Inspecting Batch Adaptation Policy in Machine Learning Inference Systems

Axel Agelii, Jiahui (Jack) Zhang, Tzu-Tao Chang

1 Abstract

We introduce the spread-drop policy, a batch adaptation policy for time-critical applications. The spread-drop policy aims to satisfy constraints that capture the frequency and consecutiveness of request deadline misses. We believe those constraints are crucial to real-time applications and have not been adequately addressed by the current state-of-the-art [5]. In this paper, we introduce the spread-drop policy and the gaurantees it provides with respect to the Maximum Consecutive Misses (MCD) and Weakly-Hard Constraints, which are defined in 3.3 and Section 3.4, respectively [1]. This paper focuses on the theoretical gaurantees provided by the spread-drop policy and simulates the policy under different queue distributions. We propose a machine learning inference system with a runtime scheduler using the spread-drop policy. Addtionally, we derive the maximum request rate sustainable under a specified SLO and drop policy. To evaluate our policies, we perform simulations that model a machine learning inference system processing incoming queries. Aside from testing whether our policies satisfy the SLOs. Through simulations we assess the trade-offs among batch size, latency, and throughput under different batch adaption policies.

2 Introduction

Machine learning has been leveraged in a growing range of applications such as video analytics [7] and recommendation systems [8]. As a general pipeline, models are designed and trained on a large amount of data and then deployed as a service to make predictions (inferences) based on data and queries from clients. The training phase typically takes hours to days, while the inference phase happens at the time scale of milliseconds to seconds. As a result, unlike model training, where throughput is the main concern, model serving (or inference) service is latency-sensitive. This is especially crucial in time-sensitive and safety-critical real-time applications such as autonomous vehicles [3] and spacecraft control [6]. For example, given a frame of road image, an object detection model on an autonomous vehicle is expected to be able to realize the presence of obstacles in time.

On top of the requirement for low latency in real-time applications, these systems are often subject to weakly-hard constraints [1]. It is important to note that weakly-hard constraints are defined as requiring a defined bound on the number of misses, not necessarily consecutive misses (as defined in Definition 1 of [1]). These constraints are tolerant to a single or a few deadline misses, but could fail if there are frequent or consecutive deadline misses. Using the example of an autonomous vehicle, it may not be harmful for the object detection model to miss detecting an obstacle in a single frame. However, if the model fails to make inferences on consecutive frames or too many frames within a given period of time, the control system may not be able to avoid obstacles in a timely manner, leading to severe losses and injuries.

Deadline misses in model inference systems are often a result of fluctuation in input rate; when there is a surge in the number of queries, the model may not be able to service all queries in time. It is thus important for an inference system to adopt query-selection policies and adaptive batching in order to meet Service Level Objectives (SLOs) defined by the application or the user, where the SLOs contain not only the latency objective but also the weakly-hard constraint. In addition, given the batch profile (latency for processing a batch) and SLOs of queries, a safety-critical system should be able to determine the maximum input rate the system could be sustaining (maximum sustainable rate) in order to meet the weakly-hard constraint SLO so that some rate control mechanism could be deployed beforehand. When the input rate exceeds the maximum sustainable rate during runtime, the system would be able to fall back to a safe mode. In the autonomous vehicle example, this could be handing control back to the driver herself.

Existing systems have proposed several inference policies. Clipper [2] adopts a lazy-drop policy, where the system drops any query that has already missed its deadline and dynamically changes the batch size to meet the deadline of the earliest query with a deadline that can be met that is still in the queue. Nexus [5], on the other hand, adopts an early-drop policy, where the system maintains a fixed batch size, and scans through the

queue of queries, in a sliding window manner, from the earliest query until a batch of consecutive queries is all able to be scheduled and inferred in time, with older queries being dropped. This prevents the batch size from shrinking, which could lead to further deadline misses in future queries, and thus results in higher throughput and larger average batch sizes than the lazy-drop policy [5]. However, both policies are designed only to maximize throughput and satisfy latency SLOs, and are therefore subject to consecutive or frequent deadline misses. This is insufficient for weakly-hard real-time inference systems in which consecutive drops are not desired.

In this project, we aim to design policies that are aware of the latency SLO as well as the weakly-hard constraint SLO. We conjecture that a mix between the lazy-drop policy and the early-drop policy might be beneficial: by using the dynamic batch size and dropping some earlier queries, the system should end up with fewer consecutive and less frequent deadline misses. This would allow the system to satisfy the weakly-hard constraint SLO while sacrificing an acceptable amount of throughput. In addition, we aim to derive the maximum sustainable rate given the batch profile and SLOs.

To evaluate our policies, we plan to perform simulations that model a machine learning inference system processing incoming queries. Aside from testing whether our policies satisfy the SLOs, we will also measure the throughput and compare it with those of existing approaches to assess the trade-offs among batch size, latency, and throughput.

3 Batch Adaptation Policies

In this section, we propose a novel batch adaptation policy that is designed to not only satisfy the latency SLO but also additional objectives that capture consecutiveness or frequency of request deadline misses. In Subsection 3.1, we introduce our system model and outline the notations used throughout the paper. In Subsection 3.2, we briefly restate the early-drop policy adopted by Nexus, and we make some observation on it that opens up the opportunity to satisfy additional objectives. In Subsection 3.3 and 3.4, we introduce the policy we devised and show how it could be used to satisfy two different objectives. In Subsection 3.5, we comment on how our policy could be applied to multi-model serving systems.

Subsection 4.2, shows that given a latency SLO and either of the two additional types of SLO, we are able to derive the maximum request rate that a system is able to sustain order to satisfy these SLOs. That is, as long as the request rate does not exceed this derived maximum request rate, the SLOs are guaranteed to be satisfied. In Subsection 4.3, we delineate a machine learning inference system for time-critical applications that utilizes spread-drop policy.

3.1 System Model and Notations

A sequence of homogeneous requests $\tau_1, \tau_2, ...$ is submitted to the system to be processed (perform inference) by the model. The request τ_i arrives at the system at time a_i, where $a_i \leq a_{i+1}$. The requests are queued up to be processed by the model. All the requests have latency SLO L. That is, request τ_i has the deadline $a_i + L$. The system processes requests in batches, where B is the batch size and P_B is the time it takes the system to process a batch with batch size B. We make the assumption that $2P_B \leq L$, which will be explained in a few paragraphs.

Let $\tilde{Q}_t = \{\tau_i : a_i \leq t\}$ denote the set of requests that has arrived before time t. Let $Q_t = \{\tau_i : a_i \leq t \wedge a_i + L \geq t + P_B\}$ denote the set of requests that has arrived at the system and are still able to meet their deadlines at time t. Intuitively, Q_t is the state of the request queue at time t after dropping all requests that are either processed, dropped, or unable to meet their deadlines.

We also define the **candidate set** at time t to be $C_t = \{\tau_i : a_i \leq t \wedge a_i + L \geq t + P_B \wedge a_i + L \leq t + 2P_B\}$. Intuitively, with the assumption $2P_B \leq L$, this is the set that contains all requests such that, if the model processes a batch of requests at time t and if $\tau \in C_t$ is not included in this batch, τ will miss its deadline. Since $|C_t|$ may be greater than B, some of these requests are subject to being dropped.

The runtime scheduler is described in Algorithm 1. In order to accumulate requests to form a batch, the scheduler waits until $t = a_i + L - P_B$, where τ_i is the earliest-arrived request currently in the queue after dropping all requests that are not able to meet their deadlines. The scheduler then forms a batch according to some policy and then submits the batch to the processor for inference. Note that the batch-adaptation policies we introduce in Subsection 3.2, 3.3 and 3.4 can be plugged into Line 9.

Revisiting the assumption $2P_B \leq L$: the candidate set contains requests τ_j that satisfy $a_j \leq a_i + L - P_B$, which means that they arrive before the batch is formed and processed, and $a_j \leq a_i + P_B$, which captures that if the requests are not processed now, they will miss the deadline. In order for the candidate set to be "complete" (i.e. contains all such requests), we need $a_i + P_B \leq a_i + L - P_B$, which is $2P_B \leq L$.

Algorithm 1 SCHEDULER

Input: Latency SLO L, Batch size B, Processing time P_B

while TRUE **do**
  Drop all requests in the request queue that are not able to meet deadline
  if request queue is empty **then**
    Wait until a request arrives in the request queue
  end if
  $\tau_i \leftarrow$ The earliest-arrived request currently in queue
  $t \leftarrow a_i + L - P_B$
  Wait until time t
  $b \leftarrow$ form_batch_by_policy(B, Q_t, C_t)
  Send batch b to processor
end while

Algorithm 2 EARLY-DROP

Input: Batch size B, request queue $Q = \{\tau_i, \tau_{i+1}, ..., \tau_{i+|Q|-1}\}$, Candidate set $C = \{\tau_i, \tau_{i+1}, ..., \tau_{i+|C|-1}\}$

$b \leftarrow \{\tau_i, \tau_{i+1}, ..., \tau_{i+min(B,|Q|)-1}\}$
Drop all requests in $C - b$
return b

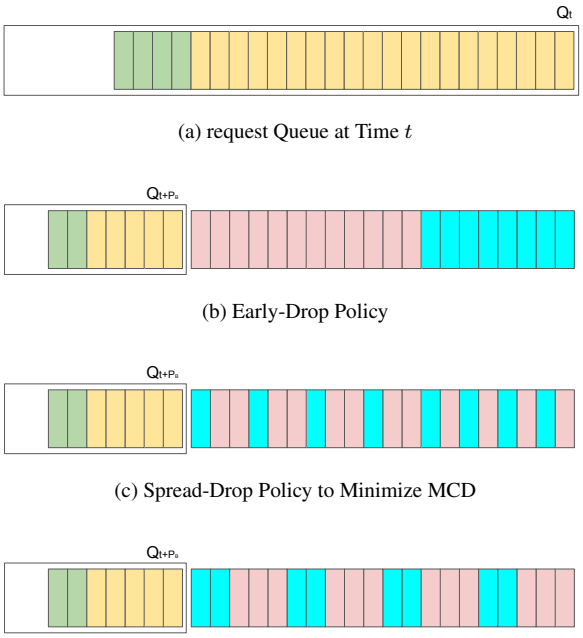
3.2 Improvement on Early-Drop Policy

The early-drop policy is described in Algorithm 2. It forms a batch $b = \{\tau_j, ..., \tau_{j+\min(B,|Q_t|)-1})\}$ by selecting the first $\min(B, |Q_t|)$ requests in the request queue. The system then processes this batch. This will result in the requests $\tau_{j+\min(B,|Q_t|)}, ..., \tau_{j+|C_t|-1}$ being dropped, if they exist. This is illustrated in Figure 1b, where the policy forms a batch by selecting the first 8 requests and dropping the following 12 requests.

We observe that there is flexibility in the selection of requests to form a batch. By spreading out the selected requests, the system can avoid consecutive deadline misses or drops. We coin this as the **spread-drop policy**. This policy could be adapted to satisfy different additional objectives. For the following two subsections, we delineate the spread-drop policies for minimizing MCD and for satisfying weakly-hard constraints.

3.3 Spread-Drop Policy: Minimizing MCD

The spread-drop policy to minimize MCD is described in Algorithm 3. Intuitively, it spreads out the request selected by repeating the pattern: drop at most $\lceil \frac{|C|}{B} \rceil - 1$ consecutive requests and select the next request. This



(a) request Queue at Time t

(b) Early-Drop Policy

(c) Spread-Drop Policy to Minimize MCD

(d) Spread-Drop Policy to Satisfy Weakly-Hard Constraint $(3, 5)$

Figure 1: Example request queues at time t (Figure 1a) and $t + P_B$ with different drop policies (Figure 1b, 1c and 1d) and batch size 8. Yellow requests are requests in candidate sets. Green requests are requests that are in the queue but not in candidate sets. Blue requests are requests that are selected to form a batch and processed at time t. Red requests are requests that are dropped because they are unable to meet their deadlines at time $t + P_B$.

results in MCD being $\lceil \frac{|C|}{B} \rceil - 1$. This is illustrated in Figure 1c. For the first 8 requests, the policy selects every other request. For the following 12 requests, the policy repeats dropping 2 requests and selecting the next request. In the end, 8 requests are selected to form a batch, while the other 12 are dropped. Note that the policy results in the same amount of requests being processed and dropped as in the early-drop policy.

As the spread-drop policy considers the same candidate sets and processes and drops the same amount of requests as the early-drop policy, the spread-drop policy achieves the same throughput and bad rate as the early-drop policy, but with significantly lower MCD.

3.4 WH-Drop Policy: Satisfying Weakly-Hard Constraint

The weakly-hard constraint provides a more relaxed deadline by allowing missed deadlines but enforces weaker constraints [1]. The weakly-hard constraint may be more robust in certain real-time applications because

Algorithm 3 SPREAD-DROP: MINIMIZING MCD

Input: Batch size B, Candidate set $C = \{\tau_i, \tau_{i+1}, ..., \tau_{i+|C|-1}\}$

if $|C| \bmod B == 0$ **then**
  $b \leftarrow \{\tau_j \in C : (j-i+1) \bmod \frac{|C|}{B} = 0\}$
else
  $x \leftarrow \frac{B\lceil\frac{|C|}{B}\rceil - |C|}{\lceil\frac{|C|}{B}\rceil - \lfloor\frac{|C|}{B}\rfloor}$
  $b \leftarrow \{\tau_j : j-i+1 \leq x\lfloor\frac{|C|}{B}\rfloor \wedge (j-i+1) \bmod \lfloor\frac{|C|}{B}\rfloor = 0\}$
  $b \leftarrow b \bigcup \{\tau_j : j-i+1 > x\lfloor\frac{|C|}{B}\rfloor \wedge (j - x\lfloor\frac{|C|}{B}\rfloor + 1) \bmod \lceil\frac{|C|}{B}\rceil = 0\}$
end if
Drop all requests in $C - b$
return b

Algorithm 4 SPREAD-DROP: SATISFYING WEAKLY-HARD CONSTRAINTS

Input: Batch size B, Candidate set $C = \{\tau_i, \tau_{i+1}, ..., \tau_{i+|C|-1}\}$, Weakly-Hard Constraint (m, K)

$b \leftarrow \{\tau_j : j-i+1 \leq K\lfloor\frac{|C|-B}{m}\rfloor \wedge (j-i+1) \bmod K \geq m\}$
$b \leftarrow b \bigcup \{\tau_j : j-i+1 > K\lfloor\frac{|C|-B}{m}\rfloor + ((|C|-B) \bmod m)\}$
Drop all requests in $C - b$
return b

it bounds the frequency of drops, which is crucial when consecutive deadline misses may be undesirable.

Definition 1. We say that a system satisfies a **weakly hard constraint** (m, K) if there are at most m request deadline misses among any K consecutive requests.

Given a weakly-hard constraint (m, K), assuming that $|C_t| \leq \lfloor\frac{B}{K-m}\rfloor \times K + (B \bmod (K-m))$ for any t (the assumption is explained in the proof of Theorem 2), we propose Algorithm 4. Intuitively, the policy repeats the pattern to satisfy the weakly-hard constraint: drop at most m requests, and select the next $K-m$ requests. This ensures that no more than m requests are dropped within any K consecutive requests. This is illustrated in Figure 1d. To satisfy the weakly-hard constraint $(3, 5)$, the policy selects the last 2 requests in every 5 requests. In the end, 8 requests are selected to form a batch, while the other 12 are dropped. Note that the request dropping pattern in Figure 1c does not satisfy the weakly-hard constraint $(3, 5)$, even though it has an MCD of 3.

3.5 Extensibility To Multi-Model Serving

In this paper, we consider homogeneous requests that are processed on the same model. We note that, however, the proposed batch adaptation policies can also be applied to a multi-model serving system like Nexus. Assume that the processor executes batches of different types of models in a round-robin manner over a time period of time D, which we refer to as the **duty cycle** [5]. By modifying the assumption in Subsection 3.1 to be $2D \leq L$, and additional modifications on the waiting time of the scheduler to be aware of the duty cycle, the policies are able to lower MCD or satisfy weakly-hard constraint on a per-model basis.

4 On Providing Guarantees

For batch adaptation policies to be useful for time-critical systems, guarantees on the satisfaction of SLOs have to be provided. Aside from a latency SLO L, which specifies that a request to be processed within L time units, we also consider two additional types of SLO: MCD SLO M, which specifies that no more than M requests be dropped consecutively; and weakly-hard constraint SLO (m, K), which specifies that no more than m requests be dropped among any K consecutive requests. In Subsection 4.1 and 4.2, we show that given a latency SLO and either of the two additional types of SLO, we are able to derive the maximum request rate that a system is able to sustain order to satisfy these SLOs. That is, as long as the request rate does not exceed this derived maximum request rate, the SLOs are guaranteed to be satisfied. In Subsection 4.3, we delineate a machine learning inference system for time-critical applications that utilizes spread-drop policy.

4.1 Maximum Request Rate to Satisfy MCD SLO

Theorem 1. *Given a latency SLO L, a MCD SLO M, batch size B and processing time P_B, a system with a runtime scheduler described in Algorithm 1 using Algorithm 3 as batch-adaptation policy is able to satisfy the SLOs given that no more than $B \times (1+M)$ requests arrive within any P_B unit time interval. That is, the maximum instantaneous request rate is $\frac{B\times(1+M)}{P_B}$ requests per time unit.*

Proof. According to Algorithm 1, a batch is formed at most every P_B time unit. With the number of requests not exceeding $B \times (1+M)$ in any P_B time unit, we have that $|C_t| \leq B \times (1+M)$ at any time t, so Algorithm 3 drops $\lceil\frac{|C_t|}{B}\rceil - 1 \leq \lceil\frac{B\times(1+M)}{B}\rceil - 1 = M$ consecutive requests. □

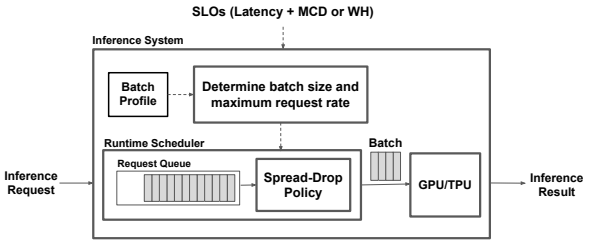


Figure 2: Proposed inference system.

4.2 Maximum Request Rate to Satisfy Weakly-Hard Constraint SLO

To satisfy weakly-hard constraint SLO, we just need to make sure the assumption made in Subsection 3.4 is satisfied.

Theorem 2. *Given a latency SLO L, a weakly-hard constraint SLO (m, K), batch size B and processing time P_B, a system with a runtime scheduler described in Algorithm 1 using Algorithm 4 as batch-adaptation policy is able to satisfy the SLOs given that no more than $\lfloor \frac{B}{K-m} \rfloor \times K + (B \bmod (K-m))$ requests arrive within any P_B unit time interval. That is, the maximum instantaneous request rate is $\frac{\lfloor \frac{B}{K-m} \rfloor \times K + (B \bmod (K-m))}{P_B}$ per time unit.*

Proof. As Algorithm 4 repeatedly drop at most m tasks and select $(K-m)$ tasks, $|C_t|$ cannot exceed $\lfloor \frac{B}{K-m} \rfloor \times K + (B \bmod (K-m))$ in order to select B tasks. Therefore, as long as $|C_t|$ does not exceed $\lfloor \frac{B}{K-m} \rfloor \times K + (B \bmod (K-m))$ for any time t, the SLOs are satisfied. According to Algorithm 1, a batch is formed at most every P_B time unit, so as long as there are more no than $\lfloor \frac{B}{K-m} \rfloor \times K + (B \bmod (K-m))$ within any P_B unit time interval, the SLOs are satisfied. □

4.3 Proposed System

With spread-drop policy and its guarantees, we propose a machine learning inference system illustrated in Figure 2. Given SLOs and a batch profile, which is provided either by the system administrator or through an existing profiler, the system determines the maximum batch size B such that the assumption $L \geq 2P_B$ is satisfied. The maximum batch size is chosen in order to maximize throughput. It also outputs the maximum request rate as derived in Subsection 4.1 and 4.2. Requests are submitted to the system and kept in the request queue of the runtime scheduler. The runtime scheduler adopts spread-drop policy to perform adaptive batching and sends the formed batch to GPU or TPU for inference, which outputs the inference results.

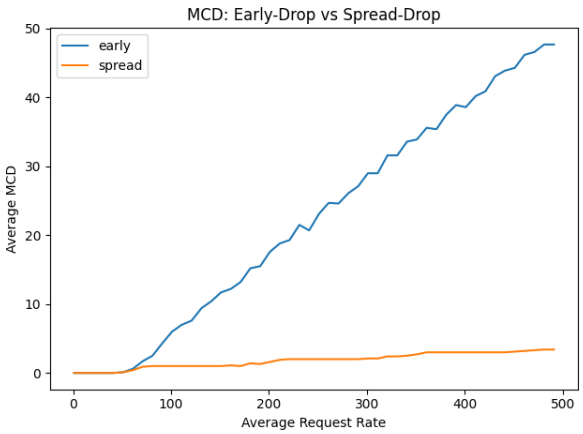


Figure 3: Average of MCD of spread-drop policy and early-drop policy.

The system administrator could utilize the maximum request in various ways based on the application. Request rate control could be applied on the input side so that the request rate does not exceed the maximum request rate and thus ensuring the SLOs never be violated. If request rate control is infeasible for the application, the system could be fallen back into a "safe mode" to lower the probability of system failure. Using the same example of an object detection model on an autonomous vehicle, if the request rate exceeds the maximum request rate, the system could hand back the control to the human driver, since the object detection system is subject to failure.

5 Evaluation

To evaluate our policies, we conduct simulations that model an inference system serving requests. All simulations are conducted on a MacBook Pro 14 with an 8-core CPU, 16 GB memory, and 512 GB SSD storage.

5.1 Comparison to Early-Drop Policy

The first experiment compares how spread-drop policy improves over early-drop policy with respect to MCD. For each request rate, we run 10 trials of simulations, each of which is a simulation of 10000 requests that follows a Poisson process with mean set to $0.9\times$ corresponding request rate. We compute the average of MCD over 10 trials for both policies. The result is depicted in Figure 3. The spread-drop policy achieves a significantly lower average MCD compared to the early-drop policy, and the discrepancy between them becomes clearer as the request rate increases. As discussed in Subsection 3.3, the spread-drop policy achieves the same throughput and

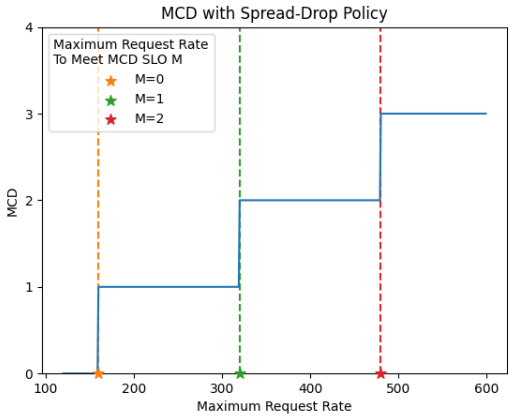


Figure 4: Maximum request rate a system could sustain in order to meet different MCD SLOs.

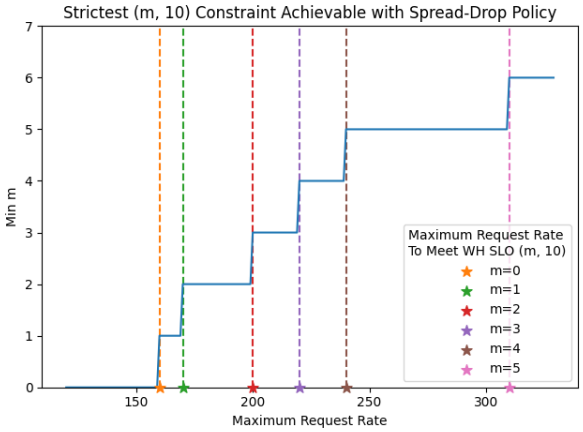


Figure 5: Maximum request rate a system could sustain in order to meet different weakly-hard constraint SLOs.

bad rate as the early-drop policy, but, as shown in this experiment, with significantly lower MCD.

5.2 Verifying the Maximum Request Rate to Satisfy MCD SLO

The second experiment aims to verify the correctness of Theorem 1, the derived maximum request rate for the spread-drop policy to satisfy an MCD SLO. For each request rate, we run a single trial of simulation with 10000 requests that arrive at the system with a fixed interval that corresponds to the request rate, and we measure the MCD. The result is shown in Figure 4. The vertical lines are the maximum request rate to satisfy MCD SLOs $M = \{0, 1, 2\}$ derived from Theorem 1. As shown in the figure, as long as the request rate does not exceed the derived maximum request rate to satisfy MCD SLO M, the MCD M' obtained from our simulation does not exceed M.

5.3 Verifying the Maximum Request Rate to Satisfy Weakly-Hard SLO

Similar to the second experiment, the third experiment aims to verify the correctness of Theorem 2, the derived maximum request rate for the spread-drop policy to satisfy a WH SLO. The simulation setup is the same as Subsection 5.2, while we measure the "strictest" weakly-hard constraint the system is able to satisfy given $K = 10$. That is, the minimum m satisfiable with weakly-hard constraint SLO $(m, 10)$. The result is shown in Figure 5. The vertical lines are the maximum request rate to satisfy weakly-hard constraint SLOs $(m, 10)$ with $m = \{0, 1, 2, 3, 4, 5\}$ derived from Theorem 2. As shown in the figure, as long as the request rate does not exceed the derived maximum request rate to satisfy weakly-hard constraint SLO $(m, 10)$, the weakly-hard constraint $(m', 10)$ is satisfied in our simulation for all $m' \leq m$.

6 Future Work

Future work on the spread drop policy could include implementing the policy in an existing model inference system, such as Nexus. This would allow for the evaluation of the policy in a real-world setting, the comparison to other existing batch policies, and could potentially lead to the identification of any challenges or limitations that may arise when applying the policy in a production environment. Additionally, this could provide an opportunity to test the effectiveness of the policy in combination with other optimization techniques, such as model compression or hardware acceleration. By implementing the policy in an existing model inference system, we can gain a better understanding of how the policy performs in a real-world setting and identify any areas for improvement.

Another potential direction for future work would be to investigate ways to minimize consecutive drops per user, rather than just overall consecutive drops. This could involve adapting the batch adaptation policy on a per-user basis, taking into account the unique characteristics and needs of each user. This could potentially lead to more personalized and effective batch adaptation policies, and may be especially useful in situations where users have different usage patterns or requirements.

For example, let's say we have two users: User A and User B. User A has a high volume of requests and is more sensitive to latency, while User B has a lower volume of requests and is less sensitive to latency. Our current batch

adaptation policy may be effective at reducing consecutive drops for User A, but may not be as effective for User B. By adapting the policy on a per-user basis, we could potentially optimize the policy for User A to minimize consecutive drops while still maintaining a high throughput, and adjust the policy for User B to prioritize other factors such as energy efficiency. This could lead to a more effective and personalized solution for both users.

A third area of research could be focused on developing a better policy for satisfying weakly-hard constraints, such as constraints on latency or energy consumption. Currently, our proposed approach focuses primarily on reducing consecutive drops while maintaining throughput, but there may be situations where other constraints are more important. Developing a policy that can effectively balance the trade-offs between different constraints could lead to more flexible and robust machine learning inference systems.

One potential approach to addressing weakly-hard constraints could involve using machine learning algorithms to learn and adapt to the unique characteristics and requirements of each constraint. For example, we could train a machine learning model to predict the impact of different batch adaptation policies on latency and energy consumption, and use this information to optimize the policy in real-time. This could involve using techniques such as reinforcement learning or evolutionary algorithms to adapt the policy over time as the system and constraints evolve.

Overall, there are many potential directions for future work on batch adaptation policies in machine learning inference systems, and these are just a few examples of the types of research that could lead to further improvements in our proposed approach. So, there is still a lot of scope for research in this field.

7 Related Work

There has been a significant effort in research to improve real-time systems scheduling, specifically pertaining to machine learning model serving. Since real-time applications require latency constraints, the goal of this research area focuses on decreasing latency while increasing throughput and accuracy. To this end, some seminal work we benchmark against our work with are TensorFlow-Serving [4], Clipper [2], and Nexus [5]. These are prominent machine learning prediction serving systems employed by many end-users in both academia and industry.

TensorFlow-Serving is an open-source machine learning serving model developed by Google to be employed in production at scale. Its system is built to be compatible with TensorFlow models out-of-the-box, but it also is compatible with other machine learning model systems. Clipper is closely related to TensorFlow-Serving as they are both machine learning serving models meant to serve diverse frameworks and applications. TensorFlow aims to be robust when deployed at scale as it is intended to be the serving model for Google. It includes the ability for model versioning, which is important for version rollbacks needed during production. However, there are several drawbacks to TensorFlow-Serving. The main drawback pertinent to this paper is that TensorFlow lacks the ability for the end-user to specify the latency SLOs for the requests. Therefore, efficient scheduling before production such as bin-packing or complex queries employed by Nexus would not be possible [5]. Clipper is another prediction serving system with a similar model to TensorFlow-Serving. However, Clipper confers several advantages over TensorFlow-Serving such as the option for caching and SLO specifications. The goal of Clipper is to provide a low-latency serving model that supports model composition and online learning [2]. To this end, Clipper delivers a modular architecture so it can be easily deployed across different applications and frameworks. One feature important to this paper is Clipper's support for dynamic batching. Clipper aims to increase throughput by adopting a dynamic policy for batch sizes while also adhering to the latency requirements specified by the application. It employs an algorithm named the additive increase multiplicative decrease (AIMD) [2]. With AIMD, Clipper increases the batch size additively until the latency requirements can no longer be maintained. At which point, Clipper, multiplicatively decreases batch size until the latency requirements are attainable again. A drawback of this approach is that AIMD employs lazy dropping which is shown to unnecessarily increase the drop rate whenever the input rate is very high. In order to accommodate the latency requirements, a high input rate may force Clipper to decrease the batch size significantly. However, an unsuitably small batch size may increase overhead and thus decrease throughput. Additionally, Nexus is another machine learning serving model that optimizes the batch size to decrease latency [5]. In particular, Nexus is developed to serve DNN-based video analysis. As a result, the latency requirements are of special interest due to the nature of video-based applications. To this end, Nexus employs an algorithm they coined "squishy bin-packing" to efficiently allocate models onto GPUs. Furthermore, Nexus is able to handle request dependencies by efficiently deciding the latency requirements for different requests of a DNN request dependency graph. Of special interest to this paper is the dropping policy aimed at decreasing the rate of dropped batches. Rather than employing lazy dropping, which Clipper uses, Nexus employs early dropping to avoid decreased throughput. Another advantage of early

dropping is that it avoids the pitfall of unnecessarily decreasing the batch size that lazy dropping suffers from. From both simulations and experimental results, Nexus has been shown to offer better performance than Clipper in part due to the switch from lazy dropping to early dropping.

The above-mentioned machine learning serving models aim to increase throughput while meeting latency requirements. It can be seen that batch policies such as dynamic batching through batch dropping drastically affects performance. In this paper, we aim to study and improve existing dropping policies that may maintain low latency while not significantly decreasing accuracy. One application targeted by more robust SLO policies is real-time machine learning serving systems used for video analysis models, as in the case of Nexus. This paper aims to explore several algorithms in which simulations are employed for benchmarking against the state-of-the-art.

8 Conclusion

In this project, we designed the spread-drop policy, a batch adaptation policy for time-critical applications. It is able to satisfy two additional SLOs that capture the frequency and consecutiveness of request deadline misses. We proposed a machine learning inference system with a runtime scheduler using the spread-drop policy, and we derived the maximum request rate such system is able to sustain in order to satisfy the additional SLOs.

References

[1] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, 2001.

[2] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, Mar. 2017. USENIX Association.

[3] H. Fujiyoshi, T. Hirakawa, and T. Yamashita. Deep learning-based image recognition for autonomous driving. *IATSS Research*, 43(4):244–252, 2019.

[4] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[5] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[6] M. Shirobokov, S. Trofimov, and M. Ovchinnikov. Survey of machine learning techniques in spacecraft control design. *Acta Astronautica*, 186:87–97, 2021.

[7] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, Mar. 2017. USENIX Association.

[8] X. Zhao, C. Gu, H. Zhang, X. Yang, X. Liu, J. Tang, and H. Liu. Dear: Deep reinforcement learning for online advertising impression in recommender systems. In *AAAI*, 2021.