLaaS Provider

Finite compute resources "Backends" for prediction

(4) The backend fetches the pink model

Multiple request dispatchers "Frontends"

(5) The request outcome is predicted
Autoscaling for MLaaS is challenging [1/3]

MLaaS Provider

Lots of trained models!

Finite compute resources
"Backends" for prediction

Multiple request dispatchers "Frontends"

(4) The backend fetches the pink model

(5) The request outcome is predicted

Challenge

Provisioning Time (4) >> Execution Time (5)
(~ a few seconds) (~ 10ms to 500ms)

Requirement

Predictive autoscaling to hide the provisioning latency
Autoscaling for MLaaS is challenging [2/3]

MLaaS architecture is large-scale, multi-tiered

- Hardware broker
- Frontends
- Backends [VMs, containers]
Autoscaling for MLaaS is challenging [2/3]

MLaaS architecture is large-scale, multi-tiered

Challenge
Multiple frontends with partial information about the workload

Requirement
Fast, coordination-free, globally-consistent autoscaling decisions on the frontends
Autoscaling for MLaaS is challenging [3/3]

Strict, model-specific SLAs on response times

"99% of requests must complete under 500ms"

"99.9% of requests must complete under 1s"

"[A] 95% of requests must complete under 850ms"

"[B] Tolerate up to 25% increase in request rates without violating [A]"
Autoscaling for MLaaS is challenging [3/3]

Strict, model-specific SLAs on response times

"99% of requests must complete under 500ms"

"99.9% of requests must complete under 1s"

"[A] 95% of requests must complete under 850ms"

"[B] Tolerate up to 25% increase in request rates without violating [A]"

Challenge
No closed-form solutions to get response-time distributions for SLA-aware autoscaling

Requirement
Accurate waiting-time and execution-time distributions
Swayam: model-driven distributed autoscaling

Challenges

Provisioning Time (4) >> Execution Time (5)
(~ a few seconds) (~ 10ms to 500ms)

Multiple frontends with partial information about the workload

No closed-form solutions to get response-time distributions for SLA-aware autoscaling

We address these challenges by leveraging specific ML workload characteristics and design an analytical model for resource estimation that allows distributed and predictive autoscaling
Outline

1. System architecture, key ideas
2. Analytical model for resource estimation
3. Evaluation results
System architecture
Objective: dedicated set of backends should dynamically scale

1. If load decreases, extra backends go back to the global pool (for resource efficiency)
2. If load increases, new backends are set up in advance (for SLA compliance)
Let's focus on the **pink model**

**Objective:** dedicated set of backends should dynamically scale

1. If load decreases, extra backends go back to the global pool *(for resource efficiency)*
2. If load increases, new backends are set up in advance *(for SLA compliance)*
Key idea 1: Assign states to each backend
Key idea 1: Assign states to each backend

- In the global pool: cold
- Dedicated to a trained model: warm
Key idea 1: Assign states to each backend

- **cold**: In the global pool
- **warm**: Dedicated to a trained model
- **not-in-use**: Haven't executed a request for a while
- **in-use**: Maybe executing a request
Key idea 1: Assign states to each backend

- **In the global pool**
  - **cold**: Haven't executed a request for a while
  - **not-in-use**: Executing a request

- **Dedicated to a trained model**
  - **warm**: Maybe executing a request
  - **in-use**: Waiting for a request

- **busy**
  - **idle**
Key idea 1: Assign states to each backend

- **In the global pool**: cold
- **Dedicated to a trained model**: warm
- **not-in-use**: Haven't executed a request for a while
- **in-use**: Executing a request
- **busy**: Can be safely garbage collected (scale-in)
- **idle**: Dedicated, but not used due to reduced load

- Maybe executing a request
- Waiting for a request

... or easily transitioned to an in-use state (scale-out)
Key idea 1: Assign states to each backend

How do frontends know which dedicated backends to use, and which to not use?
Key idea 2: Order the dedicated set of backends

Backends dedicated for the pink model

1 2 3 4 5 6
7 8 9 10 11 12
Key idea 2: Order the dedicated set of backends

If 9 backends are sufficient for SLA compliance ...
Key idea 2: Order the dedicated set of backends

If 9 backends are sufficient for SLA compliance ...

Frontends use backends 1-9

Backends 10-12 transition to not-in-use state

- = warm in-use busy/idle
- = warm not-in-use
Key idea 2: Order the dedicated set of backends

Backends dedicated for the pink model

If 9 backends are sufficient for SLA compliance ...

frontends use backends 1-9
backends 10-12 transition to not-in-use state

How do frontend know how many backends are sufficient?

Backends dedicated for the pink model

= warm in-use busy/idle
= warm not-in-use
Key idea 3: Swayam instance on every frontend

Swayam instance

Incoming requests

Frontends

Backends dedicated for the pink model

1 2 3 4 5 6

7 8 9 10 11 12

= warm in-use busy/idle

= warm not-in-use

computes globally consistent minimum # backends necessary for SLA compliance
Outline

1. System architecture, key ideas
2. Analytical model for resource estimation
3. Evaluation results
Making globally-consistent decisions at each frontend (Swayam instance)

What is the minimum # backends required for SLA compliance?
Making globally-consistent decisions
at each frontend (Swayam instance)

What is the minimum # backends required for SLA compliance?

1. Expected request execution time
2. Expected request waiting time
3. Total request load
Making globally-consistent decisions at each frontend (Swayam instance)

What is the minimum # backends required for SLA compliance?

1. Expected request execution time
2. Expected request waiting time
3. Total request load

leverage ML workload characteristics
Determining expected request execution times

Studied execution traces of 15 popular services hosted on Microsoft Azure's MLaaS platform
Determining expected request execution times

Studied execution traces of 15 popular services hosted on Microsoft Azure's MLaaS platform

Variation is low
- Fixed-sized feature vectors
- Input-independent control flow
- Non-deterministic machine & OS events main sources of variability
Determining expected request execution times

Studied execution traces of 15 popular services hosted on Microsoft Azure's MLaaS platform

- Fixed-sized feature vectors
- Input-independent control flow
- Non-deterministic machine & OS events main sources of variability

Variation is low

Modeled using log-normal distributions
Determining expected request waiting times

load balancing (LB)
Determining expected request waiting times

- Threshold (350ms)
- Global Scheduling
- Partitioned Scheduling
- Join-Idle-Queue
- Random Dispatch

load balancing (LB)
Determining expected request waiting times

load balancing (LB)

Global and partitioned perform well, but there are implementation tradeoffs.
Determining expected request waiting times

load balancing (LB)

Global Scheduling
Partitioned Scheduling
Join-Idle-Queue
Random Dispatch

Threshold (350ms)

JIQ doesn't result in good tail waiting times

Global and partitioned perform well, but there are implementation tradeoffs
Determining expected request waiting times

Global and partitioned perform well, but there are implementation tradeoffs

JIQ doesn't result in good tail waiting times

Random dispatch gives much better tail waiting times

load balancing (LB)
Determining expected request waiting times

load balancing (LB)

We use a LB policy based on random dispatch!

\[
\omega_p = d_1 + \left( \frac{\ln (1 - p/100)}{\ln (\lambda/n\mu)} - 1 \right) \cdot (d_1 + d_2 + \Delta),
\]

Global and partitioned perform well, but there are implementation tradeoffs.

JIQ doesn't result in good tail waiting.
Determining the total request load in the near future, to account for high provisioning times
Determining the total request load

in the near future, to account for high provisioning times

Since the broker spreads requests uniformly among each frontend

$L' = \frac{L}{F}$

$L'$ is the total request rate.

$L$ is the total # frontends.

Hardware broker

Frontends
Determining the total request load

in the near future, to account for high provisioning times

Each Swayam instance
- Predicts L' for near future

Depends on the time to setup a new backend

Hardware broker

L = L/F

Frontends

L' = L/F

F
Determining the total request load in the near future, to account for high provisioning times.

- Predicts $L'$ for near future
- Given $F$, computes $L = F \times L'$

$L' = \frac{L}{F}$

Each Swayam instance determined from broker / through a gossip protocol.
Making globally-consistent decisions
at each frontend (Swayam instance)

What is the minimum # backends required for SLA compliance?

1. Expected request execution time
2. Expected request waiting time
3. Total request load
SLA-aware resource estimation

For each trained model

- Response-Time Threshold: $RT_{\text{max}}$
- Service Level: $SL_{\text{min}}$
- Burst Threshold: $U$

$n = \min \# \text{backends}$
SLA-aware resource estimation

For each trained model

Response-Time Threshold
\( RT_{\text{max}} \)

Service Level
\( SL_{\text{min}} \)

Burst Threshold
\( U \)

Waiting Time Distribution

Execution Time Distribution

Response Time Modeling

Load \( x U \)

\( n = \min \# \text{backends} \)

\( SL_{\text{min}} \) percentile response time

\( \leq RT_{\text{max}} \)?

No

Yes

\( n = 1 \)

\( n++ \)
SLA-aware resource estimation

For each trained model

Response-Time Threshold
$RT_{\text{max}}$

Service Level
$SL_{\text{min}}$

Burst Threshold
$U$

$n = \text{min \# backends}$

Closed-form expression for percentile response time (see the appendix)

Load $x U$

Convolution

Waiting Time Distribution

Execution Time Distribution

Response Time Modeling

$n = 1$

$n++$

$< RT_{\text{max}}$?

Yes

No
SLA-aware resource estimation

For each trained model:

**Response-Time Threshold** \( RT_{\text{max}} \)

**Service Level** \( SL_{\text{min}} \)

**Burst Threshold** \( U \)

Response Time Modeling:

- Load \( x U \)
- Amplified based on the burst threshold
- Waiting Time Distribution
- Execution Time Distribution

\[ n = \min \# \text{ backends} \]

\[ RT_{\text{max}} \]

\[ SL_{\text{min}} \]

\[ \text{percentile response time} \]

Yes

\[ < RT_{\text{max}}? \]

No

\[ n = 1 \]

\[ n++ \]
SLA-aware resource estimation

For each trained model

Response-Time Threshold

$RT_{\text{max}}$

Service Level

$SL_{\text{min}}$

Burst Threshold

$U$

Waiting Time Distribution

Execution Time Distribution

Response Time Modeling

Load $x U$

Initialization

$n = 1$

$n = \min \# \text{ backends}$

Yes

No

Retry, as long as not SLA compliant

Compute percentile response time for $n$

$SL_{\text{min}}$ percentile response time

$< RT_{\text{max}}$?
Swayam Framework

- Swayam instance
- Incoming requests
- Frontends
- Backends dedicated for the pink model
  - Gear 1 = warm in-use busy/idle
  - Gear 2 = warm not-in-use

computes globally consistent minimum # backends necessary for SLA compliance
Outline

1. System architecture, key ideas
2. Analytical model for resource estimation
3. Evaluation results
Evaluation setup

- Prototype in C++ on top of Apache Thrift
  - 100 backends per service
  - 8 frontends
  - 1 broker
  - 1 server (for simulating the clients)
Evaluation setup

• Prototype in C++ on top of Apache Thrift
  ➪ 100 backends per service
  ➪ 8 frontends
  ➪ 1 broker
  ➪ 1 server (for simulating the clients)

• Workload
  ➪ 15 production service traces (Microsoft Azure MLaaS)
  ➪ Three-hour traces (request arrival times and computation times)
  ➪ Query computation & model setup times emulated by spinning
SLA configuration for each model

• Response-time threshold $RT_{max} = 5C$
  ➞ $C$ denotes the mean computation time for the model

• Desired service level $SL_{min} = 99$
  ➞ 99% of the requests must have response times under $RT_{max}$

• Burst threshold $U = 2x$
  ➞ Tolerate increase in request rate by up to 100%

• Initially, 5 pre-provisioned backends
Baseline: Clairvoyant Autoscaler (ClairA)

- It knows the processing time of each request beforehand
- It can travel back in time to provision a backend
- "Deadline-driven" approach to minimize resource waste
Baseline: **Clairvoyant Autoscaler (ClairA)**

- It knows the processing time of each request beforehand
- It can travel back in time to provision a backend
- "Deadline-driven" approach to minimize resource waste

- ClairA1 assumes zero setup times, immediate scale-ins
  - Reflects the size of the workload
Baseline: Clairvoyant Autoscaler (ClairA)

- It knows the processing time of each request beforehand
- It can travel back in time to provision a backend
- "Deadline-driven" approach to minimize resource waste

- ClairA1 assumes zero setup times, immediate scale-ins
  - Reflects the size of the workload

- ClairA2 assumes non-zero setup times, lazy scale-ins
  - Swayam-like
Baseline: **Clairvoyant Autoscaler (ClairA)**

- It knows the processing time of each request beforehand
  - It can travel back in time to provision a backend
  - "Deadline-driven" approach to minimize resource waste

- ClairA1 assumes zero setup times, immediate scale-ins
  - Reflects the size of the workload

- ClairA2 assumes non-zero setup times, lazy scale-ins
  - Swayam-like

- Both ClairA1 and ClairA2 depend on $RT_{max}$, but not on $SL_{min}$ and $U$
Resource usage vs. SLA compliance
Resource usage vs. SLA compliance

- Resource usage (normalized)
- Trace IDs
- Swayam (frequency of SLA compliance)

ClairA1
ClairA2

CTAC (frequency of SLA compliance)
Resource usage vs. SLA compliance

![Graph showing resource usage vs. SLA compliance for Trace IDs 1 to 15. The y-axis represents resource usage (normalized), and the x-axis represents Trace IDs. Three different traces are shown: ClairA1, ClairA2, and Swayam (frequency of SLA compliance).]
Resource usage vs. SLA compliance
Resource usage vs. SLA compliance

Frequency of SLA Compliance

ClairA1: 97%, 98%, 64%, 95%
ClairA2: 97%, 91%, 89%, 97%
Swayam: 97%, 98%, 64%, 100%

Resource usage (normalized)

Trace IDs

Swayam (frequency of SLA compliance)
Swayam performs much better than ClairA2 in terms of resource efficiency.
Resource usage vs. SLA compliance

Swayam is resource efficient but at the cost of SLA compliance.
Resource usage vs. SLA compliance

Swayam is resource efficient but at the cost of SLA compliance.
Resource usage vs. SLA compliance

Swayam seems to perform poorly because of a very bursty trace.
Summary

• Perfect SLA, irrespective of the input workload, is too expensive
  ➡ in terms of resource usage (as modeled by ClairA)
Summary

- Perfect SLA, irrespective of the input workload, is too expensive
  ➡ in terms of resource usage (as modeled by ClairA)

- To ensure resource efficiency, practical systems
  ➡ need to trade off some SLA compliance
  ➡ while managing client expectations
Summary

• Perfect SLA, irrespective of the input workload, is too expensive
  ➡ in terms of resource usage (as modeled by ClairA)

• To ensure resource efficiency, practical systems
  ➡ need to trade off some SLA compliance
  ➡ while managing client expectations

• Swayam strikes a good balance, for MLaaS prediction serving
  ➡ by realizing significant resource savings
  ➡ at the cost of occasional SLA violations
Summary

• Perfect SLA, irrespective of the input workload, is too expensive
  ➡ in terms of resource usage (as modeled by ClairA)

• To ensure resource efficiency, practical systems
  ➡ need to trade off some SLA compliance
  ➡ while managing client expectations

• Swayam strikes a good balance, for MLaaS prediction serving
  ➡ by realizing significant resource savings
  ➡ at the cost of occasional SLA violations

• Easy integration into any existing request-response architecture
Thank you. Questions?