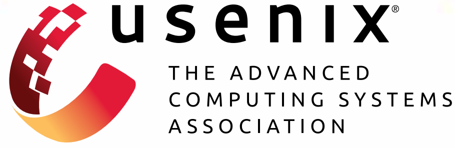


QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models

Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman, *Meta, Inc.*

https://www.usenix.org/conference/nsdi24/presentation/matam

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models

Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong,
Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman

Meta, Inc.

Abstract

Deep learning recommendation models play an important role in online companies and consume a major part of the AI infrastructure dedicated to training and inference. The accuracy of these models highly depends on how quickly they are published on the serving side. One of the main challenges in improving the model update latency and frequency is the model size, which has reached the order of Terabytes and is expected to further increase in the future. The large model size causes large latency (and write bandwidth) to update the model in geo-distributed servers. We present QuickUpdate, a system for real-time personalization of large-scale recommendation models, that publishes the model in high frequency as part of online training, providing serving accuracy that is comparable to that of a fully fresh model. The system employs novel techniques to minimize the required write bandwidth, including prioritized parameter updates, intermittent full model updates, model transformations, and relaxed consistency. We evaluate QuickUpdate using real-world data, on one of the largest production models in Meta. The results show that QuickUpdate provides a serving accuracy that is comparable to a fully fresh model, while reducing the average published update size and the required bandwidth by over 13x. It provides a scalable solution for serving production models in real-time fashion, which is otherwise not feasible at scale due to the limited network and storage bandwidth.

1 Introduction

Deep Learning Recommendation Models (DLRM) are widely used in many online companies. These models are trained using data at scale to learn user and product characteristics, providing personalized recommendations in a variety of contexts. For instance, Netflix [7] and YouTube [4] provide lists of movies for customers; Amazon [19] and Alibaba [20] recommend relevant products based on user search queries, and Google [3] and Meta [23] display ads and contents according to user interests. DLRMs consume a major part of the AI infrastructure in these companies. In Meta, for example, DLRMs consume more than 80% of the machine learning inference cycles [8] and more than 50% of the training cycles.

Recommendation models help the business grow. For instance, they contribute to 35% of the entire purchase in Amazon [8, 14]. As a result of such extensive business impact, accuracy becomes an important performance metrics for recommendation models at scale. In particular, Meta business required accuracy loss to be less than 0.01% in designing checkpoint and quantization algorithms [5]. This is a very narrow margin, indicating the importance of recommendation models and their accuracy.

Model freshness is a key contributor to the accuracy of personalized recommendation models [4, 6, 9, 22, 25]. The accuracy can deteriorate rapidly because the models run inferences in very highly dynamic environments. For instance, every day new users and items are registered in the system and user interests may be impacted by recent events. If the model is not updated frequently, it would not incorporate the changes in users and products, leading to a gradual deterioration in accuracy. To further highlight the impact of freshness, Figure 3 displays the significant accuracy loss when the model is not refreshed for hours. Thus, in order to keep accuracy at an acceptable level, recommendation models need to be retrained using the most recent data, and the updated models should be used to serve real time inferences.

A common technique to keep inference models fresh is online training. Instead of re-training the model from scratch each time, it is continuously trained and refined using real-time streaming data. Periodically, a snapshot of the model is created and published to hundreds of servers, located across different geographical regions. These servers then utilize the model to perform real-time predictions for online queries. However, updating the serving model incurs latency between the training cluster and the distributed serving hosts, resulting in a delay in refreshing the model, primarily due to the large size of modern models.

Over the years, model sizes have grown rapidly, reaching the scale of terabytes and containing trillions of param-

eters [5, 10, 15] to capture millions of sparse features and improve the model accuracy. The limited write bandwidth poses a challenge when transferring such large models to distributed servers and storage. As a consequence, the update latency can extend to the order of hours. This prolonged latency can adversely affect accuracy, as discussed in more detail in section 3.

To address the above challenges posed by the large model sizes and their consequent update latency, we present QuickUpdate. QuickUpdate employs the following design elements to provide real-time personalization of large-scale DLRM:

1. **Prioritized parameter update:** Performing a complete update of all parameters in each of the serving models across hundreds of geo-distributed nodes, would require substantial network and storage bandwidths, which constitute a bottleneck.
   QuickUpdate minimizes the update size by performing prioritized parameter selection. It ranks and selects specific parameters to be updated in the serving model while pruning the remaining ones from the update. This approach significantly reduces the overall update size and mitigates the bandwidth demands.
   The parameter ranking algorithm is important to avoid accuracy degradation when minimizing the update size.
2. **Intermittent full model update:** Intermittent full model updates occur when a series of consecutive partial updates are followed by a complete model update. The primary purpose of these full updates is to maintain long-term accuracy in the serving model. After each partial update, the serving model deviates from the training model, since the former still utilizes outdated parameter values. This deviation grows larger with more partial updates, leading to potential accuracy implications over time. To improve accuracy, a full model update is published intermittently to limit the gap between the serving model and the training model.
3. **Model transformations for real-time update:** QuickUpdate employs several model transformations to reduce the published model size, including inference pruning and quantization.
   Quantization has been successfully implemented in some studies [11, 24, 27] to reduce floating-point precision without sacrificing accuracy. It helps to reduce the required size of storage in the inference cluster and the required communication cost. Inference pruning is implemented on very large look-up tables. Entities, such as users or videos, and their corresponding vectors are stored as look-up embedding tables in DLRMs. The entity indices (or IDs) that are practically inactive are pruned from the serving platform to significantly reduce the size of the served model.
4. **Simplified serving design and relaxed consistency requirements:** In traditional serving designs, the model is fully loaded in the serving platform before starting to serve queries, to maintain strong consistency. In such designs, each inference request is executed based on a specific version of the model weights, ensuring consistent and reliable results. However, this approach incurs considerable infrastructure overhead due to the use of extra buffer nodes.
   In QuickUpdate, we introduce a more efficient serving design by relaxing the consistency requirements. Instead of using buffer nodes, the weights are directly updated in the serving nodes. This eliminates the need for extra infrastructure and reduces overhead. However, this relaxed design may result in some inconsistency in the embedding tables, as they may contain a mixture of fresh and stale weights.
   Despite the potential inconsistency in the embedding tables, our evaluation demonstrates that the accuracy of the serving model is not compromised, but rather it leads to accuracy gains.

We evaluate QuickUpdate using real-world data and one of the largest models that is deployed in production at Meta. Overall, our results demonstrate that QuickUpdate is able to provide a serving accuracy that is comparable to a fully fresh model, while minimizing the required write bandwidth by over 13x. It provides a scalable solution for serving production DLRM in real-time fashion, which is otherwise not feasible at scale due to the limited network and storage bandwidth. QuickUpdate achieves this by leveraging novel techniques, including selectively publishing the most important parts of each update, while still incorporating low-frequent intermittent full model updates that ensure long term accuracy.

2 Background

2.1 Deep Learning Recommendation Models (DLRM)

Typically, deep learning recommendation models are composed of sparse and dense layers, as displayed in Figure 1 [5, 10, 26]. Sparse layers are practically the embedding tables, where each embedding table represents a categorical feature and each row of this table represents a specific ID (e.g., user ID or video ID). The embedding table transforms each ID into a fixed size vector with float values that are trainable. The remaining trainable parts of the model are called dense layers.

Figure 1 displays how data flows in a DLRM. Sparse features are transformed by embedding tables; and dense features are transformed by the bottom dense layer. The transformed features are then concatenated and further transformed in the top dense layer to compute a likelihood for the input data.

2.1.1 Training DLRM

Parallelization is the main approach to train recommendation models at scale [5, 8]. Different parallelization logic can be implemented for the sparse and dense layers. Sparse layers

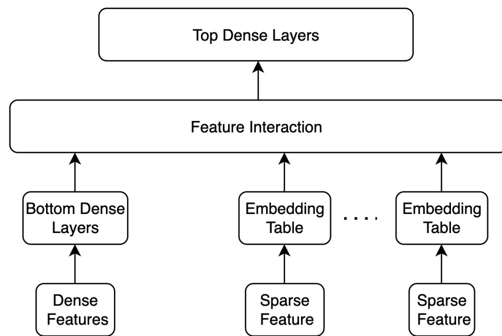


Figure 1: A DLRM architecture

contribute to $> 99\%$ of the entire model size and can be in the order of several terabytes. Because storing all the sparse layers in a single node is not feasible, a model parallelism approach is used to shard the tables on several nodes. The dense layer size, on the other hand, is small enough to be accommodated by each node, thus they are replicated across the nodes to leverage data parallelism.

At Meta, a typical training cluster includes 16 nodes, where each node contains a multi-socket CPU and 8 GPUs. Sparse layers are sharded across all the GPUs, with "all-to-all" communication during both forward and backward computations. Dense layers are replicated across all the GPUs, with "all-reduce" communication to aggregate gradients computed in multiple GPUs during the backward propagation [16]. During training, the new weights are computed and updated for sparse and dense layers synchronously to avoid accuracy degradation.

2.1.2 Serving DLRM

To efficiently serve batches of requests in a high-throughput manner, GPUs are typically used for model serving. In Meta, serving nodes are located in dedicated serving clusters. A serving node consists of a host CPU and GPUs attached to it. The serving model is replicated across the serving nodes, and data parallelization is used for model serving at scale. Ads embedding tables are stored in a single GPU because they require higher read throughput. The other embedding tables are stored in the CPU, which typically has much larger memory capacity (e.g., 1.5 TB DRAM). For storing the embedding tables, a compact data structure is used to minimize the size and store it in a GPU access friendly manner. In particular, the embedding tables are stored consecutively, and each embedding table is stored with row-major order in an array data structure.

To refresh the serving model, additional buffer nodes are utilized to avoid pausing the currently serving nodes. After the newly published model is loaded into a buffer node, request traffic is switched to the buffer node.

2.2 Online versus Offline Training

Online training is implemented when the serving model needs to be continually trained and updated using a real-time stream of data. In online training, the serving model provides predictions while being updated in regular time intervals (in the order of minutes to hours). The training continues to operate in the background (typically in a separate cluster) to fine tune the model. The rate at which the trained model is published to the serving platform can have a significant impact on the accuracy of the predictions it generates.

In contrast, offline training does not use a real-time stream of data for training and does not have a tight time constraint to train and publish the model to the serving platform. Instead, it usually uses a bulk of data that is already stored in a data storage. The model is trained using the entire available data, and the training stops when certain optimality conditions are satisfied, after which it is published to the serving platform.

Deciding between online and offline training depends on the use case. Online training systems are implemented when the model needs to be updated in a timely fashion. A typical use case could be DLRMs for ads, search and videos (e.g., [13, 18, 21]) when the environment is highly dynamic and requires the model to be updated almost in real time (in the order of minutes). The online training helps these models to incorporate the most recent data and to avoid accuracy degradation. When the business requirements do not justify real time model update, one can use an offline training approach to update models.

2.3 Optimizer State as Feature Importance Measure

In general, large DLRMs can use hundreds of thousands of features for training; however, some of these features and corresponding weights do not impact the accuracy. These features may belong to inactive users or content IDs, or some other features may not provide training signals. Maintaining all these parameters would consume some extra bandwidth when the model is published, or some extra computation and GPU storage when the inference is run in the serving platform. To mitigate these adverse impacts, we can compute feature importance and accordingly prune the features and weights that practically do not impact the accuracy.

QuickUpdate uses optimizer state (or gradient momentum) to compute feature importance. It belongs to the family of gradient-based feature importance metrics (e.g., [2, 12, 18]). The optimizer state is the historical average of gradient values and is more stable than gradient values, which sometimes oscillate between positive and negative values. The optimizer state can give us the following indications:

1) Impact on accuracy: it practically shows how often and how much a specific parameter has been updated during the course of training. A high value of gradient value shows a

high impact on improving accuracy; and if this impact persists historically, we can more confidently deduce that the parameter is important for accuracy; thus the optimizer state would be a stable indication of feature impacts on accuracy.

2) Access rate: The optimizer state of zero (or near zero) indicates that the parameter has not been practically updated during the course of training. This has twofold implications. First, it can imply that the parameter has not been accessed. This can happen for some inactive user IDs or outdated videos. Second, if the entity is active, near-zero values can imply that there may not be important signals to learn from the relevant data. For instance, it can happen for a specific user ID that is not using the platform to click on ads.

Based on the information above, QuickUpdate uses optimizer state for two different tasks: 1- to perform inference pruning when a complete model is published to reduce the size of the complete model update. In this stage, QuickUpdate focuses on low tail of optimizer state values and prunes the parameters that do not have impact on accuracy 2- to perform prioritized parameter selection when a partial update is published. In this stage, the QuickUpdate focuses on the high tail of the optimizer state values to update more important parameters.

2.4 Inference Pruning

Inference pruning is performed to reduce the size of the model when a complete snapshot is published to the serving platform. Pruning is particularly implemented on the look-up embedding tables and reduces their size by a significant amount (e.g. 50%). Since look-up tables compose more than 99% of the size of DLRMs, the pruning considerably reduces the size of the model without compromising the accuracy. Reducing the size of the model helps to consume smaller bandwidth to publish the model updates; and as a result, they can be published to hundreds of geo-distributed clusters with shorter latency. Additionally, it helps to perform inference faster because fewer number of rows are involved in the computations.

Figure 2 displays an example of a look-up table which includes indices and the corresponding row. Each index represents a unique ID which is assigned to users, videos, etc. Each row can be considered as a vector of trainable float values used by the model to generate personalized recommendations for users.

Intuitively, the inference pruning algorithm identifies the rows that represent entities that are inactive or do not provide training signals to improve accuracy. Mathematically, this is accomplished using the optimizer state vector. In particular, the trainer can provide a vector of optimizer state for each row. Each element in the optimizer state vector demonstrates the gradient momentum of the corresponding element in the row. The average of the elements in the optimizer state vector is used to quantify the row importance value. If the row importance value is close to zero, it implies that the row elements

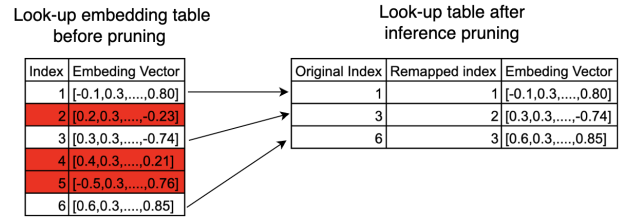


Figure 2: Inference pruning of embedding tables

have not been practically updated during the training; and as a result that row can be pruned.

Figure 2 also shows how the lookup table changes before and after pruning. Using the row importance values, the inference pruning algorithm determines the least important indices, and prunes them before publishing the complete update to servers. For operational purposes, the original index values are remapped to new indices. The new indices are simply incremented as they appear in the original table.

It should be noted that in this paper, a complete model update refers to the model snapshot after performing inference pruning.

3 Motivation

In this section, we present real-world data that highlights the motivation behind the development of QuickUpdate. First, we demonstrate how accuracy considerably drops when the model is not updated for an hour or longer. Then, we discuss the implications of scaling up the model size. Without modifying the model publishing approach, we have to either accept prolonged update latency and diminishing accuracy, or invest heavily in infrastructure to keep the update latency consistent. Finally, we highlight the limitation of lossless model updates, emphasizing the need for prioritized updates.

Accuracy gain: Updating a full serving model is a time-intensive process that may span several hours. As a result, recent updates like user actions and interests (e.g., posting new stories or engaging with specific content) would not be reflected in the serving model for several hours, potentially reducing the model accuracy.

Figure 3 demonstrates how accuracy drops as we postpone the model update for one of the large scale models at Meta. It compares the accuracy loss of a stale serving model with different model update latencies (1 to 7 hours), to a fully fresh model. It shows that accuracy loss significantly increases when model updates are delayed, reaching a loss of more than 0.6% after 7 hours.

Reducing the update size can help accelerate the model updates and improve the accuracy of the serving model.

Model size: DLRM sizes have seen a marked increase over the past years. These models utilize a large amount of data and parameters to better understand user interests and

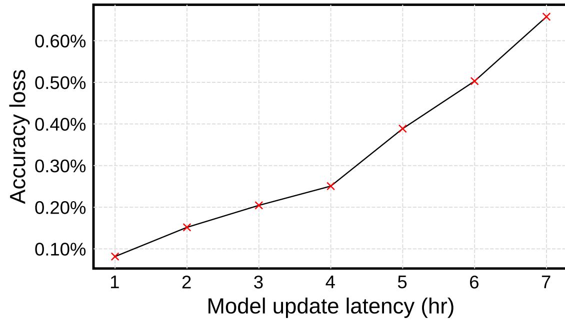


Figure 3: Accuracy loss in the case of different update latencies, when compared to a fully fresh model

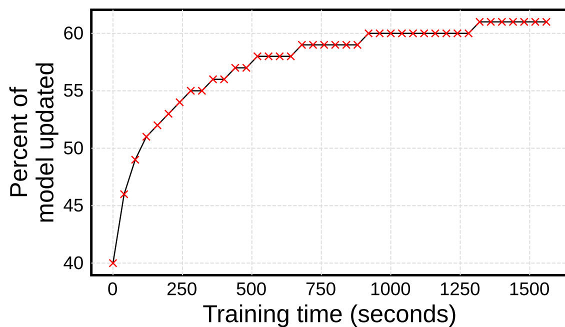


Figure 4: The percentage of the model updated over time

product characteristics, leading to improved accuracy. This progress has resulted in the development of complex models with trillions of parameters [15] and several terabyte model size [5]. Moreover, this trend is expected to continue in the future.

With the growth in model size comes the challenge of extended model update latency due to the increased bandwidth required for transfers. If left unaddressed, this is expected to lead to model freshness degradation in the future. Simply augmenting the infrastructure is not a viable long-term solution, given the relentless expansion of model sizes. Therefore, performing partial updates for large DLRM appears to be a promising strategy, aiming to reduce update latency without the demand for more infrastructure.

Lossless model updates: To better understand the proportion of the model that is modified over time, we monitored the updated embedding rows and accordingly computed the average fraction of the model touched. Figure 4 shows the percentage of the model updated over time. It is evident that a large part of the model is updated in a short span of time. For example, in just a 10-minute interval, 58% of the model gets updated. Updating 58% of the model is resource-intensive, requiring considerably more infrastructure than a full model update every hour. This leads us to explore an approach of prioritized updates to significantly reduce the update footprint.

4 System Overview

Figure 5 provides an overview of QuickUpdate architecture. The DLRM system consists of training nodes, serving nodes, and remote storage to save the model snapshots. The publishing logic of QuickUpdate has been mainly implemented in the UpdateSelector and UpdatePatcher agents, which are respectively implemented in the trainer node and serving node. The UpdateSelector is responsible to decide which portion of the model should be updated and quantize it before saving in the remote storage. UpdatePatcher implements different patching strategies depending on the type of update being performed. Additional information is provided in the following sections.

4.1 What to Update

QuickUpdate focuses on performing partial updates specifically for the embedding tables, which typically constitute the big majority of deep learning recommendation models (more than 99% in our workload). In such models, each table represents a categorical feature (e.g., users, videos), and each row within a table corresponds to a specific ID associated with that feature.

During our exploration, we considered two options for updating the embedding tables: 1. Updating all rows for selected tables. 2. Updating selected rows for all tables (the selected row indices may vary from one table to another).

We found that updating at the row level granularity resulted in improved accuracy while minimizing the overall update size. Consequently, QuickUpdate determines which specific rows within the tables need to be updated on the serving side. This approach enables QuickUpdate to prioritize the updates of certain content or user IDs that are more likely to contribute to accuracy gains, ensuring an efficient and effective update strategy. For dense layers in the model, QuickUpdate performs a full update. This is because the size of the update for these layers is relatively small, and any optimization specific to these layers would not have a significant impact on the overall update process.

4.2 UpdateSelector

The UpdateSelector component of QuickUpdate is implemented within the training cluster. This is because it requires certain model information, such as parameter values, from the trainer in order to prepare the model update.

During online training, the trainer operates in batch intervals. At the end of each training interval, the trainer shares both the model state and the optimizer state with the UpdateSelector. The model state includes sharded embedding tables and dense parameter values, while the optimizer state includes gradient values and their momentum. These states are copied from GPU memory to the host CPU memory.

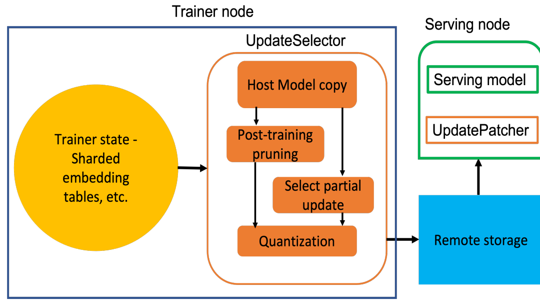


Figure 5: System architecture

UpdateSelector uses optimizer state to perform the following two tasks on the copied model in CPU:

1. Prioritized parameter selection: The primary objective of this task is to update only a small percentage of the model parameters, while minimizing any degradation in accuracy (compared to a full update). In this stage, QuickUpdate selects the embedding rows based on their optimizer state values, prioritizing those that are likely to result in a larger improvement in accuracy.
2. Inference pruning: This task is implemented when a full model is published. Inference pruning focuses on the sparse embedding tables and aims to reduce the size of the full model update. During this stage, QuickUpdate identifies the low-tail optimizer state values and prunes the embedding rows with values close to zero. These rows have negligible impact on the accuracy of the model.

Once the updates (whether full or partial) are prepared, they undergo quantization to reduce their size. The quantization is used as a compression method and has negligible impact on the model accuracy. The quantized updates are then stored in remote storage, ready to be utilized for the update process.

4.3 UpdatePatcher

UpdatePatcher is responsible for loading the published snapshots and updating the serving model. It utilizes an efficient non-atomic update approach for both partial and full model updates. In the non-atomic update process, multiple threads have access to the model parameters and gradually patch the parameters to the servers. This allows for concurrent parameter patching without the need to lock the servers or models. As a result, the servers can continue to run inference on incoming traffic simultaneously while the updates are being applied. This approach ensures efficient and uninterrupted serving of real-time traffic during the update process.

4.4 Workflow

Figure 6 demonstrates QuickUpdate workflow. For simplicity, we only show the timescale in the trainer, UpdateSelector, and a serving node. The evolution of the model is a repeatable pattern, thus we focus on one cycle, which is further divided into intervals. At the beginning of each interval i in the cycle c, UpdateSelector has access to a full model $F_{c,i}$ to determine which portion of the model should be updated. In particular, first, a full snapshot (i.e., $F_{c,1}$) will be published and loaded in the server, and then consecutive partial updates ($P_{c,i}$ for $i > 1$) will be published and patched to the full snapshot to create the serving snapshot $S_{c,i}$.

Merging more partial updates with the serving model may cause more deviation between the serving model $S_{c,i}$ and the current trainer state $F_{c,i}$. This deviation may result in accuracy degradation. As a result, another full fresh snapshot (i.e., $F_{c+1,1}$) will be published to the serving cluster, marking the end of the current cycle. This model evolution in the serving side can be represented as follows:

$S_{c,1} = F_{c,1}$

$S_{c,i} = M(S_{c,i-1}, P_{c,i})$ for $1 < i \leq I$

where I is the number of intervals in a cycle, and M is the merge operator. The merge operator simply copies the parameter values of $P_{c,i}$ and updates them in $S_{c,i-1}$.

5 Design

In this section, we discuss the design options and their impact on accuracy metrics. We begin by defining the specific accuracy metrics that guide our design and evaluation. By prioritizing accuracy throughout the design process, we aimed to create an effective system that provides high serving accuracy while addressing the network and storage bandwidth bottleneck. Note that QuickUpdate is configurable and monitored in production for the unlikely event of accuracy degradation.

5.1 Accuracy Metrics

Binary Cross Entropy or Entropy [17] is a well known aggregate metric to evaluate accuracy of ads models. In this study, we use Normalized Entropy (NE) which is defined as the Binary Cross Entropy divided by a constant. The variations of NE are computed as follows to understand how the partial update would perform with respect to the fully fresh snapshot and stale model. For simplicity, we drop cycle subscript c from the notation.

1- NE loss: It determines the accuracy reduction when model S_i is used instead of the corresponding fully fresh model F_i to run inferences.

$$NEloss(S_i) = \frac{NE_{S_i} - NE_{F_i}}{NE_{F_i}} * 100 \quad (1)$$

where NE_{S_i} and NE_{F_i} respectively denote Normalized Entropy for models S_i and F_i.

2- NE gain: It represents how much accuracy improvement to expect if S_i is used for inference instead of the stale model:

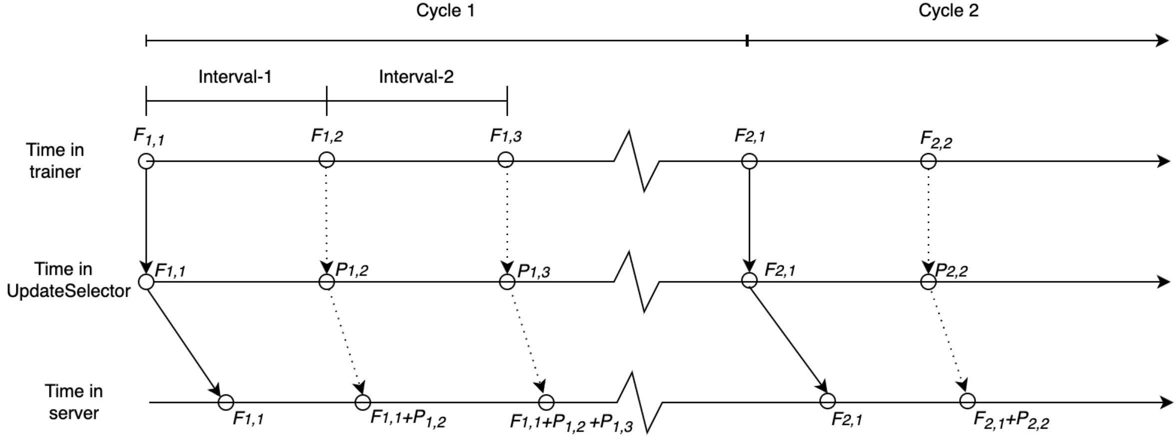


Figure 6: QuickUpdate workflow

$$NEgain(S_i) = \frac{NE_{S_i} - NE_{stale}}{NE_{stale}} * 100 \quad (2)$$

The stale model is considered to be the latest published full model F_1.

3- NE recovery: It demonstrates the percentage of maximum NE gain that has been attained by model S_i. We assume that the maximum NE gain is achieved if the fully trained model, F_i, could have been used for inference. Thus, the NE recovery is defined as:

$$NErecovery(S_i) = \frac{NEgain(S_i)}{NEgain(F_i)} * 100 \quad (3)$$

5.2 Selection Criteria

To prioritize updating rows that yield larger accuracy gains, we need a reliable indicator that remains stable throughout the training process. While the gradient vector could serve as a criterion, its oscillation between positive and negative values introduces numerical instability. Instead, we can utilize the optimizer state vector, also known as momentum, which provides a more stable measure. The optimizer state vector represents the averaged squared sum of historical gradients for a specific row. By denoting $OS^r_{c,i}$ as the optimizer state vector for a row r in the model at interval i, and $\overline{OS}^r_{c,i}$ as the average of its elements, we can leverage this measure as an indication of row importance.

Intuitively, the rows with larger values of $\overline{OS}^r_{c,i}$ would be more likely to improve accuracy. For example, these rows can represent a specific user who frequently uses the platform to click on ads, or it can represent a specific video with a high access rate. Other than the magnitude of $\overline{OS}^r_{c,i}$ for a given interval, it might be important to track how it changes over time. This can be potentially informative for situations when we prefer to prioritize parameters that have changed with respect to older versions. Based on these intuitions, we evaluate the following selection criteria:

1- Absolute optimizer state:

$$abs(\overline{OS}^r_{c,i}) \quad for \;\; i > 1 \quad (4)$$

2- Delta optimizer state:

$$abs(\overline{OS}^r_{c,i} - \overline{OS}^r_{c,i-1}) \quad for \;\; i > 1 \quad (5)$$

Table 1: NE recovery for different selection criteria

Selection criteria	NE recovery
Absolute optimizer state	70%
Delta optimizer state	100%

The choice between using absolute optimizer state or delta optimizer state as selection criteria depends on their respective advantages and trade-offs. While absolute optimizer state provides a stable and aggregate measure of how a row impacts accuracy, delta optimizer state captures the change in impact compared to the previous interval. However, using delta optimizer state requires additional memory to store the optimizer state from the previous interval. To evaluate the impact of these criteria, we conducted experiments with a 30-minute interval length and a 10% update size. After publishing a full snapshot and training for another hour, we published a partial snapshot with a 10% update size based on the two criteria. We then evaluated the serving accuracy compared to a fully fresh model. The results in Table 1 (which were consistent across multiple such experiments) indicate that delta optimizer state achieves 100% NE recovery, while absolute optimizer state achieves 70% NE recovery. This implies that selecting rows based on delta optimizer state reduces the discrepancy between the serving model and the corresponding full snapshot.

Table 2: NE recovery and NE gain for different baselines

Baseline	NE recovery
Previous full update	95.94%
Previous update	99.05%

5.3 Baseline for Delta Selection

Delta optimizer state is computed with respect to a baseline. We considered two options for choosing a baseline when computing the delta optimizer state:

1- Model state at the previous update: This option considers the model state at the time of the previous update as the baseline. It is the same as the delta optimizer definition in the previous section.

2- Model state at the previous full update: As elaborated in section 5.6, QuickUpdate also leverages intermittent full updates. In this baseline option, the model state at the time of the last intermittent full update is used as delta. The delta optimizer is defined in this case as the following:

$$abs(\overline{OS}^{r}_{c,i} - \overline{OS}^{r}_{c,1}) \quad for \;\; i > 1 \tag{6}$$

For the first option, the baseline needs to be saved at the end of every training interval, while for the second option, only one baseline is saved in the first interval of the cycle. Thus, the first option provides a more recent and fresh baseline at the cost of extra compute resources for saving it in the memory.

We examine the impact of the different baselines on the serving accuracy by conducting an experiment where each intermittent full update is followed by four partial updates. We evaluate the serving accuracy (NE recovery) compared to a fully fresh model. The results in Table 2 show that previous interval baseline would deliver 3.11% (95.94% versus 99.05%) more NE recovery. This suggests that using the model state of the previous update as a baseline can better reflect recent user interests, as it is refreshed in every update interval. In addition, the use of full updates as baseline may prioritize parameters that were important in previous intervals but are no longer contributing to accuracy. Over time, the optimizer states of these parameters may reach a plateau, yet the full update baseline may still consider them due to their large delta optimizer state values. Refreshing the baseline more frequently helps eliminate the prioritization of such parameters and instead prioritize recently changed parameters that are more likely to contribute to accuracy improvements.

5.4 Real-time Inference Pruning

Inference pruning, described in subsection 2.4, helps reduce the size of the serving model and the number of required GPUs. It practically prunes rows (or IDs) that are not active anymore or have a negligible impact on accuracy to reduce the size of embedding tables. Then, the pruned tables are stored compactly in a GPU access-friendly manner to further reduce their size in the serving cluster.

The pruning is only implemented when a full model is published to the serving cluster. For the subsequent intervals, we like partial updates to be compatible with pruned tables in full model updates. Ideally, the row IDs in partial updates should be present in the pruned table. This helps us to simply update the values of existing rows without re-structuring the tables in GPUs. However, this is not always the case. Since the training data for partial updates is different from the full model update, it is possible that some row IDs become important in partial updates while those IDs are not present in the serving-side pruned table. When such cases occur, a naive implementation would involve the insertion of the missing rows into the serving-side pruned table, which can be resource-intensive and may require reshuffling of all the embedding tables across all GPUs to ensure accessibility and efficiency (e.g., avoid memory fragmentation).

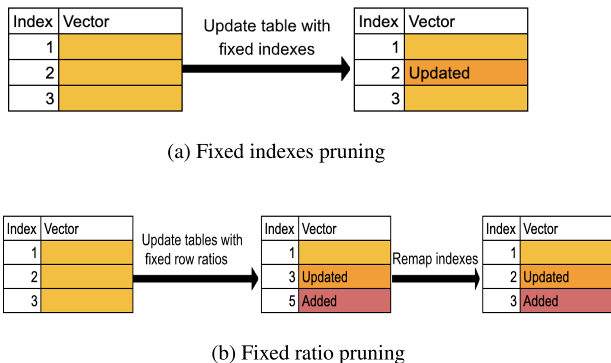
To avoid intensive reshuffling of embedding tables, we explored two alternative inference pruning strategies that are compatible with the partial updates:

1. Fixed indices pruning (see figure 7a): In this strategy, QuickUpdate performs prioritized parameter selection to choose candidate row indices for updating. However, only the rows that are already present in the embedding table are updated, while the pruned rows remain unchanged.
2. Fixed pruning ratio (see figure 7b): In this strategy, a fixed ratio of rows is pruned from the embedding table at each full update. When QuickUpdate performs prioritized parameter selection, it selects a maximum of X indices to update, where X is the total number of rows in the given table on the serving platform. This ensures that the number of rows in a table is consistent.

The first strategy avoids reshuffling, as there would be only row update operations and no row insert operations into the embedding tables. The second strategy avoids reshuffling by using both row update and remapping index operations. As the sizes of the embedding tables don't change in the second strategy, it would also avoid the need to reshard the embedding tables across the GPUs.

To evaluate the two pruning strategies, we consider three training scenarios: 1-No pruning, 2-fixed pruned indexes and 3-fixed pruned ratio per table.

Our experiments showed that NE loss due to pruning is practically negligible (<0.001%), with no accuracy difference between the pruning strategies. Considering implementation requirements, we opted for the fixed pruned indexes strategy due to its simpler implementation. Unlike the fixed pruned ratio strategy, it does not necessitate updating the index map with each new update.



(a) Fixed indexes pruning

(b) Fixed ratio pruning

Figure 7: Inference pruning strategies

5.5 Granularity

The granularity in QuickUpdate refers to the percentage of rows to include in a partial update. This parameter determines the amount of bandwidth required to transfer the update to the serving nodes. While a smaller granularity allows for faster and more efficient transfer, it may also limit the inclusion of important rows, potentially leading to a reduction in accuracy.

The granularity setting is configurable and can be adjusted based on various factors such as available infrastructure resources, desired accuracy, and the overall size of the model. In section 6 we evaluate the trade-off between update granularity and accuracy on our production workload.

5.6 Intermittent Full Model Update

After each partial update, the serving model deviates from the most recent trained model. It is because only a small percentage of parameters in the serving model are based on the most recent trained model, and the rest of the parameters are based on the older versions. As we perform more partial updates, the deviation becomes larger, leading to accuracy degradation. Thus, intermittent full model update is required to maintain the accuracy of the serving model and keep its long term accuracy at a desired level.

The main design parameter for the intermittent full model update is its frequency. Determining the desired frequency of full model updates depends on different factors such as granularity and the required accuracy. As we show in section 6, there exists a trade-off between the granularity of partial updates and the required frequency of full model updates. A system that applies larger granularity in partial updates can delay the need for a full model update, at the cost of higher average write bandwidth.

5.7 Relaxed Model Update Consistency

Traditionally, in order to ensure consistency during a full model update, the updated model is first loaded into buffer nodes. Only when loading is completed, the user request traffic is re-routed to the buffer nodes, which then serve as the new inference nodes. In QuickUpdate, we relaxed this design because it uses considerable infrastructure resources. Instead, whenever updates are available, they are directly patched to the serving model, which continues to serve real-time traffic concurrently without the need for a separate buffer node. This relaxation of the design allows for more efficient utilization of infrastructure resources.

The relaxed consistency in QuickUpdate is specifically related to the loading duration of parameters in the serving node. During this loading process, incoming inference requests may encounter inconsistent views of the embedding tables, leading to three possible cases for a particular query:

1. None of the parameters have been updated yet, and the inference is performed using the stale parameters.
2. Some parameters have been updated, and inference is performed using a mixture of stale and fresh parameters.
3. All parameters have been updated and inference is based on the fully fresh model.

Our experiments have shown that this relaxed consistency approach does not result in a negative impact on the serving accuracy when compared to a fully consistent policy, while also eliminating the need for additional buffer nodes. In section 6, we further evaluate the effect of these inconsistent embedding tables on the serving accuracy and demonstrate that they actually lead to positive accuracy gains, providing an additional benefit of the relaxed consistency approach.

6 Evaluation

We evaluate QuickUpdate on one of the largest recommendation models deployed in Meta, using real-world data, and trained on our production training cluster in a setup similar to [15]. The model is an extension of the DLRM model proposed in [16], but it is substantially larger, in the order of Terabytes. We used the same pre-recorded data stream for all the experiments, making the experiments reproducible and comparable, and eliminating potential result skew due to temporal data variations. The model was initially trained using several weeks of real-world data as the warm-up period, in order to reach a steady state. For accuracy evaluations, we evaluate the serving predictions on data stream that comes after the training data in time (i.e., the data evaluated during inference was not used in previous training).

6.1 Accuracy

In this section, we compare the accuracy implications of different update granularities, and derive the minimal full snapshot frequency such that the NE loss does not fall below 0.01%. In these experiments, we publish a full snapshot in the beginning and continue to publish partial updates with different granularities. These partial updates are applied on top of the full

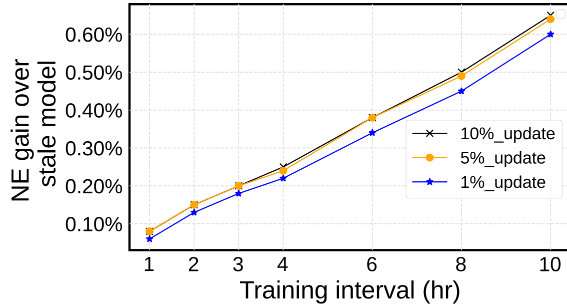


Figure 8: NE gain for partial updates with varying training periods

served snapshot and evaluated for accuracy using the same recorded dataset.

6.1.1 NE gain compared to a stale model

We begin by comparing the accuracy of QuickUpdate with the accuracy of a stale model, to quantify the accuracy gains and to validate that applying partial updates on top of a full snapshot does not negatively affect accuracy. Here, the stale model refers to the initially published full snapshot. Figure 8 shows the NE gain with respect to the stale model, for different update granularities (and without intermittent full model updates). All the update sizes result in higher NE gain than the stale model, and the NE gain increases over time. The 5% and 10% updates provide very similar NE gains, but the 1% update returns less NE gain, indicating some important rows are not included in the 1% updates. Overall, these trends indicate that even after applying partial updates for over 10 hours, there is no adverse impact and accuracy improves by 0.7% compared with the stale model.

6.1.2 NE loss compared to a fully fresh model

In this section, we investigate the NE loss of a model published by QuickUpdate using partial updates, compared to an ideal fully fresh serving mode. The results presented in figure 9 show that with 10% update, the NE loss is below 0.005% during the entire 10 hours. With 5% update, the NE loss is consistently higher than 10% update, but is still less than 0.01% during over six hours. The NE loss increases as training period increases, since the discrepancy between the serving model and the corresponding trained model increases.

The results also demonstrate the impact of employing different update granularities on the delay of full model publication, while ensuring the NE loss remains below the acceptable threshold of 0.01%. By adopting a 10% granularity, we can effectively delay the need for full model publication by over 10 hours. Similarly, when utilizing a 5% granularity, we can postpone the full model publication by 6 hours while still keeping the NE loss within the acceptable range. This highlights the efficacy of partial updates at the 5% granularity in capturing important updates and maintaining the model's accuracy and freshness over a considerable period.

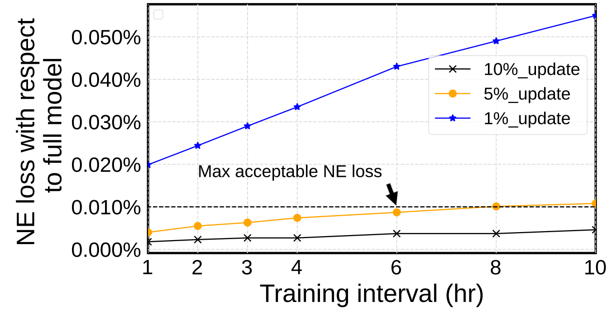


Figure 9: NE loss for partial updates with varying training periods

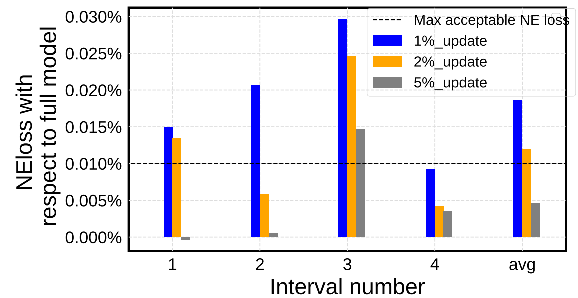


Figure 10: NE loss of partial updates with different granularities, in 4 consecutive 10 minute intervals

6.1.3 NE loss over short time periods

In order to analyze the NE loss over short time periods, we conducted an evaluation involving four consecutive 10-minute updates. The update granularities examined were 5%, 3% and 1%. The NE loss was measured after each update in comparison to a fully fresh model using unseen data. Figure 10 shows the variance across different 10 minute intervals, emphasizing the fluctuating nature of the streaming data. However, the NE loss stabilizes when averaged over multiple short time intervals. As expected, the results show that the NE loss reduces as we increase the granularity. The average NE loss presented at the last column confirms that the 5% granularity would return an acceptable NE loss (on average) in our workload.

6.1.4 Conclusion

The accuracy results demonstrate the effectiveness of QuickUpdate in employing a 5% update granularity for up to 6 hours, while maintaining accuracy levels comparable to a fully fresh model and ensuring NE loss below the threshold of 0.01%.

Additionally, the use of QuickUpdate with 5% update granularity allows for a delay of 6 hours before the necessity of publishing a full model arises. This delay is possible due to the successful incorporation of partial updates, which capture and integrate important changes, resulting in accurate and up-to-date models.

Based on these findings, QuickUpdate triggers intermittent full model publishing every 6 hours by default, optimizing the balance between accuracy and update frequency.

6.2 Analyzing Long Term Row Convergence

In the previous analysis, our focus was on minimizing partial update granularity and determining the appropriate frequency for intermittent full updates, based on the accuracy metrics. The results indicated that utilizing partial updates with granularity of 5% for 6 hours achieved satisfactory accuracy.

In this experiment, our objective is to explore what percentage of the important rows in the model is updated by partial updates.

In order to determine a proxy of the important rows, we train the model for a duration of 6 hours (i.e., the same duration with satisfactory accuracy). We determine important rows to be the top percentage of rows in the trained model such that publishing them once at the end of the 6 hours duration, instead of the entire model, would have returned the satisfactory accuracy (i.e., below 0.01% difference compared to a fully fresh model).

Figure 11 shows the NE loss (compared with a fully fresh model) with different sizes of a single update after 6 hours of training. As can be seen, a single 5% partial update is not sufficient to achieve an NE loss below the acceptable threshold of 0.01%. However, a 10% partial update proves to be adequate in reducing the NE loss to an acceptable level. This indicates that the top 10% of the ranked embedding rows are a good proxy of the important rows in this time window.

To understand the percentage of those important rows that is covered by multiple smaller 5% updates, we ran QuickUpdate for 6 hours with multiple partial updates of 5% granularity. Upon combining all these updates into a union set, we observe that this set encompasses 70% of the important rows mentioned above, and overall covers 7.3% of all the rows in the model. Thus, a large portion of the important rows are covered by consecutive, smaller partial updates.

6.3 Bandwidth Usage

In QuickUpdate, the update size is a proxy for the bandwidth usage. The amount of bandwidth usage depends on the granularity, update interval, and frequency of the intermittent full model update. In general, these parameters are configurable and may change according to the type of DLRM and the desired accuracy. In this section, we evaluate bandwidth usage for different policies based on the percentage of the model

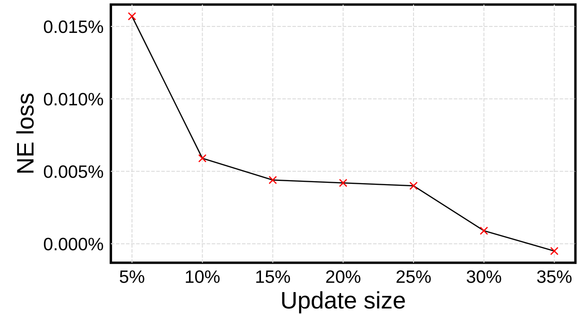


Figure 11: NE loss versus update size for a 6-hour training period

that is published. The details of them are elaborated below and in Figure 12:

1. Baseline 1: Full model is published every hour
2. Baseline 2: Full model is published every 10 minutes (not shown in the figure)
3. 5% update (default policy): A partial update is published every 10 minutes with a granularity of 5%, with intermittent full update every 6 hours (as discussed in 6.1).
4. 10% update: Similar to the previous policy, a partial update is published every 10 minutes, but with a granularity of 10%. An intermittent full update is published every 6 hours.

In order to compare these policies, we average the consumed bandwidth. The results show that the default policy of 5% update granularity with 6-hour intermittent full update interval writes on average 43.6% of the model size per hour, compared with 68.2% in policy 3 (10% update), and 100% in the baseline 1 case. Baseline 2, which provides a comparable accuracy to policies 2 and 3, would require publishing 600% of the model size per hour.

Overall, with default policy, QuickUpdate is able to reduce the consumed bandwidth by 2.3x compared to baseline 1, while providing a better accuracy that is comparable to a fully fresh model. Compared to baseline 2 (which is not feasible at scale due to the network and storage bandwidth limitation), QuickUpdate is able to reduce the required bandwidth by over 13x, while still providing comparable accuracy.

6.4 Relaxed Consistency

Traditionally, serving models are updated atomically to maintain consistent inference. This involves loading all model weights into buffer nodes, which later become the serving nodes for computing inferences. However, this approach is resource intensive due to the use of buffer nodes. To address this, QuickUpdate relaxes the consistency requirement and parameters are directly updated in the serving nodes while simultaneously performing inference queries.

We evaluate the NE recovery (compared to a fully fresh model) during an intermittent full model update in QuickUp-

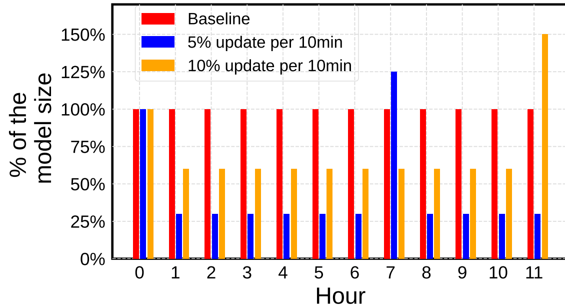


Figure 12: Bandwidth measure: update size per hour for different update scenarios

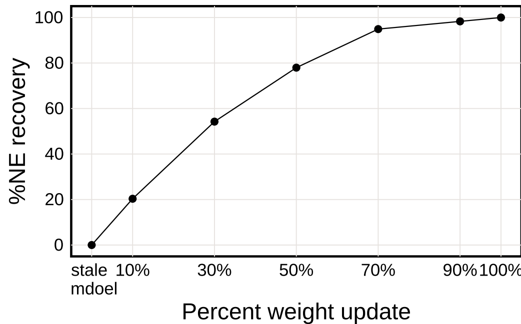


Figure 13: NE recovery during model update (relaxed consistency)

date, as a function of the percentage of the weights that have been updated. As shown in figure 13, relaxing the consistency can improve the accuracy in production during loading. The NE recovery increases as more parameters are loaded. Our data shows that we can capture about 54% of the NE recovery by patching 30% of the parameters. The NE recovery reaches around 94% after patching just 70% of the parameters.

Relaxed consistency allows for the early serving of fresh rows (rather than waiting for the entire model to update), leading to overall higher accuracy. Although there is an inconsistent view of tables during loading (implying that different rows may belong to different states), serving a subset of fresh rows already leads to in increase in accuracy. The NE recovery continues to grow over time until the entire model have been updated.

7 Related Work

Asynchronous or partial update strategy has been implemented for few real-time DLRMs [13, 18, 21]. In Kraken [21], dense parameters are updated every few seconds in a batch, while the sparse parameters are updated whenever their values change in the trainer. This is a lossless parameter update that can produce a significant amount of traffic for large models with 100-1000 billion parameters [15] and geo-distributed servers. Monolith [13] mainly focused on developing a system with collision-less embedding tables for sparse features. A sparse parameter can be updated in minute-level granularity when it is trained and its value changes from the last synchronization. Similar to Kraken, this is a lossless update which can create huge traffic. Overall, the lossless model update can be very resource intensive, as discussed in section 3. To overcome this issue, QuickUpdate can perform prioritized parameter selection that results in about $78\% - 92\%$ bandwidth reduction and a negligible accuracy loss ($< 0.01\%$). In another study, Ekko [18] is designed as an efficient system to broadcast the updates from the trainer model to all the serving inference nodes. To quickly update the larger embedding tables in the serving models, they used the sparsity and temporal locality in the embedding table updates. The Ekko system is orthogonal to QuickUpdate, and both can be implemented together. In the QuickUpdate, we optimized the design elements such as publishing interval, update granularity and parameter selection criteria to achieve the desired accuracy and minimize publishing the full model. Prioritized parameter selection is one of the techniques we used in this paper. In past studies (e.g., [1, 2, 12] Gradient based parameter selection has been explored for distributed training systems. Ekko [18] further expanded this criterion and additionally considered request frequency for each parameter and parameter freshness to the selection criteria. In QuickUpdate we decided to choose delta of gradient momentum which is a more stable measure than gradient itself, and additionally it publishes parameters such that it returns the highest accuracy compared with the baseline snapshot.

8 Conclusion

QuickUpdate is a system that enables online training to perform low-latency partial updates while providing a serving accuracy that is comparable to a fully fresh model. It offers a scalable solution for serving production-scale DLRM in real-time. This is particularly valuable because serving such models in real-time is challenging at scale due to limitations in network and storage bandwidth.

QuickUpdate achieves its scalability and accuracy goals by utilizing innovative techniques. One of these techniques involves selectively publishing the most important parts of each update, reducing the overall update size while maintaining accuracy. Additionally, QuickUpdate incorporates intermittent full model updates at a low frequency to ensure long-term accuracy. This combination of selective partial updates and intermittent full updates enables QuickUpdate to balance between low-latency serving and preserving accuracy over time.

We evaluated QuickUpdate using real world data for a large personalized ads model and showed that QuickUpdate is able to provide a serving accuracy that is comparable to a fully fresh model, while minimizing the required write bandwidth by over 13x.

References

[1] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

[2] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[4] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[5] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[6] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[7] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

[8] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[9] Bowei He, Xu He, Yingxue Zhang, Ruiming Tang, and Chen Ma. Dynamically expandable graph convolution for streaming recommendation. *arXiv preprint arXiv:2303.11700*, 2023.

[10] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3288–3298, 2022.

[11] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[12] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[13] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. Monolith: Real time recommendation system with collisionless embedding table. In *Proceedings of 5th Workshop on Online Recommender Systems and User Modeling, in conjunction with the 16th ACM Conference on Recommender Systems*, Seattle, WA, 2022.

[14] Ian MacKenzie, Chris Meyer, and Steve Noble. How retailers can keep up with consumers. *McKinsey & Company*, 18(1), 2013.

[15] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[16] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[17] Pytorch. Crossentropy loss. In *https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html*.

[18] Chijun Sima, Yao Fu, Man-Kit Sit Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In *16th USENIX Symposium on Operating Systems Design and Implementation.*, pages 821–839, Carlsbad, CA, 2022.

[19] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017.

[20] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 839–848, 2018.

[21] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. 2020.

[22] Yishi Xu, Yingxue Zhang, Wei Guo, Huifeng Guo, Ruiming Tang, and Mark Coates. Graphsail: Graph structure aware incremental learning for recommender systems. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2861–2868, 2020.

[23] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[24] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[25] Peiyan Zhang and Sunghun Kim. A survey on incremental update for neural recommender systems. *arXiv preprint arXiv:2303.02851*, 2023.

[26] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019.

[27] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.